The Supplemental Isabelle/HOL Library

April 17, 2016

Contents

1 Implementation of Association Lists

theory *AList*
imports *Main*
begin

context
begin

The operations preserve distinctness of keys and function *clearjunk* distributes over them. Since *clearjunk* enforces distinctness of keys it can be used to establish the invariant, e.g. for inductive proofs.

1.1 *update* and *updates*

qualified primrec *update* :: *'key* ⇒ *'val* ⇒ (*'key* × *'val*) *list* ⇒ (*'key* × *'val*) *list*
where
  update k v [] = [(*k*, *v*)]
| *update k v* (*p* # *ps*) = (*if fst p* = *k then* (*k*, *v*) # *ps else p* # *update k v ps*)

lemma *update-conv'*: *map-of* (*update k v al*) = (*map-of al*)(*k*↦*v*)
  by (*induct al*) (*auto simp add*: *fun-eq-iff*)

corollary *update-conv*: *map-of* (*update k v al*) *k'* = ((*map-of al*)(*k*↦*v*)) *k'*
  by (*simp add*: *update-conv'*)

lemma *dom-update*: *fst* ' *set* (*update k v al*) = {*k*} ∪ *fst* ' *set al*
  by (*induct al*) *auto*

lemma *update-keys*:
  map fst (*update k v al*) =
    (*if k* ∈ *set* (*map fst al*) *then map fst al else map fst al* @ [*k*])
  by (*induct al*) *simp-all*

lemma *distinct-update*:
  assumes *distinct* (*map fst al*)
  shows *distinct* (*map fst* (*update k v al*))
  using *assms* **by** (*simp add*: *update-keys*)

lemma *update-filter*:
  a ≠ *k* ⟹ *update k v* [*q*←*ps*. *fst q* ≠ *a*] = [*q*←*update k v ps*. *fst q* ≠ *a*]
  by (*induct ps*) *auto*

lemma *update-triv*: *map-of al k* = *Some v* ⟹ *update k v al* = *al*
  by (*induct al*) *auto*

lemma *update-nonempty* [*simp*]: *update k v al* ≠ []
  by (*induct al*) *auto*

lemma *update-eqD*: *update k v al* = *update k v′ al′* ⟹ *v* = *v′*
proof (*induct al arbitrary*: *al′*)
  case *Nil*
  then show *?case*
    by (*cases al′*) (*auto split*: *if-split-asm*)
next
  case *Cons*
  then show *?case*
    by (*cases al′*) (*auto split*: *if-split-asm*)
qed

lemma *update-last* [*simp*]: *update k v* (*update k v′ al*) = *update k v al*
  by (*induct al*) *auto*

Note that the lists are not necessarily the same: *update k v* (*update k′ v′* []) = [(*k′*, *v′*), (*k*, *v*)] and *update k′ v′* (*update k v* []) = [(*k*, *v*), (*k′*, *v′*)].

lemma *update-swap*:
  k ≠ *k′* ⟹
    map-of (*update k v* (*update k′ v′ al*)) = *map-of* (*update k′ v′* (*update k v al*))
  by (*simp add*: *update-conv′ fun-eq-iff*)

lemma *update-Some-unfold*:
  map-of (*update k v al*) *x* = *Some y* ⟷
    x = *k* ∧ *v* = *y* ∨ *x* ≠ *k* ∧ *map-of al x* = *Some y*
  by (*simp add*: *update-conv′ map-upd-Some-unfold*)

lemma *image-update* [*simp*]:
  x ∉ *A* ⟹ *map-of* (*update x y al*) ‘ *A* = *map-of al* ‘ *A*
  by (*simp add*: *update-conv′*)

qualified definition
    updates :: *′key list* ⇒ *′val list* ⇒ (*′key* × *′val*) *list* ⇒ (*′key* × *′val*) *list*
  where *updates ks vs* = *fold* (*case-prod update*) (*zip ks vs*)

lemma *updates-simps* [*simp*]:
  updates [] *vs ps* = *ps*
  updates ks [] *ps* = *ps*
  updates (*k*#*ks*) (*v*#*vs*) *ps* = *updates ks vs* (*update k v ps*)
  by (*simp-all add*: *updates-def*)

lemma *updates-key-simp* [*simp*]:
  updates (*k* # *ks*) *vs ps* =
    (*case vs of* [] ⇒ *ps* | *v* # *vs* ⇒ *updates ks vs* (*update k v ps*))
  by (*cases vs*) *simp-all*

lemma *updates-conv′*: *map-of* (*updates ks vs al*) = (*map-of al*)(*ks*[↦]*vs*)
proof −
  have *map-of* ∘ *fold* (*case-prod update*) (*zip ks vs*) =
      fold (λ(*k*, *v*) *f*. *f*(*k* ↦ *v*)) (*zip ks vs*) ∘ *map-of*

by (*rule fold-commute*) (*auto simp add*: *fun-eq-iff update-conv'*)
then show *?thesis*
by (*auto simp add*: *updates-def fun-eq-iff map-upds-fold-map-upd foldl-conv-fold split-def*)
qed

lemma *updates-conv*: *map-of* (*updates ks vs al*) *k* = ((*map-of al*)(*ks*[↦]*vs*)) *k*
by (*simp add*: *updates-conv'*)

lemma *distinct-updates*:
assumes *distinct* (*map fst al*)
shows *distinct* (*map fst* (*updates ks vs al*))
proof −
have *distinct* (*fold*
(λ(*k*, *v*) *al*. *if k* ∈ *set al then al else al* @ [*k*])
(*zip ks vs*) (*map fst al*))
by (*rule fold-invariant* [*of zip ks vs* λ-. *True*]) (*auto intro*: *assms*)
moreover have *map fst* ∘ *fold* (*case-prod update*) (*zip ks vs*) =
fold (λ(*k*, *v*) *al*. *if k* ∈ *set al then al else al* @ [*k*]) (*zip ks vs*) ∘ *map fst*
by (*rule fold-commute*) (*simp add*: *update-keys split-def case-prod-beta comp-def*)
ultimately show *?thesis*
by (*simp add*: *updates-def fun-eq-iff*)
qed

lemma *updates-append1*[*simp*]: *size ks* < *size vs* ⟹
updates (*ks*@[*k*]) *vs al* = *update k* (*vs*!*size ks*) (*updates ks vs al*)
by (*induct ks arbitrary*: *vs al*) (*auto split*: *list.splits*)

lemma *updates-list-update-drop*[*simp*]:
size ks ≤ *i* ⟹ *i* < *size vs* ⟹
updates ks (*vs*[*i*:=*v*]) *al* = *updates ks vs al*
by (*induct ks arbitrary*: *al vs i*) (*auto split*: *list.splits nat.splits*)

lemma *update-updates-conv-if*:
map-of (*updates xs ys* (*update x y al*)) =
map-of
(*if x* ∈ *set* (*take* (*length ys*) *xs*)
then updates xs ys al
else (*update x y* (*updates xs ys al*)))
by (*simp add*: *updates-conv' update-conv' map-upd-upds-conv-if*)

lemma *updates-twist* [*simp*]:
k ∉ *set ks* ⟹
map-of (*updates ks vs* (*update k v al*)) = *map-of* (*update k v* (*updates ks vs al*))
by (*simp add*: *updates-conv' update-conv'*)

lemma *updates-apply-notin* [*simp*]:
k ∉ *set ks* ⟹ *map-of* (*updates ks vs al*) *k* = *map-of al k*
by (*simp add*: *updates-conv*)

lemma *updates-append-drop* [*simp*]:
size xs = size ys ⟹ *updates* (*xs* @ *zs*) *ys al* = *updates xs ys al*
by (*induct xs arbitrary*: *ys al*) (*auto split*: *list.splits*)

lemma *updates-append2-drop* [*simp*]:
size xs = size ys ⟹ *updates xs* (*ys* @ *zs*) *al* = *updates xs ys al*
by (*induct xs arbitrary*: *ys al*) (*auto split*: *list.splits*)

1.2 *delete*

qualified definition *delete* :: *'key* ⇒ (*'key* × *'val*) *list* ⇒ (*'key* × *'val*) *list*
where *delete-eq*: *delete k* = *filter* (λ(*k'*, -). *k* ≠ *k'*)

lemma *delete-simps* [*simp*]:
delete k [] = []
delete k (*p* # *ps*) = (*if fst p* = *k then delete k ps else p* # *delete k ps*)
by (*auto simp add*: *delete-eq*)

lemma *delete-conv'*: *map-of* (*delete k al*) = (*map-of al*)(*k* := *None*)
by (*induct al*) (*auto simp add*: *fun-eq-iff*)

corollary *delete-conv*: *map-of* (*delete k al*) *k'* = ((*map-of al*)(*k* := *None*)) *k'*
by (*simp add*: *delete-conv'*)

lemma *delete-keys*: *map fst* (*delete k al*) = *removeAll k* (*map fst al*)
by (*simp add*: *delete-eq removeAll-filter-not-eq filter-map split-def comp-def*)

lemma *distinct-delete*:
assumes *distinct* (*map fst al*)
shows *distinct* (*map fst* (*delete k al*))
using *assms* **by** (*simp add*: *delete-keys distinct-removeAll*)

lemma *delete-id* [*simp*]: *k* ∉ *fst* ' *set al* ⟹ *delete k al* = *al*
by (*auto simp add*: *image-iff delete-eq filter-id-conv*)

lemma *delete-idem*: *delete k* (*delete k al*) = *delete k al*
by (*simp add*: *delete-eq*)

lemma *map-of-delete* [*simp*]: *k'* ≠ *k* ⟹ *map-of* (*delete k al*) *k'* = *map-of al k'*
by (*simp add*: *delete-conv'*)

lemma *delete-notin-dom*: *k* ∉ *fst* ' *set* (*delete k al*)
by (*auto simp add*: *delete-eq*)

lemma *dom-delete-subset*: *fst* ' *set* (*delete k al*) ⊆ *fst* ' *set al*
by (*auto simp add*: *delete-eq*)

lemma *delete-update-same*: *delete k* (*update k v al*) = *delete k al*

by (*induct al*) *simp-all*

lemma *delete-update*: $k \neq l \Longrightarrow$ *delete l* (*update k v al*) = *update k v* (*delete l al*)
by (*induct al*) *simp-all*

lemma *delete-twist*: *delete x* (*delete y al*) = *delete y* (*delete x al*)
by (*simp add*: *delete-eq conj-commute*)

lemma *length-delete-le*: *length* (*delete k al*) $\leq$ *length al*
by (*simp add*: *delete-eq*)

1.3 *update-with-aux* and *delete-aux*

qualified primrec *update-with-aux* :: $'val \Rightarrow 'key \Rightarrow ('val \Rightarrow 'val) \Rightarrow ('key \times 'val)$ *list* $\Rightarrow ('key \times 'val)$ *list*
where
update-with-aux v k f [] = [(*k*, *f v*)]
| *update-with-aux v k f* (*p* # *ps*) = (*if* (*fst p* = *k*) *then* (*k*, *f* (*snd p*)) # *ps else p* # *update-with-aux v k f ps*)

The above *delete* traverses all the list even if it has found the key. This one does not have to keep going because is assumes the invariant that keys are distinct.

qualified fun *delete-aux* :: $'key \Rightarrow ('key \times 'val)$ *list* $\Rightarrow ('key \times 'val)$ *list*
where
delete-aux k [] = []
| *delete-aux k* ((*k'*, *v*) # *xs*) = (*if k* = *k' then xs else* (*k'*, *v*) # *delete-aux k xs*)

lemma *map-of-update-with-aux'*:
map-of (*update-with-aux v k f ps*) *k'* = ((*map-of ps*)(*k* $\mapsto$ (*case map-of ps k of None* $\Rightarrow$ *f v* | *Some v* $\Rightarrow$ *f v*))) *k'*
by(*induct ps*) *auto*

lemma *map-of-update-with-aux*:
map-of (*update-with-aux v k f ps*) = (*map-of ps*)(*k* $\mapsto$ (*case map-of ps k of None* $\Rightarrow$ *f v* | *Some v* $\Rightarrow$ *f v*))
by(*simp add*: *fun-eq-iff map-of-update-with-aux'*)

lemma *dom-update-with-aux*: *fst* ‘ *set* (*update-with-aux v k f ps*) = {*k*} $\cup$ *fst* ‘ *set ps*
by (*induct ps*) *auto*

lemma *distinct-update-with-aux* [*simp*]:
distinct (*map fst* (*update-with-aux v k f ps*)) = *distinct* (*map fst ps*)
by(*induct ps*)(*auto simp add*: *dom-update-with-aux*)

lemma *set-update-with-aux*:
distinct (*map fst xs*)
$\Longrightarrow$ *set* (*update-with-aux v k f xs*) = (*set xs* − {*k*} × *UNIV* $\cup$ {(*k*, *f* (*case map-of*

xs k of None ⇒ v | Some v ⇒ v))})
by(*induct xs*)(*auto intro*: *rev-image-eqI*)

lemma *set-delete-aux*: *distinct* (*map fst xs*) ⟹ *set* (*delete-aux k xs*) = *set xs* − {*k*} × *UNIV*
apply(*induct xs*)
apply *simp-all*
apply *clarsimp*
apply(*fastforce intro*: *rev-image-eqI*)
done

lemma *dom-delete-aux*: *distinct* (*map fst ps*) ⟹ *fst* ` *set* (*delete-aux k ps*) = *fst* ` *set ps* − {*k*}
by(*auto simp add*: *set-delete-aux*)

lemma *distinct-delete-aux* [*simp*]:
  distinct (*map fst ps*) ⟹ *distinct* (*map fst* (*delete-aux k ps*))
proof(*induct ps*)
  case *Nil* **thus** *?case* **by** *simp*
next
  case (*Cons a ps*)
  obtain *k′ v* **where** *a*: *a* = (*k′*, *v*) **by**(*cases a*)
  show *?case*
  proof(*cases k′* = *k*)
    case *True* **with** *Cons a* **show** *?thesis* **by** *simp*
  next
    case *False*
    with *Cons a* **have** *k′* ∉ *fst* ` *set ps distinct* (*map fst ps*) **by** *simp-all*
    with *False a* **have** *k′* ∉ *fst* ` *set* (*delete-aux k ps*)
      by(*auto dest*!: *dom-delete-aux*[**where** *k*=*k*])
    with *Cons a* **show** *?thesis* **by** *simp*
  qed
qed

lemma *map-of-delete-aux′*:
  distinct (*map fst xs*) ⟹ *map-of* (*delete-aux k xs*) = (*map-of xs*)(*k* := *None*)
  apply (*induct xs*)
  apply (*fastforce simp add*: *map-of-eq-None-iff fun-upd-twist*)
  apply (*auto intro*!: *ext*)
  apply (*simp add*: *map-of-eq-None-iff*)
  done

lemma *map-of-delete-aux*:
  distinct (*map fst xs*) ⟹ *map-of* (*delete-aux k xs*) *k′* = ((*map-of xs*)(*k* := *None*)) *k′*
by(*simp add*: *map-of-delete-aux′*)

lemma *delete-aux-eq-Nil-conv*: *delete-aux k ts* = [] ⟷ *ts* = [] ∨ (∃ *v*. *ts* = [(*k*, *v*)])

by(*cases ts*)(*auto split*: *if-split-asm*)

1.4 *restrict*

qualified definition *restrict* :: *'key set* ⇒ (*'key* × *'val*) *list* ⇒ (*'key* × *'val*) *list*
  where *restrict-eq*: *restrict A* = *filter* (λ(*k*, *v*). *k* ∈ *A*)

lemma *restr-simps* [*simp*]:
  restrict A [] = []
  restrict A (*p*#*ps*) = (*if fst p* ∈ *A then p* # *restrict A ps else restrict A ps*)
  by (*auto simp add*: *restrict-eq*)

lemma *restr-conv'*: *map-of* (*restrict A al*) = ((*map-of al*)|' *A*)
proof
  fix *k*
  show *map-of* (*restrict A al*) *k* = ((*map-of al*)|' *A*) *k*
    by (*induct al*) (*simp*, *cases k* ∈ *A*, *auto*)
qed

corollary *restr-conv*: *map-of* (*restrict A al*) *k* = ((*map-of al*)|' *A*) *k*
  by (*simp add*: *restr-conv'*)

lemma *distinct-restr*:
  distinct (*map fst al*) ⟹ *distinct* (*map fst* (*restrict A al*))
  by (*induct al*) (*auto simp add*: *restrict-eq*)

lemma *restr-empty* [*simp*]:
  restrict {} *al* = []
  restrict A [] = []
  by (*induct al*) (*auto simp add*: *restrict-eq*)

lemma *restr-in* [*simp*]: *x* ∈ *A* ⟹ *map-of* (*restrict A al*) *x* = *map-of al x*
  by (*simp add*: *restr-conv'*)

lemma *restr-out* [*simp*]: *x* ∉ *A* ⟹ *map-of* (*restrict A al*) *x* = *None*
  by (*simp add*: *restr-conv'*)

lemma *dom-restr* [*simp*]: *fst* ' *set* (*restrict A al*) = *fst* ' *set al* ∩ *A*
  by (*induct al*) (*auto simp add*: *restrict-eq*)

lemma *restr-upd-same* [*simp*]: *restrict* (−{*x*}) (*update x y al*) = *restrict* (−{*x*}) *al*
  by (*induct al*) (*auto simp add*: *restrict-eq*)

lemma *restr-restr* [*simp*]: *restrict A* (*restrict B al*) = *restrict* (*A*∩*B*) *al*
  by (*induct al*) (*auto simp add*: *restrict-eq*)

lemma *restr-update*[*simp*]:
 map-of (*restrict D* (*update x y al*)) =

map-of ((*if* $x \in D$ *then* (*update x y* (*restrict* ($D-\{x\}$) *al*)) *else restrict D al*))
by (*simp add*: *restr-conv′ update-conv′*)

lemma *restr-delete* [*simp*]:
delete x (*restrict D al*) = (*if* $x \in D$ *then restrict* ($D - \{x\}$) *al else restrict D al*)
apply (*simp add*: *delete-eq restrict-eq*)
apply (*auto simp add*: *split-def*)
proof −
have $\bigwedge y.\ y \neq x \longleftrightarrow x \neq y$
by *auto*
then show [$p \leftarrow al.$ *fst* $p \in D \wedge x \neq$ *fst* p] = [$p \leftarrow al.$ *fst* $p \in D \wedge$ *fst* $p \neq x$]
by *simp*
assume $x \notin D$
then have $\bigwedge y.\ y \in D \longleftrightarrow y \in D \wedge x \neq y$
by *auto*
then show [$p \leftarrow al$. *fst* $p \in D \wedge x \neq$ *fst* p] = [$p \leftarrow al$. *fst* $p \in D$]
by *simp*
qed

lemma *update-restr*:
map-of (*update x y* (*restrict D al*)) = *map-of* (*update x y* (*restrict* ($D - \{x\}$) *al*))
by (*simp add*: *update-conv′ restr-conv′*) (*rule fun-upd-restrict*)

lemma *update-restr-conv* [*simp*]:
$x \in D \Longrightarrow$
map-of (*update x y* (*restrict D al*)) = *map-of* (*update x y* (*restrict* ($D - \{x\}$) *al*))
by (*simp add*: *update-conv′ restr-conv′*)

lemma *restr-updates* [*simp*]:
length xs = *length ys* $\Longrightarrow$ *set xs* $\subseteq D \Longrightarrow$
map-of (*restrict D* (*updates xs ys al*)) =
map-of (*updates xs ys* (*restrict* ($D -$ *set xs*) *al*))
by (*simp add*: *updates-conv′ restr-conv′*)

lemma *restr-delete-twist*: (*restrict A* (*delete a ps*)) = *delete a* (*restrict A ps*)
by (*induct ps*) *auto*

1.5 *clearjunk*

qualified function *clearjunk* :: ($'key \times 'val$) *list* ⇒ ($'key \times 'val$) *list*
where
clearjunk [] = []
| *clearjunk* ($p\#ps$) = p # *clearjunk* (*delete* (*fst p*) *ps*)
by *pat-completeness auto*
termination
by (*relation measure length*) (*simp-all add*: *less-Suc-eq-le length-delete-le*)

lemma *map-of-clearjunk*: *map-of* (*clearjunk al*) = *map-of al*
by (*induct al rule*: *clearjunk.induct*) (*simp-all add*: *fun-eq-iff*)

lemma *clearjunk-keys-set*: *set* (*map fst* (*clearjunk al*)) = *set* (*map fst al*)
by (*induct al rule*: *clearjunk.induct*) (*simp-all add*: *delete-keys*)

lemma *dom-clearjunk*: *fst* ' *set* (*clearjunk al*) = *fst* ' *set al*
using *clearjunk-keys-set* **by** *simp*

lemma *distinct-clearjunk* [*simp*]: *distinct* (*map fst* (*clearjunk al*))
by (*induct al rule*: *clearjunk.induct*) (*simp-all del*: *set-map add*: *clearjunk-keys-set delete-keys*)

lemma *ran-clearjunk*: *ran* (*map-of* (*clearjunk al*)) = *ran* (*map-of al*)
by (*simp add*: *map-of-clearjunk*)

lemma *ran-map-of*: *ran* (*map-of al*) = *snd* ' *set* (*clearjunk al*)
proof −
have *ran* (*map-of al*) = *ran* (*map-of* (*clearjunk al*))
by (*simp add*: *ran-clearjunk*)
also have ... = *snd* ' *set* (*clearjunk al*)
by (*simp add*: *ran-distinct*)
finally show *?thesis* **.**
qed

lemma *clearjunk-update*: *clearjunk* (*update k v al*) = *update k v* (*clearjunk al*)
by (*induct al rule*: *clearjunk.induct*) (*simp-all add*: *delete-update*)

lemma *clearjunk-updates*: *clearjunk* (*updates ks vs al*) = *updates ks vs* (*clearjunk al*)
proof −
have *clearjunk* ∘ *fold* (*case-prod update*) (*zip ks vs*) =
fold (*case-prod update*) (*zip ks vs*) ∘ *clearjunk*
by (*rule fold-commute*) (*simp add*: *clearjunk-update case-prod-beta o-def*)
then show *?thesis*
by (*simp add*: *updates-def fun-eq-iff*)
qed

lemma *clearjunk-delete*: *clearjunk* (*delete x al*) = *delete x* (*clearjunk al*)
by (*induct al rule*: *clearjunk.induct*) (*auto simp add*: *delete-idem delete-twist*)

lemma *clearjunk-restrict*: *clearjunk* (*restrict A al*) = *restrict A* (*clearjunk al*)
by (*induct al rule*: *clearjunk.induct*) (*auto simp add*: *restr-delete-twist*)

lemma *distinct-clearjunk-id* [*simp*]: *distinct* (*map fst al*) ⟹ *clearjunk al* = *al*
by (*induct al rule*: *clearjunk.induct*) *auto*

lemma *clearjunk-idem*: *clearjunk* (*clearjunk al*) = *clearjunk al*
by *simp*

lemma *length-clearjunk*: *length* (*clearjunk al*) ≤ *length al*
proof (*induct al rule*: *clearjunk.induct* [*case-names Nil Cons*])
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons kv al*)
  moreover have *length* (*delete* (*fst kv*) *al*) ≤ *length al*
    by (*fact length-delete-le*)
  ultimately have *length* (*clearjunk* (*delete* (*fst kv*) *al*)) ≤ *length al*
    by (*rule order-trans*)
  then show *?case*
    by *simp*
qed

lemma *delete-map*:
  assumes ⋀*kv*. *fst* (*f kv*) = *fst kv*
  shows *delete k* (*map f ps*) = *map f* (*delete k ps*)
  by (*simp add*: *delete-eq filter-map comp-def split-def assms*)

lemma *clearjunk-map*:
  assumes ⋀*kv*. *fst* (*f kv*) = *fst kv*
  shows *clearjunk* (*map f ps*) = *map f* (*clearjunk ps*)
  by (*induct ps rule*: *clearjunk.induct* [*case-names Nil Cons*])
    (*simp-all add*: *clearjunk-delete delete-map assms*)

1.6 *map-ran*

definition *map-ran* :: (*'key* ⇒ *'val* ⇒ *'val*) ⇒ (*'key* × *'val*) *list* ⇒ (*'key* × *'val*) *list*
  where *map-ran f* = *map* (λ(*k*, *v*). (*k*, *f k v*))

lemma *map-ran-simps* [*simp*]:
  map-ran f [] = []
  map-ran f ((*k*, *v*) # *ps*) = (*k*, *f k v*) # *map-ran f ps*
  by (*simp-all add*: *map-ran-def*)

lemma *dom-map-ran*: *fst* ‘ *set* (*map-ran f al*) = *fst* ‘ *set al*
  by (*simp add*: *map-ran-def image-image split-def*)

lemma *map-ran-conv*: *map-of* (*map-ran f al*) *k* = *map-option* (*f k*) (*map-of al k*)
  by (*induct al*) *auto*

lemma *distinct-map-ran*: *distinct* (*map fst al*) ⟹ *distinct* (*map fst* (*map-ran f al*))
  by (*simp add*: *map-ran-def split-def comp-def*)

lemma *map-ran-filter*: *map-ran f* [*p*←*ps*. *fst p* ≠ *a*] = [*p*←*map-ran f ps*. *fst p* ≠ *a*]

by (*simp add*: *map-ran-def filter-map split-def comp-def*)

lemma *clearjunk-map-ran*: *clearjunk* (*map-ran f al*) = *map-ran f* (*clearjunk al*)
by (*simp add*: *map-ran-def split-def clearjunk-map*)

1.7 *merge*

qualified definition *merge* :: ('*key* × '*val*) *list* ⇒ ('*key* × '*val*) *list* ⇒ ('*key* × '*val*) *list*
where *merge qs ps* = *foldr* (λ(*k*, *v*). *update k v*) *ps qs*

lemma *merge-simps* [*simp*]:
merge qs [] = *qs*
merge qs (*p*#*ps*) = *update* (*fst p*) (*snd p*) (*merge qs ps*)
by (*simp-all add*: *merge-def split-def*)

lemma *merge-updates*: *merge qs ps* = *updates* (*rev* (*map fst ps*)) (*rev* (*map snd ps*)) *qs*
by (*simp add*: *merge-def updates-def foldr-conv-fold zip-rev zip-map-fst-snd*)

lemma *dom-merge*: *fst* ' *set* (*merge xs ys*) = *fst* ' *set xs* ∪ *fst* ' *set ys*
by (*induct ys arbitrary*: *xs*) (*auto simp add*: *dom-update*)

lemma *distinct-merge*:
assumes *distinct* (*map fst xs*)
shows *distinct* (*map fst* (*merge xs ys*))
using *assms* **by** (*simp add*: *merge-updates distinct-updates*)

lemma *clearjunk-merge*: *clearjunk* (*merge xs ys*) = *merge* (*clearjunk xs*) *ys*
by (*simp add*: *merge-updates clearjunk-updates*)

lemma *merge-conv'*: *map-of* (*merge xs ys*) = *map-of xs* ++ *map-of ys*
proof −
have *map-of* ∘ *fold* (*case-prod update*) (*rev ys*) =
fold (λ(*k*, *v*) *m*. *m*(*k* ↦ *v*)) (*rev ys*) ∘ *map-of*
by (*rule fold-commute*) (*simp add*: *update-conv' case-prod-beta split-def fun-eq-iff*)
then show *?thesis*
by (*simp add*: *merge-def map-add-map-of-foldr foldr-conv-fold fun-eq-iff*)
qed

corollary *merge-conv*: *map-of* (*merge xs ys*) *k* = (*map-of xs* ++ *map-of ys*) *k*
by (*simp add*: *merge-conv'*)

lemma *merge-empty*: *map-of* (*merge* [] *ys*) = *map-of ys*
by (*simp add*: *merge-conv'*)

lemma *merge-assoc* [*simp*]: *map-of* (*merge m1* (*merge m2 m3*)) = *map-of* (*merge* (*merge m1 m2*) *m3*)
by (*simp add*: *merge-conv'*)

lemma *merge-Some-iff*:
map-of (*merge m n*) *k* = *Some x* ⟷
map-of n k = *Some x* ∨ *map-of n k* = *None* ∧ *map-of m k* = *Some x*
by (*simp add*: *merge-conv′ map-add-Some-iff*)

lemmas *merge-SomeD* [*dest*!] = *merge-Some-iff* [*THEN iffD1*]

lemma *merge-find-right* [*simp*]: *map-of n k* = *Some v* ⟹ *map-of* (*merge m n*) *k* = *Some v*
by (*simp add*: *merge-conv′*)

lemma *merge-None* [*iff*]:
(*map-of* (*merge m n*) *k* = *None*) = (*map-of n k* = *None* ∧ *map-of m k* = *None*)
by (*simp add*: *merge-conv′*)

lemma *merge-upd* [*simp*]:
map-of (*merge m* (*update k v n*)) = *map-of* (*update k v* (*merge m n*))
by (*simp add*: *update-conv′ merge-conv′*)

lemma *merge-updatess* [*simp*]:
map-of (*merge m* (*updates xs ys n*)) = *map-of* (*updates xs ys* (*merge m n*))
by (*simp add*: *updates-conv′ merge-conv′*)

lemma *merge-append*: *map-of* (*xs* @ *ys*) = *map-of* (*merge ys xs*)
by (*simp add*: *merge-conv′*)

1.8 *compose*

qualified function *compose* :: (*′key* × *′a*) *list* ⇒ (*′a* × *′b*) *list* ⇒ (*′key* × *′b*) *list*
where
compose [] *ys* = []
| *compose* (*x* # *xs*) *ys* =
(*case map-of ys* (*snd x*) *of*
None ⇒ *compose* (*delete* (*fst x*) *xs*) *ys*
| *Some v* ⇒ (*fst x*, *v*) # *compose xs ys*)
by *pat-completeness auto*
termination
by (*relation measure* (*length* ∘ *fst*)) (*simp-all add*: *less-Suc-eq-le length-delete-le*)

lemma *compose-first-None* [*simp*]:
assumes *map-of xs k* = *None*
shows *map-of* (*compose xs ys*) *k* = *None*
using *assms* **by** (*induct xs ys rule*: *compose.induct*) (*auto split*: *option.splits if-split-asm*)

lemma *compose-conv*: *map-of* (*compose xs ys*) *k* = (*map-of ys* $\circ_m$ *map-of xs*) *k*
proof (*induct xs ys rule*: *compose.induct*)
case *1*

```
  then show ?case by simp
next
  case (2 x xs ys)
  show ?case
  proof (cases map-of ys (snd x))
    case None
    with 2 have hyp: map-of (compose (delete (fst x) xs) ys) k =
        (map-of ys ∘m map-of (delete (fst x) xs)) k
      by simp
    show ?thesis
    proof (cases fst x = k)
      case True
      from True delete-notin-dom [of k xs]
      have map-of (delete (fst x) xs) k = None
        by (simp add: map-of-eq-None-iff)
      with hyp show ?thesis
        using True None
        by simp
    next
      case False
      from False have map-of (delete (fst x) xs) k = map-of xs k
        by simp
      with hyp show ?thesis
        using False None by (simp add: map-comp-def)
    qed
  next
    case (Some v)
    with 2
    have map-of (compose xs ys) k = (map-of ys ∘m map-of xs) k
      by simp
    with Some show ?thesis
      by (auto simp add: map-comp-def)
  qed
qed

lemma compose-conv': map-of (compose xs ys) = (map-of ys ∘m map-of xs)
  by (rule ext) (rule compose-conv)

lemma compose-first-Some [simp]:
  assumes map-of xs k = Some v
  shows map-of (compose xs ys) k = map-of ys v
  using assms by (simp add: compose-conv)

lemma dom-compose: fst ` set (compose xs ys) ⊆ fst ` set xs
proof (induct xs ys rule: compose.induct)
  case 1
  then show ?case by simp
next
  case (2 x xs ys)
```

```
  show ?case
  proof (cases map-of ys (snd x))
    case None
    with 2.hyps
    have fst ` set (compose (delete (fst x) xs) ys) ⊆ fst ` set (delete (fst x) xs)
      by simp
    also
    have ... ⊆ fst ` set xs
      by (rule dom-delete-subset)
    finally show ?thesis
      using None
      by auto
  next
    case (Some v)
    with 2.hyps
    have fst ` set (compose xs ys) ⊆ fst ` set xs
      by simp
    with Some show ?thesis
      by auto
  qed
qed

lemma distinct-compose:
  assumes distinct (map fst xs)
  shows distinct (map fst (compose xs ys))
  using assms
proof (induct xs ys rule: compose.induct)
  case 1
  then show ?case by simp
next
  case (2 x xs ys)
  show ?case
  proof (cases map-of ys (snd x))
    case None
    with 2 show ?thesis by simp
  next
    case (Some v)
    with 2 dom-compose [of xs ys] show ?thesis
      by auto
  qed
qed

lemma compose-delete-twist: compose (delete k xs) ys = delete k (compose xs ys)
proof (induct xs ys rule: compose.induct)
  case 1
  then show ?case by simp
next
  case (2 x xs ys)
  show ?case
```

```
  proof (cases map-of ys (snd x))
    case None
    with 2 have hyp: compose (delete k (delete (fst x) xs)) ys =
        delete k (compose (delete (fst x) xs) ys)
      by simp
    show ?thesis
    proof (cases fst x = k)
      case True
      with None hyp show ?thesis
        by (simp add: delete-idem)
    next
      case False
      from None False hyp show ?thesis
        by (simp add: delete-twist)
    qed
  next
    case (Some v)
    with 2 have hyp: compose (delete k xs) ys = delete k (compose xs ys)
      by simp
    with Some show ?thesis
      by simp
  qed
qed

lemma compose-clearjunk: compose xs (clearjunk ys) = compose xs ys
  by (induct xs ys rule: compose.induct)
    (auto simp add: map-of-clearjunk split: option.splits)

lemma clearjunk-compose: clearjunk (compose xs ys) = compose (clearjunk xs) ys
  by (induct xs rule: clearjunk.induct)
   (auto split: option.splits simp add: clearjunk-delete delete-idem compose-delete-twist)

lemma compose-empty [simp]: compose xs [] = []
  by (induct xs) (auto simp add: compose-delete-twist)

lemma compose-Some-iff:
  (map-of (compose xs ys) k = Some v) ⟷
    (∃ k'. map-of xs k = Some k' ∧ map-of ys k' = Some v)
  by (simp add: compose-conv map-comp-Some-iff)

lemma map-comp-None-iff:
  map-of (compose xs ys) k = None ⟷
    (map-of xs k = None ∨ (∃ k'. map-of xs k = Some k' ∧ map-of ys k' = None))
  by (simp add: compose-conv map-comp-None-iff)
```

1.9 map-entry

```
qualified fun map-entry :: 'key ⇒ ('val ⇒ 'val) ⇒ ('key × 'val) list ⇒ ('key ×
'val) list
```

where
  map-entry k f [] = []
| *map-entry k f* (*p* # *ps*) =
    (*if fst p* = *k then* (*k*, *f* (*snd p*)) # *ps else p* # *map-entry k f ps*)

lemma *map-of-map-entry*:
  map-of (*map-entry k f xs*) =
    (*map-of xs*)(*k* := *case map-of xs k of None* ⇒ *None* | *Some v′* ⇒ *Some* (*f v′*))
  by (*induct xs*) *auto*

lemma *dom-map-entry*: *fst* ‘ *set* (*map-entry k f xs*) = *fst* ‘ *set xs*
  by (*induct xs*) *auto*

lemma *distinct-map-entry*:
  assumes *distinct* (*map fst xs*)
  shows *distinct* (*map fst* (*map-entry k f xs*))
  using *assms* **by** (*induct xs*) (*auto simp add*: *dom-map-entry*)

1.10 *map-default*

fun *map-default* :: ′*key* ⇒ ′*val* ⇒ (′*val* ⇒ ′*val*) ⇒ (′*key* × ′*val*) *list* ⇒ (′*key* × ′*val*) *list*
where
  map-default k v f [] = [(*k*, *v*)]
| *map-default k v f* (*p* # *ps*) =
    (*if fst p* = *k then* (*k*, *f* (*snd p*)) # *ps else p* # *map-default k v f ps*)

lemma *map-of-map-default*:
  map-of (*map-default k v f xs*) =
    (*map-of xs*)(*k* := *case map-of xs k of None* ⇒ *Some v* | *Some v′* ⇒ *Some* (*f v′*))
  by (*induct xs*) *auto*

lemma *dom-map-default*: *fst* ‘ *set* (*map-default k v f xs*) = *insert k* (*fst* ‘ *set xs*)
  by (*induct xs*) *auto*

lemma *distinct-map-default*:
  assumes *distinct* (*map fst xs*)
  shows *distinct* (*map fst* (*map-default k v f xs*))
  using *assms* **by** (*induct xs*) (*auto simp add*: *dom-map-default*)

end

end

2 Pointwise instantiation of functions to algebra type classes

theory *Function-Algebras*
imports *Main*
begin

Pointwise operations

instantiation *fun* :: (*type*, *plus*) *plus*
begin

definition $f + g = (\lambda x.\ f\ x + g\ x)$
instance ..

end

lemma *plus-fun-apply* [*simp*]:
$(f + g)\ x = f\ x + g\ x$
by (*simp add*: *plus-fun-def*)

instantiation *fun* :: (*type*, *zero*) *zero*
begin

definition $0 = (\lambda x.\ 0)$
instance ..

end

lemma *zero-fun-apply* [*simp*]:
$0\ x = 0$
by (*simp add*: *zero-fun-def*)

instantiation *fun* :: (*type*, *times*) *times*
begin

definition $f * g = (\lambda x.\ f\ x * g\ x)$
instance ..

end

lemma *times-fun-apply* [*simp*]:
$(f * g)\ x = f\ x * g\ x$
by (*simp add*: *times-fun-def*)

instantiation *fun* :: (*type*, *one*) *one*
begin

definition $1 = (\lambda x.\ 1)$
instance ..

end

lemma *one-fun-apply* [*simp*]:
1 x = *1*
by (*simp add*: *one-fun-def*)

Additive structures

instance *fun* :: (*type*, *semigroup-add*) *semigroup-add*
by *standard* (*simp add*: *fun-eq-iff add.assoc*)

instance *fun* :: (*type*, *cancel-semigroup-add*) *cancel-semigroup-add*
by *standard* (*simp-all add*: *fun-eq-iff*)

instance *fun* :: (*type*, *ab-semigroup-add*) *ab-semigroup-add*
by *standard* (*simp add*: *fun-eq-iff add.commute*)

instance *fun* :: (*type*, *cancel-ab-semigroup-add*) *cancel-ab-semigroup-add*
by *standard* (*simp-all add*: *fun-eq-iff diff-diff-eq*)

instance *fun* :: (*type*, *monoid-add*) *monoid-add*
by *standard* (*simp-all add*: *fun-eq-iff*)

instance *fun* :: (*type*, *comm-monoid-add*) *comm-monoid-add*
by *standard simp*

instance *fun* :: (*type*, *cancel-comm-monoid-add*) *cancel-comm-monoid-add* **..**

instance *fun* :: (*type*, *group-add*) *group-add*
by *standard* (*simp-all add*: *fun-eq-iff*)

instance *fun* :: (*type*, *ab-group-add*) *ab-group-add*
by *standard simp-all*

Multiplicative structures

instance *fun* :: (*type*, *semigroup-mult*) *semigroup-mult*
by *standard* (*simp add*: *fun-eq-iff mult.assoc*)

instance *fun* :: (*type*, *ab-semigroup-mult*) *ab-semigroup-mult*
by *standard* (*simp add*: *fun-eq-iff mult.commute*)

instance *fun* :: (*type*, *monoid-mult*) *monoid-mult*
by *standard* (*simp-all add*: *fun-eq-iff*)

instance *fun* :: (*type*, *comm-monoid-mult*) *comm-monoid-mult*
by *standard simp*

Misc

instance *fun* :: (*type*, *Rings.dvd*) *Rings.dvd* **..**

instance *fun* :: (*type*, *mult-zero*) *mult-zero*
by *standard* (*simp-all add*: *fun-eq-iff*)

instance *fun* :: (*type*, *zero-neq-one*) *zero-neq-one*
by *standard* (*simp add*: *fun-eq-iff*)

Ring structures

instance *fun* :: (*type*, *semiring*) *semiring*
by *standard* (*simp-all add*: *fun-eq-iff algebra-simps*)

instance *fun* :: (*type*, *comm-semiring*) *comm-semiring*
by *standard* (*simp add*: *fun-eq-iff algebra-simps*)

instance *fun* :: (*type*, *semiring-0*) *semiring-0* **..**

instance *fun* :: (*type*, *comm-semiring-0*) *comm-semiring-0* **..**

instance *fun* :: (*type*, *semiring-0-cancel*) *semiring-0-cancel* **..**

instance *fun* :: (*type*, *comm-semiring-0-cancel*) *comm-semiring-0-cancel* **..**

instance *fun* :: (*type*, *semiring-1*) *semiring-1* **..**

lemma *of-nat-fun*: *of-nat n* = (λ*x*::'*a*. *of-nat n*)
proof −
have *comp*: *comp* = (λ*f g x*. *f* (*g x*))
by (*rule ext*)+ *simp*
have *plus-fun*: *plus* = (λ*f g x*. *f x* + *g x*)
by (*rule ext*, *rule ext*) (*fact plus-fun-def*)
have *of-nat n* = (*comp* (*plus* (*1*::'*b*)) ^^ *n*) (λ*x*::'*a*. *0*)
by (*simp add*: *of-nat-def plus-fun zero-fun-def one-fun-def comp*)
also have ... = *comp* ((*plus 1*) ^^ *n*) (λ*x*::'*a*. *0*)
by (*simp only*: *comp-funpow*)
finally show *?thesis* **by** (*simp add*: *of-nat-def comp*)
qed

lemma *of-nat-fun-apply* [*simp*]:
of-nat n x = *of-nat n*
by (*simp add*: *of-nat-fun*)

instance *fun* :: (*type*, *comm-semiring-1*) *comm-semiring-1* **..**

instance *fun* :: (*type*, *semiring-1-cancel*) *semiring-1-cancel* **..**

instance *fun* :: (*type*, *comm-semiring-1-cancel*) *comm-semiring-1-cancel*
by *standard* (*auto simp add*: *times-fun-def algebra-simps*)

instance *fun* :: (*type*, *semiring-char-0*) *semiring-char-0*

proof
  from *inj-of-nat* **have** *inj* (λn (x::$'a$). *of-nat* n :: $'b$)
    by (*rule inj-fun*)
  then have *inj* (λn. *of-nat* n :: $'a \Rightarrow 'b$)
    by (*simp add*: *of-nat-fun*)
  then show *inj* (*of-nat* :: *nat* $\Rightarrow$ $'a \Rightarrow 'b$) **.**
qed

instance *fun* :: (*type*, *ring*) *ring* **..**

instance *fun* :: (*type*, *comm-ring*) *comm-ring* **..**

instance *fun* :: (*type*, *ring-1*) *ring-1* **..**

instance *fun* :: (*type*, *comm-ring-1*) *comm-ring-1* **..**

instance *fun* :: (*type*, *ring-char-0*) *ring-char-0* **..**

Ordered structures

instance *fun* :: (*type*, *ordered-ab-semigroup-add*) *ordered-ab-semigroup-add*
  by *standard* (*auto simp add*: *le-fun-def intro*: *add-left-mono*)

instance *fun* :: (*type*, *ordered-cancel-ab-semigroup-add*) *ordered-cancel-ab-semigroup-add*
..

instance *fun* :: (*type*, *ordered-ab-semigroup-add-imp-le*) *ordered-ab-semigroup-add-imp-le*
  by *standard* (*simp add*: *le-fun-def*)

instance *fun* :: (*type*, *ordered-comm-monoid-add*) *ordered-comm-monoid-add* **..**

instance *fun* :: (*type*, *ordered-cancel-comm-monoid-add*) *ordered-cancel-comm-monoid-add*
..

instance *fun* :: (*type*, *ordered-ab-group-add*) *ordered-ab-group-add* **..**

instance *fun* :: (*type*, *ordered-semiring*) *ordered-semiring*
  by *standard* (*auto simp add*: *le-fun-def intro*: *mult-left-mono mult-right-mono*)

instance *fun* :: (*type*, *dioid*) *dioid*
proof *standard*
  fix a b :: $'a \Rightarrow 'b$
  show $a \leq b \longleftrightarrow (\exists c.\ b = a + c)$
    unfolding *le-fun-def plus-fun-def fun-eq-iff choice-iff* [*symmetric*, *of* $\lambda x\ c.\ b\ x$
$= a\ x + c$]
    by (*intro arg-cong*[**where** *f*=*All*] *ext canonically-ordered-monoid-add-class.le-iff-add*)
qed

instance *fun* :: (*type*, *ordered-comm-semiring*) *ordered-comm-semiring*
  by *standard* (*fact mult-left-mono*)

instance *fun* :: (*type*, *ordered-cancel-semiring*) *ordered-cancel-semiring* **..**

instance *fun* :: (*type*, *ordered-cancel-comm-semiring*) *ordered-cancel-comm-semiring* **..**

instance *fun* :: (*type*, *ordered-ring*) *ordered-ring* **..**

instance *fun* :: (*type*, *ordered-comm-ring*) *ordered-comm-ring* **..**

lemmas *func-plus* = *plus-fun-def*
lemmas *func-zero* = *zero-fun-def*
lemmas *func-times* = *times-fun-def*
lemmas *func-one* = *one-fun-def*

end

3 Algebraic operations on sets

theory *Set-Algebras*
imports *Main*
begin

This library lifts operations like addition and multiplication to sets. It was designed to support asymptotic calculations. See the comments at the top of theory *BigO*.

instantiation *set* :: (*plus*) *plus*
begin

definition *plus-set* :: *'a::plus set* ⇒ *'a set* ⇒ *'a set* **where**
set-plus-def: $A + B = \{c.\ \exists\, a{\in}A.\ \exists\, b{\in}B.\ c = a + b\}$

instance ..

end

instantiation *set* :: (*times*) *times*
begin

definition *times-set* :: *'a::times set* ⇒ *'a set* ⇒ *'a set* **where**
set-times-def: $A * B = \{c.\ \exists\, a{\in}A.\ \exists\, b{\in}B.\ c = a * b\}$

instance ..

end

instantiation *set* :: (*zero*) *zero*
begin

definition
set-zero[*simp*]: (*0*::′*a*::*zero set*) = {*0*}

instance ..

end

instantiation *set* :: (*one*) *one*
begin

definition
set-one[*simp*]: (*1*::′*a*::*one set*) = {*1*}

instance ..

end

definition *elt-set-plus* :: ′*a*::*plus* ⇒ ′*a set* ⇒ ′*a set* (**infixl** +*o* *70*) **where**
a +*o* *B* = {*c*. ∃ *b*∈*B*. *c* = *a* + *b*}

definition *elt-set-times* :: ′*a*::*times* ⇒ ′*a set* ⇒ ′*a set* (**infixl** ∗*o* *80*) **where**
a ∗*o* *B* = {*c*. ∃ *b*∈*B*. *c* = *a* ∗ *b*}

abbreviation (*input*) *elt-set-eq* :: ′*a* ⇒ ′*a set* ⇒ *bool* (**infix** =*o* *50*) **where**
x =*o* *A* ≡ *x* ∈ *A*

instance *set* :: (*semigroup-add*) *semigroup-add*
by *standard* (*force simp add*: *set-plus-def add.assoc*)

instance *set* :: (*ab-semigroup-add*) *ab-semigroup-add*
by *standard* (*force simp add*: *set-plus-def add.commute*)

instance *set* :: (*monoid-add*) *monoid-add*
by *standard* (*simp-all add*: *set-plus-def*)

instance *set* :: (*comm-monoid-add*) *comm-monoid-add*
by *standard* (*simp-all add*: *set-plus-def*)

instance *set* :: (*semigroup-mult*) *semigroup-mult*
by *standard* (*force simp add*: *set-times-def mult.assoc*)

instance *set* :: (*ab-semigroup-mult*) *ab-semigroup-mult*
by *standard* (*force simp add*: *set-times-def mult.commute*)

instance *set* :: (*monoid-mult*) *monoid-mult*
by *standard* (*simp-all add*: *set-times-def*)

instance *set* :: (*comm-monoid-mult*) *comm-monoid-mult*

by *standard* (*simp-all add*: *set-times-def*)

lemma *set-plus-intro* [*intro*]: $a \in C \Longrightarrow b \in D \Longrightarrow a + b \in C + D$
by (*auto simp add*: *set-plus-def*)

lemma *set-plus-elim*:
assumes $x \in A + B$
obtains *a b* **where** $x = a + b$ **and** $a \in A$ **and** $b \in B$
using *assms* **unfolding** *set-plus-def* **by** *fast*

lemma *set-plus-intro2* [*intro*]: $b \in C \Longrightarrow a + b \in a +o\ C$
by (*auto simp add*: *elt-set-plus-def*)

lemma *set-plus-rearrange*:
$((a::'a::comm\text{-}monoid\text{-}add) +o\ C) + (b +o\ D) = (a + b) +o\ (C + D)$
apply (*auto simp add*: *elt-set-plus-def set-plus-def ac-simps*)
apply (*rule-tac* $x = ba + bb$ **in** *exI*)
apply (*auto simp add*: *ac-simps*)
apply (*rule-tac* $x = aa + a$ **in** *exI*)
apply (*auto simp add*: *ac-simps*)
done

lemma *set-plus-rearrange2*: $(a::'a::semigroup\text{-}add) +o\ (b +o\ C) = (a + b) +o\ C$
by (*auto simp add*: *elt-set-plus-def add.assoc*)

lemma *set-plus-rearrange3*: $((a::'a::semigroup\text{-}add) +o\ B) + C = a +o\ (B + C)$
apply (*auto simp add*: *elt-set-plus-def set-plus-def*)
apply (*blast intro*: *ac-simps*)
apply (*rule-tac* $x = a + aa$ **in** *exI*)
apply (*rule conjI*)
apply (*rule-tac* $x = aa$ **in** *bexI*)
apply *auto*
apply (*rule-tac* $x = ba$ **in** *bexI*)
apply (*auto simp add*: *ac-simps*)
done

theorem *set-plus-rearrange4*: $C + ((a::'a::comm\text{-}monoid\text{-}add) +o\ D) = a +o\ (C + D)$
apply (*auto simp add*: *elt-set-plus-def set-plus-def ac-simps*)
apply (*rule-tac* $x = aa + ba$ **in** *exI*)
apply (*auto simp add*: *ac-simps*)
done

lemmas *set-plus-rearranges* = *set-plus-rearrange set-plus-rearrange2 set-plus-rearrange3 set-plus-rearrange4*

lemma *set-plus-mono* [*intro!*]: $C \subseteq D \Longrightarrow a +o\ C \subseteq a +o\ D$
by (*auto simp add*: *elt-set-plus-def*)

lemma *set-plus-mono2* [*intro*]: $(C::'a::plus\ set) \subseteq D \Longrightarrow E \subseteq F \Longrightarrow C + E \subseteq D + F$
  by (*auto simp add*: *set-plus-def*)

lemma *set-plus-mono3* [*intro*]: $a \in C \Longrightarrow a +o\ D \subseteq C + D$
  by (*auto simp add*: *elt-set-plus-def set-plus-def*)

lemma *set-plus-mono4* [*intro*]: $(a::'a::comm\text{-}monoid\text{-}add) \in C \Longrightarrow a +o\ D \subseteq D + C$
  by (*auto simp add*: *elt-set-plus-def set-plus-def ac-simps*)

lemma *set-plus-mono5*: $a \in C \Longrightarrow B \subseteq D \Longrightarrow a +o\ B \subseteq C + D$
  apply (*subgoal-tac* $a +o\ B \subseteq a +o\ D$)
   apply (*erule order-trans*)
   apply (*erule set-plus-mono3*)
  apply (*erule set-plus-mono*)
  done

lemma *set-plus-mono-b*: $C \subseteq D \Longrightarrow x \in a +o\ C \Longrightarrow x \in a +o\ D$
  apply (*frule set-plus-mono*)
  apply *auto*
  done

lemma *set-plus-mono2-b*: $C \subseteq D \Longrightarrow E \subseteq F \Longrightarrow x \in C + E \Longrightarrow x \in D + F$
  apply (*frule set-plus-mono2*)
   prefer *2*
   apply *force*
  apply *assumption*
  done

lemma *set-plus-mono3-b*: $a \in C \Longrightarrow x \in a +o\ D \Longrightarrow x \in C + D$
  apply (*frule set-plus-mono3*)
  apply *auto*
  done

lemma *set-plus-mono4-b*: $(a::'a::comm\text{-}monoid\text{-}add) : C \Longrightarrow x \in a +o\ D \Longrightarrow x \in D + C$
  apply (*frule set-plus-mono4*)
  apply *auto*
  done

lemma *set-zero-plus* [*simp*]: $(0::'a::comm\text{-}monoid\text{-}add) +o\ C = C$
  by (*auto simp add*: *elt-set-plus-def*)

lemma *set-zero-plus2*: $(0::'a::comm\text{-}monoid\text{-}add) \in A \Longrightarrow B \subseteq A + B$
  apply (*auto simp add*: *set-plus-def*)
  apply (*rule-tac* $x = 0$ **in** *bexI*)
   apply (*rule-tac* $x = x$ **in** *bexI*)
    apply (*auto simp add*: *ac-simps*)

done

lemma *set-plus-imp-minus*: $(a::'a::ab\text{-}group\text{-}add) : b +o\ C \Longrightarrow (a - b) \in C$
by (*auto simp add*: *elt-set-plus-def ac-simps*)

lemma *set-minus-imp-plus*: $(a::'a::ab\text{-}group\text{-}add) - b : C \Longrightarrow a \in b +o\ C$
apply (*auto simp add*: *elt-set-plus-def ac-simps*)
apply (*subgoal-tac* $a = (a + - b) + b$)
apply (*rule bexI*, *assumption*)
apply (*auto simp add*: *ac-simps*)
done

lemma *set-minus-plus*: $(a::'a::ab\text{-}group\text{-}add) - b \in C \longleftrightarrow a \in b +o\ C$
by (*rule iffI*, *rule set-minus-imp-plus*, *assumption*, *rule set-plus-imp-minus*)

lemma *set-times-intro* [*intro*]: $a \in C \Longrightarrow b \in D \Longrightarrow a * b \in C * D$
by (*auto simp add*: *set-times-def*)

lemma *set-times-elim*:
assumes $x \in A * B$
obtains *a b* **where** $x = a * b$ **and** $a \in A$ **and** $b \in B$
using *assms* **unfolding** *set-times-def* **by** *fast*

lemma *set-times-intro2* [*intro*!]: $b \in C \Longrightarrow a * b \in a *o\ C$
by (*auto simp add*: *elt-set-times-def*)

lemma *set-times-rearrange*:
$((a::'a::comm\text{-}monoid\text{-}mult) *o\ C) * (b *o\ D) = (a * b) *o\ (C * D)$
apply (*auto simp add*: *elt-set-times-def set-times-def*)
apply (*rule-tac* $x = ba * bb$ **in** *exI*)
apply (*auto simp add*: *ac-simps*)
apply (*rule-tac* $x = aa * a$ **in** *exI*)
apply (*auto simp add*: *ac-simps*)
done

lemma *set-times-rearrange2*:
$(a::'a::semigroup\text{-}mult) *o\ (b *o\ C) = (a * b) *o\ C$
by (*auto simp add*: *elt-set-times-def mult.assoc*)

lemma *set-times-rearrange3*:
$((a::'a::semigroup\text{-}mult) *o\ B) * C = a *o\ (B * C)$
apply (*auto simp add*: *elt-set-times-def set-times-def*)
apply (*blast intro*: *ac-simps*)
apply (*rule-tac* $x = a * aa$ **in** *exI*)
apply (*rule conjI*)
apply (*rule-tac* $x = aa$ **in** *bexI*)
apply *auto*
apply (*rule-tac* $x = ba$ **in** *bexI*)
apply (*auto simp add*: *ac-simps*)

done

theorem *set-times-rearrange4*:
$C * ((a::'a::comm\text{-}monoid\text{-}mult) *o\ D) = a *o\ (C * D)$
apply (*auto simp add*: *elt-set-times-def set-times-def ac-simps*)
apply (*rule-tac* $x = aa * ba$ **in** *exI*)
apply (*auto simp add*: *ac-simps*)
done

lemmas *set-times-rearranges* = *set-times-rearrange set-times-rearrange2 set-times-rearrange3 set-times-rearrange4*

lemma *set-times-mono* [*intro*]: $C \subseteq D \Longrightarrow a *o\ C \subseteq a *o\ D$
by (*auto simp add*: *elt-set-times-def*)

lemma *set-times-mono2* [*intro*]: $(C::'a::times\ set) \subseteq D \Longrightarrow E \subseteq F \Longrightarrow C * E \subseteq D * F$
by (*auto simp add*: *set-times-def*)

lemma *set-times-mono3* [*intro*]: $a \in C \Longrightarrow a *o\ D \subseteq C * D$
by (*auto simp add*: *elt-set-times-def set-times-def*)

lemma *set-times-mono4* [*intro*]: $(a::'a::comm\text{-}monoid\text{-}mult) : C \Longrightarrow a *o\ D \subseteq D * C$
by (*auto simp add*: *elt-set-times-def set-times-def ac-simps*)

lemma *set-times-mono5*: $a \in C \Longrightarrow B \subseteq D \Longrightarrow a *o\ B \subseteq C * D$
apply (*subgoal-tac* $a *o\ B \subseteq a *o\ D$)
apply (*erule order-trans*)
apply (*erule set-times-mono3*)
apply (*erule set-times-mono*)
done

lemma *set-times-mono-b*: $C \subseteq D \Longrightarrow x \in a *o\ C \Longrightarrow x \in a *o\ D$
apply (*frule set-times-mono*)
apply *auto*
done

lemma *set-times-mono2-b*: $C \subseteq D \Longrightarrow E \subseteq F \Longrightarrow x \in C * E \Longrightarrow x \in D * F$
apply (*frule set-times-mono2*)
prefer *2*
apply *force*
apply *assumption*
done

lemma *set-times-mono3-b*: $a \in C \Longrightarrow x \in a *o\ D \Longrightarrow x \in C * D$
apply (*frule set-times-mono3*)
apply *auto*
done

lemma *set-times-mono4-b*: $(a::'a::comm\text{-}monoid\text{-}mult) \in C \Longrightarrow x \in a *o\ D \Longrightarrow x \in D * C$
apply (*frule set-times-mono4*)
apply *auto*
done

lemma *set-one-times* [*simp*]: $(1::'a::comm\text{-}monoid\text{-}mult) *o\ C = C$
by (*auto simp add*: *elt-set-times-def*)

lemma *set-times-plus-distrib*:
$(a::'a::semiring) *o\ (b +o\ C) = (a * b) +o\ (a *o\ C)$
by (*auto simp add*: *elt-set-plus-def elt-set-times-def ring-distribs*)

lemma *set-times-plus-distrib2*:
$(a::'a::semiring) *o\ (B + C) = (a *o\ B) + (a *o\ C)$
apply (*auto simp add*: *set-plus-def elt-set-times-def ring-distribs*)
apply *blast*
apply (*rule-tac* $x = b + bb$ **in** *exI*)
apply (*auto simp add*: *ring-distribs*)
done

lemma *set-times-plus-distrib3*: $((a::'a::semiring) +o\ C) * D \subseteq a *o\ D + C * D$
apply (*auto simp add*:
elt-set-plus-def elt-set-times-def set-times-def
set-plus-def ring-distribs)
apply *auto*
done

lemmas *set-times-plus-distribs* =
set-times-plus-distrib
set-times-plus-distrib2

lemma *set-neg-intro*: $(a::'a::ring\text{-}1) \in (-\ 1) *o\ C \Longrightarrow -\ a \in C$
by (*auto simp add*: *elt-set-times-def*)

lemma *set-neg-intro2*: $(a::'a::ring\text{-}1) \in C \Longrightarrow -\ a \in (-\ 1) *o\ C$
by (*auto simp add*: *elt-set-times-def*)

lemma *set-plus-image*: $S + T = (\lambda(x, y).\ x + y)$ ` $(S \times T)$
unfolding *set-plus-def* **by** (*fastforce simp*: *image-iff*)

lemma *set-times-image*: $S * T = (\lambda(x, y).\ x * y)$ ` $(S \times T)$
unfolding *set-times-def* **by** (*fastforce simp*: *image-iff*)

lemma *finite-set-plus*: *finite s* $\Longrightarrow$ *finite t* $\Longrightarrow$ *finite* $(s + t)$
unfolding *set-plus-image* **by** *simp*

lemma *finite-set-times*: *finite s* $\Longrightarrow$ *finite t* $\Longrightarrow$ *finite* $(s * t)$

unfolding *set-times-image* **by** *simp*

lemma *set-setsum-alt*:
assumes *fin*: *finite I*
shows *setsum S I* = {*setsum s I* |*s*. ∀ *i*∈*I*. *s i* ∈ *S i*}
(**is** - = *?setsum I*)
using *fin*
proof *induct*
case *empty*
then show *?case* **by** *simp*
next
case (*insert x F*)
have *setsum S* (*insert x F*) = *S x* + *?setsum F*
using *insert.hyps* **by** *auto*
also have ... = {*s x* + *setsum s F* |*s*. ∀ *i*∈*insert x F*. *s i* ∈ *S i*}
unfolding *set-plus-def*
proof *safe*
fix *y s*
assume *y* ∈ *S x* ∀ *i*∈*F*. *s i* ∈ *S i*
then show ∃ *s'*. *y* + *setsum s F* = *s' x* + *setsum s' F* ∧ (∀ *i*∈*insert x F*. *s' i* ∈ *S i*)
using *insert.hyps*
by (*intro exI*[*of* - λ*i*. *if i* ∈ *F then s i else y*]) (*auto simp add*: *set-plus-def*)
qed *auto*
finally show *?case*
using *insert.hyps* **by** *auto*
qed

lemma *setsum-set-cond-linear*:
fixes *f* :: *'a*::*comm-monoid-add set* ⇒ *'b*::*comm-monoid-add set*
assumes [*intro*!]: ⋀*A B*. *P A* ⟹ *P B* ⟹ *P* (*A* + *B*) *P* {*0*}
and *f*: ⋀*A B*. *P A* ⟹ *P B* ⟹ *f* (*A* + *B*) = *f A* + *f B f* {*0*} = {*0*}
assumes *all*: ⋀*i*. *i* ∈ *I* ⟹ *P* (*S i*)
shows *f* (*setsum S I*) = *setsum* (*f* ∘ *S*) *I*
proof (*cases finite I*)
case *True*
from *this all* **show** *?thesis*
proof *induct*
case *empty*
then show *?case* **by** (*auto intro*!: *f*)
next
case (*insert x F*)
from ‹*finite F*› ‹⋀*i*. *i* ∈ *insert x F* ⟹ *P* (*S i*)› **have** *P* (*setsum S F*)
by *induct auto*
with *insert* **show** *?case*
by (*simp*, *subst f*) *auto*
qed
next
case *False*

then show *?thesis* **by** (*auto intro*!: *f*)
qed

lemma *setsum-set-linear*:
fixes f :: *'a::comm-monoid-add set* ⇒ *'b::comm-monoid-add set*
assumes $\bigwedge A\ B.\ f(A) + f(B) = f(A + B)\ f\ \{0\} = \{0\}$
shows f (*setsum S I*) = *setsum* ($f \circ S$) *I*
using *setsum-set-cond-linear*[*of* λ*x*. *True f I S*] *assms* **by** *auto*

lemma *set-times-Un-distrib*:
$A * (B \cup C) = A * B \cup A * C$
$(A \cup B) * C = A * C \cup B * C$
by (*auto simp*: *set-times-def*)

lemma *set-times-UNION-distrib*:
$A *$ *UNION I M* = $(\bigcup i{\in}I.\ A * M\ i)$
UNION I M $* A = (\bigcup i{\in}I.\ M\ i * A)$
by (*auto simp*: *set-times-def*)

end

4 Big O notation

theory *BigO*
imports *Complex-Main Function-Algebras Set-Algebras*
begin

This library is designed to support asymptotic "big O" calculations, i.e. reasoning with expressions of the form $f = O(g)$ and $f = g + O(h)$. An earlier version of this library is described in detail in [1].

The main changes in this version are as follows:

- We have eliminated the *O* operator on sets. (Most uses of this seem to be inessential.)
- We no longer use + as output syntax for +*o*
- Lemmas involving *sumr* have been replaced by more general lemmas involving '*setsum*.
- The library has been expanded, with e.g. support for expressions of the form $f < g + O(h)$.

Note also since the Big O library includes rules that demonstrate set inclusion, to use the automated reasoners effectively with the library one should redeclare the theorem *subsetI* as an intro rule, rather than as an *intro*! rule, for example, using **declare** *subsetI* [*del*, *intro*].

4.1 Definitions

definition *bigo* :: $('a \Rightarrow 'b{::}linordered\text{-}idom) \Rightarrow ('a \Rightarrow 'b)\ set$ $((1O'(\text{-}')))$
where $O(f{::}\ 'a \Rightarrow 'b) = \{h.\ \exists\, c.\ \forall\, x.\ |h\ x| \le c * |f\ x|\}$

lemma *bigo-pos-const*:
$(\exists\, c{::}'a{::}linordered\text{-}idom.\ \forall\, x.\ |h\ x| \le c * |f\ x|) \longleftrightarrow$
$(\exists\, c.\ 0 < c \wedge (\forall\, x.\ |h\ x| \le c * |f\ x|))$
apply *auto*
apply (*case-tac* $c = 0$)
apply *simp*
apply (*rule-tac* $x = 1$ **in** *exI*)
apply *simp*
apply (*rule-tac* $x = |c|$ **in** *exI*)
apply *auto*
apply (*subgoal-tac* $c * |f\ x| \le |c| * |f\ x|$)
apply (*erule-tac* $x = x$ **in** *allE*)
apply *force*
apply (*rule mult-right-mono*)
apply (*rule abs-ge-self*)
apply (*rule abs-ge-zero*)
done

lemma *bigo-alt-def*: $O(f) = \{h.\ \exists\, c.\ 0 < c \wedge (\forall\, x.\ |h\ x| \le c * |f\ x|)\}$
by (*auto simp add*: *bigo-def bigo-pos-const*)

lemma *bigo-elt-subset* [*intro*]: $f \in O(g) \Longrightarrow O(f) \le O(g)$
apply (*auto simp add*: *bigo-alt-def*)
apply (*rule-tac* $x = ca * c$ **in** *exI*)
apply (*rule conjI*)
apply *simp*
apply (*rule allI*)
apply (*drule-tac* $x = xa$ **in** *spec*)+
apply (*subgoal-tac* $ca * |f\ xa| \le ca * (c * |g\ xa|)$)
apply (*erule order-trans*)
apply (*simp add*: *ac-simps*)
apply (*rule mult-left-mono*, *assumption*)
apply (*rule order-less-imp-le*, *assumption*)
done

lemma *bigo-refl* [*intro*]: $f \in O(f)$
apply(*auto simp add*: *bigo-def*)
apply(*rule-tac* $x = 1$ **in** *exI*)
apply *simp*
done

lemma *bigo-zero*: $0 \in O(g)$
apply (*auto simp add*: *bigo-def func-zero*)
apply (*rule-tac* $x = 0$ **in** *exI*)
apply *auto*

done

lemma *bigo-zero2*: $O(\lambda x.\ 0) = \{\lambda x.\ 0\}$
by (*auto simp add*: *bigo-def*)

lemma *bigo-plus-self-subset* [*intro*]: $O(f) + O(f) \subseteq O(f)$
apply (*auto simp add*: *bigo-alt-def set-plus-def*)
apply (*rule-tac* $x = c + ca$ **in** *exI*)
apply *auto*
apply (*simp add*: *ring-distribs func-plus*)
apply (*rule order-trans*)
apply (*rule abs-triangle-ineq*)
apply (*rule add-mono*)
apply *force*
apply *force*
done

lemma *bigo-plus-idemp* [*simp*]: $O(f) + O(f) = O(f)$
apply (*rule equalityI*)
apply (*rule bigo-plus-self-subset*)
apply (*rule set-zero-plus2*)
apply (*rule bigo-zero*)
done

lemma *bigo-plus-subset* [*intro*]: $O(f + g) \subseteq O(f) + O(g)$
apply (*rule subsetI*)
apply (*auto simp add*: *bigo-def bigo-pos-const func-plus set-plus-def*)
apply (*subst bigo-pos-const* [*symmetric*])+
apply (*rule-tac* $x = \lambda n.$ *if* $|g\ n| \leq |f\ n|$ *then* $x\ n$ *else* 0 **in** *exI*)
apply (*rule conjI*)
apply (*rule-tac* $x = c + c$ **in** *exI*)
apply (*clarsimp*)
apply (*subgoal-tac* $c * |f\ xa + g\ xa| \leq (c + c) * |f\ xa|$)
apply (*erule-tac* $x = xa$ **in** *allE*)
apply (*erule order-trans*)
apply (*simp*)
apply (*subgoal-tac* $c * |f\ xa + g\ xa| \leq c * (|f\ xa| + |g\ xa|)$)
apply (*erule order-trans*)
apply (*simp add*: *ring-distribs*)
apply (*rule mult-left-mono*)
apply (*simp add*: *abs-triangle-ineq*)
apply (*simp add*: *order-less-le*)
apply (*rule-tac* $x = \lambda n.$ *if* $|f\ n| < |g\ n|$ *then* $x\ n$ *else* 0 **in** *exI*)
apply (*rule conjI*)
apply (*rule-tac* $x = c + c$ **in** *exI*)
apply *auto*
apply (*subgoal-tac* $c * |f\ xa + g\ xa| \leq (c + c) * |g\ xa|$)
apply (*erule-tac* $x = xa$ **in** *allE*)
apply (*erule order-trans*)

apply *simp*
apply (*subgoal-tac* $c * |f\ xa + g\ xa| \leq c * (|f\ xa| + |g\ xa|)$)
apply (*erule order-trans*)
apply (*simp add*: *ring-distribs*)
apply (*rule mult-left-mono*)
apply (*rule abs-triangle-ineq*)
apply (*simp add*: *order-less-le*)
done

lemma *bigo-plus-subset2* [*intro*]: $A \subseteq O(f) \Longrightarrow B \subseteq O(f) \Longrightarrow A + B \subseteq O(f)$
apply (*subgoal-tac* $A + B \subseteq O(f) + O(f)$)
apply (*erule order-trans*)
apply *simp*
apply (*auto del*: *subsetI simp del*: *bigo-plus-idemp*)
done

lemma *bigo-plus-eq*: $\forall x.\ 0 \leq f\ x \Longrightarrow \forall x.\ 0 \leq g\ x \Longrightarrow O(f + g) = O(f) + O(g)$
apply (*rule equalityI*)
apply (*rule bigo-plus-subset*)
apply (*simp add*: *bigo-alt-def set-plus-def func-plus*)
apply *clarify*
apply (*rule-tac* $x = max\ c\ ca$ **in** *exI*)
apply (*rule conjI*)
apply (*subgoal-tac* $c \leq max\ c\ ca$)
apply (*erule order-less-le-trans*)
apply *assumption*
apply (*rule max.cobounded1*)
apply *clarify*
apply (*drule-tac* $x = xa$ **in** *spec*)+
apply (*subgoal-tac* $0 \leq f\ xa + g\ xa$)
apply (*simp add*: *ring-distribs*)
apply (*subgoal-tac* $|a\ xa + b\ xa| \leq |a\ xa| + |b\ xa|$)
apply (*subgoal-tac* $|a\ xa| + |b\ xa| \leq max\ c\ ca * f\ xa + max\ c\ ca * g\ xa$)
apply *force*
apply (*rule add-mono*)
apply (*subgoal-tac* $c * f\ xa \leq max\ c\ ca * f\ xa$)
apply *force*
apply (*rule mult-right-mono*)
apply (*rule max.cobounded1*)
apply *assumption*
apply (*subgoal-tac* $ca * g\ xa \leq max\ c\ ca * g\ xa$)
apply *force*
apply (*rule mult-right-mono*)
apply (*rule max.cobounded2*)
apply *assumption*
apply (*rule abs-triangle-ineq*)
apply (*rule add-nonneg-nonneg*)
apply *assumption+*
done

lemma *bigo-bounded-alt*: $\forall x.\ 0 \leq f\,x \Longrightarrow \forall x.\ f\,x \leq c * g\,x \Longrightarrow f \in O(g)$
apply (*auto simp add*: *bigo-def*)
apply (*rule-tac* $x = |c|$ **in** *exI*)
apply *auto*
apply (*drule-tac* $x = x$ **in** *spec*)+
apply (*simp add*: *abs-mult* [*symmetric*])
done

lemma *bigo-bounded*: $\forall x.\ 0 \leq f\,x \Longrightarrow \forall x.\ f\,x \leq g\,x \Longrightarrow f \in O(g)$
apply (*erule bigo-bounded-alt* [*of f 1 g*])
apply *simp*
done

lemma *bigo-bounded2*: $\forall x.\ lb\ x \leq f\,x \Longrightarrow \forall x.\ f\,x \leq lb\ x + g\ x \Longrightarrow f \in lb +o O(g)$
apply (*rule set-minus-imp-plus*)
apply (*rule bigo-bounded*)
apply (*auto simp add*: *fun-Compl-def func-plus*)
apply (*drule-tac* $x = x$ **in** *spec*)+
apply *force*
done

lemma *bigo-abs*: $(\lambda x.\ |f\,x|) =o\ O(f)$
apply (*unfold bigo-def*)
apply *auto*
apply (*rule-tac* $x = 1$ **in** *exI*)
apply *auto*
done

lemma *bigo-abs2*: $f =o\ O(\lambda x.\ |f\,x|)$
apply (*unfold bigo-def*)
apply *auto*
apply (*rule-tac* $x = 1$ **in** *exI*)
apply *auto*
done

lemma *bigo-abs3*: $O(f) = O(\lambda x.\ |f\,x|)$
apply (*rule equalityI*)
apply (*rule bigo-elt-subset*)
apply (*rule bigo-abs2*)
apply (*rule bigo-elt-subset*)
apply (*rule bigo-abs*)
done

lemma *bigo-abs4*: $f =o\ g +o\ O(h) \Longrightarrow (\lambda x.\ |f\,x|) =o\ (\lambda x.\ |g\ x|) +o\ O(h)$
apply (*drule set-plus-imp-minus*)
apply (*rule set-minus-imp-plus*)
apply (*subst fun-diff-def*)

proof –
  assume *a*: $f - g \in O(h)$
  have $(\lambda x. |f\, x| - |g\, x|) =o\ O(\lambda x. ||f\, x| - |g\, x||)$
    by (*rule bigo-abs2*)
  also have ... $\subseteq O(\lambda x. |f\, x - g\, x|)$
    apply (*rule bigo-elt-subset*)
    apply (*rule bigo-bounded*)
    apply *force*
    apply (*rule allI*)
    apply (*rule abs-triangle-ineq3*)
    done
  also have ... $\subseteq O(f - g)$
    apply (*rule bigo-elt-subset*)
    apply (*subst fun-diff-def*)
    apply (*rule bigo-abs*)
    done
  also from *a* **have** ... $\subseteq O(h)$
    by (*rule bigo-elt-subset*)
  finally show $(\lambda x. |f\, x| - |g\, x|) \in O(h)$**.**
qed

lemma *bigo-abs5*: $f =o\ O(g) \Longrightarrow (\lambda x. |f\, x|) =o\ O(g)$
  by (*unfold bigo-def*, *auto*)

lemma *bigo-elt-subset2* [*intro*]: $f \in g +o\ O(h) \Longrightarrow O(f) \subseteq O(g) + O(h)$
proof –
  assume $f \in g +o\ O(h)$
  also have ... $\subseteq O(g) + O(h)$
    by (*auto del*: *subsetI*)
  also have ... $= O(\lambda x. |g\, x|) + O(\lambda x. |h\, x|)$
    apply (*subst bigo-abs3* [*symmetric*])+
    apply (*rule refl*)
    done
  also have ... $= O((\lambda x. |g\, x|) + (\lambda x. |h\, x|))$
    by (*rule bigo-plus-eq* [*symmetric*]) *auto*
  finally have $f \in$ ... **.**
  then have $O(f) \subseteq$ ...
    by (*elim bigo-elt-subset*)
  also have ... $= O(\lambda x. |g\, x|) + O(\lambda x. |h\, x|)$
    by (*rule bigo-plus-eq*, *auto*)
  finally show *?thesis*
    by (*simp add*: *bigo-abs3* [*symmetric*])
qed

lemma *bigo-mult* [*intro*]: $O(f)*O(g) \subseteq O(f * g)$
  apply (*rule subsetI*)
  apply (*subst bigo-def*)
  apply (*auto simp add*: *bigo-alt-def set-times-def func-times*)
  apply (*rule-tac* $x = c * ca$ **in** *exI*)

```
  apply (rule allI)
  apply (erule-tac x = x in allE)+
  apply (subgoal-tac c ∗ ca ∗ |f x ∗ g x| = (c ∗ |f x|) ∗ (ca ∗ |g x|))
  apply (erule ssubst)
  apply (subst abs-mult)
  apply (rule mult-mono)
  apply assumption+
  apply auto
  apply (simp add: ac-simps abs-mult)
  done

lemma bigo-mult2 [intro]: f ∗o O(g) ⊆ O(f ∗ g)
  apply (auto simp add: bigo-def elt-set-times-def func-times abs-mult)
  apply (rule-tac x = c in exI)
  apply auto
  apply (drule-tac x = x in spec)
  apply (subgoal-tac |f x| ∗ |b x| ≤ |f x| ∗ (c ∗ |g x|))
  apply (force simp add: ac-simps)
  apply (rule mult-left-mono, assumption)
  apply (rule abs-ge-zero)
  done

lemma bigo-mult3: f ∈ O(h) ⟹ g ∈ O(j) ⟹ f ∗ g ∈ O(h ∗ j)
  apply (rule subsetD)
  apply (rule bigo-mult)
  apply (erule set-times-intro, assumption)
  done

lemma bigo-mult4 [intro]: f ∈ k +o O(h) ⟹ g ∗ f ∈ (g ∗ k) +o O(g ∗ h)
  apply (drule set-plus-imp-minus)
  apply (rule set-minus-imp-plus)
  apply (drule bigo-mult3 [where g = g and j = g])
  apply (auto simp add: algebra-simps)
  done

lemma bigo-mult5:
  fixes f :: 'a ⇒ 'b::linordered-field
  assumes ∀ x. f x ≠ 0
  shows O(f ∗ g) ⊆ f ∗o O(g)
proof
  fix h
  assume h ∈ O(f ∗ g)
  then have (λx. 1 / (f x)) ∗ h ∈ (λx. 1 / f x) ∗o O(f ∗ g)
    by auto
  also have ... ⊆ O((λx. 1 / f x) ∗ (f ∗ g))
    by (rule bigo-mult2)
  also have (λx. 1 / f x) ∗ (f ∗ g) = g
    apply (simp add: func-times)
    apply (rule ext)
```

apply (*simp add*: *assms nonzero-divide-eq-eq ac-simps*)
done
finally have $(\lambda x.\ (1::'b)\ /\ f\ x) * h \in O(g)$ **.**
then have $f * ((\lambda x.\ (1::'b)\ /\ f\ x) * h) \in f *o\ O(g)$
by *auto*
also have $f * ((\lambda x.\ (1::'b)\ /\ f\ x) * h) = h$
apply (*simp add*: *func-times*)
apply (*rule ext*)
apply (*simp add*: *assms nonzero-divide-eq-eq ac-simps*)
done
finally show $h \in f *o\ O(g)$ **.**
qed

lemma *bigo-mult6*:
fixes $f :: 'a \Rightarrow 'b::linordered\text{-}field$
shows $\forall x.\ f\ x \neq 0 \Longrightarrow O(f * g) = f *o\ O(g)$
apply (*rule equalityI*)
apply (*erule bigo-mult5*)
apply (*rule bigo-mult2*)
done

lemma *bigo-mult7*:
fixes $f :: 'a \Rightarrow 'b::linordered\text{-}field$
shows $\forall x.\ f\ x \neq 0 \Longrightarrow O(f * g) \subseteq O(f) * O(g)$
apply (*subst bigo-mult6*)
apply *assumption*
apply (*rule set-times-mono3*)
apply (*rule bigo-refl*)
done

lemma *bigo-mult8*:
fixes $f :: 'a \Rightarrow 'b::linordered\text{-}field$
shows $\forall x.\ f\ x \neq 0 \Longrightarrow O(f * g) = O(f) * O(g)$
apply (*rule equalityI*)
apply (*erule bigo-mult7*)
apply (*rule bigo-mult*)
done

lemma *bigo-minus* [*intro*]: $f \in O(g) \Longrightarrow -\ f \in O(g)$
by (*auto simp add*: *bigo-def fun-Compl-def*)

lemma *bigo-minus2*: $f \in g +o\ O(h) \Longrightarrow -\ f \in -g +o\ O(h)$
apply (*rule set-minus-imp-plus*)
apply (*drule set-plus-imp-minus*)
apply (*drule bigo-minus*)
apply *simp*
done

lemma *bigo-minus3*: $O(-\ f) = O(f)$

by (*auto simp add*: *bigo-def fun-Compl-def*)

lemma *bigo-plus-absorb-lemma1*: $f \in O(g) \Longrightarrow f +o\ O(g) \subseteq O(g)$
proof –
  assume *a*: $f \in O(g)$
  show $f +o\ O(g) \subseteq O(g)$
  proof –
    have $f \in O(f)$ **by** *auto*
    then have $f +o\ O(g) \subseteq O(f) + O(g)$
      by (*auto del*: *subsetI*)
    also have $\ldots \subseteq O(g) + O(g)$
    proof –
      from *a* **have** $O(f) \subseteq O(g)$ **by** (*auto del*: *subsetI*)
      then show *?thesis* **by** (*auto del*: *subsetI*)
    qed
    also have $\ldots \subseteq O(g)$ **by** *simp*
    finally show *?thesis* **.**
  qed
qed

lemma *bigo-plus-absorb-lemma2*: $f \in O(g) \Longrightarrow O(g) \subseteq f +o\ O(g)$
proof –
  assume *a*: $f \in O(g)$
  show $O(g) \subseteq f +o\ O(g)$
  proof –
    from *a* **have** $- f \in O(g)$
      by *auto*
    then have $- f +o\ O(g) \subseteq O(g)$
      by (*elim bigo-plus-absorb-lemma1*)
    then have $f +o\ (- f +o\ O(g)) \subseteq f +o\ O(g)$
      by *auto*
    also have $f +o\ (- f +o\ O(g)) = O(g)$
      by (*simp add*: *set-plus-rearranges*)
    finally show *?thesis* **.**
  qed
qed

lemma *bigo-plus-absorb* [*simp*]: $f \in O(g) \Longrightarrow f +o\ O(g) = O(g)$
  apply (*rule equalityI*)
  apply (*erule bigo-plus-absorb-lemma1*)
  apply (*erule bigo-plus-absorb-lemma2*)
  done

lemma *bigo-plus-absorb2* [*intro*]: $f \in O(g) \Longrightarrow A \subseteq O(g) \Longrightarrow f +o\ A \subseteq O(g)$
  apply (*subgoal-tac* $f +o\ A \subseteq f +o\ O(g)$)
  apply *force+*
  done

lemma *bigo-add-commute-imp*: $f \in g +o\ O(h) \Longrightarrow g \in f +o\ O(h)$

apply (*subst set-minus-plus* [*symmetric*])
apply (*subgoal-tac* $g - f = - (f - g)$)
apply (*erule ssubst*)
apply (*rule bigo-minus*)
apply (*subst set-minus-plus*)
apply *assumption*
apply (*simp add*: *ac-simps*)
done

lemma *bigo-add-commute*: $f \in g +o\ O(h) \longleftrightarrow g \in f +o\ O(h)$
apply (*rule iffI*)
apply (*erule bigo-add-commute-imp*)+
done

lemma *bigo-const1*: $(\lambda x.\ c) \in O(\lambda x.\ 1)$
by (*auto simp add*: *bigo-def ac-simps*)

lemma *bigo-const2* [*intro*]: $O(\lambda x.\ c) \subseteq O(\lambda x.\ 1)$
apply (*rule bigo-elt-subset*)
apply (*rule bigo-const1*)
done

lemma *bigo-const3*:
fixes c :: *'a::linordered-field*
shows $c \neq 0 \Longrightarrow (\lambda x.\ 1) \in O(\lambda x.\ c)$
apply (*simp add*: *bigo-def*)
apply (*rule-tac* $x = |inverse\ c|$ **in** *exI*)
apply (*simp add*: *abs-mult* [*symmetric*])
done

lemma *bigo-const4*:
fixes c :: *'a::linordered-field*
shows $c \neq 0 \Longrightarrow O(\lambda x.\ 1) \subseteq O(\lambda x.\ c)$
apply (*rule bigo-elt-subset*)
apply (*rule bigo-const3*)
apply *assumption*
done

lemma *bigo-const* [*simp*]:
fixes c :: *'a::linordered-field*
shows $c \neq 0 \Longrightarrow O(\lambda x.\ c) = O(\lambda x.\ 1)$
apply (*rule equalityI*)
apply (*rule bigo-const2*)
apply (*rule bigo-const4*)
apply *assumption*
done

lemma *bigo-const-mult1*: $(\lambda x.\ c * f\ x) \in O(f)$
apply (*simp add*: *bigo-def*)

apply (*rule-tac* $x = |c|$ **in** *exI*)
apply (*auto simp add*: *abs-mult* [*symmetric*])
done

lemma *bigo-const-mult2*: $O(\lambda x.\ c * f\ x) \subseteq O(f)$
apply (*rule bigo-elt-subset*)
apply (*rule bigo-const-mult1*)
done

lemma *bigo-const-mult3*:
fixes c :: *'a*::*linordered-field*
shows $c \neq 0 \Longrightarrow f \in O(\lambda x.\ c * f\ x)$
apply (*simp add*: *bigo-def*)
apply (*rule-tac* $x = |inverse\ c|$ **in** *exI*)
apply (*simp add*: *abs-mult mult.assoc* [*symmetric*])
done

lemma *bigo-const-mult4*:
fixes c :: *'a*::*linordered-field*
shows $c \neq 0 \Longrightarrow O(f) \subseteq O(\lambda x.\ c * f\ x)$
apply (*rule bigo-elt-subset*)
apply (*rule bigo-const-mult3*)
apply *assumption*
done

lemma *bigo-const-mult* [*simp*]:
fixes c :: *'a*::*linordered-field*
shows $c \neq 0 \Longrightarrow O(\lambda x.\ c * f\ x) = O(f)$
apply (*rule equalityI*)
apply (*rule bigo-const-mult2*)
apply (*erule bigo-const-mult4*)
done

lemma *bigo-const-mult5* [*simp*]:
fixes c :: *'a*::*linordered-field*
shows $c \neq 0 \Longrightarrow (\lambda x.\ c) *o\ O(f) = O(f)$
apply (*auto del*: *subsetI*)
apply (*rule order-trans*)
apply (*rule bigo-mult2*)
apply (*simp add*: *func-times*)
apply (*auto intro*!: *simp add*: *bigo-def elt-set-times-def func-times*)
apply (*rule-tac* $x = \lambda y.\ inverse\ c * x\ y$ **in** *exI*)
apply (*simp add*: *mult.assoc* [*symmetric*] *abs-mult*)
apply (*rule-tac* $x = |inverse\ c| * ca$ **in** *exI*)
apply *auto*
done

lemma *bigo-const-mult6* [*intro*]: $(\lambda x.\ c) *o\ O(f) \subseteq O(f)$
apply (*auto intro*!: *simp add*: *bigo-def elt-set-times-def func-times*)

apply (*rule-tac* $x = ca * |c|$ **in** *exI*)
apply (*rule allI*)
apply (*subgoal-tac* $ca * |c| * |f\ x| = |c| * (ca * |f\ x|)$)
apply (*erule ssubst*)
apply (*subst abs-mult*)
apply (*rule mult-left-mono*)
apply (*erule spec*)
apply *simp*
apply(*simp add*: *ac-simps*)
done

lemma *bigo-const-mult7* [*intro*]: $f =o\ O(g) \Longrightarrow (\lambda x.\ c * f\ x) =o\ O(g)$
proof −
assume $f =o\ O(g)$
then have $(\lambda x.\ c) * f =o\ (\lambda x.\ c) *o\ O(g)$
by *auto*
also have $(\lambda x.\ c) * f = (\lambda x.\ c * f\ x)$
by (*simp add*: *func-times*)
also have $(\lambda x.\ c) *o\ O(g) \subseteq O(g)$
by (*auto del*: *subsetI*)
finally show *?thesis* **.**
qed

lemma *bigo-compose1*: $f =o\ O(g) \Longrightarrow (\lambda x.\ f\ (k\ x)) =o\ O(\lambda x.\ g\ (k\ x))$
unfolding *bigo-def* **by** *auto*

lemma *bigo-compose2*: $f =o\ g +o\ O(h) \Longrightarrow$
$(\lambda x.\ f\ (k\ x)) =o\ (\lambda x.\ g\ (k\ x)) +o\ O(\lambda x.\ h(k\ x))$
apply (*simp only*: *set-minus-plus* [*symmetric*] *fun-Compl-def func-plus*)
apply (*drule bigo-compose1*)
apply (*simp add*: *fun-diff-def*)
done

4.2 Setsum

lemma *bigo-setsum-main*: $\forall x.\ \forall y \in A\ x.\ 0 \leq h\ x\ y \Longrightarrow$
$\exists c.\ \forall x.\ \forall y \in A\ x.\ |f\ x\ y| \leq c * h\ x\ y \Longrightarrow$
$(\lambda x.\ \sum y \in A\ x.\ f\ x\ y) =o\ O(\lambda x.\ \sum y \in A\ x.\ h\ x\ y)$
apply (*auto simp add*: *bigo-def*)
apply (*rule-tac* $x = |c|$ **in** *exI*)
apply (*subst abs-of-nonneg*) **back back**
apply (*rule setsum-nonneg*)
apply *force*
apply (*subst setsum-right-distrib*)
apply (*rule allI*)
apply (*rule order-trans*)
apply (*rule setsum-abs*)
apply (*rule setsum-mono*)
apply (*rule order-trans*)

apply (*drule spec*)+
apply (*drule bspec*)+
apply *assumption*+
apply (*drule bspec*)
apply *assumption*+
apply (*rule mult-right-mono*)
apply (*rule abs-ge-self*)
apply *force*
done

lemma *bigo-setsum1*: $\forall x\ y.\ 0 \leq h\ x\ y \Longrightarrow$
$\exists c.\ \forall x\ y.\ |f\ x\ y| \leq c * h\ x\ y \Longrightarrow$
$(\lambda x.\ \sum y \in A\ x.\ f\ x\ y) =o\ O(\lambda x.\ \sum y \in A\ x.\ h\ x\ y)$
apply (*rule bigo-setsum-main*)
apply *force*
apply *clarsimp*
apply (*rule-tac* $x = c$ **in** *exI*)
apply *force*
done

lemma *bigo-setsum2*: $\forall y.\ 0 \leq h\ y \Longrightarrow$
$\exists c.\ \forall y.\ |f\ y| \leq c * (h\ y) \Longrightarrow$
$(\lambda x.\ \sum y \in A\ x.\ f\ y) =o\ O(\lambda x.\ \sum y \in A\ x.\ h\ y)$
by (*rule bigo-setsum1*) *auto*

lemma *bigo-setsum3*: $f =o\ O(h) \Longrightarrow$
$(\lambda x.\ \sum y \in A\ x.\ l\ x\ y * f\ (k\ x\ y)) =o\ O(\lambda x.\ \sum y \in A\ x.\ |l\ x\ y * h\ (k\ x\ y)|)$
apply (*rule bigo-setsum1*)
apply (*rule allI*)+
apply (*rule abs-ge-zero*)
apply (*unfold bigo-def*)
apply *auto*
apply (*rule-tac* $x = c$ **in** *exI*)
apply (*rule allI*)+
apply (*subst abs-mult*)+
apply (*subst mult.left-commute*)
apply (*rule mult-left-mono*)
apply (*erule spec*)
apply (*rule abs-ge-zero*)
done

lemma *bigo-setsum4*: $f =o\ g +o\ O(h) \Longrightarrow$
$(\lambda x.\ \sum y \in A\ x.\ l\ x\ y * f\ (k\ x\ y)) =o$
$(\lambda x.\ \sum y \in A\ x.\ l\ x\ y * g\ (k\ x\ y)) +o$
$O(\lambda x.\ \sum y \in A\ x.\ |l\ x\ y * h\ (k\ x\ y)|)$
apply (*rule set-minus-imp-plus*)
apply (*subst fun-diff-def*)
apply (*subst setsum-subtractf* [*symmetric*])
apply (*subst right-diff-distrib* [*symmetric*])

apply (*rule bigo-setsum3*)
apply (*subst fun-diff-def* [*symmetric*])
apply (*erule set-plus-imp-minus*)
done

lemma *bigo-setsum5*: $f =o\ O(h) \Longrightarrow \forall x\ y.\ 0 \leq l\ x\ y \Longrightarrow$
$\forall x.\ 0 \leq h\ x \Longrightarrow$
$(\lambda x. \sum y \in A\ x.\ l\ x\ y * f\ (k\ x\ y)) =o$
$O(\lambda x. \sum y \in A\ x.\ l\ x\ y * h\ (k\ x\ y))$
apply (*subgoal-tac* $(\lambda x. \sum y \in A\ x.\ l\ x\ y * h\ (k\ x\ y)) =$
$(\lambda x. \sum y \in A\ x.\ |l\ x\ y * h\ (k\ x\ y)|)$)
apply (*erule ssubst*)
apply (*erule bigo-setsum3*)
apply (*rule ext*)
apply (*rule setsum.cong*)
apply (*rule refl*)
apply (*subst abs-of-nonneg*)
apply *auto*
done

lemma *bigo-setsum6*: $f =o\ g +o\ O(h) \Longrightarrow \forall x\ y.\ 0 \leq l\ x\ y \Longrightarrow$
$\forall x.\ 0 \leq h\ x \Longrightarrow$
$(\lambda x. \sum y \in A\ x.\ l\ x\ y * f\ (k\ x\ y)) =o$
$(\lambda x. \sum y \in A\ x.\ l\ x\ y * g\ (k\ x\ y)) +o$
$O(\lambda x. \sum y \in A\ x.\ l\ x\ y * h\ (k\ x\ y))$
apply (*rule set-minus-imp-plus*)
apply (*subst fun-diff-def*)
apply (*subst setsum-subtractf* [*symmetric*])
apply (*subst right-diff-distrib* [*symmetric*])
apply (*rule bigo-setsum5*)
apply (*subst fun-diff-def* [*symmetric*])
apply (*drule set-plus-imp-minus*)
apply *auto*
done

4.3 Misc useful stuff

lemma *bigo-useful-intro*: $A \subseteq O(f) \Longrightarrow B \subseteq O(f) \Longrightarrow A + B \subseteq O(f)$
apply (*subst bigo-plus-idemp* [*symmetric*])
apply (*rule set-plus-mono2*)
apply *assumption+*
done

lemma *bigo-useful-add*: $f =o\ O(h) \Longrightarrow g =o\ O(h) \Longrightarrow f + g =o\ O(h)$
apply (*subst bigo-plus-idemp* [*symmetric*])
apply (*rule set-plus-intro*)
apply *assumption+*
done

lemma *bigo-useful-const-mult*:
fixes c :: *'a::linordered-field*
shows $c \neq 0 \Longrightarrow (\lambda x.\ c) * f =o\ O(h) \Longrightarrow f =o\ O(h)$
apply (*rule subsetD*)
apply (*subgoal-tac* $(\lambda x.\ 1\ /\ c) *o\ O(h) \subseteq O(h)$)
apply *assumption*
apply (*rule bigo-const-mult6*)
apply (*subgoal-tac* $f = (\lambda x.\ 1\ /\ c) * ((\lambda x.\ c) * f)$)
apply (*erule ssubst*)
apply (*erule set-times-intro2*)
apply (*simp add*: *func-times*)
done

lemma *bigo-fix*: $(\lambda x{::}nat.\ f\ (x + 1)) =o\ O(\lambda x.\ h\ (x + 1)) \Longrightarrow f\ 0 = 0 \Longrightarrow f =o\ O(h)$
apply (*simp add*: *bigo-alt-def*)
apply *auto*
apply (*rule-tac* $x = c$ **in** *exI*)
apply *auto*
apply (*case-tac* $x = 0$)
apply *simp*
apply (*subgoal-tac* $x = Suc\ (x - 1)$)
apply (*erule ssubst*) **back**
apply (*erule spec*)
apply *simp*
done

lemma *bigo-fix2*:
$(\lambda x.\ f\ ((x{::}nat) + 1)) =o\ (\lambda x.\ g(x + 1)) +o\ O(\lambda x.\ h(x + 1)) \Longrightarrow$
$f\ 0 = g\ 0 \Longrightarrow f =o\ g\ +o\ O(h)$
apply (*rule set-minus-imp-plus*)
apply (*rule bigo-fix*)
apply (*subst fun-diff-def*)
apply (*subst fun-diff-def* [*symmetric*])
apply (*rule set-plus-imp-minus*)
apply *simp*
apply (*simp add*: *fun-diff-def*)
done

4.4 Less than or equal to

definition *lesso* :: $('a \Rightarrow 'b{::}linordered\text{-}idom) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$ (**infixl** $<o$ 70)
where $f <o\ g = (\lambda x.\ max\ (f\ x - g\ x)\ 0)$

lemma *bigo-lesseq1*: $f =o\ O(h) \Longrightarrow \forall x.\ |g\ x| \leq |f\ x| \Longrightarrow g =o\ O(h)$
apply (*unfold bigo-def*)
apply *clarsimp*
apply (*rule-tac* $x = c$ **in** *exI*)

apply (*rule allI*)
apply (*rule order-trans*)
apply (*erule spec*)+
done

lemma *bigo-lesseq2*: $f =o\ O(h) \Longrightarrow \forall x.\ |g\ x| \leq f\ x \Longrightarrow g =o\ O(h)$
apply (*erule bigo-lesseq1*)
apply (*rule allI*)
apply (*drule-tac x = x* **in** *spec*)
apply (*rule order-trans*)
apply *assumption*
apply (*rule abs-ge-self*)
done

lemma *bigo-lesseq3*: $f =o\ O(h) \Longrightarrow \forall x.\ 0 \leq g\ x \Longrightarrow \forall x.\ g\ x \leq f\ x \Longrightarrow g =o\ O(h)$
apply (*erule bigo-lesseq2*)
apply (*rule allI*)
apply (*subst abs-of-nonneg*)
apply (*erule spec*)+
done

lemma *bigo-lesseq4*: $f =o\ O(h) \Longrightarrow$
$\forall x.\ 0 \leq g\ x \Longrightarrow \forall x.\ g\ x \leq |f\ x| \Longrightarrow g =o\ O(h)$
apply (*erule bigo-lesseq1*)
apply (*rule allI*)
apply (*subst abs-of-nonneg*)
apply (*erule spec*)+
done

lemma *bigo-lesso1*: $\forall x.\ f\ x \leq g\ x \Longrightarrow f <o\ g =o\ O(h)$
apply (*unfold lesso-def*)
apply (*subgoal-tac* $(\lambda x.\ max\ (f\ x - g\ x)\ 0) = 0$)
apply (*erule ssubst*)
apply (*rule bigo-zero*)
apply (*unfold func-zero*)
apply (*rule ext*)
apply (*simp split*: *split-max*)
done

lemma *bigo-lesso2*: $f =o\ g +o\ O(h) \Longrightarrow$
$\forall x.\ 0 \leq k\ x \Longrightarrow \forall x.\ k\ x \leq f\ x \Longrightarrow k <o\ g =o\ O(h)$
apply (*unfold lesso-def*)
apply (*rule bigo-lesseq4*)
apply (*erule set-plus-imp-minus*)
apply (*rule allI*)
apply (*rule max.cobounded2*)
apply (*rule allI*)
apply (*subst fun-diff-def*)

apply (*case-tac* $0 \leq k\ x - g\ x$)
apply *simp*
apply (*subst abs-of-nonneg*)
apply (*drule-tac* $x = x$ **in** *spec*) **back**
apply (*simp add*: *algebra-simps*)
apply (*subst diff-conv-add-uminus*)+
apply (*rule add-right-mono*)
apply (*erule spec*)
apply (*rule order-trans*)
prefer *2*
apply (*rule abs-ge-zero*)
apply (*simp add*: *algebra-simps*)
done

lemma *bigo-lesso3*: $f =o\ g +o\ O(h) \Longrightarrow$
$\forall x.\ 0 \leq k\ x \Longrightarrow \forall x.\ g\ x \leq k\ x \Longrightarrow f <o\ k =o\ O(h)$
apply (*unfold lesso-def*)
apply (*rule bigo-lesseq4*)
apply (*erule set-plus-imp-minus*)
apply (*rule allI*)
apply (*rule max.cobounded2*)
apply (*rule allI*)
apply (*subst fun-diff-def*)
apply (*case-tac* $0 \leq f\ x - k\ x$)
apply *simp*
apply (*subst abs-of-nonneg*)
apply (*drule-tac* $x = x$ **in** *spec*) **back**
apply (*simp add*: *algebra-simps*)
apply (*subst diff-conv-add-uminus*)+
apply (*rule add-left-mono*)
apply (*rule le-imp-neg-le*)
apply (*erule spec*)
apply (*rule order-trans*)
prefer *2*
apply (*rule abs-ge-zero*)
apply (*simp add*: *algebra-simps*)
done

lemma *bigo-lesso4*:
fixes $k :: 'a \Rightarrow 'b$::*linordered-field*
shows $f <o\ g =o\ O(k) \Longrightarrow g =o\ h +o\ O(k) \Longrightarrow f <o\ h =o\ O(k)$
apply (*unfold lesso-def*)
apply (*drule set-plus-imp-minus*)
apply (*drule bigo-abs5*) **back**
apply (*simp add*: *fun-diff-def*)
apply (*drule bigo-useful-add*)
apply *assumption*
apply (*erule bigo-lesseq2*) **back**
apply (*rule allI*)

apply (*auto simp add*: *func-plus fun-diff-def algebra-simps split*: *split-max abs-split*)
done

lemma *bigo-lesso5*: $f <o\ g =o\ O(h) \Longrightarrow \exists\, C.\ \forall\, x.\ f\, x \le g\ x + C * |h\ x|$
apply (*simp only*: *lesso-def bigo-alt-def*)
apply *clarsimp*
apply (*rule-tac* $x = c$ **in** *exI*)
apply (*rule allI*)
apply (*drule-tac* $x = x$ **in** *spec*)
apply (*subgoal-tac* $|max\ (f\, x - g\ x)\ 0| = max\ (f\, x - g\ x)\ 0$)
apply (*clarsimp simp add*: *algebra-simps*)
apply (*rule abs-of-nonneg*)
apply (*rule max.cobounded2*)
done

lemma *lesso-add*: $f <o\ g =o\ O(h) \Longrightarrow k <o\ l =o\ O(h) \Longrightarrow (f + k) <o\ (g + l) =o\ O(h)$
apply (*unfold lesso-def*)
apply (*rule bigo-lesseq3*)
apply (*erule bigo-useful-add*)
apply *assumption*
apply (*force split*: *split-max*)
apply (*auto split*: *split-max simp add*: *func-plus*)
done

lemma *bigo-LIMSEQ1*: $f =o\ O(g) \Longrightarrow g \longrightarrow 0 \Longrightarrow f \longrightarrow (0{::}real)$
apply (*simp add*: *LIMSEQ-iff bigo-alt-def*)
apply *clarify*
apply (*drule-tac* $x = r\ /\ c$ **in** *spec*)
apply (*drule mp*)
apply *simp*
apply *clarify*
apply (*rule-tac* $x = no$ **in** *exI*)
apply (*rule allI*)
apply (*drule-tac* $x = n$ **in** *spec*)+
apply (*rule impI*)
apply (*drule mp*)
apply *assumption*
apply (*rule order-le-less-trans*)
apply *assumption*
apply (*rule order-less-le-trans*)
apply (*subgoal-tac* $c * |g\ n| < c * (r\ /\ c)$)
apply *assumption*
apply (*erule mult-strict-left-mono*)
apply *assumption*
apply *simp*
done

lemma *bigo-LIMSEQ2*: $f =o\ g +o\ O(h) \Longrightarrow h \longrightarrow 0 \Longrightarrow f \longrightarrow a \Longrightarrow g$

```
⟶ (a::real)
  apply (drule set-plus-imp-minus)
  apply (drule bigo-LIMSEQ1)
  apply assumption
  apply (simp only: fun-diff-def)
  apply (erule Lim-transform2)
  apply assumption
  done

end
```

5 The Field of Integers mod 2

```
theory Bit
imports Main
begin
```

5.1 Bits as a datatype

```
typedef bit = UNIV :: bool set
  morphisms set Bit
  ..

instantiation bit :: {zero, one}
begin

definition zero-bit-def:
  0 = Bit False

definition one-bit-def:
  1 = Bit True

instance ..

end

old-rep-datatype 0::bit 1::bit
proof −
  fix P and x :: bit
  assume P (0::bit) and P (1::bit)
  then have ∀ b. P (Bit b)
    unfolding zero-bit-def one-bit-def
    by (simp add: all-bool-eq)
  then show P x
    by (induct x) simp
next
  show (0::bit) ≠ (1::bit)
    unfolding zero-bit-def one-bit-def
    by (simp add: Bit-inject)
```

qed

lemma *Bit-set-eq* [*simp*]:
Bit (*set b*) = *b*
by (*fact set-inverse*)

lemma *set-Bit-eq* [*simp*]:
set (*Bit P*) = *P*
by (*rule Bit-inverse*) *rule*

lemma *bit-eq-iff*:
$x = y \longleftrightarrow (set\ x \longleftrightarrow set\ y)$
by (*auto simp add*: *set-inject*)

lemma *Bit-inject* [*simp*]:
$Bit\ P = Bit\ Q \longleftrightarrow (P \longleftrightarrow Q)$
by (*auto simp add*: *Bit-inject*)

lemma *set* [*iff*]:
¬ *set 0*
set 1
by (*simp-all add*: *zero-bit-def one-bit-def Bit-inverse*)

lemma [*code*]:
$set\ 0 \longleftrightarrow False$
$set\ 1 \longleftrightarrow True$
by *simp-all*

lemma *set-iff*:
$set\ b \longleftrightarrow b = 1$
by (*cases b*) *simp-all*

lemma *bit-eq-iff-set*:
$b = 0 \longleftrightarrow \neg\ set\ b$
$b = 1 \longleftrightarrow set\ b$
by (*simp-all add*: *bit-eq-iff*)

lemma *Bit* [*simp*, *code*]:
Bit False = *0*
Bit True = *1*
by (*simp-all add*: *zero-bit-def one-bit-def*)

lemma *bit-not-0-iff* [*iff*]:
$(x::bit) \neq 0 \longleftrightarrow x = 1$
by (*simp add*: *bit-eq-iff*)

lemma *bit-not-1-iff* [*iff*]:
$(x::bit) \neq 1 \longleftrightarrow x = 0$
by (*simp add*: *bit-eq-iff*)

lemma [*code*]:
HOL.equal 0 b $\longleftrightarrow$ $\neg$ *set b*
HOL.equal 1 b $\longleftrightarrow$ *set b*
by (*simp-all add*: *equal set-iff*)

5.2 Type *bit* forms a field

instantiation *bit* :: *field*
begin

definition *plus-bit-def*:
x + *y* = *case-bit y* (*case-bit 1 0 y*) *x*

definition *times-bit-def*:
x ∗ *y* = *case-bit 0 y x*

definition *uminus-bit-def* [*simp*]:
− *x* = (*x* :: *bit*)

definition *minus-bit-def* [*simp*]:
x − *y* = (*x* + *y* :: *bit*)

definition *inverse-bit-def* [*simp*]:
inverse x = (*x* :: *bit*)

definition *divide-bit-def* [*simp*]:
x div y = (*x* ∗ *y* :: *bit*)

lemmas *field-bit-defs* =
plus-bit-def times-bit-def minus-bit-def uminus-bit-def
divide-bit-def inverse-bit-def

instance
by *standard* (*auto simp*: *field-bit-defs split*: *bit.split*)

end

lemma *bit-add-self*: *x* + *x* = (*0* :: *bit*)
unfolding *plus-bit-def* **by** (*simp split*: *bit.split*)

lemma *bit-mult-eq-1-iff* [*simp*]: *x* ∗ *y* = (*1* :: *bit*) $\longleftrightarrow$ *x* = *1* ∧ *y* = *1*
unfolding *times-bit-def* **by** (*simp split*: *bit.split*)

Not sure whether the next two should be simp rules.

lemma *bit-add-eq-0-iff*: *x* + *y* = (*0* :: *bit*) $\longleftrightarrow$ *x* = *y*
unfolding *plus-bit-def* **by** (*simp split*: *bit.split*)

lemma *bit-add-eq-1-iff*: *x* + *y* = (*1* :: *bit*) $\longleftrightarrow$ *x* ≠ *y*
unfolding *plus-bit-def* **by** (*simp split*: *bit.split*)

5.3 Numerals at type *bit*

All numerals reduce to either 0 or 1.

lemma *bit-minus1* [*simp*]: $-$ *1* = (*1* :: *bit*)
  by (*simp only*: *uminus-bit-def*)

lemma *bit-neg-numeral* [*simp*]: ($-$ *numeral w* :: *bit*) = *numeral w*
  by (*simp only*: *uminus-bit-def*)

lemma *bit-numeral-even* [*simp*]: *numeral* (*Num.Bit0 w*) = (*0* :: *bit*)
  by (*simp only*: *numeral-Bit0 bit-add-self*)

lemma *bit-numeral-odd* [*simp*]: *numeral* (*Num.Bit1 w*) = (*1* :: *bit*)
  by (*simp only*: *numeral-Bit1 bit-add-self add-0-left*)

5.4 Conversion from *bit*

context *zero-neq-one*
begin

definition *of-bit* :: *bit* ⇒ *'a*
where
  of-bit b = *case-bit 0 1 b*

lemma *of-bit-eq* [*simp*, *code*]:
  of-bit 0 = *0*
  of-bit 1 = *1*
  by (*simp-all add*: *of-bit-def*)

lemma *of-bit-eq-iff*:
  of-bit x = *of-bit y* ⟷ *x* = *y*
  by (*cases x*) (*cases y*, *simp-all*)+

end

context *semiring-1*
begin

lemma *of-nat-of-bit-eq*:
  of-nat (*of-bit b*) = *of-bit b*
  by (*cases b*) *simp-all*

end

context *ring-1*
begin

lemma *of-int-of-bit-eq*:
  of-int (*of-bit b*) = *of-bit b*

by (*cases b*) *simp-all*

end

hide-const (**open**) *set*

end

6 Axiomatic Declaration of Bounded Natural Functors

theory *BNF-Axiomatization*
imports *Main*
keywords
bnf-axiomatization :: *thy-decl*
begin

ML-file *../Tools/BNF/bnf-axiomatization.ML*

end

7 Generalized Corecursor Sugar (corec and friends)

theory *BNF-Corec*
imports *Main*
keywords
corec :: *thy-decl* **and**
corecursive :: *thy-goal* **and**
friend-of-corec :: *thy-goal* **and**
coinduction-upto :: *thy-decl*
begin

lemma *obj-distinct-prems*: $P \longrightarrow P \longrightarrow Q \Longrightarrow P \Longrightarrow Q$
by *auto*

lemma *inject-refine*: $g\ (f\ x) = x \Longrightarrow g\ (f\ y) = y \Longrightarrow f\ x = f\ y \longleftrightarrow x = y$
by (*metis* (*no-types*))

lemma *convol-apply*: *BNF-Def.convol f g x* = (*f x*, *g x*)
unfolding *convol-def* **..**

lemma *Grp-UNIV-id*: *BNF-Def.Grp UNIV id* = (*op* =)
unfolding *BNF-Def.Grp-def* **by** *auto*

lemma *sum-comp-cases*:
assumes *f o Inl* = *g o Inl* **and** *f o Inr* = *g o Inr*
shows *f* = *g*
proof (*rule ext*)

fix *a* **show** $f\ a = g\ a$
using *assms* **unfolding** *comp-def fun-eq-iff* **by** (*cases a*) *auto*
qed

lemma *case-sum-Inl-Inr-L*: *case-sum* (*f o Inl*) (*f o Inr*) = *f*
by (*metis case-sum-expand-Inr′*)

lemma *eq-o-InrI*: ⟦*g o Inl* = *h*; *case-sum h f* = *g*⟧ ⟹ *f* = *g o Inr*
by (*auto simp*: *fun-eq-iff split*: *sum.splits*)

lemma *id-bnf-o*: *BNF-Composition.id-bnf* ∘ *f* = *f*
unfolding *BNF-Composition.id-bnf-def* **by** (*rule o-def*)

lemma *o-id-bnf*: *f* ∘ *BNF-Composition.id-bnf* = *f*
unfolding *BNF-Composition.id-bnf-def* **by** (*rule o-def*)

lemma *if-True-False*:
(*if P then True else Q*) ⟷ *P* ∨ *Q*
(*if P then False else Q*) ⟷ ¬ *P* ∧ *Q*
(*if P then Q else True*) ⟷ ¬ *P* ∨ *Q*
(*if P then Q else False*) ⟷ *P* ∧ *Q*
by *auto*

lemma *if-distrib-fun*: (*if c then f else g*) *x* = (*if c then f x else g x*)
by *simp*

7.1 Coinduction

lemma *eq-comp-compI*: *a o b* = *f o x* ⟹ *x o c* = *id* ⟹ *f* = *a o* (*b o c*)
unfolding *fun-eq-iff* **by** *simp*

lemma *self-bounded-weaken-left*: (*a* :: *′a* :: *semilattice-inf*) ≤ *inf a b* ⟹ *a* ≤ *b*
by (*erule le-infE*)

lemma *self-bounded-weaken-right*: (*a* :: *′a* :: *semilattice-inf*) ≤ *inf b a* ⟹ *a* ≤ *b*
by (*erule le-infE*)

lemma *symp-iff*: *symp R* ⟷ *R* = *R*ˆ−−*1*
by (*metis antisym conversep.cases conversep-le-swap predicate2I symp-def*)

lemma *equivp-inf*: ⟦*equivp R*; *equivp S*⟧ ⟹ *equivp* (*inf R S*)
unfolding *equivp-def inf-fun-def inf-bool-def* **by** *metis*

lemma *vimage2p-rel-prod*:
(*λx y. rel-prod R S* (*BNF-Def.convol f1 g1 x*) (*BNF-Def.convol f2 g2 y*)) =
(*inf* (*BNF-Def.vimage2p f1 f2 R*) (*BNF-Def.vimage2p g1 g2 S*))
unfolding *vimage2p-def rel-prod.simps convol-def* **by** *auto*

lemma *predicate2I-obj*: (∀ *x y*. *P x y* ⟶ *Q x y*) ⟹ *P* ≤ *Q*

by *auto*

lemma *predicate2D-obj*: $P \leq Q \Longrightarrow P\ x\ y \longrightarrow Q\ x\ y$
by *auto*

locale *cong* =
fixes *rel* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('b \Rightarrow 'b \Rightarrow bool)$
and *eval* :: $'b \Rightarrow 'a$
and *retr* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a \Rightarrow bool)$
assumes *rel-mono*: $\bigwedge R\ S.\ R \leq S \Longrightarrow rel\ R \leq rel\ S$
and *equivp-retr*: $\bigwedge R.\ equivp\ R \Longrightarrow equivp\ (retr\ R)$
and *retr-eval*: $\bigwedge R\ x\ y.\ [\![(rel\text{-}fun\ (rel\ R)\ R)\ eval\ eval;\ rel\ (inf\ R\ (retr\ R))\ x\ y]\!]$
$\Longrightarrow$
retr R (*eval x*) (*eval y*)
begin

definition *cong* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$ **where**
$cong\ R \equiv equivp\ R \wedge (rel\text{-}fun\ (rel\ R)\ R)\ eval\ eval$

lemma *cong-retr*: $cong\ R \Longrightarrow cong\ (inf\ R\ (retr\ R))$
unfolding *cong-def*
by (*auto simp*: *rel-fun-def dest*: *predicate2D*[*OF rel-mono*, *rotated*]
intro: *equivp-inf equivp-retr retr-eval*)

lemma *cong-equivp*: $cong\ R \Longrightarrow equivp\ R$
unfolding *cong-def* **by** *simp*

definition *gen-cong* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$ **where**
$gen\text{-}cong\ R\ j1\ j2 \equiv \forall R'.\ R \leq R' \wedge cong\ R' \longrightarrow R'\ j1\ j2$

lemma *gen-cong-reflp*[*intro*, *simp*]: $x = y \Longrightarrow gen\text{-}cong\ R\ x\ y$
unfolding *gen-cong-def* **by** (*auto dest*: *cong-equivp equivp-reflp*)

lemma *gen-cong-symp*[*intro*]: $gen\text{-}cong\ R\ x\ y \Longrightarrow gen\text{-}cong\ R\ y\ x$
unfolding *gen-cong-def* **by** (*auto dest*: *cong-equivp equivp-symp*)

lemma *gen-cong-transp*[*intro*]: $gen\text{-}cong\ R\ x\ y \Longrightarrow gen\text{-}cong\ R\ y\ z \Longrightarrow gen\text{-}cong$
$R\ x\ z$
unfolding *gen-cong-def* **by** (*auto dest*: *cong-equivp equivp-transp*)

lemma *equivp-gen-cong*: *equivp* (*gen-cong R*)
by (*intro equivpI reflpI sympI transpI*) *auto*

lemma *leq-gen-cong*: $R \leq gen\text{-}cong\ R$
unfolding *gen-cong-def*[*abs-def*] **by** *auto*

lemmas *imp-gen-cong*[*intro*] = *predicate2D*[*OF leq-gen-cong*]

lemma *gen-cong-minimal*: $[\![R \leq R';\ cong\ R']\!] \Longrightarrow gen\text{-}cong\ R \leq R'$

unfolding *gen-cong-def*[*abs-def*] **by** (*rule predicate2I*) *metis*

lemma *congdd-base-gen-congdd-base-aux*:
rel (*gen-cong R*) *x y* $\Longrightarrow$ $R \leq R'$ $\Longrightarrow$ *cong* R' $\Longrightarrow$ R' (*eval x*) (*eval y*)
by (*force simp*: *rel-fun-def gen-cong-def cong-def dest*: *spec*[*of - R'*] *predicate2D*[*OF rel-mono*, *rotated* -1, *of - - - R'*])

lemma *cong-gen-cong*: *cong* (*gen-cong R*)
proof $-$
{ **fix** R' *x y*
have *rel* (*gen-cong R*) *x y* $\Longrightarrow$ $R \leq R'$ $\Longrightarrow$ *cong* R' $\Longrightarrow$ R' (*eval x*) (*eval y*)
by (*force simp*: *rel-fun-def gen-cong-def cong-def dest*: *spec*[*of - R'*]
predicate2D[*OF rel-mono*, *rotated* -1, *of - - - R'*])
}
then show *cong* (*gen-cong R*) **by** (*auto simp*: *equivp-gen-cong rel-fun-def gen-cong-def cong-def*)
qed

lemma *gen-cong-eval-rel-fun*:
(*rel-fun* (*rel* (*gen-cong R*)) (*gen-cong R*)) *eval eval*
using *cong-gen-cong*[*of R*] **unfolding** *cong-def* **by** *simp*

lemma *gen-cong-eval*:
rel (*gen-cong R*) *x y* $\Longrightarrow$ *gen-cong R* (*eval x*) (*eval y*)
by (*erule rel-funD*[*OF gen-cong-eval-rel-fun*])

lemma *gen-cong-idem*: *gen-cong* (*gen-cong R*) = *gen-cong R*
by (*simp add*: *antisym cong-gen-cong gen-cong-minimal leq-gen-cong*)

lemma *gen-cong-rho*:
ϱ = *eval o f* $\Longrightarrow$ *rel* (*gen-cong R*) (*f x*) (*f y*) $\Longrightarrow$ *gen-cong R* (ϱ *x*) (ϱ *y*)
by (*simp add*: *gen-cong-eval*)
lemma *coinduction*:
assumes *coind*: $\forall R$. $R \leq$ *retr R* $\longrightarrow$ $R \leq$ *op* =
assumes *cih*: $R \leq$ *retr* (*gen-cong R*)
shows $R \leq$ *op* =
apply (*rule order-trans*[*OF leq-gen-cong mp*[*OF spec*[*OF coind*]]])
apply (*rule self-bounded-weaken-left*[*OF gen-cong-minimal*])
apply (*rule inf-greatest*[*OF leq-gen-cong cih*])
apply (*rule cong-retr*[*OF cong-gen-cong*])
done

end

lemma *rel-sum-case-sum*:
rel-fun (*rel-sum R S*) *T* (*case-sum f1 g1*) (*case-sum f2 g2*) = (*rel-fun R T f1 f2* $\wedge$ *rel-fun S T g1 g2*)
by (*auto simp*: *rel-fun-def rel-sum.simps split*: *sum.splits*)

context
  fixes *rel eval rel′ eval′ retr emb*
  assumes *base*: *cong rel eval retr*
  and *step*: *cong rel′ eval′ retr*
  and *emb*: *eval′ o emb* = *eval*
  and *emb-transfer*: *rel-fun* (*rel R*) (*rel′ R*) *emb emb*
begin

interpretation *base*: *cong rel eval retr* **by** (*rule base*)
interpretation *step*: *cong rel′ eval′ retr* **by** (*rule step*)

lemma *gen-cong-emb*: *base.gen-cong R* ≤ *step.gen-cong R*
proof (*rule base.gen-cong-minimal*[*OF step.leq-gen-cong*])
  note *step.gen-cong-eval-rel-fun*[*transfer-rule*] *emb-transfer*[*transfer-rule*]
  have (*rel-fun* (*rel* (*step.gen-cong R*)) (*step.gen-cong R*)) *eval eval*
    unfolding *emb*[*symmetric*] **by** *transfer-prover*
  then show *base.cong* (*step.gen-cong R*)
    by (*auto simp*: *base.cong-def step.equivp-gen-cong*)
qed

end

ML-file ../*Tools*/*BNF*/*bnf-gfp-grec-tactics.ML*
ML-file ../*Tools*/*BNF*/*bnf-gfp-grec.ML*
ML-file ../*Tools*/*BNF*/*bnf-gfp-grec-sugar-util.ML*
ML-file ../*Tools*/*BNF*/*bnf-gfp-grec-sugar-tactics.ML*
ML-file ../*Tools*/*BNF*/*bnf-gfp-grec-sugar.ML*
ML-file ../*Tools*/*BNF*/*bnf-gfp-grec-unique-sugar.ML*

method-setup *corec-unique* = ‹
 Scan.succeed (*SIMPLE-METHOD′ o BNF-GFP-Grec-Unique-Sugar.corec-unique-tac*)
› *prove uniqueness of corecursive equation*

end

8 Boolean Algebras

theory *Boolean-Algebra*
imports *Main*
begin

locale *boolean* =
  fixes *conj* :: ′*a* ⇒ ′*a* ⇒ ′*a* (**infixr** ⊓ *70*)
  fixes *disj* :: ′*a* ⇒ ′*a* ⇒ ′*a* (**infixr** ⊔ *65*)
  fixes *compl* :: ′*a* ⇒ ′*a* (∼ - [*81*] *80*)
  fixes *zero* :: ′*a* (**0**)
  fixes *one* :: ′*a* (**1**)
  assumes *conj-assoc*: $(x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$
  assumes *disj-assoc*: $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$

assumes *conj-commute*: $x \sqcap y = y \sqcap x$
assumes *disj-commute*: $x \sqcup y = y \sqcup x$
assumes *conj-disj-distrib*: $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$
assumes *disj-conj-distrib*: $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$
assumes *conj-one-right* [*simp*]: $x \sqcap \mathbf{1} = x$
assumes *disj-zero-right* [*simp*]: $x \sqcup \mathbf{0} = x$
assumes *conj-cancel-right* [*simp*]: $x \sqcap \sim x = \mathbf{0}$
assumes *disj-cancel-right* [*simp*]: $x \sqcup \sim x = \mathbf{1}$
begin

sublocale *conj*: *abel-semigroup conj*
by *standard* (*fact conj-assoc conj-commute*)+

sublocale *disj*: *abel-semigroup disj*
by *standard* (*fact disj-assoc disj-commute*)+

lemmas *conj-left-commute* = *conj.left-commute*

lemmas *disj-left-commute* = *disj.left-commute*

lemmas *conj-ac* = *conj.assoc conj.commute conj.left-commute*
lemmas *disj-ac* = *disj.assoc disj.commute disj.left-commute*

lemma *dual*: *boolean disj conj compl one zero*
apply (*rule boolean.intro*)
apply (*rule disj-assoc*)
apply (*rule conj-assoc*)
apply (*rule disj-commute*)
apply (*rule conj-commute*)
apply (*rule disj-conj-distrib*)
apply (*rule conj-disj-distrib*)
apply (*rule disj-zero-right*)
apply (*rule conj-one-right*)
apply (*rule disj-cancel-right*)
apply (*rule conj-cancel-right*)
done

8.1 Complement

lemma *complement-unique*:
assumes *1*: $a \sqcap x = \mathbf{0}$
assumes *2*: $a \sqcup x = \mathbf{1}$
assumes *3*: $a \sqcap y = \mathbf{0}$
assumes *4*: $a \sqcup y = \mathbf{1}$
shows $x = y$
proof −
have $(a \sqcap x) \sqcup (x \sqcap y) = (a \sqcap y) \sqcup (x \sqcap y)$ **using** *1 3* **by** *simp*
hence $(x \sqcap a) \sqcup (x \sqcap y) = (y \sqcap a) \sqcup (y \sqcap x)$ **using** *conj-commute* **by** *simp*
hence $x \sqcap (a \sqcup y) = y \sqcap (a \sqcup x)$ **using** *conj-disj-distrib* **by** *simp*

hence $x \sqcap \mathbf{1} = y \sqcap \mathbf{1}$ **using** *2 4* **by** *simp*
thus $x = y$ **using** *conj-one-right* **by** *simp*
qed

lemma *compl-unique*: $[\![x \sqcap y = \mathbf{0}; x \sqcup y = \mathbf{1}]\!] \Longrightarrow \sim x = y$
by (*rule complement-unique* [*OF conj-cancel-right disj-cancel-right*])

lemma *double-compl* [*simp*]: $\sim (\sim x) = x$
proof (*rule compl-unique*)
from *conj-cancel-right* **show** $\sim x \sqcap x = \mathbf{0}$ **by** (*simp only*: *conj-commute*)
from *disj-cancel-right* **show** $\sim x \sqcup x = \mathbf{1}$ **by** (*simp only*: *disj-commute*)
qed

lemma *compl-eq-compl-iff* [*simp*]: $(\sim x = \sim y) = (x = y)$
by (*rule inj-eq* [*OF inj-on-inverseI*], *rule double-compl*)

8.2 Conjunction

lemma *conj-absorb* [*simp*]: $x \sqcap x = x$
proof −
have $x \sqcap x = (x \sqcap x) \sqcup \mathbf{0}$ **using** *disj-zero-right* **by** *simp*
also have ... $= (x \sqcap x) \sqcup (x \sqcap \sim x)$ **using** *conj-cancel-right* **by** *simp*
also have ... $= x \sqcap (x \sqcup \sim x)$ **using** *conj-disj-distrib* **by** (*simp only*:)
also have ... $= x \sqcap \mathbf{1}$ **using** *disj-cancel-right* **by** *simp*
also have ... $= x$ **using** *conj-one-right* **by** *simp*
finally show *?thesis* **.**
qed

lemma *conj-zero-right* [*simp*]: $x \sqcap \mathbf{0} = \mathbf{0}$
proof −
have $x \sqcap \mathbf{0} = x \sqcap (x \sqcap \sim x)$ **using** *conj-cancel-right* **by** *simp*
also have ... $= (x \sqcap x) \sqcap \sim x$ **using** *conj-assoc* **by** (*simp only*:)
also have ... $= x \sqcap \sim x$ **using** *conj-absorb* **by** *simp*
also have ... $= \mathbf{0}$ **using** *conj-cancel-right* **by** *simp*
finally show *?thesis* **.**
qed

lemma *compl-one* [*simp*]: $\sim \mathbf{1} = \mathbf{0}$
by (*rule compl-unique* [*OF conj-zero-right disj-zero-right*])

lemma *conj-zero-left* [*simp*]: $\mathbf{0} \sqcap x = \mathbf{0}$
by (*subst conj-commute*) (*rule conj-zero-right*)

lemma *conj-one-left* [*simp*]: $\mathbf{1} \sqcap x = x$
by (*subst conj-commute*) (*rule conj-one-right*)

lemma *conj-cancel-left* [*simp*]: $\sim x \sqcap x = \mathbf{0}$
by (*subst conj-commute*) (*rule conj-cancel-right*)

lemma *conj-left-absorb* [*simp*]: $x \sqcap (x \sqcap y) = x \sqcap y$
by (*simp only*: *conj-assoc* [*symmetric*] *conj-absorb*)

lemma *conj-disj-distrib2*:
$(y \sqcup z) \sqcap x = (y \sqcap x) \sqcup (z \sqcap x)$
by (*simp only*: *conj-commute conj-disj-distrib*)

lemmas *conj-disj-distribs* =
conj-disj-distrib conj-disj-distrib2

8.3 Disjunction

lemma *disj-absorb* [*simp*]: $x \sqcup x = x$
by (*rule boolean.conj-absorb* [*OF dual*])

lemma *disj-one-right* [*simp*]: $x \sqcup \mathbf{1} = \mathbf{1}$
by (*rule boolean.conj-zero-right* [*OF dual*])

lemma *compl-zero* [*simp*]: $\sim \mathbf{0} = \mathbf{1}$
by (*rule boolean.compl-one* [*OF dual*])

lemma *disj-zero-left* [*simp*]: $\mathbf{0} \sqcup x = x$
by (*rule boolean.conj-one-left* [*OF dual*])

lemma *disj-one-left* [*simp*]: $\mathbf{1} \sqcup x = \mathbf{1}$
by (*rule boolean.conj-zero-left* [*OF dual*])

lemma *disj-cancel-left* [*simp*]: $\sim x \sqcup x = \mathbf{1}$
by (*rule boolean.conj-cancel-left* [*OF dual*])

lemma *disj-left-absorb* [*simp*]: $x \sqcup (x \sqcup y) = x \sqcup y$
by (*rule boolean.conj-left-absorb* [*OF dual*])

lemma *disj-conj-distrib2*:
$(y \sqcap z) \sqcup x = (y \sqcup x) \sqcap (z \sqcup x)$
by (*rule boolean.conj-disj-distrib2* [*OF dual*])

lemmas *disj-conj-distribs* =
disj-conj-distrib disj-conj-distrib2

8.4 De Morgan's Laws

lemma *de-Morgan-conj* [*simp*]: $\sim (x \sqcap y) = \sim x \sqcup \sim y$
proof (*rule compl-unique*)
have $(x \sqcap y) \sqcap (\sim x \sqcup \sim y) = ((x \sqcap y) \sqcap \sim x) \sqcup ((x \sqcap y) \sqcap \sim y)$
by (*rule conj-disj-distrib*)
also have ... $= (y \sqcap (x \sqcap \sim x)) \sqcup (x \sqcap (y \sqcap \sim y))$
by (*simp only*: *conj-ac*)
finally show $(x \sqcap y) \sqcap (\sim x \sqcup \sim y) = \mathbf{0}$
by (*simp only*: *conj-cancel-right conj-zero-right disj-zero-right*)

next
  have $(x \sqcap y) \sqcup (\sim x \sqcup \sim y) = (x \sqcup (\sim x \sqcup \sim y)) \sqcap (y \sqcup (\sim x \sqcup \sim y))$
    by (*rule disj-conj-distrib2*)
  also have ... $= (\sim y \sqcup (x \sqcup \sim x)) \sqcap (\sim x \sqcup (y \sqcup \sim y))$
    by (*simp only*: *disj-ac*)
  finally show $(x \sqcap y) \sqcup (\sim x \sqcup \sim y) = \mathbf{1}$
    by (*simp only*: *disj-cancel-right disj-one-right conj-one-right*)
qed

lemma *de-Morgan-disj* [*simp*]: $\sim (x \sqcup y) = \sim x \sqcap \sim y$
by (*rule boolean.de-Morgan-conj* [*OF dual*])

end

8.5 Symmetric Difference

locale *boolean-xor* = *boolean* +
  fixes *xor* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixr** $\oplus$ *65*)
  assumes *xor-def*: $x \oplus y = (x \sqcap \sim y) \sqcup (\sim x \sqcap y)$
begin

sublocale *xor*: *abel-semigroup xor*
proof
  fix $x\ y\ z :: 'a$
  let $?t = (x \sqcap y \sqcap z) \sqcup (x \sqcap \sim y \sqcap \sim z) \sqcup$
      $(\sim x \sqcap y \sqcap \sim z) \sqcup (\sim x \sqcap \sim y \sqcap z)$
  have $?t \sqcup (z \sqcap x \sqcap \sim x) \sqcup (z \sqcap y \sqcap \sim y) =$
      $?t \sqcup (x \sqcap y \sqcap \sim y) \sqcup (x \sqcap z \sqcap \sim z)$
    by (*simp only*: *conj-cancel-right conj-zero-right*)
  thus $(x \oplus y) \oplus z = x \oplus (y \oplus z)$
    apply (*simp only*: *xor-def de-Morgan-disj de-Morgan-conj double-compl*)
    apply (*simp only*: *conj-disj-distribs conj-ac disj-ac*)
    done
  show $x \oplus y = y \oplus x$
    by (*simp only*: *xor-def conj-commute disj-commute*)
qed

lemmas *xor-assoc* = *xor.assoc*
lemmas *xor-commute* = *xor.commute*
lemmas *xor-left-commute* = *xor.left-commute*

lemmas *xor-ac* = *xor.assoc xor.commute xor.left-commute*

lemma *xor-def2*:
  $x \oplus y = (x \sqcup y) \sqcap (\sim x \sqcup \sim y)$
by (*simp only*: *xor-def conj-disj-distribs*
      disj-ac conj-ac conj-cancel-right disj-zero-left)

lemma *xor-zero-right* [*simp*]: $x \oplus \mathbf{0} = x$

by (*simp only*: *xor-def compl-zero conj-one-right conj-zero-right disj-zero-right*)

lemma *xor-zero-left* [*simp*]: $\mathbf{0} \oplus x = x$
by (*subst xor-commute*) (*rule xor-zero-right*)

lemma *xor-one-right* [*simp*]: $x \oplus \mathbf{1} = \sim x$
by (*simp only*: *xor-def compl-one conj-zero-right conj-one-right disj-zero-left*)

lemma *xor-one-left* [*simp*]: $\mathbf{1} \oplus x = \sim x$
by (*subst xor-commute*) (*rule xor-one-right*)

lemma *xor-self* [*simp*]: $x \oplus x = \mathbf{0}$
by (*simp only*: *xor-def conj-cancel-right conj-cancel-left disj-zero-right*)

lemma *xor-left-self* [*simp*]: $x \oplus (x \oplus y) = y$
by (*simp only*: *xor-assoc* [*symmetric*] *xor-self xor-zero-left*)

lemma *xor-compl-left* [*simp*]: $\sim x \oplus y = \sim (x \oplus y)$
apply (*simp only*: *xor-def de-Morgan-disj de-Morgan-conj double-compl*)
apply (*simp only*: *conj-disj-distribs*)
apply (*simp only*: *conj-cancel-right conj-cancel-left*)
apply (*simp only*: *disj-zero-left disj-zero-right*)
apply (*simp only*: *disj-ac conj-ac*)
done

lemma *xor-compl-right* [*simp*]: $x \oplus \sim y = \sim (x \oplus y)$
apply (*simp only*: *xor-def de-Morgan-disj de-Morgan-conj double-compl*)
apply (*simp only*: *conj-disj-distribs*)
apply (*simp only*: *conj-cancel-right conj-cancel-left*)
apply (*simp only*: *disj-zero-left disj-zero-right*)
apply (*simp only*: *disj-ac conj-ac*)
done

lemma *xor-cancel-right*: $x \oplus \sim x = \mathbf{1}$
by (*simp only*: *xor-compl-right xor-self compl-zero*)

lemma *xor-cancel-left*: $\sim x \oplus x = \mathbf{1}$
by (*simp only*: *xor-compl-left xor-self compl-zero*)

lemma *conj-xor-distrib*: $x \sqcap (y \oplus z) = (x \sqcap y) \oplus (x \sqcap z)$
proof −
  have $(x \sqcap y \sqcap \sim z) \sqcup (x \sqcap \sim y \sqcap z) =$
    $(y \sqcap x \sqcap \sim x) \sqcup (z \sqcap x \sqcap \sim x) \sqcup (x \sqcap y \sqcap \sim z) \sqcup (x \sqcap \sim y \sqcap z)$
   by (*simp only*: *conj-cancel-right conj-zero-right disj-zero-left*)
  thus $x \sqcap (y \oplus z) = (x \sqcap y) \oplus (x \sqcap z)$
   by (*simp* (*no-asm-use*) *only*:
    xor-def de-Morgan-disj de-Morgan-conj double-compl
    conj-disj-distribs conj-ac disj-ac)
qed

lemma *conj-xor-distrib2*: $(y \oplus z) \sqcap x = (y \sqcap x) \oplus (z \sqcap x)$
proof −
  have $x \sqcap (y \oplus z) = (x \sqcap y) \oplus (x \sqcap z)$
    by (*rule conj-xor-distrib*)
  thus $(y \oplus z) \sqcap x = (y \sqcap x) \oplus (z \sqcap x)$
    by (*simp only*: *conj-commute*)
qed

lemmas *conj-xor-distribs* = *conj-xor-distrib conj-xor-distrib2*

end

end

9 The Bourbaki-Witt tower construction for transfinite iteration

theory *Bourbaki-Witt-Fixpoint* **imports** *Main* **begin**

lemma *ChainsI* [*intro?*]:
  $(\bigwedge a\ b.\ [\![\ a \in Y;\ b \in Y\ ]\!] \Longrightarrow (a,\ b) \in r \lor (b,\ a) \in r) \Longrightarrow Y \in \mathit{Chains}\ r$
unfolding *Chains-def* **by** *blast*

lemma *in-Chains-subset*: $[\![\ M \in \mathit{Chains}\ r;\ M' \subseteq M\ ]\!] \Longrightarrow M' \in \mathit{Chains}\ r$
by(*auto simp add*: *Chains-def*)

lemma *FieldI1*: $(i,\ j) \in R \Longrightarrow i \in \mathit{Field}\ R$
  unfolding *Field-def* **by** *auto*

lemma *Chains-FieldD*: $[\![\ M \in \mathit{Chains}\ r;\ x \in M\ ]\!] \Longrightarrow x \in \mathit{Field}\ r$
by(*auto simp add*: *Chains-def intro*: *FieldI1 FieldI2*)

lemma *in-Chains-conv-chain*: $M \in \mathit{Chains}\ r \longleftrightarrow$ *Complete-Partial-Order.chain* $(\lambda x\ y.\ (x,\ y) \in r)\ M$
by(*simp add*: *Chains-def chain-def*)

lemma *partial-order-on-trans*:
  $[\![$ *partial-order-on A r*; $(x,\ y) \in r;\ (y,\ z) \in r\ ]\!] \Longrightarrow (x,\ z) \in r$
by(*auto simp add*: *order-on-defs dest*: *transD*)

locale *bourbaki-witt-fixpoint* =
  fixes *lub* :: *'a set ⇒ 'a*
  and *leq* :: *('a × 'a) set*
  and *f* :: *'a ⇒ 'a*
  assumes *po*: *Partial-order leq*
  and *lub-least*: $[\![\ M \in \mathit{Chains}\ leq;\ M \neq \{\};\ \bigwedge x.\ x \in M \Longrightarrow (x,\ z) \in leq\ ]\!] \Longrightarrow (\mathit{lub}\ M,\ z) \in leq$

and *lub-upper*: ⟦ $M \in$ *Chains leq*; $x \in M$ ⟧ $\Longrightarrow$ (x, *lub M*) $\in$ *leq*
and *lub-in-Field*: ⟦ $M \in$ *Chains leq*; $M \neq \{\}$ ⟧ $\Longrightarrow$ *lub M* $\in$ *Field leq*
and *increasing*: $\bigwedge x$. $x \in$ *Field leq* $\Longrightarrow$ (x, $f\ x$) $\in$ *leq*
begin

lemma *leq-trans*: ⟦ (x, y) $\in$ *leq*; (y, z) $\in$ *leq* ⟧ $\Longrightarrow$ (x, z) $\in$ *leq*
by(*rule partial-order-on-trans*[*OF po*])

lemma *leq-refl*: $x \in$ *Field leq* $\Longrightarrow$ (x, x) $\in$ *leq*
using *po* **by**(*simp add*: *order-on-defs refl-on-def*)

lemma *leq-antisym*: ⟦ (x, y) $\in$ *leq*; (y, x) $\in$ *leq* ⟧ $\Longrightarrow$ $x = y$
using *po* **by**(*simp add*: *order-on-defs antisym-def*)

inductive-set *iterates-above* :: $'a \Rightarrow 'a$ *set*
for *a*
where
base: $a \in$ *iterates-above a*
| *step*: $x \in$ *iterates-above a* $\Longrightarrow$ $f\ x \in$ *iterates-above a*
| *Sup*: ⟦ $M \in$ *Chains leq*; $M \neq \{\}$; $\bigwedge x$. $x \in M \Longrightarrow x \in$ *iterates-above a* ⟧ $\Longrightarrow$ *lub M* $\in$ *iterates-above a*

definition *fixp-above* :: $'a \Rightarrow 'a$
where *fixp-above a* = (*if a* $\in$ *Field leq then lub* (*iterates-above a*) *else a*)

lemma *fixp-above-outside*: $a \notin$ *Field leq* $\Longrightarrow$ *fixp-above a* = *a*
by(*simp add*: *fixp-above-def*)

lemma *fixp-above-inside*: $a \in$ *Field leq* $\Longrightarrow$ *fixp-above a* = *lub* (*iterates-above a*)
by(*simp add*: *fixp-above-def*)

context
notes *leq-refl* [*intro*!, *simp*]
and *base* [*intro*]
and *step* [*intro*]
and *Sup* [*intro*]
and *leq-trans* [*trans*]
begin

lemma *iterates-above-le-f*: ⟦ $x \in$ *iterates-above a*; $a \in$ *Field leq* ⟧ $\Longrightarrow$ (x, $f\ x$) $\in$ *leq*
by(*induction x rule*: *iterates-above.induct*)(*blast intro*: *increasing FieldI2 lub-in-Field*)+

lemma *iterates-above-Field*: ⟦ $x \in$ *iterates-above a*; $a \in$ *Field leq* ⟧ $\Longrightarrow$ $x \in$ *Field leq*
by(*drule* (*1*) *iterates-above-le-f*)(*rule FieldI1*)

lemma *iterates-above-ge*:
assumes *y*: $y \in$ *iterates-above a*

and *a*: $a \in$ *Field leq*
shows $(a, y) \in$ *leq*
using *y* **by**(*induction*)(*auto intro*: *a increasing iterates-above-le-f leq-trans leq-trans*[*OF - lub-upper*])

lemma *iterates-above-lub*:
assumes *M*: $M \in$ *Chains leq*
and *nempty*: $M \neq \{\}$
and *upper*: $\bigwedge y.\ y \in M \Longrightarrow \exists z \in M.\ (y, z) \in$ *leq* $\wedge z \in$ *iterates-above a*
shows *lub M* $\in$ *iterates-above a*
proof −
let *?M* = *M* $\cap$ *iterates-above a*
from *M* **have** *M'*: *?M* $\in$ *Chains leq* **by**(*rule in-Chains-subset*)*simp*
have *?M* $\neq \{\}$ **using** *nempty* **by**(*auto dest*: *upper*)
with *M'* **have** *lub ?M* $\in$ *iterates-above a* **by**(*rule Sup*) *blast*
also have *lub ?M* = *lub M* **using** *nempty*
by(*intro leq-antisym*)(*blast intro*!: *lub-least*[*OF M*] *lub-least*[*OF M'*] *intro*: *lub-upper*[*OF M'*] *lub-upper*[*OF M*] *leq-trans dest*: *upper*)+
finally show *?thesis* **.**
qed

lemma *iterates-above-successor*:
assumes *y*: $y \in$ *iterates-above a*
and *a*: $a \in$ *Field leq*
shows $y = a \vee y \in$ *iterates-above* (*f a*)
using *y*
proof *induction*
case *base* **thus** *?case* **by** *simp*
next
case (*step x*) **thus** *?case* **by** *auto*
next
case (*Sup M*)
show *?case*
proof(*cases* $\exists x.\ M \subseteq \{x\}$)
case *True*
with ‹$M \neq \{\}$› **obtain** *y* **where** *M*: $M = \{y\}$ **by** *auto*
have *lub M* = *y*
by(*rule leq-antisym*)(*auto intro*!: *lub-upper Sup lub-least ChainsI simp add*: *a M Sup.hyps*(*3*)[*of y*, *THEN iterates-above-Field*] *dest*: *iterates-above-Field*)
with *Sup.IH*[*of y*] *M* **show** *?thesis* **by** *simp*
next
case *False*
from *Sup*(*1*−*2*) **have** *lub M* $\in$ *iterates-above* (*f a*)
proof(*rule iterates-above-lub*)
fix *y*
assume *y*: $y \in M$
from *Sup.IH*[*OF this*] **show** $\exists z \in M.\ (y, z) \in$ *leq* $\wedge z \in$ *iterates-above* (*f a*)
proof
assume $y = a$

```
      from y False obtain z where z: z ∈ M and neq: y ≠ z by (metis insertI1
subsetI)
       with Sup.IH[OF z] ‹y = a› Sup.hyps(3)[OF z]
       show ?thesis by(auto dest: iterates-above-ge intro: a)
     next
       assume y ∈ iterates-above (f a)
       moreover with increasing[OF a] have y ∈ Field leq
         by(auto dest!: iterates-above-Field intro: FieldI2)
       ultimately show ?thesis using y by(auto)
     qed
   qed
   thus ?thesis by simp
  qed
qed

lemma iterates-above-Sup-aux:
  assumes M: M ∈ Chains leq M ≠ {}
  and M': M' ∈ Chains leq M' ≠ {}
  and comp: ⋀x. x ∈ M ⟹ x ∈ iterates-above (lub M') ∨ lub M' ∈ iterates-above
x
  shows (lub M, lub M') ∈ leq ∨ lub M ∈ iterates-above (lub M')
proof(cases ∃ x ∈ M. x ∈ iterates-above (lub M'))
  case True
  then obtain x where x: x ∈ M x ∈ iterates-above (lub M') by blast
  have lub-M': lub M' ∈ Field leq using M' by(rule lub-in-Field)
  have lub M ∈ iterates-above (lub M') using M
  proof(rule iterates-above-lub)
    fix y
    assume y: y ∈ M
    from comp[OF y] show ∃ z∈M. (y, z) ∈ leq ∧ z ∈ iterates-above (lub M')
    proof
      assume y ∈ iterates-above (lub M')
      from this iterates-above-Field[OF this] y lub-M' show ?thesis by blast
    next
      assume lub M' ∈ iterates-above y
     hence (y, lub M') ∈ leq using Chains-FieldD[OF M(1) y] by(rule iterates-above-ge)
      also have (lub M', x) ∈ leq using x(2) lub-M' by(rule iterates-above-ge)
      finally show ?thesis using x by blast
    qed
  qed
  thus ?thesis ..
next
  case False
  have (lub M, lub M') ∈ leq using M
  proof(rule lub-least)
    fix x
    assume x: x ∈ M
    from comp[OF x] x False have lub M' ∈ iterates-above x by auto
    moreover from M(1) x have x ∈ Field leq by(rule Chains-FieldD)
```

ultimately show (*x*, *lub M′*) ∈ *leq* **by**(*rule iterates-above-ge*)
qed
thus *?thesis* **..**
qed

lemma *iterates-above-triangle*:
assumes *x*: *x* ∈ *iterates-above a*
and *y*: *y* ∈ *iterates-above a*
and *a*: *a* ∈ *Field leq*
shows *x* ∈ *iterates-above y* ∨ *y* ∈ *iterates-above x*
using *x y*
proof(*induction arbitrary*: *y*)
case *base* **then show** *?case* **by** *simp*
next
case (*step x*) **thus** *?case* **using** *a*
by(*auto dest*: *iterates-above-successor intro*: *iterates-above-Field*)
next
case *x*: (*Sup M*)
hence *lub*: *lub M* ∈ *iterates-above a* **by** *blast*
from ‹*y* ∈ *iterates-above a*› **show** *?case*
proof(*induction*)
case *base* **show** *?case* **using** *lub* **by** *simp*
next
case (*step y*) **thus** *?case* **using** *a*
by(*auto dest*: *iterates-above-successor intro*: *iterates-above-Field*)
next
case *y*: (*Sup M′*)
hence *lub′*: *lub M′* ∈ *iterates-above a* **by** *blast*
have ∗: *x* ∈ *iterates-above* (*lub M′*) ∨ *lub M′* ∈ *iterates-above x* **if** *x* ∈ *M* **for** *x*
using *that lub′* **by**(*rule x.IH*)
with *x*(*1*−*2*) *y*(*1*−*2*) **have** (*lub M*, *lub M′*) ∈ *leq* ∨ *lub M* ∈ *iterates-above* (*lub M′*)
by(*rule iterates-above-Sup-aux*)
moreover from *y*(*1*−*2*) *x*(*1*−*2*) **have** (*lub M′*, *lub M*) ∈ *leq* ∨ *lub M′* ∈ *iterates-above* (*lub M*)
by(*rule iterates-above-Sup-aux*)(*blast dest*: *y.IH*)
ultimately show *?case* **by**(*auto 4 3 dest*: *leq-antisym*)
qed
qed

lemma *chain-iterates-above*:
assumes *a*: *a* ∈ *Field leq*
shows *iterates-above a* ∈ *Chains leq* (**is** *?C* ∈ -)
proof (*rule ChainsI*)
fix *x y*
assume *x* ∈ *?C* *y* ∈ *?C*
hence *x* ∈ *iterates-above y* ∨ *y* ∈ *iterates-above x* **using** *a* **by**(*rule iterates-above-triangle*)
moreover from ‹*x* ∈ *?C*› *a* **have** *x* ∈ *Field leq* **by**(*rule iterates-above-Field*)
moreover from ‹*y* ∈ *?C*› *a* **have** *y* ∈ *Field leq* **by**(*rule iterates-above-Field*)

ultimately show $(x, y) \in leq \vee (y, x) \in leq$ **by**(*auto dest*: *iterates-above-ge*)
qed

lemma *fixp-iterates-above*: *fixp-above a* $\in$ *iterates-above a*
by(*auto intro*: *chain-iterates-above simp add*: *fixp-above-def*)

lemma *fixp-above-Field*: $a \in$ *Field leq* $\Longrightarrow$ *fixp-above a* $\in$ *Field leq*
using *fixp-iterates-above* **by**(*rule iterates-above-Field*)

lemma *fixp-above-unfold*:
assumes *a*: $a \in$ *Field leq*
shows *fixp-above a* = *f* (*fixp-above a*) (**is** *?a* = *f ?a*)
proof(*rule leq-antisym*)
show (*?a*, *f ?a*) $\in$ *leq* **using** *fixp-above-Field*[*OF a*] **by**(*rule increasing*)

have *f ?a* $\in$ *iterates-above a* **using** *fixp-iterates-above* **by**(*rule iterates-above.step*)
with *chain-iterates-above*[*OF a*] **show** (*f ?a*, *?a*) $\in$ *leq*
by(*simp add*: *fixp-above-inside assms lub-upper*)
qed

end

lemma *fixp-induct* [*case-names adm base step*]:
assumes *adm*: *ccpo.admissible lub* ($\lambda x\ y$. $(x, y) \in leq$) *P*
and *base*: *P a*
and *step*: $\bigwedge x$. *P x* $\Longrightarrow$ *P* (*f x*)
shows *P* (*fixp-above a*)
proof(*cases a* $\in$ *Field leq*)
case *True*
from *adm chain-iterates-above*[*OF True*]
show *?thesis* **unfolding** *fixp-above-inside*[*OF True*] *in-Chains-conv-chain*
proof(*rule ccpo.admissibleD*)
have $a \in$ *iterates-above a* **..**
then show *iterates-above a* $\neq$ {} **by**(*auto*)
show *P x* **if** $x \in$ *iterates-above a* **for** *x* **using** *that*
by *induction*(*auto intro*: *base step simp add*: *in-Chains-conv-chain dest*: *ccpo.admissibleD*[*OF adm*])
qed
qed(*simp add*: *fixp-above-outside base*)

end

end

10 Order on characters

theory *Char-ord*
imports *Main*
begin

instantiation *char* :: *linorder*
begin

definition $c1 \leq c2 \longleftrightarrow$ *nat-of-char c1* $\leq$ *nat-of-char c2*
definition $c1 < c2 \longleftrightarrow$ *nat-of-char c1* $<$ *nat-of-char c2*

instance
  by *standard* (*auto simp add*: *less-eq-char-def less-char-def*)

end

lemma *less-eq-char-simps*:
  $(0 :: char) \leq c$
  Char k $\leq 0 \longleftrightarrow$ *numeral k mod 256* $= (0 :: nat)$
  Char k $\leq$ *Char l* $\longleftrightarrow$ *numeral k mod 256* $\leq$ (*numeral l mod 256* :: *nat*)
  by (*simp-all add*: *Char-def less-eq-char-def*)

lemma *less-char-simps*:
  $\neg\ c < (0 :: char)$
  $0 <$ *Char k* $\longleftrightarrow (0 :: nat) <$ *numeral k mod 256*
  Char k $<$ *Char l* $\longleftrightarrow$ *numeral k mod 256* $<$ (*numeral l mod 256* :: *nat*)
  by (*simp-all add*: *Char-def less-char-def*)

instantiation *char* :: *distrib-lattice*
begin

definition (*inf* :: *char* $\Rightarrow$ -) = *min*
definition (*sup* :: *char* $\Rightarrow$ -) = *max*

instance
  by *standard* (*auto simp add*: *inf-char-def sup-char-def max-min-distrib2*)

end

instantiation *String.literal* :: *linorder*
begin

context includes *literal.lifting* **begin**
lift-definition *less-literal* :: *String.literal* $\Rightarrow$ *String.literal* $\Rightarrow$ *bool* **is** *ord.lexordp op* $<$ **.**
lift-definition *less-eq-literal* :: *String.literal* $\Rightarrow$ *String.literal* $\Rightarrow$ *bool* **is** *ord.lexordp-eq op* $<$ **.**

instance
proof −
  interpret *linorder ord.lexordp-eq op* $<$ *ord.lexordp op* $<$ :: *string* $\Rightarrow$ *string* $\Rightarrow$ *bool*
    by(*rule linorder.lexordp-linorder*[**where** *less-eq*=*op* $\leq$])(*unfold-locales*)

show *PROP ?thesis*
by(*intro-classes*)(*transfer*, *simp add*: *less-le-not-le linear*)+
qed

end
end

lemma *less-literal-code* [*code*]:
op < = (λ*xs ys*. *ord*.*lexordp op* < (*String*.*explode xs*) (*String*.*explode ys*))
by(*simp add*: *less-literal*.*rep-eq fun-eq-iff*)

lemma *less-eq-literal-code* [*code*]:
op ≤ = (λ*xs ys*. *ord*.*lexordp-eq op* < (*String*.*explode xs*) (*String*.*explode ys*))
by(*simp add*: *less-eq-literal*.*rep-eq fun-eq-iff*)

lifting-update *literal*.*lifting*
lifting-forget *literal*.*lifting*

end

theory *Code-Test*
imports *Main*
keywords *test-code* :: *diag*
begin

10.1 YXML encoding for *term*

datatype (*plugins del*: *code size quickcheck*) *yxml-of-term* = *YXML*

lemma *yot-anything*: *x* = (*y* :: *yxml-of-term*)
by(*cases x y rule*: *yxml-of-term*.*exhaust*[*case-product yxml-of-term*.*exhaust*])(*simp*)

definition *yot-empty* :: *yxml-of-term* **where** [*code del*]: *yot-empty* = *YXML*
definition *yot-literal* :: *String*.*literal* ⇒ *yxml-of-term*
where [*code del*]: *yot-literal* - = *YXML*
definition *yot-append* :: *yxml-of-term* ⇒ *yxml-of-term* ⇒ *yxml-of-term*
where [*code del*]: *yot-append* - - = *YXML*
definition *yot-concat* :: *yxml-of-term list* ⇒ *yxml-of-term*
where [*code del*]: *yot-concat* - = *YXML*

Serialise *yxml-of-term* to native string of target language

code-printing type-constructor *yxml-of-term*
⇀ (*SML*) *string*
and (*OCaml*) *string*
and (*Haskell*) *String*
and (*Scala*) *String*
| **constant** *yot-empty*
⇀ (*SML*)
and (*OCaml*)

and (*Haskell*)
and (*Scala*)
| **constant** *yot-literal*
⇀ (*SML*) -
and (*OCaml*) -
and (*Haskell*) -
and (*Scala*) -
| **constant** *yot-append*
⇀ (*SML*) *String.concat* [(-), (-)]
and (*OCaml*) *String.concat* [(-); (-)]
and (*Haskell*) **infixr** *5* ++
and (*Scala*) **infixl** *5* +
| **constant** *yot-concat*
⇀ (*SML*) *String.concat*
and (*OCaml*) *String.concat*
and (*Haskell*) *Prelude.concat*
and (*Scala*) -.*mkString*()

Stripped-down implementations of Isabelle's XML tree with YXML encoding as defined in `~~/src/Pure/PIDE/xml.ML`, `~~/src/Pure/PIDE/yxml.ML` sufficient to encode *term* as in `~~/src/Pure/term_xml.ML`.

datatype (*plugins del*: *code size quickcheck*) *xml-tree* = *XML-Tree*

lemma *xml-tree-anything*: *x* = (*y* :: *xml-tree*)
by(*cases x y rule*: *xml-tree.exhaust*[*case-product xml-tree.exhaust*])(*simp*)

context begin
local-setup ‹*Local-Theory.map-background-naming* (*Name-Space.mandatory-path xml*)›

type-synonym *attributes* = (*String.literal* × *String.literal*) *list*
type-synonym *body* = *xml-tree list*

definition *Elem* :: *String.literal* ⇒ *attributes* ⇒ *xml-tree list* ⇒ *xml-tree*
where [*code del*]: *Elem* - - - = *XML-Tree*

definition *Text* :: *String.literal* ⇒ *xml-tree*
where [*code del*]: *Text* - = *XML-Tree*

definition *node* :: *xml-tree list* ⇒ *xml-tree*
where *node ts* = *Elem* (*STR* '':'') [] *ts*

definition *tagged* :: *String.literal* ⇒ *String.literal option* ⇒ *xml-tree list* ⇒ *xml-tree*
where *tagged tag x ts* = *Elem tag* (*case x of None* ⇒ [] | *Some x'* ⇒ [(*STR* ''*0*'', *x'*)]) *ts*

definition *list* **where** *list f xs* = *map* (*node* ∘ *f*) *xs*

definition *X* :: *yxml-of-term* **where** *X* = *yot-literal* (*STR* [*Char* (*num.Bit1* (*num.Bit0*

num.One))])
definition *Y* :: *yxml-of-term* **where** *Y* = *yot-literal* (*STR* [*Char* (*num.Bit0* (*num.Bit1* *num.One*))])
definition *XY* :: *yxml-of-term* **where** *XY* = *yot-append X Y*
definition *XYX* :: *yxml-of-term* **where** *XYX* = *yot-append XY X*

end

code-datatype *xml.Elem xml.Text*

definition *yxml-string-of-xml-tree* :: *xml-tree* ⇒ *yxml-of-term* ⇒ *yxml-of-term*
where [*code del*]: *yxml-string-of-xml-tree* - - = *YXML*

lemma *yxml-string-of-xml-tree-code* [*code*]:
  yxml-string-of-xml-tree (*xml.Elem name atts ts*) *rest* =
   yot-append xml.XY (
   yot-append (*yot-literal name*) (
   foldr (λ(*a*, *x*) *rest*.
     yot-append xml.Y (
     yot-append (*yot-literal a*) (
     yot-append (*yot-literal* (*STR* ''='')) (
     yot-append (*yot-literal x*) *rest*)))) *atts* (
   foldr yxml-string-of-xml-tree ts (
   yot-append xml.XYX rest))))
  yxml-string-of-xml-tree (*xml.Text s*) *rest* = *yot-append* (*yot-literal s*) *rest*
by(*rule yot-anything*)+

definition *yxml-string-of-body* :: *xml.body* ⇒ *yxml-of-term*
where *yxml-string-of-body ts* = *foldr yxml-string-of-xml-tree ts yot-empty*

Encoding *term* into XML trees as defined in `~~/src/Pure/term_xml.ML`

definition *xml-of-typ* :: *Typerep.typerep* ⇒ *xml.body*
where [*code del*]: *xml-of-typ* - = [*XML-Tree*]

definition *xml-of-term* :: *Code-Evaluation.term* ⇒ *xml.body*
where [*code del*]: *xml-of-term* - = [*XML-Tree*]

lemma *xml-of-typ-code* [*code*]:
  xml-of-typ (*typerep.Typerep t args*) = [*xml.tagged* (*STR* ''0'') (*Some t*) (*xml.list* *xml-of-typ args*)]
by(*simp add*: *xml-of-typ-def xml-tree-anything*)

lemma *xml-of-term-code* [*code*]:
  xml-of-term (*Code-Evaluation.Const x ty*) = [*xml.tagged* (*STR* ''0'') (*Some x*) (*xml-of-typ ty*)]
  xml-of-term (*Code-Evaluation.App t1 t2*) = [*xml.tagged* (*STR* ''5'') *None* [*xml.node* (*xml-of-term t1*), *xml.node* (*xml-of-term t2*)]]
  xml-of-term (*Code-Evaluation.Abs x ty t*) = [*xml.tagged* (*STR* ''4'') (*Some x*) [*xml.node* (*xml-of-typ ty*), *xml.node* (*xml-of-term t*)]]

— FIXME: *Code-Evaluation.Free* is used only in *Quickcheck-Narrowing* to represent uninstantiated parameters in constructors. Here, we always translate them to `Free` variables.
xml-of-term (*Code-Evaluation.Free x ty*) = [*xml.tagged* (*STR ''1''*) (*Some x*) (*xml-of-typ ty*)]
by(*simp-all add*: *xml-of-term-def xml-tree-anything*)

definition *yxml-string-of-term* :: *Code-Evaluation.term* ⇒ *yxml-of-term*
where *yxml-string-of-term* = *yxml-string-of-body* ∘ *xml-of-term*

10.2 Test engine and drivers

ML-file *code-test.ML*

end

11 Old Datatype package: constructing datatypes from Cartesian Products and Disjoint Sums

theory *Old-Datatype*
imports *../Main*
keywords *old-datatype* :: *thy-decl*
begin

ML-file *~~/src/HOL/Tools/datatype-realizer.ML*

11.1 The datatype universe

definition *Node* = {*p. EX f x k. p* = (*f* :: *nat* => *'b* + *nat*, *x* ::*'a* + *nat*) & *f k* = *Inr 0*}

typedef (*'a*, *'b*) *node* = *Node* :: ((*nat* => *'b* + *nat*) ∗ (*'a* + *nat*)) *set*
morphisms *Rep-Node Abs-Node*
unfolding *Node-def* **by** *auto*

Datatypes will be represented by sets of type *node*

type-synonym *'a item* = (*'a*, *unit*) *node set*
type-synonym (*'a*, *'b*) *dtree* = (*'a*, *'b*) *node set*

definition *Push* :: [(*'b* + *nat*), *nat* => (*'b* + *nat*)] => (*nat* => (*'b* + *nat*))

where *Push* == (%*b h. case-nat b h*)

definition *Push-Node* :: [(*'b* + *nat*), (*'a*, *'b*) *node*] => (*'a*, *'b*) *node*
where *Push-Node* == (%*n x. Abs-Node* (*apfst* (*Push n*) (*Rep-Node x*)))

definition *Atom* :: ('a + nat) => ('a, 'b) dtree
  where *Atom* == (%x. {Abs-Node((%k. Inr 0, x))})
definition *Scons* :: [('a, 'b) dtree, ('a, 'b) dtree] => ('a, 'b) dtree
  where *Scons* M N == (Push-Node (Inr 1) ‘ M) Un (Push-Node (Inr (Suc 1)) ‘ N)

definition *Leaf* :: 'a => ('a, 'b) dtree
  where *Leaf* == Atom o Inl
definition *Numb* :: nat => ('a, 'b) dtree
  where *Numb* == Atom o Inr

definition *In0* :: ('a, 'b) dtree => ('a, 'b) dtree
  where *In0*(M) == Scons (Numb 0) M
definition *In1* :: ('a, 'b) dtree => ('a, 'b) dtree
  where *In1*(M) == Scons (Numb 1) M

definition *Lim* :: ('b => ('a, 'b) dtree) => ('a, 'b) dtree
  where *Lim* f == ⋃{z. ? x. z = Push-Node (Inl x) ‘ (f x)}

definition *ndepth* :: ('a, 'b) node => nat
  where *ndepth*(n) == (%(f,x). LEAST k. f k = Inr 0) (Rep-Node n)
definition *ntrunc* :: [nat, ('a, 'b) dtree] => ('a, 'b) dtree
  where *ntrunc* k N == {n. n:N & ndepth(n)<k}

definition *uprod* :: [('a, 'b) dtree set, ('a, 'b) dtree set]=> ('a, 'b) dtree set
  where *uprod* A B == UN x:A. UN y:B. { Scons x y }
definition *usum* :: [('a, 'b) dtree set, ('a, 'b) dtree set]=> ('a, 'b) dtree set
  where *usum* A B == In0‘A Un In1‘B

definition *Split* :: [[('a, 'b) dtree, ('a, 'b) dtree]=>'c, ('a, 'b) dtree] => 'c
  where *Split* c M == THE u. EX x y. M = Scons x y & u = c x y

definition *Case* :: [[('a, 'b) dtree]=>'c, [('a, 'b) dtree]=>'c, ('a, 'b) dtree] => 'c
  where *Case* c d M == THE u. (EX x . M = In0(x) & u = c(x)) | (EX y . M = In1(y) & u = d(y))

definition *dprod* :: [(('a, 'b) dtree * ('a, 'b) dtree)set, (('a, 'b) dtree * ('a, 'b) dtree)set]
    => (('a, 'b) dtree * ('a, 'b) dtree)set

where *dprod r s* == *UN* (*x*,*x'*):*r*. *UN* (*y*,*y'*):*s*. {(*Scons x y*, *Scons x' y'*)}

definition *dsum* :: [(('*a*, '*b*) *dtree* ∗ ('*a*, '*b*) *dtree*)*set*, (('*a*, '*b*) *dtree* ∗ ('*a*, '*b*) *dtree*)*set*]
=> (('*a*, '*b*) *dtree* ∗ ('*a*, '*b*) *dtree*)*set*
where *dsum r s* == (*UN* (*x*,*x'*):*r*. {(*In0*(*x*),*In0*(*x'*))}) *Un* (*UN* (*y*,*y'*):*s*. {(*In1*(*y*),*In1*(*y'*))})

lemma *apfst-convE*:
[| *q* = *apfst f p*; !!*x y*. [| *p* = (*x*,*y*); *q* = (*f*(*x*),*y*) |] ==> *R*
|] ==> *R*
by (*force simp add*: *apfst-def*)

lemma *Push-inject1*: *Push i f* = *Push j g* ==> *i*=*j*
apply (*simp add*: *Push-def fun-eq-iff*)
apply (*drule-tac x*=*0* **in** *spec*, *simp*)
done

lemma *Push-inject2*: *Push i f* = *Push j g* ==> *f*=*g*
apply (*auto simp add*: *Push-def fun-eq-iff*)
apply (*drule-tac x*=*Suc x* **in** *spec*, *simp*)
done

lemma *Push-inject*:
[| *Push i f* =*Push j g*; [| *i*=*j*; *f*=*g* |] ==> *P* |] ==> *P*
by (*blast dest*: *Push-inject1 Push-inject2*)

lemma *Push-neq-K0*: *Push* (*Inr* (*Suc k*)) *f* = (%*z*. *Inr 0*) ==> *P*
by (*auto simp add*: *Push-def fun-eq-iff split*: *nat.split-asm*)

lemmas *Abs-Node-inj* = *Abs-Node-inject* [*THEN* [*2*] *rev-iffD1*]

lemma *Node-K0-I*: (%*k*. *Inr 0*, *a*) : *Node*
by (*simp add*: *Node-def*)

lemma *Node-Push-I*: *p*: *Node* ==> *apfst* (*Push i*) *p* : *Node*
apply (*simp add*: *Node-def Push-def*)
apply (*fast intro*!: *apfst-conv nat.case*(*2*)[*THEN trans*])
done

11.2 Freeness: Distinctness of Constructors

lemma *Scons-not-Atom* [*iff*]: *Scons M N* ≠ *Atom*(*a*)
unfolding *Atom-def Scons-def Push-Node-def One-nat-def*

by (*blast intro*: *Node-K0-I Rep-Node* [*THEN Node-Push-I*]
dest!: *Abs-Node-inj*
elim!: *apfst-convE sym* [*THEN Push-neq-K0*])

lemmas *Atom-not-Scons* [*iff*] = *Scons-not-Atom* [*THEN not-sym*]

lemma *inj-Atom*: *inj*(*Atom*)
apply (*simp add*: *Atom-def*)
apply (*blast intro*!: *inj-onI Node-K0-I dest*!: *Abs-Node-inj*)
done
lemmas *Atom-inject* = *inj-Atom* [*THEN injD*]

lemma *Atom-Atom-eq* [*iff*]: (*Atom*(*a*)=*Atom*(*b*)) = (*a*=*b*)
by (*blast dest*!: *Atom-inject*)

lemma *inj-Leaf*: *inj*(*Leaf*)
apply (*simp add*: *Leaf-def o-def*)
apply (*rule inj-onI*)
apply (*erule Atom-inject* [*THEN Inl-inject*])
done

lemmas *Leaf-inject* [*dest*!] = *inj-Leaf* [*THEN injD*]

lemma *inj-Numb*: *inj*(*Numb*)
apply (*simp add*: *Numb-def o-def*)
apply (*rule inj-onI*)
apply (*erule Atom-inject* [*THEN Inr-inject*])
done

lemmas *Numb-inject* [*dest*!] = *inj-Numb* [*THEN injD*]

lemma *Push-Node-inject*:
[| *Push-Node i m* =*Push-Node j n*; [| *i*=*j*; *m*=*n* |] ==> *P*
|] ==> *P*
apply (*simp add*: *Push-Node-def*)
apply (*erule Abs-Node-inj* [*THEN apfst-convE*])
apply (*rule Rep-Node* [*THEN Node-Push-I*])+
apply (*erule sym* [*THEN apfst-convE*])
apply (*blast intro*: *Rep-Node-inject* [*THEN iffD1*] *trans sym elim*!: *Push-inject*)
done

lemma *Scons-inject-lemma1*: *Scons M N* <= *Scons M′ N′* ==> *M*<=*M′*
unfolding *Scons-def One-nat-def*
by (*blast dest*!: *Push-Node-inject*)

lemma *Scons-inject-lemma2*: *Scons M N* <= *Scons M′ N′* ==> *N*<=*N′*
unfolding *Scons-def One-nat-def*
by (*blast dest*!: *Push-Node-inject*)

lemma *Scons-inject1*: *Scons M N* = *Scons M′ N′* ==> *M*=*M′*
apply (*erule equalityE*)
apply (*iprover intro*: *equalityI Scons-inject-lemma1*)
done

lemma *Scons-inject2*: *Scons M N* = *Scons M′ N′* ==> *N*=*N′*
apply (*erule equalityE*)
apply (*iprover intro*: *equalityI Scons-inject-lemma2*)
done

lemma *Scons-inject*:
  [| *Scons M N* = *Scons M′ N′*; [| *M*=*M′*; *N*=*N′* |] ==> *P* |] ==> *P*
by (*iprover dest*: *Scons-inject1 Scons-inject2*)

lemma *Scons-Scons-eq* [*iff*]: (*Scons M N* = *Scons M′ N′*) = (*M*=*M′* & *N*=*N′*)
by (*blast elim*!: *Scons-inject*)

lemma *Scons-not-Leaf* [*iff*]: *Scons M N* ≠ *Leaf*(*a*)
unfolding *Leaf-def o-def* **by** (*rule Scons-not-Atom*)

lemmas *Leaf-not-Scons* [*iff*] = *Scons-not-Leaf* [*THEN not-sym*]

lemma *Scons-not-Numb* [*iff*]: *Scons M N* ≠ *Numb*(*k*)
unfolding *Numb-def o-def* **by** (*rule Scons-not-Atom*)

lemmas *Numb-not-Scons* [*iff*] = *Scons-not-Numb* [*THEN not-sym*]

lemma *Leaf-not-Numb* [*iff*]: *Leaf*(*a*) ≠ *Numb*(*k*)
by (*simp add*: *Leaf-def Numb-def*)

lemmas *Numb-not-Leaf* [*iff*] = *Leaf-not-Numb* [*THEN not-sym*]

lemma *ndepth-K0*: *ndepth* (*Abs-Node*(%*k*. *Inr 0*, *x*)) = *0*
by (*simp add*: *ndepth-def Node-K0-I* [*THEN Abs-Node-inverse*] *Least-equality*)

lemma *ndepth-Push-Node-aux*:
  case-nat (*Inr* (*Suc i*)) *f k* = *Inr 0* −−> *Suc*(*LEAST x*. *f x* = *Inr 0*) <= *k*
apply (*induct-tac k*, *auto*)
apply (*erule Least-le*)
done

lemma *ndepth-Push-Node*:
  ndepth (*Push-Node* (*Inr* (*Suc i*)) *n*) = *Suc*(*ndepth*(*n*))
apply (*insert Rep-Node* [*of n*, *unfolded Node-def*])
apply (*auto simp add*: *ndepth-def Push-Node-def*
    Rep-Node [*THEN Node-Push-I*, *THEN Abs-Node-inverse*])
apply (*rule Least-equality*)
apply (*auto simp add*: *Push-def ndepth-Push-Node-aux*)
apply (*erule LeastI*)
done

lemma *ntrunc-0* [*simp*]: *ntrunc 0 M* = {}
by (*simp add*: *ntrunc-def*)

lemma *ntrunc-Atom* [*simp*]: *ntrunc* (*Suc k*) (*Atom a*) = *Atom*(*a*)
by (*auto simp add*: *Atom-def ntrunc-def ndepth-K0*)

lemma *ntrunc-Leaf* [*simp*]: *ntrunc* (*Suc k*) (*Leaf a*) = *Leaf*(*a*)
unfolding *Leaf-def o-def* **by** (*rule ntrunc-Atom*)

lemma *ntrunc-Numb* [*simp*]: *ntrunc* (*Suc k*) (*Numb i*) = *Numb*(*i*)
unfolding *Numb-def o-def* **by** (*rule ntrunc-Atom*)

lemma *ntrunc-Scons* [*simp*]:
  ntrunc (*Suc k*) (*Scons M N*) = *Scons* (*ntrunc k M*) (*ntrunc k N*)
unfolding *Scons-def ntrunc-def One-nat-def*
by (*auto simp add*: *ndepth-Push-Node*)

lemma *ntrunc-one-In0* [*simp*]: *ntrunc* (*Suc 0*) (*In0 M*) = {}
apply (*simp add*: *In0-def*)
apply (*simp add*: *Scons-def*)
done

lemma *ntrunc-In0* [*simp*]: *ntrunc* (*Suc*(*Suc k*)) (*In0 M*) = *In0* (*ntrunc* (*Suc k*) *M*)
by (*simp add*: *In0-def*)

lemma *ntrunc-one-In1* [*simp*]: *ntrunc* (*Suc 0*) (*In1 M*) = {}
apply (*simp add*: *In1-def*)
apply (*simp add*: *Scons-def*)
done

lemma *ntrunc-In1* [*simp*]: *ntrunc* (*Suc*(*Suc k*)) (*In1 M*) = *In1* (*ntrunc* (*Suc k*) *M*)
by (*simp add*: *In1-def*)

11.3 Set Constructions

lemma *uprodI* [*intro*!]: [| *M*:*A*; *N*:*B* |] ==> *Scons M N* : *uprod A B*
by (*simp add*: *uprod-def*)

lemma *uprodE* [*elim*!]:
  [| *c* : *uprod A B*;
    !!*x y*. [| *x*:*A*; *y*:*B*; *c* = *Scons x y* |] ==> *P*
  |] ==> *P*
by (*auto simp add*: *uprod-def*)

lemma *uprodE2*: [| *Scons M N* : *uprod A B*; [| *M*:*A*; *N*:*B* |] ==> *P* |] ==> *P*
by (*auto simp add*: *uprod-def*)

lemma *usum-In0I* [*intro*]: *M*:*A* ==> *In0*(*M*) : *usum A B*
by (*simp add*: *usum-def*)

lemma *usum-In1I* [*intro*]: *N*:*B* ==> *In1*(*N*) : *usum A B*
by (*simp add*: *usum-def*)

lemma *usumE* [*elim*!]:
  [| *u* : *usum A B*;
    !!*x*. [| *x*:*A*; *u*=*In0*(*x*) |] ==> *P*;
    !!*y*. [| *y*:*B*; *u*=*In1*(*y*) |] ==> *P*
  |] ==> *P*

by (*auto simp add*: *usum-def*)

lemma *In0-not-In1* [*iff*]: *In0*(*M*) ≠ *In1*(*N*)
unfolding *In0-def In1-def One-nat-def* **by** *auto*

lemmas *In1-not-In0* [*iff*] = *In0-not-In1* [*THEN not-sym*]

lemma *In0-inject*: *In0*(*M*) = *In0*(*N*) ==> *M*=*N*
by (*simp add*: *In0-def*)

lemma *In1-inject*: *In1*(*M*) = *In1*(*N*) ==> *M*=*N*
by (*simp add*: *In1-def*)

lemma *In0-eq* [*iff*]: (*In0 M* = *In0 N*) = (*M*=*N*)
by (*blast dest*!: *In0-inject*)

lemma *In1-eq* [*iff*]: (*In1 M* = *In1 N*) = (*M*=*N*)
by (*blast dest*!: *In1-inject*)

lemma *inj-In0*: *inj In0*
by (*blast intro*!: *inj-onI*)

lemma *inj-In1*: *inj In1*
by (*blast intro*!: *inj-onI*)

lemma *Lim-inject*: *Lim f* = *Lim g* ==> *f* = *g*
apply (*simp add*: *Lim-def*)
apply (*rule ext*)
apply (*blast elim*!: *Push-Node-inject*)
done

lemma *ntrunc-subsetI*: *ntrunc k M* <= *M*
by (*auto simp add*: *ntrunc-def*)

lemma *ntrunc-subsetD*: (!!*k*. *ntrunc k M* <= *N*) ==> *M*<=*N*
by (*auto simp add*: *ntrunc-def*)

lemma *ntrunc-equality*: (!!*k*. *ntrunc k M* = *ntrunc k N*) ==> *M*=*N*
apply (*rule equalityI*)

apply (*rule-tac* [!] *ntrunc-subsetD*)
apply (*rule-tac* [!] *ntrunc-subsetI* [*THEN* [*2*] *subset-trans*], *auto*)
done

lemma *ntrunc-o-equality*:
[| !!*k*. (*ntrunc*(*k*) *o h1*) = (*ntrunc*(*k*) *o h2*) |] ==> *h1*=*h2*
apply (*rule ntrunc-equality* [*THEN ext*])
apply (*simp add*: *fun-eq-iff*)
done

lemma *uprod-mono*: [| *A*<=*A'*; *B*<=*B'* |] ==> *uprod A B* <= *uprod A' B'*
by (*simp add*: *uprod-def*, *blast*)

lemma *usum-mono*: [| *A*<=*A'*; *B*<=*B'* |] ==> *usum A B* <= *usum A' B'*
by (*simp add*: *usum-def*, *blast*)

lemma *Scons-mono*: [| *M*<=*M'*; *N*<=*N'* |] ==> *Scons M N* <= *Scons M' N'*
by (*simp add*: *Scons-def*, *blast*)

lemma *In0-mono*: *M*<=*N* ==> *In0*(*M*) <= *In0*(*N*)
by (*simp add*: *In0-def Scons-mono*)

lemma *In1-mono*: *M*<=*N* ==> *In1*(*M*) <= *In1*(*N*)
by (*simp add*: *In1-def Scons-mono*)

lemma *Split* [*simp*]: *Split c* (*Scons M N*) = *c M N*
by (*simp add*: *Split-def*)

lemma *Case-In0* [*simp*]: *Case c d* (*In0 M*) = *c*(*M*)
by (*simp add*: *Case-def*)

lemma *Case-In1* [*simp*]: *Case c d* (*In1 N*) = *d*(*N*)
by (*simp add*: *Case-def*)

lemma *ntrunc-UN1*: *ntrunc k* (*UN x*. *f*(*x*)) = (*UN x*. *ntrunc k* (*f x*))
by (*simp add*: *ntrunc-def*, *blast*)

lemma *Scons-UN1-x*: *Scons* (*UN x*. *f x*) *M* = (*UN x*. *Scons* (*f x*) *M*)
by (*simp add*: *Scons-def*, *blast*)

lemma *Scons-UN1-y*: *Scons M* (*UN x*. *f x*) = (*UN x*. *Scons M* (*f x*))
by (*simp add*: *Scons-def*, *blast*)

lemma *In0-UN1*: *In0*(*UN x*. *f*(*x*)) = (*UN x*. *In0*(*f*(*x*)))
by (*simp add*: *In0-def Scons-UN1-y*)

lemma *In1-UN1*: *In1*(*UN x*. *f*(*x*)) = (*UN x*. *In1*(*f*(*x*)))
by (*simp add*: *In1-def Scons-UN1-y*)

lemma *dprodI* [*intro*!]:
[| (*M*,*M'*):*r*; (*N*,*N'*):*s* |] ==> (*Scons M N*, *Scons M' N'*) : *dprod r s*
by (*auto simp add*: *dprod-def*)

lemma *dprodE* [*elim*!]:
[| *c* : *dprod r s*;
!!*x y x' y'*. [| (*x*,*x'*) : *r*; (*y*,*y'*) : *s*;
c = (*Scons x y*, *Scons x' y'*) |] ==> *P*
|] ==> *P*
by (*auto simp add*: *dprod-def*)

lemma *dsum-In0I* [*intro*]: (*M*,*M'*):*r* ==> (*In0*(*M*), *In0*(*M'*)) : *dsum r s*
by (*auto simp add*: *dsum-def*)

lemma *dsum-In1I* [*intro*]: (*N*,*N'*):*s* ==> (*In1*(*N*), *In1*(*N'*)) : *dsum r s*
by (*auto simp add*: *dsum-def*)

lemma *dsumE* [*elim*!]:
[| *w* : *dsum r s*;
!!*x x'*. [| (*x*,*x'*) : *r*; *w* = (*In0*(*x*), *In0*(*x'*)) |] ==> *P*;
!!*y y'*. [| (*y*,*y'*) : *s*; *w* = (*In1*(*y*), *In1*(*y'*)) |] ==> *P*
|] ==> *P*
by (*auto simp add*: *dsum-def*)

lemma *dprod-mono*: [| *r*<=*r'*; *s*<=*s'* |] ==> *dprod r s* <= *dprod r' s'*
by *blast*

lemma *dsum-mono*: [| *r*<=*r'*; *s*<=*s'* |] ==> *dsum r s* <= *dsum r' s'*
by *blast*

lemma *dprod-Sigma*: (*dprod* (*A* × *B*) (*C* × *D*)) <= (*uprod A C*) × (*uprod B D*)
by *blast*

lemmas *dprod-subset-Sigma* = *subset-trans* [*OF dprod-mono dprod-Sigma*]

lemma *dprod-subset-Sigma2*:
(*dprod* (*Sigma A B*) (*Sigma C D*)) <= *Sigma* (*uprod A C*) (*Split* (%*x y*. *uprod* (*B x*) (*D y*)))
by *auto*

lemma *dsum-Sigma*: (*dsum* (*A* × *B*) (*C* × *D*)) <= (*usum A C*) × (*usum B D*)
by *blast*

lemmas *dsum-subset-Sigma* = *subset-trans* [*OF dsum-mono dsum-Sigma*]

lemma *Domain-dprod* [*simp*]: *Domain* (*dprod r s*) = *uprod* (*Domain r*) (*Domain s*)
by *auto*

lemma *Domain-dsum* [*simp*]: *Domain* (*dsum r s*) = *usum* (*Domain r*) (*Domain s*)
by *auto*

hides popular names

hide-type (**open**) *node item*
hide-const (**open**) *Push Node Atom Leaf Numb Lim Split Case*

ML-file *~~/src/HOL/Tools/Old-Datatype/old-datatype.ML*
ML-file *~~/src/HOL/Tools/inductive-realizer.ML*

end

12 Bijections between natural numbers and other types

theory *Nat-Bijection*
imports *Main*
begin

12.1 Type *nat* × *nat*

Triangle numbers: 0, 1, 3, 6, 10, 15, ...

definition *triangle* :: *nat* ⇒ *nat*
  where *triangle n* = (*n* ∗ *Suc n*) *div 2*

lemma *triangle-0* [*simp*]: *triangle 0* = *0*
unfolding *triangle-def* **by** *simp*

lemma *triangle-Suc* [*simp*]: *triangle* (*Suc n*) = *triangle n* + *Suc n*
unfolding *triangle-def* **by** *simp*

definition *prod-encode* :: *nat* × *nat* ⇒ *nat*
  where *prod-encode* = (λ(*m*, *n*). *triangle* (*m* + *n*) + *m*)

In this auxiliary function, *triangle k* + *m* is an invariant.

fun *prod-decode-aux* :: *nat* ⇒ *nat* ⇒ *nat* × *nat*
where
  prod-decode-aux k m =
    (*if m* ≤ *k then* (*m*, *k* − *m*) *else prod-decode-aux* (*Suc k*) (*m* − *Suc k*))

declare *prod-decode-aux.simps* [*simp del*]

definition *prod-decode* :: *nat* ⇒ *nat* × *nat*
  where *prod-decode* = *prod-decode-aux 0*

lemma *prod-encode-prod-decode-aux*:
  prod-encode (*prod-decode-aux k m*) = *triangle k* + *m*
apply (*induct k m rule*: *prod-decode-aux.induct*)
apply (*subst prod-decode-aux.simps*)
apply (*simp add*: *prod-encode-def*)
done

lemma *prod-decode-inverse* [*simp*]: *prod-encode* (*prod-decode n*) = *n*
unfolding *prod-decode-def* **by** (*simp add*: *prod-encode-prod-decode-aux*)

lemma *prod-decode-triangle-add*: *prod-decode* (*triangle k* + *m*) = *prod-decode-aux k m*
apply (*induct k arbitrary*: *m*)
apply (*simp add*: *prod-decode-def*)
apply (*simp only*: *triangle-Suc add.assoc*)
apply (*subst prod-decode-aux.simps*, *simp*)
done

lemma *prod-encode-inverse* [*simp*]: *prod-decode* (*prod-encode x*) = *x*
unfolding *prod-encode-def*
apply (*induct x*)
apply (*simp add*: *prod-decode-triangle-add*)
apply (*subst prod-decode-aux.simps*, *simp*)

done

lemma *inj-prod-encode*: *inj-on prod-encode A*
by (*rule inj-on-inverseI*, *rule prod-encode-inverse*)

lemma *inj-prod-decode*: *inj-on prod-decode A*
by (*rule inj-on-inverseI*, *rule prod-decode-inverse*)

lemma *surj-prod-encode*: *surj prod-encode*
by (*rule surjI*, *rule prod-decode-inverse*)

lemma *surj-prod-decode*: *surj prod-decode*
by (*rule surjI*, *rule prod-encode-inverse*)

lemma *bij-prod-encode*: *bij prod-encode*
by (*rule bijI* [*OF inj-prod-encode surj-prod-encode*])

lemma *bij-prod-decode*: *bij prod-decode*
by (*rule bijI* [*OF inj-prod-decode surj-prod-decode*])

lemma *prod-encode-eq*: *prod-encode x* = *prod-encode y* ⟷ *x* = *y*
by (*rule inj-prod-encode* [*THEN inj-eq*])

lemma *prod-decode-eq*: *prod-decode x* = *prod-decode y* ⟷ *x* = *y*
by (*rule inj-prod-decode* [*THEN inj-eq*])

Ordering properties

lemma *le-prod-encode-1*: *a* ≤ *prod-encode* (*a*, *b*)
unfolding *prod-encode-def* **by** *simp*

lemma *le-prod-encode-2*: *b* ≤ *prod-encode* (*a*, *b*)
unfolding *prod-encode-def* **by** (*induct b*, *simp-all*)

12.2 Type *nat* + *nat*

definition *sum-encode* :: *nat* + *nat* ⇒ *nat*
where
sum-encode x = (*case x of Inl a* ⇒ *2* ∗ *a* | *Inr b* ⇒ *Suc* (*2* ∗ *b*))

definition *sum-decode* :: *nat* ⇒ *nat* + *nat*
where
sum-decode n = (*if even n then Inl* (*n div 2*) *else Inr* (*n div 2*))

lemma *sum-encode-inverse* [*simp*]: *sum-decode* (*sum-encode x*) = *x*
unfolding *sum-decode-def sum-encode-def*
by (*induct x*) *simp-all*

lemma *sum-decode-inverse* [*simp*]: *sum-encode* (*sum-decode n*) = *n*
by (*simp add*: *even-two-times-div-two sum-decode-def sum-encode-def*)

lemma *inj-sum-encode*: *inj-on sum-encode A*
by (*rule inj-on-inverseI*, *rule sum-encode-inverse*)

lemma *inj-sum-decode*: *inj-on sum-decode A*
by (*rule inj-on-inverseI*, *rule sum-decode-inverse*)

lemma *surj-sum-encode*: *surj sum-encode*
by (*rule surjI*, *rule sum-decode-inverse*)

lemma *surj-sum-decode*: *surj sum-decode*
by (*rule surjI*, *rule sum-encode-inverse*)

lemma *bij-sum-encode*: *bij sum-encode*
by (*rule bijI* [*OF inj-sum-encode surj-sum-encode*])

lemma *bij-sum-decode*: *bij sum-decode*
by (*rule bijI* [*OF inj-sum-decode surj-sum-decode*])

lemma *sum-encode-eq*: *sum-encode x = sum-encode y* ⟷ *x = y*
by (*rule inj-sum-encode* [*THEN inj-eq*])

lemma *sum-decode-eq*: *sum-decode x = sum-decode y* ⟷ *x = y*
by (*rule inj-sum-decode* [*THEN inj-eq*])

12.3 Type *int*

definition *int-encode* :: *int* ⇒ *nat*
where
  int-encode i = sum-encode (*if 0* ≤ *i then Inl* (*nat i*) *else Inr* (*nat* (− *i* − *1*)))

definition *int-decode* :: *nat* ⇒ *int*
where
  int-decode n = (*case sum-decode n of Inl a* ⇒ *int a* | *Inr b* ⇒ − *int b* − *1*)

lemma *int-encode-inverse* [*simp*]: *int-decode* (*int-encode x*) = *x*
unfolding *int-decode-def int-encode-def* **by** *simp*

lemma *int-decode-inverse* [*simp*]: *int-encode* (*int-decode n*) = *n*
unfolding *int-decode-def int-encode-def* **using** *sum-decode-inverse* [*of n*]
by (*cases sum-decode n*, *simp-all*)

lemma *inj-int-encode*: *inj-on int-encode A*
by (*rule inj-on-inverseI*, *rule int-encode-inverse*)

lemma *inj-int-decode*: *inj-on int-decode A*
by (*rule inj-on-inverseI*, *rule int-decode-inverse*)

lemma *surj-int-encode*: *surj int-encode*
by (*rule surjI*, *rule int-decode-inverse*)

lemma *surj-int-decode*: *surj int-decode*
by (*rule surjI*, *rule int-encode-inverse*)

lemma *bij-int-encode*: *bij int-encode*
by (*rule bijI* [*OF inj-int-encode surj-int-encode*])

lemma *bij-int-decode*: *bij int-decode*
by (*rule bijI* [*OF inj-int-decode surj-int-decode*])

lemma *int-encode-eq*: *int-encode* x = *int-encode* $y \longleftrightarrow x = y$
by (*rule inj-int-encode* [*THEN inj-eq*])

lemma *int-decode-eq*: *int-decode* x = *int-decode* $y \longleftrightarrow x = y$
by (*rule inj-int-decode* [*THEN inj-eq*])

12.4 Type *nat list*

fun *list-encode* :: *nat list* ⇒ *nat*
where
  list-encode [] = *0*
| *list-encode* (*x* # *xs*) = *Suc* (*prod-encode* (*x*, *list-encode xs*))

function *list-decode* :: *nat* ⇒ *nat list*
where
  list-decode 0 = []
| *list-decode* (*Suc n*) = (*case prod-decode n of* (*x*, *y*) ⇒ *x* # *list-decode y*)
by *pat-completeness auto*

termination *list-decode*
apply (*relation measure id*, *simp-all*)
apply (*drule arg-cong* [**where** *f*=*prod-encode*])
apply (*drule sym*)
apply (*simp add*: *le-imp-less-Suc le-prod-encode-2*)
done

lemma *list-encode-inverse* [*simp*]: *list-decode* (*list-encode x*) = *x*
by (*induct x rule*: *list-encode.induct*) *simp-all*

lemma *list-decode-inverse* [*simp*]: *list-encode* (*list-decode n*) = *n*
apply (*induct n rule*: *list-decode.induct*, *simp*)
apply (*simp split*: *prod.split*)
apply (*simp add*: *prod-decode-eq* [*symmetric*])
done

lemma *inj-list-encode*: *inj-on list-encode A*
by (*rule inj-on-inverseI*, *rule list-encode-inverse*)

lemma *inj-list-decode*: *inj-on list-decode A*

by (*rule inj-on-inverseI*, *rule list-decode-inverse*)

lemma *surj-list-encode*: *surj list-encode*
by (*rule surjI*, *rule list-decode-inverse*)

lemma *surj-list-decode*: *surj list-decode*
by (*rule surjI*, *rule list-encode-inverse*)

lemma *bij-list-encode*: *bij list-encode*
by (*rule bijI* [*OF inj-list-encode surj-list-encode*])

lemma *bij-list-decode*: *bij list-decode*
by (*rule bijI* [*OF inj-list-decode surj-list-decode*])

lemma *list-encode-eq*: *list-encode x* = *list-encode y* ⟷ *x* = *y*
by (*rule inj-list-encode* [*THEN inj-eq*])

lemma *list-decode-eq*: *list-decode x* = *list-decode y* ⟷ *x* = *y*
by (*rule inj-list-decode* [*THEN inj-eq*])

12.5 Finite sets of naturals

12.5.1 Preliminaries

lemma *finite-vimage-Suc-iff*: *finite* (*Suc* −' *F*) ⟷ *finite F*
apply (*safe intro*!: *finite-vimageI inj-Suc*)
apply (*rule finite-subset* [**where** *B*=*insert 0* (*Suc* ' *Suc* −' *F*)])
apply (*rule subsetI*, *case-tac x*, *simp*, *simp*)
apply (*rule finite-insert* [*THEN iffD2*])
apply (*erule finite-imageI*)
done

lemma *vimage-Suc-insert-0*: *Suc* −' *insert 0 A* = *Suc* −' *A*
by *auto*

lemma *vimage-Suc-insert-Suc*:
  Suc −' *insert* (*Suc n*) *A* = *insert n* (*Suc* −' *A*)
by *auto*

lemma *div2-even-ext-nat*:
  fixes *x y* :: *nat*
  assumes *x div 2* = *y div 2*
  and *even x* ⟷ *even y*
  shows *x* = *y*
proof −
  from ‹*even x* ⟷ *even y*› **have** *x mod 2* = *y mod 2*
    by (*simp only*: *even-iff-mod-2-eq-zero*) *auto*
  with *assms* **have** *x div 2* ∗ *2* + *x mod 2* = *y div 2* ∗ *2* + *y mod 2*
    by *simp*
  then show *?thesis*

by *simp*
qed

12.5.2 From sets to naturals

definition *set-encode* :: *nat set* ⇒ *nat*
where *set-encode* = *setsum* (*op* ^ *2*)

lemma *set-encode-empty* [*simp*]: *set-encode* {} = *0*
by (*simp add*: *set-encode-def*)

lemma *set-encode-inf*: ~ *finite A* ⟹ *set-encode A* = *0*
by (*simp add*: *set-encode-def*)

lemma *set-encode-insert* [*simp*]:
⟦*finite A*; *n* ∉ *A*⟧ ⟹ *set-encode* (*insert n A*) = *2*^*n* + *set-encode A*
by (*simp add*: *set-encode-def*)

lemma *even-set-encode-iff*: *finite A* ⟹ *even* (*set-encode A*) ⟷ *0* ∉ *A*
unfolding *set-encode-def* **by** (*induct set*: *finite*, *auto*)

lemma *set-encode-vimage-Suc*: *set-encode* (*Suc* −' *A*) = *set-encode A div 2*
apply (*cases finite A*)
apply (*erule finite-induct*, *simp*)
apply (*case-tac x*)
apply (*simp add*: *even-set-encode-iff vimage-Suc-insert-0*)
apply (*simp add*: *finite-vimageI add.commute vimage-Suc-insert-Suc*)
apply (*simp add*: *set-encode-def finite-vimage-Suc-iff*)
done

lemmas *set-encode-div-2* = *set-encode-vimage-Suc* [*symmetric*]

12.5.3 From naturals to sets

definition *set-decode* :: *nat* ⇒ *nat set*
where *set-decode x* = {*n*. *odd* (*x div 2* ^ *n*)}

lemma *set-decode-0* [*simp*]: *0* ∈ *set-decode x* ⟷ *odd x*
by (*simp add*: *set-decode-def*)

lemma *set-decode-Suc* [*simp*]:
Suc n ∈ *set-decode x* ⟷ *n* ∈ *set-decode* (*x div 2*)
by (*simp add*: *set-decode-def div-mult2-eq*)

lemma *set-decode-zero* [*simp*]: *set-decode 0* = {}
by (*simp add*: *set-decode-def*)

lemma *set-decode-div-2*: *set-decode* (*x div 2*) = *Suc* −' *set-decode x*
by *auto*

lemma *set-decode-plus-power-2*:
  $n \notin$ *set-decode* $z \Longrightarrow$ *set-decode* $(2 \hat{} n + z) =$ *insert* n (*set-decode* z)
proof (*induct n arbitrary*: *z*)
  case *0* **show** *?case*
  proof (*rule set-eqI*)
    fix *q* **show** $q \in$ *set-decode* $(2 \hat{} 0 + z) \longleftrightarrow q \in$ *insert 0* (*set-decode z*)
      by (*induct q*) (*insert 0*, *simp-all*)
  qed
next
  case (*Suc n*) **show** *?case*
  proof (*rule set-eqI*)
    fix *q* **show** $q \in$ *set-decode* $(2 \hat{}$ *Suc* $n + z) \longleftrightarrow q \in$ *insert* (*Suc n*) (*set-decode*
z)
      by (*induct q*) (*insert Suc*, *simp-all*)
  qed
qed

lemma *finite-set-decode* [*simp*]: *finite* (*set-decode n*)
apply (*induct n rule*: *nat-less-induct*)
apply (*case-tac* $n = 0$, *simp*)
apply (*drule-tac x=n div 2* **in** *spec*, *simp*)
apply (*simp add*: *set-decode-div-2*)
apply (*simp add*: *finite-vimage-Suc-iff*)
done

12.5.4 Proof of isomorphism

lemma *set-decode-inverse* [*simp*]: *set-encode* (*set-decode n*) $= n$
apply (*induct n rule*: *nat-less-induct*)
apply (*case-tac* $n = 0$, *simp*)
apply (*drule-tac x=n div 2* **in** *spec*, *simp*)
apply (*simp add*: *set-decode-div-2 set-encode-vimage-Suc*)
apply (*erule div2-even-ext-nat*)
apply (*simp add*: *even-set-encode-iff*)
done

lemma *set-encode-inverse* [*simp*]: *finite A* $\Longrightarrow$ *set-decode* (*set-encode A*) $= A$
apply (*erule finite-induct*, *simp-all*)
apply (*simp add*: *set-decode-plus-power-2*)
done

lemma *inj-on-set-encode*: *inj-on set-encode* (*Collect finite*)
by (*rule inj-on-inverseI* [**where** *g=set-decode*], *simp*)

lemma *set-encode-eq*:
  ⟦*finite A*; *finite B*⟧ $\Longrightarrow$ *set-encode* $A =$ *set-encode* $B \longleftrightarrow A = B$
by (*rule iffI*, *simp add*: *inj-onD* [*OF inj-on-set-encode*], *simp*)

lemma *subset-decode-imp-le*:

```
  assumes set-decode m ⊆ set-decode n
  shows m ≤ n
proof –
  have n = m + set-encode (set-decode n − set-decode m)
  proof –
    obtain A B where m = set-encode A finite A
                     n = set-encode B finite B
      by (metis finite-set-decode set-decode-inverse)
  thus ?thesis using assms
    apply auto
    apply (simp add: set-encode-def add.commute setsum.subset-diff)
    done
  qed
  thus ?thesis
    by (metis le-add1)
qed

end
```

13 Encoding (almost) everything into natural numbers

```
theory Countable
imports Old-Datatype ~~/src/HOL/Rat Nat-Bijection
begin
```

13.1 The class of countable types

```
class countable =
  assumes ex-inj: ∃ to-nat :: 'a ⇒ nat. inj to-nat

lemma countable-classI:
  fixes f :: 'a ⇒ nat
  assumes ⋀x y. f x = f y ⟹ x = y
  shows OFCLASS('a, countable-class)
proof (intro-classes, rule exI)
  show inj f
    by (rule injI [OF assms]) assumption
qed
```

13.2 Conversion functions

```
definition to-nat :: 'a::countable ⇒ nat where
  to-nat = (SOME f. inj f)

definition from-nat :: nat ⇒ 'a::countable where
  from-nat = inv (to-nat :: 'a ⇒ nat)

lemma inj-to-nat [simp]: inj to-nat
```

by (*rule exE-some* [*OF ex-inj*]) (*simp add*: *to-nat-def*)

lemma *inj-on-to-nat*[*simp*, *intro*]: *inj-on to-nat S*
using *inj-to-nat* **by** (*auto simp*: *inj-on-def*)

lemma *surj-from-nat* [*simp*]: *surj from-nat*
unfolding *from-nat-def* **by** (*simp add*: *inj-imp-surj-inv*)

lemma *to-nat-split* [*simp*]: *to-nat* x = *to-nat* $y \longleftrightarrow x = y$
using *injD* [*OF inj-to-nat*] **by** *auto*

lemma *from-nat-to-nat* [*simp*]:
from-nat (*to-nat* x) = x
by (*simp add*: *from-nat-def*)

13.3 Finite types are countable

subclass (**in** *finite*) *countable*
proof
have *finite* (*UNIV*::'*a set*) **by** (*rule finite-UNIV*)
with *finite-conv-nat-seg-image* [*of UNIV*::'*a set*]
obtain n **and** f :: *nat* ⇒ '*a*
where *UNIV* = f ` $\{i.\ i < n\}$ **by** *auto*
then have *surj f* **unfolding** *surj-def* **by** *auto*
then have *inj* (*inv f*) **by** (*rule surj-imp-inj-inv*)
then show ∃ *to-nat* :: '*a* ⇒ *nat*. *inj to-nat* **by** (*rule exI*[*of inj*])
qed

13.4 Automatically proving countability of old-style datatypes

context
begin

qualified inductive *finite-item* :: '*a Old-Datatype.item* ⇒ *bool* **where**
undefined: *finite-item undefined*
| *In0*: *finite-item x* ⟹ *finite-item* (*Old-Datatype.In0 x*)
| *In1*: *finite-item x* ⟹ *finite-item* (*Old-Datatype.In1 x*)
| *Leaf*: *finite-item* (*Old-Datatype.Leaf a*)
| *Scons*: ⟦*finite-item x*; *finite-item y*⟧ ⟹ *finite-item* (*Old-Datatype.Scons x y*)

qualified function *nth-item* :: *nat* ⇒ ('*a*::*countable*) *Old-Datatype.item*
where
nth-item 0 = *undefined*
| *nth-item* (*Suc n*) =
(*case sum-decode n of*
Inl i ⇒
(*case sum-decode i of*
Inl j ⇒ *Old-Datatype.In0* (*nth-item j*)
| *Inr j* ⇒ *Old-Datatype.In1* (*nth-item j*))
| *Inr i* ⇒

  (*case sum-decode i of*
    Inl j ⇒ *Old-Datatype.Leaf* (*from-nat j*)
  | *Inr j* ⇒
    (*case prod-decode j of*
      (*a*, *b*) ⇒ *Old-Datatype.Scons* (*nth-item a*) (*nth-item b*))))
by *pat-completeness auto*

lemma *le-sum-encode-Inl*: $x \leq y \Longrightarrow x \leq$ *sum-encode* (*Inl y*)
unfolding *sum-encode-def* **by** *simp*

lemma *le-sum-encode-Inr*: $x \leq y \Longrightarrow x \leq$ *sum-encode* (*Inr y*)
unfolding *sum-encode-def* **by** *simp*

qualified termination
by (*relation measure id*)
  (*auto simp add*: *sum-encode-eq* [*symmetric*] *prod-encode-eq* [*symmetric*]
    le-imp-less-Suc le-sum-encode-Inl le-sum-encode-Inr
    le-prod-encode-1 le-prod-encode-2)

lemma *nth-item-covers*: *finite-item x* $\Longrightarrow \exists\, n.$ *nth-item n* = *x*
proof (*induct set*: *finite-item*)
  case *undefined*
  have *nth-item 0* = *undefined* **by** *simp*
  thus *?case* ..
next
  case (*In0 x*)
  then obtain *n* **where** *nth-item n* = *x* **by** *fast*
  hence *nth-item* (*Suc* (*sum-encode* (*Inl* (*sum-encode* (*Inl n*))))) = *Old-Datatype.In0*
x **by** *simp*
  thus *?case* ..
next
  case (*In1 x*)
  then obtain *n* **where** *nth-item n* = *x* **by** *fast*
  hence *nth-item* (*Suc* (*sum-encode* (*Inl* (*sum-encode* (*Inr n*))))) = *Old-Datatype.In1*
x **by** *simp*
  thus *?case* ..
next
  case (*Leaf a*)
  have *nth-item* (*Suc* (*sum-encode* (*Inr* (*sum-encode* (*Inl* (*to-nat a*)))))) = *Old-Datatype.Leaf*
a
    by *simp*
  thus *?case* ..
next
  case (*Scons x y*)
  then obtain *i j* **where** *nth-item i* = *x* **and** *nth-item j* = *y* **by** *fast*
  hence *nth-item*
   (*Suc* (*sum-encode* (*Inr* (*sum-encode* (*Inr* (*prod-encode* (*i*, *j*))))))) = *Old-Datatype.Scons*
x y
    by *simp*

```
  thus ?case ..
qed

theorem countable-datatype:
  fixes Rep :: 'b ⇒ ('a::countable) Old-Datatype.item
  fixes Abs :: ('a::countable) Old-Datatype.item ⇒ 'b
  fixes rep-set :: ('a::countable) Old-Datatype.item ⇒ bool
  assumes type: type-definition Rep Abs (Collect rep-set)
  assumes finite-item: ⋀x. rep-set x ⟹ finite-item x
  shows OFCLASS('b, countable-class)
proof
  def f ≡ λy. LEAST n. nth-item n = Rep y
  {
    fix y :: 'b
    have rep-set (Rep y)
      using type-definition.Rep [OF type] by simp
    hence finite-item (Rep y)
      by (rule finite-item)
    hence ∃ n. nth-item n = Rep y
      by (rule nth-item-covers)
    hence nth-item (f y) = Rep y
      unfolding f-def by (rule LeastI-ex)
    hence Abs (nth-item (f y)) = y
      using type-definition.Rep-inverse [OF type] by simp
  }
  hence inj f
    by (rule inj-on-inverseI)
  thus ∃f::'b ⇒ nat. inj f
    by − (rule exI)
qed

ML ‹
  fun old-countable-datatype-tac ctxt =
    SUBGOAL (fn (goal, -) =>
      let
        val ty-name =
          (case goal of
            (- $ Const (@{const-name Pure.type}, Type (@{type-name itself}, [Type
(n, -)]))) => n
          | - => raise Match)
        val typedef-info = hd (Typedef.get-info ctxt ty-name)
        val typedef-thm = #type-definition (snd typedef-info)
        val pred-name =
          (case HOLogic.dest-Trueprop (Thm.concl-of typedef-thm) of
            (- $ - $ - $ (- $ Const (n, -))) => n
          | - => raise Match)
        val induct-info = Inductive.the-inductive ctxt pred-name
        val pred-names = #names (fst induct-info)
        val induct-thms = #inducts (snd induct-info)
```

```
    val alist = pred-names ~~ induct-thms
    val induct-thm = the (AList.lookup (op =) alist pred-name)
    val vars = rev (Term.add-vars (Thm.prop-of induct-thm) [])
    val insts = vars |> map (fn (-, T) => try (Thm.cterm-of ctxt)
      (Const (@{const-name Countable.finite-item}, T)))
    val induct-thm' = Thm.instantiate' [] insts induct-thm
    val rules = @{thms finite-item.intros}
  in
    SOLVED' (fn i => EVERY
      [resolve-tac ctxt @{thms countable-datatype} i,
       resolve-tac ctxt [typedef-thm] i,
       eresolve-tac ctxt [induct-thm'] i,
       REPEAT (resolve-tac ctxt rules i ORELSE assume-tac ctxt i)]) 1
  end)
›
```

end

13.5 Automatically proving countability of datatypes

ML-file *../Tools/BNF/bnf-lfp-countable.ML*

ML ‹

```
fun countable-datatype-tac ctxt st =
  (case try (fn () => HEADGOAL (old-countable-datatype-tac ctxt) st) () of
    SOME res => res
  | NONE => BNF-LFP-Countable.countable-datatype-tac ctxt st);

(* compatibility *)
fun countable-tac ctxt =
  SELECT-GOAL (countable-datatype-tac ctxt);
›
```

method-setup *countable-datatype* = ‹
  Scan.succeed (SIMPLE-METHOD o countable-datatype-tac)
› *prove countable class instances for datatypes*

13.6 More Countable types

Naturals

instance *nat* :: *countable*
  by (*rule countable-classI* [*of id*]) *simp*

Pairs

instance *prod* :: (*countable*, *countable*) *countable*
  by (*rule countable-classI* [*of* $\lambda(x, y)$. *prod-encode* (*to-nat* x, *to-nat* y)])
    (*auto simp add*: *prod-encode-eq*)

Sums

instance *sum* :: (*countable*, *countable*) *countable*
  by (*rule countable-classI* [*of* (λ*x*. *case x of Inl a* ⇒ *to-nat* (*False*, *to-nat a*)
                                   | *Inr b* ⇒ *to-nat* (*True*, *to-nat b*))])
    (*simp split*: *sum.split-asm*)

Integers

instance *int* :: *countable*
  by (*rule countable-classI* [*of int-encode*]) (*simp add*: *int-encode-eq*)

Options

instance *option* :: (*countable*) *countable*
  by *countable-datatype*

Lists

instance *list* :: (*countable*) *countable*
  by *countable-datatype*

String literals

instance *String.literal* :: *countable*
  by (*rule countable-classI* [*of to-nat* ∘ *String.explode*]) (*auto simp add*: *explode-inject*)

Functions

instance *fun* :: (*finite*, *countable*) *countable*
proof
  obtain *xs* :: ′*a list* **where** *xs*: *set xs* = *UNIV*
    using *finite-list* [*OF finite-UNIV*] **..**
  show ∃ *to-nat*::(′*a* ⇒ ′*b*) ⇒ *nat*. *inj to-nat*
  proof
    show *inj* (λ*f*. *to-nat* (*map f xs*))
      by (*rule injI*, *simp add*: *xs fun-eq-iff*)
  qed
qed

Typereps

instance *typerep* :: *countable*
  by *countable-datatype*

13.7 The rationals are countably infinite

definition *nat-to-rat-surj* :: *nat* ⇒ *rat* **where**
  nat-to-rat-surj n = (*let* (*a*, *b*) = *prod-decode n in Fract* (*int-decode a*) (*int-decode b*))

lemma *surj-nat-to-rat-surj*: *surj nat-to-rat-surj*
unfolding *surj-def*
proof
  fix *r*::*rat*
  show ∃ *n*. *r* = *nat-to-rat-surj n*
  proof (*cases r*)

fix *i j* **assume** [*simp*]: *r* = *Fract i j* **and** *j* > *0*
have *r* = (*let m* = *int-encode i*; *n* = *int-encode j in nat-to-rat-surj* (*prod-encode* (*m*, *n*)))
by (*simp add*: *Let-def nat-to-rat-surj-def*)
thus ∃ *n*. *r* = *nat-to-rat-surj n* **by**(*auto simp*: *Let-def*)
qed
qed

lemma *Rats-eq-range-nat-to-rat-surj*: ℚ = *range nat-to-rat-surj*
by (*simp add*: *Rats-def surj-nat-to-rat-surj*)

context *field-char-0*
begin

lemma *Rats-eq-range-of-rat-o-nat-to-rat-surj*:
ℚ = *range* (*of-rat* ∘ *nat-to-rat-surj*)
using *surj-nat-to-rat-surj*
by (*auto simp*: *Rats-def image-def surj-def*) (*blast intro*: *arg-cong*[**where** *f* = *of-rat*])

lemma *surj-of-rat-nat-to-rat-surj*:
r ∈ ℚ ⟹ ∃ *n*. *r* = *of-rat* (*nat-to-rat-surj n*)
by (*simp add*: *Rats-eq-range-of-rat-o-nat-to-rat-surj image-def*)

end

instance *rat* :: *countable*
proof
show ∃ *to-nat*::*rat* ⇒ *nat*. *inj to-nat*
proof
have *surj nat-to-rat-surj*
by (*rule surj-nat-to-rat-surj*)
then show *inj* (*inv nat-to-rat-surj*)
by (*rule surj-imp-inj-inv*)
qed
qed

end

14 Infinite Sets and Related Concepts

theory *Infinite-Set*
imports *Main*
begin

The set of natural numbers is infinite.

lemma *infinite-nat-iff-unbounded-le*: *infinite* (*S*::*nat set*) ⟷ (∀ *m*. ∃ *n*≥*m*. *n* ∈ *S*)
using *frequently-cofinite*[*of* λ*x*. *x* ∈ *S*]

by (*simp add*: *cofinite-eq-sequentially frequently-def eventually-sequentially*)

lemma *infinite-nat-iff-unbounded*: *infinite* (S::*nat set*) $\longleftrightarrow$ ($\forall m. \exists n>m. n \in S$)
using *frequently-cofinite*[*of* $\lambda x. x \in S$]
by (*simp add*: *cofinite-eq-sequentially frequently-def eventually-at-top-dense*)

lemma *finite-nat-iff-bounded*: *finite* (S::*nat set*) $\longleftrightarrow$ ($\exists k. S \subseteq \{..<k\}$)
using *infinite-nat-iff-unbounded-le*[*of* S] **by** (*simp add*: *subset-eq*) (*metis not-le*)

lemma *finite-nat-iff-bounded-le*: *finite* (S::*nat set*) $\longleftrightarrow$ ($\exists k. S \subseteq \{.. k\}$)
using *infinite-nat-iff-unbounded*[*of* S] **by** (*simp add*: *subset-eq*) (*metis not-le*)

lemma *finite-nat-bounded*: *finite* (S::*nat set*) $\Longrightarrow \exists k. S \subseteq \{..<k\}$
by (*simp add*: *finite-nat-iff-bounded*)

For a set of natural numbers to be infinite, it is enough to know that for any number larger than some k, there is some larger number that is an element of the set.

lemma *unbounded-k-infinite*: $\forall m>k. \exists n>m. n \in S \Longrightarrow$ *infinite* (S::*nat set*)
apply (*clarsimp simp add*: *finite-nat-set-iff-bounded*)
apply (*drule-tac x=Suc* (*max m k*) **in** *spec*)
using *less-Suc-eq* **by** *fastforce*

lemma *nat-not-finite*: *finite* (*UNIV*::*nat set*) $\Longrightarrow R$
by *simp*

lemma *range-inj-infinite*:
inj (f::*nat* $\Rightarrow$ *'a*) $\Longrightarrow$ *infinite* (*range f*)
proof
assume *finite* (*range f*) **and** *inj f*
then have *finite* (*UNIV*::*nat set*)
by (*rule finite-imageD*)
then show *False* **by** *simp*
qed

The set of integers is also infinite.

lemma *infinite-int-iff-infinite-nat-abs*: *infinite* (S::*int set*) $\longleftrightarrow$ *infinite* ((*nat o abs*) ` S)
by (*auto simp*: *transfer-nat-int-set-relations o-def image-comp dest*: *finite-image-absD*)

proposition *infinite-int-iff-unbounded-le*: *infinite* (S::*int set*) $\longleftrightarrow$ ($\forall m. \exists n. |n| \geq m \land n \in S$)
apply (*simp add*: *infinite-int-iff-infinite-nat-abs infinite-nat-iff-unbounded-le o-def image-def*)
apply (*metis abs-ge-zero nat-le-eq-zle le-nat-iff*)
done

proposition *infinite-int-iff-unbounded*: *infinite* (S::*int set*) $\longleftrightarrow$ ($\forall m. \exists n. |n| > m \land n \in S$)

apply (*simp add*: *infinite-int-iff-infinite-nat-abs infinite-nat-iff-unbounded o-def image-def*)
apply (*metis* (*full-types*) *nat-le-iff nat-mono not-le*)
done

proposition *finite-int-iff-bounded*: *finite* (*S*::*int set*) ⟷ (∃ *k*. *abs* ‘ *S* ⊆ {..<*k*})
using *infinite-int-iff-unbounded-le*[*of S*] **by** (*simp add*: *subset-eq*) (*metis not-le*)

proposition *finite-int-iff-bounded-le*: *finite* (*S*::*int set*) ⟷ (∃ *k*. *abs* ‘ *S* ⊆ {.. *k*})
using *infinite-int-iff-unbounded*[*of S*] **by** (*simp add*: *subset-eq*) (*metis not-le*)

14.1 Infinitely Many and Almost All

We often need to reason about the existence of infinitely many (resp., all but finitely many) objects satisfying some predicate, so we introduce corresponding binders and their proof rules.

lemma *not-INFM* [*simp*]: ¬ (*INFM x*. *P x*) ⟷ (*MOST x*. ¬ *P x*) **by** (*fact not-frequently*)
lemma *not-MOST* [*simp*]: ¬ (*MOST x*. *P x*) ⟷ (*INFM x*. ¬ *P x*) **by** (*fact not-eventually*)

lemma *INFM-const* [*simp*]: (*INFM x*::'*a*. *P*) ⟷ *P* ∧ *infinite* (*UNIV*::'*a set*)
by (*simp add*: *frequently-const-iff*)

lemma *MOST-const* [*simp*]: (*MOST x*::'*a*. *P*) ⟷ *P* ∨ *finite* (*UNIV*::'*a set*)
by (*simp add*: *eventually-const-iff*)

lemma *INFM-imp-distrib*: (*INFM x*. *P x* ⟶ *Q x*) ⟷ ((*MOST x*. *P x*) ⟶ (*INFM x*. *Q x*))
by (*simp only*: *imp-conv-disj frequently-disj-iff not-eventually*)

lemma *MOST-imp-iff*: *MOST x*. *P x* ⟹ (*MOST x*. *P x* ⟶ *Q x*) ⟷ (*MOST x*. *Q x*)
by (*auto intro*: *eventually-rev-mp eventually-mono*)

lemma *INFM-conjI*: *INFM x*. *P x* ⟹ *MOST x*. *Q x* ⟹ *INFM x*. *P x* ∧ *Q x*
by (*rule frequently-rev-mp*[*of P*]) (*auto elim*: *eventually-mono*)

Properties of quantifiers with injective functions.

lemma *INFM-inj*: *INFM x*. *P* (*f x*) ⟹ *inj f* ⟹ *INFM x*. *P x*
using *finite-vimageI*[*of* {*x*. *P x*} *f*] **by** (*auto simp*: *frequently-cofinite*)

lemma *MOST-inj*: *MOST x*. *P x* ⟹ *inj f* ⟹ *MOST x*. *P* (*f x*)
using *finite-vimageI*[*of* {*x*. ¬ *P x*} *f*] **by** (*auto simp*: *eventually-cofinite*)

Properties of quantifiers with singletons.

lemma *not-INFM-eq* [*simp*]:
¬ (*INFM x*. *x* = *a*)
¬ (*INFM x*. *a* = *x*)

unfolding *frequently-cofinite* **by** *simp-all*

lemma *MOST-neq* [*simp*]:
$MOST\ x.\ x \neq a$
$MOST\ x.\ a \neq x$
unfolding *eventually-cofinite* **by** *simp-all*

lemma *INFM-neq* [*simp*]:
$(INFM\ x::'a.\ x \neq a) \longleftrightarrow infinite\ (UNIV::'a\ set)$
$(INFM\ x::'a.\ a \neq x) \longleftrightarrow infinite\ (UNIV::'a\ set)$
unfolding *frequently-cofinite* **by** *simp-all*

lemma *MOST-eq* [*simp*]:
$(MOST\ x::'a.\ x = a) \longleftrightarrow finite\ (UNIV::'a\ set)$
$(MOST\ x::'a.\ a = x) \longleftrightarrow finite\ (UNIV::'a\ set)$
unfolding *eventually-cofinite* **by** *simp-all*

lemma *MOST-eq-imp*:
$MOST\ x.\ x = a \longrightarrow P\ x$
$MOST\ x.\ a = x \longrightarrow P\ x$
unfolding *eventually-cofinite* **by** *simp-all*

Properties of quantifiers over the naturals.

lemma *MOST-nat*: $(\forall_\infty n.\ P\ (n::nat)) \longleftrightarrow (\exists\ m.\ \forall\ n>m.\ P\ n)$
by (*auto simp add*: *eventually-cofinite finite-nat-iff-bounded-le subset-eq not-le*[*symmetric*])

lemma *MOST-nat-le*: $(\forall_\infty n.\ P\ (n::nat)) \longleftrightarrow (\exists\ m.\ \forall\ n\geq m.\ P\ n)$
by (*auto simp add*: *eventually-cofinite finite-nat-iff-bounded subset-eq not-le*[*symmetric*])

lemma *INFM-nat*: $(\exists_\infty n.\ P\ (n::nat)) \longleftrightarrow (\forall\ m.\ \exists\ n>m.\ P\ n)$
by (*simp add*: *frequently-cofinite infinite-nat-iff-unbounded*)

lemma *INFM-nat-le*: $(\exists_\infty n.\ P\ (n::nat)) \longleftrightarrow (\forall\ m.\ \exists\ n\geq m.\ P\ n)$
by (*simp add*: *frequently-cofinite infinite-nat-iff-unbounded-le*)

lemma *MOST-INFM*: $infinite\ (UNIV::'a\ set) \Longrightarrow MOST\ x::'a.\ P\ x \Longrightarrow INFM\ x::'a.\ P\ x$
by (*simp add*: *eventually-frequently*)

lemma *MOST-Suc-iff*: $(MOST\ n.\ P\ (Suc\ n)) \longleftrightarrow (MOST\ n.\ P\ n)$
by (*simp add*: *cofinite-eq-sequentially eventually-sequentially-Suc*)

lemma
shows *MOST-SucI*: $MOST\ n.\ P\ n \Longrightarrow MOST\ n.\ P\ (Suc\ n)$
and *MOST-SucD*: $MOST\ n.\ P\ (Suc\ n) \Longrightarrow MOST\ n.\ P\ n$
by (*simp-all add*: *MOST-Suc-iff*)

lemma *MOST-ge-nat*: $MOST\ n::nat.\ m \leq n$
by (*simp add*: *cofinite-eq-sequentially eventually-ge-at-top*)

lemma *Inf-many-def*: *Inf-many P* $\longleftrightarrow$ *infinite* $\{x.\ P\ x\}$ **by** (*fact frequently-cofinite*)
lemma *Alm-all-def*: *Alm-all P* $\longleftrightarrow \neg$ (*INFM x*. $\neg$ *P x*) **by** *simp*
lemma *INFM-iff-infinite*: (*INFM x*. *P x*) $\longleftrightarrow$ *infinite* $\{x.\ P\ x\}$ **by** (*fact frequently-cofinite*)
lemma *MOST-iff-cofinite*: (*MOST x*. *P x*) $\longleftrightarrow$ *finite* $\{x.\ \neg\ P\ x\}$ **by** (*fact eventually-cofinite*)
lemma *INFM-EX*: $(\exists_\infty x.\ P\ x) \Longrightarrow (\exists x.\ P\ x)$ **by** (*fact frequently-ex*)
lemma *ALL-MOST*: $\forall x.\ P\ x \Longrightarrow \forall_\infty x.\ P\ x$ **by** (*fact always-eventually*)
lemma *INFM-mono*: $\exists_\infty x.\ P\ x \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow Q\ x) \Longrightarrow \exists_\infty x.\ Q\ x$ **by** (*fact frequently-elim1*)
lemma *MOST-mono*: $\forall_\infty x.\ P\ x \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow Q\ x) \Longrightarrow \forall_\infty x.\ Q\ x$ **by** (*fact eventually-mono*)
lemma *INFM-disj-distrib*: $(\exists_\infty x.\ P\ x \vee Q\ x) \longleftrightarrow (\exists_\infty x.\ P\ x) \vee (\exists_\infty x.\ Q\ x)$ **by** (*fact frequently-disj-iff*)
lemma *MOST-rev-mp*: $\forall_\infty x.\ P\ x \Longrightarrow \forall_\infty x.\ P\ x \longrightarrow Q\ x \Longrightarrow \forall_\infty x.\ Q\ x$ **by** (*fact eventually-rev-mp*)
lemma *MOST-conj-distrib*: $(\forall_\infty x.\ P\ x \wedge Q\ x) \longleftrightarrow (\forall_\infty x.\ P\ x) \wedge (\forall_\infty x.\ Q\ x)$ **by** (*fact eventually-conj-iff*)
lemma *MOST-conjI*: *MOST x*. *P x* $\Longrightarrow$ *MOST x*. *Q x* $\Longrightarrow$ *MOST x*. *P x* $\wedge$ *Q x* **by** (*fact eventually-conj*)
lemma *INFM-finite-Bex-distrib*: *finite A* $\Longrightarrow$ (*INFM y*. $\exists x \in A.\ P\ x\ y$) $\longleftrightarrow$ ($\exists x \in A$. *INFM y*. *P x y*) **by** (*fact frequently-bex-finite-distrib*)
lemma *MOST-finite-Ball-distrib*: *finite A* $\Longrightarrow$ (*MOST y*. $\forall x \in A.\ P\ x\ y$) $\longleftrightarrow$ ($\forall x \in A$. *MOST y*. *P x y*) **by** (*fact eventually-ball-finite-distrib*)
lemma *INFM-E*: *INFM x*. *P x* $\Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow thesis) \Longrightarrow thesis$ **by** (*fact frequentlyE*)
lemma *MOST-I*: $(\bigwedge x.\ P\ x) \Longrightarrow$ *MOST x*. *P x* **by** (*rule eventuallyI*)
lemmas *MOST-iff-finiteNeg* = *MOST-iff-cofinite*

14.2 Enumeration of an Infinite Set

The set's element type must be wellordered (e.g. the natural numbers).

Could be generalized to *enumerate′ S n* = (*SOME t*. $t \in s \wedge$ *finite* $\{s \in S.\ s < t\} \wedge$ *card* $\{s \in S.\ s < t\} = n$).

primrec (**in** *wellorder*) *enumerate* :: $'a$ *set* $\Rightarrow$ *nat* $\Rightarrow$ $'a$
where
enumerate-0: *enumerate S 0* = (*LEAST n*. $n \in S$)
| *enumerate-Suc*: *enumerate S* (*Suc n*) = *enumerate* ($S - \{$*LEAST n*. $n \in S\}$) *n*

lemma *enumerate-Suc′*: *enumerate S* (*Suc n*) = *enumerate* ($S - \{$*enumerate S 0*$\}$) *n*
by *simp*

lemma *enumerate-in-set*: *infinite S* $\Longrightarrow$ *enumerate S n* $\in S$
apply (*induct n arbitrary*: *S*)
apply (*fastforce intro*: *LeastI dest*!: *infinite-imp-nonempty*)
apply *simp*
apply (*metis DiffE infinite-remove*)

done

declare *enumerate-0* [*simp del*] *enumerate-Suc* [*simp del*]

lemma *enumerate-step*: *infinite S* $\Longrightarrow$ *enumerate S n* $<$ *enumerate S* (*Suc n*)
apply (*induct n arbitrary*: *S*)
apply (*rule order-le-neq-trans*)
apply (*simp add*: *enumerate-0 Least-le enumerate-in-set*)
apply (*simp only*: *enumerate-Suc'*)
apply (*subgoal-tac enumerate* (*S* $-$ {*enumerate S 0*}) *0* $\in$ *S* $-$ {*enumerate S 0*})
apply (*blast intro*: *sym*)
apply (*simp add*: *enumerate-in-set del*: *Diff-iff*)
apply (*simp add*: *enumerate-Suc'*)
done

lemma *enumerate-mono*: *m* $<$ *n* $\Longrightarrow$ *infinite S* $\Longrightarrow$ *enumerate S m* $<$ *enumerate S n*
apply (*erule less-Suc-induct*)
apply (*auto intro*: *enumerate-step*)
done

lemma *le-enumerate*:
assumes *S*: *infinite S*
shows *n* $\leq$ *enumerate S n*
using *S*
proof (*induct n*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
then have *n* $\leq$ *enumerate S n* **by** *simp*
also note *enumerate-mono*[*of n Suc n*, *OF* - ‹*infinite S*›]
finally show *?case* **by** *simp*
qed

lemma *enumerate-Suc''*:
fixes *S* :: *'a*::*wellorder set*
assumes *infinite S*
shows *enumerate S* (*Suc n*) = (*LEAST s*. *s* $\in$ *S* $\wedge$ *enumerate S n* $<$ *s*)
using *assms*
proof (*induct n arbitrary*: *S*)
case *0*
then have $\forall$ *s* $\in$ *S*. *enumerate S 0* $\leq$ *s*
by (*auto simp*: *enumerate.simps intro*: *Least-le*)
then show *?case*
unfolding *enumerate-Suc' enumerate-0*[*of S* $-$ {*enumerate S 0*}]
by (*intro arg-cong*[**where** *f* = *Least*] *ext*) *auto*

next
  case (*Suc n S*)
  show *?case*
    using *enumerate-mono*[*OF zero-less-Suc* ‹*infinite S*›, *of n*] ‹*infinite S*›
    apply (*subst* (*1 2*) *enumerate-Suc*′)
    apply (*subst Suc*)
    using ‹*infinite S*›
    apply *simp*
    apply (*intro arg-cong*[**where** *f* = *Least*] *ext*)
    apply (*auto simp*: *enumerate-Suc*′[*symmetric*])
    done
qed

lemma *enumerate-Ex*:
  assumes *S*: *infinite* (*S*::*nat set*)
  shows $s \in S \Longrightarrow \exists\, n.\ enumerate\ S\ n = s$
proof (*induct s rule*: *less-induct*)
  case (*less s*)
  show *?case*
  proof *cases*
    let $?y = Max\ \{s'{\in}S.\ s' < s\}$
    assume $\exists\, y{\in}S.\ y < s$
    then have *y*: $\bigwedge x.\ ?y < x \longleftrightarrow (\forall\, s'{\in}S.\ s' < s \longrightarrow s' < x)$
      by (*subst Max-less-iff*) *auto*
    then have *y-in*: $?y \in \{s'{\in}S.\ s' < s\}$
      by (*intro Max-in*) *auto*
    with *less.hyps*[*of ?y*] **obtain** *n* **where** $enumerate\ S\ n = ?y$
      by *auto*
    with *S* **have** $enumerate\ S\ (Suc\ n) = s$
      by (*auto simp*: *y less enumerate-Suc*′′ *intro*!: *Least-equality*)
    then show *?case* **by** *auto*
  next
    assume $*: \neg\ (\exists\, y{\in}S.\ y < s)$
    then have $\forall\, t{\in}S.\ s \le t$ **by** *auto*
    with ‹$s \in S$› **show** *?thesis*
      by (*auto intro*!: *exI*[*of - 0*] *Least-equality simp*: *enumerate-0*)
  qed
qed

lemma *bij-enumerate*:
  fixes *S* :: *nat set*
  assumes *S*: *infinite S*
  shows *bij-betw* (*enumerate S*) *UNIV S*
proof −
  have $\bigwedge n\ m.\ n \neq m \Longrightarrow enumerate\ S\ n \neq enumerate\ S\ m$
    using *enumerate-mono*[*OF -* ‹*infinite S*›] **by** (*auto simp*: *neq-iff*)
  then have *inj* (*enumerate S*)
    by (*auto simp*: *inj-on-def*)
  moreover have $\forall\, s \in S.\ \exists\, i.\ enumerate\ S\ i = s$

  using *enumerate-Ex*[*OF S*] **by** *auto*
 moreover note ⟨*infinite S*⟩
 ultimately show *?thesis*
  unfolding *bij-betw-def* **by** (*auto intro*: *enumerate-in-set*)
qed

end

15 Countable sets

theory *Countable-Set*
imports *Countable Infinite-Set*
begin

15.1 Predicate for countable sets

definition *countable* :: *'a set* ⇒ *bool* **where**
 countable S ⟷ (∃ *f*::*'a* ⇒ *nat*. *inj-on f S*)

lemma *countableE*:
 assumes *S*: *countable S* **obtains** *f* :: *'a* ⇒ *nat* **where** *inj-on f S*
 using *S* **by** (*auto simp*: *countable-def*)

lemma *countableI*: *inj-on* (*f*::*'a* ⇒ *nat*) *S* ⟹ *countable S*
 by (*auto simp*: *countable-def*)

lemma *countableI'*: *inj-on* (*f*::*'a* ⇒ *'b*::*countable*) *S* ⟹ *countable S*
 using *comp-inj-on*[*of f S to-nat*] **by** (*auto intro*: *countableI*)

lemma *countableE-bij*:
 assumes *S*: *countable S* **obtains** *f* :: *nat* ⇒ *'a* **and** *C* :: *nat set* **where** *bij-betw f C S*
 using *S* **by** (*blast elim*: *countableE dest*: *inj-on-imp-bij-betw bij-betw-inv*)

lemma *countableI-bij*: *bij-betw f* (*C*::*nat set*) *S* ⟹ *countable S*
 by (*blast intro*: *countableI bij-betw-inv-into bij-betw-imp-inj-on*)

lemma *countable-finite*: *finite S* ⟹ *countable S*
 by (*blast dest*: *finite-imp-inj-to-nat-seg countableI*)

lemma *countableI-bij1*: *bij-betw f A B* ⟹ *countable A* ⟹ *countable B*
 by (*blast elim*: *countableE-bij intro*: *bij-betw-trans countableI-bij*)

lemma *countableI-bij2*: *bij-betw f B A* ⟹ *countable A* ⟹ *countable B*
 by (*blast elim*: *countableE-bij intro*: *bij-betw-trans bij-betw-inv-into countableI-bij*)

lemma *countable-iff-bij*[*simp*]: *bij-betw f A B* ⟹ *countable A* ⟷ *countable B*
 by (*blast intro*: *countableI-bij1 countableI-bij2*)

lemma *countable-subset*: $A \subseteq B \Longrightarrow$ *countable B* $\Longrightarrow$ *countable A*
by (*auto simp*: *countable-def intro*: *subset-inj-on*)

lemma *countableI-type*[*intro*, *simp*]: *countable* (*A*:: *'a* :: *countable set*)
using *countableI*[*of to-nat A*] **by** *auto*

15.2 Enumerate a countable set

lemma *countableE-infinite*:
assumes *countable S infinite S*
obtains *e* :: *'a* ⇒ *nat* **where** *bij-betw e S UNIV*
proof −
obtain *f* :: *'a* ⇒ *nat* **where** *inj-on f S*
using ‹*countable S*› **by** (*rule countableE*)
then have *bij-betw f S* (*f'S*)
unfolding *bij-betw-def* **by** *simp*
moreover
from ‹*inj-on f S*› ‹*infinite S*› **have** *inf-fS*: *infinite* (*f'S*)
by (*auto dest*: *finite-imageD*)
then have *bij-betw* (*the-inv-into UNIV* (*enumerate* (*f'S*))) (*f'S*) *UNIV*
by (*intro bij-betw-the-inv-into bij-enumerate*)
ultimately have *bij-betw* (*the-inv-into UNIV* (*enumerate* (*f'S*)) ∘ *f*) *S UNIV*
by (*rule bij-betw-trans*)
then show *thesis* **..**
qed

lemma *countable-enum-cases*:
assumes *countable S*
obtains (*finite*) *f* :: *'a* ⇒ *nat* **where** *finite S bij-betw f S* {..<*card S*}
| (*infinite*) *f* :: *'a* ⇒ *nat* **where** *infinite S bij-betw f S UNIV*
using *ex-bij-betw-finite-nat*[*of S*] *countableE-infinite* ‹*countable S*›
by (*cases finite S*) (*auto simp add*: *atLeast0LessThan*)

definition *to-nat-on* :: *'a set* ⇒ *'a* ⇒ *nat* **where**
to-nat-on S = (*SOME f*. *if finite S then bij-betw f S* {..< *card S*} *else bij-betw f S UNIV*)

definition *from-nat-into* :: *'a set* ⇒ *nat* ⇒ *'a* **where**
from-nat-into S n = (*if n* ∈ *to-nat-on S* ' *S then inv-into S* (*to-nat-on S*) *n else SOME s*. *s*∈*S*)

lemma *to-nat-on-finite*: *finite S* $\Longrightarrow$ *bij-betw* (*to-nat-on S*) *S* {..< *card S*}
using *ex-bij-betw-finite-nat* **unfolding** *to-nat-on-def*
by (*intro someI2-ex*[**where** *Q*=λ*f*. *bij-betw f S* {..<*card S*}]) (*auto simp add*: *atLeast0LessThan*)

lemma *to-nat-on-infinite*: *countable S* $\Longrightarrow$ *infinite S* $\Longrightarrow$ *bij-betw* (*to-nat-on S*) *S UNIV*
using *countableE-infinite* **unfolding** *to-nat-on-def*

by (*intro someI2-ex*[**where** $Q=\lambda f.$ *bij-betw f S UNIV*]) *auto*

lemma *bij-betw-from-nat-into-finite*: *finite S* $\Longrightarrow$ *bij-betw* (*from-nat-into S*) {..< *card S*} *S*
unfolding *from-nat-into-def*[*abs-def*]
using *to-nat-on-finite*[*of S*]
apply (*subst bij-betw-cong*)
apply (*split if-split*)
apply (*simp add*: *bij-betw-def*)
apply (*auto cong*: *bij-betw-cong*
intro: *bij-betw-inv-into to-nat-on-finite*)
done

lemma *bij-betw-from-nat-into*: *countable S* $\Longrightarrow$ *infinite S* $\Longrightarrow$ *bij-betw* (*from-nat-into S*) *UNIV S*
unfolding *from-nat-into-def*[*abs-def*]
using *to-nat-on-infinite*[*of S*, *unfolded bij-betw-def*]
by (*auto cong*: *bij-betw-cong intro*: *bij-betw-inv-into to-nat-on-infinite*)

lemma *inj-on-to-nat-on*[*intro*]: *countable A* $\Longrightarrow$ *inj-on* (*to-nat-on A*) *A*
using *to-nat-on-infinite*[*of A*] *to-nat-on-finite*[*of A*]
by (*cases finite A*) (*auto simp*: *bij-betw-def*)

lemma *to-nat-on-inj*[*simp*]:
countable A $\Longrightarrow a \in A \Longrightarrow b \in A \Longrightarrow$ *to-nat-on A a* = *to-nat-on A b* $\longleftrightarrow a = b$
using *inj-on-to-nat-on*[*of A*] **by** (*auto dest*: *inj-onD*)

lemma *from-nat-into-to-nat-on*[*simp*]: *countable A* $\Longrightarrow a \in A \Longrightarrow$ *from-nat-into A* (*to-nat-on A a*) = *a*
by (*auto simp*: *from-nat-into-def intro*!: *inv-into-f-f*)

lemma *subset-range-from-nat-into*: *countable A* $\Longrightarrow A \subseteq$ *range* (*from-nat-into A*)
by (*auto intro*: *from-nat-into-to-nat-on*[*symmetric*])

lemma *from-nat-into*: $A \neq \{\} \Longrightarrow$ *from-nat-into A n* $\in A$
unfolding *from-nat-into-def* **by** (*metis equals0I inv-into-into someI-ex*)

lemma *range-from-nat-into-subset*: $A \neq \{\} \Longrightarrow$ *range* (*from-nat-into A*) $\subseteq A$
using *from-nat-into*[*of A*] **by** *auto*

lemma *range-from-nat-into*[*simp*]: $A \neq \{\} \Longrightarrow$ *countable A* $\Longrightarrow$ *range* (*from-nat-into A*) = *A*
by (*metis equalityI range-from-nat-into-subset subset-range-from-nat-into*)

lemma *image-to-nat-on*: *countable A* $\Longrightarrow$ *infinite A* $\Longrightarrow$ *to-nat-on A ‘ A* = *UNIV*
using *to-nat-on-infinite*[*of A*] **by** (*simp add*: *bij-betw-def*)

lemma *to-nat-on-surj*: *countable A* $\Longrightarrow$ *infinite A* $\Longrightarrow \exists\, a \in A.$ *to-nat-on A a* = *n*

by (*metis* (*no-types*) *image-iff iso-tuple-UNIV-I image-to-nat-on*)

lemma *to-nat-on-from-nat-into*[*simp*]: $n \in$ *to-nat-on A ' A* $\Longrightarrow$ *to-nat-on A* (*from-nat-into A n*) = *n*
by (*simp add*: *f-inv-into-f from-nat-into-def*)

lemma *to-nat-on-from-nat-into-infinite*[*simp*]:
countable A $\Longrightarrow$ *infinite A* $\Longrightarrow$ *to-nat-on A* (*from-nat-into A n*) = *n*
by (*metis image-iff to-nat-on-surj to-nat-on-from-nat-into*)

lemma *from-nat-into-inj*:
countable A $\Longrightarrow$ $m \in$ *to-nat-on A ' A* $\Longrightarrow$ $n \in$ *to-nat-on A ' A* $\Longrightarrow$
from-nat-into A m = *from-nat-into A n* $\longleftrightarrow$ *m* = *n*
by (*subst to-nat-on-inj*[*symmetric*, *of A*]) *auto*

lemma *from-nat-into-inj-infinite*[*simp*]:
countable A $\Longrightarrow$ *infinite A* $\Longrightarrow$ *from-nat-into A m* = *from-nat-into A n* $\longleftrightarrow$ *m* = *n*
using *image-to-nat-on*[*of A*] *from-nat-into-inj*[*of A m n*] **by** *simp*

lemma *eq-from-nat-into-iff*:
countable A $\Longrightarrow$ $x \in A$ $\Longrightarrow$ $i \in$ *to-nat-on A ' A* $\Longrightarrow$ *x* = *from-nat-into A i* $\longleftrightarrow$ *i* = *to-nat-on A x*
by *auto*

lemma *from-nat-into-surj*: *countable A* $\Longrightarrow$ $a \in A$ $\Longrightarrow$ $\exists$ *n*. *from-nat-into A n* = *a*
by (*rule exI*[*of - to-nat-on A a*]) *simp*

lemma *from-nat-into-inject*[*simp*]:
$A \neq \{\}$ $\Longrightarrow$ *countable A* $\Longrightarrow$ $B \neq \{\}$ $\Longrightarrow$ *countable B* $\Longrightarrow$ *from-nat-into A* = *from-nat-into B* $\longleftrightarrow$ *A* = *B*
by (*metis range-from-nat-into*)

lemma *inj-on-from-nat-into*: *inj-on from-nat-into* ({*A*. $A \neq \{\}$ $\wedge$ *countable A*})
unfolding *inj-on-def* **by** *auto*

15.3 Closure properties of countability

lemma *countable-SIGMA*[*intro*, *simp*]:
countable I $\Longrightarrow$ ($\bigwedge$*i*. $i \in I$ $\Longrightarrow$ *countable* (*A i*)) $\Longrightarrow$ *countable* (*SIGMA i* : *I*. *A i*)
by (*intro countableI'*[*of* λ(*i*, *a*). (*to-nat-on I i*, *to-nat-on* (*A i*) *a*)]) (*auto simp*: *inj-on-def*)

lemma *countable-image*[*intro*, *simp*]:
assumes *countable A*
shows *countable* (*f'A*)
proof –

obtain $g :: 'a \Rightarrow nat$ **where** *inj-on g A*
using *assms* **by** (*rule countableE*)
moreover have *inj-on* (*inv-into A f*) (*f'A*) *inv-into A f ' f ' A* $\subseteq$ *A*
by (*auto intro*: *inj-on-inv-into inv-into-into*)
ultimately show *?thesis*
by (*blast dest*: *comp-inj-on subset-inj-on intro*: *countableI*)
qed

lemma *countable-image-inj-on*: *countable* (*f ' A*) $\Longrightarrow$ *inj-on f A* $\Longrightarrow$ *countable A*
by (*metis countable-image the-inv-into-onto*)

lemma *countable-UN*[*intro*, *simp*]:
fixes *I* :: *'i set* **and** *A* :: *'i => 'a set*
assumes *I*: *countable I*
assumes *A*: $\bigwedge i.\ i \in I \Longrightarrow$ *countable* (*A i*)
shows *countable* ($\bigcup i{\in}I.\ A\ i$)
proof −
have ($\bigcup i{\in}I.\ A\ i$) = *snd '* (*SIGMA i* : *I. A i*) **by** (*auto simp*: *image-iff*)
then show *?thesis* **by** (*simp add*: *assms*)
qed

lemma *countable-Un*[*intro*]: *countable A* $\Longrightarrow$ *countable B* $\Longrightarrow$ *countable* ($A \cup B$)
by (*rule countable-UN*[*of* {*True*, *False*} λ*True* $\Rightarrow$ *A* | *False* $\Rightarrow$ *B*, *simplified*])
(*simp split*: *bool.split*)

lemma *countable-Un-iff*[*simp*]: *countable* ($A \cup B$) $\longleftrightarrow$ *countable A* $\wedge$ *countable B*
by (*metis countable-Un countable-subset inf-sup-ord*(*3*,*4*))

lemma *countable-Plus*[*intro*, *simp*]:
countable A $\Longrightarrow$ *countable B* $\Longrightarrow$ *countable* (*A* <+> *B*)
by (*simp add*: *Plus-def*)

lemma *countable-empty*[*intro*, *simp*]: *countable* {}
by (*blast intro*: *countable-finite*)

lemma *countable-insert*[*intro*, *simp*]: *countable A* $\Longrightarrow$ *countable* (*insert a A*)
using *countable-Un*[*of* {*a*} *A*] **by** (*auto simp*: *countable-finite*)

lemma *countable-Int1*[*intro*, *simp*]: *countable A* $\Longrightarrow$ *countable* ($A \cap B$)
by (*force intro*: *countable-subset*)

lemma *countable-Int2*[*intro*, *simp*]: *countable B* $\Longrightarrow$ *countable* ($A \cap B$)
by (*blast intro*: *countable-subset*)

lemma *countable-INT*[*intro*, *simp*]: $i \in I \Longrightarrow$ *countable* (*A i*) $\Longrightarrow$ *countable* ($\bigcap i{\in}I.\ A\ i$)
by (*blast intro*: *countable-subset*)

lemma *countable-Diff* [*intro*, *simp*]: *countable A* $\Longrightarrow$ *countable* $(A - B)$
by (*blast intro*: *countable-subset*)

lemma *countable-insert-eq* [*simp*]: *countable* (*insert x A*) = *countable A*
by *auto* (*metis Diff-insert-absorb countable-Diff insert-absorb*)

lemma *countable-vimage*: $B \subseteq$ *range f* $\Longrightarrow$ *countable* (f −' B) $\Longrightarrow$ *countable B*
by (*metis Int-absorb2 assms countable-image image-vimage-eq*)

lemma *surj-countable-vimage*: *surj f* $\Longrightarrow$ *countable* (f −' B) $\Longrightarrow$ *countable B*
by (*metis countable-vimage top-greatest*)

lemma *countable-Collect*[*simp*]: *countable A* $\Longrightarrow$ *countable* $\{a \in A.\ \varphi\ a\}$
by (*metis Collect-conj-eq Int-absorb Int-commute Int-def countable-Int1*)

lemma *countable-Image*:
assumes $\bigwedge y.\ y \in Y \Longrightarrow$ *countable* (X `` $\{y\}$)
assumes *countable Y*
shows *countable* (X `` Y)
proof −
have *countable* (X `` ($\bigcup y{\in}Y.\ \{y\}$))
unfolding *Image-UN* **by** (*intro countable-UN assms*)
then show *?thesis* **by** *simp*
qed

lemma *countable-relpow*:
fixes X :: *'a rel*
assumes *Image-X*: $\bigwedge Y.$ *countable Y* $\Longrightarrow$ *countable* (X `` Y)
assumes Y: *countable Y*
shows *countable* ((X ^^ i) `` Y)
using Y **by** (*induct i arbitrary*: Y) (*auto simp*: *relcomp-Image Image-X*)

lemma *countable-funpow*:
fixes f :: *'a set* $\Rightarrow$ *'a set*
assumes $\bigwedge A.$ *countable A* $\Longrightarrow$ *countable* (f A)
and *countable A*
shows *countable* ((f ^^ n) A)
by(*induction n*)(*simp-all add*: *assms*)

lemma *countable-rtrancl*:
($\bigwedge Y.$ *countable Y* $\Longrightarrow$ *countable* (X `` $Y$)) $\Longrightarrow$ *countable Y* $\Longrightarrow$ *countable* ($X$^* `` Y)
unfolding *rtrancl-is-UN-relpow UN-Image* **by** (*intro countable-UN countableI-type countable-relpow*)

lemma *countable-lists*[*intro*, *simp*]:
assumes A: *countable A* **shows** *countable* (*lists A*)
proof −
have *countable* (*lists* (*range* (*from-nat-into A*)))

by (*auto simp*: *lists-image*)
with *A* **show** *?thesis*
by (*auto dest*: *subset-range-from-nat-into countable-subset lists-mono*)
qed

lemma *Collect-finite-eq-lists*: *Collect finite* = *set* ' *lists UNIV*
using *finite-list* **by** *auto*

lemma *countable-Collect-finite*: *countable* (*Collect* (*finite*::'*a*::*countable set*⇒*bool*))
by (*simp add*: *Collect-finite-eq-lists*)

lemma *countable-rat*: *countable* $\mathbb{Q}$
unfolding *Rats-def* **by** *auto*

lemma *Collect-finite-subset-eq-lists*: {*A*. *finite A* ∧ *A* ⊆ *T*} = *set* ' *lists T*
using *finite-list* **by** (*auto simp*: *lists-eq-set*)

lemma *countable-Collect-finite-subset*:
countable T ⟹ *countable* {*A*. *finite A* ∧ *A* ⊆ *T*}
unfolding *Collect-finite-subset-eq-lists* **by** *auto*

lemma *countable-set-option* [*simp*]: *countable* (*set-option x*)
by(*cases x*) *auto*

15.4 Misc lemmas

lemma *infinite-countable-subset'*:
assumes *X*: *infinite X* **shows** ∃ *C*⊆*X*. *countable C* ∧ *infinite C*
proof −
from *infinite-countable-subset*[*OF X*] **guess** *f* **..**
then show *?thesis*
by (*intro exI*[*of - range f*]) (*auto simp*: *range-inj-infinite*)
qed

lemma *countable-all*:
assumes *S*: *countable S*
shows (∀ *s*∈*S*. *P s*) ⟷ (∀ *n*::*nat*. *from-nat-into S n* ∈ *S* ⟶ *P* (*from-nat-into S n*))
using *S*[*THEN subset-range-from-nat-into*] **by** *auto*

lemma *finite-sequence-to-countable-set*:
assumes *countable X* **obtains** *F* **where** ⋀*i*. *F i* ⊆ *X* ⋀*i*. *F i* ⊆ *F* (*Suc i*) ⋀*i*. *finite* (*F i*) (⋃*i*. *F i*) = *X*
proof − **show** *thesis*
apply (*rule that*[*of* λ*i*. *if X* = {} *then* {} *else from-nat-into X* ' {..*i*}])
apply (*auto simp*: *image-iff Ball-def intro*: *from-nat-into split*: *if-split-asm*)
proof −
fix *x n* **assume** *x* ∈ *X* ∀ *i m*. *m* ≤ *i* ⟶ *x* ≠ *from-nat-into X m*
with *from-nat-into-surj*[*OF* ‹*countable X*› ‹*x* ∈ *X*›]

show *False*
by *auto*
qed
qed

lemma *transfer-countable*[*transfer-rule*]:
bi-unique R $\Longrightarrow$ *rel-fun* (*rel-set R*) *op* = *countable countable*
by (*rule rel-funI*, *erule* (*1*) *bi-unique-rel-set-lemma*)
(*auto dest*: *countable-image-inj-on*)

15.5 Uncountable

abbreviation *uncountable* **where**
uncountable A $\equiv \neg$ *countable A*

lemma *uncountable-def*: *uncountable A* $\longleftrightarrow$ *A* $\neq$ {} $\wedge$ $\neg$ ($\exists$*f*::(*nat* $\Rightarrow$ *'a*). *range f* = *A*)
by (*auto intro*: *inj-on-inv-into simp*: *countable-def*)
(*metis all-not-in-conv inj-on-iff-surj subset-UNIV*)

lemma *uncountable-bij-betw*: *bij-betw f A B* $\Longrightarrow$ *uncountable B* $\Longrightarrow$ *uncountable A*
unfolding *bij-betw-def* **by** (*metis countable-image*)

lemma *uncountable-infinite*: *uncountable A* $\Longrightarrow$ *infinite A*
by (*metis countable-finite*)

lemma *uncountable-minus-countable*:
uncountable A $\Longrightarrow$ *countable B* $\Longrightarrow$ *uncountable* (*A* − *B*)
using *countable-Un*[*of B A* − *B*] *assms* **by** *auto*

lemma *countable-Diff-eq* [*simp*]: *countable* (*A* − {*x*}) = *countable A*
by (*meson countable-Diff countable-empty countable-insert uncountable-minus-countable*)

end

16 Non-denumerability of the Continuum.

theory *ContNotDenum*
imports *Complex-Main Countable-Set*
begin

16.1 Abstract

The following document presents a proof that the Continuum is uncountable. It is formalised in the Isabelle/Isar theorem proving system.

Theorem: The Continuum $\mathbb{R}$ is not denumerable. In other words, there does not exist a function f: $\mathbb{N} \Rightarrow \mathbb{R}$ such that f is surjective.

Outline: An elegant informal proof of this result uses Cantor's Diagonalisation argument. The proof presented here is not this one. First we formalise some properties of closed intervals, then we prove the Nested Interval Property. This property relies on the completeness of the Real numbers and is the foundation for our argument. Informally it states that an intersection of countable closed intervals (where each successive interval is a subset of the last) is non-empty. We then assume a surjective function f: $\mathbb{N} \Rightarrow \mathbb{R}$ exists and find a real x such that x is not in the range of f by generating a sequence of closed intervals then using the NIP.

theorem *real-non-denum*: $\neg$ ($\exists f$ :: *nat* $\Rightarrow$ *real*. *surj f*)
proof
  assume $\exists f$::*nat* $\Rightarrow$ *real*. *surj f*
  then obtain *f* :: *nat* $\Rightarrow$ *real* **where** *surj f* **..**

First we construct a sequence of nested intervals, ignoring *range f*.

  have $\forall$ *a b c*::*real*. *a* < *b* $\longrightarrow$ ($\exists$ *ka kb*. *ka* < *kb* $\wedge$ {*ka*..*kb*} $\subseteq$ {*a*..*b*} $\wedge$ *c* $\notin$ {*ka*..*kb*})
    using *assms*
    by (*auto simp add*: *not-le cong*: *conj-cong*)
      (*metis dense le-less-linear less-linear less-trans order-refl*)
  then obtain *i j* **where** *ij*:
    $\bigwedge$*a b c*::*real*. *a* < *b* $\Longrightarrow$ *i a b c* < *j a b c*
    $\bigwedge$*a b c*. *a* < *b* $\Longrightarrow$ {*i a b c* .. *j a b c*} $\subseteq$ {*a* .. *b*}
    $\bigwedge$*a b c*. *a* < *b* $\Longrightarrow$ *c* $\notin$ {*i a b c* .. *j a b c*}
    by *metis*

  def *ivl* $\equiv$ *rec-nat* (*f 0* + *1*, *f 0* + *2*) (λ*n x*. (*i* (*fst x*) (*snd x*) (*f n*), *j* (*fst x*) (*snd x*) (*f n*)))
  def *I* $\equiv$ λ*n*. {*fst* (*ivl n*) .. *snd* (*ivl n*)}

  have *ivl*[*simp*]:
    ivl 0 = (*f 0* + *1*, *f 0* + *2*)
    $\bigwedge$*n*. *ivl* (*Suc n*) = (*i* (*fst* (*ivl n*)) (*snd* (*ivl n*)) (*f n*), *j* (*fst* (*ivl n*)) (*snd* (*ivl n*)) (*f n*))
    unfolding *ivl-def* **by** *simp-all*

This is a decreasing sequence of non-empty intervals.

  { fix *n* **have** *fst* (*ivl n*) < *snd* (*ivl n*)
      by (*induct n*) (*auto intro*!: *ij*) **}**
  note *less* = *this*

  have *decseq I*
    unfolding *I-def decseq-Suc-iff ivl fst-conv snd-conv* **by** (*intro ij allI less*)

Now we apply the finite intersection property of compact sets.

  have *I 0* $\cap$ ($\bigcap$*i*. *I i*) $\neq$ {}
  proof (*rule compact-imp-fip-image*)
    fix *S* :: *nat set* **assume** *fin*: *finite S*

have {} ⊂ *I* (*Max* (*insert 0 S*))
unfolding *I-def* **using** *less*[*of Max* (*insert 0 S*)] **by** *auto*
also have *I* (*Max* (*insert 0 S*)) ⊆ (⋂*i*∈*insert 0 S*. *I i*)
using *fin decseqD*[*OF* ‹*decseq I*›, *of - Max* (*insert 0 S*)] **by** (*auto simp*: *Max-ge-iff*)
also have (⋂*i*∈*insert 0 S*. *I i*) = *I 0* ∩ (⋂*i*∈*S*. *I i*)
by *auto*
finally show *I 0* ∩ (⋂*i*∈*S*. *I i*) ≠ {}
by *auto*
qed (*auto simp*: *I-def*)
then obtain *x* **where** ⋀*n*. *x* ∈ *I n*
by *blast*
moreover from ‹*surj f*› **obtain** *j* **where** *x* = *f j*
by *blast*
ultimately have *f j* ∈ *I* (*Suc j*)
by *blast*
with *ij*(*3*)[*OF less*] **show** *False*
unfolding *I-def ivl fst-conv snd-conv* **by** *auto*
qed

lemma *uncountable-UNIV-real*: *uncountable* (*UNIV*::*real set*)
using *real-non-denum* **unfolding** *uncountable-def* **by** *auto*

lemma *bij-betw-open-intervals*:
fixes *a b c d* :: *real*
assumes *a* < *b c* < *d*
shows ∃*f*. *bij-betw f* {*a*<..<*b*} {*c*<..<*d*}
proof −
def *f* ≡ λ*a b c d x*::*real*. (*d* − *c*)/(*b* − *a*) ∗ (*x* − *a*) + *c*
{ fix *a b c d x* :: *real* **assume** ∗: *a* < *b c* < *d a* < *x x* < *b*
moreover from ∗ **have** (*d* − *c*) ∗ (*x* − *a*) < (*d* − *c*) ∗ (*b* − *a*)
by (*intro mult-strict-left-mono*) *simp-all*
moreover from ∗ **have** *0* < (*d* − *c*) ∗ (*x* − *a*) / (*b* − *a*)
by *simp*
ultimately have *f a b c d x* < *d c* < *f a b c d x*
by (*simp-all add*: *f-def field-simps*) **}**
with *assms* **have** *bij-betw* (*f a b c d*) {*a*<..<*b*} {*c*<..<*d*}
by (*intro bij-betw-byWitness*[**where** *f'*=*f c d a b*]) (*auto simp*: *f-def*)
thus *?thesis* **by** *auto*
qed

lemma *bij-betw-tan*: *bij-betw tan* {−*pi*/*2*<..<*pi*/*2*} *UNIV*
using *arctan-ubound* **by** (*intro bij-betw-byWitness*[**where** *f'*=*arctan*]) (*auto simp*: *arctan arctan-tan*)

lemma *uncountable-open-interval*:
fixes *a b* :: *real*
shows *uncountable* {*a*<..<*b*} ⟷ *a* < *b*
proof

```
  assume uncountable {a<..<b}
  then show a < b
    using uncountable-def by force
next
  assume a < b
  show uncountable {a<..<b}
  proof -
    obtain f where bij-betw f {a <..< b} {-pi/2<..<pi/2}
      using bij-betw-open-intervals[OF ‹a < b›, of -pi/2 pi/2] by auto
    then show ?thesis
      by (metis bij-betw-tan uncountable-bij-betw uncountable-UNIV-real)
  qed
qed

lemma uncountable-half-open-interval-1:
  fixes a :: real shows uncountable {a..<b} ⟷ a<b
  apply auto
  using atLeastLessThan-empty-iff apply fastforce
  using uncountable-open-interval [of a b]
  by (metis countable-Un-iff ivl-disj-un-singleton(3))

lemma uncountable-half-open-interval-2:
  fixes a :: real shows uncountable {a<..b} ⟷ a<b
  apply auto
  using atLeastLessThan-empty-iff apply fastforce
  using uncountable-open-interval [of a b]
  by (metis countable-Un-iff ivl-disj-un-singleton(4))

lemma real-interval-avoid-countable-set:
  fixes a b :: real and A :: real set
  assumes a < b and countable A
  shows ∃x∈{a<..<b}. x ∉ A
proof -
  from ‹countable A› have countable (A ∩ {a<..<b}) by auto
  moreover with ‹a < b› have ¬ countable {a<..<b}
    by (simp add: uncountable-open-interval)
  ultimately have A ∩ {a<..<b} ≠ {a<..<b} by auto
  hence A ∩ {a<..<b} ⊂ {a<..<b}
    by (intro psubsetI, auto)
  hence ∃x. x ∈ {a<..<b} - A ∩ {a<..<b}
    by (rule psubset-imp-ex-mem)
  thus ?thesis by auto
qed

lemma open-minus-countable:
  fixes S A :: real set assumes countable A S ≠ {} open S
  shows ∃x∈S. x ∉ A
proof -
  obtain x where x ∈ S
```

using ‹$S \neq \{\}$› **by** *auto*
then obtain e **where** $0 < e$ $\{y.\ dist\ y\ x < e\} \subseteq S$
using ‹*open S*› **by** (*auto simp*: *open-dist subset-eq*)
moreover have $\{y.\ dist\ y\ x < e\} = \{x - e <..< x + e\}$
by (*auto simp*: *dist-real-def*)
ultimately have *uncountable* $(S - A)$
using *uncountable-open-interval*[*of* $x - e\ x + e$] ‹*countable A*›
by (*intro uncountable-minus-countable*) (*auto dest*: *countable-subset*)
then show *?thesis*
unfolding *uncountable-def* **by** *auto*
qed

end

17 Inner Product Spaces and the Gradient Derivative

theory *Inner-Product*
imports ~~/*src*/*HOL*/*Complex-Main*
begin

17.1 Real inner product spaces

Temporarily relax type constraints for *open*, *uniformity*, *dist*, and *norm*.

setup ‹*Sign.add-const-constraint*
(@{*const-name open*}, *SOME* @{*typ* $'a$::*open set* ⇒ *bool*})›

setup ‹*Sign.add-const-constraint*
(@{*const-name dist*}, *SOME* @{*typ* $'a$::*dist* ⇒ $'a$ ⇒ *real*})›

setup ‹*Sign.add-const-constraint*
(@{*const-name uniformity*}, *SOME* @{*typ* ($'a$::*uniformity* × $'a$) *filter*})›

setup ‹*Sign.add-const-constraint*
(@{*const-name norm*}, *SOME* @{*typ* $'a$::*norm* ⇒ *real*})›

class *real-inner* = *real-vector* + *sgn-div-norm* + *dist-norm* + *uniformity-dist* + *open-uniformity* +
fixes *inner* :: $'a \Rightarrow 'a \Rightarrow real$
assumes *inner-commute*: *inner x y* = *inner y x*
and *inner-add-left*: *inner* $(x + y)$ z = *inner x z* + *inner y z*
and *inner-scaleR-left* [*simp*]: *inner* (*scaleR r x*) $y = r *$ (*inner x y*)
and *inner-ge-zero* [*simp*]: $0 \leq$ *inner x x*
and *inner-eq-zero-iff* [*simp*]: *inner x x* $= 0 \longleftrightarrow x = 0$
and *norm-eq-sqrt-inner*: *norm x* = *sqrt* (*inner x x*)
begin

lemma *inner-zero-left* [*simp*]: *inner* $0\ x = 0$

using *inner-add-left* [*of 0 0 x*] **by** *simp*

lemma *inner-minus-left* [*simp*]: *inner* (− *x*) *y* = − *inner x y*
using *inner-add-left* [*of x* − *x y*] **by** *simp*

lemma *inner-diff-left*: *inner* (*x* − *y*) *z* = *inner x z* − *inner y z*
using *inner-add-left* [*of x* − *y z*] **by** *simp*

lemma *inner-setsum-left*: *inner* ($\sum$ *x*∈*A*. *f x*) *y* = ($\sum$ *x*∈*A*. *inner* (*f x*) *y*)
by (*cases finite A*, *induct set*: *finite*, *simp-all add*: *inner-add-left*)

Transfer distributivity rules to right argument.

lemma *inner-add-right*: *inner x* (*y* + *z*) = *inner x y* + *inner x z*
using *inner-add-left* [*of y z x*] **by** (*simp only*: *inner-commute*)

lemma *inner-scaleR-right* [*simp*]: *inner x* (*scaleR r y*) = *r* ∗ (*inner x y*)
using *inner-scaleR-left* [*of r y x*] **by** (*simp only*: *inner-commute*)

lemma *inner-zero-right* [*simp*]: *inner x 0* = *0*
using *inner-zero-left* [*of x*] **by** (*simp only*: *inner-commute*)

lemma *inner-minus-right* [*simp*]: *inner x* (− *y*) = − *inner x y*
using *inner-minus-left* [*of y x*] **by** (*simp only*: *inner-commute*)

lemma *inner-diff-right*: *inner x* (*y* − *z*) = *inner x y* − *inner x z*
using *inner-diff-left* [*of y z x*] **by** (*simp only*: *inner-commute*)

lemma *inner-setsum-right*: *inner x* ($\sum$ *y*∈*A*. *f y*) = ($\sum$ *y*∈*A*. *inner x* (*f y*))
using *inner-setsum-left* [*of f A x*] **by** (*simp only*: *inner-commute*)

lemmas *inner-add* [*algebra-simps*] = *inner-add-left inner-add-right*
lemmas *inner-diff* [*algebra-simps*] = *inner-diff-left inner-diff-right*
lemmas *inner-scaleR* = *inner-scaleR-left inner-scaleR-right*

Legacy theorem names

lemmas *inner-left-distrib* = *inner-add-left*
lemmas *inner-right-distrib* = *inner-add-right*
lemmas *inner-distrib* = *inner-left-distrib inner-right-distrib*

lemma *inner-gt-zero-iff* [*simp*]: *0* < *inner x x* ⟷ *x* ≠ *0*
by (*simp add*: *order-less-le*)

lemma *power2-norm-eq-inner*: (*norm x*)^2 = *inner x x*
by (*simp add*: *norm-eq-sqrt-inner*)

Identities involving real multiplication and division.

lemma *inner-mult-left*: *inner* (*of-real m* ∗ *a*) *b* = *m* ∗ (*inner a b*)
by (*metis real-inner-class.inner-scaleR-left scaleR-conv-of-real*)

```
lemma inner-mult-right: inner a (of-real m * b) = m * (inner a b)
  by (metis real-inner-class.inner-scaleR-right scaleR-conv-of-real)

lemma inner-mult-left': inner (a * of-real m) b = m * (inner a b)
  by (simp add: of-real-def)

lemma inner-mult-right': inner a (b * of-real m) = (inner a b) * m
  by (simp add: of-real-def real-inner-class.inner-scaleR-right)

lemma Cauchy-Schwarz-ineq:
  (inner x y)^2 ≤ inner x x * inner y y
proof (cases)
  assume y = 0
  thus ?thesis by simp
next
  assume y: y ≠ 0
  let ?r = inner x y / inner y y
  have 0 ≤ inner (x − scaleR ?r y) (x − scaleR ?r y)
    by (rule inner-ge-zero)
  also have ... = inner x x − inner y x * ?r
    by (simp add: inner-diff)
  also have ... = inner x x − (inner x y)^2 / inner y y
    by (simp add: power2-eq-square inner-commute)
  finally have 0 ≤ inner x x − (inner x y)^2 / inner y y .
  hence (inner x y)^2 / inner y y ≤ inner x x
    by (simp add: le-diff-eq)
  thus (inner x y)^2 ≤ inner x x * inner y y
    by (simp add: pos-divide-le-eq y)
qed

lemma Cauchy-Schwarz-ineq2:
  |inner x y| ≤ norm x * norm y
proof (rule power2-le-imp-le)
  have (inner x y)^2 ≤ inner x x * inner y y
    using Cauchy-Schwarz-ineq .
  thus |inner x y|^2 ≤ (norm x * norm y)^2
    by (simp add: power-mult-distrib power2-norm-eq-inner)
  show 0 ≤ norm x * norm y
    unfolding norm-eq-sqrt-inner
    by (intro mult-nonneg-nonneg real-sqrt-ge-zero inner-ge-zero)
qed

lemma norm-cauchy-schwarz: inner x y ≤ norm x * norm y
  using Cauchy-Schwarz-ineq2 [of x y] by auto

subclass real-normed-vector
proof
  fix a :: real and x y :: 'a
  show norm x = 0 ⟷ x = 0
```

unfolding *norm-eq-sqrt-inner* **by** *simp*
show *norm* $(x + y) \leq$ *norm* x + *norm* y
proof (*rule power2-le-imp-le*)
have *inner* x $y \leq$ *norm* x $*$ *norm* y
by (*rule norm-cauchy-schwarz*)
thus $(norm\ (x + y))^2 \leq (norm\ x + norm\ y)^2$
unfolding *power2-sum power2-norm-eq-inner*
by (*simp add*: *inner-add inner-commute*)
show $0 \leq$ *norm* x + *norm* y
unfolding *norm-eq-sqrt-inner* **by** *simp*
qed
have *sqrt* $(a^2 *$ *inner* x $x) = |a| *$ *sqrt* (*inner* x x)
by (*simp add*: *real-sqrt-mult-distrib*)
then show *norm* $(a *_R x) = |a| *$ *norm* x
unfolding *norm-eq-sqrt-inner*
by (*simp add*: *power2-eq-square mult.assoc*)
qed

end

lemma *inner-divide-left*:
fixes *a* :: *'a* :: {*real-inner*,*real-div-algebra*}
shows *inner* (*a* / *of-real m*) *b* = (*inner a b*) / *m*
by (*metis* (*no-types*) *divide-inverse inner-commute inner-scaleR-right mult.left-neutral mult.right-neutral mult-scaleR-right of-real-inverse scaleR-conv-of-real times-divide-eq-left*)

lemma *inner-divide-right*:
fixes *a* :: *'a* :: {*real-inner*,*real-div-algebra*}
shows *inner a* (*b* / *of-real m*) = (*inner a b*) / *m*
by (*metis inner-commute inner-divide-left*)

Re-enable constraints for *open*, *uniformity*, *dist*, and *norm*.

setup ‹*Sign.add-const-constraint*
(@{*const-name open*}, *SOME* @{*typ 'a*::*topological-space set* ⇒ *bool*})›

setup ‹*Sign.add-const-constraint*
(@{*const-name uniformity*}, *SOME* @{*typ* (*'a*::*uniform-space* × *'a*) *filter*})›

setup ‹*Sign.add-const-constraint*
(@{*const-name dist*}, *SOME* @{*typ 'a*::*metric-space* ⇒ *'a* ⇒ *real*})›

setup ‹*Sign.add-const-constraint*
(@{*const-name norm*}, *SOME* @{*typ 'a*::*real-normed-vector* ⇒ *real*})›

lemma *bounded-bilinear-inner*:
bounded-bilinear (*inner*::*'a*::*real-inner* ⇒ *'a* ⇒ *real*)
proof
fix *x y z* :: *'a* **and** *r* :: *real*
show *inner* (*x* + *y*) *z* = *inner x z* + *inner y z*

  by (*rule inner-add-left*)
 show *inner x* (*y* + *z*) = *inner x y* + *inner x z*
  by (*rule inner-add-right*)
 show *inner* (*scaleR r x*) *y* = *scaleR r* (*inner x y*)
  unfolding *real-scaleR-def* **by** (*rule inner-scaleR-left*)
 show *inner x* (*scaleR r y*) = *scaleR r* (*inner x y*)
  unfolding *real-scaleR-def* **by** (*rule inner-scaleR-right*)
 show $\exists K. \forall x\ y::'a.$ *norm* (*inner x y*) $\leq$ *norm x* $*$ *norm y* $*$ *K*
 proof
  show $\forall x\ y::'a.$ *norm* (*inner x y*) $\leq$ *norm x* $*$ *norm y* $*$ *1*
   by (*simp add*: *Cauchy-Schwarz-ineq2*)
 qed
qed

lemmas *tendsto-inner* [*tendsto-intros*] =
 bounded-bilinear.tendsto [*OF bounded-bilinear-inner*]

lemmas *isCont-inner* [*simp*] =
 bounded-bilinear.isCont [*OF bounded-bilinear-inner*]

lemmas *has-derivative-inner* [*derivative-intros*] =
 bounded-bilinear.FDERIV [*OF bounded-bilinear-inner*]

lemmas *bounded-linear-inner-left* =
 bounded-bilinear.bounded-linear-left [*OF bounded-bilinear-inner*]

lemmas *bounded-linear-inner-right* =
 bounded-bilinear.bounded-linear-right [*OF bounded-bilinear-inner*]

lemmas *bounded-linear-inner-left-comp* = *bounded-linear-inner-left*[*THEN bounded-linear-compose*]

lemmas *bounded-linear-inner-right-comp* = *bounded-linear-inner-right*[*THEN bounded-linear-compose*]

lemmas *has-derivative-inner-right* [*derivative-intros*] =
 bounded-linear.has-derivative [*OF bounded-linear-inner-right*]

lemmas *has-derivative-inner-left* [*derivative-intros*] =
 bounded-linear.has-derivative [*OF bounded-linear-inner-left*]

lemma *differentiable-inner* [*simp*]:
 f differentiable (*at x within s*) $\Longrightarrow$ *g differentiable at x within s* $\Longrightarrow$ ($\lambda x.$ *inner* (*f x*) (*g x*)) *differentiable at x within s*
 unfolding *differentiable-def* **by** (*blast intro*: *has-derivative-inner*)

17.2 Class instances

instantiation *real* :: *real-inner*
begin

definition *inner-real-def* [*simp*]: *inner* = *op* ∗

instance
proof
  fix *x y z r* :: *real*
  show *inner x y* = *inner y x*
    unfolding *inner-real-def* **by** (*rule mult.commute*)
  show *inner* (*x* + *y*) *z* = *inner x z* + *inner y z*
    unfolding *inner-real-def* **by** (*rule distrib-right*)
  show *inner* (*scaleR r x*) *y* = *r* ∗ *inner x y*
    unfolding *inner-real-def real-scaleR-def* **by** (*rule mult.assoc*)
  show *0* ≤ *inner x x*
    unfolding *inner-real-def* **by** *simp*
  show *inner x x* = *0* ⟷ *x* = *0*
    unfolding *inner-real-def* **by** *simp*
  show *norm x* = *sqrt* (*inner x x*)
    unfolding *inner-real-def* **by** *simp*
qed

end

instantiation *complex* :: *real-inner*
begin

definition *inner-complex-def*:
  inner x y = *Re x* ∗ *Re y* + *Im x* ∗ *Im y*

instance
proof
  fix *x y z* :: *complex* **and** *r* :: *real*
  show *inner x y* = *inner y x*
    unfolding *inner-complex-def* **by** (*simp add*: *mult.commute*)
  show *inner* (*x* + *y*) *z* = *inner x z* + *inner y z*
    unfolding *inner-complex-def* **by** (*simp add*: *distrib-right*)
  show *inner* (*scaleR r x*) *y* = *r* ∗ *inner x y*
    unfolding *inner-complex-def* **by** (*simp add*: *distrib-left*)
  show *0* ≤ *inner x x*
    unfolding *inner-complex-def* **by** *simp*
  show *inner x x* = *0* ⟷ *x* = *0*
    unfolding *inner-complex-def*
    by (*simp add*: *add-nonneg-eq-0-iff complex-Re-Im-cancel-iff*)
  show *norm x* = *sqrt* (*inner x x*)
    unfolding *inner-complex-def complex-norm-def*
    by (*simp add*: *power2-eq-square*)
qed

end

lemma *complex-inner-1* [*simp*]: *inner 1 x* = *Re x*

unfolding *inner-complex-def* **by** *simp*

lemma *complex-inner-1-right* [*simp*]: *inner x 1 = Re x*
unfolding *inner-complex-def* **by** *simp*

lemma *complex-inner-ii-left* [*simp*]: *inner ii x = Im x*
unfolding *inner-complex-def* **by** *simp*

lemma *complex-inner-ii-right* [*simp*]: *inner x ii = Im x*
unfolding *inner-complex-def* **by** *simp*

17.3 Gradient derivative

definition
gderiv ::
['*a*::*real-inner* ⇒ *real*, '*a*, '*a*] ⇒ *bool*
((*GDERIV* (-)/ (-)/ :> (-)) [*1000*, *1000*, *60*] *60*)
where
GDERIV f x :> *D* ⟷ *FDERIV f x* :> (λ*h*. *inner h D*)

lemma *gderiv-deriv* [*simp*]: *GDERIV f x* :> *D* ⟷ *DERIV f x* :> *D*
by (*simp only*: *gderiv-def has-field-derivative-def inner-real-def mult-commute-abs*)

lemma *GDERIV-DERIV-compose*:
⟦*GDERIV f x* :> *df*; *DERIV g* (*f x*) :> *dg*⟧
⟹ *GDERIV* (λ*x*. *g* (*f x*)) *x* :> *scaleR dg df*
unfolding *gderiv-def has-field-derivative-def*
apply (*drule* (*1*) *has-derivative-compose*)
apply (*simp add*: *ac-simps*)
done

lemma *has-derivative-subst*: ⟦*FDERIV f x* :> *df*; *df* = *d*⟧ ⟹ *FDERIV f x* :> *d*
by *simp*

lemma *GDERIV-subst*: ⟦*GDERIV f x* :> *df*; *df* = *d*⟧ ⟹ *GDERIV f x* :> *d*
by *simp*

lemma *GDERIV-const*: *GDERIV* (λ*x*. *k*) *x* :> *0*
unfolding *gderiv-def inner-zero-right* **by** (*rule has-derivative-const*)

lemma *GDERIV-add*:
⟦*GDERIV f x* :> *df*; *GDERIV g x* :> *dg*⟧
⟹ *GDERIV* (λ*x*. *f x* + *g x*) *x* :> *df* + *dg*
unfolding *gderiv-def inner-add-right* **by** (*rule has-derivative-add*)

lemma *GDERIV-minus*:
GDERIV f x :> *df* ⟹ *GDERIV* (λ*x*. − *f x*) *x* :> − *df*
unfolding *gderiv-def inner-minus-right* **by** (*rule has-derivative-minus*)

lemma *GDERIV-diff*:
⟦*GDERIV f x* :> *df*; *GDERIV g x* :> *dg*⟧
⟹ *GDERIV* (λ*x*. *f x* − *g x*) *x* :> *df* − *dg*
unfolding *gderiv-def inner-diff-right* **by** (*rule has-derivative-diff*)

lemma *GDERIV-scaleR*:
⟦*DERIV f x* :> *df*; *GDERIV g x* :> *dg*⟧
⟹ *GDERIV* (λ*x*. *scaleR* (*f x*) (*g x*)) *x*
:> (*scaleR* (*f x*) *dg* + *scaleR df* (*g x*))
unfolding *gderiv-def has-field-derivative-def inner-add-right inner-scaleR-right*
apply (*rule has-derivative-subst*)
apply (*erule* (*1*) *has-derivative-scaleR*)
apply (*simp add*: *ac-simps*)
done

lemma *GDERIV-mult*:
⟦*GDERIV f x* :> *df*; *GDERIV g x* :> *dg*⟧
⟹ *GDERIV* (λ*x*. *f x* ∗ *g x*) *x* :> *scaleR* (*f x*) *dg* + *scaleR* (*g x*) *df*
unfolding *gderiv-def*
apply (*rule has-derivative-subst*)
apply (*erule* (*1*) *has-derivative-mult*)
apply (*simp add*: *inner-add ac-simps*)
done

lemma *GDERIV-inverse*:
⟦*GDERIV f x* :> *df*; *f x* ≠ *0*⟧
⟹ *GDERIV* (λ*x*. *inverse* (*f x*)) *x* :> − (*inverse* (*f x*))^2 $*_R$ *df*
apply (*erule GDERIV-DERIV-compose*)
apply (*erule DERIV-inverse* [*folded numeral-2-eq-2*])
done

lemma *GDERIV-norm*:
assumes *x* ≠ *0* **shows** *GDERIV* (λ*x*. *norm x*) *x* :> *sgn x*
proof −
have *1*: *FDERIV* (λ*x*. *inner x x*) *x* :> (λ*h*. *inner x h* + *inner h x*)
by (*intro has-derivative-inner has-derivative-ident*)
have *2*: (λ*h*. *inner x h* + *inner h x*) = (λ*h*. *inner h* (*scaleR 2 x*))
by (*simp add*: *fun-eq-iff inner-commute*)
have *0* < *inner x x* **using** ‹*x* ≠ *0*› **by** *simp*
then have *3*: *DERIV sqrt* (*inner x x*) :> (*inverse* (*sqrt* (*inner x x*)) / *2*)
by (*rule DERIV-real-sqrt*)
have *4*: (*inverse* (*sqrt* (*inner x x*)) / *2*) $*_R$ *2* $*_R$ *x* = *sgn x*
by (*simp add*: *sgn-div-norm norm-eq-sqrt-inner*)
show *?thesis*
unfolding *norm-eq-sqrt-inner*
apply (*rule GDERIV-subst* [*OF - 4*])
apply (*rule GDERIV-DERIV-compose* [**where** *g*=*sqrt* **and** *df*=*scaleR 2 x*])
apply (*subst gderiv-def*)
apply (*rule has-derivative-subst* [*OF - 2*])

apply (*rule 1*)
apply (*rule 3*)
done
qed

lemmas *has-derivative-norm* = *GDERIV-norm* [*unfolded gderiv-def*]

end

18 Additive group operations on product types

theory *Product-plus*
imports *Main*
begin

18.1 Operations

instantiation *prod* :: (*zero*, *zero*) *zero*
begin

definition *zero-prod-def*: *0* = (*0*, *0*)

instance ..
end

instantiation *prod* :: (*plus*, *plus*) *plus*
begin

definition *plus-prod-def*:
$x + y = (fst\ x + fst\ y,\ snd\ x + snd\ y)$

instance ..
end

instantiation *prod* :: (*minus*, *minus*) *minus*
begin

definition *minus-prod-def*:
$x - y = (fst\ x - fst\ y,\ snd\ x - snd\ y)$

instance ..
end

instantiation *prod* :: (*uminus*, *uminus*) *uminus*
begin

definition *uminus-prod-def*:
$- x = (- fst\ x,\ - snd\ x)$

instance ..
end

lemma *fst-zero* [*simp*]: *fst 0* = *0*
unfolding *zero-prod-def* **by** *simp*

lemma *snd-zero* [*simp*]: *snd 0* = *0*
unfolding *zero-prod-def* **by** *simp*

lemma *fst-add* [*simp*]: *fst* (*x* + *y*) = *fst x* + *fst y*
unfolding *plus-prod-def* **by** *simp*

lemma *snd-add* [*simp*]: *snd* (*x* + *y*) = *snd x* + *snd y*
unfolding *plus-prod-def* **by** *simp*

lemma *fst-diff* [*simp*]: *fst* (*x* − *y*) = *fst x* − *fst y*
unfolding *minus-prod-def* **by** *simp*

lemma *snd-diff* [*simp*]: *snd* (*x* − *y*) = *snd x* − *snd y*
unfolding *minus-prod-def* **by** *simp*

lemma *fst-uminus* [*simp*]: *fst* (− *x*) = − *fst x*
unfolding *uminus-prod-def* **by** *simp*

lemma *snd-uminus* [*simp*]: *snd* (− *x*) = − *snd x*
unfolding *uminus-prod-def* **by** *simp*

lemma *add-Pair* [*simp*]: (*a*, *b*) + (*c*, *d*) = (*a* + *c*, *b* + *d*)
unfolding *plus-prod-def* **by** *simp*

lemma *diff-Pair* [*simp*]: (*a*, *b*) − (*c*, *d*) = (*a* − *c*, *b* − *d*)
unfolding *minus-prod-def* **by** *simp*

lemma *uminus-Pair* [*simp*, *code*]: − (*a*, *b*) = (− *a*, − *b*)
unfolding *uminus-prod-def* **by** *simp*

18.2 Class instances

instance *prod* :: (*semigroup-add*, *semigroup-add*) *semigroup-add*
by *standard* (*simp add*: *prod-eq-iff add.assoc*)

instance *prod* :: (*ab-semigroup-add*, *ab-semigroup-add*) *ab-semigroup-add*
by *standard* (*simp add*: *prod-eq-iff add.commute*)

instance *prod* :: (*monoid-add*, *monoid-add*) *monoid-add*
by *standard* (*simp-all add*: *prod-eq-iff*)

instance *prod* :: (*comm-monoid-add*, *comm-monoid-add*) *comm-monoid-add*
by *standard* (*simp add*: *prod-eq-iff*)

instance *prod* :: (*cancel-semigroup-add*, *cancel-semigroup-add*) *cancel-semigroup-add*
  by *standard* (*simp-all add*: *prod-eq-iff*)

instance *prod* :: (*cancel-ab-semigroup-add*, *cancel-ab-semigroup-add*) *cancel-ab-semigroup-add*
  by *standard* (*simp-all add*: *prod-eq-iff diff-diff-eq*)

instance *prod* :: (*cancel-comm-monoid-add*, *cancel-comm-monoid-add*) *cancel-comm-monoid-add*
..

instance *prod* :: (*group-add*, *group-add*) *group-add*
 by *standard* (*simp-all add*: *prod-eq-iff*)

instance *prod* :: (*ab-group-add*, *ab-group-add*) *ab-group-add*
 by *standard* (*simp-all add*: *prod-eq-iff*)

lemma *fst-setsum*: *fst* $(\sum x{\in}A.\ f\ x) = (\sum x{\in}A.\ fst\ (f\ x))$
proof (*cases finite A*)
 case *True*
 then show *?thesis* **by** *induct simp-all*
next
 case *False*
 then show *?thesis* **by** *simp*
qed

lemma *snd-setsum*: *snd* $(\sum x{\in}A.\ f\ x) = (\sum x{\in}A.\ snd\ (f\ x))$
proof (*cases finite A*)
 case *True*
 then show *?thesis* **by** *induct simp-all*
next
 case *False*
 then show *?thesis* **by** *simp*
qed

lemma *setsum-prod*: $(\sum x{\in}A.\ (f\ x,\ g\ x)) = (\sum x{\in}A.\ f\ x,\ \sum x{\in}A.\ g\ x)$
proof (*cases finite A*)
 case *True*
 then show *?thesis* **by** *induct* (*simp-all add*: *zero-prod-def*)
next
 case *False*
 then show *?thesis* **by** (*simp add*: *zero-prod-def*)
qed

end

19 Cartesian Products as Vector Spaces

theory *Product-Vector*
imports *Inner-Product Product-plus*

begin

19.1 Product is a real vector space

instantiation *prod* :: (*real-vector*, *real-vector*) *real-vector*
begin

definition *scaleR-prod-def*:
  scaleR r A = (*scaleR r* (*fst A*), *scaleR r* (*snd A*))

lemma *fst-scaleR* [*simp*]: *fst* (*scaleR r A*) = *scaleR r* (*fst A*)
  unfolding *scaleR-prod-def* **by** *simp*

lemma *snd-scaleR* [*simp*]: *snd* (*scaleR r A*) = *scaleR r* (*snd A*)
  unfolding *scaleR-prod-def* **by** *simp*

lemma *scaleR-Pair* [*simp*]: *scaleR r* (*a*, *b*) = (*scaleR r a*, *scaleR r b*)
  unfolding *scaleR-prod-def* **by** *simp*

instance
proof
  fix *a b* :: *real* **and** *x y* :: *'a* × *'b*
  show *scaleR a* (*x* + *y*) = *scaleR a x* + *scaleR a y*
    by (*simp add*: *prod-eq-iff scaleR-right-distrib*)
  show *scaleR* (*a* + *b*) *x* = *scaleR a x* + *scaleR b x*
    by (*simp add*: *prod-eq-iff scaleR-left-distrib*)
  show *scaleR a* (*scaleR b x*) = *scaleR* (*a* ∗ *b*) *x*
    by (*simp add*: *prod-eq-iff*)
  show *scaleR 1 x* = *x*
    by (*simp add*: *prod-eq-iff*)
qed

end

19.2 Product is a metric space

instantiation *prod* :: (*metric-space*, *metric-space*) *dist*
begin

definition *dist-prod-def* [*code del*]:
  dist x y = *sqrt* ((*dist* (*fst x*) (*fst y*))^2 + (*dist* (*snd x*) (*snd y*))^2)

instance ..
end

instantiation *prod* :: (*metric-space*, *metric-space*) *uniformity-dist*
begin

definition [*code del*]:
  (*uniformity* :: ((*'a* × *'b*) × (*'a* × *'b*)) *filter*) =

```
  (INF e:{0 <..}. principal {(x, y). dist x y < e})

instance
  by standard (rule uniformity-prod-def)
end

declare uniformity-Abort[where 'a='a :: metric-space × 'b :: metric-space, code]

instantiation prod :: (metric-space, metric-space) metric-space
begin

lemma dist-Pair-Pair: dist (a, b) (c, d) = sqrt ((dist a c)^2 + (dist b d)^2)
  unfolding dist-prod-def by simp

lemma dist-fst-le: dist (fst x) (fst y) ≤ dist x y
  unfolding dist-prod-def by (rule real-sqrt-sum-squares-ge1)

lemma dist-snd-le: dist (snd x) (snd y) ≤ dist x y
  unfolding dist-prod-def by (rule real-sqrt-sum-squares-ge2)

instance
proof
  fix x y :: 'a × 'b
  show dist x y = 0 ⟷ x = y
    unfolding dist-prod-def prod-eq-iff by simp
next
  fix x y z :: 'a × 'b
  show dist x y ≤ dist x z + dist y z
    unfolding dist-prod-def
    by (intro order-trans [OF - real-sqrt-sum-squares-triangle-ineq]
        real-sqrt-le-mono add-mono power-mono dist-triangle2 zero-le-dist)
next
  fix S :: ('a × 'b) set
  have *: open S ⟷ (∀x∈S. ∃e>0. ∀y. dist y x < e ⟶ y ∈ S)
  proof
    assume open S show ∀x∈S. ∃e>0. ∀y. dist y x < e ⟶ y ∈ S
    proof
      fix x assume x ∈ S
      obtain A B where open A open B x ∈ A × B A × B ⊆ S
        using ‹open S› and ‹x ∈ S› by (rule open-prod-elim)
      obtain r where r: 0 < r ∀y. dist y (fst x) < r ⟶ y ∈ A
        using ‹open A› and ‹x ∈ A × B› unfolding open-dist by auto
      obtain s where s: 0 < s ∀y. dist y (snd x) < s ⟶ y ∈ B
        using ‹open B› and ‹x ∈ A × B› unfolding open-dist by auto
      let ?e = min r s
      have 0 < ?e ∧ (∀y. dist y x < ?e ⟶ y ∈ S)
      proof (intro allI impI conjI)
        show 0 < min r s by (simp add: r(1) s(1))
      next
```

```
      fix y assume dist y x < min r s
      hence dist y x < r and dist y x < s
        by simp-all
      hence dist (fst y) (fst x) < r and dist (snd y) (snd x) < s
        by (auto intro: le-less-trans dist-fst-le dist-snd-le)
      hence fst y ∈ A and snd y ∈ B
        by (simp-all add: r(2) s(2))
      hence y ∈ A × B by (induct y, simp)
      with ‹A × B ⊆ S› show y ∈ S ..
    qed
    thus ∃ e>0. ∀ y. dist y x < e ⟶ y ∈ S ..
  qed
next
  assume *: ∀ x∈S. ∃ e>0. ∀ y. dist y x < e ⟶ y ∈ S show open S
  proof (rule open-prod-intro)
    fix x assume x ∈ S
    then obtain e where 0 < e and S: ∀ y. dist y x < e ⟶ y ∈ S
      using * by fast
    def r ≡ e / sqrt 2 and s ≡ e / sqrt 2
    from ‹0 < e› have 0 < r and 0 < s
      unfolding r-def s-def by simp-all
    from ‹0 < e› have e = sqrt (r² + s²)
      unfolding r-def s-def by (simp add: power-divide)
    def A ≡ {y. dist (fst x) y < r} and B ≡ {y. dist (snd x) y < s}
    have open A and open B
      unfolding A-def B-def by (simp-all add: open-ball)
    moreover have x ∈ A × B
      unfolding A-def B-def mem-Times-iff
      using ‹0 < r› and ‹0 < s› by simp
    moreover have A × B ⊆ S
    proof (clarify)
      fix a b assume a ∈ A and b ∈ B
      hence dist a (fst x) < r and dist b (snd x) < s
        unfolding A-def B-def by (simp-all add: dist-commute)
      hence dist (a, b) x < e
        unfolding dist-prod-def ‹e = sqrt (r² + s²)›
        by (simp add: add-strict-mono power-strict-mono)
      thus (a, b) ∈ S
        by (simp add: S)
    qed
    ultimately show ∃ A B. open A ∧ open B ∧ x ∈ A × B ∧ A × B ⊆ S by
fast
  qed
qed
show open S = (∀ x∈S. ∀F (x', y) in uniformity. x' = x ⟶ y ∈ S)
  unfolding * eventually-uniformity-metric
  by (simp del: split-paired-All add: dist-prod-def dist-commute)
qed
```

end

declare [[*code abort*: *dist*::('*a*::*metric-space*∗'*b*::*metric-space*)⇒('*a*∗'*b*) ⇒ *real*]]

lemma *Cauchy-fst*: *Cauchy X* ⟹ *Cauchy* (λ*n*. *fst* (*X n*))
  unfolding *Cauchy-def* **by** (*fast elim*: *le-less-trans* [*OF dist-fst-le*])

lemma *Cauchy-snd*: *Cauchy X* ⟹ *Cauchy* (λ*n*. *snd* (*X n*))
  unfolding *Cauchy-def* **by** (*fast elim*: *le-less-trans* [*OF dist-snd-le*])

lemma *Cauchy-Pair*:
  assumes *Cauchy X* **and** *Cauchy Y*
  shows *Cauchy* (λ*n*. (*X n*, *Y n*))
proof (*rule metric-CauchyI*)
  fix *r* :: *real* **assume** *0* < *r*
  hence *0* < *r* / *sqrt 2* (**is** *0* < *?s*) **by** *simp*
  obtain *M* **where** *M*: ∀ *m*≥*M*. ∀ *n*≥*M*. *dist* (*X m*) (*X n*) < *?s*
    using *metric-CauchyD* [*OF* ‹*Cauchy X*› ‹*0* < *?s*›] **..**
  obtain *N* **where** *N*: ∀ *m*≥*N*. ∀ *n*≥*N*. *dist* (*Y m*) (*Y n*) < *?s*
    using *metric-CauchyD* [*OF* ‹*Cauchy Y*› ‹*0* < *?s*›] **..**
  have ∀ *m*≥*max M N*. ∀ *n*≥*max M N*. *dist* (*X m*, *Y m*) (*X n*, *Y n*) < *r*
    using *M N* **by** (*simp add*: *real-sqrt-sum-squares-less dist-Pair-Pair*)
  then show ∃ *n0*. ∀ *m*≥*n0*. ∀ *n*≥*n0*. *dist* (*X m*, *Y m*) (*X n*, *Y n*) < *r* **..**
qed

19.3 Product is a complete metric space

instance *prod* :: (*complete-space*, *complete-space*) *complete-space*
proof
  fix *X* :: *nat* ⇒ '*a* × '*b* **assume** *Cauchy X*
  have *1*: (λ*n*. *fst* (*X n*)) ⟶ *lim* (λ*n*. *fst* (*X n*))
    using *Cauchy-fst* [*OF* ‹*Cauchy X*›]
    by (*simp add*: *Cauchy-convergent-iff convergent-LIMSEQ-iff*)
  have *2*: (λ*n*. *snd* (*X n*)) ⟶ *lim* (λ*n*. *snd* (*X n*))
    using *Cauchy-snd* [*OF* ‹*Cauchy X*›]
    by (*simp add*: *Cauchy-convergent-iff convergent-LIMSEQ-iff*)
  have *X* ⟶ (*lim* (λ*n*. *fst* (*X n*)), *lim* (λ*n*. *snd* (*X n*)))
    using *tendsto-Pair* [*OF 1 2*] **by** *simp*
  then show *convergent X*
    by (*rule convergentI*)
qed

19.4 Product is a normed vector space

instantiation *prod* :: (*real-normed-vector*, *real-normed-vector*) *real-normed-vector*
begin

definition *norm-prod-def* [*code del*]:
  norm x = *sqrt* ((*norm* (*fst x*))^2 + (*norm* (*snd x*))^2)

definition *sgn-prod-def*:
sgn $(x{::}'a \times 'b) = scaleR\ (inverse\ (norm\ x))\ x$

lemma *norm-Pair*: $norm\ (a,\ b) = sqrt\ ((norm\ a)^2 + (norm\ b)^2)$
unfolding *norm-prod-def* **by** *simp*

instance
proof
fix r :: *real* **and** $x\ y$ :: $'a \times 'b$
show $norm\ x = 0 \longleftrightarrow x = 0$
unfolding *norm-prod-def*
by (*simp add*: *prod-eq-iff*)
show $norm\ (x + y) \le norm\ x + norm\ y$
unfolding *norm-prod-def*
apply (*rule order-trans* [*OF - real-sqrt-sum-squares-triangle-ineq*])
apply (*simp add*: *add-mono power-mono norm-triangle-ineq*)
done
show $norm\ (scaleR\ r\ x) = |r| * norm\ x$
unfolding *norm-prod-def*
apply (*simp add*: *power-mult-distrib*)
apply (*simp add*: *distrib-left* [*symmetric*])
apply (*simp add*: *real-sqrt-mult-distrib*)
done
show $sgn\ x = scaleR\ (inverse\ (norm\ x))\ x$
by (*rule sgn-prod-def*)
show $dist\ x\ y = norm\ (x - y)$
unfolding *dist-prod-def norm-prod-def*
by (*simp add*: *dist-norm*)
qed

end

declare [[*code abort*: *norm*::($'a$::*real-normed-vector*$*'b$::*real-normed-vector*) $\Rightarrow$ *real*]]

instance *prod* :: (*banach*, *banach*) *banach* **..**

19.4.1 Pair operations are linear

lemma *bounded-linear-fst*: *bounded-linear fst*
using *fst-add fst-scaleR*
by (*rule bounded-linear-intro* [**where** *K=1*], *simp add*: *norm-prod-def*)

lemma *bounded-linear-snd*: *bounded-linear snd*
using *snd-add snd-scaleR*
by (*rule bounded-linear-intro* [**where** *K=1*], *simp add*: *norm-prod-def*)

lemmas *bounded-linear-fst-comp* = *bounded-linear-fst*[*THEN bounded-linear-compose*]

lemmas *bounded-linear-snd-comp* = *bounded-linear-snd*[*THEN bounded-linear-compose*]

lemma *bounded-linear-Pair*:
assumes *f*: *bounded-linear f*
assumes *g*: *bounded-linear g*
shows *bounded-linear* $(\lambda x.\ (f\ x,\ g\ x))$
proof
interpret *f*: *bounded-linear f* **by** *fact*
interpret *g*: *bounded-linear g* **by** *fact*
fix *x y* **and** *r* :: *real*
show $(f\ (x + y),\ g\ (x + y)) = (f\ x,\ g\ x) + (f\ y,\ g\ y)$
by (*simp add*: *f.add g.add*)
show $(f\ (r *_R x),\ g\ (r *_R x)) = r *_R (f\ x,\ g\ x)$
by (*simp add*: *f.scaleR g.scaleR*)
obtain *Kf* **where** $0 < Kf$ **and** *norm-f*: $\bigwedge x.\ norm\ (f\ x) \leq norm\ x * Kf$
using *f.pos-bounded* **by** *fast*
obtain *Kg* **where** $0 < Kg$ **and** *norm-g*: $\bigwedge x.\ norm\ (g\ x) \leq norm\ x * Kg$
using *g.pos-bounded* **by** *fast*
have $\forall x.\ norm\ (f\ x,\ g\ x) \leq norm\ x * (Kf + Kg)$
apply (*rule allI*)
apply (*simp add*: *norm-Pair*)
apply (*rule order-trans* [*OF sqrt-add-le-add-sqrt*], *simp*, *simp*)
apply (*simp add*: *distrib-left*)
apply (*rule add-mono* [*OF norm-f norm-g*])
done
then show $\exists K.\ \forall x.\ norm\ (f\ x,\ g\ x) \leq norm\ x * K$ **..**
qed

19.4.2 Frechet derivatives involving pairs

lemma *has-derivative-Pair* [*derivative-intros*]:
assumes *f*: (*f has-derivative f'*) (*at x within s*) **and** *g*: (*g has-derivative g'*) (*at x within s*)
shows $((\lambda x.\ (f\ x,\ g\ x))$ *has-derivative* $(\lambda h.\ (f'\ h,\ g'\ h)))$ (*at x within s*)
proof (*rule has-derivativeI-sandwich*[*of 1*])
show *bounded-linear* $(\lambda h.\ (f'\ h,\ g'\ h))$
using *f g* **by** (*intro bounded-linear-Pair has-derivative-bounded-linear*)
let $?Rf = \lambda y.\ f\ y - f\ x - f'\ (y - x)$
let $?Rg = \lambda y.\ g\ y - g\ x - g'\ (y - x)$
let $?R = \lambda y.\ ((f\ y,\ g\ y) - (f\ x,\ g\ x) - (f'\ (y - x),\ g'\ (y - x)))$

show $((\lambda y.\ norm\ (?Rf\ y)\ /\ norm\ (y - x) + norm\ (?Rg\ y)\ /\ norm\ (y - x)) \longrightarrow 0)$ (*at x within s*)
using *f g* **by** (*intro tendsto-add-zero*) (*auto simp*: *has-derivative-iff-norm*)

fix *y* :: *'a* **assume** $y \neq x$
show $norm\ (?R\ y)\ /\ norm\ (y - x) \leq norm\ (?Rf\ y)\ /\ norm\ (y - x) + norm\ (?Rg\ y)\ /\ norm\ (y - x)$
unfolding *add-divide-distrib* [*symmetric*]
by (*simp add*: *norm-Pair divide-right-mono order-trans* [*OF sqrt-add-le-add-sqrt*])

qed *simp*

lemmas *has-derivative-fst* [*derivative-intros*] = *bounded-linear.has-derivative* [*OF bounded-linear-fst*]
lemmas *has-derivative-snd* [*derivative-intros*] = *bounded-linear.has-derivative* [*OF bounded-linear-snd*]

lemma *has-derivative-split* [*derivative-intros*]:
  $((\lambda p.\ f\ (fst\ p)\ (snd\ p))\ has\text{-}derivative\ f')\ F \Longrightarrow ((\lambda(a,\ b).\ f\ a\ b)\ has\text{-}derivative\ f')\ F$
  unfolding *split-beta'* **.**

19.5 Product is an inner product space

instantiation *prod* :: (*real-inner*, *real-inner*) *real-inner*
begin

definition *inner-prod-def*:
  inner x y = inner (*fst x*) (*fst y*) + *inner* (*snd x*) (*snd y*)

lemma *inner-Pair* [*simp*]: *inner* (*a*, *b*) (*c*, *d*) = *inner a c* + *inner b d*
  unfolding *inner-prod-def* **by** *simp*

instance
proof
  fix *r* :: *real*
  fix *x y z* :: *'a::real-inner* × *'b::real-inner*
  show *inner x y* = *inner y x*
    unfolding *inner-prod-def*
    by (*simp add*: *inner-commute*)
  show *inner* (*x* + *y*) *z* = *inner x z* + *inner y z*
    unfolding *inner-prod-def*
    by (*simp add*: *inner-add-left*)
  show *inner* (*scaleR r x*) *y* = *r* ∗ *inner x y*
    unfolding *inner-prod-def*
    by (*simp add*: *distrib-left*)
  show *0* ≤ *inner x x*
    unfolding *inner-prod-def*
    by (*intro add-nonneg-nonneg inner-ge-zero*)
  show *inner x x* = *0* ⟷ *x* = *0*
    unfolding *inner-prod-def prod-eq-iff*
    by (*simp add*: *add-nonneg-eq-0-iff*)
  show *norm x* = *sqrt* (*inner x x*)
    unfolding *norm-prod-def inner-prod-def*
    by (*simp add*: *power2-norm-eq-inner*)
qed

end

lemma *inner-Pair-0*: *inner x (0, b) = inner (snd x) b inner x (a, 0) = inner (fst x) a*
by (*cases x*, *simp*)+

lemma
fixes *x* :: *'a::real-normed-vector*
shows *norm-Pair1* [*simp*]: *norm (0,x) = norm x*
and *norm-Pair2* [*simp*]: *norm (x,0) = norm x*
by (*auto simp*: *norm-Pair*)

lemma *norm-commute*: *norm (x,y) = norm (y,x)*
by (*simp add*: *norm-Pair*)

lemma *norm-fst-le*: *norm x ≤ norm (x,y)*
by (*metis dist-fst-le fst-conv fst-zero norm-conv-dist*)

lemma *norm-snd-le*: *norm y ≤ norm (x,y)*
by (*metis dist-snd-le snd-conv snd-zero norm-conv-dist*)

end

20 Convexity in real vector spaces

theory *Convex*
imports *Product-Vector*
begin

20.1 Convexity

definition *convex* :: *'a::real-vector set ⇒ bool*
where *convex s* ⟷ (∀ *x*∈*s*. ∀ *y*∈*s*. ∀ *u*≥*0*. ∀ *v*≥*0*. *u* + *v* = *1* ⟶ $u *_R x + v *_R y$ ∈ *s*)

lemma *convexI*:
assumes ⋀*x y u v*. *x* ∈ *s* ⟹ *y* ∈ *s* ⟹ *0* ≤ *u* ⟹ *0* ≤ *v* ⟹ *u* + *v* = *1* ⟹ $u *_R x + v *_R y$ ∈ *s*
shows *convex s*
using *assms* **unfolding** *convex-def* **by** *fast*

lemma *convexD*:
assumes *convex s* **and** *x* ∈ *s* **and** *y* ∈ *s* **and** *0* ≤ *u* **and** *0* ≤ *v* **and** *u* + *v* = *1*
shows $u *_R x + v *_R y$ ∈ *s*
using *assms* **unfolding** *convex-def* **by** *fast*

lemma *convex-alt*:
convex s ⟷ (∀ *x*∈*s*. ∀ *y*∈*s*. ∀ *u*. *0* ≤ *u* ∧ *u* ≤ *1* ⟶ ($(1 - u) *_R x + u *_R y$) ∈ *s*)
(**is** - ⟷ *?alt*)
proof

```
  assume alt[rule-format]: ?alt
  {
    fix x y and u v :: real
    assume mem: x ∈ s y ∈ s
    assume 0 ≤ u 0 ≤ v
    moreover
    assume u + v = 1
    then have u = 1 − v by auto
    ultimately have u *R x + v *R y ∈ s
      using alt[OF mem] by auto
  }
  then show convex s
    unfolding convex-def by auto
qed (auto simp: convex-def)

lemma convexD-alt:
  assumes convex s a ∈ s b ∈ s 0 ≤ u u ≤ 1
  shows ((1 − u) *R a + u *R b) ∈ s
  using assms unfolding convex-alt by auto

lemma mem-convex-alt:
  assumes convex S x ∈ S y ∈ S u ≥ 0 v ≥ 0 u + v > 0
  shows ((u/(u+v)) *R x + (v/(u+v)) *R y) ∈ S
  apply (rule convexD)
  using assms
  apply (simp-all add: zero-le-divide-iff add-divide-distrib [symmetric])
  done

lemma convex-empty[intro,simp]: convex {}
  unfolding convex-def by simp

lemma convex-singleton[intro,simp]: convex {a}
  unfolding convex-def by (auto simp: scaleR-left-distrib[symmetric])

lemma convex-UNIV[intro,simp]: convex UNIV
  unfolding convex-def by auto

lemma convex-Inter: (∀ s∈f. convex s) =⇒ convex(⋂ f)
  unfolding convex-def by auto

lemma convex-Int: convex s =⇒ convex t =⇒ convex (s ∩ t)
  unfolding convex-def by auto

lemma convex-INT: ∀ i∈A. convex (B i) =⇒ convex (⋂ i∈A. B i)
  unfolding convex-def by auto

lemma convex-Times: convex s =⇒ convex t =⇒ convex (s × t)
  unfolding convex-def by auto
```

lemma *convex-halfspace-le*: *convex* {*x*. *inner a x* ≤ *b*}
  unfolding *convex-def*
  by (*auto simp*: *inner-add intro*!: *convex-bound-le*)

lemma *convex-halfspace-ge*: *convex* {*x*. *inner a x* ≥ *b*}
proof −
  have ∗: {*x*. *inner a x* ≥ *b*} = {*x*. *inner* (−*a*) *x* ≤ −*b*}
    by *auto*
  show *?thesis*
    unfolding ∗ **using** *convex-halfspace-le*[*of* −*a* −*b*] **by** *auto*
qed

lemma *convex-hyperplane*: *convex* {*x*. *inner a x* = *b*}
proof −
  have ∗: {*x*. *inner a x* = *b*} = {*x*. *inner a x* ≤ *b*} ∩ {*x*. *inner a x* ≥ *b*}
    by *auto*
  show *?thesis* **using** *convex-halfspace-le convex-halfspace-ge*
    by (*auto intro*!: *convex-Int simp*: ∗)
qed

lemma *convex-halfspace-lt*: *convex* {*x*. *inner a x* < *b*}
  unfolding *convex-def*
  by (*auto simp*: *convex-bound-lt inner-add*)

lemma *convex-halfspace-gt*: *convex* {*x*. *inner a x* > *b*}
  using *convex-halfspace-lt*[*of* −*a* −*b*] **by** *auto*

lemma *convex-real-interval* [*iff*]:
  fixes *a b* :: *real*
  shows *convex* {*a*..} **and** *convex* {..*b*}
    and *convex* {*a*<..} **and** *convex* {..<*b*}
    and *convex* {*a*..*b*} **and** *convex* {*a*<..*b*}
    and *convex* {*a*..<*b*} **and** *convex* {*a*<..<*b*}
proof −
  have {*a*..} = {*x*. *a* ≤ *inner 1 x*}
    by *auto*
  then show *1*: *convex* {*a*..}
    by (*simp only*: *convex-halfspace-ge*)
  have {..*b*} = {*x*. *inner 1 x* ≤ *b*}
    by *auto*
  then show *2*: *convex* {..*b*}
    by (*simp only*: *convex-halfspace-le*)
  have {*a*<..} = {*x*. *a* < *inner 1 x*}
    by *auto*
  then show *3*: *convex* {*a*<..}
    by (*simp only*: *convex-halfspace-gt*)
  have {..<*b*} = {*x*. *inner 1 x* < *b*}
    by *auto*
  then show *4*: *convex* {..<*b*}

by (*simp only*: *convex-halfspace-lt*)
have $\{a..b\} = \{a..\} \cap \{..b\}$
by *auto*
then show *convex* $\{a..b\}$
by (*simp only*: *convex-Int 1 2*)
have $\{a<..b\} = \{a<..\} \cap \{..b\}$
by *auto*
then show *convex* $\{a<..b\}$
by (*simp only*: *convex-Int 3 2*)
have $\{a..<b\} = \{a..\} \cap \{..<b\}$
by *auto*
then show *convex* $\{a..<b\}$
by (*simp only*: *convex-Int 1 4*)
have $\{a<..<b\} = \{a<..\} \cap \{..<b\}$
by *auto*
then show *convex* $\{a<..<b\}$
by (*simp only*: *convex-Int 3 4*)
qed

lemma *convex-Reals*: *convex* $\mathbb{R}$
by (*simp add*: *convex-def scaleR-conv-of-real*)

20.2 Explicit expressions for convexity in terms of arbitrary sums

lemma *convex-setsum*:
fixes *C* :: *'a::real-vector set*
assumes *finite s*
and *convex C*
and $(\sum\ i \in s.\ a\ i) = 1$
assumes $\bigwedge i.\ i \in s \Longrightarrow a\ i \geq 0$
and $\bigwedge i.\ i \in s \Longrightarrow y\ i \in C$
shows $(\sum\ j \in s.\ a\ j *_R y\ j) \in C$
using *assms*(*1*,*3*,*4*,*5*)
proof (*induct arbitrary*: *a set*: *finite*)
case *empty*
then show *?case* **by** *simp*
next
case (*insert i s*) **note** *IH* = *this*(*3*)
have $a\ i + setsum\ a\ s = 1$
and $0 \leq a\ i$
and $\forall j{\in}s.\ 0 \leq a\ j$
and $y\ i \in C$
and $\forall j{\in}s.\ y\ j \in C$
using *insert.hyps*(*1*,*2*) *insert.prems* **by** *simp-all*
then have $0 \leq setsum\ a\ s$
by (*simp add*: *setsum-nonneg*)
have $a\ i *_R y\ i + (\sum j{\in}s.\ a\ j *_R y\ j) \in C$
proof (*cases*)

```
    assume z: setsum a s = 0
    with ‹a i + setsum a s = 1› have a i = 1
      by simp
    from setsum-nonneg-0 [OF ‹finite s› - z] ‹∀j∈s. 0 ≤ a j› have ∀j∈s. a j = 0
      by simp
    show ?thesis using ‹a i = 1› and ‹∀j∈s. a j = 0› and ‹y i ∈ C›
      by simp
  next
    assume nz: setsum a s ≠ 0
    with ‹0 ≤ setsum a s› have 0 < setsum a s
      by simp
    then have (∑j∈s. (a j / setsum a s) *R y j) ∈ C
      using ‹∀j∈s. 0 ≤ a j› and ‹∀j∈s. y j ∈ C›
      by (simp add: IH setsum-divide-distrib [symmetric])
    from ‹convex C› and ‹y i ∈ C› and this and ‹0 ≤ a i›
      and ‹0 ≤ setsum a s› and ‹a i + setsum a s = 1›
    have a i *R y i + setsum a s *R (∑j∈s. (a j / setsum a s) *R y j) ∈ C
      by (rule convexD)
    then show ?thesis
      by (simp add: scaleR-setsum-right nz)
  qed
  then show ?case using ‹finite s› and ‹i ∉ s›
    by simp
qed

lemma convex:
  convex s ⟷ (∀(k::nat) u x. (∀i. 1≤i ∧ i≤k ⟶ 0 ≤ u i ∧ x i ∈s) ∧ (setsum
u {1..k} = 1)
      ⟶ setsum (λi. u i *R x i) {1..k} ∈ s)
proof safe
  fix k :: nat
  fix u :: nat ⇒ real
  fix x
  assume convex s
    ∀i. 1 ≤ i ∧ i ≤ k ⟶ 0 ≤ u i ∧ x i ∈ s
    setsum u {1..k} = 1
  with convex-setsum[of {1 .. k} s] show (∑j∈{1 .. k}. u j *R x j) ∈ s
    by auto
next
  assume *: ∀k u x. (∀ i :: nat. 1 ≤ i ∧ i ≤ k ⟶ 0 ≤ u i ∧ x i ∈ s) ∧ setsum
u {1..k} = 1
    ⟶ (∑i = 1..k. u i *R (x i :: 'a)) ∈ s
  {
    fix μ :: real
    fix x y :: 'a
    assume xy: x ∈ s y ∈ s
    assume mu: μ ≥ 0 μ ≤ 1
    let ?u = λi. if (i :: nat) = 1 then μ else 1 − μ
    let ?x = λi. if (i :: nat) = 1 then x else y
```

```
    have {1 :: nat .. 2} ∩ − {x. x = 1} = {2}
      by auto
    then have card: card ({1 :: nat .. 2} ∩ − {x. x = 1}) = 1
      by simp
    then have setsum ?u {1 .. 2} = 1
      using setsum.If-cases[of {(1 :: nat) .. 2} λ x. x = 1 λ x. μ λ x. 1 − μ]
      by auto
    with ∗[rule-format, of 2 ?u ?x] have s: (∑ j ∈ {1..2}. ?u j ∗R ?x j) ∈ s
      using mu xy by auto
    have grarr: (∑ j ∈ {Suc (Suc 0)..2}. ?u j ∗R ?x j) = (1 − μ) ∗R y
      using setsum-head-Suc[of Suc (Suc 0) 2 λ j. (1 − μ) ∗R y] by auto
    from setsum-head-Suc[of Suc 0 2 λ j. ?u j ∗R ?x j, simplified this]
    have (∑ j ∈ {1..2}. ?u j ∗R ?x j) = μ ∗R x + (1 − μ) ∗R y
      by auto
    then have (1 − μ) ∗R y + μ ∗R x ∈ s
      using s by (auto simp: add.commute)
  }
  then show convex s
    unfolding convex-alt by auto
qed


lemma convex-explicit:
  fixes s :: 'a::real-vector set
  shows convex s ⟷
    (∀ t u. finite t ∧ t ⊆ s ∧ (∀ x∈t. 0 ≤ u x) ∧ setsum u t = 1 ⟶ setsum (λx.
u x ∗R x) t ∈ s)
proof safe
  fix t
  fix u :: 'a ⇒ real
  assume convex s
    and finite t
    and t ⊆ s ∀ x∈t. 0 ≤ u x setsum u t = 1
  then show (∑ x∈t. u x ∗R x) ∈ s
    using convex-setsum[of t s u λ x. x] by auto
next
  assume ∗: ∀ t. ∀ u. finite t ∧ t ⊆ s ∧ (∀ x∈t. 0 ≤ u x) ∧
    setsum u t = 1 ⟶ (∑ x∈t. u x ∗R x) ∈ s
  show convex s
    unfolding convex-alt
  proof safe
    fix x y
    fix μ :: real
    assume ∗∗: x ∈ s y ∈ s 0 ≤ μ μ ≤ 1
    show (1 − μ) ∗R x + μ ∗R y ∈ s
    proof (cases x = y)
      case False
      then show ?thesis
        using ∗[rule-format, of {x, y} λ z. if z = x then 1 − μ else μ] ∗∗
```

```
        by auto
    next
      case True
      then show ?thesis
        using *[rule-format, of {x, y} λ z. 1] **
          by (auto simp: field-simps real-vector.scale-left-diff-distrib)
    qed
  qed
qed
```

```
lemma convex-finite:
  assumes finite s
  shows convex s ⟷ (∀ u. (∀ x∈s. 0 ≤ u x) ∧ setsum u s = 1 ⟶ setsum (λx.
u x *R x) s ∈ s)
  unfolding convex-explicit
proof safe
  fix t u
  assume sum: ∀ u. (∀ x∈s. 0 ≤ u x) ∧ setsum u s = 1 ⟶ (∑ x∈s. u x *R x)
∈ s
    and as: finite t t ⊆ s ∀ x∈t. 0 ≤ u x setsum u t = (1::real)
  have *: s ∩ t = t
    using as(2) by auto
  have if-distrib-arg: ⋀P f g x. (if P then f else g) x = (if P then f x else g x)
    by simp
  show (∑ x∈t. u x *R x) ∈ s
   using sum[THEN spec[where x=λx. if x∈t then u x else 0]] as *
   by (auto simp: assms setsum.If-cases if-distrib if-distrib-arg)
qed (erule-tac x=s in allE, erule-tac x=u in allE, auto)
```

20.3 Functions that are convex on a set

```
definition convex-on :: 'a::real-vector set ⇒ ('a ⇒ real) ⇒ bool
  where convex-on s f ⟷
    (∀ x∈s. ∀ y∈s. ∀ u≥0. ∀ v≥0. u + v = 1 ⟶ f (u *R x + v *R y) ≤ u * f x
+ v * f y)
```

```
lemma convex-onI [intro?]:
  assumes ⋀t x y. t > 0 ⟹ t < 1 ⟹ x ∈ A ⟹ y ∈ A ⟹
           f ((1 − t) *R x + t *R y) ≤ (1 − t) * f x + t * f y
  shows   convex-on A f
  unfolding convex-on-def
proof clarify
  fix x y u v assume A: x ∈ A y ∈ A (u::real) ≥ 0 v ≥ 0 u + v = 1
  from A(5) have [simp]: v = 1 − u by (simp add: algebra-simps)
  from A(1−4) show f (u *R x + v *R y) ≤ u * f x + v * f y using assms[of
u y x]
    by (cases u = 0 ∨ u = 1) (auto simp: algebra-simps)
qed
```

lemma *convex-on-linorderI* [*intro?*]:
  fixes A :: ('*a*::{*linorder*,*real-vector*}) *set*
  assumes $\bigwedge t\ x\ y.\ t > 0 \Longrightarrow t < 1 \Longrightarrow x \in A \Longrightarrow y \in A \Longrightarrow x < y \Longrightarrow$
$f\ ((1 - t) *_R x + t *_R y) \leq (1 - t) * f\ x + t * f\ y$
  shows *convex-on A f*
proof
  fix $t\ x\ y$ **assume** A: $x \in A\ y \in A\ (t::real) > 0\ t < 1$
  with *assms*[*of t x y*] *assms*[*of 1 − t y x*]
  show $f\ ((1 - t) *_R x + t *_R y) \leq (1 - t) * f\ x + t * f\ y$
    by (*cases x y rule*: *linorder-cases*) (*auto simp*: *algebra-simps*)
qed

lemma *convex-onD*:
  assumes *convex-on A f*
  shows $\bigwedge t\ x\ y.\ t \geq 0 \Longrightarrow t \leq 1 \Longrightarrow x \in A \Longrightarrow y \in A \Longrightarrow$
$f\ ((1 - t) *_R x + t *_R y) \leq (1 - t) * f\ x + t * f\ y$
  using *assms* **unfolding** *convex-on-def* **by** *auto*

lemma *convex-onD-Icc*:
  assumes *convex-on* $\{x..y\}$ $f\ x \leq (y$ :: - :: {*real-vector*,*preorder*})
  shows $\bigwedge t.\ t \geq 0 \Longrightarrow t \leq 1 \Longrightarrow$
$f\ ((1 - t) *_R x + t *_R y) \leq (1 - t) * f\ x + t * f\ y$
  using *assms*(*2*) **by** (*intro convex-onD*[*OF assms*(*1*)]) *simp-all*

lemma *convex-on-subset*: *convex-on t f* $\Longrightarrow s \subseteq t \Longrightarrow$ *convex-on s f*
  unfolding *convex-on-def* **by** *auto*

lemma *convex-on-add* [*intro*]:
  assumes *convex-on s f*
    and *convex-on s g*
  shows *convex-on s* $(\lambda x.\ f\ x + g\ x)$
proof −
  {
    fix $x\ y$
    assume $x \in s\ y \in s$
    moreover
    fix $u\ v$ :: *real*
    assume $0 \leq u\ 0 \leq v\ u + v = 1$
    ultimately
    have $f\ (u *_R x + v *_R y) + g\ (u *_R x + v *_R y) \leq (u * f\ x + v * f\ y) + (u * g\ x + v * g\ y)$
      using *assms* **unfolding** *convex-on-def* **by** (*auto simp*: *add-mono*)
    then have $f\ (u *_R x + v *_R y) + g\ (u *_R x + v *_R y) \leq u * (f\ x + g\ x) + v * (f\ y + g\ y)$
      by (*simp add*: *field-simps*)
  }
  then show *?thesis*
    unfolding *convex-on-def* **by** *auto*
qed

lemma *convex-on-cmul* [*intro*]:
  fixes c :: *real*
  assumes $0 \leq c$
    and *convex-on s f*
  shows *convex-on s* ($\lambda x.\ c * f\ x$)
proof −
  have ∗: $\bigwedge u\ c\ fx\ v\ fy$ :: *real*. $u * (c * fx) + v * (c * fy) = c * (u * fx + v * fy)$
    by (*simp add*: *field-simps*)
  show *?thesis* **using** *assms*(*2*) **and** *mult-left-mono* [*OF - assms*(*1*)]
    unfolding *convex-on-def* **and** ∗ **by** *auto*
qed

lemma *convex-lower*:
  assumes *convex-on s f*
    and $x \in s$
    and $y \in s$
    and $0 \leq u$
    and $0 \leq v$
    and $u + v = 1$
  shows $f\ (u *_R x + v *_R y) \leq max\ (f\ x)\ (f\ y)$
proof −
  let $?m = max\ (f\ x)\ (f\ y)$
  have $u * f\ x + v * f\ y \leq u * max\ (f\ x)\ (f\ y) + v * max\ (f\ x)\ (f\ y)$
    using *assms*(*4*,*5*) **by** (*auto simp*: *mult-left-mono add-mono*)
  also have $\ldots = max\ (f\ x)\ (f\ y)$
    using *assms*(*6*) **by** (*simp add*: *distrib-right* [*symmetric*])
  finally show *?thesis*
    using *assms* **unfolding** *convex-on-def* **by** *fastforce*
qed

lemma *convex-on-dist* [*intro*]:
  fixes *s* :: *'a*::*real-normed-vector set*
  shows *convex-on s* ($\lambda x.\ dist\ a\ x$)
proof (*auto simp*: *convex-on-def dist-norm*)
  fix *x y*
  assume $x \in s\ y \in s$
  fix *u v* :: *real*
  assume $0 \leq u$
  assume $0 \leq v$
  assume $u + v = 1$
  have $a = u *_R a + v *_R a$
    unfolding *scaleR-left-distrib*[*symmetric*] **and** ‹$u + v = 1$› **by** *simp*
  then have ∗: $a - (u *_R x + v *_R y) = (u *_R (a - x)) + (v *_R (a - y))$
    by (*auto simp*: *algebra-simps*)
  show $norm\ (a - (u *_R x + v *_R y)) \leq u * norm\ (a - x) + v * norm\ (a - y)$
    unfolding ∗ **using** *norm-triangle-ineq*[*of* $u *_R (a - x)\ v *_R (a - y)$]
    using ‹$0 \leq u$› ‹$0 \leq v$› **by** *auto*
qed

20.4 Arithmetic operations on sets preserve convexity

lemma *convex-linear-image*:
  assumes *linear f*
    and *convex s*
  shows *convex* (*f* ‘ *s*)
proof –
  interpret *f*: *linear f* **by** *fact*
  from ‹*convex s*› **show** *convex* (*f* ‘ *s*)
    by (*simp add*: *convex-def f.scaleR* [*symmetric*] *f.add* [*symmetric*])
qed

lemma *convex-linear-vimage*:
  assumes *linear f*
    and *convex s*
  shows *convex* (*f* –‘ *s*)
proof –
  interpret *f*: *linear f* **by** *fact*
  from ‹*convex s*› **show** *convex* (*f* –‘ *s*)
    by (*simp add*: *convex-def f.add f.scaleR*)
qed

lemma *convex-scaling*:
  assumes *convex s*
  shows *convex* (($\lambda x.\ c *_R x$) ‘ *s*)
proof –
  have *linear* ($\lambda x.\ c *_R x$)
    by (*simp add*: *linearI scaleR-add-right*)
  then show *?thesis*
    using ‹*convex s*› **by** (*rule convex-linear-image*)
qed

lemma *convex-scaled*:
  assumes *convex s*
  shows *convex* (($\lambda x.\ x *_R c$) ‘ *s*)
proof –
  have *linear* ($\lambda x.\ x *_R c$)
    by (*simp add*: *linearI scaleR-add-left*)
  then show *?thesis*
    using ‹*convex s*› **by** (*rule convex-linear-image*)
qed

lemma *convex-negations*:
  assumes *convex s*
  shows *convex* (($\lambda x.\ -\ x$) ‘ *s*)
proof –
  have *linear* ($\lambda x.\ -\ x$)
    by (*simp add*: *linearI*)
  then show *?thesis*
    using ‹*convex s*› **by** (*rule convex-linear-image*)

qed

lemma *convex-sums*:
  assumes *convex s*
    and *convex t*
  shows *convex* $\{x + y|\ x\ y.\ x \in s \wedge y \in t\}$
proof –
  have *linear* $(\lambda(x, y).\ x + y)$
    by (*auto intro*: *linearI simp*: *scaleR-add-right*)
  with *assms* **have** *convex* $((\lambda(x, y).\ x + y)$ ‘ $(s \times t))$
    by (*intro convex-linear-image convex-Times*)
  also have $((\lambda(x, y).\ x + y)$ ‘ $(s \times t)) = \{x + y|\ x\ y.\ x \in s \wedge y \in t\}$
    by *auto*
  finally show *?thesis* **.**
qed

lemma *convex-differences*:
  assumes *convex s convex t*
  shows *convex* $\{x - y|\ x\ y.\ x \in s \wedge y \in t\}$
proof –
  have $\{x - y|\ x\ y.\ x \in s \wedge y \in t\} = \{x + y\ |x\ y.\ x \in s \wedge y \in$ *uminus* ‘ $t\}$
    by (*auto simp*: *diff-conv-add-uminus simp del*: *add-uminus-conv-diff*)
  then show *?thesis*
    using *convex-sums*[*OF assms*(*1*) *convex-negations*[*OF assms*(*2*)]] **by** *auto*
qed

lemma *convex-translation*:
  assumes *convex s*
  shows *convex* $((\lambda x.\ a + x)$ ‘ $s)$
proof –
  have $\{a + y\ |y.\ y \in s\} = (\lambda x.\ a + x)$ ‘ s
    by *auto*
  then show *?thesis*
    using *convex-sums*[*OF convex-singleton*[*of a*] *assms*] **by** *auto*
qed

lemma *convex-affinity*:
  assumes *convex s*
  shows *convex* $((\lambda x.\ a + c *_R x)$ ‘ $s)$
proof –
  have $(\lambda x.\ a + c *_R x)$ ‘ $s = op + a$ ‘ $op *_R c$ ‘ s
    by *auto*
  then show *?thesis*
    using *convex-translation*[*OF convex-scaling*[*OF assms*], *of a c*] **by** *auto*
qed

lemma *pos-is-convex*: *convex* $\{0 :: real <..\}$
  unfolding *convex-alt*
proof *safe*

```
  fix y x μ :: real
  assume *: y > 0 x > 0 μ ≥ 0 μ ≤ 1
  {
    assume μ = 0
    then have μ *R x + (1 − μ) *R y = y by simp
    then have μ *R x + (1 − μ) *R y > 0 using * by simp
  }
  moreover
  {
    assume μ = 1
    then have μ *R x + (1 − μ) *R y > 0 using * by simp
  }
  moreover
  {
    assume μ ≠ 1 μ ≠ 0
    then have μ > 0 (1 − μ) > 0 using * by auto
    then have μ *R x + (1 − μ) *R y > 0 using *
      by (auto simp: add-pos-pos)
  }
  ultimately show (1 − μ) *R y + μ *R x > 0
    using assms by fastforce
qed

lemma convex-on-setsum:
  fixes a :: 'a ⇒ real
    and y :: 'a ⇒ 'b::real-vector
    and f :: 'b ⇒ real
  assumes finite s s ≠ {}
    and convex-on C f
    and convex C
    and (∑ i ∈ s. a i) = 1
    and ⋀i. i ∈ s ⟹ a i ≥ 0
    and ⋀i. i ∈ s ⟹ y i ∈ C
  shows f (∑ i ∈ s. a i *R y i) ≤ (∑ i ∈ s. a i * f (y i))
  using assms
proof (induct s arbitrary: a rule: finite-ne-induct)
  case (singleton i)
  then have ai: a i = 1 by auto
  then show ?case by auto
next
  case (insert i s)
  then have convex-on C f by simp
  from this[unfolded convex-on-def, rule-format]
  have conv: ⋀x y μ. x ∈ C ⟹ y ∈ C ⟹ 0 ≤ μ ⟹ μ ≤ 1 ⟹
      f (μ *R x + (1 − μ) *R y) ≤ μ * f x + (1 − μ) * f y
    by simp
  show ?case
  proof (cases a i = 1)
    case True
```

then have ($\sum$ $j \in s.$ $a\ j$) = 0
  using *insert* **by** *auto*
then have $\bigwedge j.\ j \in s \Longrightarrow a\ j = 0$
  using *insert* **by** (*fastforce simp*: *setsum-nonneg-eq-0-iff*)
then show *?thesis*
  using *insert* **by** *auto*
next
  case *False*
  from *insert* **have** *yai*: $y\ i \in C\ a\ i \geq 0$
    by *auto*
  have *fis*: *finite* (*insert i s*)
    using *insert* **by** *auto*
  then have *ai1*: $a\ i \leq 1$
    using *setsum-nonneg-leq-bound*[*of insert i s a*] *insert* **by** *simp*
  then have $a\ i < 1$
    using *False* **by** *auto*
  then have *i0*: $1 - a\ i > 0$
    by *auto*
  let $?a = \lambda j.\ a\ j\ /\ (1 - a\ i)$
  have *a-nonneg*: $?a\ j \geq 0$ **if** $j \in s$ **for** j
    using *i0 insert that* **by** *fastforce*
  have ($\sum$ $j \in$ *insert* $i\ s.$ $a\ j$) = 1
    using *insert* **by** *auto*
  then have ($\sum$ $j \in s.$ $a\ j$) = $1 - a\ i$
    using *setsum.insert insert* **by** *fastforce*
  then have ($\sum$ $j \in s.$ $a\ j$) / ($1 - a\ i$) = 1
    using *i0* **by** *auto*
  then have *a1*: ($\sum$ $j \in s.$ $?a\ j$) = 1
    unfolding *setsum-divide-distrib* **by** *simp*
  have *convex C* **using** *insert* **by** *auto*
  then have *asum*: ($\sum$ $j \in s.$ $?a\ j *_R y\ j$) $\in C$
    using *insert convex-setsum*[*OF* ‹*finite s*›
      ‹*convex C*› *a1 a-nonneg*] **by** *auto*
  have *asum-le*: f ($\sum$ $j \in s.$ $?a\ j *_R y\ j$) $\leq$ ($\sum$ $j \in s.$ $?a\ j * f\ (y\ j)$)
    using *a-nonneg a1 insert* **by** *blast*
  have f ($\sum$ $j \in$ *insert* $i\ s.$ $a\ j *_R y\ j$) = f (($\sum$ $j \in s.$ $a\ j *_R y\ j$) + $a\ i *_R y\ i$)
    using *setsum.insert*[*of s i* λ $j.$ $a\ j *_R y\ j$, *OF* ‹*finite s*› ‹$i \notin s$›] *insert*
    by (*auto simp only*: *add.commute*)
  also have ... = f ((($1 - a\ i$) $*$ *inverse* ($1 - a\ i$)) $*_R$ ($\sum$ $j \in s.$ $a\ j *_R y\ j$)
$+\ a\ i *_R y\ i$)
    using *i0* **by** *auto*
  also have ... = f (($1 - a\ i$) $*_R$ ($\sum$ $j \in s.$ ($a\ j$ $*$ *inverse* ($1 - a\ i$)) $*_R y\ j$)
$+\ a\ i *_R y\ i$)
    using *scaleR-right.setsum*[*of inverse* ($1 - a\ i$) λ $j.$ $a\ j *_R y\ j\ s$, *symmetric*]
    by (*auto simp*: *algebra-simps*)
  also have ... = f (($1 - a\ i$) $*_R$ ($\sum$ $j \in s.$ $?a\ j *_R y\ j$) + $a\ i *_R y\ i$)
    by (*auto simp*: *divide-inverse*)
  also have ... $\leq$ ($1 - a\ i$) $*_R f$ (($\sum$ $j \in s.$ $?a\ j *_R y\ j$)) + $a\ i * f\ (y\ i)$
    using *conv*[*of y i* ($\sum$ $j \in s.$ $?a\ j *_R y\ j$) $a\ i$, *OF yai*(*1*) *asum yai*(*2*) *ai1*]

```
    by (auto simp: add.commute)
  also have ... ≤ (1 − a i) ∗ (∑ j ∈ s. ?a j ∗ f (y j)) + a i ∗ f (y i)
    using add-right-mono[OF mult-left-mono[of - - 1 − a i,
      OF asum-le less-imp-le[OF i0]], of a i ∗ f (y i)] by simp
  also have ... = (∑ j ∈ s. (1 − a i) ∗ ?a j ∗ f (y j)) + a i ∗ f (y i)
    unfolding setsum-right-distrib[of 1 − a i λ j. ?a j ∗ f (y j)] using i0 by
auto
  also have ... = (∑ j ∈ s. a j ∗ f (y j)) + a i ∗ f (y i)
    using i0 by auto
  also have ... = (∑ j ∈ insert i s. a j ∗ f (y j))
    using insert by auto
  finally show ?thesis
    by simp
 qed
qed


lemma convex-on-alt:
 fixes C :: 'a::real-vector set
 assumes convex C
 shows convex-on C f ⟷
  (∀ x ∈ C. ∀ y ∈ C. ∀ μ :: real. μ ≥ 0 ∧ μ ≤ 1 ⟶
    f (μ ∗R x + (1 − μ) ∗R y) ≤ μ ∗ f x + (1 − μ) ∗ f y)
proof safe
 fix x y
 fix μ :: real
 assume ∗: convex-on C f x ∈ C y ∈ C 0 ≤ μ μ ≤ 1
 from this[unfolded convex-on-def, rule-format]
 have ⋀u v. 0 ≤ u ⟹ 0 ≤ v ⟹ u + v = 1 ⟹ f (u ∗R x + v ∗R y) ≤ u ∗ f
x + v ∗ f y
  by auto
 from this[of μ 1 − μ, simplified] ∗
 show f (μ ∗R x + (1 − μ) ∗R y) ≤ μ ∗ f x + (1 − μ) ∗ f y
  by auto
next
 assume ∗: ∀ x∈C. ∀ y∈C. ∀ μ. 0 ≤ μ ∧ μ ≤ 1 ⟶
  f (μ ∗R x + (1 − μ) ∗R y) ≤ μ ∗ f x + (1 − μ) ∗ f y
 {
  fix x y
  fix u v :: real
  assume ∗∗: x ∈ C y ∈ C u ≥ 0 v ≥ 0 u + v = 1
  then have[simp]: 1 − u = v by auto
  from ∗[rule-format, of x y u]
  have f (u ∗R x + v ∗R y) ≤ u ∗ f x + v ∗ f y
    using ∗∗ by auto
 }
 then show convex-on C f
  unfolding convex-on-def by auto
qed
```

lemma *convex-on-diff*:
  fixes f :: *real* ⇒ *real*
  assumes f: *convex-on* $I\ f$
    and I: $x \in I\ y \in I$
    and t: $x < t\ t < y$
  shows $(f\ x - f\ t) / (x - t) \le (f\ x - f\ y) / (x - y)$
    and $(f\ x - f\ y) / (x - y) \le (f\ t - f\ y) / (t - y)$
proof −
  def $a \equiv (t - y) / (x - y)$
  with t **have** $0 \le a\ 0 \le 1 - a$
    by (*auto simp*: *field-simps*)
  with f ‹$x \in I$› ‹$y \in I$› **have** *cvx*: $f\ (a * x + (1 - a) * y) \le a * f\ x + (1 - a) * f\ y$
    by (*auto simp*: *convex-on-def*)
  have $a * x + (1 - a) * y = a * (x - y) + y$
    by (*simp add*: *field-simps*)
  also have ... $= t$
    unfolding *a-def* **using** ‹$x < t$› ‹$t < y$› **by** *simp*
  finally have $f\ t \le a * f\ x + (1 - a) * f\ y$
    using *cvx* **by** *simp*
  also have ... $= a * (f\ x - f\ y) + f\ y$
    by (*simp add*: *field-simps*)
  finally have $f\ t - f\ y \le a * (f\ x - f\ y)$
    by *simp*
  with t **show** $(f\ x - f\ t) / (x - t) \le (f\ x - f\ y) / (x - y)$
    by (*simp add*: *le-divide-eq divide-le-eq field-simps a-def*)
  with t **show** $(f\ x - f\ y) / (x - y) \le (f\ t - f\ y) / (t - y)$
    by (*simp add*: *le-divide-eq divide-le-eq field-simps*)
qed

lemma *pos-convex-function*:
  fixes f :: *real* ⇒ *real*
  assumes *convex* C
    and *leq*: $\bigwedge x\ y.\ x \in C \Longrightarrow y \in C \Longrightarrow f'\ x * (y - x) \le f\ y - f\ x$
  shows *convex-on* $C\ f$
  unfolding *convex-on-alt*[*OF assms*(*1*)]
  using *assms*
proof *safe*
  fix $x\ y\ \mu$ :: *real*
  let $?x = \mu *_R x + (1 - \mu) *_R y$
  assume ∗: *convex* $C\ x \in C\ y \in C\ \mu \ge 0\ \mu \le 1$
  then have $1 - \mu \ge 0$ **by** *auto*
  then have *xpos*: $?x \in C$
    using ∗ **unfolding** *convex-alt* **by** *fastforce*
  have *geq*: $\mu * (f\ x - f\ ?x) + (1 - \mu) * (f\ y - f\ ?x) \ge$
      $\mu * f'\ ?x * (x - ?x) + (1 - \mu) * f'\ ?x * (y - ?x)$
    using *add-mono*[*OF mult-left-mono*[*OF leq*[*OF xpos* ∗(*2*)] ‹$\mu \ge 0$›]
      mult-left-mono[*OF leq*[*OF xpos* ∗(*3*)] ‹$1 - \mu \ge 0$›]]
    by *auto*

```
  then have μ ∗ f x + (1 − μ) ∗ f y − f ?x ≥ 0
    by (auto simp: field-simps)
  then show f (μ ∗R x + (1 − μ) ∗R y) ≤ μ ∗ f x + (1 − μ) ∗ f y
    using convex-on-alt by auto
qed

lemma atMostAtLeast-subset-convex:
  fixes C :: real set
  assumes convex C
    and x ∈ C y ∈ C x < y
  shows {x .. y} ⊆ C
proof safe
  fix z assume z: z ∈ {x .. y}
  have less: z ∈ C if ∗: x < z z < y
  proof −
    let ?μ = (y − z) / (y − x)
    have 0 ≤ ?μ ?μ ≤ 1
      using assms ∗ by (auto simp: field-simps)
    then have comb: ?μ ∗ x + (1 − ?μ) ∗ y ∈ C
      using assms iffD1[OF convex-alt, rule-format, of C y x ?μ]
      by (simp add: algebra-simps)
    have ?μ ∗ x + (1 − ?μ) ∗ y = (y − z) ∗ x / (y − x) + (1 − (y − z) / (y −
x)) ∗ y
      by (auto simp: field-simps)
    also have . . . = ((y − z) ∗ x + (y − x − (y − z)) ∗ y) / (y − x)
      using assms unfolding add-divide-distrib by (auto simp: field-simps)
    also have . . . = z
      using assms by (auto simp: field-simps)
    finally show ?thesis
      using comb by auto
  qed
  show z ∈ C using z less assms
    unfolding atLeastAtMost-iff le-less by auto
qed

lemma f''-imp-f':
  fixes f :: real ⇒ real
  assumes convex C
    and f': ⋀x. x ∈ C ⟹ DERIV f x :> (f' x)
    and f'': ⋀x. x ∈ C ⟹ DERIV f' x :> (f'' x)
    and pos: ⋀x. x ∈ C ⟹ f'' x ≥ 0
    and x ∈ C y ∈ C
  shows f' x ∗ (y − x) ≤ f y − f x
  using assms
proof −
  {
    fix x y :: real
    assume ∗: x ∈ C y ∈ C y > x
    then have ge: y − x > 0 y − x ≥ 0
```

by *auto*
from ∗ **have** *le*: $x - y < 0$ $x - y \leq 0$
by *auto*
then obtain *z1* **where** *z1*: $z1 > x$ $z1 < y$ $f\ y - f\ x = (y - x) * f'\ z1$
using *subsetD*[*OF atMostAtLeast-subset-convex*[*OF* ‹*convex C*› ‹$x \in C$› ‹$y \in C$› ‹$x < y$›],
THEN f', *THEN MVT2*[*OF* ‹$x < y$›, *rule-format*, *unfolded atLeastAtMost-iff*[*symmetric*]]]
by *auto*
then have $z1 \in C$
using *atMostAtLeast-subset-convex* ‹*convex C*› ‹$x \in C$› ‹$y \in C$› ‹$x < y$›
by *fastforce*
from *z1* **have** *z1'*: $f\ x - f\ y = (x - y) * f'\ z1$
by (*simp add*: *field-simps*)
obtain *z2* **where** *z2*: $z2 > x$ $z2 < z1$ $f'\ z1 - f'\ x = (z1 - x) * f''\ z2$
using *subsetD*[*OF atMostAtLeast-subset-convex*[*OF* ‹*convex C*› ‹$x \in C$› ‹$z1 \in C$› ‹$x < z1$›],
THEN f'', *THEN MVT2*[*OF* ‹$x < z1$›, *rule-format*, *unfolded atLeastAtMost-iff*[*symmetric*]]] *z1*
by *auto*
obtain *z3* **where** *z3*: $z3 > z1$ $z3 < y$ $f'\ y - f'\ z1 = (y - z1) * f''\ z3$
using *subsetD*[*OF atMostAtLeast-subset-convex*[*OF* ‹*convex C*› ‹$z1 \in C$› ‹$y \in C$› ‹$z1 < y$›],
THEN f'', *THEN MVT2*[*OF* ‹$z1 < y$›, *rule-format*, *unfolded atLeastAtMost-iff*[*symmetric*]]] *z1*
by *auto*
have $f'\ y - (f\ x - f\ y) / (x - y) = f'\ y - f'\ z1$
using ∗ *z1'* **by** *auto*
also have $\ldots = (y - z1) * f''\ z3$
using *z3* **by** *auto*
finally have *cool'*: $f'\ y - (f\ x - f\ y) / (x - y) = (y - z1) * f''\ z3$
by *simp*
have *A'*: $y - z1 \geq 0$
using *z1* **by** *auto*
have $z3 \in C$
using *z3* ∗ *atMostAtLeast-subset-convex* ‹*convex C*› ‹$x \in C$› ‹$z1 \in C$› ‹$x < z1$›
by *fastforce*
then have *B'*: $f''\ z3 \geq 0$
using *assms* **by** *auto*
from *A' B'* **have** $(y - z1) * f''\ z3 \geq 0$
by *auto*
from *cool' this* **have** $f'\ y - (f\ x - f\ y) / (x - y) \geq 0$
by *auto*
from *mult-right-mono-neg*[*OF this le*(*2*)]
have $f'\ y * (x - y) - (f\ x - f\ y) / (x - y) * (x - y) \leq 0 * (x - y)$
by (*simp add*: *algebra-simps*)
then have $f'\ y * (x - y) - (f\ x - f\ y) \leq 0$
using *le* **by** *auto*
then have *res*: $f'\ y * (x - y) \leq f\ x - f\ y$

    by *auto*
  have $(f\,y - f\,x) / (y - x) - f'\,x = f'\,z1 - f'\,x$
    using ∗ *z1* **by** *auto*
  also have ... $= (z1 - x) * f''\,z2$
    using *z2* **by** *auto*
  finally have *cool*: $(f\,y - f\,x) / (y - x) - f'\,x = (z1 - x) * f''\,z2$
    by *simp*
  have *A*: $z1 - x \geq 0$
    using *z1* **by** *auto*
  have $z2 \in C$
    using *z2 z1* ∗ *atMostAtLeast-subset-convex* ‹*convex C*› ‹$z1 \in C$› ‹$y \in C$› ‹$z1$
$< y$›
    by *fastforce*
  then have *B*: $f''\,z2 \geq 0$
    using *assms* **by** *auto*
  from *A B* **have** $(z1 - x) * f''\,z2 \geq 0$
    by *auto*
  with *cool* **have** $(f\,y - f\,x) / (y - x) - f'\,x \geq 0$
    by *auto*
  from *mult-right-mono*[*OF this ge*(*2*)]
  have $(f\,y - f\,x) / (y - x) * (y - x) - f'\,x * (y - x) \geq 0 * (y - x)$
    by (*simp add*: *algebra-simps*)
  then have $f\,y - f\,x - f'\,x * (y - x) \geq 0$
    using *ge* **by** *auto*
  then have $f\,y - f\,x \geq f'\,x * (y - x)$ $f'\,y * (x - y) \leq f\,x - f\,y$
    using *res* **by** *auto*
 } **note** *less-imp* = *this*
 {
  fix *x y* :: *real*
  assume $x \in C$ $y \in C$ $x \neq y$
  then have$f\,y - f\,x \geq f'\,x * (y - x)$
  unfolding *neq-iff* **using** *less-imp* **by** *auto*
 }
 moreover
 {
  fix *x y* :: *real*
  assume $x \in C$ $y \in C$ $x = y$
  then have $f\,y - f\,x \geq f'\,x * (y - x)$ **by** *auto*
 }
 ultimately show *?thesis* **using** *assms* **by** *blast*
qed

lemma *f''-ge0-imp-convex*:
 fixes *f* :: *real* ⇒ *real*
 assumes *conv*: *convex C*
  and *f'*: $\bigwedge x.\ x \in C \Longrightarrow$ *DERIV f x* :> $(f'\,x)$
  and *f''*: $\bigwedge x.\ x \in C \Longrightarrow$ *DERIV f' x* :> $(f''\,x)$
  and *pos*: $\bigwedge x.\ x \in C \Longrightarrow f''\,x \geq 0$
 shows *convex-on C f*

using *f''-imp-f'[OF conv f' f'' pos] assms pos-convex-function*
by *fastforce*

lemma *minus-log-convex*:
fixes *b* :: *real*
assumes *b* > *1*
shows *convex-on* {*0* <..} (λ *x*. − *log b x*)
proof −
have ⋀*z*. *z* > *0* ⟹ *DERIV* (*log b*) *z* :> *1* / (*ln b* ∗ *z*)
using *DERIV-log* **by** *auto*
then have *f'*: ⋀*z*. *z* > *0* ⟹ *DERIV* (λ *z*. − *log b z*) *z* :> − *1* / (*ln b* ∗ *z*)
by (*auto simp*: *DERIV-minus*)
have ⋀*z* :: *real*. *z* > *0* ⟹ *DERIV inverse z* :> − (*inverse z* ˆ *Suc* (*Suc 0*))
using *less-imp-neq*[*THEN not-sym*, *THEN DERIV-inverse*] **by** *auto*
from *this*[*THEN DERIV-cmult*, *of* - − *1* / *ln b*]
have ⋀*z* :: *real*. *z* > *0* ⟹
DERIV (λ *z*. (− *1* / *ln b*) ∗ *inverse z*) *z* :> (− *1* / *ln b*) ∗ (− (*inverse z* ˆ *Suc* (*Suc 0*)))
by *auto*
then have *f''0*: ⋀*z*::*real*. *z* > *0* ⟹
DERIV (λ *z*. − *1* / (*ln b* ∗ *z*)) *z* :> *1* / (*ln b* ∗ *z* ∗ *z*)
unfolding *inverse-eq-divide* **by** (*auto simp*: *mult.assoc*)
have *f''-ge0*: ⋀*z*::*real*. *z* > *0* ⟹ *1* / (*ln b* ∗ *z* ∗ *z*) ≥ *0*
using ‹*b* > *1*› **by** (*auto intro*!: *less-imp-le*)
from *f''-ge0-imp-convex*[*OF pos-is-convex*,
unfolded greaterThan-iff, *OF f' f''0 f''-ge0*]
show *?thesis* **by** *auto*
qed

20.5 Convexity of real functions

lemma *convex-on-realI*:
assumes *connected A*
assumes ⋀*x*. *x* ∈ *A* ⟹ (*f has-real-derivative f' x*) (*at x*)
assumes ⋀*x y*. *x* ∈ *A* ⟹ *y* ∈ *A* ⟹ *x* ≤ *y* ⟹ *f' x* ≤ *f' y*
shows *convex-on A f*
proof (*rule convex-on-linorderI*)
fix *t x y* :: *real*
assume *t*: *t* > *0* *t* < *1* **and** *xy*: *x* ∈ *A* *y* ∈ *A* *x* < *y*
def *z* ≡ (*1* − *t*) ∗ *x* + *t* ∗ *y*
with ‹*connected A*› **and** *xy* **have** *ivl*: {*x*..*y*} ⊆ *A* **using** *connected-contains-Icc* **by** *blast*

from *xy t* **have** *xz*: *z* > *x* **by** (*simp add*: *z-def algebra-simps*)
have *y* − *z* = (*1* − *t*) ∗ (*y* − *x*) **by** (*simp add*: *z-def algebra-simps*)
also from *xy t* **have** ... > *0* **by** (*intro mult-pos-pos*) *simp-all*
finally have *yz*: *z* < *y* **by** *simp*

from *assms xz yz ivl t* **have** ∃ξ. ξ > *x* ∧ ξ < *z* ∧ *f z* − *f x* = (*z* − *x*) ∗ *f' ξ*

by (*intro MVT2*) (*auto intro*!: *assms*(*2*))
then obtain ξ **where** ξ: $\xi > x$ $\xi < z$ $f' \xi = (f z - f x) / (z - x)$ **by** *auto*
from *assms xz yz ivl t* **have** $\exists \eta.\ \eta > z \wedge \eta < y \wedge f y - f z = (y - z) * f' \eta$
by (*intro MVT2*) (*auto intro*!: *assms*(*2*))
then obtain η **where** η: $\eta > z$ $\eta < y$ $f' \eta = (f y - f z) / (y - z)$ **by** *auto*

from $\eta(3)$ **have** $(f y - f z) / (y - z) = f' \eta$ **..**
also from ξ η *ivl* **have** $\xi \in A$ $\eta \in A$ **by** *auto*
with ξ η **have** $f' \eta \geq f' \xi$ **by** (*intro assms*(*3*)) *auto*
also from $\xi(3)$ **have** $f' \xi = (f z - f x) / (z - x)$ **.**
finally have $(f y - f z) * (z - x) \geq (f z - f x) * (y - z)$
using *xz yz* **by** (*simp add*: *field-simps*)
also have $z - x = t * (y - x)$ **by** (*simp add*: *z-def algebra-simps*)
also have $y - z = (1 - t) * (y - x)$ **by** (*simp add*: *z-def algebra-simps*)
finally have $(f y - f z) * t \geq (f z - f x) * (1 - t)$ **using** *xy* **by** *simp*
thus $(1 - t) * f x + t * f y \geq f ((1 - t) *_R x + t *_R y)$
by (*simp add*: *z-def algebra-simps*)
qed

lemma *convex-on-inverse*:
assumes $A \subseteq \{0<..\}$
shows *convex-on A* (*inverse* :: *real* $\Rightarrow$ *real*)
proof (*rule convex-on-subset*[*OF - assms*], *intro convex-on-realI*[*of - -* $\lambda x.\ -$*inverse* $(x\hat{}2)$]])
fix *u v* :: *real* **assume** $u \in \{0<..\}$ $v \in \{0<..\}$ $u \leq v$
with *assms* **show** $-$*inverse* $(u\hat{}2) \leq -$*inverse* $(v\hat{}2)$
by (*intro le-imp-neg-le le-imp-inverse-le power-mono*) (*simp-all*)
qed (*insert assms*, *auto intro*!: *derivative-eq-intros simp*: *divide-simps power2-eq-square*)

lemma *convex-onD-Icc'*:
assumes *convex-on* $\{x..y\}$ f $c \in \{x..y\}$
defines $d \equiv y - x$
shows $f c \leq (f y - f x) / d * (c - x) + f x$
proof (*cases y x rule*: *linorder-cases*)
assume *less*: $x < y$
hence *d*: $d > 0$ **by** (*simp add*: *d-def*)
from *assms*(*2*) *less* **have** *A*: $0 \leq (c - x) / d$ $(c - x) / d \leq 1$
by (*simp-all add*: *d-def divide-simps*)
have $f c = f (x + (c - x) * 1)$ **by** *simp*
also from *less* **have** $1 = ((y - x) / d)$ **by** (*simp add*: *d-def*)
also from *d* **have** $x + (c - x) * \ldots = (1 - (c - x) / d) *_R x + ((c - x) / d) *_R y$
by (*simp add*: *field-simps*)
also have $f \ldots \leq (1 - (c - x) / d) * f x + (c - x) / d * f y$ **using** *assms less*
by (*intro convex-onD-Icc*) *simp-all*
also from *d* **have** $\ldots = (f y - f x) / d * (c - x) + f x$ **by** (*simp add*: *field-simps*)
finally show *?thesis* **.**
qed (*insert assms*(*2*), *simp-all*)

lemma *convex-onD-Icc''*:
assumes *convex-on* $\{x..y\}$ f $c \in \{x..y\}$
defines $d \equiv y - x$
shows $f\ c \leq (f\ x - f\ y) \ / \ d * (y - c) + f\ y$
proof (*cases y x rule*: *linorder-cases*)
assume *less*: $x < y$
hence *d*: $d > 0$ **by** (*simp add*: *d-def*)
from *assms*(*2*) *less* **have** *A*: $0 \leq (y - c) \ / \ d \ (y - c) \ / \ d \leq 1$
by (*simp-all add*: *d-def divide-simps*)
have $f\ c = f\ (y - (y - c) * 1)$ **by** *simp*
also from *less* **have** $1 = ((y - x) \ / \ d)$ **by** (*simp add*: *d-def*)
also from *d* **have** $y - (y - c) * \ldots = (1 - (1 - (y - c) \ / \ d)) *_R x + (1 - (y - c) \ / \ d) *_R y$
by (*simp add*: *field-simps*)
also have $f \ \ldots \leq (1 - (1 - (y - c) \ / \ d)) * f\ x + (1 - (y - c) \ / \ d) * f\ y$ **using** *assms less*
by (*intro convex-onD-Icc*) (*simp-all add*: *field-simps*)
also from *d* **have** $\ldots = (f\ x - f\ y) \ / \ d * (y - c) + f\ y$ **by** (*simp add*: *field-simps*)
finally show *?thesis* **.**
qed (*insert assms*(*2*), *simp-all*)

end

21 Pretty syntax for lattice operations

theory *Lattice-Syntax*
imports *Complete-Lattices*
begin

notation
bot (⊥) **and**
top (⊤) **and**
inf (**infixl** ⊓ *70*) **and**
sup (**infixl** ⊔ *65*) **and**
Inf (⨅- [*900*] *900*) **and**
Sup (⨆- [*900*] *900*)

syntax
-INF1 :: *pttrns* ⇒ *'b* ⇒ *'b* ((*3*⨅-./ -) [*0*, *10*] *10*)
-INF :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b* ((*3*⨅-∈-./ -) [*0*, *0*, *10*] *10*)
-SUP1 :: *pttrns* ⇒ *'b* ⇒ *'b* ((*3*⨆-./ -) [*0*, *10*] *10*)
-SUP :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b* ((*3*⨆-∈-./ -) [*0*, *0*, *10*] *10*)

end

22 Formalisation of chain-complete partial orders, continuity and admissibility

theory *Complete-Partial-Order2* **imports**
Main
~~/src/HOL/Library/Lattice-Syntax
begin

context begin interpretation *lifting-syntax* **.**

lemma *chain-transfer* [*transfer-rule*]:
((*A* ===> *A* ===> *op* =) ===> *rel-set A* ===> *op* =) *Complete-Partial-Order.chain*
Complete-Partial-Order.chain
unfolding *chain-def*[*abs-def*] **by** *transfer-prover*

end

lemma *linorder-chain* [*simp*, *intro*!]:
fixes *Y* :: - :: *linorder set*
shows *Complete-Partial-Order.chain op* ≤ *Y*
by(*auto intro*: *chainI*)

lemma *fun-lub-apply*: ⋀*Sup*. *fun-lub Sup Y x* = *Sup* ((λ*f*. *f x*) ' *Y*)
by(*simp add*: *fun-lub-def image-def*)

lemma *fun-lub-empty* [*simp*]: *fun-lub lub* {} = (λ-. *lub* {})
by(*rule ext*)(*simp add*: *fun-lub-apply*)

lemma *chain-fun-ordD*:
assumes *Complete-Partial-Order.chain* (*fun-ord le*) *Y*
shows *Complete-Partial-Order.chain le* ((λ*f*. *f x*) ' *Y*)
by(*rule chainI*)(*auto dest*: *chainD*[*OF assms*] *simp add*: *fun-ord-def*)

lemma *chain-Diff*:
Complete-Partial-Order.chain ord A
⟹ *Complete-Partial-Order.chain ord* (*A* − *B*)
by(*erule chain-subset*) *blast*

lemma *chain-rel-prodD1*:
Complete-Partial-Order.chain (*rel-prod orda ordb*) *Y*
⟹ *Complete-Partial-Order.chain orda* (*fst* ' *Y*)
by(*auto 4 3 simp add*: *chain-def*)

lemma *chain-rel-prodD2*:
Complete-Partial-Order.chain (*rel-prod orda ordb*) *Y*
⟹ *Complete-Partial-Order.chain ordb* (*snd* ' *Y*)
by(*auto 4 3 simp add*: *chain-def*)

context *ccpo* **begin**

lemma *ccpo-fun*: *class.ccpo* (*fun-lub Sup*) (*fun-ord op* $\leq$) (*mk-less* (*fun-ord op* $\leq$))
  by *standard* (*auto 4 3 simp add*: *mk-less-def fun-ord-def fun-lub-apply*
    intro: *order.trans antisym chain-imageI ccpo-Sup-upper ccpo-Sup-least*)

lemma *ccpo-Sup-below-iff*: *Complete-Partial-Order.chain op* $\leq$ $Y \Longrightarrow$ *Sup* $Y \leq x \longleftrightarrow (\forall y{\in}Y.\ y \leq x)$
by(*fast intro*: *order-trans*[*OF ccpo-Sup-upper*] *ccpo-Sup-least*)

lemma *Sup-minus-bot*:
  assumes *chain*: *Complete-Partial-Order.chain op* $\leq$ *A*
  shows $\bigsqcup(A - \{\bigsqcup\{\}\}) = \bigsqcup A$
apply(*rule antisym*)
apply(*blast intro*: *ccpo-Sup-least chain-Diff*[*OF chain*] *ccpo-Sup-upper*[*OF chain*])
apply(*rule ccpo-Sup-least*[*OF chain*])
apply(*case-tac* $x = \bigsqcup\{\}$)
by(*blast intro*: *ccpo-Sup-least chain-empty ccpo-Sup-upper*[*OF chain-Diff*[*OF chain*]])+

lemma *mono-lub*:
  fixes *le-b* (**infix** $\sqsubseteq$ *60*)
  assumes *chain*: *Complete-Partial-Order.chain* (*fun-ord op* $\leq$) *Y*
  and *mono*: $\bigwedge f.\ f \in Y \Longrightarrow$ *monotone le-b op* $\leq f$
  shows *monotone op* $\sqsubseteq$ *op* $\leq$ (*fun-lub Sup Y*)
proof(*rule monotoneI*)
  fix *x y*
  assume $x \sqsubseteq y$

  have *chain''*: $\bigwedge x.$ *Complete-Partial-Order.chain op* $\leq ((\lambda f.\ f\ x)$ ‘ $Y)$
    using *chain* **by**(*rule chain-imageI*)(*simp add*: *fun-ord-def*)
  then show *fun-lub Sup Y x* $\leq$ *fun-lub Sup Y y* **unfolding** *fun-lub-apply*
  proof(*rule ccpo-Sup-least*)
    fix x'
    assume $x' \in (\lambda f.\ f\ x)$ ‘ Y
    then obtain *f* **where** $f \in Y\ x' = f\ x$ **by** *blast*
    note ‹$x' = f\ x$› **also**
    from ‹$f \in Y$› ‹$x \sqsubseteq y$› **have** $f\ x \leq f\ y$ **by**(*blast dest*: *mono monotoneD*)
    also have $\ldots \leq \bigsqcup((\lambda f.\ f\ y)$ ‘ $Y)$ **using** *chain''*
      by(*rule ccpo-Sup-upper*)(*simp add*: ‹$f \in Y$›)
    finally show $x' \leq \bigsqcup((\lambda f.\ f\ y)$ ‘ $Y)$ **.**
  qed
qed

context
  fixes *le-b* (**infix** $\sqsubseteq$ *60*) **and** *Y f*
  assumes *chain*: *Complete-Partial-Order.chain le-b Y*
  and *mono1*: $\bigwedge y.\ y \in Y \Longrightarrow$ *monotone le-b op* $\leq (\lambda x.\ f\ x\ y)$
  and *mono2*: $\bigwedge x\ a\ b.\ [\![\ x \in Y;\ a \sqsubseteq b;\ a \in Y;\ b \in Y\ ]\!] \Longrightarrow f\ x\ a \leq f\ x\ b$
begin

lemma *Sup-mono*:
assumes *le*: $x \sqsubseteq y$ **and** *x*: $x \in Y$ **and** *y*: $y \in Y$
shows $\bigsqcup(f\ x\ `\ Y) \leq \bigsqcup(f\ y\ `\ Y)$ (**is** - $\leq$ *?rhs*)
proof(*rule ccpo-Sup-least*)
from *chain* **show** *chain'*: *Complete-Partial-Order.chain op* $\leq$ $(f\ x\ `\ Y)$ **when** $x \in Y$ **for** x
by(*rule chain-imageI*) (*insert that*, *auto dest*: *mono2*)

fix x'
assume $x' \in f\ x\ `\ Y$
then obtain y' **where** $y' \in Y\ x' = f\ x\ y'$ **by** *blast* **note** *this*(*2*)
also from *mono1*[*OF* ‹$y' \in Y$›] *le* **have** $\ldots \leq f\ y\ y'$ **by**(*rule monotoneD*)
also have $\ldots \leq$ *?rhs* **using** *chain'*[*OF y*]
by (*auto intro*!: *ccpo-Sup-upper simp add*: ‹$y' \in Y$›)
finally show $x' \leq$ *?rhs* **.**
qed(*rule x*)

lemma *diag-Sup*: $\bigsqcup((\lambda x. \bigsqcup(f\ x\ `\ Y))\ `\ Y) = \bigsqcup((\lambda x. f\ x\ x)\ `\ Y)$ (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
have *chain1*: *Complete-Partial-Order.chain op* $\leq$ $((\lambda x. \bigsqcup(f\ x\ `\ Y))\ `\ Y)$
using *chain* **by**(*rule chain-imageI*)(*rule Sup-mono*)
have *chain2*: $\bigwedge y'. y' \in Y \Longrightarrow$ *Complete-Partial-Order.chain op* $\leq$ $(f\ y'\ `\ Y)$ **using** *chain*
by(*rule chain-imageI*)(*auto dest*: *mono2*)
have *chain3*: *Complete-Partial-Order.chain op* $\leq$ $((\lambda x. f\ x\ x)\ `\ Y)$
using *chain* **by**(*rule chain-imageI*)(*auto intro*: *monotoneD*[*OF mono1*] *mono2 order.trans*)

show *?lhs* $\leq$ *?rhs* **using** *chain1*
proof(*rule ccpo-Sup-least*)
fix x'
assume $x' \in (\lambda x. \bigsqcup(f\ x\ `\ Y))\ `\ Y$
then obtain y' **where** $y' \in Y\ x' = \bigsqcup(f\ y'\ `\ Y)$ **by** *blast* **note** *this*(*2*)
also have $\ldots \leq$ *?rhs* **using** *chain2*[*OF* ‹$y' \in Y$›]
proof(*rule ccpo-Sup-least*)
fix x
assume $x \in f\ y'\ `\ Y$
then obtain y **where** $y \in Y$ **and** *x*: $x = f\ y'\ y$ **by** *blast*
def $y'' \equiv$ *if* $y \sqsubseteq y'$ *then* y' *else* y
from *chain* ‹$y \in Y$› ‹$y' \in Y$› **have** $y \sqsubseteq y' \vee y' \sqsubseteq y$ **by**(*rule chainD*)
hence $f\ y'\ y \leq f\ y''\ y''$ **using** ‹$y \in Y$› ‹$y' \in Y$›
by(*auto simp add*: y''*-def intro*: *mono2 monotoneD*[*OF mono1*])
also from ‹$y \in Y$› ‹$y' \in Y$› **have** $y'' \in Y$ **by**(*simp add*: y''*-def*)
from *chain3* **have** $f\ y''\ y'' \leq$ *?rhs* **by**(*rule ccpo-Sup-upper*)(*simp add*: ‹$y'' \in Y$›)
finally show $x \leq$ *?rhs* **by**(*simp add*: *x*)
qed
finally show $x' \leq$ *?rhs* **.**

```
  qed

  show ?rhs ≤ ?lhs using chain3
  proof(rule ccpo-Sup-least)
    fix y
    assume y ∈ (λx. f x x) ‘ Y
    then obtain x where x ∈ Y and y = f x x by blast note this(2)
    also from chain2[OF ‹x ∈ Y›] have ... ≤ ⨆(f x ‘ Y)
      by(rule ccpo-Sup-upper)(simp add: ‹x ∈ Y›)
    also have ... ≤ ?lhs by(rule ccpo-Sup-upper[OF chain1])(simp add: ‹x ∈ Y›)
    finally show y ≤ ?lhs .
  qed
qed

end

lemma Sup-image-mono-le:
  fixes le-b (infix ⊑ 60) and Sup-b (⋁- [900] 900)
  assumes ccpo: class.ccpo Sup-b op ⊑ lt-b
  assumes chain: Complete-Partial-Order.chain op ⊑ Y
  and mono: ⋀x y. ⟦ x ⊑ y; x ∈ Y ⟧ ⟹ f x ≤ f y
  shows Sup (f ‘ Y) ≤ f (⋁Y)
proof(rule ccpo-Sup-least)
  show Complete-Partial-Order.chain op ≤ (f ‘ Y)
    using chain by(rule chain-imageI)(rule mono)

  fix x
  assume x ∈ f ‘ Y
  then obtain y where y ∈ Y and x = f y by blast note this(2)
  also have y ⊑ ⋁Y using ccpo chain ‹y ∈ Y› by(rule ccpo.ccpo-Sup-upper)
  hence f y ≤ f (⋁Y) using ‹y ∈ Y› by(rule mono)
  finally show x ≤ ... .
qed

lemma swap-Sup:
  fixes le-b (infix ⊑ 60)
  assumes Y: Complete-Partial-Order.chain op ⊑ Y
  and Z: Complete-Partial-Order.chain (fun-ord op ≤) Z
  and mono: ⋀f. f ∈ Z ⟹ monotone op ⊑ op ≤ f
  shows ⨆((λx. ⨆(x ‘ Y)) ‘ Z) = ⨆((λx. ⨆((λf. f x) ‘ Z)) ‘ Y)
  (is ?lhs = ?rhs)
proof(cases Y = {})
  case True
  then show ?thesis
    by (simp add: image-constant-conv cong del: strong-SUP-cong)
next
  case False
  have chain1: ⋀f. f ∈ Z ⟹ Complete-Partial-Order.chain op ≤ (f ‘ Y)
    by(rule chain-imageI[OF Y])(rule monotoneD[OF mono])
```

```
have chain2: Complete-Partial-Order.chain op ≤ ((λx. ⨆(x ‘ Y)) ‘ Z) using Z
proof(rule chain-imageI)
  fix f g
  assume f ∈ Z g ∈ Z
    and fun-ord op ≤ f g
  from chain1[OF ‹f ∈ Z›] show ⨆(f ‘ Y) ≤ ⨆(g ‘ Y)
  proof(rule ccpo-Sup-least)
    fix x
    assume x ∈ f ‘ Y
    then obtain y where y ∈ Y x = f y by blast note this(2)
    also have ... ≤ g y using ‹fun-ord op ≤ f g› by(simp add: fun-ord-def)
    also have ... ≤ ⨆(g ‘ Y) using chain1[OF ‹g ∈ Z›]
      by(rule ccpo-Sup-upper)(simp add: ‹y ∈ Y›)
    finally show x ≤ ⨆(g ‘ Y) .
  qed
qed
have chain3: ⋀x. Complete-Partial-Order.chain op ≤ ((λf. f x) ‘ Z)
  using Z by(rule chain-imageI)(simp add: fun-ord-def)
have chain4: Complete-Partial-Order.chain op ≤ ((λx. ⨆((λf. f x) ‘ Z)) ‘ Y)
  using Y
proof(rule chain-imageI)
  fix f x y
  assume x ⊑ y
  show ⨆((λf. f x) ‘ Z) ≤ ⨆((λf. f y) ‘ Z) (is - ≤ ?rhs) using chain3
  proof(rule ccpo-Sup-least)
    fix x'
    assume x' ∈ (λf. f x) ‘ Z
    then obtain f where f ∈ Z x' = f x by blast note this(2)
    also have f x ≤ f y using ‹f ∈ Z› ‹x ⊑ y› by(rule monotoneD[OF mono])
    also have f y ≤ ?rhs using chain3
      by(rule ccpo-Sup-upper)(simp add: ‹f ∈ Z›)
    finally show x' ≤ ?rhs .
  qed
qed

from chain2 have ?lhs ≤ ?rhs
proof(rule ccpo-Sup-least)
  fix x
  assume x ∈ (λx. ⨆(x ‘ Y)) ‘ Z
  then obtain f where f ∈ Z x = ⨆(f ‘ Y) by blast note this(2)
  also have ... ≤ ?rhs using chain1[OF ‹f ∈ Z›]
  proof(rule ccpo-Sup-least)
    fix x'
    assume x' ∈ f ‘ Y
    then obtain y where y ∈ Y x' = f y by blast note this(2)
    also have f y ≤ ⨆((λf. f y) ‘ Z) using chain3
      by(rule ccpo-Sup-upper)(simp add: ‹f ∈ Z›)
    also have ... ≤ ?rhs using chain4 by(rule ccpo-Sup-upper)(simp add: ‹y ∈
Y›)
```

finally show $x' \leq$ *?rhs* **.**
qed
finally show $x \leq$ *?rhs* **.**
qed
moreover
have *?rhs* $\leq$ *?lhs* **using** *chain4*
proof(*rule ccpo-Sup-least*)
fix x
assume $x \in (\lambda x. \bigsqcup((\lambda f. f\ x)\ `\ Z))\ `\ Y$
then obtain y **where** $y \in Y\ x = \bigsqcup((\lambda f. f\ y)\ `\ Z)$ **by** *blast* **note** *this*(*2*)
also have ... $\leq$ *?lhs* **using** *chain3*
proof(*rule ccpo-Sup-least*)
fix x'
assume $x' \in (\lambda f. f\ y)\ `\ Z$
then obtain f **where** $f \in Z\ x' = f\ y$ **by** *blast* **note** *this*(*2*)
also have $f\ y \leq \bigsqcup(f\ `\ Y)$ **using** *chain1*[*OF* ‹$f \in Z$›]
by(*rule ccpo-Sup-upper*)(*simp add*: ‹$y \in Y$›)
also have ... $\leq$ *?lhs* **using** *chain2*
by(*rule ccpo-Sup-upper*)(*simp add*: ‹$f \in Z$›)
finally show $x' \leq$ *?lhs* **.**
qed
finally show $x \leq$ *?lhs* **.**
qed
ultimately show *?lhs* = *?rhs* **by**(*rule antisym*)
qed

lemma *fixp-mono*:
assumes *fg*: *fun-ord op* $\leq$ *f g*
and *f*: *monotone op* $\leq$ *op* $\leq$ *f*
and *g*: *monotone op* $\leq$ *op* $\leq$ *g*
shows *ccpo-class.fixp f* $\leq$ *ccpo-class.fixp g*
unfolding *fixp-def*
proof(*rule ccpo-Sup-least*)
fix x
assume $x \in$ *ccpo-class.iterates f*
thus $x \leq \bigsqcup$ *ccpo-class.iterates g*
proof *induction*
case (*step x*)
from *f step.IH* **have** $f\ x \leq f$ ($\bigsqcup$ *ccpo-class.iterates g*) **by**(*rule monotoneD*)
also have ... $\leq g$ ($\bigsqcup$ *ccpo-class.iterates g*) **using** *fg* **by**(*simp add*: *fun-ord-def*)
also have ... $= \bigsqcup$ *ccpo-class.iterates g* **by**(*fold fixp-def fixp-unfold*[*OF g*]) *simp*
finally show *?case* **.**
qed(*blast intro*: *ccpo-Sup-least*)
qed(*rule chain-iterates*[*OF f*])

context fixes *ordb* :: $'b \Rightarrow 'b \Rightarrow$ *bool* (**infix** $\sqsubseteq$ *60*) **begin**

lemma *iterates-mono*:
assumes *f*: $f \in$ *ccpo.iterates* (*fun-lub Sup*) (*fun-ord op* $\leq$) *F*

and *mono*: $\bigwedge f.$ *monotone op* $\sqsubseteq$ *op* $\leq$ *f* $\Longrightarrow$ *monotone op* $\sqsubseteq$ *op* $\leq$ (*F f*)
shows *monotone op* $\sqsubseteq$ *op* $\leq$ *f*
using *f*
by(*induction rule*: *ccpo.iterates.induct*[*OF ccpo-fun*, *consumes 1*, *case-names step Sup*])(*blast intro*: *mono mono-lub*)+

lemma *fixp-preserves-mono*:
assumes *mono*: $\bigwedge x.$ *monotone* (*fun-ord op* $\leq$) *op* $\leq$ ($\lambda f.$ *F f x*)
and *mono2*: $\bigwedge f.$ *monotone op* $\sqsubseteq$ *op* $\leq$ *f* $\Longrightarrow$ *monotone op* $\sqsubseteq$ *op* $\leq$ (*F f*)
shows *monotone op* $\sqsubseteq$ *op* $\leq$ (*ccpo.fixp* (*fun-lub Sup*) (*fun-ord op* $\leq$) *F*)
(**is** *monotone - - ?fixp*)
proof(*rule monotoneI*)
have *mono*: *monotone* (*fun-ord op* $\leq$) (*fun-ord op* $\leq$) *F*
by(*rule monotoneI*)(*auto simp add*: *fun-ord-def intro*: *monotoneD*[*OF mono*])
let *?iter* = *ccpo.iterates* (*fun-lub Sup*) (*fun-ord op* $\leq$) *F*
have *chain*: $\bigwedge x.$ *Complete-Partial-Order.chain op* $\leq$ (($\lambda f.$ *f x*) ' *?iter*)
by(*rule chain-imageI*[*OF ccpo.chain-iterates*[*OF ccpo-fun mono*]])(*simp add*: *fun-ord-def*)

fix *x y*
assume *x* $\sqsubseteq$ *y*
show *?fixp x* $\leq$ *?fixp y*
unfolding *ccpo.fixp-def*[*OF ccpo-fun*] *fun-lub-apply* **using** *chain*
proof(*rule ccpo-Sup-least*)
fix x'
assume $x' \in$ ($\lambda f.$ *f x*) ' *?iter*
then obtain *f* **where** *f* $\in$ *?iter* $x' = f\ x$ **by** *blast* **note** *this*(*2*)
also have *f x* $\leq$ *f y*
by(*rule monotoneD*[*OF iterates-mono*[*OF* ‹*f* $\in$ *?iter*› *mono2*]])(*blast intro*: ‹*x* $\sqsubseteq$ *y*›)+
also have *f y* $\leq$ $\bigsqcup$(($\lambda f.$ *f y*) ' *?iter*) **using** *chain*
by(*rule ccpo-Sup-upper*)(*simp add*: ‹*f* $\in$ *?iter*›)
finally show $x' \leq \ldots$ **.**
qed
qed

end

end

lemma *monotone2monotone*:
assumes *2*: $\bigwedge x.$ *monotone ordb ordc* ($\lambda y.$ *f x y*)
and *t*: *monotone orda ordb* ($\lambda x.$ *t x*)
and *1*: $\bigwedge y.$ *monotone orda ordc* ($\lambda x.$ *f x y*)
and *trans*: *transp ordc*
shows *monotone orda ordc* ($\lambda x.$ *f x* (*t x*))
by(*blast intro*: *monotoneI transpD*[*OF trans*] *monotoneD*[*OF t*] *monotoneD*[*OF 2*] *monotoneD*[*OF 1*])

22.1 Continuity

definition *cont* :: ('a set ⇒ 'a) ⇒ ('a ⇒ 'a ⇒ bool) ⇒ ('b set ⇒ 'b) ⇒ ('b ⇒ 'b ⇒ bool) ⇒ ('a ⇒ 'b) ⇒ bool
where
cont luba orda lubb ordb f ⟷
(∀ *Y*. *Complete-Partial-Order.chain orda Y* ⟶ *Y* ≠ {} ⟶ *f* (*luba Y*) = *lubb* (*f* ' *Y*))

definition *mcont* :: ('a set ⇒ 'a) ⇒ ('a ⇒ 'a ⇒ bool) ⇒ ('b set ⇒ 'b) ⇒ ('b ⇒ 'b ⇒ bool) ⇒ ('a ⇒ 'b) ⇒ bool
where
mcont luba orda lubb ordb f ⟷
monotone orda ordb f ∧ *cont luba orda lubb ordb f*

22.1.1 Theorem collection *cont-intro*

named-theorems *cont-intro continuity and admissibility intro rules*
ML ‹

```
(* apply cont-intro rules as intro and try to solve
   the remaining of the emerging subgoals with simp *)
fun cont-intro-tac ctxt =
 REPEAT-ALL-NEW (resolve-tac ctxt (rev (Named-Theorems.get ctxt @{named-theorems
cont-intro})))
  THEN-ALL-NEW (SOLVED' (simp-tac ctxt))

fun cont-intro-simproc ctxt ct =
  let
    fun mk-stmt t = t
      |> HOLogic.mk-Trueprop
      |> Thm.cterm-of ctxt
      |> Goal.init
    fun mk-thm t =
      case SINGLE (cont-intro-tac ctxt 1) (mk-stmt t) of
        SOME thm => SOME (Goal.finish ctxt thm RS @{thm Eq-TrueI})
      | NONE => NONE
  in
    case Thm.term-of ct of
      t as Const (@{const-name ccpo.admissible}, -) $ - $ - $ - => mk-thm t
    | t as Const (@{const-name mcont}, -) $ - $ - $ - $ - $ - => mk-thm t
    | t as Const (@{const-name monotone}, -) $ - $ - $ - => mk-thm t
    | - => NONE
  end
  handle THM - => NONE
  | TYPE - => NONE
```
›

simproc-setup *cont-intro*
(*ccpo.admissible lub ord P*
| *mcont lub ord lub' ord' f*

| *monotone ord ord' f*
) = ‹*K cont-intro-simproc*›

lemmas [*cont-intro*] =
call-mono
let-mono
if-mono
option.const-mono
tailrec.const-mono
bind-mono

declare *if-mono*[*simp*]

lemma *monotone-id'* [*cont-intro*]: *monotone ord ord* ($\lambda x.\ x$)
by(*simp add*: *monotone-def*)

lemma *monotone-applyI*:
monotone orda ordb F $\Longrightarrow$ *monotone* (*fun-ord orda*) *ordb* ($\lambda f.\ F\ (f\ x)$)
by(*rule monotoneI*)(*auto simp add*: *fun-ord-def dest*: *monotoneD*)

lemma *monotone-if-fun* [*partial-function-mono*]:
⟦ *monotone* (*fun-ord orda*) (*fun-ord ordb*) *F*; *monotone* (*fun-ord orda*) (*fun-ord ordb*) *G* ⟧
$\Longrightarrow$ *monotone* (*fun-ord orda*) (*fun-ord ordb*) ($\lambda f\ n.$ *if c n then F f n else G f n*)
by(*simp add*: *monotone-def fun-ord-def*)

lemma *monotone-fun-apply-fun* [*partial-function-mono*]:
monotone (*fun-ord* (*fun-ord ord*)) (*fun-ord ord*) ($\lambda f\ n.\ f\ t\ (g\ n)$)
by(*rule monotoneI*)(*simp add*: *fun-ord-def*)

lemma *monotone-fun-ord-apply*:
monotone orda (*fun-ord ordb*) *f* $\longleftrightarrow$ ($\forall x.$ *monotone orda ordb* ($\lambda y.\ f\ y\ x$))
by(*auto simp add*: *monotone-def fun-ord-def*)

context *preorder* **begin**

lemma *transp-le* [*simp*, *cont-intro*]: *transp op* $\leq$
by(*rule transpI*)(*rule order-trans*)

lemma *monotone-const* [*simp*, *cont-intro*]: *monotone ord op* $\leq$ (λ-. *c*)
by(*rule monotoneI*) *simp*

end

lemma *transp-le* [*cont-intro*, *simp*]:
class.preorder ord (*mk-less ord*) $\Longrightarrow$ *transp ord*
by(*rule preorder.transp-le*)

context *partial-function-definitions* **begin**

declare *const-mono* [*cont-intro*, *simp*]

lemma *transp-le* [*cont-intro*, *simp*]: *transp leq*
by(*rule transpI*)(*rule leq-trans*)

lemma *preorder* [*cont-intro*, *simp*]: *class.preorder leq* (*mk-less leq*)
by(*unfold-locales*)(*auto simp add*: *mk-less-def intro*: *leq-refl leq-trans*)

declare *ccpo*[*cont-intro*, *simp*]

end

lemma *contI* [*intro?*]:
$(\bigwedge Y.\ \llbracket$ *Complete-Partial-Order.chain orda* $Y;\ Y \neq \{\}\ \rrbracket \Longrightarrow f\ (luba\ Y) = lubb\ (f\ `\ Y))$
$\Longrightarrow$ *cont luba orda lubb ordb f*
unfolding *cont-def* **by** *blast*

lemma *contD*:
$\llbracket$ *cont luba orda lubb ordb f*; *Complete-Partial-Order.chain orda* $Y;\ Y \neq \{\}\ \rrbracket$
$\Longrightarrow f\ (luba\ Y) = lubb\ (f\ `\ Y)$
unfolding *cont-def* **by** *blast*

lemma *cont-id* [*simp*, *cont-intro*]: $\bigwedge$*Sup*. *cont Sup ord Sup ord id*
by(*rule contI*) *simp*

lemma *cont-id′* [*simp*, *cont-intro*]: $\bigwedge$*Sup*. *cont Sup ord Sup ord* $(\lambda x.\ x)$
using *cont-id*[*unfolded id-def*] **.**

lemma *cont-applyI* [*cont-intro*]:
assumes *cont*: *cont luba orda lubb ordb g*
shows *cont* (*fun-lub luba*) (*fun-ord orda*) *lubb ordb* $(\lambda f.\ g\ (f\ x))$
by(*rule contI*)(*drule chain-fun-ordD*[**where** *x*=*x*], *simp add*: *fun-lub-apply image-image contD*[*OF cont*])

lemma *call-cont*: *cont* (*fun-lub lub*) (*fun-ord ord*) *lub ord* $(\lambda f.\ f\ t)$
by(*simp add*: *cont-def fun-lub-apply*)

lemma *cont-if* [*cont-intro*]:
$\llbracket$ *cont luba orda lubb ordb f*; *cont luba orda lubb ordb g* $\rrbracket$
$\Longrightarrow$ *cont luba orda lubb ordb* $(\lambda x.$ *if c then f x else g x*$)$
by(*cases c*) *simp-all*

lemma *mcontI* [*intro?*]:
$\llbracket$ *monotone orda ordb f*; *cont luba orda lubb ordb f* $\rrbracket \Longrightarrow$ *mcont luba orda lubb ordb f*
by(*simp add*: *mcont-def*)

lemma *mcont-mono*: *mcont luba orda lubb ordb f* $\Longrightarrow$ *monotone orda ordb f*
by(*simp add*: *mcont-def*)

lemma *mcont-cont* [*simp*]: *mcont luba orda lubb ordb f* $\Longrightarrow$ *cont luba orda lubb ordb f*
by(*simp add*: *mcont-def*)

lemma *mcont-monoD*:
 $[\![$ *mcont luba orda lubb ordb f*; *orda x y* $]\!]$ $\Longrightarrow$ *ordb* (*f x*) (*f y*)
by(*auto simp add*: *mcont-def dest*: *monotoneD*)

lemma *mcont-contD*:
 $[\![$ *mcont luba orda lubb ordb f*; *Complete-Partial-Order.chain orda Y*; $Y \neq \{\}$ $]\!]$
 $\Longrightarrow$ *f* (*luba Y*) = *lubb* (*f ‘ Y*)
by(*auto simp add*: *mcont-def dest*: *contD*)

lemma *mcont-call* [*cont-intro*, *simp*]:
 mcont (*fun-lub lub*) (*fun-ord ord*) *lub ord* (λf. *f t*)
by(*simp add*: *mcont-def call-mono call-cont*)

lemma *mcont-id′* [*cont-intro*, *simp*]: *mcont lub ord lub ord* (λx. *x*)
by(*simp add*: *mcont-def monotone-id′*)

lemma *mcont-applyI*:
 mcont luba orda lubb ordb (λx. *F x*) $\Longrightarrow$ *mcont* (*fun-lub luba*) (*fun-ord orda*) *lubb ordb* (λf. *F* (*f x*))
by(*simp add*: *mcont-def monotone-applyI cont-applyI*)

lemma *mcont-if* [*cont-intro*, *simp*]:
 $[\![$ *mcont luba orda lubb ordb* (λx. *f x*); *mcont luba orda lubb ordb* (λx. *g x*) $]\!]$
 $\Longrightarrow$ *mcont luba orda lubb ordb* (λx. *if c then f x else g x*)
by(*simp add*: *mcont-def cont-if*)

lemma *cont-fun-lub-apply*:
 cont luba orda (*fun-lub lubb*) (*fun-ord ordb*) *f* $\longleftrightarrow$ ($\forall$ *x*. *cont luba orda lubb ordb* (λy. *f y x*))
by(*simp add*: *cont-def fun-lub-def fun-eq-iff*)(*auto simp add*: *image-def*)

lemma *mcont-fun-lub-apply*:
 mcont luba orda (*fun-lub lubb*) (*fun-ord ordb*) *f* $\longleftrightarrow$ ($\forall$ *x*. *mcont luba orda lubb ordb* (λy. *f y x*))
by(*auto simp add*: *monotone-fun-ord-apply cont-fun-lub-apply mcont-def*)

context *ccpo* **begin**

lemma *cont-const* [*simp*, *cont-intro*]: *cont luba orda Sup op* $\leq$ (λx. *c*)
by (*rule contI*) (*simp add*: *image-constant-conv cong del*: *strong-SUP-cong*)

lemma *mcont-const* [*cont-intro*, *simp*]:

mcont luba orda Sup op $\leq$ ($\lambda x.$ *c*)
by(*simp add*: *mcont-def*)

lemma *cont-apply*:
assumes *2*: $\bigwedge x.$ *cont lubb ordb Sup op* $\leq$ ($\lambda y.$ *f x y*)
and *t*: *cont luba orda lubb ordb* ($\lambda x.$ *t x*)
and *1*: $\bigwedge y.$ *cont luba orda Sup op* $\leq$ ($\lambda x.$ *f x y*)
and *mono*: *monotone orda ordb* ($\lambda x.$ *t x*)
and *mono2*: $\bigwedge x.$ *monotone ordb op* $\leq$ ($\lambda y.$ *f x y*)
and *mono1*: $\bigwedge y.$ *monotone orda op* $\leq$ ($\lambda x.$ *f x y*)
shows *cont luba orda Sup op* $\leq$ ($\lambda x.$ *f x* (*t x*))
proof
fix *Y*
assume *chain*: *Complete-Partial-Order.chain orda Y* **and** *Y* $\neq$ {}
moreover from *chain* **have** *chain'*: *Complete-Partial-Order.chain ordb* (*t* ‘ *Y*)
by(*rule chain-imageI*)(*rule monotoneD*[*OF mono*])
ultimately show *f* (*luba Y*) (*t* (*luba Y*)) = $\bigsqcup$(($\lambda x.$ *f x* (*t x*)) ‘ *Y*)
by(*simp add*: *contD*[*OF 1*] *contD*[*OF t*] *contD*[*OF 2*] *image-image*)
(*rule diag-Sup*[*OF chain*], *auto intro*: *monotone2monotone*[*OF mono2 mono monotone-const transpI*] *monotoneD*[*OF mono1*])
qed

lemma *mcont2mcont'*:
⟦ $\bigwedge x.$ *mcont lub' ord' Sup op* $\leq$ ($\lambda y.$ *f x y*);
$\bigwedge y.$ *mcont lub ord Sup op* $\leq$ ($\lambda x.$ *f x y*);
mcont lub ord lub' ord' ($\lambda y.$ *t y*) ⟧
$\Longrightarrow$ *mcont lub ord Sup op* $\leq$ ($\lambda x.$ *f x* (*t x*))
unfolding *mcont-def* **by**(*blast intro*: *transp-le monotone2monotone cont-apply*)

lemma *mcont2mcont*:
⟦*mcont lub' ord' Sup op* $\leq$ ($\lambda x.$ *f x*); *mcont lub ord lub' ord'* ($\lambda x.$ *t x*)⟧
$\Longrightarrow$ *mcont lub ord Sup op* $\leq$ ($\lambda x.$ *f* (*t x*))
by(*rule mcont2mcont'*[*OF - mcont-const*])

context
fixes *ord* :: *'b* $\Rightarrow$ *'b* $\Rightarrow$ *bool* (**infix** $\sqsubseteq$ *60*)
and *lub* :: *'b set* $\Rightarrow$ *'b* ($\bigvee$- [*900*] *900*)
begin

lemma *cont-fun-lub-Sup*:
assumes *chainM*: *Complete-Partial-Order.chain* (*fun-ord op* $\leq$) *M*
and *mcont* [*rule-format*]: $\forall f \in M.$ *mcont lub op* $\sqsubseteq$ *Sup op* $\leq$ *f*
shows *cont lub op* $\sqsubseteq$ *Sup op* $\leq$ (*fun-lub Sup M*)
proof(*rule contI*)
fix *Y*
assume *chain*: *Complete-Partial-Order.chain op* $\sqsubseteq$ *Y*
and *Y*: *Y* $\neq$ {}
from *swap-Sup*[*OF chain chainM mcont*[*THEN mcont-mono*]]
show *fun-lub Sup M* ($\bigvee$*Y*) = $\bigsqcup$(*fun-lub Sup M* ‘ *Y*)

by(*simp add*: *mcont-contD*[*OF mcont chain Y*] *fun-lub-apply cong*: *image-cong*)
qed

lemma *mcont-fun-lub-Sup*:
⟦ *Complete-Partial-Order.chain* (*fun-ord op* ≤) *M*;
∀ *f*∈*M*. *mcont lub ord Sup op* ≤ *f* ⟧
⟹ *mcont lub op* ⊑ *Sup op* ≤ (*fun-lub Sup M*)
by(*simp add*: *mcont-def cont-fun-lub-Sup mono-lub*)

lemma *iterates-mcont*:
assumes *f*: *f* ∈ *ccpo.iterates* (*fun-lub Sup*) (*fun-ord op* ≤) *F*
and *mono*: ⋀*f*. *mcont lub op* ⊑ *Sup op* ≤ *f* ⟹ *mcont lub op* ⊑ *Sup op* ≤ (*F f*)
shows *mcont lub op* ⊑ *Sup op* ≤ *f*
using *f*
by(*induction rule*: *ccpo.iterates.induct*[*OF ccpo-fun*, *consumes 1*, *case-names step Sup*])(*blast intro*: *mono mcont-fun-lub-Sup*)+

lemma *fixp-preserves-mcont*:
assumes *mono*: ⋀*x*. *monotone* (*fun-ord op* ≤) *op* ≤ (λ*f*. *F f x*)
and *mcont*: ⋀*f*. *mcont lub op* ⊑ *Sup op* ≤ *f* ⟹ *mcont lub op* ⊑ *Sup op* ≤ (*F f*)
shows *mcont lub op* ⊑ *Sup op* ≤ (*ccpo.fixp* (*fun-lub Sup*) (*fun-ord op* ≤) *F*)
(**is** *mcont* - - - - *?fixp*)
unfolding *mcont-def*
proof(*intro conjI monotoneI contI*)
have *mono*: *monotone* (*fun-ord op* ≤) (*fun-ord op* ≤) *F*
by(*rule monotoneI*)(*auto simp add*: *fun-ord-def intro*: *monotoneD*[*OF mono*])
let *?iter* = *ccpo.iterates* (*fun-lub Sup*) (*fun-ord op* ≤) *F*
have *chain*: ⋀*x*. *Complete-Partial-Order.chain op* ≤ ((λ*f*. *f x*) ‘ *?iter*)
by(*rule chain-imageI*[*OF ccpo.chain-iterates*[*OF ccpo-fun mono*]])(*simp add*: *fun-ord-def*)

{
fix *x y*
assume *x* ⊑ *y*
show *?fixp x* ≤ *?fixp y*
unfolding *ccpo.fixp-def*[*OF ccpo-fun*] *fun-lub-apply* **using** *chain*
proof(*rule ccpo-Sup-least*)
fix *x′*
assume *x′* ∈ (λ*f*. *f x*) ‘ *?iter*
then obtain *f* **where** *f* ∈ *?iter x′* = *f x* **by** *blast* **note** *this*(*2*)
also from - ‹*x* ⊑ *y*› **have** *f x* ≤ *f y*
by(*rule mcont-monoD*[*OF iterates-mcont*[*OF* ‹*f* ∈ *?iter*› *mcont*]])
also have *f y* ≤ ⨆((λ*f*. *f y*) ‘ *?iter*) **using** *chain*
by(*rule ccpo-Sup-upper*)(*simp add*: ‹*f* ∈ *?iter*›)
finally show *x′* ≤ . . . **.**
qed
next
fix *Y*

assume *chain*: *Complete-Partial-Order.chain op* $\sqsubseteq$ *Y*
and *Y*: $Y \neq \{\}$
{ fix *f*
assume $f \in$ *?iter*
hence f ($\bigvee Y$) = $\bigsqcup$(f ' Y)
using *mcont chain Y* **by**(*rule mcont-contD*[*OF iterates-mcont*]) **}**
moreover have $\bigsqcup$(($\lambda f.$ $\bigsqcup$(f ' Y)) ' *?iter*) = $\bigsqcup$(($\lambda x.$ $\bigsqcup$(($\lambda f.$ f x) ' *?iter*)) ' Y)
using *chain ccpo.chain-iterates*[*OF ccpo-fun mono*]
by(*rule swap-Sup*)(*rule mcont-mono*[*OF iterates-mcont*[*OF - mcont*]])
ultimately show *?fixp* ($\bigvee Y$) = $\bigsqcup$(*?fixp* ' Y) **unfolding** *ccpo.fixp-def*[*OF ccpo-fun*]
by(*simp add*: *fun-lub-apply cong*: *image-cong*)
}
qed

end

context
fixes F :: $'c \Rightarrow 'c$ **and** U :: $'c \Rightarrow 'b \Rightarrow 'a$ **and** C :: $('b \Rightarrow 'a) \Rightarrow 'c$ **and** f
assumes *mono*: $\bigwedge x.$ *monotone* (*fun-ord op* $\leq$) *op* $\leq$ ($\lambda f.$ U (F (C f)) x)
and *eq*: $f \equiv C$ (*ccpo.fixp* (*fun-lub Sup*) (*fun-ord op* $\leq$) ($\lambda f.$ U (F (C f))))
and *inverse*: $\bigwedge f.$ U (C f) = f
begin

lemma *fixp-preserves-mono-uc*:
assumes *mono2*: $\bigwedge f.$ *monotone ord op* $\leq$ (U f) $\Longrightarrow$ *monotone ord op* $\leq$ (U (F f))
shows *monotone ord op* $\leq$ (U f)
using *fixp-preserves-mono*[*OF mono mono2*] **by**(*subst eq*)(*simp add*: *inverse*)

lemma *fixp-preserves-mcont-uc*:
assumes *mcont*: $\bigwedge f.$ *mcont lubb ordb Sup op* $\leq$ (U f) $\Longrightarrow$ *mcont lubb ordb Sup op* $\leq$ (U (F f))
shows *mcont lubb ordb Sup op* $\leq$ (U f)
using *fixp-preserves-mcont*[*OF mono mcont*] **by**(*subst eq*)(*simp add*: *inverse*)

end

lemmas *fixp-preserves-mono1* = *fixp-preserves-mono-uc*[*of* $\lambda x.$ *x* - $\lambda x.$ *x*, *OF - - refl*]
lemmas *fixp-preserves-mono2* =
fixp-preserves-mono-uc[*of case-prod - curry*, *unfolded case-prod-curry curry-case-prod*, *OF - - refl*]
lemmas *fixp-preserves-mono3* =
fixp-preserves-mono-uc[*of* $\lambda f.$ *case-prod* (*case-prod f*) - $\lambda f.$ *curry* (*curry f*), *unfolded case-prod-curry curry-case-prod*, *OF - - refl*]
lemmas *fixp-preserves-mono4* =
fixp-preserves-mono-uc[*of* $\lambda f.$ *case-prod* (*case-prod* (*case-prod f*)) - $\lambda f.$ *curry*

(*curry* (*curry f*)), *unfolded case-prod-curry curry-case-prod*, *OF - - refl*]

lemmas *fixp-preserves-mcont1* = *fixp-preserves-mcont-uc*[*of* λ*x*. *x* - λ*x*. *x*, *OF - - refl*]
lemmas *fixp-preserves-mcont2* =
fixp-preserves-mcont-uc[*of case-prod - curry*, *unfolded case-prod-curry curry-case-prod*, *OF - - refl*]
lemmas *fixp-preserves-mcont3* =
fixp-preserves-mcont-uc[*of* λ*f*. *case-prod* (*case-prod f*) - λ*f*. *curry* (*curry f*), *unfolded case-prod-curry curry-case-prod*, *OF - - refl*]
lemmas *fixp-preserves-mcont4* =
fixp-preserves-mcont-uc[*of* λ*f*. *case-prod* (*case-prod* (*case-prod f*)) - λ*f*. *curry* (*curry* (*curry f*)), *unfolded case-prod-curry curry-case-prod*, *OF - - refl*]

end

lemma (**in** *preorder*) *monotone-if-bot*:
  fixes *bot*
  assumes *mono*: ⋀*x y*. ⟦ *x* ≤ *y*; ¬ (*x* ≤ *bound*) ⟧ ⟹ *ord* (*f x*) (*f y*)
  and *bot*: ⋀*x*. ¬ *x* ≤ *bound* ⟹ *ord bot* (*f x*) *ord bot bot*
  shows *monotone op* ≤ *ord* (λ*x*. *if x* ≤ *bound then bot else f x*)
by(*rule monotoneI*)(*auto intro*: *bot intro*: *mono order-trans*)

lemma (**in** *ccpo*) *mcont-if-bot*:
  fixes *bot* **and** *lub* (⋁- [*900*] *900*) **and** *ord* (**infix** ⊑ *60*)
  assumes *ccpo*: *class.ccpo lub op* ⊑ *lt*
  and *mono*: ⋀*x y*. ⟦ *x* ≤ *y*; ¬ *x* ≤ *bound* ⟧ ⟹ *f x* ⊑ *f y*
  and *cont*: ⋀*Y*. ⟦ *Complete-Partial-Order.chain op* ≤ *Y*; *Y* ≠ {}; ⋀*x*. *x* ∈ *Y* ⟹ ¬ *x* ≤ *bound* ⟧ ⟹ *f* (⨆ *Y*) = ⋁(*f* ‘ *Y*)
  and *bot*: ⋀*x*. ¬ *x* ≤ *bound* ⟹ *bot* ⊑ *f x*
  shows *mcont Sup op* ≤ *lub op* ⊑ (λ*x*. *if x* ≤ *bound then bot else f x*) (**is** *mcont - - - - ?g*)
proof(*intro mcontI contI*)
  interpret *c*: *ccpo lub op* ⊑ *lt* **by**(*fact ccpo*)
  show *monotone op* ≤ *op* ⊑ *?g* **by**(*rule monotone-if-bot*)(*simp-all add*: *mono bot*)

  fix *Y*
  assume *chain*: *Complete-Partial-Order.chain op* ≤ *Y* **and** *Y*: *Y* ≠ {}
  show *?g* (⨆ *Y*) = ⋁(*?g* ‘ *Y*)
  proof(*cases Y* ⊆ {*x*. *x* ≤ *bound*})
    case *True*
    hence ⨆ *Y* ≤ *bound* **using** *chain* **by**(*auto intro*: *ccpo-Sup-least*)
    moreover have *Y* ∩ {*x*. ¬ *x* ≤ *bound*} = {} **using** *True* **by** *auto*
    ultimately show *?thesis* **using** *True Y*
      by (*auto simp add*: *image-constant-conv cong del*: *c.strong-SUP-cong*)
  next
    case *False*
    let *?Y* = *Y* ∩ {*x*. ¬ *x* ≤ *bound*}
    have *chain'*: *Complete-Partial-Order.chain op* ≤ *?Y*

using *chain* **by**(*rule chain-subset*) *simp*

from *False* **obtain** *y* **where** *ybound*: ¬ $y \leq$ *bound* **and** *y*: $y \in Y$ **by** *blast*
hence ¬ $\bigsqcup Y \leq$ *bound* **by** (*metis ccpo-Sup-upper chain order.trans*)
hence *?g* ($\bigsqcup Y$) = *f* ($\bigsqcup Y$) **by** *simp*
also have $\bigsqcup Y \leq \bigsqcup$ *?Y* **using** *chain*
proof(*rule ccpo-Sup-least*)
fix *x*
assume *x*: $x \in Y$
show $x \leq \bigsqcup$ *?Y*
proof(*cases* $x \leq$ *bound*)
case *True*
with *chainD*[*OF chain x y*] **have** $x \leq y$ **using** *ybound* **by**(*auto intro*: *order-trans*)
thus *?thesis* **by**(*rule order-trans*)(*auto intro*: *ccpo-Sup-upper*[*OF chain'*] *simp add*: *y ybound*)
qed(*auto intro*: *ccpo-Sup-upper*[*OF chain'*] *simp add*: *x*)
qed
hence $\bigsqcup Y = \bigsqcup$ *?Y* **by**(*rule antisym*)(*blast intro*: *ccpo-Sup-least*[*OF chain'*] *ccpo-Sup-upper*[*OF chain*])
hence *f* ($\bigsqcup Y$) = *f* ($\bigsqcup$ *?Y*) **by** *simp*
also have *f* ($\bigsqcup$ *?Y*) = $\bigvee$(*f* ‘ *?Y*) **using** *chain'* **by**(*rule cont*)(*insert y ybound*, *auto*)
also have $\bigvee$(*f* ‘ *?Y*) = $\bigvee$(*?g* ‘ *Y*)
proof(*cases* $Y \cap \{x.\ x \leq bound\} = \{\}$)
case *True*
hence *f* ‘ *?Y* = *?g* ‘ *Y* **by** *auto*
thus *?thesis* **by**(*rule arg-cong*)
next
case *False*
have *chain''*: *Complete-Partial-Order.chain op* ⊑ (*insert bot* (*f* ‘ *?Y*))
using *chain* **by**(*auto intro*!: *chainI bot dest*: *chainD intro*: *mono*)
hence *chain'''*: *Complete-Partial-Order.chain op* ⊑ (*f* ‘ *?Y*) **by**(*rule chain-subset*) *blast*
have *bot* ⊑ $\bigvee$(*f* ‘ *?Y*) **using** *y ybound* **by**(*blast intro*: *c.order-trans*[*OF bot*] *c.ccpo-Sup-upper*[*OF chain'''*])
hence $\bigvee$(*insert bot* (*f* ‘ *?Y*)) ⊑ $\bigvee$(*f* ‘ *?Y*) **using** *chain''*
by(*auto intro*: *c.ccpo-Sup-least c.ccpo-Sup-upper*[*OF chain''*])
with - **have** ... = $\bigvee$(*insert bot* (*f* ‘ *?Y*))
by(*rule c.antisym*)(*blast intro*: *c.ccpo-Sup-least*[*OF chain'''*] *c.ccpo-Sup-upper*[*OF chain''*])
also have *insert bot* (*f* ‘ *?Y*) = *?g* ‘ *Y* **using** *False* **by** *auto*
finally show *?thesis* **.**
qed
finally show *?thesis* **.**
qed
qed

context *partial-function-definitions* **begin**

lemma *mcont-const* [*cont-intro*, *simp*]:
  mcont luba orda lub leq (λ*x*. *c*)
by(*rule ccpo.mcont-const*)(*rule Partial-Function.ccpo*[*OF partial-function-definitions-axioms*])

lemmas [*cont-intro*, *simp*] =
  ccpo.cont-const[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]

lemma *mono2mono*:
  assumes *monotone ordb leq* (λ*y*. *f y*) *monotone orda ordb* (λ*x*. *t x*)
  shows *monotone orda leq* (λ*x*. *f* (*t x*))
using *assms* **by**(*rule monotone2monotone*) *simp-all*

lemmas *mcont2mcont'* = *ccpo.mcont2mcont'*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*
lemmas *mcont2mcont* = *ccpo.mcont2mcont*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]

lemmas *fixp-preserves-mono1* = *ccpo.fixp-preserves-mono1*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]
lemmas *fixp-preserves-mono2* = *ccpo.fixp-preserves-mono2*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]
lemmas *fixp-preserves-mono3* = *ccpo.fixp-preserves-mono3*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]
lemmas *fixp-preserves-mono4* = *ccpo.fixp-preserves-mono4*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]
lemmas *fixp-preserves-mcont1* = *ccpo.fixp-preserves-mcont1*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]
lemmas *fixp-preserves-mcont2* = *ccpo.fixp-preserves-mcont2*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]
lemmas *fixp-preserves-mcont3* = *ccpo.fixp-preserves-mcont3*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]
lemmas *fixp-preserves-mcont4* = *ccpo.fixp-preserves-mcont4*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]

lemma *monotone-if-bot*:
  fixes *bot*
  assumes *g*: ⋀*x*. *g x* = (*if leq x bound then bot else f x*)
  and *mono*: ⋀*x y*. ⟦ *leq x y*; ¬ *leq x bound* ⟧ ⟹ *ord* (*f x*) (*f y*)
  and *bot*: ⋀*x*. ¬ *leq x bound* ⟹ *ord bot* (*f x*) *ord bot bot*
  shows *monotone leq ord g*
unfolding *g*[*abs-def*] **using** *preorder mono bot* **by**(*rule preorder.monotone-if-bot*)

lemma *mcont-if-bot*:
  fixes *bot*
  assumes *ccpo*: *class.ccpo lub' ord* (*mk-less ord*)
  and *bot*: ⋀*x*. ¬ *leq x bound* ⟹ *ord bot* (*f x*)
  and *g*: ⋀*x*. *g x* = (*if leq x bound then bot else f x*)
  and *mono*: ⋀*x y*. ⟦ *leq x y*; ¬ *leq x bound* ⟧ ⟹ *ord* (*f x*) (*f y*)
  and *cont*: ⋀*Y*. ⟦ *Complete-Partial-Order.chain leq Y*; *Y* ≠ {}; ⋀*x*. *x* ∈ *Y* ⟹ ¬ *leq x bound* ⟧ ⟹ *f* (*lub Y*) = *lub'* (*f* ' *Y*)

shows *mcont lub leq lub' ord g*
unfolding *g*[*abs-def*] **using** *ccpo mono cont bot* **by**(*rule ccpo.mcont-if-bot*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]])

end

22.2 Admissibility

lemma *admissible-subst*:
assumes *adm*: *ccpo.admissible luba orda* ($\lambda x.\ P\ x$)
and *mcont*: *mcont lubb ordb luba orda f*
shows *ccpo.admissible lubb ordb* ($\lambda x.\ P\ (f\ x)$)
apply(*rule ccpo.admissibleI*)
apply(*frule* (*1*) *mcont-contD*[*OF mcont*])
apply(*auto intro*: *ccpo.admissibleD*[*OF adm*] *chain-imageI dest*: *mcont-monoD*[*OF mcont*])
done

lemmas [*simp*, *cont-intro*] =
admissible-all
admissible-ball
admissible-const
admissible-conj

lemma *admissible-disj'* [*simp*, *cont-intro*]:
⟦ *class.ccpo lub ord* (*mk-less ord*); *ccpo.admissible lub ord P*; *ccpo.admissible lub ord Q* ⟧
$\Longrightarrow$ *ccpo.admissible lub ord* ($\lambda x.\ P\ x \vee Q\ x$)
by(*rule ccpo.admissible-disj*)

lemma *admissible-imp'* [*cont-intro*]:
⟦ *class.ccpo lub ord* (*mk-less ord*);
ccpo.admissible lub ord ($\lambda x.\ \neg\ P\ x$);
ccpo.admissible lub ord ($\lambda x.\ Q\ x$) ⟧
$\Longrightarrow$ *ccpo.admissible lub ord* ($\lambda x.\ P\ x \longrightarrow Q\ x$)
unfolding *imp-conv-disj* **by**(*rule ccpo.admissible-disj*)

lemma *admissible-imp* [*cont-intro*]:
($Q \Longrightarrow$ *ccpo.admissible lub ord* ($\lambda x.\ P\ x$))
$\Longrightarrow$ *ccpo.admissible lub ord* ($\lambda x.\ Q \longrightarrow P\ x$)
by(*rule ccpo.admissibleI*)(*auto dest*: *ccpo.admissibleD*)

lemma *admissible-not-mem'* [*THEN admissible-subst*, *cont-intro*, *simp*]:
shows *admissible-not-mem*: *ccpo.admissible Union op* $\subseteq$ ($\lambda A.\ x \notin A$)
by(*rule ccpo.admissibleI*) *auto*

lemma *admissible-eqI*:
assumes *f*: *cont luba orda lub ord* ($\lambda x.\ f\ x$)
and *g*: *cont luba orda lub ord* ($\lambda x.\ g\ x$)

shows *ccpo.admissible luba orda* ($\lambda x.$ *f x* = *g x*)
apply(*rule ccpo.admissibleI*)
apply(*simp-all add*: *contD*[*OF f*] *contD*[*OF g*] *cong*: *image-cong*)
done

corollary *admissible-eq-mcontI* [*cont-intro*]:
⟦ *mcont luba orda lub ord* ($\lambda x.$ *f x*);
mcont luba orda lub ord ($\lambda x.$ *g x*) ⟧
$\Longrightarrow$ *ccpo.admissible luba orda* ($\lambda x.$ *f x* = *g x*)
by(*rule admissible-eqI*)(*auto simp add*: *mcont-def*)

lemma *admissible-iff* [*cont-intro*, *simp*]:
⟦ *ccpo.admissible lub ord* ($\lambda x.$ *P x* $\longrightarrow$ *Q x*); *ccpo.admissible lub ord* ($\lambda x.$ *Q x* $\longrightarrow$ *P x*) ⟧
$\Longrightarrow$ *ccpo.admissible lub ord* ($\lambda x.$ *P x* $\longleftrightarrow$ *Q x*)
by(*subst iff-conv-conj-imp*)(*rule admissible-conj*)

context *ccpo* **begin**

lemma *admissible-leI*:
assumes *f*: *mcont luba orda Sup op* $\leq$ ($\lambda x.$ *f x*)
and *g*: *mcont luba orda Sup op* $\leq$ ($\lambda x.$ *g x*)
shows *ccpo.admissible luba orda* ($\lambda x.$ *f x* $\leq$ *g x*)
proof(*rule ccpo.admissibleI*)
fix *A*
assume *chain*: *Complete-Partial-Order.chain orda A*
and *le*: $\forall x \in A.$ *f x* $\leq$ *g x*
and *False*: *A* $\neq$ {}
have *f* (*luba A*) = $\bigsqcup$(*f* ' *A*) **by**(*simp add*: *mcont-contD*[*OF f*] *chain False*)
also have ... $\leq$ $\bigsqcup$(*g* ' *A*)
proof(*rule ccpo-Sup-least*)
from *chain* **show** *Complete-Partial-Order.chain op* $\leq$ (*f* ' *A*)
by(*rule chain-imageI*)(*rule mcont-monoD*[*OF f*])

fix *x*
assume *x* $\in$ *f* ' *A*
then obtain *y* **where** *y* $\in$ *A* *x* = *f y* **by** *blast* **note** *this*(*2*)
also have *f y* $\leq$ *g y* **using** *le* ‹*y* $\in$ *A*› **by** *simp*
also have *Complete-Partial-Order.chain op* $\leq$ (*g* ' *A*)
using *chain* **by**(*rule chain-imageI*)(*rule mcont-monoD*[*OF g*])
hence *g y* $\leq$ $\bigsqcup$(*g* ' *A*) **by**(*rule ccpo-Sup-upper*)(*simp add*: ‹*y* $\in$ *A*›)
finally show *x* $\leq$ ... **.**
qed
also have ... = *g* (*luba A*) **by**(*simp add*: *mcont-contD*[*OF g*] *chain False*)
finally show *f* (*luba A*) $\leq$ *g* (*luba A*) **.**
qed

end

lemma *admissible-leI*:
  fixes *ord* (**infix** $\sqsubseteq$ *60*) **and** *lub* ($\bigvee$- [*900*] *900*)
  assumes *class.ccpo lub op* $\sqsubseteq$ (*mk-less op* $\sqsubseteq$)
  and *mcont luba orda lub op* $\sqsubseteq$ ($\lambda x.$ *f x*)
  and *mcont luba orda lub op* $\sqsubseteq$ ($\lambda x.$ *g x*)
  shows *ccpo.admissible luba orda* ($\lambda x.$ *f x* $\sqsubseteq$ *g x*)
using *assms* **by**(*rule ccpo.admissible-leI*)

declare *ccpo-class.admissible-leI*[*cont-intro*]

context *ccpo* **begin**

lemma *admissible-not-below*: *ccpo.admissible Sup op* $\leq$ ($\lambda x.$ $\neg$ *op* $\leq$ *x y*)
by(*rule ccpo.admissibleI*)(*simp add*: *ccpo-Sup-below-iff*)

end

lemma (**in** *preorder*) *preorder* [*cont-intro*, *simp*]: *class.preorder op* $\leq$ (*mk-less op* $\leq$)
by(*unfold-locales*)(*auto simp add*: *mk-less-def intro*: *order-trans*)

context *partial-function-definitions* **begin**

lemmas [*cont-intro*, *simp*] =
  admissible-leI[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]
  ccpo.admissible-not-below[*THEN admissible-subst*, *OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]

end

inductive *compact* :: (*'a set* $\Rightarrow$ *'a*) $\Rightarrow$ (*'a* $\Rightarrow$ *'a* $\Rightarrow$ *bool*) $\Rightarrow$ *'a* $\Rightarrow$ *bool*
  for *lub ord x*
where *compact*:
  $\llbracket$ *ccpo.admissible lub ord* ($\lambda y.$ $\neg$ *ord x y*);
    ccpo.admissible lub ord ($\lambda y.$ $x \neq y$) $\rrbracket$
  $\Longrightarrow$ *compact lub ord x*

hide-fact (**open**) *compact*

context *ccpo* **begin**

lemma *compactI*:
  assumes *ccpo.admissible Sup op* $\leq$ ($\lambda y.$ $\neg$ $x \leq y$)
  shows *compact Sup op* $\leq$ *x*
using *assms*
proof(*rule compact.intros*)
  have *neq*: ($\lambda y.$ $x \neq y$) = ($\lambda y.$ $\neg$ $x \leq y$ $\vee$ $\neg$ $y \leq x$) **by**(*auto*)
  show *ccpo.admissible Sup op* $\leq$ ($\lambda y.$ $x \neq y$)

by(*subst neq*)(*rule admissible-disj admissible-not-below assms*)+
qed

lemma *compact-bot*:
assumes $x = Sup\ \{\}$
shows *compact Sup op* $\leq x$
proof(*rule compactI*)
show *ccpo.admissible Sup op* $\leq (\lambda y.\ \neg\ x \leq y)$ **using** *assms*
by(*auto intro*!: *ccpo.admissibleI intro*: *ccpo-Sup-least chain-empty*)
qed

end

lemma *admissible-compact-neq'* [*THEN admissible-subst*, *cont-intro*, *simp*]:
shows *admissible-compact-neq*: *compact lub ord k* $\Longrightarrow$ *ccpo.admissible lub ord* $(\lambda x.\ k \neq x)$
by(*simp add*: *compact.simps*)

lemma *admissible-neq-compact'* [*THEN admissible-subst*, *cont-intro*, *simp*]:
shows *admissible-neq-compact*: *compact lub ord k* $\Longrightarrow$ *ccpo.admissible lub ord* $(\lambda x.\ x \neq k)$
by(*subst eq-commute*)(*rule admissible-compact-neq*)

context *partial-function-definitions* **begin**

lemmas [*cont-intro*, *simp*] = *ccpo.compact-bot*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axion*

end

context *ccpo* **begin**

lemma *fixp-strong-induct*:
assumes [*cont-intro*]: *ccpo.admissible Sup op* $\leq P$
and *mono*: *monotone op* $\leq$ *op* $\leq f$
and *bot*: $P\ (\bigsqcup\{\})$
and *step*: $\bigwedge x.\ [\![\ x \leq$ *ccpo-class.fixp f*; $P\ x\ ]\!] \Longrightarrow P\ (f\ x)$
shows P (*ccpo-class.fixp f*)
proof(*rule fixp-induct*[**where** $P=\lambda x.\ x \leq$ *ccpo-class.fixp f* $\wedge\ P\ x$, *THEN conjunct2*])
note [*cont-intro*] = *admissible-leI*
show *ccpo.admissible Sup op* $\leq (\lambda x.\ x \leq$ *ccpo-class.fixp f* $\wedge\ P\ x)$ **by** *simp*
next
show $\bigsqcup\{\} \leq$ *ccpo-class.fixp f* $\wedge\ P\ (\bigsqcup\{\})$
by(*auto simp add*: *bot intro*: *ccpo-Sup-least chain-empty*)
next
fix x
assume $x \leq$ *ccpo-class.fixp f* $\wedge\ P\ x$
thus $f\ x \leq$ *ccpo-class.fixp f* $\wedge\ P\ (f\ x)$
by(*subst fixp-unfold*[*OF mono*])(*auto dest*: *monotoneD*[*OF mono*] *intro*: *step*)

qed(*rule mono*)

end

context *partial-function-definitions* **begin**

lemma *fixp-strong-induct-uc*:
  fixes F :: $'c \Rightarrow 'c$
    and U :: $'c \Rightarrow 'b \Rightarrow 'a$
    and C :: $('b \Rightarrow 'a) \Rightarrow 'c$
    and P :: $('b \Rightarrow 'a) \Rightarrow bool$
  assumes *mono*: $\bigwedge x.$ *mono-body* $(\lambda f.\ U\ (F\ (C\ f))\ x)$
    and *eq*: $f \equiv C$ (*fixp-fun* $(\lambda f.\ U\ (F\ (C\ f))))$
    and *inverse*: $\bigwedge f.\ U\ (C\ f) = f$
    and *adm*: *ccpo.admissible lub-fun le-fun P*
    and *bot*: P (λ-. *lub* {})
    and *step*: $\bigwedge f'.$ ⟦ $P\ (U\ f')$; *le-fun* $(U\ f')\ (U\ f)$ ⟧ $\Longrightarrow P\ (U\ (F\ f'))$
  shows $P\ (U\ f)$
unfolding *eq inverse*
apply (*rule ccpo.fixp-strong-induct*[*OF ccpo adm*])
apply (*insert mono*, *auto simp*: *monotone-def fun-ord-def bot fun-lub-def*)[*2*]
apply (*rule-tac f'5=C x* **in** *step*)
apply (*simp-all add*: *inverse eq*)
done

end

22.3 *op* = as order

definition *lub-singleton* :: ('*a set* ⇒ '*a*) ⇒ *bool*
where *lub-singleton lub* ⟷ (∀ *a*. *lub* {*a*} = *a*)

definition *the-Sup* :: '*a set* ⇒ '*a*
where *the-Sup A* = (*THE a*. *a* ∈ *A*)

lemma *lub-singleton-the-Sup* [*cont-intro*, *simp*]: *lub-singleton the-Sup*
by(*simp add*: *lub-singleton-def the-Sup-def*)

lemma (**in** *ccpo*) *lub-singleton*: *lub-singleton Sup*
by(*simp add*: *lub-singleton-def*)

lemma (**in** *partial-function-definitions*) *lub-singleton* [*cont-intro*, *simp*]: *lub-singleton lub*
by(*rule ccpo.lub-singleton*)(*rule Partial-Function.ccpo*[*OF partial-function-definitions-axioms*])

lemma *preorder-eq* [*cont-intro*, *simp*]:
  class.preorder op = (*mk-less op* =)
by(*unfold-locales*)(*simp-all add*: *mk-less-def*)

lemma *monotone-eqI* [*cont-intro*]:
  assumes *class.preorder ord* (*mk-less ord*)
  shows *monotone op* = *ord f*
proof –
  interpret *preorder ord mk-less ord* **by** *fact*
  show *?thesis* **by**(*simp add*: *monotone-def*)
qed

lemma *cont-eqI* [*cont-intro*]:
  fixes *f* :: *'a* ⇒ *'b*
  assumes *lub-singleton lub*
  shows *cont the-Sup op* = *lub ord f*
proof(*rule contI*)
  fix *Y* :: *'a set*
  assume *Complete-Partial-Order.chain op* = *Y Y* ≠ {}
  then obtain *a* **where** *Y* = {*a*} **by**(*auto simp add*: *chain-def*)
  thus *f* (*the-Sup Y*) = *lub* (*f* ' *Y*) **using** *assms*
    by(*simp add*: *the-Sup-def lub-singleton-def*)
qed

lemma *mcont-eqI* [*cont-intro*, *simp*]:
  ⟦ *class.preorder ord* (*mk-less ord*); *lub-singleton lub* ⟧
  ⟹ *mcont the-Sup op* = *lub ord f*
by(*simp add*: *mcont-def cont-eqI monotone-eqI*)

22.4 ccpo for products

definition *prod-lub* :: (*'a set* ⇒ *'a*) ⇒ (*'b set* ⇒ *'b*) ⇒ (*'a* × *'b*) *set* ⇒ *'a* × *'b*
where *prod-lub Sup-a Sup-b Y* = (*Sup-a* (*fst* ' *Y*), *Sup-b* (*snd* ' *Y*))

lemma *lub-singleton-prod-lub* [*cont-intro*, *simp*]:
  ⟦ *lub-singleton luba*; *lub-singleton lubb* ⟧ ⟹ *lub-singleton* (*prod-lub luba lubb*)
by(*simp add*: *lub-singleton-def prod-lub-def*)

lemma *prod-lub-empty* [*simp*]: *prod-lub luba lubb* {} = (*luba* {}, *lubb* {})
by(*simp add*: *prod-lub-def*)

lemma *preorder-rel-prodI* [*cont-intro*, *simp*]:
  assumes *class.preorder orda* (*mk-less orda*)
  and *class.preorder ordb* (*mk-less ordb*)
  shows *class.preorder* (*rel-prod orda ordb*) (*mk-less* (*rel-prod orda ordb*))
proof –
  interpret *a*: *preorder orda mk-less orda* **by** *fact*
  interpret *b*: *preorder ordb mk-less ordb* **by** *fact*
  show *?thesis* **by**(*unfold-locales*)(*auto simp add*: *mk-less-def intro*: *a.order-trans b.order-trans*)
qed

lemma *order-rel-prodI*:

assumes *a*: *class.order orda* (*mk-less orda*)
and *b*: *class.order ordb* (*mk-less ordb*)
shows *class.order* (*rel-prod orda ordb*) (*mk-less* (*rel-prod orda ordb*))
(**is** *class.order ?ord ?ord'*)
proof(*intro class.order.intro class.order-axioms.intro*)
interpret *a*: *order orda mk-less orda* **by**(*fact a*)
interpret *b*: *order ordb mk-less ordb* **by**(*fact b*)
show *class.preorder ?ord ?ord'* **by**(*rule preorder-rel-prodI*) *unfold-locales*

fix *x y*
assume *?ord x y ?ord y x*
thus $x = y$ **by**(*cases x y rule*: *prod.exhaust*[*case-product prod.exhaust*]) *auto*
qed

lemma *monotone-rel-prodI*:
assumes *mono2*: $\bigwedge a$. *monotone ordb ordc* (λb. *f* (*a*, *b*))
and *mono1*: $\bigwedge b$. *monotone orda ordc* (λa. *f* (*a*, *b*))
and *a*: *class.preorder orda* (*mk-less orda*)
and *b*: *class.preorder ordb* (*mk-less ordb*)
and *c*: *class.preorder ordc* (*mk-less ordc*)
shows *monotone* (*rel-prod orda ordb*) *ordc f*
proof −
interpret *a*: *preorder orda mk-less orda* **by**(*rule a*)
interpret *b*: *preorder ordb mk-less ordb* **by**(*rule b*)
interpret *c*: *preorder ordc mk-less ordc* **by**(*rule c*)
show *?thesis* **using** *mono2 mono1*
by(*auto 7 2 simp add*: *monotone-def intro*: *c.order-trans*)
qed

lemma *monotone-rel-prodD1*:
assumes *mono*: *monotone* (*rel-prod orda ordb*) *ordc f*
and *preorder*: *class.preorder ordb* (*mk-less ordb*)
shows *monotone orda ordc* (λa. *f* (*a*, *b*))
proof −
interpret *preorder ordb mk-less ordb* **by**(*rule preorder*)
show *?thesis* **using** *mono* **by**(*simp add*: *monotone-def*)
qed

lemma *monotone-rel-prodD2*:
assumes *mono*: *monotone* (*rel-prod orda ordb*) *ordc f*
and *preorder*: *class.preorder orda* (*mk-less orda*)
shows *monotone ordb ordc* (λb. *f* (*a*, *b*))
proof −
interpret *preorder orda mk-less orda* **by**(*rule preorder*)
show *?thesis* **using** *mono* **by**(*simp add*: *monotone-def*)
qed

lemma *monotone-case-prodI*:
⟦ $\bigwedge a$. *monotone ordb ordc* (*f a*); $\bigwedge b$. *monotone orda ordc* (λa. *f a b*);

class.preorder orda (*mk-less orda*); *class.preorder ordb* (*mk-less ordb*);
class.preorder ordc (*mk-less ordc*) ⟧
⟹ *monotone* (*rel-prod orda ordb*) *ordc* (*case-prod f*)
by(*rule monotone-rel-prodI*) *simp-all*

lemma *monotone-case-prodD1*:
assumes *mono*: *monotone* (*rel-prod orda ordb*) *ordc* (*case-prod f*)
and *preorder*: *class.preorder ordb* (*mk-less ordb*)
shows *monotone orda ordc* (*λa. f a b*)
using *monotone-rel-prodD1*[*OF assms*] **by** *simp*

lemma *monotone-case-prodD2*:
assumes *mono*: *monotone* (*rel-prod orda ordb*) *ordc* (*case-prod f*)
and *preorder*: *class.preorder orda* (*mk-less orda*)
shows *monotone ordb ordc* (*f a*)
using *monotone-rel-prodD2*[*OF assms*] **by** *simp*

context
fixes *orda ordb ordc*
assumes *a*: *class.preorder orda* (*mk-less orda*)
and *b*: *class.preorder ordb* (*mk-less ordb*)
and *c*: *class.preorder ordc* (*mk-less ordc*)
begin

lemma *monotone-rel-prod-iff*:
monotone (*rel-prod orda ordb*) *ordc f* ⟷
(∀ *a*. *monotone ordb ordc* (*λb. f* (*a*, *b*))) ∧
(∀ *b*. *monotone orda ordc* (*λa. f* (*a*, *b*)))
using *a b c* **by**(*blast intro*: *monotone-rel-prodI dest*: *monotone-rel-prodD1 monotone-rel-prodD2*)

lemma *monotone-case-prod-iff* [*simp*]:
monotone (*rel-prod orda ordb*) *ordc* (*case-prod f*) ⟷
(∀ *a*. *monotone ordb ordc* (*f a*)) ∧ (∀ *b*. *monotone orda ordc* (*λa. f a b*))
by(*simp add*: *monotone-rel-prod-iff*)

end

lemma *monotone-case-prod-apply-iff*:
monotone orda ordb (*λx.* (*case-prod f x*) *y*) ⟷ *monotone orda ordb* (*case-prod*
(*λa b. f a b y*))
by(*simp add*: *monotone-def*)

lemma *monotone-case-prod-applyD*:
monotone orda ordb (*λx.* (*case-prod f x*) *y*)
⟹ *monotone orda ordb* (*case-prod* (*λa b. f a b y*))
by(*simp add*: *monotone-case-prod-apply-iff*)

lemma *monotone-case-prod-applyI*:
monotone orda ordb (*case-prod* (*λa b. f a b y*))

$\Longrightarrow$ *monotone orda ordb* $(\lambda x.$ (*case-prod f x*) $y)$
by(*simp add*: *monotone-case-prod-apply-iff*)

lemma *cont-case-prod-apply-iff*:
cont luba orda lubb ordb $(\lambda x.$ (*case-prod f x*) $y) \longleftrightarrow$ *cont luba orda lubb ordb* (*case-prod* $(\lambda a\ b.\ f\ a\ b\ y))$
by(*simp add*: *cont-def split-def*)

lemma *cont-case-prod-applyI*:
cont luba orda lubb ordb (*case-prod* $(\lambda a\ b.\ f\ a\ b\ y))$
$\Longrightarrow$ *cont luba orda lubb ordb* $(\lambda x.$ (*case-prod f x*) $y)$
by(*simp add*: *cont-case-prod-apply-iff*)

lemma *cont-case-prod-applyD*:
cont luba orda lubb ordb $(\lambda x.$ (*case-prod f x*) $y)$
$\Longrightarrow$ *cont luba orda lubb ordb* (*case-prod* $(\lambda a\ b.\ f\ a\ b\ y))$
by(*simp add*: *cont-case-prod-apply-iff*)

lemma *mcont-case-prod-apply-iff* [*simp*]:
mcont luba orda lubb ordb $(\lambda x.$ (*case-prod f x*) $y) \longleftrightarrow$
mcont luba orda lubb ordb (*case-prod* $(\lambda a\ b.\ f\ a\ b\ y))$
by(*simp add*: *mcont-def monotone-case-prod-apply-iff cont-case-prod-apply-iff*)

lemma *cont-prodD1*:
assumes *cont*: *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *lubc ordc f*
and *class.preorder orda* (*mk-less orda*)
and *luba*: *lub-singleton luba*
shows *cont lubb ordb lubc ordc* $(\lambda y.\ f\ (x, y))$
proof(*rule contI*)
interpret *preorder orda mk-less orda* **by** *fact*

fix $Y :: 'b$ *set*
let $?Y = \{x\} \times Y$
assume *Complete-Partial-Order.chain ordb* Y $Y \neq \{\}$
hence *Complete-Partial-Order.chain* (*rel-prod orda ordb*) $?Y$ $?Y \neq \{\}$
by(*simp-all add*: *chain-def*)
with *cont* **have** f (*prod-lub luba lubb* $?Y$) $=$ *lubc* $(f$ ' $?Y)$ **by**(*rule contD*)
moreover have f ' $?Y = (\lambda y.\ f\ (x, y))$ ' Y **by** *auto*
ultimately show $f\ (x,$ *lubb* $Y) =$ *lubc* $((\lambda y.\ f\ (x, y))$ ' $Y)$ **using** *luba*
by(*simp add*: *prod-lub-def* ‹$Y \neq \{\}$› *lub-singleton-def*)
qed

lemma *cont-prodD2*:
assumes *cont*: *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *lubc ordc f*
and *class.preorder ordb* (*mk-less ordb*)
and *lubb*: *lub-singleton lubb*
shows *cont luba orda lubc ordc* $(\lambda x.\ f\ (x, y))$
proof(*rule contI*)

interpret *preorder ordb mk-less ordb* **by** *fact*

fix *Y*
assume *Y*: *Complete-Partial-Order.chain orda Y Y* ≠ {}
let *?Y* = *Y* × {*y*}
have *f* (*luba Y*, *y*) = *f* (*prod-lub luba lubb ?Y*)
using *lubb* **by**(*simp add*: *prod-lub-def Y lub-singleton-def*)
also from *Y* **have** *Complete-Partial-Order.chain* (*rel-prod orda ordb*) *?Y ?Y* ≠ {}
by(*simp-all add*: *chain-def*)
with *cont* **have** *f* (*prod-lub luba lubb ?Y*) = *lubc* (*f* ' *?Y*) **by**(*rule contD*)
also have *f* ' *?Y* = (λ*x*. *f* (*x*, *y*)) ' *Y* **by** *auto*
finally show *f* (*luba Y*, *y*) = *lubc* . . . **.**
qed

lemma *cont-case-prodD1*:
assumes *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *lubc ordc* (*case-prod f*)
and *class.preorder orda* (*mk-less orda*)
and *lub-singleton luba*
shows *cont lubb ordb lubc ordc* (*f x*)
using *cont-prodD1*[*OF assms*] **by** *simp*

lemma *cont-case-prodD2*:
assumes *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *lubc ordc* (*case-prod f*)
and *class.preorder ordb* (*mk-less ordb*)
and *lub-singleton lubb*
shows *cont luba orda lubc ordc* (λ*x*. *f x y*)
using *cont-prodD2*[*OF assms*] **by** *simp*

context *ccpo* **begin**

lemma *cont-prodI*:
assumes *mono*: *monotone* (*rel-prod orda ordb*) *op* ≤ *f*
and *cont1*: ⋀*x*. *cont lubb ordb Sup op* ≤ (λ*y*. *f* (*x*, *y*))
and *cont2*: ⋀*y*. *cont luba orda Sup op* ≤ (λ*x*. *f* (*x*, *y*))
and *class.preorder orda* (*mk-less orda*)
and *class.preorder ordb* (*mk-less ordb*)
shows *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *Sup op* ≤ *f*
proof(*rule contI*)
interpret *a*: *preorder orda mk-less orda* **by** *fact*
interpret *b*: *preorder ordb mk-less ordb* **by** *fact*

fix *Y*
assume *chain*: *Complete-Partial-Order.chain* (*rel-prod orda ordb*) *Y*
and *Y* ≠ {}
have *f* (*prod-lub luba lubb Y*) = *f* (*luba* (*fst* ' *Y*), *lubb* (*snd* ' *Y*))
by(*simp add*: *prod-lub-def*)
also from *cont2* **have** *f* (*luba* (*fst* ' *Y*), *lubb* (*snd* ' *Y*)) = ⨆((λ*x*. *f* (*x*, *lubb* (*snd* ' *Y*))) ' *fst* ' *Y*)

by(*rule contD*)(*simp-all add*: *chain-rel-prodD1*[*OF chain*] ‹$Y \neq \{\}$›)
also from *cont1* **have** $\bigwedge x.\ f\ (x,\ lubb\ (snd\ `\ Y)) = \bigsqcup((\lambda y.\ f\ (x,\ y))\ `\ snd\ `\ Y)$
by(*rule contD*)(*simp-all add*: *chain-rel-prodD2*[*OF chain*] ‹$Y \neq \{\}$›)
hence $\bigsqcup((\lambda x.\ f\ (x,\ lubb\ (snd\ `\ Y)))\ `\ fst\ `\ Y) = \bigsqcup((\lambda x.\ \ldots\ x)\ `\ fst\ `\ Y)$ **by** *simp*
also have $\ldots = \bigsqcup((\lambda x.\ f\ (fst\ x,\ snd\ x))\ `\ Y)$
unfolding *image-image split-def* **using** *chain*
apply(*rule diag-Sup*)
using *monotoneD*[*OF mono*]
by(*auto intro*: *monotoneI*)
finally show $f\ (prod\text{-}lub\ luba\ lubb\ Y) = \bigsqcup(f\ `\ Y)$ **by** *simp*
qed

lemma *cont-case-prodI*:
assumes *monotone* (*rel-prod orda ordb*) *op* ≤ (*case-prod f*)
and $\bigwedge x.$ *cont lubb ordb Sup op* ≤ ($\lambda y.$ *f x y*)
and $\bigwedge y.$ *cont luba orda Sup op* ≤ ($\lambda x.$ *f x y*)
and *class.preorder orda* (*mk-less orda*)
and *class.preorder ordb* (*mk-less ordb*)
shows *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *Sup op* ≤ (*case-prod f*)
by(*rule cont-prodI*)(*simp-all add*: *assms*)

lemma *cont-case-prod-iff*:
⟦ *monotone* (*rel-prod orda ordb*) *op* ≤ (*case-prod f*);
class.preorder orda (*mk-less orda*); *lub-singleton luba*;
class.preorder ordb (*mk-less ordb*); *lub-singleton lubb* ⟧
⟹ *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *Sup op* ≤ (*case-prod f*) ⟷
(∀ *x*. *cont lubb ordb Sup op* ≤ ($\lambda y.$ *f x y*)) ∧ (∀ *y*. *cont luba orda Sup op* ≤ ($\lambda x.$ *f x y*))
by(*blast dest*: *cont-case-prodD1 cont-case-prodD2 intro*: *cont-case-prodI*)

end

context *partial-function-definitions* **begin**

lemma *mono2mono2*:
assumes *f*: *monotone* (*rel-prod ordb ordc*) *leq* ($\lambda(x,\ y).$ *f x y*)
and *t*: *monotone orda ordb* ($\lambda x.$ *t x*)
and *t'*: *monotone orda ordc* ($\lambda x.$ *t' x*)
shows *monotone orda leq* ($\lambda x.$ *f* (*t x*) (*t' x*))
proof(*rule monotoneI*)
fix *x y*
assume *orda x y*
hence *rel-prod ordb ordc* (*t x*, *t' x*) (*t y*, *t' y*)
using *t t'* **by**(*auto dest*: *monotoneD*)
from *monotoneD*[*OF f this*] **show** *leq* (*f* (*t x*) (*t' x*)) (*f* (*t y*) (*t' y*)) **by** *simp*
qed

lemma *cont-case-prodI* [*cont-intro*]:

$\llbracket$ *monotone* (*rel-prod orda ordb*) *leq* (*case-prod f*);
$\bigwedge x$. *cont lubb ordb lub leq* (λy. *f x y*);
$\bigwedge y$. *cont luba orda lub leq* (λx. *f x y*);
class.preorder orda (*mk-less orda*);
class.preorder ordb (*mk-less ordb*) $\rrbracket$
$\Longrightarrow$ *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *lub leq* (*case-prod f*)
by(*rule ccpo.cont-case-prodI*)(*rule Partial-Function.ccpo*[*OF partial-function-definitions-axioms*])

lemma *cont-case-prod-iff*:
$\llbracket$ *monotone* (*rel-prod orda ordb*) *leq* (*case-prod f*);
class.preorder orda (*mk-less orda*); *lub-singleton luba*;
class.preorder ordb (*mk-less ordb*); *lub-singleton lubb* $\rrbracket$
$\Longrightarrow$ *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *lub leq* (*case-prod f*) $\longleftrightarrow$
($\forall x$. *cont lubb ordb lub leq* (λy. *f x y*)) $\wedge$ ($\forall y$. *cont luba orda lub leq* (λx. *f x y*))
by(*blast dest*: *cont-case-prodD1 cont-case-prodD2 intro*: *cont-case-prodI*)

lemma *mcont-case-prod-iff* [*simp*]:
$\llbracket$ *class.preorder orda* (*mk-less orda*); *lub-singleton luba*;
class.preorder ordb (*mk-less ordb*); *lub-singleton lubb* $\rrbracket$
$\Longrightarrow$ *mcont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *lub leq* (*case-prod f*) $\longleftrightarrow$
($\forall x$. *mcont lubb ordb lub leq* (λy. *f x y*)) $\wedge$ ($\forall y$. *mcont luba orda lub leq* (λx. *f x y*))
unfolding *mcont-def* **by**(*auto simp add*: *cont-case-prod-iff*)

end

lemma *mono2mono-case-prod* [*cont-intro*]:
assumes $\bigwedge x\ y$. *monotone orda ordb* (λf. *pair f x y*)
shows *monotone orda ordb* (λf. *case-prod* (*pair f*) *x*)
by(*rule monotoneI*)(*auto split*: *prod.split dest*: *monotoneD*[*OF assms*])

22.5 Complete lattices as ccpo

context *complete-lattice* **begin**

lemma *complete-lattice-ccpo*: *class.ccpo Sup op* $\leq$ *op* $<$
by(*unfold-locales*)(*fast intro*: *Sup-upper Sup-least*)+

lemma *complete-lattice-ccpo'*: *class.ccpo Sup op* $\leq$ (*mk-less op* $\leq$)
by(*unfold-locales*)(*auto simp add*: *mk-less-def intro*: *Sup-upper Sup-least*)

lemma *complete-lattice-partial-function-definitions*:
partial-function-definitions op $\leq$ *Sup*
by(*unfold-locales*)(*auto intro*: *Sup-least Sup-upper*)

lemma *complete-lattice-partial-function-definitions-dual*:
partial-function-definitions op $\geq$ *Inf*
by(*unfold-locales*)(*auto intro*: *Inf-lower Inf-greatest*)

```
lemmas [cont-intro, simp] =
  Partial-Function.ccpo[OF complete-lattice-partial-function-definitions]
  Partial-Function.ccpo[OF complete-lattice-partial-function-definitions-dual]

lemma mono2mono-inf:
  assumes f: monotone ord op ≤ (λx. f x)
  and g: monotone ord op ≤ (λx. g x)
  shows monotone ord op ≤ (λx. f x ⊓ g x)
by(auto 4 3 dest: monotoneD[OF f] monotoneD[OF g] intro: le-infI1 le-infI2 in-
tro!: monotoneI)

lemma mcont-const [simp]: mcont lub ord Sup op ≤ (λ-. c)
by(rule ccpo.mcont-const[OF complete-lattice-ccpo])

lemma mono2mono-sup:
  assumes f: monotone ord op ≤ (λx. f x)
  and g: monotone ord op ≤ (λx. g x)
  shows monotone ord op ≤ (λx. f x ⊔ g x)
by(auto 4 3 intro!: monotoneI intro: sup.coboundedI1 sup.coboundedI2 dest: mono-
toneD[OF f] monotoneD[OF g])

lemma Sup-image-sup:
  assumes Y ≠ {}
  shows ⨆(op ⊔ x ‘ Y) = x ⊔ ⨆ Y
proof(rule Sup-eqI)
  fix y
  assume y ∈ op ⊔ x ‘ Y
  then obtain z where y = x ⊔ z and z ∈ Y by blast
  from ‹z ∈ Y› have z ≤ ⨆ Y by(rule Sup-upper)
  with - show y ≤ x ⊔ ⨆ Y unfolding ‹y = x ⊔ z› by(rule sup-mono) simp
next
  fix y
  assume upper: ⋀z. z ∈ op ⊔ x ‘ Y ⟹ z ≤ y
  show x ⊔ ⨆ Y ≤ y unfolding Sup-insert[symmetric]
  proof(rule Sup-least)
    fix z
    assume z ∈ insert x Y
    from assms obtain z' where z' ∈ Y by blast
    let ?z = if z ∈ Y then x ⊔ z else x ⊔ z'
    have z ≤ x ⊔ ?z using ‹z' ∈ Y› ‹z ∈ insert x Y› by auto
    also have ... ≤ y by(rule upper)(auto split: if-split-asm intro: ‹z' ∈ Y›)
    finally show z ≤ y .
  qed
qed

lemma mcont-sup1: mcont Sup op ≤ Sup op ≤ (λy. x ⊔ y)
by(auto 4 3 simp add: mcont-def sup.coboundedI1 sup.coboundedI2 intro!: mono-
toneI contI intro: Sup-image-sup[symmetric])
```

lemma *mcont-sup2*: *mcont Sup op* ≤ *Sup op* ≤ (λ*x*. *x* ⊔ *y*)
by(*subst sup-commute*)(*rule mcont-sup1*)

lemma *mcont2mcont-sup* [*cont-intro*, *simp*]:
⟦ *mcont lub ord Sup op* ≤ (λ*x*. *f x*);
mcont lub ord Sup op ≤ (λ*x*. *g x*) ⟧
⟹ *mcont lub ord Sup op* ≤ (λ*x*. *f x* ⊔ *g x*)
by(*best intro*: *ccpo.mcont2mcont'*[*OF complete-lattice-ccpo*] *mcont-sup1 mcont-sup2 ccpo.mcont-const*[*OF complete-lattice-ccpo*])

end

lemmas [*cont-intro*] = *admissible-leI*[*OF complete-lattice-ccpo'*]

context *complete-distrib-lattice* **begin**

lemma *mcont-inf1*: *mcont Sup op* ≤ *Sup op* ≤ (λ*y*. *x* ⊓ *y*)
by(*auto intro*: *monotoneI contI simp add*: *le-infI2 inf-Sup mcont-def*)

lemma *mcont-inf2*: *mcont Sup op* ≤ *Sup op* ≤ (λ*x*. *x* ⊓ *y*)
by(*auto intro*: *monotoneI contI simp add*: *le-infI1 Sup-inf mcont-def*)

lemma *mcont2mcont-inf* [*cont-intro*, *simp*]:
⟦ *mcont lub ord Sup op* ≤ (λ*x*. *f x*);
mcont lub ord Sup op ≤ (λ*x*. *g x*) ⟧
⟹ *mcont lub ord Sup op* ≤ (λ*x*. *f x* ⊓ *g x*)
by(*best intro*: *ccpo.mcont2mcont'*[*OF complete-lattice-ccpo*] *mcont-inf1 mcont-inf2 ccpo.mcont-const*[*OF complete-lattice-ccpo*])

end

interpretation *lfp*: *partial-function-definitions op* ≤ :: - :: *complete-lattice* ⇒ - *Sup*
by(*rule complete-lattice-partial-function-definitions*)

declaration ‹*Partial-Function.init lfp* @{*term lfp.fixp-fun*} @{*term lfp.mono-body*}
@{*thm lfp.fixp-rule-uc*} @{*thm lfp.fixp-induct-uc*} *NONE*›

interpretation *gfp*: *partial-function-definitions op* ≥ :: - :: *complete-lattice* ⇒ - *Inf*
by(*rule complete-lattice-partial-function-definitions-dual*)

declaration ‹*Partial-Function.init gfp* @{*term gfp.fixp-fun*} @{*term gfp.mono-body*}
@{*thm gfp.fixp-rule-uc*} @{*thm gfp.fixp-induct-uc*} *NONE*›

lemma *insert-mono* [*partial-function-mono*]:
monotone (*fun-ord op* ⊆) *op* ⊆ *A* ⟹ *monotone* (*fun-ord op* ⊆) *op* ⊆ (λ*y*. *insert x* (*A y*))
by(*rule monotoneI*)(*auto simp add*: *fun-ord-def dest*: *monotoneD*)

lemma *mono2mono-insert* [*THEN lfp.mono2mono*, *cont-intro*, *simp*]:
shows *monotone-insert*: *monotone op* $\subseteq$ *op* $\subseteq$ (*insert x*)
by(*rule monotoneI*) *blast*

lemma *mcont2mcont-insert*[*THEN lfp.mcont2mcont*, *cont-intro*, *simp*]:
shows *mcont-insert*: *mcont Union op* $\subseteq$ *Union op* $\subseteq$ (*insert x*)
by(*blast intro*: *mcontI contI monotone-insert*)

lemma *mono2mono-image* [*THEN lfp.mono2mono*, *cont-intro*, *simp*]:
shows *monotone-image*: *monotone op* $\subseteq$ *op* $\subseteq$ (*op* ‘ *f*)
by(*rule monotoneI*) *blast*

lemma *cont-image*: *cont Union op* $\subseteq$ *Union op* $\subseteq$ (*op* ‘ *f*)
by(*rule contI*)(*auto*)

lemma *mcont2mcont-image* [*THEN lfp.mcont2mcont*, *cont-intro*, *simp*]:
shows *mcont-image*: *mcont Union op* $\subseteq$ *Union op* $\subseteq$ (*op* ‘ *f*)
by(*blast intro*: *mcontI monotone-image cont-image*)

context *complete-lattice* **begin**

lemma *monotone-Sup* [*cont-intro*, *simp*]:
monotone ord op $\subseteq$ *f* $\Longrightarrow$ *monotone ord op* $\leq$ ($\lambda x.$ $\bigsqcup$*f x*)
by(*blast intro*: *monotoneI Sup-least Sup-upper dest*: *monotoneD*)

lemma *cont-Sup*:
assumes *cont lub ord Union op* $\subseteq$ *f*
shows *cont lub ord Sup op* $\leq$ ($\lambda x.$ $\bigsqcup$*f x*)
apply(*rule contI*)
apply(*simp add*: *contD*[*OF assms*])
apply(*blast intro*: *Sup-least Sup-upper order-trans antisym*)
done

lemma *mcont-Sup*: *mcont lub ord Union op* $\subseteq$ *f* $\Longrightarrow$ *mcont lub ord Sup op* $\leq$ ($\lambda x.$ $\bigsqcup$*f x*)
unfolding *mcont-def* **by**(*blast intro*: *monotone-Sup cont-Sup*)

lemma *monotone-SUP*:
$\llbracket$ *monotone ord op* $\subseteq$ *f*; $\bigwedge y.$ *monotone ord op* $\leq$ ($\lambda x.$ *g x y*) $\rrbracket$ $\Longrightarrow$ *monotone ord op* $\leq$ ($\lambda x.$ $\bigsqcup$ *y*$\in$*f x*. *g x y*)
by(*rule monotoneI*)(*blast dest*: *monotoneD intro*: *Sup-upper order-trans intro*!: *Sup-least*)

lemma *monotone-SUP2*:
($\bigwedge y.$ *y* $\in$ *A* $\Longrightarrow$ *monotone ord op* $\leq$ ($\lambda x.$ *g x y*)) $\Longrightarrow$ *monotone ord op* $\leq$ ($\lambda x.$ $\bigsqcup$ *y*$\in$*A*. *g x y*)
by(*rule monotoneI*)(*blast intro*: *Sup-upper order-trans dest*: *monotoneD intro*!: *Sup-least*)

lemma *cont-SUP*:
  assumes f: *mcont lub ord Union op* $\subseteq$ *f*
  and g: $\bigwedge y$. *mcont lub ord Sup op* $\leq$ (λx. *g x y*)
  shows *cont lub ord Sup op* $\leq$ (λx. $\bigsqcup y \in f\ x$. *g x y*)
proof(*rule contI*)
  fix *Y*
  assume *chain*: *Complete-Partial-Order.chain ord Y*
    and Y: $Y \neq \{\}$
  show $\bigsqcup(g\ (lub\ Y)$ ' $f\ (lub\ Y)) = \bigsqcup((\lambda x. \bigsqcup(g\ x$ ' $f\ x))$ ' $Y)$ (**is** *?lhs* = *?rhs*)
  proof(*rule antisym*)
    show *?lhs* $\leq$ *?rhs*
    proof(*rule Sup-least*)
      fix *x*
      assume $x \in g\ (lub\ Y)$ ' $f\ (lub\ Y)$
      with *mcont-contD*[*OF f chain Y*] *mcont-contD*[*OF g chain Y*]
      obtain *y z* **where** $y \in Y$ $z \in f\ y$
        and x: $x = \bigsqcup((\lambda x.\ g\ x\ z)$ ' $Y)$ **by** *auto*
      show $x \leq$ *?rhs* **unfolding** *x*
      proof(*rule Sup-least*)
        fix *u*
        assume $u \in (\lambda x.\ g\ x\ z)$ ' Y
        then obtain y' **where** $u = g\ y'\ z$ $y' \in Y$ **by** *auto*
        from *chain* ‹$y \in Y$› ‹$y' \in Y$› **have** $ord\ y\ y' \vee ord\ y'\ y$ **by**(*rule chainD*)
        thus $u \leq$ *?rhs*
        proof
          note ‹$u = g\ y'\ z$› **also**
          assume $ord\ y\ y'$
          with *f* **have** $f\ y \subseteq f\ y'$ **by**(*rule mcont-monoD*)
          with ‹$z \in f\ y$›
          have $g\ y'\ z \leq \bigsqcup(g\ y'$ ' $f\ y')$ **by**(*auto intro*: *Sup-upper*)
          also have ... $\leq$ *?rhs* **using** ‹$y' \in Y$› **by**(*auto intro*: *Sup-upper*)
          finally show *?thesis* **.**
        next
          note ‹$u = g\ y'\ z$› **also**
          assume $ord\ y'\ y$
          with *g* **have** $g\ y'\ z \leq g\ y\ z$ **by**(*rule mcont-monoD*)
          also have ... $\leq \bigsqcup(g\ y$ ' $f\ y)$ **using** ‹$z \in f\ y$›
            by(*auto intro*: *Sup-upper*)
          also have ... $\leq$ *?rhs* **using** ‹$y \in Y$› **by**(*auto intro*: *Sup-upper*)
          finally show *?thesis* **.**
        qed
      qed
    qed
  next
    show *?rhs* $\leq$ *?lhs*
    proof(*rule Sup-least*)
      fix *x*
      assume $x \in (\lambda x. \bigsqcup(g\ x$ ' $f\ x))$ ' Y

then obtain *y* **where** *x*: $x = \bigsqcup(g\ y\ `\ f\ y)$ **and** $y \in Y$ **by** *auto*
show $x \leq$ *?lhs* **unfolding** *x*
proof(*rule Sup-least*)
fix *u*
assume $u \in g\ y\ `\ f\ y$
then obtain *z* **where** $u = g\ y\ z\ z \in f\ y$ **by** *auto*
note ‹$u = g\ y\ z$›
also have $g\ y\ z \leq \bigsqcup((\lambda x.\ g\ x\ z)\ `\ Y)$
using ‹$y \in Y$› **by**(*auto intro*: *Sup-upper*)
also have $\ldots = g\ (lub\ Y)\ z$ **by**(*simp add*: *mcont-contD*[*OF g chain Y*])
also have $\ldots \leq$ *?lhs* **using** ‹$z \in f\ y$› ‹$y \in Y$›
by(*auto intro*: *Sup-upper simp add*: *mcont-contD*[*OF f chain Y*])
finally show $u \leq$ *?lhs* **.**
qed
qed
qed
qed

lemma *mcont-SUP* [*cont-intro*, *simp*]:
⟦ *mcont lub ord Union op* ⊆ *f*; $\bigwedge y$. *mcont lub ord Sup op* ≤ $(\lambda x.\ g\ x\ y)$ ⟧
⟹ *mcont lub ord Sup op* ≤ $(\lambda x.\ \bigsqcup y{\in}f\ x.\ g\ x\ y)$
by(*blast intro*: *mcontI cont-SUP*[*OF assms*] *monotone-SUP mcont-mono*)

end

lemma *admissible-Ball* [*cont-intro*, *simp*]:
⟦ $\bigwedge x$. *ccpo.admissible lub ord* $(\lambda A.\ P\ A\ x)$;
mcont lub ord Union op ⊆ *f*;
class.ccpo lub ord (*mk-less ord*) ⟧
⟹ *ccpo.admissible lub ord* $(\lambda A.\ \forall x{\in}f\ A.\ P\ A\ x)$
unfolding *Ball-def* **by** *simp*

lemma *admissible-Bex'*[*THEN admissible-subst*, *cont-intro*, *simp*]:
shows *admissible-Bex*: *ccpo.admissible Union op* ⊆ $(\lambda A.\ \exists x{\in}A.\ P\ x)$
by(*rule ccpo.admissibleI*)(*auto*)

22.6 Parallel fixpoint induction

context
fixes *luba* :: *'a set* ⇒ *'a*
and *orda* :: *'a* ⇒ *'a* ⇒ *bool*
and *lubb* :: *'b set* ⇒ *'b*
and *ordb* :: *'b* ⇒ *'b* ⇒ *bool*
assumes *a*: *class.ccpo luba orda* (*mk-less orda*)
and *b*: *class.ccpo lubb ordb* (*mk-less ordb*)
begin

interpretation *a*: *ccpo luba orda mk-less orda* **by**(*rule a*)
interpretation *b*: *ccpo lubb ordb mk-less ordb* **by**(*rule b*)

lemma *ccpo-rel-prodI*:
class.ccpo (*prod-lub luba lubb*) (*rel-prod orda ordb*) (*mk-less* (*rel-prod orda ordb*))
(**is** *class.ccpo ?lub ?ord ?ord'*)
proof(*intro class.ccpo.intro class.ccpo-axioms.intro*)
show *class.order ?ord ?ord'* **by**(*rule order-rel-prodI*) *intro-locales*
qed(*auto 4 4 simp add*: *prod-lub-def intro*: *a.ccpo-Sup-upper b.ccpo-Sup-upper a.ccpo-Sup-least b.ccpo-Sup-least rev-image-eqI dest*: *chain-rel-prodD1 chain-rel-prodD2*)

interpretation *ab*: *ccpo prod-lub luba lubb rel-prod orda ordb mk-less* (*rel-prod orda ordb*)
by(*rule ccpo-rel-prodI*)

lemma *monotone-map-prod* [*simp*]:
monotone (*rel-prod orda ordb*) (*rel-prod ordc ordd*) (*map-prod f g*) $\longleftrightarrow$
monotone orda ordc f $\wedge$ *monotone ordb ordd g*
by(*auto simp add*: *monotone-def*)

lemma *parallel-fixp-induct*:
assumes *adm*: *ccpo.admissible* (*prod-lub luba lubb*) (*rel-prod orda ordb*) (λx. *P* (*fst x*) (*snd x*))
and *f*: *monotone orda orda f*
and *g*: *monotone ordb ordb g*
and *bot*: *P* (*luba* {}) (*lubb* {})
and *step*: $\bigwedge x\ y.\ P\ x\ y \Longrightarrow P\ (f\ x)\ (g\ y)$
shows *P* (*ccpo.fixp luba orda f*) (*ccpo.fixp lubb ordb g*)
proof −
let *?lub* = *prod-lub luba lubb*
and *?ord* = *rel-prod orda ordb*
and *?P* = $\lambda(x, y)$. *P x y*
from *adm* **have** *adm'*: *ccpo.admissible ?lub ?ord ?P* **by**(*simp add*: *split-def*)
hence *?P* (*ccpo.fixp* (*prod-lub luba lubb*) (*rel-prod orda ordb*) (*map-prod f g*))
by(*rule ab.fixp-induct*)(*auto simp add*: *f g step bot*)
also have *ccpo.fixp* (*prod-lub luba lubb*) (*rel-prod orda ordb*) (*map-prod f g*) =
(*ccpo.fixp luba orda f*, *ccpo.fixp lubb ordb g*) (**is** *?lhs* = (*?rhs1*, *?rhs2*))
proof(*rule ab.antisym*)
have *ccpo.admissible ?lub ?ord* (λxy. *?ord xy* (*?rhs1*, *?rhs2*))
by(*rule admissible-leI*[*OF ccpo-rel-prodI*])(*auto simp add*: *prod-lub-def chain-empty intro*: *a.ccpo-Sup-least b.ccpo-Sup-least*)
thus *?ord ?lhs* (*?rhs1*, *?rhs2*)
by(*rule ab.fixp-induct*)(*auto 4 3 dest*: *monotoneD*[*OF f*] *monotoneD*[*OF g*] *simp add*: *b.fixp-unfold*[*OF g*, *symmetric*] *a.fixp-unfold*[*OF f*, *symmetric*] *f g intro*: *a.ccpo-Sup-least b.ccpo-Sup-least chain-empty*)
next
have *ccpo.admissible luba orda* (λx. *orda x* (*fst ?lhs*))
by(*rule admissible-leI*[*OF a*])(*auto intro*: *a.ccpo-Sup-least simp add*: *chain-empty*)
hence *orda ?rhs1* (*fst ?lhs*) **using** *f*
proof(*rule a.fixp-induct*)
fix *x*

assume *orda x* (*fst ?lhs*)
thus *orda* (*f x*) (*fst ?lhs*)
by(*subst ab.fixp-unfold*)(*auto simp add*: *f g dest*: *monotoneD*[*OF f*])
qed(*auto intro*: *a.ccpo-Sup-least chain-empty*)
moreover
have *ccpo.admissible lubb ordb* (λy. *ordb y* (*snd ?lhs*))
by(*rule admissible-leI*[*OF b*])(*auto intro*: *b.ccpo-Sup-least simp add*: *chain-empty*)
hence *ordb ?rhs2* (*snd ?lhs*) **using** *g*
proof(*rule b.fixp-induct*)
fix *y*
assume *ordb y* (*snd ?lhs*)
thus *ordb* (*g y*) (*snd ?lhs*)
by(*subst ab.fixp-unfold*)(*auto simp add*: *f g dest*: *monotoneD*[*OF g*])
qed(*auto intro*: *b.ccpo-Sup-least chain-empty*)
ultimately show *?ord* (*?rhs1*, *?rhs2*) *?lhs*
by(*simp add*: *rel-prod-conv split-beta*)
qed
finally show *?thesis* **by** *simp*
qed

end

lemma *parallel-fixp-induct-uc*:
assumes *a*: *partial-function-definitions orda luba*
and *b*: *partial-function-definitions ordb lubb*
and *F*: $\bigwedge x$. *monotone* (*fun-ord orda*) *orda* (λf. *U1* (*F* (*C1 f*)) *x*)
and *G*: $\bigwedge y$. *monotone* (*fun-ord ordb*) *ordb* (λg. *U2* (*G* (*C2 g*)) *y*)
and *eq1*: $f \equiv$ *C1* (*ccpo.fixp* (*fun-lub luba*) (*fun-ord orda*) (λf. *U1* (*F* (*C1 f*))))
and *eq2*: $g \equiv$ *C2* (*ccpo.fixp* (*fun-lub lubb*) (*fun-ord ordb*) (λg. *U2* (*G* (*C2 g*))))
and *inverse*: $\bigwedge f$. *U1* (*C1 f*) = *f*
and *inverse2*: $\bigwedge g$. *U2* (*C2 g*) = *g*
and *adm*: *ccpo.admissible* (*prod-lub* (*fun-lub luba*) (*fun-lub lubb*)) (*rel-prod* (*fun-ord orda*) (*fun-ord ordb*)) (λx. *P* (*fst x*) (*snd x*))
and *bot*: *P* (λ-. *luba* {}) (λ-. *lubb* {})
and *step*: $\bigwedge f\ g$. *P* (*U1 f*) (*U2 g*) $\Longrightarrow$ *P* (*U1* (*F f*)) (*U2* (*G g*))
shows *P* (*U1 f*) (*U2 g*)
apply(*unfold eq1 eq2 inverse inverse2*)
apply(*rule parallel-fixp-induct*[*OF partial-function-definitions.ccpo*[*OF a*] *partial-function-definitions.ccpo*[*OF b*] *adm*])
using *F* **apply**(*simp add*: *monotone-def fun-ord-def*)
using *G* **apply**(*simp add*: *monotone-def fun-ord-def*)
apply(*simp add*: *fun-lub-def bot*)
apply(*rule step*, *simp add*: *inverse inverse2*)
done

lemmas *parallel-fixp-induct-1-1* = *parallel-fixp-induct-uc*[
of - - - - λx. *x* - λx. *x* λx. *x* - λx. *x*,
OF - - - - - - *refl refl*]

lemmas *parallel-fixp-induct-2-2* = *parallel-fixp-induct-uc*[
of - - - - *case-prod* - *curry case-prod* - *curry*,
where *P*=λ*f g*. *P* (*curry f*) (*curry g*),
unfolded case-prod-curry curry-case-prod curry-K,
OF - - - - - - *refl refl*]
for *P*

lemma *monotone-fst*: *monotone* (*rel-prod orda ordb*) *orda fst*
by(*auto intro*: *monotoneI*)

lemma *mcont-fst*: *mcont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *luba orda fst*
by(*auto intro*!: *mcontI monotoneI contI simp add*: *prod-lub-def*)

lemma *mcont2mcont-fst* [*cont-intro*, *simp*]:
mcont lub ord (*prod-lub luba lubb*) (*rel-prod orda ordb*) *t*
⟹ *mcont lub ord luba orda* (λ*x*. *fst* (*t x*))
by(*auto intro*!: *mcontI monotoneI contI dest*: *mcont-monoD mcont-contD simp add*: *rel-prod-sel split-beta prod-lub-def image-image*)

lemma *monotone-snd*: *monotone* (*rel-prod orda ordb*) *ordb snd*
by(*auto intro*: *monotoneI*)

lemma *mcont-snd*: *mcont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *lubb ordb snd*
by(*auto intro*!: *mcontI monotoneI contI simp add*: *prod-lub-def*)

lemma *mcont2mcont-snd* [*cont-intro*, *simp*]:
mcont lub ord (*prod-lub luba lubb*) (*rel-prod orda ordb*) *t*
⟹ *mcont lub ord lubb ordb* (λ*x*. *snd* (*t x*))
by(*auto intro*!: *mcontI monotoneI contI dest*: *mcont-monoD mcont-contD simp add*: *rel-prod-sel split-beta prod-lub-def image-image*)

context *partial-function-definitions* **begin**

Specialised versions of *mcont-call* for admissibility proofs for parallel fixpoint inductions

lemmas *mcont-call-fst* [*cont-intro*] = *mcont-call*[*THEN mcont2mcont*, *OF mcont-fst*]
lemmas *mcont-call-snd* [*cont-intro*] = *mcont-call*[*THEN mcont2mcont*, *OF mcont-snd*]
end

end

23 Countable Complete Lattices

theory *Countable-Complete-Lattices*
imports *Main Countable-Set*
begin

lemma *UNIV-nat-eq*: *UNIV* = *insert 0* (*range Suc*)
by (*metis UNIV-eq-I nat.nchotomy insertCI rangeI*)

class *countable-complete-lattice* = *lattice* + *Inf* + *Sup* + *bot* + *top* +
assumes *ccInf-lower*: *countable* $A \Longrightarrow x \in A \Longrightarrow Inf\ A \leq x$
assumes *ccInf-greatest*: *countable* $A \Longrightarrow (\bigwedge x.\ x \in A \Longrightarrow z \leq x) \Longrightarrow z \leq Inf\ A$
assumes *ccSup-upper*: *countable* $A \Longrightarrow x \in A \Longrightarrow x \leq Sup\ A$
assumes *ccSup-least*: *countable* $A \Longrightarrow (\bigwedge x.\ x \in A \Longrightarrow x \leq z) \Longrightarrow Sup\ A \leq z$
assumes *ccInf-empty* [*simp*]: *Inf* {} = *top*
assumes *ccSup-empty* [*simp*]: *Sup* {} = *bot*
begin

subclass *bounded-lattice*
proof
fix *a*
show $bot \leq a$ **by** (*auto intro*: *ccSup-least simp only*: *ccSup-empty* [*symmetric*])
show $a \leq top$ **by** (*auto intro*: *ccInf-greatest simp only*: *ccInf-empty* [*symmetric*])
qed

lemma *ccINF-lower*: *countable* $A \Longrightarrow i \in A \Longrightarrow (INF\ i\ :A.\ f\ i) \leq f\ i$
using *ccInf-lower* [*of f ‘ A*] **by** *simp*

lemma *ccINF-greatest*: *countable* $A \Longrightarrow (\bigwedge i.\ i \in A \Longrightarrow u \leq f\ i) \Longrightarrow u \leq (INF\ i\ :A.\ f\ i)$
using *ccInf-greatest* [*of f ‘ A*] **by** *auto*

lemma *ccSUP-upper*: *countable* $A \Longrightarrow i \in A \Longrightarrow f\ i \leq (SUP\ i\ :A.\ f\ i)$
using *ccSup-upper* [*of f ‘ A*] **by** *simp*

lemma *ccSUP-least*: *countable* $A \Longrightarrow (\bigwedge i.\ i \in A \Longrightarrow f\ i \leq u) \Longrightarrow (SUP\ i\ :A.\ f\ i) \leq u$
using *ccSup-least* [*of f ‘ A*] **by** *auto*

lemma *ccInf-lower2*: *countable* $A \Longrightarrow u \in A \Longrightarrow u \leq v \Longrightarrow Inf\ A \leq v$
using *ccInf-lower* [*of A u*] **by** *auto*

lemma *ccINF-lower2*: *countable* $A \Longrightarrow i \in A \Longrightarrow f\ i \leq u \Longrightarrow (INF\ i\ :A.\ f\ i) \leq u$
using *ccINF-lower* [*of A i f*] **by** *auto*

lemma *ccSup-upper2*: *countable* $A \Longrightarrow u \in A \Longrightarrow v \leq u \Longrightarrow v \leq Sup\ A$
using *ccSup-upper* [*of A u*] **by** *auto*

lemma *ccSUP-upper2*: *countable* $A \Longrightarrow i \in A \Longrightarrow u \leq f\ i \Longrightarrow u \leq (SUP\ i\ :A.\ f\ i)$
using *ccSUP-upper* [*of A i f*] **by** *auto*

lemma *le-ccInf-iff*: *countable* $A \Longrightarrow b \leq Inf\ A \longleftrightarrow (\forall a \in A.\ b \leq a)$
by (*auto intro*: *ccInf-greatest dest*: *ccInf-lower*)

lemma *le-ccINF-iff*: *countable* $A \Longrightarrow u \leq (INF\ i\ :A.\ f\ i) \longleftrightarrow (\forall i \in A.\ u \leq f\ i)$
using *le-ccInf-iff* [*of f ‘ A*] **by** *simp*

lemma *ccSup-le-iff*: *countable A* $\Longrightarrow$ *Sup A* $\leq$ *b* $\longleftrightarrow$ ($\forall$ *a*$\in$*A*. *a* $\leq$ *b*)
by (*auto intro*: *ccSup-least dest*: *ccSup-upper*)

lemma *ccSUP-le-iff*: *countable A* $\Longrightarrow$ (*SUP i* :*A*. *f i*) $\leq$ *u* $\longleftrightarrow$ ($\forall$ *i*$\in$*A*. *f i* $\leq$ *u*)
using *ccSup-le-iff* [*of f* ' *A*] **by** *simp*

lemma *ccInf-insert* [*simp*]: *countable A* $\Longrightarrow$ *Inf* (*insert a A*) = *inf a* (*Inf A*)
by (*force intro*: *le-infI le-infI1 le-infI2 antisym ccInf-greatest ccInf-lower*)

lemma *ccINF-insert* [*simp*]: *countable A* $\Longrightarrow$ (*INF x*:*insert a A*. *f x*) = *inf* (*f a*) (*INFIMUM A f*)
unfolding *image-insert* **by** *simp*

lemma *ccSup-insert* [*simp*]: *countable A* $\Longrightarrow$ *Sup* (*insert a A*) = *sup a* (*Sup A*)
by (*force intro*: *le-supI le-supI1 le-supI2 antisym ccSup-least ccSup-upper*)

lemma *ccSUP-insert* [*simp*]: *countable A* $\Longrightarrow$ (*SUP x*:*insert a A*. *f x*) = *sup* (*f a*) (*SUPREMUM A f*)
unfolding *image-insert* **by** *simp*

lemma *ccINF-empty* [*simp*]: (*INF x*:{}. *f x*) = *top*
unfolding *image-empty* **by** *simp*

lemma *ccSUP-empty* [*simp*]: (*SUP x*:{}. *f x*) = *bot*
unfolding *image-empty* **by** *simp*

lemma *ccInf-superset-mono*: *countable A* $\Longrightarrow$ *B* $\subseteq$ *A* $\Longrightarrow$ *Inf A* $\leq$ *Inf B*
by (*auto intro*: *ccInf-greatest ccInf-lower countable-subset*)

lemma *ccSup-subset-mono*: *countable B* $\Longrightarrow$ *A* $\subseteq$ *B* $\Longrightarrow$ *Sup A* $\leq$ *Sup B*
by (*auto intro*: *ccSup-least ccSup-upper countable-subset*)

lemma *ccInf-mono*:
assumes [*intro*]: *countable B countable A*
assumes $\bigwedge$*b*. *b* $\in$ *B* $\Longrightarrow$ $\exists$ *a*$\in$*A*. *a* $\leq$ *b*
shows *Inf A* $\leq$ *Inf B*
proof (*rule ccInf-greatest*)
fix *b* **assume** *b* $\in$ *B*
with *assms* **obtain** *a* **where** *a* $\in$ *A* **and** *a* $\leq$ *b* **by** *blast*
from ‹*a* $\in$ *A*› **have** *Inf A* $\leq$ *a* **by** (*rule ccInf-lower*[*rotated*]) *auto*
with ‹*a* $\leq$ *b*› **show** *Inf A* $\leq$ *b* **by** *auto*
qed *auto*

lemma *ccINF-mono*:
countable A $\Longrightarrow$ *countable B* $\Longrightarrow$ ($\bigwedge$*m*. *m* $\in$ *B* $\Longrightarrow$ $\exists$ *n*$\in$*A*. *f n* $\leq$ *g m*) $\Longrightarrow$ (*INF n*:*A*. *f n*) $\leq$ (*INF n*:*B*. *g n*)
using *ccInf-mono* [*of g* ' *B f* ' *A*] **by** *auto*

lemma *ccSup-mono*:
assumes [*intro*]: *countable B countable A*
assumes $\bigwedge a.\ a \in A \Longrightarrow \exists\, b{\in}B.\ a \leq b$
shows *Sup A* $\leq$ *Sup B*
proof (*rule ccSup-least*)
fix *a* **assume** $a \in A$
with *assms* **obtain** *b* **where** $b \in B$ **and** $a \leq b$ **by** *blast*
from ‹$b \in B$› **have** $b \leq$ *Sup B* **by** (*rule ccSup-upper*[*rotated*]) *auto*
with ‹$a \leq b$› **show** $a \leq$ *Sup B* **by** *auto*
qed *auto*

lemma *ccSUP-mono*:
countable A $\Longrightarrow$ *countable B* $\Longrightarrow$ ($\bigwedge n.\ n \in A \Longrightarrow \exists\, m{\in}B.\ f\ n \leq g\ m$) $\Longrightarrow$ (*SUP* *n*:*A*. *f n*) $\leq$ (*SUP n*:*B*. *g n*)
using *ccSup-mono* [*of g ' B f ' A*] **by** *auto*

lemma *ccINF-superset-mono*:
countable A $\Longrightarrow B \subseteq A \Longrightarrow$ ($\bigwedge x.\ x \in B \Longrightarrow f\ x \leq g\ x$) $\Longrightarrow$ (*INF x*:*A*. *f x*) $\leq$ (*INF x*:*B*. *g x*)
by (*blast intro*: *ccINF-mono countable-subset dest*: *subsetD*)

lemma *ccSUP-subset-mono*:
countable B $\Longrightarrow A \subseteq B \Longrightarrow$ ($\bigwedge x.\ x \in A \Longrightarrow f\ x \leq g\ x$) $\Longrightarrow$ (*SUP x*:*A*. *f x*) $\leq$ (*SUP x*:*B*. *g x*)
by (*blast intro*: *ccSUP-mono countable-subset dest*: *subsetD*)

lemma *less-eq-ccInf-inter*: *countable A* $\Longrightarrow$ *countable B* $\Longrightarrow$ *sup* (*Inf A*) (*Inf B*) $\leq$ *Inf* ($A \cap B$)
by (*auto intro*: *ccInf-greatest ccInf-lower*)

lemma *ccSup-inter-less-eq*: *countable A* $\Longrightarrow$ *countable B* $\Longrightarrow$ *Sup* ($A \cap B$) $\leq$ *inf* (*Sup A*) (*Sup B*)
by (*auto intro*: *ccSup-least ccSup-upper*)

lemma *ccInf-union-distrib*: *countable A* $\Longrightarrow$ *countable B* $\Longrightarrow$ *Inf* ($A \cup B$) = *inf* (*Inf A*) (*Inf B*)
by (*rule antisym*) (*auto intro*: *ccInf-greatest ccInf-lower le-infI1 le-infI2*)

lemma *ccINF-union*:
countable A $\Longrightarrow$ *countable B* $\Longrightarrow$ (*INF i*:$A \cup B$. *M i*) = *inf* (*INF i*:*A*. *M i*) (*INF i*:*B*. *M i*)
by (*auto intro*!: *antisym ccINF-mono intro*: *le-infI1 le-infI2 ccINF-greatest ccINF-lower*)

lemma *ccSup-union-distrib*: *countable A* $\Longrightarrow$ *countable B* $\Longrightarrow$ *Sup* ($A \cup B$) = *sup* (*Sup A*) (*Sup B*)
by (*rule antisym*) (*auto intro*: *ccSup-least ccSup-upper le-supI1 le-supI2*)

lemma *ccSUP-union*:

countable A ⟹ *countable B* ⟹ (*SUP i*:*A* ∪ *B*. *M i*) = *sup* (*SUP i*:*A*. *M i*) (*SUP i*:*B*. *M i*)
by (*auto intro*!: *antisym ccSUP-mono intro*: *le-supI1 le-supI2 ccSUP-least ccSUP-upper*)

lemma *ccINF-inf-distrib*: *countable A* ⟹ *inf* (*INF a*:*A*. *f a*) (*INF a*:*A*. *g a*) = (*INF a*:*A*. *inf* (*f a*) (*g a*))
by (*rule antisym*) (*rule ccINF-greatest*, *auto intro*: *le-infI1 le-infI2 ccINF-lower ccINF-mono*)

lemma *ccSUP-sup-distrib*: *countable A* ⟹ *sup* (*SUP a*:*A*. *f a*) (*SUP a*:*A*. *g a*) = (*SUP a*:*A*. *sup* (*f a*) (*g a*))
by (*rule antisym*[*rotated*]) (*rule ccSUP-least*, *auto intro*: *le-supI1 le-supI2 ccSUP-upper ccSUP-mono*)

lemma *ccINF-const* [*simp*]: *A* ≠ {} ⟹ (*INF i* :*A*. *f*) = *f*
unfolding *image-constant-conv* **by** *auto*

lemma *ccSUP-const* [*simp*]: *A* ≠ {} ⟹ (*SUP i* :*A*. *f*) = *f*
unfolding *image-constant-conv* **by** *auto*

lemma *ccINF-top* [*simp*]: (*INF x*:*A*. *top*) = *top*
by (*cases A* = {}) *simp-all*

lemma *ccSUP-bot* [*simp*]: (*SUP x*:*A*. *bot*) = *bot*
by (*cases A* = {}) *simp-all*

lemma *ccINF-commute*: *countable A* ⟹ *countable B* ⟹ (*INF i*:*A*. *INF j*:*B*. *f i j*) = (*INF j*:*B*. *INF i*:*A*. *f i j*)
by (*iprover intro*: *ccINF-lower ccINF-greatest order-trans antisym*)

lemma *ccSUP-commute*: *countable A* ⟹ *countable B* ⟹ (*SUP i*:*A*. *SUP j*:*B*. *f i j*) = (*SUP j*:*B*. *SUP i*:*A*. *f i j*)
by (*iprover intro*: *ccSUP-upper ccSUP-least order-trans antisym*)

end

context
fixes *a* :: ′*a*::{*countable-complete-lattice*, *linorder*}
begin

lemma *less-ccSup-iff*: *countable S* ⟹ *a* < *Sup S* ⟷ (∃ *x*∈*S*. *a* < *x*)
unfolding *not-le* [*symmetric*] **by** (*subst ccSup-le-iff*) *auto*

lemma *less-ccSUP-iff*: *countable A* ⟹ *a* < (*SUP i*:*A*. *f i*) ⟷ (∃ *x*∈*A*. *a* < *f x*)
using *less-ccSup-iff* [*of f* ' *A*] **by** *simp*

lemma *ccInf-less-iff*: *countable S* ⟹ *Inf S* < *a* ⟷ (∃ *x*∈*S*. *x* < *a*)
unfolding *not-le* [*symmetric*] **by** (*subst le-ccInf-iff*) *auto*

lemma *ccINF-less-iff*: *countable A* ⟹ (*INF i*:*A*. *f i*) < *a* ⟷ (∃ *x*∈*A*. *f x* < *a*)
  using *ccInf-less-iff* [*of f* ' *A*] **by** *simp*

end

class *countable-complete-distrib-lattice* = *countable-complete-lattice* +
  assumes *sup-ccInf*: *countable B* ⟹ *sup a* (*Inf B*) = (*INF b*:*B*. *sup a b*)
  assumes *inf-ccSup*: *countable B* ⟹ *inf a* (*Sup B*) = (*SUP b*:*B*. *inf a b*)
begin

lemma *sup-ccINF*:
  countable B ⟹ *sup a* (*INF b*:*B*. *f b*) = (*INF b*:*B*. *sup a* (*f b*))
  by (*simp only*: *sup-ccInf image-image countable-image*)

lemma *inf-ccSUP*:
  countable B ⟹ *inf a* (*SUP b*:*B*. *f b*) = (*SUP b*:*B*. *inf a* (*f b*))
  by (*simp only*: *inf-ccSup image-image countable-image*)

subclass *distrib-lattice*
proof
  fix *a b c*
  from *sup-ccInf*[*of* {*b*, *c*} *a*] **have** *sup a* (*Inf* {*b*, *c*}) = (*INF d*:{*b*, *c*}. *sup a d*)
    by *simp*
  then show *sup a* (*inf b c*) = *inf* (*sup a b*) (*sup a c*)
    by *simp*
qed

lemma *ccInf-sup*:
  countable B ⟹ *sup* (*Inf B*) *a* = (*INF b*:*B*. *sup b a*)
  by (*simp add*: *sup-ccInf sup-commute*)

lemma *ccSup-inf*:
  countable B ⟹ *inf* (*Sup B*) *a* = (*SUP b*:*B*. *inf b a*)
  by (*simp add*: *inf-ccSup inf-commute*)

lemma *ccINF-sup*:
  countable B ⟹ *sup* (*INF b*:*B*. *f b*) *a* = (*INF b*:*B*. *sup* (*f b*) *a*)
  by (*simp add*: *sup-ccINF sup-commute*)

lemma *ccSUP-inf*:
  countable B ⟹ *inf* (*SUP b*:*B*. *f b*) *a* = (*SUP b*:*B*. *inf* (*f b*) *a*)
  by (*simp add*: *inf-ccSUP inf-commute*)

lemma *ccINF-sup-distrib2*:
  countable A ⟹ *countable B* ⟹ *sup* (*INF a*:*A*. *f a*) (*INF b*:*B*. *g b*) = (*INF a*:*A*. *INF b*:*B*. *sup* (*f a*) (*g b*))
  by (*subst ccINF-commute*) (*simp-all add*: *sup-ccINF ccINF-sup*)

lemma *ccSUP-inf-distrib2*:

countable A $\Longrightarrow$ *countable B* $\Longrightarrow$ *inf* (*SUP a*:*A*. *f a*) (*SUP b*:*B*. *g b*) = (*SUP a*:*A*. *SUP b*:*B*. *inf* (*f a*) (*g b*))
by (*subst ccSUP-commute*) (*simp-all add*: *inf-ccSUP ccSUP-inf*)

context
fixes *f* :: *'a* ⇒ *'b*::*countable-complete-lattice*
assumes *mono f*
begin

lemma *mono-ccInf*:
countable A $\Longrightarrow$ *f* (*Inf A*) ≤ (*INF x*:*A*. *f x*)
using ‹*mono f*›
by (*auto intro*!: *countable-complete-lattice-class.ccINF-greatest intro*: *ccInf-lower dest*: *monoD*)

lemma *mono-ccSup*:
countable A $\Longrightarrow$ (*SUP x*:*A*. *f x*) ≤ *f* (*Sup A*)
using ‹*mono f*› **by** (*auto intro*: *countable-complete-lattice-class.ccSUP-least ccSup-upper dest*: *monoD*)

lemma *mono-ccINF*:
countable I $\Longrightarrow$ *f* (*INF i* : *I*. *A i*) ≤ (*INF x* : *I*. *f* (*A x*))
by (*intro countable-complete-lattice-class.ccINF-greatest monoD*[*OF* ‹*mono f*›] *ccINF-lower*)

lemma *mono-ccSUP*:
countable I $\Longrightarrow$ (*SUP x* : *I*. *f* (*A x*)) ≤ *f* (*SUP i* : *I*. *A i*)
by (*intro countable-complete-lattice-class.ccSUP-least monoD*[*OF* ‹*mono f*›] *ccSUP-upper*)

end

end

23.0.1 Instances of countable complete lattices

instance *fun* :: (*type*, *countable-complete-lattice*) *countable-complete-lattice*
by *standard*
(*auto simp*: *le-fun-def intro*!: *ccSUP-upper ccSUP-least ccINF-lower ccINF-greatest*)

subclass (**in** *complete-lattice*) *countable-complete-lattice*
by *standard* (*auto intro*: *Sup-upper Sup-least Inf-lower Inf-greatest*)

subclass (**in** *complete-distrib-lattice*) *countable-complete-distrib-lattice*
by *standard* (*auto intro*: *sup-Inf inf-Sup*)

end

24 Cardinal Notations

theory *Cardinal-Notations*
imports *Main*
begin

notation
ordLeq2 (**infix** *<=o 50*) **and**
ordLeq3 (**infix** $\leq$*o 50*) **and**
ordLess2 (**infix** *<o 50*) **and**
ordIso2 (**infix** *=o 50*) **and**
card-of (|-|) **and**
BNF-Cardinal-Arithmetic.csum (**infixr** *+c 65*) **and**
BNF-Cardinal-Arithmetic.cprod (**infixr** **c 80*) **and**
BNF-Cardinal-Arithmetic.cexp (**infixr** *^c 90*)

abbreviation *cinfinite* $\equiv$ *BNF-Cardinal-Arithmetic.cinfinite*
abbreviation *czero* $\equiv$ *BNF-Cardinal-Arithmetic.czero*
abbreviation *cone* $\equiv$ *BNF-Cardinal-Arithmetic.cone*
abbreviation *ctwo* $\equiv$ *BNF-Cardinal-Arithmetic.ctwo*

end

25 Type of (at Most) Countable Sets

theory *Countable-Set-Type*
imports *Countable-Set Cardinal-Notations Conditionally-Complete-Lattices*
begin

25.1 Cardinal stuff

lemma *countable-card-of-nat*: *countable A* $\longleftrightarrow$ $|A| \leq$*o* $|$*UNIV*::*nat set*$|$
unfolding *countable-def card-of-ordLeq*[*symmetric*] **by** *auto*

lemma *countable-card-le-natLeq*: *countable A* $\longleftrightarrow$ $|A| \leq$*o natLeq*
unfolding *countable-card-of-nat* **using** *card-of-nat ordLeq-ordIso-trans ordIso-symmetric*
by *blast*

lemma *countable-or-card-of*:
assumes *countable A*
shows (*finite A* $\wedge$ $|A|$ *<o* $|$*UNIV*::*nat set*$|$) $\vee$
(*infinite A* $\wedge$ $|A|$ *=o* $|$*UNIV*::*nat set*$|$)
by (*metis assms countable-card-of-nat infinite-iff-card-of-nat ordIso-iff-ordLeq ordLeq-iff-ordLess-or-ordIso*)

lemma *countable-cases-card-of*[*elim*]:
assumes *countable A*
obtains (*Fin*) *finite A* $|A|$ *<o* $|$*UNIV*::*nat set*$|$
| (*Inf*) *infinite A* $|A|$ *=o* $|$*UNIV*::*nat set*$|$

using *assms countable-or-card-of* **by** *blast*

lemma *countable-or*:
countable A $\Longrightarrow$ ($\exists$ *f*::'*a*$\Rightarrow$*nat*. *finite A* $\wedge$ *inj-on f A*) $\vee$ ($\exists$ *f*::'*a*$\Rightarrow$*nat*. *infinite A* $\wedge$ *bij-betw f A UNIV*)
by (*elim countable-enum-cases*) *fastforce*+

lemma *countable-cases*[*elim*]:
assumes *countable A*
obtains (*Fin*) *f* :: '*a*$\Rightarrow$*nat* **where** *finite A inj-on f A*
| (*Inf*) *f* :: '*a*$\Rightarrow$*nat* **where** *infinite A bij-betw f A UNIV*
using *assms countable-or* **by** *metis*

lemma *countable-ordLeq*:
assumes $|A| \le o\ |B|$ **and** *countable B*
shows *countable A*
using *assms* **unfolding** *countable-card-of-nat* **by**(*rule ordLeq-transitive*)

lemma *countable-ordLess*:
assumes *AB*: $|A| < o\ |B|$ **and** *B*: *countable B*
shows *countable A*
using *countable-ordLeq*[*OF ordLess-imp-ordLeq*[*OF AB*] *B*] **.**

25.2 The type of countable sets

typedef '*a cset* = {*A* :: '*a set*. *countable A*} **morphisms** *rcset acset*
by (*rule exI*[*of* - {}]) *simp*

setup-lifting *type-definition-cset*

declare
rcset-inverse[*simp*]
acset-inverse[*Transfer.transferred*, *unfolded mem-Collect-eq*, *simp*]
acset-inject[*Transfer.transferred*, *unfolded mem-Collect-eq*, *simp*]
rcset[*Transfer.transferred*, *unfolded mem-Collect-eq*, *simp*]

instantiation *cset* :: (*type*) {*bounded-lattice-bot*, *distrib-lattice*, *minus*}
begin

interpretation *lifting-syntax* **.**

lift-definition *bot-cset* :: '*a cset* **is** {} **parametric** *empty-transfer* **by** *simp*

lift-definition *less-eq-cset* :: '*a cset* $\Rightarrow$ '*a cset* $\Rightarrow$ *bool*
is *subset-eq* **parametric** *subset-transfer* **.**

definition *less-cset* :: '*a cset* $\Rightarrow$ '*a cset* $\Rightarrow$ *bool*
where $xs < ys \equiv xs \le ys \wedge xs \ne (ys::'a\ cset)$

lemma *less-cset-transfer*[*transfer-rule*]:
  assumes [*transfer-rule*]: *bi-unique A*
  shows ((*pcr-cset A*) ===> (*pcr-cset A*) ===> *op* =) *op* ⊂ *op* <
unfolding *less-cset-def*[*abs-def*] *psubset-eq*[*abs-def*] **by** *transfer-prover*

lift-definition *sup-cset* :: 'a cset ⇒ 'a cset ⇒ 'a cset
is *union* **parametric** *union-transfer* **by** *simp*

lift-definition *inf-cset* :: 'a cset ⇒ 'a cset ⇒ 'a cset
is *inter* **parametric** *inter-transfer* **by** *simp*

lift-definition *minus-cset* :: 'a cset ⇒ 'a cset ⇒ 'a cset
is *minus* **parametric** *Diff-transfer* **by** *simp*

instance by *standard* (*transfer*; *auto*)+

end

abbreviation *cempty* :: 'a cset **where** *cempty* ≡ *bot*
abbreviation *csubset-eq* :: 'a cset ⇒ 'a cset ⇒ bool **where** *csubset-eq xs ys* ≡ *xs* ≤ *ys*
abbreviation *csubset* :: 'a cset ⇒ 'a cset ⇒ bool **where** *csubset xs ys* ≡ *xs* < *ys*
abbreviation *cUn* :: 'a cset ⇒ 'a cset ⇒ 'a cset **where** *cUn xs ys* ≡ *sup xs ys*
abbreviation *cInt* :: 'a cset ⇒ 'a cset ⇒ 'a cset **where** *cInt xs ys* ≡ *inf xs ys*
abbreviation *cDiff* :: 'a cset ⇒ 'a cset ⇒ 'a cset **where** *cDiff xs ys* ≡ *minus xs ys*

lift-definition *cin* :: 'a ⇒ 'a cset ⇒ bool **is** *op* ∈ **parametric** *member-transfer*
  .
lift-definition *cinsert* :: 'a ⇒ 'a cset ⇒ 'a cset **is** *insert* **parametric** *Lifting-Set.insert-transfer*
  by (*rule countable-insert*)
abbreviation *csingle* :: 'a ⇒ 'a cset **where** *csingle x* ≡ *cinsert x cempty*
lift-definition *cimage* :: ('a ⇒ 'b) ⇒ 'a cset ⇒ 'b cset **is** *op* ' **parametric** *image-transfer*
  by (*rule countable-image*)
lift-definition *cBall* :: 'a cset ⇒ ('a ⇒ bool) ⇒ bool **is** *Ball* **parametric** *Ball-transfer*
.
lift-definition *cBex* :: 'a cset ⇒ ('a ⇒ bool) ⇒ bool **is** *Bex* **parametric** *Bex-transfer*
.
lift-definition *cUNION* :: 'a cset ⇒ ('a ⇒ 'b cset) ⇒ 'b cset
  is *UNION* **parametric** *UNION-transfer* **by** *simp*
definition *cUnion* :: 'a cset cset ⇒ 'a cset **where** *cUnion A* = *cUNION A id*

lemma *Union-conv-UNION*: ⋃ *A* = *UNION A id*
by *auto*

lemma *cUnion-transfer* [*transfer-rule*]:
  rel-fun (*pcr-cset* (*pcr-cset A*)) (*pcr-cset A*) *Union cUnion*
proof −

have *rel-fun* (*pcr-cset* (*pcr-cset* *A*)) (*pcr-cset* *A*) (*λA. UNION A id*) (*λA. cUNION A id*)
by *transfer-prover*
then show *?thesis* **by** (*simp add*: *cUnion-def* [*symmetric*])
qed

lemmas *cset-eqI* = *set-eqI*[*Transfer.transferred*]
lemmas *cset-eq-iff*[*no-atp*] = *set-eq-iff*[*Transfer.transferred*]
lemmas *cBallI*[*intro*!] = *ballI*[*Transfer.transferred*]
lemmas *cbspec*[*dest?*] = *bspec*[*Transfer.transferred*]
lemmas *cBallE*[*elim*] = *ballE*[*Transfer.transferred*]
lemmas *cBexI*[*intro*] = *bexI*[*Transfer.transferred*]
lemmas *rev-cBexI*[*intro?*] = *rev-bexI*[*Transfer.transferred*]
lemmas *cBexCI* = *bexCI*[*Transfer.transferred*]
lemmas *cBexE*[*elim*!] = *bexE*[*Transfer.transferred*]
lemmas *cBall-triv*[*simp*] = *ball-triv*[*Transfer.transferred*]
lemmas *cBex-triv*[*simp*] = *bex-triv*[*Transfer.transferred*]
lemmas *cBex-triv-one-point1*[*simp*] = *bex-triv-one-point1*[*Transfer.transferred*]
lemmas *cBex-triv-one-point2*[*simp*] = *bex-triv-one-point2*[*Transfer.transferred*]
lemmas *cBex-one-point1*[*simp*] = *bex-one-point1*[*Transfer.transferred*]
lemmas *cBex-one-point2*[*simp*] = *bex-one-point2*[*Transfer.transferred*]
lemmas *cBall-one-point1*[*simp*] = *ball-one-point1*[*Transfer.transferred*]
lemmas *cBall-one-point2*[*simp*] = *ball-one-point2*[*Transfer.transferred*]
lemmas *cBall-conj-distrib* = *ball-conj-distrib*[*Transfer.transferred*]
lemmas *cBex-disj-distrib* = *bex-disj-distrib*[*Transfer.transferred*]
lemmas *cBall-cong* = *ball-cong*[*Transfer.transferred*]
lemmas *cBex-cong* = *bex-cong*[*Transfer.transferred*]
lemmas *csubsetI*[*intro*!] = *subsetI*[*Transfer.transferred*]
lemmas *csubsetD*[*elim*, *intro?*] = *subsetD*[*Transfer.transferred*]
lemmas *rev-csubsetD*[*no-atp*,*intro?*] = *rev-subsetD*[*Transfer.transferred*]
lemmas *csubsetCE*[*no-atp*,*elim*] = *subsetCE*[*Transfer.transferred*]
lemmas *csubset-eq*[*no-atp*] = *subset-eq*[*Transfer.transferred*]
lemmas *contra-csubsetD*[*no-atp*] = *contra-subsetD*[*Transfer.transferred*]
lemmas *csubset-refl* = *subset-refl*[*Transfer.transferred*]
lemmas *csubset-trans* = *subset-trans*[*Transfer.transferred*]
lemmas *cset-rev-mp* = *set-rev-mp*[*Transfer.transferred*]
lemmas *cset-mp* = *set-mp*[*Transfer.transferred*]
lemmas *csubset-not-fsubset-eq*[*code*] = *subset-not-subset-eq*[*Transfer.transferred*]
lemmas *eq-cmem-trans* = *eq-mem-trans*[*Transfer.transferred*]
lemmas *csubset-antisym*[*intro*!] = *subset-antisym*[*Transfer.transferred*]
lemmas *cequalityD1* = *equalityD1*[*Transfer.transferred*]
lemmas *cequalityD2* = *equalityD2*[*Transfer.transferred*]
lemmas *cequalityE* = *equalityE*[*Transfer.transferred*]
lemmas *cequalityCE*[*elim*] = *equalityCE*[*Transfer.transferred*]
lemmas *eqcset-imp-iff* = *eqset-imp-iff*[*Transfer.transferred*]
lemmas *eqcelem-imp-iff* = *eqelem-imp-iff*[*Transfer.transferred*]
lemmas *cempty-iff*[*simp*] = *empty-iff*[*Transfer.transferred*]
lemmas *cempty-fsubsetI*[*iff*] = *empty-subsetI*[*Transfer.transferred*]
lemmas *equals-cemptyI* = *equals0I*[*Transfer.transferred*]

lemmas *equals-cemptyD* = *equals0D*[*Transfer.transferred*]
lemmas *cBall-cempty*[*simp*] = *ball-empty*[*Transfer.transferred*]
lemmas *cBex-cempty*[*simp*] = *bex-empty*[*Transfer.transferred*]
lemmas *cInt-iff*[*simp*] = *Int-iff*[*Transfer.transferred*]
lemmas *cIntI*[*intro*!] = *IntI*[*Transfer.transferred*]
lemmas *cIntD1* = *IntD1*[*Transfer.transferred*]
lemmas *cIntD2* = *IntD2*[*Transfer.transferred*]
lemmas *cIntE*[*elim*!] = *IntE*[*Transfer.transferred*]
lemmas *cUn-iff*[*simp*] = *Un-iff*[*Transfer.transferred*]
lemmas *cUnI1*[*elim?*] = *UnI1*[*Transfer.transferred*]
lemmas *cUnI2*[*elim?*] = *UnI2*[*Transfer.transferred*]
lemmas *cUnCI*[*intro*!] = *UnCI*[*Transfer.transferred*]
lemmas *cuUnE*[*elim*!] = *UnE*[*Transfer.transferred*]
lemmas *cDiff-iff*[*simp*] = *Diff-iff*[*Transfer.transferred*]
lemmas *cDiffI*[*intro*!] = *DiffI*[*Transfer.transferred*]
lemmas *cDiffD1* = *DiffD1*[*Transfer.transferred*]
lemmas *cDiffD2* = *DiffD2*[*Transfer.transferred*]
lemmas *cDiffE*[*elim*!] = *DiffE*[*Transfer.transferred*]
lemmas *cinsert-iff*[*simp*] = *insert-iff*[*Transfer.transferred*]
lemmas *cinsertI1* = *insertI1*[*Transfer.transferred*]
lemmas *cinsertI2* = *insertI2*[*Transfer.transferred*]
lemmas *cinsertE*[*elim*!] = *insertE*[*Transfer.transferred*]
lemmas *cinsertCI*[*intro*!] = *insertCI*[*Transfer.transferred*]
lemmas *csubset-cinsert-iff* = *subset-insert-iff*[*Transfer.transferred*]
lemmas *cinsert-ident* = *insert-ident*[*Transfer.transferred*]
lemmas *csingletonI*[*intro*!,*no-atp*] = *singletonI*[*Transfer.transferred*]
lemmas *csingletonD*[*dest*!,*no-atp*] = *singletonD*[*Transfer.transferred*]
lemmas *fsingletonE* = *csingletonD* [*elim-format*]
lemmas *csingleton-iff* = *singleton-iff*[*Transfer.transferred*]
lemmas *csingleton-inject*[*dest*!] = *singleton-inject*[*Transfer.transferred*]
lemmas *csingleton-finsert-inj-eq*[*iff*,*no-atp*] = *singleton-insert-inj-eq*[*Transfer.transferred*]
lemmas *csingleton-finsert-inj-eq'*[*iff*,*no-atp*] = *singleton-insert-inj-eq'*[*Transfer.transferred*]
lemmas *csubset-csingletonD* = *subset-singletonD*[*Transfer.transferred*]
lemmas *cDiff-single-cinsert* = *Diff-single-insert*[*Transfer.transferred*]
lemmas *cdoubleton-eq-iff* = *doubleton-eq-iff*[*Transfer.transferred*]
lemmas *cUn-csingleton-iff* = *Un-singleton-iff*[*Transfer.transferred*]
lemmas *csingleton-cUn-iff* = *singleton-Un-iff*[*Transfer.transferred*]
lemmas *cimage-eqI*[*simp*, *intro*] = *image-eqI*[*Transfer.transferred*]
lemmas *cimageI* = *imageI*[*Transfer.transferred*]
lemmas *rev-cimage-eqI* = *rev-image-eqI*[*Transfer.transferred*]
lemmas *cimageE*[*elim*!] = *imageE*[*Transfer.transferred*]
lemmas *Compr-cimage-eq* = *Compr-image-eq*[*Transfer.transferred*]
lemmas *cimage-cUn* = *image-Un*[*Transfer.transferred*]
lemmas *cimage-iff* = *image-iff*[*Transfer.transferred*]
lemmas *cimage-csubset-iff*[*no-atp*] = *image-subset-iff*[*Transfer.transferred*]
lemmas *cimage-csubsetI* = *image-subsetI*[*Transfer.transferred*]
lemmas *cimage-ident*[*simp*] = *image-ident*[*Transfer.transferred*]
lemmas *if-split-cin1* = *if-split-mem1*[*Transfer.transferred*]
lemmas *if-split-cin2* = *if-split-mem2*[*Transfer.transferred*]

lemmas *cpsubsetI*[*intro*!,*no-atp*] = *psubsetI*[*Transfer.transferred*]
lemmas *cpsubsetE*[*elim*!,*no-atp*] = *psubsetE*[*Transfer.transferred*]
lemmas *cpsubset-finsert-iff* = *psubset-insert-iff*[*Transfer.transferred*]
lemmas *cpsubset-eq* = *psubset-eq*[*Transfer.transferred*]
lemmas *cpsubset-imp-fsubset* = *psubset-imp-subset*[*Transfer.transferred*]
lemmas *cpsubset-trans* = *psubset-trans*[*Transfer.transferred*]
lemmas *cpsubsetD* = *psubsetD*[*Transfer.transferred*]
lemmas *cpsubset-csubset-trans* = *psubset-subset-trans*[*Transfer.transferred*]
lemmas *csubset-cpsubset-trans* = *subset-psubset-trans*[*Transfer.transferred*]
lemmas *cpsubset-imp-ex-fmem* = *psubset-imp-ex-mem*[*Transfer.transferred*]
lemmas *csubset-cinsertI* = *subset-insertI*[*Transfer.transferred*]
lemmas *csubset-cinsertI2* = *subset-insertI2*[*Transfer.transferred*]
lemmas *csubset-cinsert* = *subset-insert*[*Transfer.transferred*]
lemmas *cUn-upper1* = *Un-upper1*[*Transfer.transferred*]
lemmas *cUn-upper2* = *Un-upper2*[*Transfer.transferred*]
lemmas *cUn-least* = *Un-least*[*Transfer.transferred*]
lemmas *cInt-lower1* = *Int-lower1*[*Transfer.transferred*]
lemmas *cInt-lower2* = *Int-lower2*[*Transfer.transferred*]
lemmas *cInt-greatest* = *Int-greatest*[*Transfer.transferred*]
lemmas *cDiff-csubset* = *Diff-subset*[*Transfer.transferred*]
lemmas *cDiff-csubset-conv* = *Diff-subset-conv*[*Transfer.transferred*]
lemmas *csubset-cempty*[*simp*] = *subset-empty*[*Transfer.transferred*]
lemmas *not-cpsubset-cempty*[*iff*] = *not-psubset-empty*[*Transfer.transferred*]
lemmas *cinsert-is-cUn* = *insert-is-Un*[*Transfer.transferred*]
lemmas *cinsert-not-cempty*[*simp*] = *insert-not-empty*[*Transfer.transferred*]
lemmas *cempty-not-cinsert* = *empty-not-insert*[*Transfer.transferred*]
lemmas *cinsert-absorb* = *insert-absorb*[*Transfer.transferred*]
lemmas *cinsert-absorb2*[*simp*] = *insert-absorb2*[*Transfer.transferred*]
lemmas *cinsert-commute* = *insert-commute*[*Transfer.transferred*]
lemmas *cinsert-csubset*[*simp*] = *insert-subset*[*Transfer.transferred*]
lemmas *cinsert-cinter-cinsert*[*simp*] = *insert-inter-insert*[*Transfer.transferred*]
lemmas *cinsert-disjoint*[*simp*,*no-atp*] = *insert-disjoint*[*Transfer.transferred*]
lemmas *disjoint-cinsert*[*simp*,*no-atp*] = *disjoint-insert*[*Transfer.transferred*]
lemmas *cimage-cempty*[*simp*] = *image-empty*[*Transfer.transferred*]
lemmas *cimage-cinsert*[*simp*] = *image-insert*[*Transfer.transferred*]
lemmas *cimage-constant* = *image-constant*[*Transfer.transferred*]
lemmas *cimage-constant-conv* = *image-constant-conv*[*Transfer.transferred*]
lemmas *cimage-cimage* = *image-image*[*Transfer.transferred*]
lemmas *cinsert-cimage*[*simp*] = *insert-image*[*Transfer.transferred*]
lemmas *cimage-is-cempty*[*iff*] = *image-is-empty*[*Transfer.transferred*]
lemmas *cempty-is-cimage*[*iff*] = *empty-is-image*[*Transfer.transferred*]
lemmas *cimage-cong* = *image-cong*[*Transfer.transferred*]
lemmas *cimage-cInt-csubset* = *image-Int-subset*[*Transfer.transferred*]
lemmas *cimage-cDiff-csubset* = *image-diff-subset*[*Transfer.transferred*]
lemmas *cInt-absorb* = *Int-absorb*[*Transfer.transferred*]
lemmas *cInt-left-absorb* = *Int-left-absorb*[*Transfer.transferred*]
lemmas *cInt-commute* = *Int-commute*[*Transfer.transferred*]
lemmas *cInt-left-commute* = *Int-left-commute*[*Transfer.transferred*]
lemmas *cInt-assoc* = *Int-assoc*[*Transfer.transferred*]

lemmas *cInt-ac = Int-ac[Transfer.transferred]*
lemmas *cInt-absorb1 = Int-absorb1[Transfer.transferred]*
lemmas *cInt-absorb2 = Int-absorb2[Transfer.transferred]*
lemmas *cInt-cempty-left = Int-empty-left[Transfer.transferred]*
lemmas *cInt-cempty-right = Int-empty-right[Transfer.transferred]*
lemmas *disjoint-iff-cnot-equal = disjoint-iff-not-equal[Transfer.transferred]*
lemmas *cInt-cUn-distrib = Int-Un-distrib[Transfer.transferred]*
lemmas *cInt-cUn-distrib2 = Int-Un-distrib2[Transfer.transferred]*
lemmas *cInt-csubset-iff[no-atp, simp] = Int-subset-iff[Transfer.transferred]*
lemmas *cUn-absorb = Un-absorb[Transfer.transferred]*
lemmas *cUn-left-absorb = Un-left-absorb[Transfer.transferred]*
lemmas *cUn-commute = Un-commute[Transfer.transferred]*
lemmas *cUn-left-commute = Un-left-commute[Transfer.transferred]*
lemmas *cUn-assoc = Un-assoc[Transfer.transferred]*
lemmas *cUn-ac = Un-ac[Transfer.transferred]*
lemmas *cUn-absorb1 = Un-absorb1[Transfer.transferred]*
lemmas *cUn-absorb2 = Un-absorb2[Transfer.transferred]*
lemmas *cUn-cempty-left = Un-empty-left[Transfer.transferred]*
lemmas *cUn-cempty-right = Un-empty-right[Transfer.transferred]*
lemmas *cUn-cinsert-left[simp] = Un-insert-left[Transfer.transferred]*
lemmas *cUn-cinsert-right[simp] = Un-insert-right[Transfer.transferred]*
lemmas *cInt-cinsert-left = Int-insert-left[Transfer.transferred]*
lemmas *cInt-cinsert-left-if0[simp] = Int-insert-left-if0[Transfer.transferred]*
lemmas *cInt-cinsert-left-if1[simp] = Int-insert-left-if1[Transfer.transferred]*
lemmas *cInt-cinsert-right = Int-insert-right[Transfer.transferred]*
lemmas *cInt-cinsert-right-if0[simp] = Int-insert-right-if0[Transfer.transferred]*
lemmas *cInt-cinsert-right-if1[simp] = Int-insert-right-if1[Transfer.transferred]*
lemmas *cUn-cInt-distrib = Un-Int-distrib[Transfer.transferred]*
lemmas *cUn-cInt-distrib2 = Un-Int-distrib2[Transfer.transferred]*
lemmas *cUn-cInt-crazy = Un-Int-crazy[Transfer.transferred]*
lemmas *csubset-cUn-eq = subset-Un-eq[Transfer.transferred]*
lemmas *cUn-cempty[iff] = Un-empty[Transfer.transferred]*
lemmas *cUn-csubset-iff[no-atp, simp] = Un-subset-iff[Transfer.transferred]*
lemmas *cUn-cDiff-cInt = Un-Diff-Int[Transfer.transferred]*
lemmas *cDiff-cInt2 = Diff-Int2[Transfer.transferred]*
lemmas *cUn-cInt-assoc-eq = Un-Int-assoc-eq[Transfer.transferred]*
lemmas *cBall-cUn = ball-Un[Transfer.transferred]*
lemmas *cBex-cUn = bex-Un[Transfer.transferred]*
lemmas *cDiff-eq-cempty-iff[simp,no-atp] = Diff-eq-empty-iff[Transfer.transferred]*
lemmas *cDiff-cancel[simp] = Diff-cancel[Transfer.transferred]*
lemmas *cDiff-idemp[simp] = Diff-idemp[Transfer.transferred]*
lemmas *cDiff-triv = Diff-triv[Transfer.transferred]*
lemmas *cempty-cDiff[simp] = empty-Diff[Transfer.transferred]*
lemmas *cDiff-cempty[simp] = Diff-empty[Transfer.transferred]*
lemmas *cDiff-cinsert0[simp,no-atp] = Diff-insert0[Transfer.transferred]*
lemmas *cDiff-cinsert = Diff-insert[Transfer.transferred]*
lemmas *cDiff-cinsert2 = Diff-insert2[Transfer.transferred]*
lemmas *cinsert-cDiff-if = insert-Diff-if[Transfer.transferred]*
lemmas *cinsert-cDiff1[simp] = insert-Diff1[Transfer.transferred]*

lemmas *cinsert-cDiff-single*[*simp*] = *insert-Diff-single*[*Transfer.transferred*]
lemmas *cinsert-cDiff* = *insert-Diff*[*Transfer.transferred*]
lemmas *cDiff-cinsert-absorb* = *Diff-insert-absorb*[*Transfer.transferred*]
lemmas *cDiff-disjoint*[*simp*] = *Diff-disjoint*[*Transfer.transferred*]
lemmas *cDiff-partition* = *Diff-partition*[*Transfer.transferred*]
lemmas *double-cDiff* = *double-diff*[*Transfer.transferred*]
lemmas *cUn-cDiff-cancel*[*simp*] = *Un-Diff-cancel*[*Transfer.transferred*]
lemmas *cUn-cDiff-cancel2*[*simp*] = *Un-Diff-cancel2*[*Transfer.transferred*]
lemmas *cDiff-cUn* = *Diff-Un*[*Transfer.transferred*]
lemmas *cDiff-cInt* = *Diff-Int*[*Transfer.transferred*]
lemmas *cUn-cDiff* = *Un-Diff*[*Transfer.transferred*]
lemmas *cInt-cDiff* = *Int-Diff*[*Transfer.transferred*]
lemmas *cDiff-cInt-distrib* = *Diff-Int-distrib*[*Transfer.transferred*]
lemmas *cDiff-cInt-distrib2* = *Diff-Int-distrib2*[*Transfer.transferred*]
lemmas *cset-eq-csubset* = *set-eq-subset*[*Transfer.transferred*]
lemmas *csubset-iff*[*no-atp*] = *subset-iff*[*Transfer.transferred*]
lemmas *csubset-iff-pfsubset-eq* = *subset-iff-psubset-eq*[*Transfer.transferred*]
lemmas *all-not-cin-conv*[*simp*] = *all-not-in-conv*[*Transfer.transferred*]
lemmas *ex-cin-conv* = *ex-in-conv*[*Transfer.transferred*]
lemmas *cimage-mono* = *image-mono*[*Transfer.transferred*]
lemmas *cinsert-mono* = *insert-mono*[*Transfer.transferred*]
lemmas *cunion-mono* = *Un-mono*[*Transfer.transferred*]
lemmas *cinter-mono* = *Int-mono*[*Transfer.transferred*]
lemmas *cminus-mono* = *Diff-mono*[*Transfer.transferred*]
lemmas *cin-mono* = *in-mono*[*Transfer.transferred*]
lemmas *cLeast-mono* = *Least-mono*[*Transfer.transferred*]
lemmas *cequalityI* = *equalityI*[*Transfer.transferred*]
lemmas *cUN-iff* [*simp*] = *UN-iff*[*Transfer.transferred*]
lemmas *cUN-I* [*intro*] = *UN-I*[*Transfer.transferred*]
lemmas *cUN-E* [*elim*!] = *UN-E*[*Transfer.transferred*]
lemmas *cUN-upper* = *UN-upper*[*Transfer.transferred*]
lemmas *cUN-least* = *UN-least*[*Transfer.transferred*]
lemmas *cUN-cinsert-distrib* = *UN-insert-distrib*[*Transfer.transferred*]
lemmas *cUN-empty* [*simp*] = *UN-empty*[*Transfer.transferred*]
lemmas *cUN-empty2* [*simp*] = *UN-empty2*[*Transfer.transferred*]
lemmas *cUN-absorb* = *UN-absorb*[*Transfer.transferred*]
lemmas *cUN-cinsert* [*simp*] = *UN-insert*[*Transfer.transferred*]
lemmas *cUN-cUn* [*simp*] = *UN-Un*[*Transfer.transferred*]
lemmas *cUN-cUN-flatten* = *UN-UN-flatten*[*Transfer.transferred*]
lemmas *cUN-csubset-iff* = *UN-subset-iff*[*Transfer.transferred*]
lemmas *cUN-constant* [*simp*] = *UN-constant*[*Transfer.transferred*]
lemmas *cimage-cUnion* = *image-Union*[*Transfer.transferred*]
lemmas *cUNION-cempty-conv* [*simp*] = *UNION-empty-conv*[*Transfer.transferred*]
lemmas *cBall-cUN* = *ball-UN*[*Transfer.transferred*]
lemmas *cBex-cUN* = *bex-UN*[*Transfer.transferred*]
lemmas *cUn-eq-cUN* = *Un-eq-UN*[*Transfer.transferred*]
lemmas *cUN-mono* = *UN-mono*[*Transfer.transferred*]
lemmas *cimage-cUN* = *image-UN*[*Transfer.transferred*]
lemmas *cUN-csingleton* [*simp*] = *UN-singleton*[*Transfer.transferred*]

25.3 Additional lemmas

25.3.1 *cempty*

lemma *cemptyE* [*elim*!]: *cin a cempty* $\Longrightarrow$ *P* **by** *simp*

25.3.2 *cinsert*

lemma *countable-insert-iff*: *countable* (*insert x A*) $\longleftrightarrow$ *countable A*
by (*metis Diff-eq-empty-iff countable-empty countable-insert subset-insertI uncountable-minus-countable*)

lemma *set-cinsert*:
assumes *cin x A*
obtains *B* **where** *A* = *cinsert x B* **and** ¬ *cin x B*
using *assms* **by** *transfer*(*erule Set.set-insert*, *simp add*: *countable-insert-iff*)

lemma *mk-disjoint-cinsert*: *cin a A* $\Longrightarrow$ ∃ *B*. *A* = *cinsert a B* ∧ ¬ *cin a B*
by (*rule exI*[**where** *x* = *cDiff A* (*csingle a*)]) *blast*

25.3.3 *cimage*

lemma *subset-cimage-iff*: *csubset-eq B* (*cimage f A*) $\longleftrightarrow$ (∃ *AA*. *csubset-eq AA A* ∧ *B* = *cimage f AA*)
by *transfer* (*metis countable-subset image-mono mem-Collect-eq subset-imageE*)

25.3.4 bounded quantification

lemma *cBex-simps* [*simp*, *no-atp*]:
$\bigwedge$*A P Q*. *cBex A* (λ*x*. *P x* ∧ *Q*) = (*cBex A P* ∧ *Q*)
$\bigwedge$*A P Q*. *cBex A* (λ*x*. *P* ∧ *Q x*) = (*P* ∧ *cBex A Q*)
$\bigwedge$*P*. *cBex cempty P* = *False*
$\bigwedge$*a B P*. *cBex* (*cinsert a B*) *P* = (*P a* ∨ *cBex B P*)
$\bigwedge$*A P f*. *cBex* (*cimage f A*) *P* = *cBex A* (λ*x*. *P* (*f x*))
$\bigwedge$*A P*. (¬ *cBex A P*) = *cBall A* (λ*x*. ¬ *P x*)
by *auto*

lemma *cBall-simps* [*simp*, *no-atp*]:
$\bigwedge$*A P Q*. *cBall A* (λ*x*. *P x* ∨ *Q*) = (*cBall A P* ∨ *Q*)
$\bigwedge$*A P Q*. *cBall A* (λ*x*. *P* ∨ *Q x*) = (*P* ∨ *cBall A Q*)
$\bigwedge$*A P Q*. *cBall A* (λ*x*. *P* ⟶ *Q x*) = (*P* ⟶ *cBall A Q*)
$\bigwedge$*A P Q*. *cBall A* (λ*x*. *P x* ⟶ *Q*) = (*cBex A P* ⟶ *Q*)
$\bigwedge$*P*. *cBall cempty P* = *True*
$\bigwedge$*a B P*. *cBall* (*cinsert a B*) *P* = (*P a* ∧ *cBall B P*)
$\bigwedge$*A P f*. *cBall* (*cimage f A*) *P* = *cBall A* (λ*x*. *P* (*f x*))
$\bigwedge$*A P*. (¬ *cBall A P*) = *cBex A* (λ*x*. ¬ *P x*)
by *auto*

lemma *atomize-cBall*:
($\bigwedge$*x*. *cin x A* ==> *P x*) == *Trueprop* (*cBall A* (λ*x*. *P x*))
apply (*simp only*: *atomize-all atomize-imp*)
apply (*rule equal-intr-rule*)

by (*transfer*, *simp*)+

25.3.5 *cUnion*

lemma *cUNION-cimage*: *cUNION* (*cimage f A*) *g* = *cUNION A* (*g* ∘ *f*)
including *cset.lifting* **by** *transfer auto*

25.4 Setup for Lifting/Transfer

25.4.1 Relator and predicator properties

lift-definition *rel-cset* :: (*'a* ⇒ *'b* ⇒ *bool*) ⇒ *'a cset* ⇒ *'b cset* ⇒ *bool*
is *rel-set* **parametric** *rel-set-transfer* **.**

lemma *rel-cset-alt-def*:
rel-cset R a b ⟷
(∀ *t* ∈ *rcset a*. ∃ *u* ∈ *rcset b*. *R t u*) ∧
(∀ *t* ∈ *rcset b*. ∃ *u* ∈ *rcset a*. *R u t*)
by(*simp add*: *rel-cset-def rel-set-def*)

lemma *rel-cset-iff*:
rel-cset R a b ⟷
(∀ *t*. *cin t a* ⟶ (∃ *u*. *cin u b* ∧ *R t u*)) ∧
(∀ *t*. *cin t b* ⟶ (∃ *u*. *cin u a* ∧ *R u t*))
by *transfer*(*auto simp add*: *rel-set-def*)

lemma *rel-cset-cUNION*:
⟦ *rel-cset Q A B*; *rel-fun Q* (*rel-cset R*) *f g* ⟧
⟹ *rel-cset R* (*cUNION A f*) (*cUNION B g*)
unfolding *rel-fun-def* **by** *transfer*(*erule rel-set-UNION*, *simp add*: *rel-fun-def*)

lemma *rel-cset-csingle-iff* [*simp*]: *rel-cset R* (*csingle x*) (*csingle y*) ⟷ *R x y*
by *transfer*(*auto simp add*: *rel-set-def*)

25.4.2 Transfer rules for the Transfer package

Unconditional transfer rules

context begin interpretation *lifting-syntax* **.**

lemmas *cempty-parametric* [*transfer-rule*] = *empty-transfer*[*Transfer.transferred*]

lemma *cinsert-parametric* [*transfer-rule*]:
(*A* ===> *rel-cset A* ===> *rel-cset A*) *cinsert cinsert*
unfolding *rel-fun-def rel-cset-iff* **by** *blast*

lemma *cUn-parametric* [*transfer-rule*]:
(*rel-cset A* ===> *rel-cset A* ===> *rel-cset A*) *cUn cUn*
unfolding *rel-fun-def rel-cset-iff* **by** *blast*

lemma *cUnion-parametric* [*transfer-rule*]:

(*rel-cset* (*rel-cset* *A*) ===> *rel-cset* *A*) *cUnion* *cUnion*
unfolding *rel-fun-def*
by *transfer* (*auto simp*: *rel-set-def*, *metis*+)

lemma *cimage-parametric* [*transfer-rule*]:
((*A* ===> *B*) ===> *rel-cset* *A* ===> *rel-cset* *B*) *cimage* *cimage*
unfolding *rel-fun-def* *rel-cset-iff* **by** *blast*

lemma *cBall-parametric* [*transfer-rule*]:
(*rel-cset* *A* ===> (*A* ===> *op* =) ===> *op* =) *cBall* *cBall*
unfolding *rel-cset-iff* *rel-fun-def* **by** *blast*

lemma *cBex-parametric* [*transfer-rule*]:
(*rel-cset* *A* ===> (*A* ===> *op* =) ===> *op* =) *cBex* *cBex*
unfolding *rel-cset-iff* *rel-fun-def* **by** *blast*

lemma *rel-cset-parametric* [*transfer-rule*]:
((*A* ===> *B* ===> *op* =) ===> *rel-cset* *A* ===> *rel-cset* *B* ===> *op* =) *rel-cset* *rel-cset*
unfolding *rel-fun-def*
using *rel-set-transfer*[*unfolded* *rel-fun-def*, *rule-format*, *Transfer.transferred*, **where** *A* = *A* **and** *B* = *B*]
by *simp*

Rules requiring bi-unique, bi-total or right-total relations

lemma *cin-parametric* [*transfer-rule*]:
bi-unique *A* ⟹ (*A* ===> *rel-cset* *A* ===> *op* =) *cin* *cin*
unfolding *rel-fun-def* *rel-cset-iff* *bi-unique-def* **by** *metis*

lemma *cInt-parametric* [*transfer-rule*]:
bi-unique *A* ⟹ (*rel-cset* *A* ===> *rel-cset* *A* ===> *rel-cset* *A*) *cInt* *cInt*
unfolding *rel-fun-def*
using *inter-transfer*[*unfolded* *rel-fun-def*, *rule-format*, *Transfer.transferred*]
by *blast*

lemma *cDiff-parametric* [*transfer-rule*]:
bi-unique *A* ⟹ (*rel-cset* *A* ===> *rel-cset* *A* ===> *rel-cset* *A*) *cDiff* *cDiff*
unfolding *rel-fun-def*
using *Diff-transfer*[*unfolded* *rel-fun-def*, *rule-format*, *Transfer.transferred*] **by** *blast*

lemma *csubset-parametric* [*transfer-rule*]:
bi-unique *A* ⟹ (*rel-cset* *A* ===> *rel-cset* *A* ===> *op* =) *csubset-eq* *csubset-eq*
unfolding *rel-fun-def*
using *subset-transfer*[*unfolded* *rel-fun-def*, *rule-format*, *Transfer.transferred*] **by** *blast*

end

lifting-update *cset.lifting*

lifting-forget *cset.lifting*

25.5 Registration as BNF

lemma *card-of-countable-sets-range*:
fixes A :: *'a set*
shows $|\{X.\ X \subseteq A \wedge \mathit{countable}\ X \wedge X \neq \{\}\}| \leq o\ |\{f{::}\mathit{nat} \Rightarrow {'a}.\ \mathit{range}\ f \subseteq A\}|$
apply(*rule card-of-ordLeqI*[*of from-nat-into*]) **using** *inj-on-from-nat-into*
unfolding *inj-on-def* **by** *auto*

lemma *card-of-countable-sets-Func*:
$|\{X.\ X \subseteq A \wedge \mathit{countable}\ X \wedge X \neq \{\}\}| \leq o\ |A|\ \hat{}c\ \mathit{natLeq}$
using *card-of-countable-sets-range card-of-Func-UNIV*[*THEN ordIso-symmetric*]
unfolding *cexp-def Field-natLeq Field-card-of*
by (*rule ordLeq-ordIso-trans*)

lemma *ordLeq-countable-subsets*:
$|A| \leq o\ |\{X.\ X \subseteq A \wedge \mathit{countable}\ X\}|$
apply (*rule card-of-ordLeqI*[*of* λ *a*. {*a*}]) **unfolding** *inj-on-def* **by** *auto*

lemma *finite-countable-subset*:
finite $\{X.\ X \subseteq A \wedge \mathit{countable}\ X\} \longleftrightarrow$ *finite A*
apply (*rule iffI*)
 apply (*erule contrapos-pp*)
 apply (*rule card-of-ordLeq-infinite*)
 apply (*rule ordLeq-countable-subsets*)
 apply *assumption*
apply (*rule finite-Collect-conjI*)
apply (*rule disjI1*)
apply (*erule finite-Collect-subsets*)
done

lemma *rcset-to-rcset*: *countable A* $\Longrightarrow$ *rcset* (*the-inv rcset A*) = *A*
 including *cset.lifting*
 apply (*rule f-the-inv-into-f*[*unfolded inj-on-def image-iff*])
  apply *transfer'* **apply** *simp*
 apply *transfer'* **apply** *simp*
 done

lemma *Collect-Int-Times*: $\{(x, y).\ R\ x\ y\} \cap A \times B = \{(x, y).\ R\ x\ y \wedge x \in A \wedge y \in B\}$
by *auto*

lemma *rel-cset-aux*:
$(\forall t \in \mathit{rcset}\ a.\ \exists u \in \mathit{rcset}\ b.\ R\ t\ u) \wedge (\forall t \in \mathit{rcset}\ b.\ \exists u \in \mathit{rcset}\ a.\ R\ u\ t) \longleftrightarrow$
 $((\mathit{BNF\text{-}Def.Grp}\ \{x.\ \mathit{rcset}\ x \subseteq \{(a, b).\ R\ a\ b\}\}\ (\mathit{cimage}\ \mathit{fst}))^{-1-1}\ \mathit{OO}$
  $\mathit{BNF\text{-}Def.Grp}\ \{x.\ \mathit{rcset}\ x \subseteq \{(a, b).\ R\ a\ b\}\}\ (\mathit{cimage}\ \mathit{snd}))\ a\ b$ (**is** *?L = ?R*)
proof

assume *?L*
def $R' \equiv$ *the-inv rcset* (*Collect* (*case-prod R*) $\cap$ (*rcset a* $\times$ *rcset b*))
(**is** *the-inv rcset ?L'*)
have *L*: *countable ?L'* **by** *auto*
hence $*$: *rcset* R' = *?L'* **unfolding** R'*-def* **by** (*intro rcset-to-rcset*)
thus *?R* **unfolding** *Grp-def relcompp.simps conversep.simps* **including** *cset.lifting*
proof (*intro CollectI case-prodI exI*[*of - a*] *exI*[*of - b*] *exI*[*of - R'*] *conjI refl*)
from $*$ ‹*?L*› **show** *a* = *cimage fst R'* **by** *transfer* (*auto simp*: *image-def Collect-Int-Times*)
from $*$ ‹*?L*› **show** *b* = *cimage snd R'* **by** *transfer* (*auto simp*: *image-def Collect-Int-Times*)
qed *simp-all*
next
assume *?R* **thus** *?L* **unfolding** *Grp-def relcompp.simps conversep.simps*
by (*simp add*: *subset-eq Ball-def*)(*transfer*, *auto simp add*: *cimage.rep-eq*, *metis snd-conv*, *metis fst-conv*)
qed

bnf *'a cset*
map: *cimage*
sets: *rcset*
bd: *natLeq*
wits: *cempty*
rel: *rel-cset*
proof −
show *cimage id* = *id* **by** *auto*
next
fix *f g* **show** *cimage* ($g \circ f$) = *cimage g* $\circ$ *cimage f* **by** *fastforce*
next
fix *C f g* **assume** *eq*: $\bigwedge a.\ a \in$ *rcset C* $\Longrightarrow f\ a = g\ a$
thus *cimage f C* = *cimage g C* **including** *cset.lifting* **by** *transfer force*
next
fix *f* **show** *rcset* $\circ$ *cimage f* = *op* ' *f* $\circ$ *rcset* **including** *cset.lifting* **by** *transfer' fastforce*
next
show *card-order natLeq* **by** (*rule natLeq-card-order*)
next
show *cinfinite natLeq* **by** (*rule natLeq-cinfinite*)
next
fix *C* **show** |*rcset C*| $\leq o$ *natLeq*
including *cset.lifting* **by** *transfer* (*unfold countable-card-le-natLeq*)
next
fix *R S*
show *rel-cset R OO rel-cset S* $\leq$ *rel-cset* (*R OO S*)
unfolding *rel-cset-alt-def*[*abs-def*] **by** *fast*
next
fix *R*
show *rel-cset R* = ($\lambda x\ y.\ \exists z.$ *rcset z* $\subseteq$ {(*x*, *y*). *R x y*} $\wedge$
cimage fst z = *x* $\wedge$ *cimage snd z* = *y*)

unfolding *rel-cset-alt-def*[*abs-def*] *rel-cset-aux*[*unfolded OO-Grp-alt*] **by** *simp*
qed(*simp add*: *bot-cset.rep-eq*)

end

26 Debugging facilities for code generated towards Isabelle/ML

theory *Debug*
imports *Main*
begin

context
begin

qualified definition *trace* :: *String.literal* ⇒ *unit* **where**
[*simp*]: *trace s* = ()

qualified definition *tracing* :: *String.literal* ⇒ ′*a* ⇒ ′*a* **where**
[*simp*]: *tracing s* = *id*

lemma [*code*]:
tracing s = (*let u* = *trace s in id*)
by *simp*

qualified definition *flush* :: ′*a* ⇒ *unit* **where**
[*simp*]: *flush x* = ()

qualified definition *flushing* :: ′*a* ⇒ ′*b* ⇒ ′*b* **where**
[*simp*]: *flushing x* = *id*

lemma [*code*, *code-unfold*]:
flushing x = (*let u* = *flush x in id*)
by *simp*

qualified definition *timing* :: *String.literal* ⇒ (′*a* ⇒ ′*b*) ⇒ ′*a* ⇒ ′*b* **where**
[*simp*]: *timing s f x* = *f x*

end

code-printing
constant *Debug.trace* ⇀ (*Eval*) *Output.tracing*
| **constant** *Debug.flush* ⇀ (*Eval*) *Output.tracing*/ (@{*make*′*-string*} -) — note indirection via antiquotation
| **constant** *Debug.timing* ⇀ (*Eval*) *Timing.timeap*′*-msg*

code-reserved *Eval Output Timing*

end

27 Sequence of Properties on Subsequences

theory *Diagonal-Subsequence*
imports *Complex-Main*
begin

locale *subseqs* =
  fixes $P::nat \Rightarrow (nat \Rightarrow nat) \Rightarrow bool$
  assumes *ex-subseq*: $\bigwedge n\ s.\ subseq\ s \Longrightarrow \exists\, r'.\ subseq\ r' \wedge P\ n\ (s\ o\ r')$
begin

definition *reduce* **where** $reduce\ s\ n = (SOME\ r'.\ subseq\ r' \wedge P\ n\ (s\ o\ r'))$

lemma *subseq-reduce*[*intro*, *simp*]:
  $subseq\ s \Longrightarrow subseq\ (reduce\ s\ n)$
  unfolding *reduce-def* **by** (*rule someI2-ex*[*OF ex-subseq*]) *auto*

lemma *reduce-holds*:
  $subseq\ s \Longrightarrow P\ n\ (s\ o\ reduce\ s\ n)$
  unfolding *reduce-def* **by** (*rule someI2-ex*[*OF ex-subseq*]) (*auto simp*: *o-def*)

primrec *seqseq* **where**
  $seqseq\ 0 = id$
| $seqseq\ (Suc\ n) = seqseq\ n\ o\ reduce\ (seqseq\ n)\ n$

lemma *subseq-seqseq*[*intro*, *simp*]: $subseq\ (seqseq\ n)$
proof (*induct n*)
  case *0* **thus** *?case* **by** (*simp add*: *subseq-def*)
next
  case (*Suc n*) **thus** *?case* **by** (*subst seqseq.simps*) (*auto intro*!: *subseq-o*)
qed

lemma *seqseq-holds*:
  $P\ n\ (seqseq\ (Suc\ n))$
proof −
  have $P\ n\ (seqseq\ n\ o\ reduce\ (seqseq\ n)\ n)$
    by (*intro reduce-holds subseq-seqseq*)
  thus *?thesis* **by** *simp*
qed

definition *diagseq* **where** $diagseq\ i = seqseq\ i\ i$

lemma *subseq-mono*: $subseq\ f \Longrightarrow a \le b \Longrightarrow f\ a \le f\ b$
  by (*metis le-eq-less-or-eq subseq-mono*)

lemma *subseq-strict-mono*: $subseq\ f \Longrightarrow a < b \Longrightarrow f\ a < f\ b$
  by (*simp add*: *subseq-def*)

lemma *diagseq-mono*: *diagseq n* < *diagseq* (*Suc n*)
proof −
  have *diagseq n* < *seqseq n* (*Suc n*)
    using *subseq-seqseq*[*of n*] **by** (*simp add*: *diagseq-def subseq-def*)
  also have ... ≤ *seqseq n* (*reduce* (*seqseq n*) *n* (*Suc n*))
    by (*auto intro*: *subseq-mono seq-suble*)
  also have ... = *diagseq* (*Suc n*) **by** (*simp add*: *diagseq-def*)
  finally show *?thesis* **.**
qed

lemma *subseq-diagseq*: *subseq diagseq*
  using *diagseq-mono* **by** (*simp add*: *subseq-Suc-iff diagseq-def*)

primrec *fold-reduce* **where**
  fold-reduce n 0 = *id*
| *fold-reduce n* (*Suc k*) = *fold-reduce n k o reduce* (*seqseq* (*n* + *k*)) (*n* + *k*)

lemma *subseq-fold-reduce*[*intro*, *simp*]: *subseq* (*fold-reduce n k*)
proof (*induct k*)
  case (*Suc k*) **from** *subseq-o*[*OF this subseq-reduce*] **show** *?case* **by** (*simp add*: *o-def*)
qed (*simp add*: *subseq-def*)

lemma *ex-subseq-reduce-index*: *seqseq* (*n* + *k*) = *seqseq n o fold-reduce n k*
  by (*induct k*) *simp-all*

lemma *seqseq-fold-reduce*: *seqseq n* = *fold-reduce 0 n*
  by (*induct n*) (*simp-all*)

lemma *diagseq-fold-reduce*: *diagseq n* = *fold-reduce 0 n n*
  using *seqseq-fold-reduce* **by** (*simp add*: *diagseq-def*)

lemma *fold-reduce-add*: *fold-reduce 0* (*m* + *n*) = *fold-reduce 0 m o fold-reduce m n*
  by (*induct n*) *simp-all*

lemma *diagseq-add*: *diagseq* (*k* + *n*) = (*seqseq k o* (*fold-reduce k n*)) (*k* + *n*)
proof −
  have *diagseq* (*k* + *n*) = *fold-reduce 0* (*k* + *n*) (*k* + *n*)
    by (*simp add*: *diagseq-fold-reduce*)
  also have ... = (*seqseq k o fold-reduce k n*) (*k* + *n*)
    unfolding *fold-reduce-add seqseq-fold-reduce* **..**
  finally show *?thesis* **.**
qed

lemma *diagseq-sub*:
  assumes *m* ≤ *n* **shows** *diagseq n* = (*seqseq m o* (*fold-reduce m* (*n* − *m*))) *n*
  using *diagseq-add*[*of m n* − *m*] *assms* **by** *simp*

lemma *subseq-diagonal-rest*: *subseq* $(\lambda x.\ \textit{fold-reduce}\ k\ x\ (k + x))$
unfolding *subseq-Suc-iff fold-reduce.simps o-def*
proof
fix n
have *fold-reduce* $k\ n\ (k + n) <$ *fold-reduce* $k\ n\ (k + \textit{Suc}\ n)$ (**is** *?lhs* < -)
by (*auto intro*: *subseq-strict-mono*)
also have ... $\le$ *fold-reduce* $k\ n$ (*reduce* (*seqseq* $(k + n)$) $(k + n)$ $(k + \textit{Suc}\ n)$)
by (*rule subseq-mono*) (*auto intro*!: *seq-suble subseq-mono*)
finally show *?lhs* < ... .
qed

lemma *diagseq-seqseq*: *diagseq* o $(op + k) =$ (*seqseq* k o $(\lambda x.\ \textit{fold-reduce}\ k\ x\ (k + x)))$
by (*auto simp*: *o-def diagseq-add*)

lemma *diagseq-holds*:
assumes *subseq-stable*: $\bigwedge r\ s\ n.\ \textit{subseq}\ r \Longrightarrow P\ n\ s \Longrightarrow P\ n\ (s\ o\ r)$
shows $P\ k$ (*diagseq* o $(op + (\textit{Suc}\ k)))$
unfolding *diagseq-seqseq* **by** (*intro subseq-stable subseq-diagonal-rest seqseq-holds*)

end

end

28 Handling Disjoint Sets

theory *Disjoint-Sets*
imports *Main*
begin

lemma *range-subsetD*: *range* $f \subseteq B \Longrightarrow f\ i \in B$
by *blast*

lemma *Int-Diff-disjoint*: $A \cap B \cap (A - B) = \{\}$
by *blast*

lemma *Int-Diff-Un*: $A \cap B \cup (A - B) = A$
by *blast*

lemma *mono-Un*: *mono* $A \Longrightarrow (\bigcup i{\le}n.\ A\ i) = A\ n$
unfolding *mono-def* **by** *auto*

28.1 Set of Disjoint Sets

abbreviation *disjoint* :: *'a set set* $\Rightarrow$ *bool* **where** *disjoint* $\equiv$ *pairwise disjnt*

lemma *disjoint-def*: *disjoint* $A \longleftrightarrow (\forall a{\in}A.\ \forall b{\in}A.\ a \neq b \longrightarrow a \cap b = \{\})$
unfolding *pairwise-def disjnt-def* **by** *auto*

lemma *disjointI*:
($\bigwedge a\ b.\ a \in A \Longrightarrow b \in A \Longrightarrow a \neq b \Longrightarrow a \cap b = \{\}) \Longrightarrow disjoint\ A$
unfolding *disjoint-def* **by** *auto*

lemma *disjointD*:
$disjoint\ A \Longrightarrow a \in A \Longrightarrow b \in A \Longrightarrow a \neq b \Longrightarrow a \cap b = \{\}$
unfolding *disjoint-def* **by** *auto*

lemma *disjoint-INT*:
assumes $*$: $\bigwedge i.\ i \in I \Longrightarrow disjoint\ (F\ i)$
shows $disjoint\ \{\bigcap i{\in}I.\ X\ i \mid X.\ \forall i{\in}I.\ X\ i \in F\ i\}$
proof (*safe intro*!: *disjointI del*: *equalityI*)
fix $A\ B :: 'a \Rightarrow 'b\ set$ **assume** $(\bigcap i{\in}I.\ A\ i) \neq (\bigcap i{\in}I.\ B\ i)$
then obtain i **where** $A\ i \neq B\ i\ i \in I$
by *auto*
moreover assume $\forall i{\in}I.\ A\ i \in F\ i\ \forall i{\in}I.\ B\ i \in F\ i$
ultimately show $(\bigcap i{\in}I.\ A\ i) \cap (\bigcap i{\in}I.\ B\ i) = \{\}$
using $*$[*OF* ‹$i{\in}I$›, *THEN disjointD*, *of A i B i*]
by (*auto simp*: *INT-Int-distrib*[*symmetric*])
qed

28.1.1 Family of Disjoint Sets

definition *disjoint-family-on* :: $('i \Rightarrow 'a\ set) \Rightarrow 'i\ set \Rightarrow bool$ **where**
$disjoint\text{-}family\text{-}on\ A\ S \longleftrightarrow (\forall m{\in}S.\ \forall n{\in}S.\ m \neq n \longrightarrow A\ m \cap A\ n = \{\})$

abbreviation $disjoint\text{-}family\ A \equiv disjoint\text{-}family\text{-}on\ A\ UNIV$

lemma *disjoint-family-onD*:
$disjoint\text{-}family\text{-}on\ A\ I \Longrightarrow i \in I \Longrightarrow j \in I \Longrightarrow i \neq j \Longrightarrow A\ i \cap A\ j = \{\}$
by (*auto simp*: *disjoint-family-on-def*)

lemma *disjoint-family-subset*: $disjoint\text{-}family\ A \Longrightarrow (\bigwedge x.\ B\ x \subseteq A\ x) \Longrightarrow disjoint\text{-}family\ B$
by (*force simp add*: *disjoint-family-on-def*)

lemma *disjoint-family-on-bisimulation*:
assumes $disjoint\text{-}family\text{-}on\ f\ S$
and $\bigwedge n\ m.\ n \in S \Longrightarrow m \in S \Longrightarrow n \neq m \Longrightarrow f\ n \cap f\ m = \{\} \Longrightarrow g\ n \cap g\ m = \{\}$
shows $disjoint\text{-}family\text{-}on\ g\ S$
using *assms* **unfolding** *disjoint-family-on-def* **by** *auto*

lemma *disjoint-family-on-mono*:
$A \subseteq B \Longrightarrow disjoint\text{-}family\text{-}on\ f\ B \Longrightarrow disjoint\text{-}family\text{-}on\ f\ A$
unfolding *disjoint-family-on-def* **by** *auto*

lemma *disjoint-family-Suc*:

$(\bigwedge n.\ A\ n \subseteq A\ (Suc\ n)) \Longrightarrow$ *disjoint-family* $(\lambda i.\ A\ (Suc\ i) - A\ i)$
using *lift-Suc-mono-le*[*of A*]
by (*auto simp add*: *disjoint-family-on-def*)
(*metis insert-absorb insert-subset le-SucE le-antisym not-le-imp-less less-imp-le*)

lemma *disjoint-family-on-disjoint-image*:
disjoint-family-on $A\ I \Longrightarrow$ *disjoint* $(A\ `\ I)$
unfolding *disjoint-family-on-def disjoint-def* **by** *force*

lemma *disjoint-family-on-vimageI*: *disjoint-family-on* $F\ I \Longrightarrow$ *disjoint-family-on* $(\lambda i.\ f\ -`\ F\ i)\ I$
by (*auto simp*: *disjoint-family-on-def*)

lemma *disjoint-image-disjoint-family-on*:
assumes *d*: *disjoint* $(A\ `\ I)$ **and** *i*: *inj-on* $A\ I$
shows *disjoint-family-on* $A\ I$
unfolding *disjoint-family-on-def*
proof (*intro ballI impI*)
fix $n\ m$ **assume** *nm*: $m \in I\ n \in I$ **and** $n \neq m$
with *i*[*THEN inj-onD*, *of n m*] **show** $A\ n \cap A\ m = \{\}$
by (*intro disjointD*[*OF d*]) *auto*
qed

lemma *disjoint-UN*:
assumes *F*: $\bigwedge i.\ i \in I \Longrightarrow$ *disjoint* $(F\ i)$ **and** $*$: *disjoint-family-on* $(\lambda i.\ \bigcup F\ i)\ I$
shows *disjoint* $(\bigcup i{\in}I.\ F\ i)$
proof (*safe intro*!: *disjointI del*: *equalityI*)
fix $A\ B\ i\ j$ **assume** $A \neq B\ A \in F\ i\ i \in I\ B \in F\ j\ j \in I$
show $A \cap B = \{\}$
proof *cases*
assume $i = j$ **with** *F*[*of i*] ‹$i \in I$› ‹$A \in F\ i$› ‹$B \in F\ j$› ‹$A \neq B$› **show** $A \cap B = \{\}$
by (*auto dest*: *disjointD*)
next
assume $i \neq j$
with $*$ ‹$i{\in}I$› ‹$j{\in}I$› **have** $(\bigcup F\ i) \cap (\bigcup F\ j) = \{\}$
by (*rule disjoint-family-onD*)
with ‹$A{\in}F\ i$› ‹$i{\in}I$› ‹$B{\in}F\ j$› ‹$j{\in}I$›
show $A \cap B = \{\}$
by *auto*
qed
qed

lemma *disjoint-union*: *disjoint* $C \Longrightarrow$ *disjoint* $B \Longrightarrow \bigcup C \cap \bigcup B = \{\} \Longrightarrow$ *disjoint* $(C \cup B)$
using *disjoint-UN*[*of* $\{C, B\}\ \lambda x.\ x$] **by** (*auto simp add*: *disjoint-family-on-def*)

28.2 Construct Disjoint Sequences

definition *disjointed* :: (*nat* ⇒ *'a set*) ⇒ *nat* ⇒ *'a set* **where**
disjointed A n = *A n* − (⋃*i*∈{*0*..<*n*}. *A i*)

lemma *finite-UN-disjointed-eq*: (⋃*i*∈{*0*..<*n*}. *disjointed A i*) = (⋃*i*∈{*0*..<*n*}. *A i*)
proof (*induct n*)
case *0* **show** *?case* **by** *simp*
next
case (*Suc n*)
thus *?case* **by** (*simp add*: *atLeastLessThanSuc disjointed-def*)
qed

lemma *UN-disjointed-eq*: (⋃*i*. *disjointed A i*) = (⋃*i*. *A i*)
by (*rule UN-finite2-eq* [**where** *k*=*0*])
(*simp add*: *finite-UN-disjointed-eq*)

lemma *less-disjoint-disjointed*: *m* < *n* ⟹ *disjointed A m* ∩ *disjointed A n* = {}
by (*auto simp add*: *disjointed-def*)

lemma *disjoint-family-disjointed*: *disjoint-family* (*disjointed A*)
by (*simp add*: *disjoint-family-on-def*)
(*metis neq-iff Int-commute less-disjoint-disjointed*)

lemma *disjointed-subset*: *disjointed A n* ⊆ *A n*
by (*auto simp add*: *disjointed-def*)

lemma *disjointed-0*[*simp*]: *disjointed A 0* = *A 0*
by (*simp add*: *disjointed-def*)

lemma *disjointed-mono*: *mono A* ⟹ *disjointed A* (*Suc n*) = *A* (*Suc n*) − *A n*
using *mono-Un*[*of A*] **by** (*simp add*: *disjointed-def atLeastLessThanSuc-atLeastAtMost atLeast0AtMost*)

end

29 Lists with elements distinct as canonical example for datatype invariants

theory *Dlist*
imports *Main*
begin

29.1 The type of distinct lists

typedef *'a dlist* = {*xs*::*'a list*. *distinct xs*}
morphisms *list-of-dlist Abs-dlist*
proof

show [] $\in$ {*xs*. *distinct xs*} **by** *simp*
qed

setup-lifting *type-definition-dlist*

lemma *dlist-eq-iff*:
dxs = *dys* $\longleftrightarrow$ *list-of-dlist dxs* = *list-of-dlist dys*
by (*simp add*: *list-of-dlist-inject*)

lemma *dlist-eqI*:
list-of-dlist dxs = *list-of-dlist dys* $\Longrightarrow$ *dxs* = *dys*
by (*simp add*: *dlist-eq-iff*)

Formal, totalized constructor for *'a dlist*:

definition *Dlist* :: *'a list* $\Rightarrow$ *'a dlist* **where**
Dlist xs = *Abs-dlist* (*remdups xs*)

lemma *distinct-list-of-dlist* [*simp*, *intro*]:
distinct (*list-of-dlist dxs*)
using *list-of-dlist* [*of dxs*] **by** *simp*

lemma *list-of-dlist-Dlist* [*simp*]:
list-of-dlist (*Dlist xs*) = *remdups xs*
by (*simp add*: *Dlist-def Abs-dlist-inverse*)

lemma *remdups-list-of-dlist* [*simp*]:
remdups (*list-of-dlist dxs*) = *list-of-dlist dxs*
by *simp*

lemma *Dlist-list-of-dlist* [*simp*, *code abstype*]:
Dlist (*list-of-dlist dxs*) = *dxs*
by (*simp add*: *Dlist-def list-of-dlist-inverse distinct-remdups-id*)

Fundamental operations:

context
begin

qualified definition *empty* :: *'a dlist* **where**
empty = *Dlist* []

qualified definition *insert* :: *'a* $\Rightarrow$ *'a dlist* $\Rightarrow$ *'a dlist* **where**
insert x dxs = *Dlist* (*List.insert x* (*list-of-dlist dxs*))

qualified definition *remove* :: *'a* $\Rightarrow$ *'a dlist* $\Rightarrow$ *'a dlist* **where**
remove x dxs = *Dlist* (*remove1 x* (*list-of-dlist dxs*))

qualified definition *map* :: (*'a* $\Rightarrow$ *'b*) $\Rightarrow$ *'a dlist* $\Rightarrow$ *'b dlist* **where**
map f dxs = *Dlist* (*remdups* (*List.map f* (*list-of-dlist dxs*)))

qualified definition *filter* :: ('a ⇒ bool) ⇒ 'a dlist ⇒ 'a dlist **where**
filter P dxs = Dlist (List.filter P (list-of-dlist dxs))

end

Derived operations:

context
begin

qualified definition *null* :: 'a dlist ⇒ bool **where**
null dxs = List.null (list-of-dlist dxs)

qualified definition *member* :: 'a dlist ⇒ 'a ⇒ bool **where**
member dxs = List.member (list-of-dlist dxs)

qualified definition *length* :: 'a dlist ⇒ nat **where**
length dxs = List.length (list-of-dlist dxs)

qualified definition *fold* :: ('a ⇒ 'b ⇒ 'b) ⇒ 'a dlist ⇒ 'b ⇒ 'b **where**
fold f dxs = List.fold f (list-of-dlist dxs)

qualified definition *foldr* :: ('a ⇒ 'b ⇒ 'b) ⇒ 'a dlist ⇒ 'b ⇒ 'b **where**
foldr f dxs = List.foldr f (list-of-dlist dxs)

end

29.2 Executable version obeying invariant

lemma *list-of-dlist-empty* [*simp*, *code abstract*]:
list-of-dlist Dlist.empty = []
by (*simp add*: *Dlist.empty-def*)

lemma *list-of-dlist-insert* [*simp*, *code abstract*]:
list-of-dlist (Dlist.insert x dxs) = List.insert x (list-of-dlist dxs)
by (*simp add*: *Dlist.insert-def*)

lemma *list-of-dlist-remove* [*simp*, *code abstract*]:
list-of-dlist (Dlist.remove x dxs) = remove1 x (list-of-dlist dxs)
by (*simp add*: *Dlist.remove-def*)

lemma *list-of-dlist-map* [*simp*, *code abstract*]:
list-of-dlist (Dlist.map f dxs) = remdups (List.map f (list-of-dlist dxs))
by (*simp add*: *Dlist.map-def*)

lemma *list-of-dlist-filter* [*simp*, *code abstract*]:
list-of-dlist (Dlist.filter P dxs) = List.filter P (list-of-dlist dxs)
by (*simp add*: *Dlist.filter-def*)

Explicit executable conversion

definition *dlist-of-list* [*simp*]:

dlist-of-list = *Dlist*

lemma [*code abstract*]:
list-of-dlist (*dlist-of-list xs*) = *remdups xs*
by *simp*

Equality

instantiation *dlist* :: (*equal*) *equal*
begin

definition *HOL.equal dxs dys* ⟷ *HOL.equal* (*list-of-dlist dxs*) (*list-of-dlist dys*)

instance
by *standard* (*simp add*: *equal-dlist-def equal list-of-dlist-inject*)

end

declare *equal-dlist-def* [*code*]

lemma [*code nbe*]: *HOL.equal* (*dxs* :: '*a*::*equal dlist*) *dxs* ⟷ *True*
by (*fact equal-refl*)

29.3 Induction principle and case distinction

lemma *dlist-induct* [*case-names empty insert*, *induct type*: *dlist*]:
assumes *empty*: *P Dlist.empty*
assumes *insrt*: ⋀*x dxs*. ¬ *Dlist.member dxs x* ⟹ *P dxs* ⟹ *P* (*Dlist.insert x dxs*)
shows *P dxs*
proof (*cases dxs*)
case (*Abs-dlist xs*)
then have *distinct xs* **and** *dxs*: *dxs* = *Dlist xs*
by (*simp-all add*: *Dlist-def distinct-remdups-id*)
from ‹*distinct xs*› **have** *P* (*Dlist xs*)
proof (*induct xs*)
case *Nil* **from** *empty* **show** *?case* **by** (*simp add*: *Dlist.empty-def*)
next
case (*Cons x xs*)
then have ¬ *Dlist.member* (*Dlist xs*) *x* **and** *P* (*Dlist xs*)
by (*simp-all add*: *Dlist.member-def List.member-def*)
with *insrt* **have** *P* (*Dlist.insert x* (*Dlist xs*)) **.**
with *Cons* **show** *?case* **by** (*simp add*: *Dlist.insert-def distinct-remdups-id*)
qed
with *dxs* **show** *P dxs* **by** *simp*
qed

lemma *dlist-case* [*cases type*: *dlist*]:
obtains (*empty*) *dxs* = *Dlist.empty*
| (*insert*) *x dys* **where** ¬ *Dlist.member dys x* **and** *dxs* = *Dlist.insert x dys*
proof (*cases dxs*)

```
  case (Abs-dlist xs)
  then have dxs: dxs = Dlist xs and distinct: distinct xs
    by (simp-all add: Dlist-def distinct-remdups-id)
  show thesis
  proof (cases xs)
    case Nil with dxs
    have dxs = Dlist.empty by (simp add: Dlist.empty-def)
    with empty show ?thesis .
  next
    case (Cons x xs)
    with dxs distinct have ¬ Dlist.member (Dlist xs) x
      and dxs = Dlist.insert x (Dlist xs)
     by (simp-all add: Dlist.member-def List.member-def Dlist.insert-def distinct-remdups-id)
    with insert show ?thesis .
  qed
qed
```

29.4 Functorial structure

```
functor map: map
  by (simp-all add: remdups-map-remdups fun-eq-iff dlist-eq-iff)
```

29.5 Quickcheck generators

```
quickcheck-generator dlist predicate: distinct constructors: Dlist.empty, Dlist.insert
```

29.6 BNF instance

```
context begin

qualified fun wpull :: ('a × 'b) list ⇒ ('b × 'c) list ⇒ ('a × 'c) list
where
  wpull [] ys = []
| wpull xs [] = []
| wpull ((a, b) # xs) ((b', c) # ys) =
  (if b ∈ snd ‘ set xs then
     (a, the (map-of (rev ((b', c) # ys)) b)) # wpull xs ((b', c) # ys)
   else if b' ∈ fst ‘ set ys then
     (the (map-of (map prod.swap (rev ((a, b) # xs))) b'), c) # wpull ((a, b) #
xs) ys
   else (a, c) # wpull xs ys)

qualified lemma wpull-eq-Nil-iff [simp]: wpull xs ys = [] ⟷ xs = [] ∨ ys = []
by(cases (xs, ys) rule: wpull.cases) simp-all

qualified lemma wpull-induct
  [consumes 1,
   case-names Nil left[xs eq in-set IH] right[xs ys eq in-set IH] step[xs ys eq IH] ]:
  assumes eq: remdups (map snd xs) = remdups (map fst ys)
  and Nil: P [] []
```

and *left*: $\bigwedge a\ b\ xs\ b'\ c\ ys.$
⟦ $b \in snd$ ‘ *set xs*; *remdups* (*map snd xs*) = *remdups* (*map fst* ((b', *c*) # *ys*));
(b, *the* (*map-of* (*rev* ((b', *c*) # *ys*)) *b*)) ∈ *set* ((b', *c*) # *ys*); *P xs* ((b', *c*) # *ys*) ⟧
⟹ *P* ((*a*, *b*) # *xs*) ((b', *c*) # *ys*)
and *right*: $\bigwedge a\ b\ xs\ b'\ c\ ys.$
⟦ $b \notin snd$ ‘ *set xs*; $b' \in fst$ ‘ *set ys*;
remdups (*map snd* ((*a*, *b*) # *xs*)) = *remdups* (*map fst ys*);
(*the* (*map-of* (*map prod.swap* (*rev* ((*a*, *b*) #*xs*))) b'), b') ∈ *set* ((*a*, *b*) # *xs*);
P ((*a*, *b*) # *xs*) *ys* ⟧
⟹ *P* ((*a*, *b*) # *xs*) ((b', *c*) # *ys*)
and *step*: $\bigwedge a\ b\ xs\ c\ ys.$
⟦ $b \notin snd$ ‘ *set xs*; $b \notin fst$ ‘ *set ys*; *remdups* (*map snd xs*) = *remdups* (*map fst ys*);
P xs ys ⟧
⟹ *P* ((*a*, *b*) # *xs*) ((*b*, *c*) # *ys*)
shows *P xs ys*
using *eq*
proof(*induction xs ys rule*: *wpull.induct*)
case *1* **thus** *?case* **by**(*simp add*: *Nil*)
next
case *2* **thus** *?case* **by**(*simp split*: *if-split-asm*)
next
case *Cons*: (*3 a b xs* b' *c ys*)
let *?xs* = (*a*, *b*) # *xs* **and** *?ys* = (b', *c*) # *ys*
consider (*xs*) $b \in snd$ ‘ *set xs* | (*ys*) $b \notin snd$ ‘ *set xs* $b' \in fst$ ‘ *set ys*
| (*step*) $b \notin snd$ ‘ *set xs* $b' \notin fst$ ‘ *set ys* **by** *auto*
thus *?case*
proof *cases*
case *xs*
with *Cons.prems* **have** *eq*: *remdups* (*map snd xs*) = *remdups* (*map fst ?ys*) **by** *auto*
from *xs eq* **have** $b \in fst$ ‘ *set ?ys* **by** (*metis list.set-map set-remdups*)
hence *map-of* (*rev ?ys*) *b* ≠ *None* **unfolding** *map-of-eq-None-iff* **by** *auto*
then obtain c' **where** *map-of* (*rev ?ys*) *b* = *Some* c' **by** *blast*
then have (*b*, *the* (*map-of* (*rev ?ys*) *b*)) ∈ *set ?ys* **by**(*auto dest*: *map-of-SomeD split*: *if-split-asm*)
from *xs eq this Cons.IH*(*1*)[*OF xs eq*] **show** *?thesis* **by**(*rule left*)
next
case *ys*
from *ys Cons.prems* **have** *eq*: *remdups* (*map snd ?xs*) = *remdups* (*map fst ys*) **by** *auto*
from *ys eq* **have** $b' \in snd$ ‘ *set ?xs* **by** (*metis list.set-map set-remdups*)
hence *map-of* (*map prod.swap* (*rev ?xs*)) b' ≠ *None*
unfolding *map-of-eq-None-iff* **by**(*auto simp add*: *image-image*)
then obtain a' **where** *map-of* (*map prod.swap* (*rev ?xs*)) b' = *Some* a' **by** *blast*
then have (*the* (*map-of* (*map prod.swap* (*rev ?xs*)) b'), b') ∈ *set ?xs*
by(*auto dest*: *map-of-SomeD split*: *if-split-asm*)

from *ys eq this Cons.IH(2)[OF ys eq]* **show** *?thesis* **by**(*rule right*)
next
case $*$: *step*
hence *remdups (map snd xs) = remdups (map fst ys) b = b'* **using** *Cons.prems* **by** *auto*
from $*$ *this(1) Cons.IH(3)[OF* $*$ *this(1)]* **show** *?thesis* **unfolding** ‹*b = b'*› **by**(*rule step*)
qed
qed

qualified lemma *set-wpull-subset*:
assumes *remdups (map snd xs) = remdups (map fst ys)*
shows *set (wpull xs ys)* ⊆ *set xs O set ys*
using *assms* **by**(*induction xs ys rule: wpull-induct*) *auto*

qualified lemma *set-fst-wpull*:
assumes *remdups (map snd xs) = remdups (map fst ys)*
shows *fst ‘ set (wpull xs ys) = fst ‘ set xs*
using *assms* **by**(*induction xs ys rule: wpull-induct*)(*auto intro: rev-image-eqI*)

qualified lemma *set-snd-wpull*:
assumes *remdups (map snd xs) = remdups (map fst ys)*
shows *snd ‘ set (wpull xs ys) = snd ‘ set ys*
using *assms* **by**(*induction xs ys rule: wpull-induct*)(*auto intro: rev-image-eqI*)

qualified lemma *wpull*:
assumes *distinct xs*
and *distinct ys*
and *set xs* ⊆ {(*x*, *y*). *R x y*}
and *set ys* ⊆ {(*x*, *y*). *S x y*}
and *eq*: *remdups (map snd xs) = remdups (map fst ys)*
shows ∃ *zs*. *distinct zs* ∧ *set zs* ⊆ {(*x*, *y*). (*R OO S*) *x y*} ∧ *remdups (map fst zs) = remdups (map fst xs)* ∧ *remdups (map snd zs) = remdups (map snd ys)*
proof(*intro exI conjI*)
let *?zs = remdups (wpull xs ys)*
show *distinct ?zs* **by** *simp*
show *set ?zs* ⊆ {(*x*, *y*). (*R OO S*) *x y*} **using** *assms(3−4) set-wpull-subset[OF eq]* **by** *fastforce*
show *remdups (map fst ?zs) = remdups (map fst xs)* **unfolding** *remdups-map-remdups* **using** *eq*
by(*induction xs ys rule: wpull-induct*)(*auto simp add: set-fst-wpull intro: rev-image-eqI*)
show *remdups (map snd ?zs) = remdups (map snd ys)* **unfolding** *remdups-map-remdups* **using** *eq*
by(*induction xs ys rule: wpull-induct*)(*auto simp add: set-snd-wpull intro: rev-image-eqI*)
qed

qualified lift-definition *set* :: *'a dlist* ⇒ *'a set* **is** *List.set* **.**

qualified lemma *map-transfer* [*transfer-rule*]:
(*rel-fun op* = (*rel-fun* (*pcr-dlist op* =) (*pcr-dlist op* =))) ($\lambda f\, x$. *remdups* (*List.map f x*)) *Dlist.map*
by(*simp add*: *rel-fun-def dlist.pcr-cr-eq cr-dlist-def Dlist.map-def remdups-remdups*)

bnf *'a dlist*
map: *Dlist.map*
sets: *set*
bd: *natLeq*
wits: *Dlist.empty*
unfolding *OO-Grp-alt mem-Collect-eq*
subgoal by(*rule ext*)(*simp add*: *dlist-eq-iff*)
subgoal by(*rule ext*)(*simp add*: *dlist-eq-iff remdups-map-remdups*)
subgoal by(*simp add*: *dlist-eq-iff set-def cong*: *list.map-cong*)
subgoal by(*simp add*: *set-def fun-eq-iff*)
subgoal by(*simp add*: *natLeq-card-order*)
subgoal by(*simp add*: *natLeq-cinfinite*)
subgoal by(*rule ordLess-imp-ordLeq*)(*simp add*: *finite-iff-ordLess-natLeq*[*symmetric*] *set-def*)
subgoal by(*rule predicate2I*)(*transfer*; *auto simp add*: *wpull*)
subgoal by(*simp add*: *set-def*)
done

lifting-update *dlist.lifting*
lifting-forget *dlist.lifting*

end

end

theory *Simps-Case-Conv*
imports *Main*
keywords *simps-of-case* :: *thy-decl* == **and** *case-of-simps* :: *thy-decl*
begin

ML-file *simps-case-conv.ML*

end

theory *Extended*
imports
Main
~~/src/HOL/Library/Simps-Case-Conv
begin

datatype *'a extended* = *Fin 'a* | *Pinf* (∞) | *Minf* ($-\infty$)

instantiation *extended* :: (*order*)*order*
begin

fun *less-eq-extended* :: *'a extended* ⇒ *'a extended* ⇒ *bool* **where**
Fin x ≤ *Fin y* = (*x* ≤ *y*) |
- ≤ *Pinf* = *True* |
Minf ≤ - = *True* |
(-::*'a extended*) ≤ - = *False*

case-of-simps *less-eq-extended-case*: *less-eq-extended.simps*

definition *less-extended* :: *'a extended* ⇒ *'a extended* ⇒ *bool* **where**
((*x*::*'a extended*) < *y*) = (*x* ≤ *y* & ¬ *y* ≤ *x*)

instance
by *intro-classes* (*auto simp*: *less-extended-def less-eq-extended-case split*: *extended.splits*)

end

instance *extended* :: (*linorder*)*linorder*
by *intro-classes* (*auto simp*: *less-eq-extended-case split*:*extended.splits*)

lemma *Minf-le*[*simp*]: *Minf* ≤ *y*
by(*cases y*) *auto*
lemma *le-Pinf*[*simp*]: *x* ≤ *Pinf*
by(*cases x*) *auto*
lemma *le-Minf*[*simp*]: *x* ≤ *Minf* ⟷ *x* = *Minf*
by(*cases x*) *auto*
lemma *Pinf-le*[*simp*]: *Pinf* ≤ *x* ⟷ *x* = *Pinf*
by(*cases x*) *auto*

lemma *less-extended-simps*[*simp*]:
Fin x < *Fin y* = (*x* < *y*)
Fin x < *Pinf* = *True*
Fin x < *Minf* = *False*
Pinf < *h* = *False*
Minf < *Fin x* = *True*
Minf < *Pinf* = *True*
l < *Minf* = *False*
by (*auto simp add*: *less-extended-def*)

lemma *min-extended-simps*[*simp*]:
min (*Fin x*) (*Fin y*) = *Fin*(*min x y*)
min xx Pinf = *xx*
min xx Minf = *Minf*
min Pinf yy = *yy*
min Minf yy = *Minf*

by (*auto simp add*: *min-def*)

lemma *max-extended-simps*[*simp*]:
max (*Fin x*) (*Fin y*) = *Fin*(*max x y*)
max xx *Pinf* = *Pinf*
max xx *Minf* = *xx*
max Pinf *yy* = *Pinf*
max Minf *yy* = *yy*
by (*auto simp add*: *max-def*)

instantiation *extended* :: (*zero*)*zero*
begin
definition *0* = *Fin*(*0*::'*a*)
instance ..
end

declare *zero-extended-def*[*symmetric*, *code-post*]

instantiation *extended* :: (*one*)*one*
begin
definition *1* = *Fin*(*1*::'*a*)
instance ..
end

declare *one-extended-def*[*symmetric*, *code-post*]

instantiation *extended* :: (*plus*)*plus*
begin

The following definition of of addition is totalized to make it asociative and commutative. Normally the sum of plus and minus infinity is undefined.

fun *plus-extended* **where**
Fin x + *Fin y* = *Fin*(*x*+*y*) |
Fin x + *Pinf* = *Pinf* |
Pinf + *Fin x* = *Pinf* |
Pinf + *Pinf* = *Pinf* |
Minf + *Fin y* = *Minf* |
Fin x + *Minf* = *Minf* |
Minf + *Minf* = *Minf* |
Minf + *Pinf* = *Pinf* |
Pinf + *Minf* = *Pinf*

case-of-simps *plus-case*: *plus-extended.simps*

instance ..

end

instance *extended* :: (*ab-semigroup-add*)*ab-semigroup-add*
by *intro-classes* (*simp-all add*: *ac-simps plus-case split*: *extended.splits*)

instance *extended* :: (*ordered-ab-semigroup-add*)*ordered-ab-semigroup-add*
by *intro-classes* (*auto simp*: *add-left-mono plus-case split*: *extended.splits*)

instance *extended* :: (*comm-monoid-add*)*comm-monoid-add*
proof
fix x :: *'a extended* **show** $0 + x = x$ **unfolding** *zero-extended-def* **by**(*cases x*)*auto*
qed

instantiation *extended* :: (*uminus*)*uminus*
begin

fun *uminus-extended* **where**
$-$ (*Fin x*) = *Fin* ($-$ *x*) |
$-$ *Pinf* = *Minf* |
$-$ *Minf* = *Pinf*

instance ..

end

instantiation *extended* :: (*ab-group-add*)*minus*
begin
definition $x - y = x + -(y::'a\ extended)$
instance ..
end

lemma *minus-extended-simps*[*simp*]:
Fin x $-$ *Fin y* = *Fin*($x - y$)
Fin x $-$ *Pinf* = *Minf*
Fin x $-$ *Minf* = *Pinf*
Pinf $-$ *Fin y* = *Pinf*
Pinf $-$ *Minf* = *Pinf*
Minf $-$ *Fin y* = *Minf*
Minf $-$ *Pinf* = *Minf*
Minf $-$ *Minf* = *Pinf*
Pinf $-$ *Pinf* = *Pinf*
by (*simp-all add*: *minus-extended-def*)

Numerals:

instance *extended* :: ({*ab-semigroup-add*,*one*})*numeral* **..**

lemma *Fin-numeral*[*code-post*]: *Fin*(*numeral w*) = *numeral w*

apply (*induct w rule*: *num-induct*)
apply (*simp only*: *numeral-One one-extended-def*)
apply (*simp only*: *numeral-inc one-extended-def plus-extended.simps*(*1*)[*symmetric*])
done

lemma *Fin-neg-numeral*[*code-post*]: *Fin* (− *numeral w*) = − *numeral w*
by (*simp only*: *Fin-numeral uminus-extended.simps*[*symmetric*])

instantiation *extended* :: (*lattice*)*bounded-lattice*
begin

definition *bot* = *Minf*
definition *top* = *Pinf*

fun *inf-extended* :: *'a extended* ⇒ *'a extended* ⇒ *'a extended* **where**
inf-extended (*Fin i*) (*Fin j*) = *Fin* (*inf i j*) |
inf-extended a Minf = *Minf* |
inf-extended Minf a = *Minf* |
inf-extended Pinf a = *a* |
inf-extended a Pinf = *a*

fun *sup-extended* :: *'a extended* ⇒ *'a extended* ⇒ *'a extended* **where**
sup-extended (*Fin i*) (*Fin j*) = *Fin* (*sup i j*) |
sup-extended a Pinf = *Pinf* |
sup-extended Pinf a = *Pinf* |
sup-extended Minf a = *a* |
sup-extended a Minf = *a*

case-of-simps *inf-extended-case*: *inf-extended.simps*
case-of-simps *sup-extended-case*: *sup-extended.simps*

instance
by (*intro-classes*) (*auto simp*: *inf-extended-case sup-extended-case less-eq-extended-case bot-extended-def top-extended-def split*: *extended.splits*)
end

end

30 Continuity and iterations

theory *Order-Continuity*
imports *Complex-Main Countable-Complete-Lattices*
begin

lemma *SUP-nat-binary*:
(*SUP n*::*nat*. *if n* = *0 then A else B*) = (*sup A B*::*'a*::*countable-complete-lattice*)

apply (*auto intro*!: *antisym ccSUP-least*)
apply (*rule ccSUP-upper2*[**where** *i=0*])
apply *simp-all*
apply (*rule ccSUP-upper2*[**where** *i=1*])
apply *simp-all*
done

lemma *INF-nat-binary*:
(*INF n*::*nat*. *if n* = *0 then A else B*) = (*inf A B*::*'a*::*countable-complete-lattice*)
apply (*auto intro*!: *antisym ccINF-greatest*)
apply (*rule ccINF-lower2*[**where** *i=0*])
apply *simp-all*
apply (*rule ccINF-lower2*[**where** *i=1*])
apply *simp-all*
done

The name *continuous* is already taken in *Complex-Main*, so we use *sup-continuous* and *inf-continuous*. These names appear sometimes in literature and have the advantage that these names are duals.

named-theorems *order-continuous-intros*

30.1 Continuity for complete lattices

definition
sup-continuous :: (*'a*::*countable-complete-lattice* ⇒ *'b*::*countable-complete-lattice*) ⇒ *bool*
where
sup-continuous F ⟷ (∀ *M*::*nat* ⇒ *'a*. *mono M* ⟶ *F* (*SUP i*. *M i*) = (*SUP i*. *F* (*M i*)))

lemma *sup-continuousD*: *sup-continuous F* ⟹ *mono M* ⟹ *F* (*SUP i*::*nat*. *M i*) = (*SUP i*. *F* (*M i*))
by (*auto simp*: *sup-continuous-def*)

lemma *sup-continuous-mono*:
assumes [*simp*]: *sup-continuous F* **shows** *mono F*
proof
fix *A B* :: *'a* **assume** [*simp*]: *A* ≤ *B*
have *F B* = *F* (*SUP n*::*nat*. *if n* = *0 then A else B*)
by (*simp add*: *sup-absorb2 SUP-nat-binary*)
also have . . . = (*SUP n*::*nat*. *if n* = *0 then F A else F B*)
by (*auto simp*: *sup-continuousD mono-def intro*!: *SUP-cong*)
finally show *F A* ≤ *F B*
by (*simp add*: *SUP-nat-binary le-iff-sup*)
qed

lemma [*order-continuous-intros*]:
shows *sup-continuous-const*: *sup-continuous* (λ*x*. *c*)
and *sup-continuous-id*: *sup-continuous* (λ*x*. *x*)

and *sup-continuous-apply*: *sup-continuous* $(\lambda f.\ f\ x)$
and *sup-continuous-fun*: $(\bigwedge s.$ *sup-continuous* $(\lambda x.\ P\ x\ s)) \Longrightarrow$ *sup-continuous* P
and *sup-continuous-If*: *sup-continuous* $F \Longrightarrow$ *sup-continuous* $G \Longrightarrow$ *sup-continuous* $(\lambda f.$ *if* C *then* $F\ f$ *else* $G\ f)$
by (*auto simp*: *sup-continuous-def*)

lemma *sup-continuous-compose*:
assumes *f*: *sup-continuous f* **and** *g*: *sup-continuous g*
shows *sup-continuous* $(\lambda x.\ f\ (g\ x))$
unfolding *sup-continuous-def*
proof *safe*
fix $M :: nat \Rightarrow 'c$ **assume** *mono M*
moreover then have *mono* $(\lambda i.\ g\ (M\ i))$
using *sup-continuous-mono*[*OF g*] **by** (*auto simp*: *mono-def*)
ultimately show $f\ (g\ (SUPREMUM\ UNIV\ M)) = (SUP\ i.\ f\ (g\ (M\ i)))$
by (*auto simp*: *sup-continuous-def g*[*THEN sup-continuousD*] *f*[*THEN sup-continuousD*])
qed

lemma *sup-continuous-sup*[*order-continuous-intros*]:
sup-continuous f $\Longrightarrow$ *sup-continuous g* $\Longrightarrow$ *sup-continuous* $(\lambda x.\ sup\ (f\ x)\ (g\ x))$
by (*simp add*: *sup-continuous-def ccSUP-sup-distrib*)

lemma *sup-continuous-inf*[*order-continuous-intros*]:
fixes $P\ Q :: 'a ::$ *countable-complete-lattice* $\Rightarrow 'b ::$ *countable-complete-distrib-lattice*
assumes *P*: *sup-continuous P* **and** *Q*: *sup-continuous Q*
shows *sup-continuous* $(\lambda x.\ inf\ (P\ x)\ (Q\ x))$
unfolding *sup-continuous-def*
proof (*safe intro*!: *antisym*)
fix $M :: nat \Rightarrow 'a$ **assume** *M*: *incseq M*
have $inf\ (P\ (SUP\ i.\ M\ i))\ (Q\ (SUP\ i.\ M\ i)) \leq (SUP\ j\ i.\ inf\ (P\ (M\ i))\ (Q\ (M\ j)))$
by (*simp add*: *sup-continuousD*[*OF P M*] *sup-continuousD*[*OF Q M*] *inf-ccSUP ccSUP-inf*)
also have $\ldots \leq (SUP\ i.\ inf\ (P\ (M\ i))\ (Q\ (M\ i)))$
proof (*intro ccSUP-least*)
fix *i j* **from** *M assms*[*THEN sup-continuous-mono*] **show** $inf\ (P\ (M\ i))\ (Q\ (M\ j)) \leq (SUP\ i.\ inf\ (P\ (M\ i))\ (Q\ (M\ i)))$
by (*intro ccSUP-upper2*[*of - sup i j*] *inf-mono*) (*auto simp*: *mono-def*)
qed *auto*
finally show $inf\ (P\ (SUP\ i.\ M\ i))\ (Q\ (SUP\ i.\ M\ i)) \leq (SUP\ i.\ inf\ (P\ (M\ i))\ (Q\ (M\ i)))$ **.**

show $(SUP\ i.\ inf\ (P\ (M\ i))\ (Q\ (M\ i))) \leq inf\ (P\ (SUP\ i.\ M\ i))\ (Q\ (SUP\ i.\ M\ i))$
unfolding *sup-continuousD*[*OF P M*] *sup-continuousD*[*OF Q M*] **by** (*intro ccSUP-least inf-mono ccSUP-upper*) *auto*
qed

lemma *sup-continuous-and*[*order-continuous-intros*]:
sup-continuous P ⟹ *sup-continuous Q* ⟹ *sup-continuous* (λ*x*. *P x* ∧ *Q x*)
using *sup-continuous-inf*[*of P Q*] **by** *simp*

lemma *sup-continuous-or*[*order-continuous-intros*]:
sup-continuous P ⟹ *sup-continuous Q* ⟹ *sup-continuous* (λ*x*. *P x* ∨ *Q x*)
by (*auto simp*: *sup-continuous-def*)

lemma *sup-continuous-lfp*:
assumes *sup-continuous F* **shows** *lfp F* = (*SUP i*. (*F* ^^ *i*) *bot*) (**is** *lfp F* = *?U*)
proof (*rule antisym*)
note *mono* = *sup-continuous-mono*[*OF* ‹*sup-continuous F*›]
show *?U* ≤ *lfp F*
proof (*rule SUP-least*)
fix *i* **show** (*F* ^^ *i*) *bot* ≤ *lfp F*
proof (*induct i*)
case (*Suc i*)
have (*F* ^^ *Suc i*) *bot* = *F* ((*F* ^^ *i*) *bot*) **by** *simp*
also have ... ≤ *F* (*lfp F*) **by** (*rule monoD*[*OF mono Suc*])
also have ... = *lfp F* **by** (*simp add*: *lfp-unfold*[*OF mono*, *symmetric*])
finally show *?case* **.**
qed *simp*
qed
show *lfp F* ≤ *?U*
proof (*rule lfp-lowerbound*)
have *mono* (λ*i*::*nat*. (*F* ^^ *i*) *bot*)
proof –
{ fix *i*::*nat* **have** (*F* ^^ *i*) *bot* ≤ (*F* ^^ (*Suc i*)) *bot*
proof (*induct i*)
case *0* **show** *?case* **by** *simp*
next
case *Suc* **thus** *?case* **using** *monoD*[*OF mono Suc*] **by** *auto*
qed }
thus *?thesis* **by** (*auto simp add*: *mono-iff-le-Suc*)
qed
hence *F ?U* = (*SUP i*. (*F* ^^ *Suc i*) *bot*)
using ‹*sup-continuous F*› **by** (*simp add*: *sup-continuous-def*)
also have ... ≤ *?U*
by (*fast intro*: *SUP-least SUP-upper*)
finally show *F ?U* ≤ *?U* **.**
qed
qed

lemma *lfp-transfer-bounded*:
assumes *P*: *P bot* ⋀*x*. *P x* ⟹ *P* (*f x*) ⋀*M*. (⋀*i*. *P* (*M i*)) ⟹ *P* (*SUP i*::*nat*. *M i*)
assumes α: ⋀*M*. *mono M* ⟹ (⋀*i*::*nat*. *P* (*M i*)) ⟹ α (*SUP i*. *M i*) = (*SUP i*. α (*M i*))
assumes *f*: *sup-continuous f* **and** *g*: *sup-continuous g*

assumes [*simp*]: $\bigwedge x.\ P\ x \Longrightarrow x \leq \textit{lfp}\ f \Longrightarrow \alpha\ (f\ x) = g\ (\alpha\ x)$
assumes *g-bound*: $\bigwedge x.\ \alpha\ \textit{bot} \leq g\ x$
shows $\alpha\ (\textit{lfp}\ f) = \textit{lfp}\ g$
proof (*rule antisym*)
note *mono-g* = *sup-continuous-mono*[*OF g*]
note *mono-f* = *sup-continuous-mono*[*OF f*]
have *lfp-bound*: $\alpha\ \textit{bot} \leq \textit{lfp}\ g$
by (*subst lfp-unfold*[*OF mono-g*]) (*rule g-bound*)

have *P-pow*: *P* ((*f* ^^ *i*) *bot*) **for** *i*
by (*induction i*) (*auto intro*!: *P*)
have *incseq-pow*: *mono* (λ*i*. (*f* ^^ *i*) *bot*)
unfolding *mono-iff-le-Suc*
proof
fix *i* **show** (*f* ^^ *i*) *bot* ≤ (*f* ^^ (*Suc i*)) *bot*
proof (*induct i*)
case *Suc* **thus** *?case* **using** *monoD*[*OF sup-continuous-mono*[*OF f*] *Suc*] **by** *auto*
qed (*simp add*: *le-fun-def*)
qed
have *P-lfp*: *P* (*lfp f*)
using *P-pow* **unfolding** *sup-continuous-lfp*[*OF f*] **by** (*auto intro*!: *P*)

have *iter-le-lfp*: (*f* ^^ *n*) *bot* ≤ *lfp f* **for** *n*
apply (*induction n*)
apply *simp*
apply (*subst lfp-unfold*[*OF mono-f*])
apply (*auto intro*!: *monoD*[*OF mono-f*])
done

have α (*lfp f*) = (*SUP i*. α ((*f*^^*i*) *bot*))
unfolding *sup-continuous-lfp*[*OF f*] **using** *incseq-pow P-pow* **by** (*rule* α)
also have ... ≤ *lfp g*
proof (*rule SUP-least*)
fix *i* **show** α ((*f*^^*i*) *bot*) ≤ *lfp g*
proof (*induction i*)
case (*Suc n*) **then show** *?case*
by (*subst lfp-unfold*[*OF mono-g*]) (*simp add*: *monoD*[*OF mono-g*] *P-pow iter-le-lfp*)
qed (*simp add*: *lfp-bound*)
qed
finally show α (*lfp f*) ≤ *lfp g* **.**

show *lfp g* ≤ α (*lfp f*)
proof (*induction rule*: *lfp-ordinal-induct*[*OF mono-g*])
case (*1 S*) **then show** *?case*
by (*subst lfp-unfold*[*OF sup-continuous-mono*[*OF f*]])
(*simp add*: *monoD*[*OF mono-g*] *P-lfp*)
qed (*auto intro*: *Sup-least*)

qed

lemma *lfp-transfer*:
sup-continuous $\alpha \Longrightarrow$ *sup-continuous* $f \Longrightarrow$ *sup-continuous* $g \Longrightarrow$
$(\bigwedge x.\ \alpha\ bot \leq g\ x) \Longrightarrow (\bigwedge x.\ x \leq lfp\ f \Longrightarrow \alpha\ (f\ x) = g\ (\alpha\ x)) \Longrightarrow \alpha\ (lfp\ f) = lfp\ g$
by (*rule lfp-transfer-bounded*[**where** *P*=*top*]) (*auto dest*: *sup-continuousD*)

definition
inf-continuous :: ('*a*::*countable-complete-lattice* $\Rightarrow$ '*b*::*countable-complete-lattice*) $\Rightarrow$ *bool*
where
inf-continuous F $\longleftrightarrow$ ($\forall$ *M*::*nat* $\Rightarrow$ '*a*. *antimono M* $\longrightarrow$ *F* (*INF i*. *M i*) = (*INF i*. *F* (*M i*)))

lemma *inf-continuousD*: *inf-continuous F* $\Longrightarrow$ *antimono M* $\Longrightarrow$ *F* (*INF i*::*nat*. *M i*) = (*INF i*. *F* (*M i*))
by (*auto simp*: *inf-continuous-def*)

lemma *inf-continuous-mono*:
assumes [*simp*]: *inf-continuous F* **shows** *mono F*
proof
fix *A B* :: '*a* **assume** [*simp*]: $A \leq B$
have *F A* = *F* (*INF n*::*nat*. *if n* = *0 then B else A*)
by (*simp add*: *inf-absorb2 INF-nat-binary*)
also have ... = (*INF n*::*nat*. *if n* = *0 then F B else F A*)
by (*auto simp*: *inf-continuousD antimono-def intro*!: *INF-cong*)
finally show $F\ A \leq F\ B$
by (*simp add*: *INF-nat-binary le-iff-inf inf-commute*)
qed

lemma [*order-continuous-intros*]:
shows *inf-continuous-const*: *inf-continuous* ($\lambda x.\ c$)
and *inf-continuous-id*: *inf-continuous* ($\lambda x.\ x$)
and *inf-continuous-apply*: *inf-continuous* ($\lambda f.\ f\ x$)
and *inf-continuous-fun*: ($\bigwedge s.$ *inf-continuous* ($\lambda x.\ P\ x\ s$)) $\Longrightarrow$ *inf-continuous P*
and *inf-continuous-If*: *inf-continuous F* $\Longrightarrow$ *inf-continuous G* $\Longrightarrow$ *inf-continuous* ($\lambda f.$ *if C then F f else G f*)
by (*auto simp*: *inf-continuous-def*)

lemma *inf-continuous-inf*[*order-continuous-intros*]:
inf-continuous f $\Longrightarrow$ *inf-continuous g* $\Longrightarrow$ *inf-continuous* ($\lambda x.$ *inf* (*f x*) (*g x*))
by (*simp add*: *inf-continuous-def ccINF-inf-distrib*)

lemma *inf-continuous-sup*[*order-continuous-intros*]:
fixes *P Q* :: '*a* :: *countable-complete-lattice* $\Rightarrow$ '*b* :: *countable-complete-distrib-lattice*
assumes *P*: *inf-continuous P* **and** *Q*: *inf-continuous Q*
shows *inf-continuous* ($\lambda x.$ *sup* (*P x*) (*Q x*))
unfolding *inf-continuous-def*

proof (*safe intro*!: *antisym*)
fix *M* :: *nat* ⇒ *'a* **assume** *M*: *decseq M*
show *sup* (*P* (*INF i. M i*)) (*Q* (*INF i. M i*)) ≤ (*INF i. sup* (*P* (*M i*)) (*Q* (*M i*)))
unfolding *inf-continuousD*[*OF P M*] *inf-continuousD*[*OF Q M*] **by** (*intro ccINF-greatest sup-mono ccINF-lower*) *auto*

have (*INF i. sup* (*P* (*M i*)) (*Q* (*M i*))) ≤ (*INF j i. sup* (*P* (*M i*)) (*Q* (*M j*)))
proof (*intro ccINF-greatest*)
fix *i j* **from** *M assms*[*THEN inf-continuous-mono*] **show** *sup* (*P* (*M i*)) (*Q* (*M j*)) ≥ (*INF i. sup* (*P* (*M i*)) (*Q* (*M i*)))
by (*intro ccINF-lower2*[*of - sup i j*] *sup-mono*) (*auto simp*: *mono-def antimono-def*)
qed *auto*
also have ... ≤ *sup* (*P* (*INF i. M i*)) (*Q* (*INF i. M i*))
by (*simp add*: *inf-continuousD*[*OF P M*] *inf-continuousD*[*OF Q M*] *ccINF-sup sup-ccINF*)
finally show *sup* (*P* (*INF i. M i*)) (*Q* (*INF i. M i*)) ≥ (*INF i. sup* (*P* (*M i*)) (*Q* (*M i*))) **.**
qed

lemma *inf-continuous-and*[*order-continuous-intros*]:
inf-continuous P ⟹ *inf-continuous Q* ⟹ *inf-continuous* (*λx. P x* ∧ *Q x*)
using *inf-continuous-inf*[*of P Q*] **by** *simp*

lemma *inf-continuous-or*[*order-continuous-intros*]:
inf-continuous P ⟹ *inf-continuous Q* ⟹ *inf-continuous* (*λx. P x* ∨ *Q x*)
using *inf-continuous-sup*[*of P Q*] **by** *simp*

lemma *inf-continuous-compose*:
assumes *f*: *inf-continuous f* **and** *g*: *inf-continuous g*
shows *inf-continuous* (*λx. f* (*g x*))
unfolding *inf-continuous-def*
proof *safe*
fix *M* :: *nat* ⇒ *'c* **assume** *antimono M*
moreover then have *antimono* (*λi. g* (*M i*))
using *inf-continuous-mono*[*OF g*] **by** (*auto simp*: *mono-def antimono-def*)
ultimately show *f* (*g* (*INFIMUM UNIV M*)) = (*INF i. f* (*g* (*M i*)))
by (*auto simp*: *inf-continuous-def g*[*THEN inf-continuousD*] *f*[*THEN inf-continuousD*])
qed

lemma *inf-continuous-gfp*:
assumes *inf-continuous F* **shows** *gfp F* = (*INF i.* (*F ^^ i*) *top*) (**is** *gfp F* = *?U*)
proof (*rule antisym*)
note *mono* = *inf-continuous-mono*[*OF* ‹*inf-continuous F*›]
show *gfp F* ≤ *?U*
proof (*rule INF-greatest*)
fix *i* **show** *gfp F* ≤ (*F ^^ i*) *top*
proof (*induct i*)
case (*Suc i*)

have *gfp F* = *F* (*gfp F*) **by** (*simp add*: *gfp-unfold*[*OF mono*, *symmetric*])
also have ... ≤ *F* ((*F* ^^ *i*) *top*) **by** (*rule monoD*[*OF mono Suc*])
also have ... = (*F* ^^ *Suc i*) *top* **by** *simp*
finally show *?case* **.**
qed *simp*
qed
show *?U* ≤ *gfp F*
proof (*rule gfp-upperbound*)
have ∗: *antimono* (λ*i*::*nat*. (*F* ^^ *i*) *top*)
proof −
{ fix *i*::*nat* **have** (*F* ^^ *Suc i*) *top* ≤ (*F* ^^ *i*) *top*
proof (*induct i*)
case *0* **show** *?case* **by** *simp*
next
case *Suc* **thus** *?case* **using** *monoD*[*OF mono Suc*] **by** *auto*
qed }
thus *?thesis* **by** (*auto simp add*: *antimono-iff-le-Suc*)
qed
have *?U* ≤ (*INF i*. (*F* ^^ *Suc i*) *top*)
by (*fast intro*: *INF-greatest INF-lower*)
also have ... ≤ *F ?U*
by (*simp add*: *inf-continuousD* ‹*inf-continuous F*› ∗)
finally show *?U* ≤ *F ?U* **.**
qed
qed

lemma *gfp-transfer*:
assumes α: *inf-continuous* α **and** *f*: *inf-continuous f* **and** *g*: *inf-continuous g*
assumes [*simp*]: α *top* = *top* ⋀*x*. α (*f x*) = *g* (α *x*)
shows α (*gfp f*) = *gfp g*
proof −
have α (*gfp f*) = (*INF i*. α ((*f*^^*i*) *top*))
unfolding *inf-continuous-gfp*[*OF f*] **by** (*intro f* α *inf-continuousD antimono-funpow inf-continuous-mono*)
moreover have α ((*f*^^*i*) *top*) = (*g*^^*i*) *top* **for** *i*
by (*induction i*; *simp*)
ultimately show *?thesis*
unfolding *inf-continuous-gfp*[*OF g*] **by** *simp*
qed

lemma *gfp-transfer-bounded*:
assumes *P*: *P* (*f top*) ⋀*x*. *P x* ⟹ *P* (*f x*) ⋀*M*. *antimono M* ⟹ (⋀*i*. *P* (*M i*)) ⟹ *P* (*INF i*::*nat*. *M i*)
assumes α: ⋀*M*. *antimono M* ⟹ (⋀*i*::*nat*. *P* (*M i*)) ⟹ α (*INF i*. *M i*) = (*INF i*. α (*M i*))
assumes *f*: *inf-continuous f* **and** *g*: *inf-continuous g*
assumes [*simp*]: ⋀*x*. *P x* ⟹ α (*f x*) = *g* (α *x*)
assumes *g-bound*: ⋀*x*. *g x* ≤ α (*f top*)
shows α (*gfp f*) = *gfp g*

proof (*rule antisym*)
note *mono-g* = *inf-continuous-mono*[*OF g*]

have *P-pow*: *P* ((*f* ^^ *i*) (*f top*)) **for** *i*
by (*induction i*) (*auto intro*!: *P*)

have *antimono-pow*: *antimono* (λ*i*. (*f* ^^ *i*) *top*)
unfolding *antimono-iff-le-Suc*
proof
fix *i* **show** (*f* ^^ *Suc i*) *top* ≤ (*f* ^^ *i*) *top*
proof (*induct i*)
case *Suc* **thus** *?case* **using** *monoD*[*OF inf-continuous-mono*[*OF f*] *Suc*] **by** *auto*
qed (*simp add*: *le-fun-def*)
qed
have *antimono-pow2*: *antimono* (λ*i*. (*f* ^^ *i*) (*f top*))
proof
show *x* ≤ *y* ⟹ (*f* ^^ *y*) (*f top*) ≤ (*f* ^^ *x*) (*f top*) **for** *x y*
using *antimono-pow*[*THEN antimonoD*, *of Suc x Suc y*]
unfolding *funpow-Suc-right* **by** *simp*
qed

have *gfp-f*: *gfp f* = (*INF i*. (*f* ^^ *i*) (*f top*))
unfolding *inf-continuous-gfp*[*OF f*]
proof (*rule INF-eq*)
show ∃ *j*∈*UNIV*. (*f* ^^ *j*) (*f top*) ≤ (*f* ^^ *i*) *top* **for** *i*
by (*intro bexI*[*of* - *i* − *1*]) (*auto simp*: *diff-Suc funpow-Suc-right simp del*: *funpow.simps*(*2*) *split*: *nat.split*)
show ∃ *j*∈*UNIV*. (*f* ^^ *j*) *top* ≤ (*f* ^^ *i*) (*f top*) **for** *i*
by (*intro bexI*[*of* - *Suc i*]) (*auto simp*: *funpow-Suc-right simp del*: *funpow.simps*(*2*))
qed

have *P-lfp*: *P* (*gfp f*)
unfolding *gfp-f* **by** (*auto intro*!: *P P-pow antimono-pow2*)

have α (*gfp f*) = (*INF i*. α ((*f*^^*i*) (*f top*)))
unfolding *gfp-f* **by** (*rule* α) (*auto intro*!: *P-pow antimono-pow2*)
also have ... ≥ *gfp g*
proof (*rule INF-greatest*)
fix *i* **show** *gfp g* ≤ α ((*f*^^*i*) (*f top*))
proof (*induction i*)
case (*Suc n*) **then show** *?case*
by (*subst gfp-unfold*[*OF mono-g*]) (*simp add*: *monoD*[*OF mono-g*] *P-pow*)
next
case *0*
have *gfp g* ≤ α (*f top*)
by (*subst gfp-unfold*[*OF mono-g*]) (*rule g-bound*)
then show *?case*

```
      by simp
    qed
  qed
  finally show gfp g ≤ α (gfp f) .

  show α (gfp f) ≤ gfp g
  proof (induction rule: gfp-ordinal-induct[OF mono-g])
    case (1 S) then show ?case
      by (subst gfp-unfold[OF inf-continuous-mono[OF f]])
         (simp add: monoD[OF mono-g] P-lfp)
  qed (auto intro: Inf-greatest)
qed
```

30.1.1 Least fixed points in countable complete lattices

```
definition (in countable-complete-lattice) cclfp :: ('a ⇒ 'a) ⇒ 'a
  where cclfp f = (SUP i. (f ^^ i) bot)

lemma cclfp-unfold:
  assumes sup-continuous F shows cclfp F = F (cclfp F)
proof -
  have cclfp F = (SUP i. F ((F ^^ i) bot))
    unfolding cclfp-def by (subst UNIV-nat-eq) auto
  also have ... = F (cclfp F)
    unfolding cclfp-def
   by (intro sup-continuousD[symmetric] assms mono-funpow sup-continuous-mono)
  finally show ?thesis .
qed

lemma cclfp-lowerbound: assumes f: mono f and A: f A ≤ A shows cclfp f ≤
A
  unfolding cclfp-def
proof (intro ccSUP-least)
  fix i show (f ^^ i) bot ≤ A
  proof (induction i)
    case (Suc i) from monoD[OF f this] A show ?case
      by auto
  qed simp
qed simp

lemma cclfp-transfer:
  assumes sup-continuous α mono f
  assumes α bot = bot ⋀x. α (f x) = g (α x)
  shows α (cclfp f) = cclfp g
proof -
  have α (cclfp f) = (SUP i. α ((f ^^ i) bot))
   unfolding cclfp-def by (intro sup-continuousD assms mono-funpow sup-continuous-mono)
  moreover have α ((f ^^ i) bot) = (g ^^ i) bot for i
    by (induction i) (simp-all add: assms)
```

ultimately show *?thesis*
by (*simp add*: *cclfp-def*)
qed

end

31 Extended natural numbers (i.e. with infinity)

theory *Extended-Nat*
imports *Main Countable Order-Continuity*
begin

class *infinity* =
fixes *infinity* :: *'a* (∞)

context
fixes *f* :: *nat* ⇒ *'a*::{*canonically-ordered-monoid-add*, *linorder-topology*, *complete-linorder*}
begin

lemma *sums-SUP*[*simp*, *intro*]: *f sums* (*SUP n*. $\sum$ *i*<*n*. *f i*)
unfolding *sums-def* **by** (*intro LIMSEQ-SUP monoI setsum-mono2 zero-le*) *auto*

lemma *suminf-eq-SUP*: *suminf f* = (*SUP n*. $\sum$ *i*<*n*. *f i*)
using *sums-SUP* **by** (*rule sums-unique*[*symmetric*])

end

31.1 Type definition

We extend the standard natural numbers by a special value indicating infinity.

typedef *enat* = *UNIV* :: *nat option set* **..**

TODO: introduce enat as coinductive datatype, enat is just *of-nat*

definition *enat* :: *nat* ⇒ *enat* **where**
enat n = *Abs-enat* (*Some n*)

instantiation *enat* :: *infinity*
begin

definition ∞ = *Abs-enat None*
instance ..

end

instance *enat* :: *countable*
proof
show ∃ *to-nat*::*enat* ⇒ *nat*. *inj to-nat*

by (*rule exI*[*of* - *to-nat* ∘ *Rep-enat*]) (*simp add*: *inj-on-def Rep-enat-inject*)
qed

old-rep-datatype *enat* ∞ :: *enat*
proof −
fix *P i* **assume** $\bigwedge j.\ P\ (enat\ j)\ P\ \infty$
then show *P i*
proof *induct*
case (*Abs-enat y*) **then show** *?case*
by (*cases y rule*: *option.exhaust*)
(*auto simp*: *enat-def infinity-enat-def*)
qed
qed (*auto simp add*: *enat-def infinity-enat-def Abs-enat-inject*)

declare [[*coercion enat*::*nat*⇒*enat*]]

lemmas *enat2-cases* = *enat.exhaust*[*case-product enat.exhaust*]
lemmas *enat3-cases* = *enat.exhaust*[*case-product enat.exhaust enat.exhaust*]

lemma *not-infinity-eq* [*iff*]: $(x \neq \infty) = (\exists\, i.\ x = enat\ i)$
by (*cases x*) *auto*

lemma *not-enat-eq* [*iff*]: $(\forall\, y.\ x \neq enat\ y) = (x = \infty)$
by (*cases x*) *auto*

lemma *enat-ex-split*: $(\exists\, c{::}enat.\ P\ c) \longleftrightarrow P\ \infty \vee (\exists\, c{::}nat.\ P\ c)$
by (*metis enat.exhaust*)

primrec *the-enat* :: *enat* ⇒ *nat*
where *the-enat* (*enat n*) = *n*

31.2 Constructors and numbers

instantiation *enat* :: *zero-neq-one*
begin

definition
0 = *enat 0*

definition
1 = *enat 1*

instance
proof qed (*simp add*: *zero-enat-def one-enat-def*)

end

definition *eSuc* :: *enat* ⇒ *enat* **where**
eSuc i = (*case i of enat n* ⇒ *enat* (*Suc n*) | ∞ ⇒ ∞)

lemma *enat-0* [*code-post*]: *enat 0* = *0*
by (*simp add*: *zero-enat-def*)

lemma *enat-1* [*code-post*]: *enat 1* = *1*
by (*simp add*: *one-enat-def*)

lemma *enat-0-iff*: *enat x* = *0* ⟷ *x* = *0 0* = *enat x* ⟷ *x* = *0*
by (*auto simp add*: *zero-enat-def*)

lemma *enat-1-iff*: *enat x* = *1* ⟷ *x* = *1 1* = *enat x* ⟷ *x* = *1*
by (*auto simp add*: *one-enat-def*)

lemma *one-eSuc*: *1* = *eSuc 0*
by (*simp add*: *zero-enat-def one-enat-def eSuc-def*)

lemma *infinity-ne-i0* [*simp*]: (∞::*enat*) ≠ *0*
by (*simp add*: *zero-enat-def*)

lemma *i0-ne-infinity* [*simp*]: *0* ≠ (∞::*enat*)
by (*simp add*: *zero-enat-def*)

lemma *zero-one-enat-neq*:
¬ *0* = (*1*::*enat*)
¬ *1* = (*0*::*enat*)
unfolding *zero-enat-def one-enat-def* **by** *simp-all*

lemma *infinity-ne-i1* [*simp*]: (∞::*enat*) ≠ *1*
by (*simp add*: *one-enat-def*)

lemma *i1-ne-infinity* [*simp*]: *1* ≠ (∞::*enat*)
by (*simp add*: *one-enat-def*)

lemma *eSuc-enat*: *eSuc* (*enat n*) = *enat* (*Suc n*)
by (*simp add*: *eSuc-def*)

lemma *eSuc-infinity* [*simp*]: *eSuc* ∞ = ∞
by (*simp add*: *eSuc-def*)

lemma *eSuc-ne-0* [*simp*]: *eSuc n* ≠ *0*
by (*simp add*: *eSuc-def zero-enat-def split*: *enat.splits*)

lemma *zero-ne-eSuc* [*simp*]: *0* ≠ *eSuc n*
by (*rule eSuc-ne-0* [*symmetric*])

lemma *eSuc-inject* [*simp*]: *eSuc m* = *eSuc n* ⟷ *m* = *n*
by (*simp add*: *eSuc-def split*: *enat.splits*)

lemma *eSuc-enat-iff*: *eSuc x* = *enat y* ⟷ (∃ *n*. *y* = *Suc n* ∧ *x* = *enat n*)

by (*cases y*) (*auto simp*: *enat-0 eSuc-enat*[*symmetric*])

lemma *enat-eSuc-iff*: *enat y* = *eSuc x* ⟷ (∃ *n*. *y* = *Suc n* ∧ *enat n* = *x*)
by (*cases y*) (*auto simp*: *enat-0 eSuc-enat*[*symmetric*])

31.3 Addition

instantiation *enat* :: *comm-monoid-add*
begin

definition [*nitpick-simp*]:
m + *n* = (*case m of* ∞ ⇒ ∞ | *enat m* ⇒ (*case n of* ∞ ⇒ ∞ | *enat n* ⇒ *enat* (*m* + *n*)))

lemma *plus-enat-simps* [*simp*, *code*]:
fixes *q* :: *enat*
shows *enat m* + *enat n* = *enat* (*m* + *n*)
and ∞ + *q* = ∞
and *q* + ∞ = ∞
by (*simp-all add*: *plus-enat-def split*: *enat.splits*)

instance
proof
fix *n m q* :: *enat*
show *n* + *m* + *q* = *n* + (*m* + *q*)
by (*cases n m q rule*: *enat3-cases*) *auto*
show *n* + *m* = *m* + *n*
by (*cases n m rule*: *enat2-cases*) *auto*
show *0* + *n* = *n*
by (*cases n*) (*simp-all add*: *zero-enat-def*)
qed

end

lemma *eSuc-plus-1*:
eSuc n = *n* + *1*
by (*cases n*) (*simp-all add*: *eSuc-enat one-enat-def*)

lemma *plus-1-eSuc*:
1 + *q* = *eSuc q*
q + *1* = *eSuc q*
by (*simp-all add*: *eSuc-plus-1 ac-simps*)

lemma *iadd-Suc*: *eSuc m* + *n* = *eSuc* (*m* + *n*)
by (*simp-all add*: *eSuc-plus-1 ac-simps*)

lemma *iadd-Suc-right*: *m* + *eSuc n* = *eSuc* (*m* + *n*)
by (*simp only*: *add.commute*[*of m*] *iadd-Suc*)

31.4 Multiplication

instantiation *enat* :: {*comm-semiring-1*, *semiring-no-zero-divisors*}
begin

definition *times-enat-def* [*nitpick-simp*]:
  $m * n = (case\ m\ of\ \infty \Rightarrow if\ n = 0\ then\ 0\ else\ \infty \mid enat\ m \Rightarrow$
    $(case\ n\ of\ \infty \Rightarrow if\ m = 0\ then\ 0\ else\ \infty \mid enat\ n \Rightarrow enat\ (m * n)))$

lemma *times-enat-simps* [*simp*, *code*]:
  $enat\ m * enat\ n = enat\ (m * n)$
  $\infty * \infty = (\infty::enat)$
  $\infty * enat\ n = (if\ n = 0\ then\ 0\ else\ \infty)$
  $enat\ m * \infty = (if\ m = 0\ then\ 0\ else\ \infty)$
  unfolding *times-enat-def zero-enat-def*
  by (*simp-all split*: *enat.split*)

instance
proof
  fix *a b c* :: *enat*
  show $(a * b) * c = a * (b * c)$
    unfolding *times-enat-def zero-enat-def*
    by (*simp split*: *enat.split*)
  show *comm*: $a * b = b * a$
    unfolding *times-enat-def zero-enat-def*
    by (*simp split*: *enat.split*)
  show $1 * a = a$
    unfolding *times-enat-def zero-enat-def one-enat-def*
    by (*simp split*: *enat.split*)
  show *distr*: $(a + b) * c = a * c + b * c$
    unfolding *times-enat-def zero-enat-def*
    by (*simp split*: *enat.split add*: *distrib-right*)
  show $0 * a = 0$
    unfolding *times-enat-def zero-enat-def*
    by (*simp split*: *enat.split*)
  show $a * 0 = 0$
    unfolding *times-enat-def zero-enat-def*
    by (*simp split*: *enat.split*)
  show $a * (b + c) = a * b + a * c$
    by (*cases a b c rule*: *enat3-cases*) (*auto simp*: *times-enat-def zero-enat-def distrib-left*)
  show $a \neq 0 \Longrightarrow b \neq 0 \Longrightarrow a * b \neq 0$
    by (*cases a b rule*: *enat2-cases*) (*auto simp*: *times-enat-def zero-enat-def*)
qed

end

lemma *mult-eSuc*: $eSuc\ m * n = n + m * n$
  unfolding *eSuc-plus-1* **by** (*simp add*: *algebra-simps*)

lemma *mult-eSuc-right*: $m * eSuc\ n = m + m * n$
unfolding *eSuc-plus-1* **by** (*simp add*: *algebra-simps*)

lemma *of-nat-eq-enat*: *of-nat n* = *enat n*
apply (*induct n*)
apply (*simp add*: *enat-0*)
apply (*simp add*: *plus-1-eSuc eSuc-enat*)
done

instance *enat* :: *semiring-char-0*
proof
have *inj enat* **by** (*rule injI*) *simp*
then show *inj* (λ*n*. *of-nat n* :: *enat*) **by** (*simp add*: *of-nat-eq-enat*)
qed

lemma *imult-is-infinity*: $((a{::}enat) * b = \infty) = (a = \infty \wedge b \neq 0 \vee b = \infty \wedge a \neq 0)$
by (*auto simp add*: *times-enat-def zero-enat-def split*: *enat.split*)

31.5 Numerals

lemma *numeral-eq-enat*:
numeral k = *enat* (*numeral k*)
using *of-nat-eq-enat* [*of numeral k*] **by** *simp*

lemma *enat-numeral* [*code-abbrev*]:
enat (*numeral k*) = *numeral k*
using *numeral-eq-enat* **..**

lemma *infinity-ne-numeral* [*simp*]: (∞::*enat*) ≠ *numeral k*
by (*simp add*: *numeral-eq-enat*)

lemma *numeral-ne-infinity* [*simp*]: *numeral k* ≠ (∞::*enat*)
by (*simp add*: *numeral-eq-enat*)

lemma *eSuc-numeral* [*simp*]: *eSuc* (*numeral k*) = *numeral* (*k* + *Num.One*)
by (*simp only*: *eSuc-plus-1 numeral-plus-one*)

31.6 Subtraction

instantiation *enat* :: *minus*
begin

definition *diff-enat-def*:
$a - b = (case\ a\ of\ (enat\ x) \Rightarrow (case\ b\ of\ (enat\ y) \Rightarrow enat\ (x - y) \mid \infty \Rightarrow 0)$
$\mid \infty \Rightarrow \infty)$

instance ..

end

lemma *idiff-enat-enat* [*simp*, *code*]: *enat a* − *enat b* = *enat* (*a* − *b*)
  by (*simp add*: *diff-enat-def*)

lemma *idiff-infinity* [*simp*, *code*]: ∞ − *n* = (∞::*enat*)
  by (*simp add*: *diff-enat-def*)

lemma *idiff-infinity-right* [*simp*, *code*]: *enat a* − ∞ = *0*
  by (*simp add*: *diff-enat-def*)

lemma *idiff-0* [*simp*]: (*0*::*enat*) − *n* = *0*
  by (*cases n*, *simp-all add*: *zero-enat-def*)

lemmas *idiff-enat-0* [*simp*] = *idiff-0* [*unfolded zero-enat-def*]

lemma *idiff-0-right* [*simp*]: (*n*::*enat*) − *0* = *n*
  by (*cases n*) (*simp-all add*: *zero-enat-def*)

lemmas *idiff-enat-0-right* [*simp*] = *idiff-0-right* [*unfolded zero-enat-def*]

lemma *idiff-self* [*simp*]: *n* ≠ ∞ ⟹ (*n*::*enat*) − *n* = *0*
  by (*auto simp*: *zero-enat-def*)

lemma *eSuc-minus-eSuc* [*simp*]: *eSuc n* − *eSuc m* = *n* − *m*
  by (*simp add*: *eSuc-def split*: *enat.split*)

lemma *eSuc-minus-1* [*simp*]: *eSuc n* − *1* = *n*
  by (*simp add*: *one-enat-def eSuc-enat*[*symmetric*] *zero-enat-def*[*symmetric*])

31.7 Ordering

instantiation *enat* :: *linordered-ab-semigroup-add*
begin

definition [*nitpick-simp*]:
  m ≤ *n* = (*case n of enat n1* ⇒ (*case m of enat m1* ⇒ *m1* ≤ *n1* | ∞ ⇒ *False*)
    | ∞ ⇒ *True*)

definition [*nitpick-simp*]:
  m < *n* = (*case m of enat m1* ⇒ (*case n of enat n1* ⇒ *m1* < *n1* | ∞ ⇒ *True*)
    | ∞ ⇒ *False*)

lemma *enat-ord-simps* [*simp*]:
  enat m ≤ *enat n* ⟷ *m* ≤ *n*
  enat m < *enat n* ⟷ *m* < *n*
  q ≤ (∞::*enat*)
  q < (∞::*enat*) ⟷ *q* ≠ ∞
  (∞::*enat*) ≤ *q* ⟷ *q* = ∞
  (∞::*enat*) < *q* ⟷ *False*

by (*simp-all add*: *less-eq-enat-def less-enat-def split*: *enat.splits*)

lemma *numeral-le-enat-iff* [*simp*]:
shows *numeral m* $\leq$ *enat n* $\longleftrightarrow$ *numeral m* $\leq$ *n*
by (*auto simp*: *numeral-eq-enat*)

lemma *numeral-less-enat-iff* [*simp*]:
shows *numeral m* $<$ *enat n* $\longleftrightarrow$ *numeral m* $<$ *n*
by (*auto simp*: *numeral-eq-enat*)

lemma *enat-ord-code* [*code*]:
enat m $\leq$ *enat n* $\longleftrightarrow$ *m* $\leq$ *n*
enat m $<$ *enat n* $\longleftrightarrow$ *m* $<$ *n*
q $\leq$ (∞::*enat*) $\longleftrightarrow$ *True*
enat m $< \infty \longleftrightarrow$ *True*
$\infty \leq$ *enat n* $\longleftrightarrow$ *False*
(∞::*enat*) $<$ *q* $\longleftrightarrow$ *False*
by *simp-all*

instance
by *standard* (*auto simp add*: *less-eq-enat-def less-enat-def plus-enat-def split*: *enat.splits*)

end

instance *enat* :: *dioid*
proof
fix *a b* :: *enat* **show** ($a \leq b$) = ($\exists\, c.\ b = a + c$)
by (*cases a b rule*: *enat2-cases*) (*auto simp*: *le-iff-add enat-ex-split*)
qed

instance *enat* :: {*linordered-nonzero-semiring*, *strict-ordered-comm-monoid-add*}
proof
fix *a b c* :: *enat*
show $a \leq b \Longrightarrow 0 \leq c \Longrightarrow c * a \leq c * b$
unfolding *times-enat-def less-eq-enat-def zero-enat-def*
by (*simp split*: *enat.splits*)
show $a < b \Longrightarrow c < d \Longrightarrow a + c < b + d$ **for** *a b c d* :: *enat*
by (*cases a b c d rule*: *enat2-cases*[*case-product enat2-cases*]) *auto*
qed (*simp add*: *zero-enat-def one-enat-def*)

lemma *enat-ord-number* [*simp*]:
(*numeral m* :: *enat*) $\leq$ *numeral n* $\longleftrightarrow$ (*numeral m* :: *nat*) $\leq$ *numeral n*
(*numeral m* :: *enat*) $<$ *numeral n* $\longleftrightarrow$ (*numeral m* :: *nat*) $<$ *numeral n*
by (*simp-all add*: *numeral-eq-enat*)

lemma *infinity-ileE* [*elim!*]: $\infty \leq$ *enat m* $\Longrightarrow R$

by (*simp add*: *zero-enat-def less-eq-enat-def split*: *enat.splits*)

lemma *infinity-ilessE* [*elim*!]: $\infty < enat\ m \Longrightarrow R$
by *simp*

lemma *eSuc-ile-mono* [*simp*]: $eSuc\ n \leq eSuc\ m \longleftrightarrow n \leq m$
by (*simp add*: *eSuc-def less-eq-enat-def split*: *enat.splits*)

lemma *eSuc-mono* [*simp*]: $eSuc\ n < eSuc\ m \longleftrightarrow n < m$
by (*simp add*: *eSuc-def less-enat-def split*: *enat.splits*)

lemma *ile-eSuc* [*simp*]: $n \leq eSuc\ n$
by (*simp add*: *eSuc-def less-eq-enat-def split*: *enat.splits*)

lemma *not-eSuc-ilei0* [*simp*]: $\neg\ eSuc\ n \leq 0$
by (*simp add*: *zero-enat-def eSuc-def less-eq-enat-def split*: *enat.splits*)

lemma *i0-iless-eSuc* [*simp*]: $0 < eSuc\ n$
by (*simp add*: *zero-enat-def eSuc-def less-enat-def split*: *enat.splits*)

lemma *iless-eSuc0*[*simp*]: $(n < eSuc\ 0) = (n = 0)$
by (*simp add*: *zero-enat-def eSuc-def less-enat-def split*: *enat.split*)

lemma *ileI1*: $m < n \Longrightarrow eSuc\ m \leq n$
by (*simp add*: *eSuc-def less-eq-enat-def less-enat-def split*: *enat.splits*)

lemma *Suc-ile-eq*: $enat\ (Suc\ m) \leq n \longleftrightarrow enat\ m < n$
by (*cases n*) *auto*

lemma *iless-Suc-eq* [*simp*]: $enat\ m < eSuc\ n \longleftrightarrow enat\ m \leq n$
by (*auto simp add*: *eSuc-def less-enat-def split*: *enat.splits*)

lemma *imult-infinity*: $(0{::}enat) < n \Longrightarrow \infty * n = \infty$
by (*simp add*: *zero-enat-def less-enat-def split*: *enat.splits*)

lemma *imult-infinity-right*: $(0{::}enat) < n \Longrightarrow n * \infty = \infty$
by (*simp add*: *zero-enat-def less-enat-def split*: *enat.splits*)

lemma *enat-0-less-mult-iff*: $(0 < (m{::}enat) * n) = (0 < m \land 0 < n)$
by (*simp only*: *zero-less-iff-neq-zero mult-eq-0-iff*, *simp*)

lemma *mono-eSuc*: *mono eSuc*
by (*simp add*: *mono-def*)

lemma *min-enat-simps* [*simp*]:
$min\ (enat\ m)\ (enat\ n) = enat\ (min\ m\ n)$
$min\ q\ 0 = 0$
$min\ 0\ q = 0$
$min\ q\ (\infty{::}enat) = q$

min (∞::*enat*) *q* = *q*
by (*auto simp add*: *min-def*)

lemma *max-enat-simps* [*simp*]:
max (*enat m*) (*enat n*) = *enat* (*max m n*)
max q 0 = *q*
max 0 q = *q*
max q ∞ = (∞::*enat*)
max ∞ *q* = (∞::*enat*)
by (*simp-all add*: *max-def*)

lemma *enat-ile*: $n \leq$ *enat m* $\Longrightarrow \exists k.\ n =$ *enat k*
by (*cases n*) *simp-all*

lemma *enat-iless*: $n <$ *enat m* $\Longrightarrow \exists k.\ n =$ *enat k*
by (*cases n*) *simp-all*

lemma *iadd-le-enat-iff*:
$x + y \leq$ *enat n* $\longleftrightarrow (\exists y'\ x'.\ x =$ *enat* $x' \wedge y =$ *enat* $y' \wedge x' + y' \leq n)$
by(*cases x y rule*: *enat.exhaust*[*case-product enat.exhaust*]) *simp-all*

lemma *chain-incr*: $\forall i.\ \exists j.\ Y\ i < Y\ j \Longrightarrow \exists j.$ *enat k* $< Y\ j$
apply (*induct-tac k*)
 apply (*simp* (*no-asm*) *only*: *enat-0*)
 apply (*fast intro*: *le-less-trans* [*OF zero-le*])
apply (*erule exE*)
apply (*drule spec*)
apply (*erule exE*)
apply (*drule ileI1*)
apply (*rule eSuc-enat* [*THEN subst*])
apply (*rule exI*)
apply (*erule* (*1*) *le-less-trans*)
done

lemma *eSuc-max*: *eSuc* (*max x y*) = *max* (*eSuc x*) (*eSuc y*)
by (*simp add*: *eSuc-def split*: *enat.split*)

lemma *eSuc-Max*:
assumes *finite A A* ≠ {}
shows *eSuc* (*Max A*) = *Max* (*eSuc* ' *A*)
using *assms* **proof** *induction*
case (*insert x A*)
thus *?case* **by**(*cases A* = {})(*simp-all add*: *eSuc-max*)
qed *simp*

instantiation *enat* :: {*order-bot*, *order-top*}
begin

definition *bot-enat* :: *enat* **where** *bot-enat* = *0*

definition *top-enat* :: *enat* **where** *top-enat* = ∞

instance
  by *standard* (*simp-all add*: *bot-enat-def top-enat-def*)

end

lemma *finite-enat-bounded*:
  assumes *le-fin*: ⋀*y*. *y* ∈ *A* ⟹ *y* ≤ *enat n*
  shows *finite A*
proof (*rule finite-subset*)
  show *finite* (*enat* ‘ {..*n*}) **by** *blast*
  have *A* ⊆ {..*enat n*} **using** *le-fin* **by** *fastforce*
  also have ... ⊆ *enat* ‘ {..*n*}
    apply (*rule subsetI*)
    subgoal for *x* **by** (*cases x*) *auto*
    done
  finally show *A* ⊆ *enat* ‘ {..*n*} **.**
qed

31.8 Cancellation simprocs

lemma *enat-add-left-cancel*: *a* + *b* = *a* + *c* ⟷ *a* = (∞::*enat*) ∨ *b* = *c*
  unfolding *plus-enat-def* **by** (*simp split*: *enat.split*)

lemma *enat-add-left-cancel-le*: *a* + *b* ≤ *a* + *c* ⟷ *a* = (∞::*enat*) ∨ *b* ≤ *c*
  unfolding *plus-enat-def* **by** (*simp split*: *enat.split*)

lemma *enat-add-left-cancel-less*: *a* + *b* < *a* + *c* ⟷ *a* ≠ (∞::*enat*) ∧ *b* < *c*
  unfolding *plus-enat-def* **by** (*simp split*: *enat.split*)

ML ‹
```
structure Cancel-Enat-Common =
struct
  (* copied from src/HOL/Tools/nat-numeral-simprocs.ML *)
  fun find-first-t -    - []         = raise TERM(find-first-t, [])
    | find-first-t past u (t::terms) =
          if u aconv t then (rev past @ terms)
          else find-first-t (t::past) u terms

  fun dest-summing (Const (@{const-name Groups.plus}, -) $ t $ u, ts) =
        dest-summing (t, dest-summing (u, ts))
    | dest-summing (t, ts) = t :: ts

  val mk-sum = Arith-Data.long-mk-sum
  fun dest-sum t = dest-summing (t, [])
  val find-first = find-first-t []
  val trans-tac = Numeral-Simprocs.trans-tac
  val norm-ss =
```

```
    simpset-of (put-simpset HOL-basic-ss @{context}
      addsimps @{thms ac-simps add-0-left add-0-right})
  fun norm-tac ctxt = ALLGOALS (simp-tac (put-simpset norm-ss ctxt))
  fun simplify-meta-eq ctxt cancel-th th =
    Arith-Data.simplify-meta-eq [] ctxt
      ([th, cancel-th] MRS trans)
  fun mk-eq (a, b) = HOLogic.mk-Trueprop (HOLogic.mk-eq (a, b))
end

structure Eq-Enat-Cancel = ExtractCommonTermFun
(open Cancel-Enat-Common
  val mk-bal = HOLogic.mk-eq
  val dest-bal = HOLogic.dest-bin @{const-name HOL.eq} @{typ enat}
  fun simp-conv - - = SOME @{thm enat-add-left-cancel}
)

structure Le-Enat-Cancel = ExtractCommonTermFun
(open Cancel-Enat-Common
  val mk-bal = HOLogic.mk-binrel @{const-name Orderings.less-eq}
  val dest-bal = HOLogic.dest-bin @{const-name Orderings.less-eq} @{typ enat}
  fun simp-conv - - = SOME @{thm enat-add-left-cancel-le}
)

structure Less-Enat-Cancel = ExtractCommonTermFun
(open Cancel-Enat-Common
  val mk-bal = HOLogic.mk-binrel @{const-name Orderings.less}
  val dest-bal = HOLogic.dest-bin @{const-name Orderings.less} @{typ enat}
  fun simp-conv - - = SOME @{thm enat-add-left-cancel-less}
)
›
```

simproc-setup *enat-eq-cancel*
  $((l::enat) + m = n \mid (l::enat) = m + n) =$
  ‹*fn phi => fn ctxt => fn ct => Eq-Enat-Cancel.proc ctxt (Thm.term-of ct)*›

simproc-setup *enat-le-cancel*
  $((l::enat) + m \leq n \mid (l::enat) \leq m + n) =$
  ‹*fn phi => fn ctxt => fn ct => Le-Enat-Cancel.proc ctxt (Thm.term-of ct)*›

simproc-setup *enat-less-cancel*
  $((l::enat) + m < n \mid (l::enat) < m + n) =$
  ‹*fn phi => fn ctxt => fn ct => Less-Enat-Cancel.proc ctxt (Thm.term-of ct)*›

TODO: add regression tests for these simprocs

TODO: add simprocs for combining and cancelling numerals

31.9 Well-ordering

lemma *less-enatE*:

[| $n < enat\ m$; !!k. $n = enat\ k$ ==> $k < m$ ==> P |] ==> P
by (*induct n*) *auto*

lemma *less-infinityE*:
[| $n < \infty$; !!k. $n = enat\ k$ ==> P |] ==> P
by (*induct n*) *auto*

lemma *enat-less-induct*:
assumes *prem*: !!n. $\forall m$::*enat*. $m < n$ --> $P\ m$ ==> $P\ n$ **shows** $P\ n$
proof −
have *P-enat*: !!k. P (*enat k*)
apply (*rule nat-less-induct*)
apply (*rule prem*, *clarify*)
apply (*erule less-enatE*, *simp*)
done
show *?thesis*
proof (*induct n*)
fix *nat*
show P (*enat nat*) **by** (*rule P-enat*)
next
show $P\ \infty$
apply (*rule prem*, *clarify*)
apply (*erule less-infinityE*)
apply (*simp add*: *P-enat*)
done
qed
qed

instance *enat* :: *wellorder*
proof
fix P **and** n
assume *hyp*: ($\bigwedge n$::*enat*. ($\bigwedge m$::*enat*. $m < n \Longrightarrow P\ m$) $\Longrightarrow P\ n$)
show $P\ n$ **by** (*blast intro*: *enat-less-induct hyp*)
qed

31.10 Complete Lattice

instantiation *enat* :: *complete-lattice*
begin

definition *inf-enat* :: *enat* ⇒ *enat* ⇒ *enat* **where**
inf-enat = *min*

definition *sup-enat* :: *enat* ⇒ *enat* ⇒ *enat* **where**
sup-enat = *max*

definition *Inf-enat* :: *enat set* ⇒ *enat* **where**
Inf-enat A = (*if* A = {} *then* ∞ *else* (*LEAST* x. $x \in A$))

definition *Sup-enat* :: *enat set* ⇒ *enat* **where**
Sup-enat A = (*if A* = {} *then 0 else if finite A then Max A else* ∞)
instance
proof
fix *x* :: *enat* **and** *A* :: *enat set*
{ assume $x \in A$ **then show** *Inf A* ≤ *x*
unfolding *Inf-enat-def* **by** (*auto intro*: *Least-le*) **}**
{ assume $\bigwedge y.\ y \in A \Longrightarrow x \le y$ **then show** *x* ≤ *Inf A*
unfolding *Inf-enat-def*
by (*cases A* = {}) (*auto intro*: *LeastI2-ex*) **}**
{ assume $x \in A$ **then show** *x* ≤ *Sup A*
unfolding *Sup-enat-def* **by** (*cases finite A*) *auto* **}**
{ assume $\bigwedge y.\ y \in A \Longrightarrow y \le x$ **then show** *Sup A* ≤ *x*
unfolding *Sup-enat-def* **using** *finite-enat-bounded* **by** *auto* **}**
qed (*simp-all add*:
inf-enat-def sup-enat-def bot-enat-def top-enat-def Inf-enat-def Sup-enat-def)
end

instance *enat* :: *complete-linorder* **..**

lemma *eSuc-Sup*: *A* ≠ {} ⟹ *eSuc* (*Sup A*) = *Sup* (*eSuc* ' *A*)
by(*auto simp add*: *Sup-enat-def eSuc-Max inj-on-def dest*: *finite-imageD*)

lemma *sup-continuous-eSuc*: *sup-continuous f* ⟹ *sup-continuous* (λ*x*. *eSuc* (*f* *x*))
using *eSuc-Sup*[*of* - ' *UNIV*] **by** (*auto simp*: *sup-continuous-def*)

31.11 Traditional theorem names

lemmas *enat-defs* = *zero-enat-def one-enat-def eSuc-def*
plus-enat-def less-eq-enat-def less-enat-def

lemma *iadd-is-0*: (*m* + *n* = (*0*::*enat*)) = (*m* = *0* ∧ *n* = *0*)
by (*rule add-eq-0-iff-both-eq-0*)

lemma *i0-lb* : (*0*::*enat*) ≤ *n*
by (*rule zero-le*)

lemma *ile0-eq*: *n* ≤ (*0*::*enat*) ⟷ *n* = *0*
by (*rule le-zero-eq*)

lemma *not-iless0*: ¬ *n* < (*0*::*enat*)
by (*rule not-less-zero*)

lemma *i0-less*[*simp*]: (*0*::*enat*) < *n* ⟷ *n* ≠ *0*
by (*rule zero-less-iff-neq-zero*)

lemma *imult-is-0*: ((*m*::*enat*) ∗ *n* = *0*) = (*m* = *0* ∨ *n* = *0*)
by (*rule mult-eq-0-iff*)

end

32 Liminf and Limsup on conditionally complete lattices

theory *Liminf-Limsup*
imports *Complex-Main*
begin

lemma (**in** *conditionally-complete-linorder*) *le-cSup-iff*:
  assumes $A \neq \{\}$ *bdd-above A*
  shows $x \leq Sup\ A \longleftrightarrow (\forall y<x.\ \exists a\in A.\ y < a)$
proof *safe*
  fix y **assume** $x \leq Sup\ A\ y < x$
  then have $y < Sup\ A$ **by** *auto*
  then show $\exists a\in A.\ y < a$
    unfolding *less-cSup-iff*[*OF assms*] **.**
qed (*auto elim*!: *allE*[*of - Sup A*] *simp add*: *not-le*[*symmetric*] *cSup-upper assms*)

lemma (**in** *conditionally-complete-linorder*) *le-cSUP-iff*:
  $A \neq \{\} \Longrightarrow$ *bdd-above* $(f`A) \Longrightarrow x \leq SUPREMUM\ A\ f \longleftrightarrow (\forall y<x.\ \exists i\in A.\ y < f\ i)$
  using *le-cSup-iff* [*of f ` A*] **by** *simp*

lemma *le-cSup-iff-less*:
  fixes $x :: 'a :: \{conditionally\text{-}complete\text{-}linorder, dense\text{-}linorder\}$
  shows $A \neq \{\} \Longrightarrow$ *bdd-above* $(f`A) \Longrightarrow x \leq (SUP\ i{:}A.\ f\ i) \longleftrightarrow (\forall y<x.\ \exists i\in A.\ y \leq f\ i)$
  by (*simp add*: *le-cSUP-iff*)
    (*blast intro*: *less-imp-le less-trans less-le-trans dest*: *dense*)

lemma *le-Sup-iff-less*:
  fixes $x :: 'a :: \{complete\text{-}linorder, dense\text{-}linorder\}$
  shows $x \leq (SUP\ i{:}A.\ f\ i) \longleftrightarrow (\forall y<x.\ \exists i\in A.\ y \leq f\ i)$ (**is** *?lhs* = *?rhs*)
  unfolding *le-SUP-iff*
  by (*blast intro*: *less-imp-le less-trans less-le-trans dest*: *dense*)

lemma (**in** *conditionally-complete-linorder*) *cInf-le-iff*:
  assumes $A \neq \{\}$ *bdd-below A*
  shows $Inf\ A \leq x \longleftrightarrow (\forall y>x.\ \exists a\in A.\ y > a)$
proof *safe*
  fix y **assume** $x \geq Inf\ A\ y > x$
  then have $y > Inf\ A$ **by** *auto*
  then show $\exists a\in A.\ y > a$
    unfolding *cInf-less-iff*[*OF assms*] **.**
qed (*auto elim*!: *allE*[*of - Inf A*] *simp add*: *not-le*[*symmetric*] *cInf-lower assms*)

lemma (**in** *conditionally-complete-linorder*) *cINF-le-iff*:
$A \neq \{\} \Longrightarrow$ *bdd-below* $(f`A) \Longrightarrow$ *INFIMUM* $A\ f \leq x \longleftrightarrow (\forall y>x.\ \exists i\in A.\ y > f\ i)$
using *cInf-le-iff* [*of f ' A*] **by** *simp*

lemma *cInf-le-iff-less*:
fixes $x :: 'a ::$ {*conditionally-complete-linorder*, *dense-linorder*}
shows $A \neq \{\} \Longrightarrow$ *bdd-below* $(f`A) \Longrightarrow$ (*INF* $i{:}A.\ f\ i) \leq x \longleftrightarrow (\forall y>x.\ \exists i\in A.\ f\ i \leq y)$
by (*simp add*: *cINF-le-iff*)
(*blast intro*: *less-imp-le less-trans le-less-trans dest*: *dense*)

lemma *Inf-le-iff-less*:
fixes $x :: 'a ::$ {*complete-linorder*, *dense-linorder*}
shows (*INF* $i{:}A.\ f\ i) \leq x \longleftrightarrow (\forall y>x.\ \exists i\in A.\ f\ i \leq y)$
unfolding *INF-le-iff*
by (*blast intro*: *less-imp-le less-trans le-less-trans dest*: *dense*)

lemma *SUP-pair*:
fixes $f :: - \Rightarrow - \Rightarrow - ::$ *complete-lattice*
shows (*SUP* $i : A.$ *SUP* $j : B.\ f\ i\ j) =$ (*SUP* $p : A \times B.\ f$ (*fst* p) (*snd* p))
by (*rule antisym*) (*auto intro*!: *SUP-least SUP-upper2*)

lemma *INF-pair*:
fixes $f :: - \Rightarrow - \Rightarrow - ::$ *complete-lattice*
shows (*INF* $i : A.$ *INF* $j : B.\ f\ i\ j) =$ (*INF* $p : A \times B.\ f$ (*fst* p) (*snd* p))
by (*rule antisym*) (*auto intro*!: *INF-greatest INF-lower2*)

32.0.1 *Liminf* and *Limsup*

definition *Liminf* :: $'a$ *filter* $\Rightarrow ('a \Rightarrow 'b) \Rightarrow 'b ::$ *complete-lattice* **where**
Liminf $F\ f =$ (*SUP* $P{:}\{P.$ *eventually* $P\ F\}.$ *INF* $x{:}\{x.\ P\ x\}.\ f\ x)$

definition *Limsup* :: $'a$ *filter* $\Rightarrow ('a \Rightarrow 'b) \Rightarrow 'b ::$ *complete-lattice* **where**
Limsup $F\ f =$ (*INF* $P{:}\{P.$ *eventually* $P\ F\}.$ *SUP* $x{:}\{x.\ P\ x\}.\ f\ x)$

abbreviation *liminf* $\equiv$ *Liminf sequentially*

abbreviation *limsup* $\equiv$ *Limsup sequentially*

lemma *Liminf-eqI*:
$(\bigwedge P.$ *eventually* $P\ F \Longrightarrow$ *INFIMUM* (*Collect* P) $f \leq x) \Longrightarrow$
$(\bigwedge y.\ (\bigwedge P.$ *eventually* $P\ F \Longrightarrow$ *INFIMUM* (*Collect* P) $f \leq y) \Longrightarrow x \leq y) \Longrightarrow$ *Liminf* $F\ f = x$
unfolding *Liminf-def* **by** (*auto intro*!: *SUP-eqI*)

lemma *Limsup-eqI*:
$(\bigwedge P.$ *eventually* $P\ F \Longrightarrow x \leq$ *SUPREMUM* (*Collect* P) $f) \Longrightarrow$
$(\bigwedge y.\ (\bigwedge P.$ *eventually* $P\ F \Longrightarrow y \leq$ *SUPREMUM* (*Collect* P) $f) \Longrightarrow y \leq x)$

$\Longrightarrow$ *Limsup F f* = *x*
unfolding *Limsup-def* **by** (*auto intro*!: *INF-eqI*)

lemma *liminf-SUP-INF*: *liminf f* = (*SUP n. INF m*:{*n*..}. *f m*)
unfolding *Liminf-def eventually-sequentially*
by (*rule SUP-eq*) (*auto simp*: *atLeast-def intro*!: *INF-mono*)

lemma *limsup-INF-SUP*: *limsup f* = (*INF n. SUP m*:{*n*..}. *f m*)
unfolding *Limsup-def eventually-sequentially*
by (*rule INF-eq*) (*auto simp*: *atLeast-def intro*!: *SUP-mono*)

lemma *Limsup-const*:
assumes *ntriv*: ¬ *trivial-limit F*
shows *Limsup F* (λ*x. c*) = *c*
proof –
have ∗: ⋀*P. Ex P* ⟷ *P* ≠ (λ*x. False*) **by** *auto*
have ⋀*P. eventually P F* ⟹ (*SUP x* : {*x. P x*}. *c*) = *c*
using *ntriv* **by** (*intro SUP-const*) (*auto simp*: *eventually-False* ∗)
then show *?thesis*
unfolding *Limsup-def* **using** *eventually-True*
by (*subst INF-cong*[**where** *D*=λ*x. c*])
(*auto intro*!: *INF-const simp del*: *eventually-True*)
qed

lemma *Liminf-const*:
assumes *ntriv*: ¬ *trivial-limit F*
shows *Liminf F* (λ*x. c*) = *c*
proof –
have ∗: ⋀*P. Ex P* ⟷ *P* ≠ (λ*x. False*) **by** *auto*
have ⋀*P. eventually P F* ⟹ (*INF x* : {*x. P x*}. *c*) = *c*
using *ntriv* **by** (*intro INF-const*) (*auto simp*: *eventually-False* ∗)
then show *?thesis*
unfolding *Liminf-def* **using** *eventually-True*
by (*subst SUP-cong*[**where** *D*=λ*x. c*])
(*auto intro*!: *SUP-const simp del*: *eventually-True*)
qed

lemma *Liminf-mono*:
assumes *ev*: *eventually* (λ*x. f x* ≤ *g x*) *F*
shows *Liminf F f* ≤ *Liminf F g*
unfolding *Liminf-def*
proof (*safe intro*!: *SUP-mono*)
fix *P* **assume** *eventually P F*
with *ev* **have** *eventually* (λ*x. f x* ≤ *g x* ∧ *P x*) *F* (**is** *eventually ?Q F*) **by** (*rule eventually-conj*)
then show ∃ *Q*∈{*P. eventually P F*}. *INFIMUM* (*Collect P*) *f* ≤ *INFIMUM* (*Collect Q*) *g*
by (*intro bexI*[*of - ?Q*]) (*auto intro*!: *INF-mono*)
qed

lemma *Liminf-eq*:
assumes *eventually* (λx. *f x* = *g x*) *F*
shows *Liminf F f* = *Liminf F g*
by (*intro antisym Liminf-mono eventually-mono*[*OF assms*]) *auto*

lemma *Limsup-mono*:
assumes *ev*: *eventually* (λx. *f x* ≤ *g x*) *F*
shows *Limsup F f* ≤ *Limsup F g*
unfolding *Limsup-def*
proof (*safe intro*!: *INF-mono*)
fix *P* **assume** *eventually P F*
with *ev* **have** *eventually* (λx. *f x* ≤ *g x* ∧ *P x*) *F* (**is** *eventually ?Q F*) **by** (*rule eventually-conj*)
then show ∃ *Q*∈{*P*. *eventually P F*}. *SUPREMUM* (*Collect Q*) *f* ≤ *SUPREMUM* (*Collect P*) *g*
by (*intro bexI*[*of - ?Q*]) (*auto intro*!: *SUP-mono*)
qed

lemma *Limsup-eq*:
assumes *eventually* (λx. *f x* = *g x*) *net*
shows *Limsup net f* = *Limsup net g*
by (*intro antisym Limsup-mono eventually-mono*[*OF assms*]) *auto*

lemma *Liminf-le-Limsup*:
assumes *ntriv*: ¬ *trivial-limit F*
shows *Liminf F f* ≤ *Limsup F f*
unfolding *Limsup-def Liminf-def*
apply (*rule SUP-least*)
apply (*rule INF-greatest*)
proof *safe*
fix *P Q* **assume** *eventually P F eventually Q F*
then have *eventually* (λx. *P x* ∧ *Q x*) *F* (**is** *eventually ?C F*) **by** (*rule eventually-conj*)
then have *not-False*: (λx. *P x* ∧ *Q x*) ≠ (λx. *False*)
using *ntriv* **by** (*auto simp add*: *eventually-False*)
have *INFIMUM* (*Collect P*) *f* ≤ *INFIMUM* (*Collect ?C*) *f*
by (*rule INF-mono*) *auto*
also have ... ≤ *SUPREMUM* (*Collect ?C*) *f*
using *not-False* **by** (*intro INF-le-SUP*) *auto*
also have ... ≤ *SUPREMUM* (*Collect Q*) *f*
by (*rule SUP-mono*) *auto*
finally show *INFIMUM* (*Collect P*) *f* ≤ *SUPREMUM* (*Collect Q*) *f* **.**
qed

lemma *Liminf-bounded*:
assumes *ntriv*: ¬ *trivial-limit F*
assumes *le*: *eventually* (λn. *C* ≤ *X n*) *F*
shows *C* ≤ *Liminf F X*
using *Liminf-mono*[*OF le*] *Liminf-const*[*OF ntriv*, *of C*] **by** *simp*

lemma *Limsup-bounded*:
  assumes *ntriv*: $\neg$ *trivial-limit F*
  assumes *le*: *eventually* ($\lambda n.\ X\ n \leq C$) *F*
  shows *Limsup F X* $\leq C$
  using *Limsup-mono*[*OF le*] *Limsup-const*[*OF ntriv*, *of C*] **by** *simp*

lemma *le-Limsup*:
  assumes *F*: $F \neq bot$ **and** *x*: $\forall_F\ x\ in\ F.\ l \leq f\ x$
  shows $l \leq$ *Limsup F f*
proof –
  have $l =$ *Limsup F* ($\lambda x.\ l$)
    using *F* **by** (*simp add*: *Limsup-const*)
  also have ... $\leq$ *Limsup F f*
    by (*intro Limsup-mono x*)
  finally show *?thesis* **.**
qed

lemma *le-Liminf-iff*:
  fixes *X* :: - $\Rightarrow$ - :: *complete-linorder*
  shows $C \leq$ *Liminf F X* $\longleftrightarrow$ ($\forall y<C.$ *eventually* ($\lambda x.\ y < X\ x$) *F*)
proof –
  have *eventually* ($\lambda x.\ y < X\ x$) *F*
    if *eventually P F* $y <$ *INFIMUM* (*Collect P*) *X* **for** *y P*
    using *that* **by** (*auto elim*!: *eventually-mono dest*: *less-INF-D*)
  moreover
  have $\exists P.$ *eventually P F* $\wedge\ y <$ *INFIMUM* (*Collect P*) *X*
    if $y < C$ **and** *y*: $\forall y<C.$ *eventually* ($\lambda x.\ y < X\ x$) *F* **for** *y P*
  proof (*cases* $\exists z.\ y < z \wedge z < C$)
    case *True*
    then obtain *z* **where** *z*: $y < z \wedge z < C$ **..**
    moreover from *z* **have** $z \leq$ *INFIMUM* $\{x.\ z < X\ x\}$ *X*
      by (*auto intro*!: *INF-greatest*)
    ultimately show *?thesis*
      using *y* **by** (*intro exI*[*of* - $\lambda x.\ z < X\ x$]) *auto*
  next
    case *False*
    then have $C \leq$ *INFIMUM* $\{x.\ y < X\ x\}$ *X*
      by (*intro INF-greatest*) *auto*
    with ‹$y < C$› **show** *?thesis*
      using *y* **by** (*intro exI*[*of* - $\lambda x.\ y < X\ x$]) *auto*
  qed
  ultimately show *?thesis*
    unfolding *Liminf-def le-SUP-iff* **by** *auto*
qed

lemma *Limsup-le-iff*:
  fixes *X* :: - $\Rightarrow$ - :: *complete-linorder*
  shows $C \geq$ *Limsup F X* $\longleftrightarrow$ ($\forall y>C.$ *eventually* ($\lambda x.\ y > X\ x$) *F*)

```
proof -
  { fix y P assume eventually P F y > SUPREMUM (Collect P) X
    then have eventually (λx. y > X x) F
      by (auto elim!: eventually-mono dest: SUP-lessD) }
  moreover
  { fix y P assume y > C and y: ∀y>C. eventually (λx. y > X x) F
    have ∃P. eventually P F ∧ y > SUPREMUM (Collect P) X
    proof (cases ∃z. C < z ∧ z < y)
      case True
      then obtain z where z: C < z ∧ z < y ..
      moreover from z have z ≥ SUPREMUM {x. z > X x} X
        by (auto intro!: SUP-least)
      ultimately show ?thesis
        using y by (intro exI[of - λx. z > X x]) auto
    next
      case False
      then have C ≥ SUPREMUM {x. y > X x} X
        by (intro SUP-least) (auto simp: not-less)
      with ‹y > C› show ?thesis
        using y by (intro exI[of - λx. y > X x]) auto
    qed }
  ultimately show ?thesis
    unfolding Limsup-def INF-le-iff by auto
qed

lemma less-LiminfD:
  y < Liminf F (f :: - ⇒ 'a :: complete-linorder) ⟹ eventually (λx. f x > y) F
  using le-Liminf-iff[of Liminf F f F f] by simp

lemma Limsup-lessD:
  y > Limsup F (f :: - ⇒ 'a :: complete-linorder) ⟹ eventually (λx. f x < y) F
  using Limsup-le-iff[of F f Limsup F f] by simp

lemma lim-imp-Liminf:
  fixes f :: 'a ⇒ - :: {complete-linorder,linorder-topology}
  assumes ntriv: ¬ trivial-limit F
  assumes lim: (f ⟶ f0) F
  shows Liminf F f = f0
proof (intro Liminf-eqI)
  fix P assume P: eventually P F
  then have eventually (λx. INFIMUM (Collect P) f ≤ f x) F
    by eventually-elim (auto intro!: INF-lower)
  then show INFIMUM (Collect P) f ≤ f0
    by (rule tendsto-le[OF ntriv lim tendsto-const])
next
  fix y assume upper: ⋀P. eventually P F ⟹ INFIMUM (Collect P) f ≤ y
  show f0 ≤ y
  proof cases
    assume ∃z. y < z ∧ z < f0
```

```
    then obtain z where y < z ∧ z < f0 ..
    moreover have z ≤ INFIMUM {x. z < f x} f
      by (rule INF-greatest) simp
    ultimately show ?thesis
      using lim[THEN topological-tendstoD, THEN upper, of {z <..}] by auto
  next
    assume discrete: ¬ (∃ z. y < z ∧ z < f0)
    show ?thesis
    proof (rule classical)
      assume ¬ f0 ≤ y
      then have eventually (λx. y < f x) F
        using lim[THEN topological-tendstoD, of {y <..}] by auto
      then have eventually (λx. f0 ≤ f x) F
        using discrete by (auto elim!: eventually-mono)
      then have INFIMUM {x. f0 ≤ f x} f ≤ y
        by (rule upper)
      moreover have f0 ≤ INFIMUM {x. f0 ≤ f x} f
        by (intro INF-greatest) simp
      ultimately show f0 ≤ y by simp
    qed
  qed
qed

lemma lim-imp-Limsup:
  fixes f :: 'a ⇒ - :: {complete-linorder,linorder-topology}
  assumes ntriv: ¬ trivial-limit F
  assumes lim: (f ⟶ f0) F
  shows Limsup F f = f0
proof (intro Limsup-eqI)
  fix P assume P: eventually P F
  then have eventually (λx. f x ≤ SUPREMUM (Collect P) f) F
    by eventually-elim (auto intro!: SUP-upper)
  then show f0 ≤ SUPREMUM (Collect P) f
    by (rule tendsto-le[OF ntriv tendsto-const lim])
next
  fix y assume lower: ⋀P. eventually P F ⟹ y ≤ SUPREMUM (Collect P) f
  show y ≤ f0
  proof (cases ∃ z. f0 < z ∧ z < y)
    case True
    then obtain z where f0 < z ∧ z < y ..
    moreover have SUPREMUM {x. f x < z} f ≤ z
      by (rule SUP-least) simp
    ultimately show ?thesis
      using lim[THEN topological-tendstoD, THEN lower, of {..< z}] by auto
  next
    case False
    show ?thesis
    proof (rule classical)
      assume ¬ y ≤ f0
```

then have *eventually* $(\lambda x.\ f\ x < y)\ F$
  using *lim*[*THEN topological-tendstoD*, *of* $\{..< y\}$] **by** *auto*
then have *eventually* $(\lambda x.\ f\ x \leq f0)\ F$
  using *False* **by** (*auto elim*!: *eventually-mono simp*: *not-less*)
then have $y \leq SUPREMUM\ \{x.\ f\ x \leq f0\}\ f$
  by (*rule lower*)
moreover have $SUPREMUM\ \{x.\ f\ x \leq f0\}\ f \leq f0$
  by (*intro SUP-least*) *simp*
ultimately show $y \leq f0$ **by** *simp*
qed
qed
qed

lemma *Liminf-eq-Limsup*:
fixes *f0* :: *'a* :: {*complete-linorder*,*linorder-topology*}
assumes *ntriv*: ¬ *trivial-limit F*
and *lim*: *Liminf F f* = *f0 Limsup F f* = *f0*
shows $(f \longrightarrow f0)\ F$
proof (*rule order-tendstoI*)
fix *a* **assume** $f0 < a$
with *assms* **have** $Limsup\ F\ f < a$ **by** *simp*
then obtain *P* **where** *eventually P F SUPREMUM* (*Collect P*) $f < a$
unfolding *Limsup-def INF-less-iff* **by** *auto*
then show *eventually* $(\lambda x.\ f\ x < a)\ F$
by (*auto elim*!: *eventually-mono dest*: *SUP-lessD*)
next
fix *a* **assume** $a < f0$
with *assms* **have** $a < Liminf\ F\ f$ **by** *simp*
then obtain *P* **where** *eventually P F* $a <$ *INFIMUM* (*Collect P*) *f*
unfolding *Liminf-def less-SUP-iff* **by** *auto*
then show *eventually* $(\lambda x.\ a < f\ x)\ F$
by (*auto elim*!: *eventually-mono dest*: *less-INF-D*)
qed

lemma *tendsto-iff-Liminf-eq-Limsup*:
fixes *f0* :: *'a* :: {*complete-linorder*,*linorder-topology*}
shows ¬ *trivial-limit F* $\Longrightarrow (f \longrightarrow f0)\ F \longleftrightarrow$ (*Liminf F f* = *f0* ∧ *Limsup F f* = *f0*)
by (*metis Liminf-eq-Limsup lim-imp-Limsup lim-imp-Liminf*)

lemma *liminf-subseq-mono*:
fixes *X* :: *nat* ⇒ *'a* :: *complete-linorder*
assumes *subseq r*
shows *liminf X* ≤ *liminf* $(X \circ r)$
proof−
have ⋀*n*. (*INF m*:{*n*..}. *X m*) ≤ (*INF m*:{*n*..}. $(X \circ r)$ *m*)
proof (*safe intro*!: *INF-mono*)
fix *n m* :: *nat* **assume** $n \leq m$ **then show** ∃ *ma*∈{*n*..}. *X ma* ≤ $(X \circ r)$ *m*
using *seq-suble*[*OF* ‹*subseq r*›, *of m*] **by** (*intro bexI*[*of - r m*]) *auto*

qed
then show *?thesis* **by** (*auto intro*!: *SUP-mono simp*: *liminf-SUP-INF comp-def*)
qed

lemma *limsup-subseq-mono*:
fixes X :: *nat* ⇒ *'a* :: *complete-linorder*
assumes *subseq r*
shows *limsup* $(X \circ r) \leq$ *limsup* X
proof−
have (*SUP* m:{n..}. $(X \circ r)$ m) $\leq$ (*SUP* m:{n..}. X m) **for** n
proof (*safe intro*!: *SUP-mono*)
fix m :: *nat*
assume $n \leq m$
then show ∃ *ma*∈{n..}. $(X \circ r)$ $m \leq X$ *ma*
using *seq-suble*[*OF* ‹*subseq r*›, *of m*] **by** (*intro bexI*[*of - r m*]) *auto*
qed
then show *?thesis*
by (*auto intro*!: *INF-mono simp*: *limsup-INF-SUP comp-def*)
qed

lemma *continuous-on-imp-continuous-within*:
continuous-on s f ⟹ $t \subseteq s$ ⟹ $x \in s$ ⟹ *continuous* (*at x within t*) *f*
unfolding *continuous-on-eq-continuous-within*
by (*auto simp*: *continuous-within intro*: *tendsto-within-subset*)

lemma *Liminf-compose-continuous-mono*:
fixes *f* :: *'a*::{*complete-linorder*, *linorder-topology*} ⇒ *'b*::{*complete-linorder*, *linorder-topology*}
assumes *c*: *continuous-on UNIV f* **and** *am*: *mono f* **and** *F*: $F \neq$ *bot*
shows *Liminf F* (λ*n*. *f* (*g n*)) = *f* (*Liminf F g*)
proof −
{ fix *P* **assume** *eventually P F*
have ∃ *x*. *P x*
proof (*rule ccontr*)
assume ¬ (∃ *x*. *P x*) **then have** *P* = (λ*x*. *False*)
by *auto*
with ‹*eventually P F*› *F* **show** *False*
by *auto*
qed }
note ∗ = *this*

have *f* (*Liminf F g*) = (*SUP P* : {*P*. *eventually P F*}. *f* (*Inf* (*g* ‘ *Collect P*)))
unfolding *Liminf-def*
by (*subst continuous-at-Sup-mono*[*OF am continuous-on-imp-continuous-within*[*OF c*]])
(*auto intro*: *eventually-True*)
also have ... = (*SUP P* : {*P*. *eventually P F*}. *INFIMUM* (*g* ‘ *Collect P*) *f*)
by (*intro SUP-cong refl continuous-at-Inf-mono*[*OF am continuous-on-imp-continuous-within*[*OF c*]])

(*auto dest*!: *eventually-happens simp*: *F*)
finally show *?thesis* **by** (*auto simp*: *Liminf-def*)
qed

lemma *Limsup-compose-continuous-mono*:
fixes *f* :: *'a*::{*complete-linorder*, *linorder-topology*} ⇒ *'b*::{*complete-linorder*, *linorder-topology*}
assumes *c*: *continuous-on UNIV f* **and** *am*: *mono f* **and** *F*: *F* ≠ *bot*
shows *Limsup F* (λ*n*. *f* (*g n*)) = *f* (*Limsup F g*)
proof −
{ fix *P* **assume** *eventually P F*
have ∃ *x*. *P x*
proof (*rule ccontr*)
assume ¬ (∃ *x*. *P x*) **then have** *P* = (λ*x*. *False*)
by *auto*
with ‹*eventually P F*› *F* **show** *False*
by *auto*
qed }
note ∗ = *this*

have *f* (*Limsup F g*) = (*INF P* : {*P*. *eventually P F*}. *f* (*Sup* (*g* ' *Collect P*)))
unfolding *Limsup-def*
by (*subst continuous-at-Inf-mono*[*OF am continuous-on-imp-continuous-within*[*OF c*]])
(*auto intro*: *eventually-True*)
also have . . . = (*INF P* : {*P*. *eventually P F*}. *SUPREMUM* (*g* ' *Collect P*) *f*)
by (*intro INF-cong refl continuous-at-Sup-mono*[*OF am continuous-on-imp-continuous-within*[*OF c*]])
(*auto dest*!: *eventually-happens simp*: *F*)
finally show *?thesis* **by** (*auto simp*: *Limsup-def*)
qed

lemma *Liminf-compose-continuous-antimono*:
fixes *f* :: *'a*::{*complete-linorder*,*linorder-topology*} ⇒ *'b*::{*complete-linorder*,*linorder-topology*}
assumes *c*: *continuous-on UNIV f*
and *am*: *antimono f*
and *F*: *F* ≠ *bot*
shows *Liminf F* (λ*n*. *f* (*g n*)) = *f* (*Limsup F g*)
proof −
have ∗: ∃ *x*. *P x* **if** *eventually P F* **for** *P*
proof (*rule ccontr*)
assume ¬ (∃ *x*. *P x*) **then have** *P* = (λ*x*. *False*)
by *auto*
with ‹*eventually P F*› *F* **show** *False*
by *auto*
qed
have *f* (*Limsup F g*) = (*SUP P* : {*P*. *eventually P F*}. *f* (*Sup* (*g* ' *Collect P*)))
unfolding *Limsup-def*
by (*subst continuous-at-Inf-antimono*[*OF am continuous-on-imp-continuous-within*[*OF*

c]])
(*auto intro*: *eventually-True*)
also have ... = (*SUP P* : {*P*. *eventually P F*}. *INFIMUM* (*g* ' *Collect P*) *f*)
by (*intro SUP-cong refl continuous-at-Sup-antimono*[*OF am continuous-on-imp-continuous-within*[*OF c*]])
(*auto dest*!: *eventually-happens simp*: *F*)
finally show *?thesis*
by (*auto simp*: *Liminf-def*)
qed

lemma *Limsup-compose-continuous-antimono*:
fixes *f* :: *'a*::{*complete-linorder*, *linorder-topology*} ⇒ *'b*::{*complete-linorder*, *linorder-topology*}
assumes *c*: *continuous-on UNIV f* **and** *am*: *antimono f* **and** *F*: *F* ≠ *bot*
shows *Limsup F* (λ*n*. *f* (*g n*)) = *f* (*Liminf F g*)
proof −
{ fix *P* **assume** *eventually P F*
have ∃ *x*. *P x*
proof (*rule ccontr*)
assume ¬ (∃ *x*. *P x*) **then have** *P* = (λ*x*. *False*)
by *auto*
with ‹*eventually P F*› *F* **show** *False*
by *auto*
qed }
note ∗ = *this*

have *f* (*Liminf F g*) = (*INF P* : {*P*. *eventually P F*}. *f* (*Inf* (*g* ' *Collect P*)))
unfolding *Liminf-def*
by (*subst continuous-at-Sup-antimono*[*OF am continuous-on-imp-continuous-within*[*OF c*]])
(*auto intro*: *eventually-True*)
also have ... = (*INF P* : {*P*. *eventually P F*}. *SUPREMUM* (*g* ' *Collect P*) *f*)
by (*intro INF-cong refl continuous-at-Inf-antimono*[*OF am continuous-on-imp-continuous-within*[*OF c*]])
(*auto dest*!: *eventually-happens simp*: *F*)
finally show *?thesis*
by (*auto simp*: *Limsup-def*)
qed

32.1 More Limits

lemma *convergent-limsup-cl*:
fixes *X* :: *nat* ⇒ *'a*::{*complete-linorder*,*linorder-topology*}
shows *convergent X* ⟹ *limsup X* = *lim X*
by (*auto simp*: *convergent-def limI lim-imp-Limsup*)

lemma *convergent-liminf-cl*:
fixes *X* :: *nat* ⇒ *'a*::{*complete-linorder*,*linorder-topology*}
shows *convergent X* ⟹ *liminf X* = *lim X*

by (*auto simp*: *convergent-def limI lim-imp-Liminf*)

lemma *lim-increasing-cl*:
assumes $\bigwedge n\ m.\ n \geq m \Longrightarrow f\ n \geq f\ m$
obtains l **where** $f \longrightarrow (l::'a::\{complete\text{-}linorder,linorder\text{-}topology\})$
proof
show $f \longrightarrow (SUP\ n.\ f\ n)$
using *assms*
by (*intro increasing-tendsto*)
(*auto simp*: *SUP-upper eventually-sequentially less-SUP-iff intro*: *less-le-trans*)
qed

lemma *lim-decreasing-cl*:
assumes $\bigwedge n\ m.\ n \geq m \Longrightarrow f\ n \leq f\ m$
obtains l **where** $f \longrightarrow (l::'a::\{complete\text{-}linorder,linorder\text{-}topology\})$
proof
show $f \longrightarrow (INF\ n.\ f\ n)$
using *assms*
by (*intro decreasing-tendsto*)
(*auto simp*: *INF-lower eventually-sequentially INF-less-iff intro*: *le-less-trans*)
qed

lemma *compact-complete-linorder*:
fixes $X :: nat \Rightarrow 'a::\{complete\text{-}linorder,linorder\text{-}topology\}$
shows $\exists\ l\ r.\ subseq\ r \wedge (X \circ r) \longrightarrow l$
proof −
obtain r **where** *subseq r* **and** *mono*: *monoseq* $(X \circ r)$
using *seq-monosub*[*of X*]
unfolding *comp-def*
by *auto*
then have $(\forall\ n\ m.\ m \leq n \longrightarrow (X \circ r)\ m \leq (X \circ r)\ n) \vee (\forall\ n\ m.\ m \leq n \longrightarrow (X \circ r)\ n \leq (X \circ r)\ m)$
by (*auto simp add*: *monoseq-def*)
then obtain l **where** $(X \circ r) \longrightarrow l$
using *lim-increasing-cl*[*of X* ∘ *r*] *lim-decreasing-cl*[*of X* ∘ *r*]
by *auto*
then show *?thesis*
using ‹*subseq r*› **by** *auto*
qed

lemma *tendsto-Limsup*:
fixes $f :: \text{-} \Rightarrow 'a :: \{complete\text{-}linorder,linorder\text{-}topology\}$
shows $F \neq bot \Longrightarrow Limsup\ F\ f = Liminf\ F\ f \Longrightarrow (f \longrightarrow Limsup\ F\ f)\ F$
by (*subst tendsto-iff-Liminf-eq-Limsup*) *auto*

lemma *tendsto-Liminf*:
fixes $f :: \text{-} \Rightarrow 'a :: \{complete\text{-}linorder,linorder\text{-}topology\}$
shows $F \neq bot \Longrightarrow Limsup\ F\ f = Liminf\ F\ f \Longrightarrow (f \longrightarrow Liminf\ F\ f)\ F$
by (*subst tendsto-iff-Liminf-eq-Limsup*) *auto*

end

33 Extended real number line

theory *Extended-Real*
imports *Complex-Main Extended-Nat Liminf-Limsup*
begin

This should be part of *Extended-Nat* or *Order-Continuity*, but then the AFP-entry *Jinja-Thread* fails, as it does overload certain named from *Complex-Main*.

lemma *incseq-setsumI2*:
  fixes f :: $'i \Rightarrow nat \Rightarrow 'a$::*ordered-comm-monoid-add*
  shows $(\bigwedge n.\ n \in A \Longrightarrow mono\ (f\ n)) \Longrightarrow mono\ (\lambda i.\ \sum n{\in}A.\ f\ n\ i)$
  unfolding *incseq-def* **by** (*auto intro*: *setsum-mono*)

lemma *incseq-setsumI*:
  fixes f :: $nat \Rightarrow 'a$::*ordered-comm-monoid-add*
  assumes $\bigwedge i.\ 0 \le f\ i$
  shows *incseq* ($\lambda i.$ *setsum* f {..< i})
proof (*intro incseq-SucI*)
  fix n
  have *setsum* f {..< n} + $0 \le$ *setsum* f {..<n} + $f\ n$
    using *assms* **by** (*rule add-left-mono*)
  then show *setsum* f {..< n} $\le$ *setsum* f {..< *Suc* n}
    by *auto*
qed

lemma *continuous-at-left-imp-sup-continuous*:
  fixes f :: $'a$::{*complete-linorder*, *linorder-topology*} $\Rightarrow$ $'b$::{*complete-linorder*, *linorder-topology*}
  assumes *mono* f $\bigwedge x.$ *continuous* (*at-left* x) f
  shows *sup-continuous* f
  unfolding *sup-continuous-def*
proof *safe*
  fix M :: $nat \Rightarrow 'a$ **assume** *incseq* M **then show** f (*SUP* i. M i) = (*SUP* i. f (M i))
    using *continuous-at-Sup-mono*[*OF assms*, *of range M*] **by** *simp*
qed

lemma *sup-continuous-at-left*:
  fixes f :: $'a$::{*complete-linorder*, *linorder-topology*, *first-countable-topology*} $\Rightarrow$
    $'b$::{*complete-linorder*, *linorder-topology*}
  assumes f: *sup-continuous* f
  shows *continuous* (*at-left* x) f
proof *cases*
  assume x = *bot* **then show** *?thesis*
    by (*simp add*: *trivial-limit-at-left-bot*)

next
assume *x*: $x \neq bot$
show *?thesis*
unfolding *continuous-within*
proof (*intro tendsto-at-left-sequentially*[*of bot*])
fix *S* :: *nat* ⇒ *'a* **assume** *S*: *incseq S* **and** *S-x*: $S \longrightarrow x$
from *S-x* **have** *x-eq*: $x = (SUP\ i.\ S\ i)$
by (*rule LIMSEQ-unique*) (*intro LIMSEQ-SUP S*)
show $(\lambda n.\ f\ (S\ n)) \longrightarrow f\ x$
unfolding *x-eq sup-continuousD*[*OF f S*]
using *S sup-continuous-mono*[*OF f*] **by** (*intro LIMSEQ-SUP*) (*auto simp*: *mono-def*)
qed (*insert x*, *auto simp*: *bot-less*)
qed

lemma *sup-continuous-iff-at-left*:
fixes *f* :: *'a*::{*complete-linorder*, *linorder-topology*, *first-countable-topology*} ⇒ *'b*::{*complete-linorder*, *linorder-topology*}
shows *sup-continuous f* ⟷ (∀ *x*. *continuous* (*at-left x*) *f*) ∧ *mono f*
using *sup-continuous-at-left*[*of f*] *continuous-at-left-imp-sup-continuous*[*of f*] *sup-continuous-mono*[*of f*] **by** *auto*

lemma *continuous-at-right-imp-inf-continuous*:
fixes *f* :: *'a*::{*complete-linorder*, *linorder-topology*} ⇒ *'b*::{*complete-linorder*, *linorder-topology*}
assumes *mono f* ⋀*x*. *continuous* (*at-right x*) *f*
shows *inf-continuous f*
unfolding *inf-continuous-def*
proof *safe*
fix *M* :: *nat* ⇒ *'a* **assume** *decseq M* **then show** $f\ (INF\ i.\ M\ i) = (INF\ i.\ f\ (M\ i))$
using *continuous-at-Inf-mono*[*OF assms*, *of range M*] **by** *simp*
qed

lemma *inf-continuous-at-right*:
fixes *f* :: *'a*::{*complete-linorder*, *linorder-topology*, *first-countable-topology*} ⇒ *'b*::{*complete-linorder*, *linorder-topology*}
assumes *f*: *inf-continuous f*
shows *continuous* (*at-right x*) *f*
proof *cases*
assume $x = top$ **then show** *?thesis*
by (*simp add*: *trivial-limit-at-right-top*)
next
assume *x*: $x \neq top$
show *?thesis*
unfolding *continuous-within*
proof (*intro tendsto-at-right-sequentially*[*of - top*])
fix *S* :: *nat* ⇒ *'a* **assume** *S*: *decseq S* **and** *S-x*: $S \longrightarrow x$
from *S-x* **have** *x-eq*: $x = (INF\ i.\ S\ i)$

by (*rule LIMSEQ-unique*) (*intro LIMSEQ-INF S*)
show ($\lambda n.\ f\ (S\ n)$) $\longrightarrow$ *f x*
unfolding *x-eq inf-continuousD*[*OF f S*]
using *S inf-continuous-mono*[*OF f*] **by** (*intro LIMSEQ-INF*) (*auto simp*: *mono-def antimono-def*)
qed (*insert x*, *auto simp*: *less-top*)
qed

lemma *inf-continuous-iff-at-right*:
fixes *f* :: *'a*::{*complete-linorder*, *linorder-topology*, *first-countable-topology*} ⇒ *'b*::{*complete-linorder*, *linorder-topology*}
shows *inf-continuous f* ⟷ (∀ *x*. *continuous* (*at-right x*) *f*) ∧ *mono f*
using *inf-continuous-at-right*[*of f*] *continuous-at-right-imp-inf-continuous*[*of f*] *inf-continuous-mono*[*of f*] **by** *auto*

instantiation *enat* :: *linorder-topology*
begin

definition *open-enat* :: *enat set* ⇒ *bool* **where**
open-enat = *generate-topology* (*range lessThan* ∪ *range greaterThan*)

instance
proof qed (*rule open-enat-def*)

end

lemma *open-enat*: *open* {*enat n*}
proof (*cases n*)
case *0*
then have {*enat n*} = {..< *eSuc 0*}
by (*auto simp*: *enat-0*)
then show *?thesis*
by *simp*
next
case (*Suc n'*)
then have {*enat n*} = {*enat n'* <..< *enat* (*Suc n*)}
apply *auto*
apply (*case-tac x*)
apply *auto*
done
then show *?thesis*
by *simp*
qed

lemma *open-enat-iff*:
fixes *A* :: *enat set*
shows *open A* ⟷ (∞ ∈ *A* ⟶ (∃ *n*::*nat*. {*n* <..} ⊆ *A*))
proof *safe*
assume ∞ ∉ *A*

```
  then have A = (⋃n∈{n. enat n ∈ A}. {enat n})
    apply auto
    apply (case-tac x)
    apply auto
    done
  moreover have open ...
    by (auto intro: open-enat)
  ultimately show open A
    by simp
next
  fix n assume {enat n <..} ⊆ A
  then have A = (⋃n∈{n. enat n ∈ A}. {enat n}) ∪ {enat n <..}
    apply auto
    apply (case-tac x)
    apply auto
    done
  moreover have open ...
    by (intro open-Un open-UN ballI open-enat open-greaterThan)
  ultimately show open A
    by simp
next
  assume open A ∞ ∈ A
  then have generate-topology (range lessThan ∪ range greaterThan) A ∞ ∈ A
    unfolding open-enat-def by auto
  then show ∃n::nat. {n <..} ⊆ A
  proof induction
    case (Int A B)
    then obtain n m where {enat n<..} ⊆ A {enat m<..} ⊆ B
      by auto
    then have {enat (max n m) <..} ⊆ A ∩ B
       by (auto simp add: subset-eq Ball-def max-def enat-ord-code(1)[symmetric]
simp del: enat-ord-code(1))
    then show ?case
      by auto
  next
    case (UN K)
    then obtain k where k ∈ K ∞ ∈ k
      by auto
    with UN.IH[OF this] show ?case
      by auto
  qed auto
qed

lemma nhds-enat: nhds x = (if x = ∞ then INF i. principal {enat i..} else prin-
cipal {x})
proof auto
  show nhds ∞ = (INF i. principal {enat i..})
    unfolding nhds-def
   apply (auto intro!: antisym INF-greatest simp add: open-enat-iff cong: rev-conj-cong)
```

apply (*auto intro*!: *INF-lower Ioi-le-Ico*) []
subgoal for *x i*
by (*auto intro*!: *INF-lower2*[*of Suc i*] *simp*: *subset-eq Ball-def eSuc-enat Suc-ile-eq*)
done
show *nhds* (*enat i*) = *principal* {*enat i*} **for** *i*
by (*simp add*: *nhds-discrete-open open-enat*)
qed

instance *enat* :: *topological-comm-monoid-add*
proof
have [*simp*]: *enat i* ≤ *aa* ⟹ *enat i* ≤ *aa* + *ba* **for** *aa ba i*
by (*rule order-trans*[*OF - add-mono*[*of aa aa 0 ba*]]) *auto*
then have [*simp*]: *enat i* ≤ *ba* ⟹ *enat i* ≤ *aa* + *ba* **for** *aa ba i*
by (*metis add.commute*)
fix *a b* :: *enat* **show** ((λ*x*. *fst x* + *snd x*) ⟶ *a* + *b*) (*nhds a* $\times_F$ *nhds b*)
apply (*auto simp*: *nhds-enat filterlim-INF prod-filter-INF1 prod-filter-INF2 filterlim-principal principal-prod-principal eventually-principal*)
subgoal for *i*
by (*auto intro*!: *eventually-INF1*[*of i*] *simp*: *eventually-principal*)
subgoal for *j i*
by (*auto intro*!: *eventually-INF1*[*of i*] *simp*: *eventually-principal*)
subgoal for *j i*
by (*auto intro*!: *eventually-INF1*[*of i*] *simp*: *eventually-principal*)
done
qed

For more lemmas about the extended real numbers go to `~~/src/HOL/Multivariate_Analysis/Extended_Real_Limits.thy`

33.1 Definition and basic properties

datatype *ereal* = *ereal real* | *PInfty* | *MInfty*

instantiation *ereal* :: *uminus*
begin

fun *uminus-ereal* **where**
− (*ereal r*) = *ereal* (− *r*)
| − *PInfty* = *MInfty*
| − *MInfty* = *PInfty*

instance ..

end

instantiation *ereal* :: *infinity*
begin

definition (∞::*ereal*) = *PInfty*
instance ..

end

declare [[*coercion ereal* :: *real* ⇒ *ereal*]]

lemma *ereal-uminus-uminus*[*simp*]:
  fixes *a* :: *ereal*
  shows − (− *a*) = *a*
  by (*cases a*) *simp-all*

lemma
  shows *PInfty-eq-infinity*[*simp*]: *PInfty* = ∞
    and *MInfty-eq-minfinity*[*simp*]: *MInfty* = − ∞
    and *MInfty-neq-PInfty*[*simp*]: ∞ ≠ − (∞::*ereal*) − ∞ ≠ (∞::*ereal*)
    and *MInfty-neq-ereal*[*simp*]: *ereal r* ≠ − ∞ − ∞ ≠ *ereal r*
    and *PInfty-neq-ereal*[*simp*]: *ereal r* ≠ ∞ ∞ ≠ *ereal r*
    and *PInfty-cases*[*simp*]: (*case* ∞ *of ereal r* ⇒ *f r* | *PInfty* ⇒ *y* | *MInfty* ⇒ *z*) = *y*
    and *MInfty-cases*[*simp*]: (*case* − ∞ *of ereal r* ⇒ *f r* | *PInfty* ⇒ *y* | *MInfty* ⇒ *z*) = *z*
  by (*simp-all add*: *infinity-ereal-def*)

declare
  PInfty-eq-infinity[*code-post*]
  MInfty-eq-minfinity[*code-post*]

lemma [*code-unfold*]:
  ∞ = *PInfty*
  − *PInfty* = *MInfty*
  by *simp-all*

lemma *inj-ereal*[*simp*]: *inj-on ereal A*
  unfolding *inj-on-def* **by** *auto*

lemma *ereal-cases*[*cases type*: *ereal*]:
  obtains (*real*) *r* **where** *x* = *ereal r*
    | (*PInf*) *x* = ∞
    | (*MInf*) *x* = −∞
  using *assms* **by** (*cases x*) *auto*

lemmas *ereal2-cases* = *ereal-cases*[*case-product ereal-cases*]
lemmas *ereal3-cases* = *ereal2-cases*[*case-product ereal-cases*]

lemma *ereal-all-split*: $\bigwedge P.\ (\forall x{::}ereal.\ P\ x) \longleftrightarrow P\ \infty \wedge (\forall x.\ P\ (ereal\ x)) \wedge P\ (-\infty)$
  by (*metis ereal-cases*)

lemma *ereal-ex-split*: $\bigwedge P. (\exists x::ereal. P\ x) \longleftrightarrow P\ \infty \lor (\exists x. P\ (ereal\ x)) \lor P\ (-\infty)$
  by (*metis ereal-cases*)

lemma *ereal-uminus-eq-iff* [*simp*]:
  fixes *a b* :: *ereal*
  shows $-a = -b \longleftrightarrow a = b$
  by (*cases rule*: *ereal2-cases*[*of a b*]) *simp-all*

function *real-of-ereal* :: *ereal* ⇒ *real* **where**
  real-of-ereal (*ereal r*) = *r*
| *real-of-ereal* ∞ = *0*
| *real-of-ereal* (−∞) = *0*
  by (*auto intro*: *ereal-cases*)
termination by *standard* (*rule wf-empty*)

lemma *real-of-ereal*[*simp*]:
  real-of-ereal (− *x* :: *ereal*) = − (*real-of-ereal x*)
  by (*cases x*) *simp-all*

lemma *range-ereal*[*simp*]: *range ereal* = *UNIV* − {∞, −∞}
proof *safe*
  fix *x*
  assume $x \notin range\ ereal\ x \neq \infty$
  then show $x = -\infty$
    by (*cases x*) *auto*
qed *auto*

lemma *ereal-range-uminus*[*simp*]: *range uminus* = (*UNIV*::*ereal set*)
proof *safe*
  fix *x* :: *ereal*
  show $x \in range\ uminus$
    by (*intro image-eqI*[*of - - −x*]) *auto*
qed *auto*

instantiation *ereal* :: *abs*
begin

function *abs-ereal* **where**
  |*ereal r*| = *ereal* |*r*|
| |−∞| = (∞::*ereal*)
| |∞| = (∞::*ereal*)
by (*auto intro*: *ereal-cases*)
termination proof qed (*rule wf-empty*)

instance ..

end

lemma *abs-eq-infinity-cases*[*elim*!]:
fixes x :: *ereal*
assumes $|x| = \infty$
obtains $x = \infty \mid x = -\infty$
using *assms* **by** (*cases* x) *auto*

lemma *abs-neq-infinity-cases*[*elim*!]:
fixes x :: *ereal*
assumes $|x| \neq \infty$
obtains r **where** $x = \mathit{ereal}\ r$
using *assms* **by** (*cases* x) *auto*

lemma *abs-ereal-uminus*[*simp*]:
fixes x :: *ereal*
shows $|-\ x| = |x|$
by (*cases* x) *auto*

lemma *ereal-infinity-cases*:
fixes a :: *ereal*
shows $a \neq \infty \Longrightarrow a \neq -\infty \Longrightarrow |a| \neq \infty$
by *auto*

33.1.1 Addition

instantiation *ereal* :: {*one*,*comm-monoid-add*,*zero-neq-one*}
begin

definition $0 = \mathit{ereal}\ 0$
definition $1 = \mathit{ereal}\ 1$

function *plus-ereal* **where**
$\mathit{ereal}\ r + \mathit{ereal}\ p = \mathit{ereal}\ (r + p)$
| $\infty + a = (\infty{::}\mathit{ereal})$
| $a + \infty = (\infty{::}\mathit{ereal})$
| $\mathit{ereal}\ r + -\infty = -\ \infty$
| $-\infty + \mathit{ereal}\ p = -(\infty{::}\mathit{ereal})$
| $-\infty + -\infty = -(\infty{::}\mathit{ereal})$
proof *goal-cases*
case *prems*: (*1 P x*)
then obtain *a b* **where** $x = (a,\ b)$
by (*cases* x) *auto*
with *prems* **show** P
by (*cases rule*: *ereal2-cases*[*of a b*]) *auto*
qed *auto*
termination by *standard* (*rule wf-empty*)

lemma *Infty-neq-0*[*simp*]:
$(\infty{::}\mathit{ereal}) \neq 0$ $0 \neq (\infty{::}\mathit{ereal})$
$-(\infty{::}\mathit{ereal}) \neq 0$ $0 \neq -(\infty{::}\mathit{ereal})$

by (*simp-all add*: *zero-ereal-def*)

lemma *ereal-eq-0*[*simp*]:
ereal r = *0* ⟷ *r* = *0*
0 = *ereal r* ⟷ *r* = *0*
unfolding *zero-ereal-def* **by** *simp-all*

lemma *ereal-eq-1*[*simp*]:
ereal r = *1* ⟷ *r* = *1*
1 = *ereal r* ⟷ *r* = *1*
unfolding *one-ereal-def* **by** *simp-all*

instance
proof
fix *a b c* :: *ereal*
show *0* + *a* = *a*
by (*cases a*) (*simp-all add*: *zero-ereal-def*)
show *a* + *b* = *b* + *a*
by (*cases rule*: *ereal2-cases*[*of a b*]) *simp-all*
show *a* + *b* + *c* = *a* + (*b* + *c*)
by (*cases rule*: *ereal3-cases*[*of a b c*]) *simp-all*
show *0* ≠ (*1*::*ereal*)
by (*simp add*: *one-ereal-def zero-ereal-def*)
qed

end

lemma *ereal-0-plus* [*simp*]: *ereal 0* + *x* = *x*
and *plus-ereal-0* [*simp*]: *x* + *ereal 0* = *x*
by(*simp-all add*: *zero-ereal-def*[*symmetric*])

instance *ereal* :: *numeral* ..

lemma *real-of-ereal-0*[*simp*]: *real-of-ereal* (*0*::*ereal*) = *0*
unfolding *zero-ereal-def* **by** *simp*

lemma *abs-ereal-zero*[*simp*]: |*0*| = (*0*::*ereal*)
unfolding *zero-ereal-def abs-ereal.simps* **by** *simp*

lemma *ereal-uminus-zero*[*simp*]: − *0* = (*0*::*ereal*)
by (*simp add*: *zero-ereal-def*)

lemma *ereal-uminus-zero-iff*[*simp*]:
fixes *a* :: *ereal*
shows −*a* = *0* ⟷ *a* = *0*
by (*cases a*) *simp-all*

lemma *ereal-plus-eq-PInfty*[*simp*]:
fixes *a b* :: *ereal*

shows $a + b = \infty \longleftrightarrow a = \infty \vee b = \infty$
by (*cases rule*: *ereal2-cases*[*of a b*]) *auto*

lemma *ereal-plus-eq-MInfty*[*simp*]:
fixes *a b* :: *ereal*
shows $a + b = -\infty \longleftrightarrow (a = -\infty \vee b = -\infty) \wedge a \neq \infty \wedge b \neq \infty$
by (*cases rule*: *ereal2-cases*[*of a b*]) *auto*

lemma *ereal-add-cancel-left*:
fixes *a b* :: *ereal*
assumes $a \neq -\infty$
shows $a + b = a + c \longleftrightarrow a = \infty \vee b = c$
using *assms* **by** (*cases rule*: *ereal3-cases*[*of a b c*]) *auto*

lemma *ereal-add-cancel-right*:
fixes *a b* :: *ereal*
assumes $a \neq -\infty$
shows $b + a = c + a \longleftrightarrow a = \infty \vee b = c$
using *assms* **by** (*cases rule*: *ereal3-cases*[*of a b c*]) *auto*

lemma *ereal-real*: *ereal* (*real-of-ereal x*) = (*if* $|x| = \infty$ *then 0 else x*)
by (*cases x*) *simp-all*

lemma *real-of-ereal-add*:
fixes *a b* :: *ereal*
shows *real-of-ereal* $(a + b) =$
(*if* $(|a| = \infty) \wedge (|b| = \infty) \vee (|a| \neq \infty) \wedge (|b| \neq \infty)$ *then real-of-ereal a* + *real-of-ereal b else 0*)
by (*cases rule*: *ereal2-cases*[*of a b*]) *auto*

33.1.2 Linear order on *ereal*

instantiation *ereal* :: *linorder*
begin

function *less-ereal*
where
ereal x < *ereal y* $\longleftrightarrow$ $x < y$
| (∞::*ereal*) < *a* $\longleftrightarrow$ *False*
| *a* < $-$(∞::*ereal*) $\longleftrightarrow$ *False*
| *ereal x* < ∞ $\longleftrightarrow$ *True*
| $-\infty$ < *ereal r* $\longleftrightarrow$ *True*
| $-\infty$ < (∞::*ereal*) $\longleftrightarrow$ *True*
proof *goal-cases*
case *prems*: (*1 P x*)
then obtain *a b* **where** *x* = (*a*,*b*) **by** (*cases x*) *auto*
with *prems* **show** *P* **by** (*cases rule*: *ereal2-cases*[*of a b*]) *auto*
qed *simp-all*
termination by (*relation* {}) *simp*

definition $x \leq (y{::}ereal) \longleftrightarrow x < y \vee x = y$

lemma *ereal-infty-less*[*simp*]:
fixes x :: *ereal*
shows $x < \infty \longleftrightarrow (x \neq \infty)$
$-\infty < x \longleftrightarrow (x \neq -\infty)$
by (*cases x*, *simp-all*) (*cases x*, *simp-all*)

lemma *ereal-infty-less-eq*[*simp*]:
fixes x :: *ereal*
shows $\infty \leq x \longleftrightarrow x = \infty$
and $x \leq -\infty \longleftrightarrow x = -\infty$
by (*auto simp add*: *less-eq-ereal-def*)

lemma *ereal-less*[*simp*]:
ereal $r < 0 \longleftrightarrow (r < 0)$
$0 <$ *ereal* $r \longleftrightarrow (0 < r)$
ereal $r < 1 \longleftrightarrow (r < 1)$
$1 <$ *ereal* $r \longleftrightarrow (1 < r)$
$0 < (\infty{::}ereal)$
$-(\infty{::}ereal) < 0$
by (*simp-all add*: *zero-ereal-def one-ereal-def*)

lemma *ereal-less-eq*[*simp*]:
$x \leq (\infty{::}ereal)$
$-(\infty{::}ereal) \leq x$
ereal $r \leq$ *ereal* $p \longleftrightarrow r \leq p$
ereal $r \leq 0 \longleftrightarrow r \leq 0$
$0 \leq$ *ereal* $r \longleftrightarrow 0 \leq r$
ereal $r \leq 1 \longleftrightarrow r \leq 1$
$1 \leq$ *ereal* $r \longleftrightarrow 1 \leq r$
by (*auto simp add*: *less-eq-ereal-def zero-ereal-def one-ereal-def*)

lemma *ereal-infty-less-eq2*:
$a \leq b \Longrightarrow a = \infty \Longrightarrow b = (\infty{::}ereal)$
$a \leq b \Longrightarrow b = -\infty \Longrightarrow a = -(\infty{::}ereal)$
by *simp-all*

instance
proof
fix $x\ y\ z$ :: *ereal*
show $x \leq x$
by (*cases x*) *simp-all*
show $x < y \longleftrightarrow x \leq y \wedge \neg\, y \leq x$
by (*cases rule*: *ereal2-cases*[*of x y*]) *auto*
show $x \leq y \vee y \leq x$
by (*cases rule*: *ereal2-cases*[*of x y*]) *auto*
{

assume $x \leq y$ $y \leq x$
then show $x = y$
by (*cases rule*: *ereal2-cases*[*of* x y]) *auto*
}
{
assume $x \leq y$ $y \leq z$
then show $x \leq z$
by (*cases rule*: *ereal3-cases*[*of* x y z]) *auto*
}
qed

end

lemma *ereal-dense2*: $x < y \Longrightarrow \exists z.\ x < \mathit{ereal}\ z \wedge \mathit{ereal}\ z < y$
using *lt-ex gt-ex dense* **by** (*cases* x y *rule*: *ereal2-cases*) *auto*

instance *ereal* :: *dense-linorder*
by *standard* (*blast dest*: *ereal-dense2*)

instance *ereal* :: *ordered-comm-monoid-add*
proof
fix a b c :: *ereal*
assume $a \leq b$
then show $c + a \leq c + b$
by (*cases rule*: *ereal3-cases*[*of* a b c]) *auto*
qed

lemma *ereal-one-not-less-zero-ereal*[*simp*]: $\neg\ 1 < (0::\mathit{ereal})$
by (*simp add*: *zero-ereal-def*)

lemma *real-of-ereal-positive-mono*:
fixes x y :: *ereal*
shows $0 \leq x \Longrightarrow x \leq y \Longrightarrow y \neq \infty \Longrightarrow \mathit{real\text{-}of\text{-}ereal}\ x \leq \mathit{real\text{-}of\text{-}ereal}\ y$
by (*cases rule*: *ereal2-cases*[*of* x y]) *auto*

lemma *ereal-MInfty-lessI*[*intro*, *simp*]:
fixes a :: *ereal*
shows $a \neq -\infty \Longrightarrow -\infty < a$
by (*cases* a) *auto*

lemma *ereal-less-PInfty*[*intro*, *simp*]:
fixes a :: *ereal*
shows $a \neq \infty \Longrightarrow a < \infty$
by (*cases* a) *auto*

lemma *ereal-less-ereal-Ex*:
fixes a b :: *ereal*
shows $x < \mathit{ereal}\ r \longleftrightarrow x = -\infty \vee (\exists p.\ p < r \wedge x = \mathit{ereal}\ p)$
by (*cases* x) *auto*

lemma *less-PInf-Ex-of-nat*: $x \neq \infty \longleftrightarrow (\exists\, n{::}nat.\ x < ereal\ (real\ n))$
proof (*cases x*)
  case (*real r*)
  then show *?thesis*
    using *reals-Archimedean2*[*of r*] **by** *simp*
qed *simp-all*

lemma *ereal-add-mono*:
  fixes *a b c d* :: *ereal*
  assumes $a \leq b$
    and $c \leq d$
  shows $a + c \leq b + d$
  using *assms*
  apply (*cases a*)
  apply (*cases rule*: *ereal3-cases*[*of b c d*], *auto*)
  apply (*cases rule*: *ereal3-cases*[*of b c d*], *auto*)
  done

lemma *ereal-minus-le-minus*[*simp*]:
  fixes *a b* :: *ereal*
  shows $-\ a \leq -\ b \longleftrightarrow b \leq a$
  by (*cases rule*: *ereal2-cases*[*of a b*]) *auto*

lemma *ereal-minus-less-minus*[*simp*]:
  fixes *a b* :: *ereal*
  shows $-\ a < -\ b \longleftrightarrow b < a$
  by (*cases rule*: *ereal2-cases*[*of a b*]) *auto*

lemma *ereal-le-real-iff*:
  $x \leq real\text{-}of\text{-}ereal\ y \longleftrightarrow (|y| \neq \infty \longrightarrow ereal\ x \leq y) \wedge (|y| = \infty \longrightarrow x \leq 0)$
  by (*cases y*) *auto*

lemma *real-le-ereal-iff*:
  $real\text{-}of\text{-}ereal\ y \leq x \longleftrightarrow (|y| \neq \infty \longrightarrow y \leq ereal\ x) \wedge (|y| = \infty \longrightarrow 0 \leq x)$
  by (*cases y*) *auto*

lemma *ereal-less-real-iff*:
  $x < real\text{-}of\text{-}ereal\ y \longleftrightarrow (|y| \neq \infty \longrightarrow ereal\ x < y) \wedge (|y| = \infty \longrightarrow x < 0)$
  by (*cases y*) *auto*

lemma *real-less-ereal-iff*:
  $real\text{-}of\text{-}ereal\ y < x \longleftrightarrow (|y| \neq \infty \longrightarrow y < ereal\ x) \wedge (|y| = \infty \longrightarrow 0 < x)$
  by (*cases y*) *auto*

lemma *real-of-ereal-pos*:
  fixes *x* :: *ereal*
  shows $0 \leq x \Longrightarrow 0 \leq real\text{-}of\text{-}ereal\ x$ **by** (*cases x*) *auto*

lemmas *real-of-ereal-ord-simps* =
ereal-le-real-iff real-le-ereal-iff ereal-less-real-iff real-less-ereal-iff

lemma *abs-ereal-ge0*[*simp*]: $0 \leq x \Longrightarrow |x :: ereal| = x$
by (*cases x*) *auto*

lemma *abs-ereal-less0*[*simp*]: $x < 0 \Longrightarrow |x :: ereal| = -x$
by (*cases x*) *auto*

lemma *abs-ereal-pos*[*simp*]: $0 \leq |x :: ereal|$
by (*cases x*) *auto*

lemma *ereal-abs-leI*:
fixes *x y* :: *ereal*
shows $[\![x \leq y; -x \leq y]\!] \Longrightarrow |x| \leq y$
by(*cases x y rule*: *ereal2-cases*)(*simp-all*)

lemma *real-of-ereal-le-0*[*simp*]: *real-of-ereal* $(x :: ereal) \leq 0 \longleftrightarrow x \leq 0 \vee x = \infty$
by (*cases x*) *auto*

lemma *abs-real-of-ereal*[*simp*]: |*real-of-ereal* (*x* :: *ereal*)| = *real-of-ereal* |*x*|
by (*cases x*) *auto*

lemma *zero-less-real-of-ereal*:
fixes *x* :: *ereal*
shows $0 <$ *real-of-ereal x* $\longleftrightarrow 0 < x \wedge x \neq \infty$
by (*cases x*) *auto*

lemma *ereal-0-le-uminus-iff*[*simp*]:
fixes *a* :: *ereal*
shows $0 \leq -\ a \longleftrightarrow a \leq 0$
by (*cases rule*: *ereal2-cases*[*of a*]) *auto*

lemma *ereal-uminus-le-0-iff*[*simp*]:
fixes *a* :: *ereal*
shows $-\ a \leq 0 \longleftrightarrow 0 \leq a$
by (*cases rule*: *ereal2-cases*[*of a*]) *auto*

lemma *ereal-add-strict-mono*:
fixes *a b c d* :: *ereal*
assumes $a \leq b$
and $0 \leq a$
and $a \neq \infty$
and $c < d$
shows $a + c < b + d$
using *assms*
by (*cases rule*: *ereal3-cases*[*case-product ereal-cases*, *of a b c d*]) *auto*

lemma *ereal-less-add*:

fixes *a b c* :: *ereal*
shows $|a| \neq \infty \Longrightarrow c < b \Longrightarrow a + c < a + b$
by (*cases rule*: *ereal2-cases*[*of b c*]) *auto*

lemma *ereal-add-nonneg-eq-0-iff*:
fixes *a b* :: *ereal*
shows $0 \leq a \Longrightarrow 0 \leq b \Longrightarrow a + b = 0 \longleftrightarrow a = 0 \wedge b = 0$
by (*cases a b rule*: *ereal2-cases*) *auto*

lemma *ereal-uminus-eq-reorder*: $- a = b \longleftrightarrow a = (-b::ereal)$
by *auto*

lemma *ereal-uminus-less-reorder*: $- a < b \longleftrightarrow -b < (a::ereal)$
by (*subst* (*3*) *ereal-uminus-uminus*[*symmetric*]) (*simp only*: *ereal-minus-less-minus*)

lemma *ereal-less-uminus-reorder*: $a < - b \longleftrightarrow b < - (a::ereal)$
by (*subst* (*3*) *ereal-uminus-uminus*[*symmetric*]) (*simp only*: *ereal-minus-less-minus*)

lemma *ereal-uminus-le-reorder*: $- a \leq b \longleftrightarrow -b \leq (a::ereal)$
by (*subst* (*3*) *ereal-uminus-uminus*[*symmetric*]) (*simp only*: *ereal-minus-le-minus*)

lemmas *ereal-uminus-reorder* =
ereal-uminus-eq-reorder ereal-uminus-less-reorder ereal-uminus-le-reorder

lemma *ereal-bot*:
fixes *x* :: *ereal*
assumes $\bigwedge B.\ x \leq ereal\ B$
shows $x = - \infty$
proof (*cases x*)
case (*real r*)
with *assms*[*of r* − *1*] **show** *?thesis*
by *auto*
next
case *PInf*
with *assms*[*of 0*] **show** *?thesis*
by *auto*
next
case *MInf*
then show *?thesis*
by *simp*
qed

lemma *ereal-top*:
fixes *x* :: *ereal*
assumes $\bigwedge B.\ x \geq ereal\ B$
shows $x = \infty$
proof (*cases x*)
case (*real r*)
with *assms*[*of r* + *1*] **show** *?thesis*

```
    by auto
next
  case MInf
  with assms[of 0] show ?thesis
    by auto
next
  case PInf
  then show ?thesis
    by simp
qed

lemma
  shows ereal-max[simp]: ereal (max x y) = max (ereal x) (ereal y)
    and ereal-min[simp]: ereal (min x y) = min (ereal x) (ereal y)
  by (simp-all add: min-def max-def)

lemma ereal-max-0: max 0 (ereal r) = ereal (max 0 r)
  by (auto simp: zero-ereal-def)

lemma
  fixes f :: nat ⇒ ereal
  shows ereal-incseq-uminus[simp]: incseq (λx. − f x) ⟷ decseq f
    and ereal-decseq-uminus[simp]: decseq (λx. − f x) ⟷ incseq f
  unfolding decseq-def incseq-def by auto

lemma incseq-ereal: incseq f ⟹ incseq (λx. ereal (f x))
  unfolding incseq-def by auto

lemma ereal-add-nonneg-nonneg[simp]:
  fixes a b :: ereal
  shows 0 ≤ a ⟹ 0 ≤ b ⟹ 0 ≤ a + b
  using add-mono[of 0 a 0 b] by simp

lemma setsum-ereal[simp]: (∑x∈A. ereal (f x)) = ereal (∑x∈A. f x)
proof (cases finite A)
  case True
  then show ?thesis by induct auto
next
  case False
  then show ?thesis by simp
qed

lemma setsum-Pinfty:
  fixes f :: 'a ⇒ ereal
  shows (∑x∈P. f x) = ∞ ⟷ finite P ∧ (∃i∈P. f i = ∞)
proof safe
  assume ∗: setsum f P = ∞
  show finite P
  proof (rule ccontr)
```

```
    assume ¬ finite P
    with ∗ show False
      by auto
  qed
  show ∃ i∈P. f i = ∞
  proof (rule ccontr)
    assume ¬ ?thesis
    then have ⋀i. i ∈ P ⟹ f i ≠ ∞
      by auto
    with ⟨finite P⟩ have setsum f P ≠ ∞
      by induct auto
    with ∗ show False
      by auto
  qed
next
  fix i
  assume finite P and i ∈ P and f i = ∞
  then show setsum f P = ∞
  proof induct
    case (insert x A)
    show ?case using insert by (cases x = i) auto
  qed simp
qed

lemma setsum-Inf:
  fixes f :: 'a ⇒ ereal
  shows |setsum f A| = ∞ ⟷ finite A ∧ (∃ i∈A. |f i| = ∞)
proof
  assume ∗: |setsum f A| = ∞
  have finite A
    by (rule ccontr) (insert ∗, auto)
  moreover have ∃ i∈A. |f i| = ∞
  proof (rule ccontr)
    assume ¬ ?thesis
    then have ∀ i∈A. ∃ r. f i = ereal r
      by auto
    from bchoice[OF this] obtain r where ∀ x∈A. f x = ereal (r x) ..
    with ∗ show False
      by auto
  qed
  ultimately show finite A ∧ (∃ i∈A. |f i| = ∞)
    by auto
next
  assume finite A ∧ (∃ i∈A. |f i| = ∞)
  then obtain i where finite A i ∈ A and |f i| = ∞
    by auto
  then show |setsum f A| = ∞
  proof induct
    case (insert j A)
```

then show *?case*
by (*cases rule*: *ereal3-cases*[*of f i f j setsum f A*]) *auto*
qed *simp*
qed

lemma *setsum-real-of-ereal*:
fixes f :: $'i \Rightarrow$ *ereal*
assumes $\bigwedge x.\ x \in S \Longrightarrow |f\ x| \neq \infty$
shows $(\sum x{\in}S.\ \textit{real-of-ereal}\ (f\ x)) = \textit{real-of-ereal}\ (\textit{setsum}\ f\ S)$
proof −
have $\forall x{\in}S.\ \exists r.\ f\ x = \textit{ereal}\ r$
proof
fix x
assume $x \in S$
from *assms*[*OF this*] **show** $\exists r.\ f\ x = \textit{ereal}\ r$
by (*cases f x*) *auto*
qed
from *bchoice*[*OF this*] **obtain** r **where** $\forall x{\in}S.\ f\ x = \textit{ereal}\ (r\ x)$ **..**
then show *?thesis*
by *simp*
qed

lemma *setsum-ereal-0*:
fixes f :: $'a \Rightarrow$ *ereal*
assumes *finite A*
and $\bigwedge i.\ i \in A \Longrightarrow 0 \leq f\ i$
shows $(\sum x{\in}A.\ f\ x) = 0 \longleftrightarrow (\forall i{\in}A.\ f\ i = 0)$
proof
assume $\textit{setsum}\ f\ A = 0$ **with** *assms* **show** $\forall i{\in}A.\ f\ i = 0$
proof (*induction A*)
case (*insert a A*)
then have $f\ a = 0 \wedge (\sum a{\in}A.\ f\ a) = 0$
by (*subst ereal-add-nonneg-eq-0-iff*[*symmetric*]) (*simp-all add*: *setsum-nonneg*)
with *insert* **show** *?case*
by *simp*
qed *simp*
qed *auto*

33.1.3 Multiplication

instantiation *ereal* :: {*comm-monoid-mult*,*sgn*}
begin

function *sgn-ereal* :: *ereal* ⇒ *ereal* **where**
sgn (*ereal r*) = *ereal* (*sgn r*)
| *sgn* (∞::*ereal*) = *1*
| *sgn* (−∞::*ereal*) = −*1*
by (*auto intro*: *ereal-cases*)
termination by *standard* (*rule wf-empty*)

function *times-ereal* **where**
  ereal r ∗ *ereal p* = *ereal* (*r* ∗ *p*)
| *ereal r* ∗ ∞ = (*if r* = *0 then 0 else if r* > *0 then* ∞ *else* −∞)
| ∞ ∗ *ereal r* = (*if r* = *0 then 0 else if r* > *0 then* ∞ *else* −∞)
| *ereal r* ∗ −∞ = (*if r* = *0 then 0 else if r* > *0 then* −∞ *else* ∞)
| −∞ ∗ *ereal r* = (*if r* = *0 then 0 else if r* > *0 then* −∞ *else* ∞)
| (∞::*ereal*) ∗ ∞ = ∞
| −(∞::*ereal*) ∗ ∞ = −∞
| (∞::*ereal*) ∗ −∞ = −∞
| −(∞::*ereal*) ∗ −∞ = ∞
proof *goal-cases*
  case *prems*: (*1 P x*)
  then obtain *a b* **where** *x* = (*a*, *b*)
    by (*cases x*) *auto*
  with *prems* **show** *P*
    by (*cases rule*: *ereal2-cases*[*of a b*]) *auto*
qed *simp-all*
termination by (*relation* {}) *simp*

instance
proof
  fix *a b c* :: *ereal*
  show *1* ∗ *a* = *a*
    by (*cases a*) (*simp-all add*: *one-ereal-def*)
  show *a* ∗ *b* = *b* ∗ *a*
    by (*cases rule*: *ereal2-cases*[*of a b*]) *simp-all*
  show *a* ∗ *b* ∗ *c* = *a* ∗ (*b* ∗ *c*)
    by (*cases rule*: *ereal3-cases*[*of a b c*])
      (*simp-all add*: *zero-ereal-def zero-less-mult-iff*)
qed

end

lemma [*simp*]:
  shows *ereal-1-times*: *ereal 1* ∗ *x* = *x*
  and *times-ereal-1*: *x* ∗ *ereal 1* = *x*
by(*simp-all add*: *one-ereal-def*[*symmetric*])

lemma *one-not-le-zero-ereal*[*simp*]: ¬ (*1* ≤ (*0*::*ereal*))
  by (*simp add*: *one-ereal-def zero-ereal-def*)

lemma *real-ereal-1*[*simp*]: *real-of-ereal* (*1*::*ereal*) = *1*
  unfolding *one-ereal-def* **by** *simp*

lemma *real-of-ereal-le-1*:
  fixes *a* :: *ereal*
  shows *a* ≤ *1* ⟹ *real-of-ereal a* ≤ *1*
  by (*cases a*) (*auto simp*: *one-ereal-def*)

lemma *abs-ereal-one*[*simp*]: $|1| = (1::ereal)$
unfolding *one-ereal-def* **by** *simp*

lemma *ereal-mult-zero*[*simp*]:
fixes $a :: ereal$
shows $a * 0 = 0$
by (*cases a*) (*simp-all add*: *zero-ereal-def*)

lemma *ereal-zero-mult*[*simp*]:
fixes $a :: ereal$
shows $0 * a = 0$
by (*cases a*) (*simp-all add*: *zero-ereal-def*)

lemma *ereal-m1-less-0*[*simp*]: $-(1::ereal) < 0$
by (*simp add*: *zero-ereal-def one-ereal-def*)

lemma *ereal-times*[*simp*]:
$1 \neq (\infty::ereal)$ $(\infty::ereal) \neq 1$
$1 \neq -(\infty::ereal)$ $-(\infty::ereal) \neq 1$
by (*auto simp*: *one-ereal-def*)

lemma *ereal-plus-1*[*simp*]:
$1 + ereal\ r = ereal\ (r + 1)$
$ereal\ r + 1 = ereal\ (r + 1)$
$1 + -(\infty::ereal) = -\infty$
$-(\infty::ereal) + 1 = -\infty$
unfolding *one-ereal-def* **by** *auto*

lemma *ereal-zero-times*[*simp*]:
fixes $a\ b :: ereal$
shows $a * b = 0 \longleftrightarrow a = 0 \vee b = 0$
by (*cases rule*: *ereal2-cases*[*of a b*]) *auto*

lemma *ereal-mult-eq-PInfty*[*simp*]:
$a * b = (\infty::ereal) \longleftrightarrow$
$(a = \infty \wedge b > 0) \vee (a > 0 \wedge b = \infty) \vee (a = -\infty \wedge b < 0) \vee (a < 0 \wedge b = -\infty)$
by (*cases rule*: *ereal2-cases*[*of a b*]) *auto*

lemma *ereal-mult-eq-MInfty*[*simp*]:
$a * b = -(\infty::ereal) \longleftrightarrow$
$(a = \infty \wedge b < 0) \vee (a < 0 \wedge b = \infty) \vee (a = -\infty \wedge b > 0) \vee (a > 0 \wedge b = -\infty)$
by (*cases rule*: *ereal2-cases*[*of a b*]) *auto*

lemma *ereal-abs-mult*: $|x * y :: ereal| = |x| * |y|$
by (*cases x y rule*: *ereal2-cases*) (*auto simp*: *abs-mult*)

lemma *ereal-0-less-1*[*simp*]: *0* < (*1*::*ereal*)
  by (*simp-all add*: *zero-ereal-def one-ereal-def*)

lemma *ereal-mult-minus-left*[*simp*]:
  fixes *a b* :: *ereal*
  shows $-a * b = -(a * b)$
  by (*cases rule*: *ereal2-cases*[*of a b*]) *auto*

lemma *ereal-mult-minus-right*[*simp*]:
  fixes *a b* :: *ereal*
  shows $a * -b = -(a * b)$
  by (*cases rule*: *ereal2-cases*[*of a b*]) *auto*

lemma *ereal-mult-infty*[*simp*]:
  a ∗ (∞::*ereal*) = (*if a* = *0 then 0 else if 0* < *a then* ∞ *else* − ∞)
  by (*cases a*) *auto*

lemma *ereal-infty-mult*[*simp*]:
  (∞::*ereal*) ∗ *a* = (*if a* = *0 then 0 else if 0* < *a then* ∞ *else* − ∞)
  by (*cases a*) *auto*

lemma *ereal-mult-strict-right-mono*:
  assumes *a* < *b*
    and *0* < *c*
    and *c* < (∞::*ereal*)
  shows *a* ∗ *c* < *b* ∗ *c*
  using *assms*
  by (*cases rule*: *ereal3-cases*[*of a b c*]) (*auto simp*: *zero-le-mult-iff*)

lemma *ereal-mult-strict-left-mono*:
  a < *b* ⟹ *0* < *c* ⟹ *c* < (∞::*ereal*) ⟹ *c* ∗ *a* < *c* ∗ *b*
  using *ereal-mult-strict-right-mono*
  by (*simp add*: *mult.commute*[*of c*])

lemma *ereal-mult-right-mono*:
  fixes *a b c* :: *ereal*
  shows *a* ≤ *b* ⟹ *0* ≤ *c* ⟹ *a* ∗ *c* ≤ *b* ∗ *c*
  using *assms*
  apply (*cases c* = *0*)
  apply *simp*
  apply (*cases rule*: *ereal3-cases*[*of a b c*])
  apply (*auto simp*: *zero-le-mult-iff*)
  done

lemma *ereal-mult-left-mono*:
  fixes *a b c* :: *ereal*
  shows *a* ≤ *b* ⟹ *0* ≤ *c* ⟹ *c* ∗ *a* ≤ *c* ∗ *b*
  using *ereal-mult-right-mono*
  by (*simp add*: *mult.commute*[*of c*])

lemma *zero-less-one-ereal*[*simp*]: $0 \le (1::ereal)$
by (*simp add*: *one-ereal-def zero-ereal-def*)

lemma *ereal-0-le-mult*[*simp*]: $0 \le a \Longrightarrow 0 \le b \Longrightarrow 0 \le a * (b :: ereal)$
by (*cases rule*: *ereal2-cases*[*of a b*]) *auto*

lemma *ereal-right-distrib*:
fixes *r a b* :: *ereal*
shows $0 \le a \Longrightarrow 0 \le b \Longrightarrow r * (a + b) = r * a + r * b$
by (*cases rule*: *ereal3-cases*[*of r a b*]) (*simp-all add*: *field-simps*)

lemma *ereal-left-distrib*:
fixes *r a b* :: *ereal*
shows $0 \le a \Longrightarrow 0 \le b \Longrightarrow (a + b) * r = a * r + b * r$
by (*cases rule*: *ereal3-cases*[*of r a b*]) (*simp-all add*: *field-simps*)

lemma *ereal-mult-le-0-iff*:
fixes *a b* :: *ereal*
shows $a * b \le 0 \longleftrightarrow (0 \le a \wedge b \le 0) \vee (a \le 0 \wedge 0 \le b)$
by (*cases rule*: *ereal2-cases*[*of a b*]) (*simp-all add*: *mult-le-0-iff*)

lemma *ereal-zero-le-0-iff*:
fixes *a b* :: *ereal*
shows $0 \le a * b \longleftrightarrow (0 \le a \wedge 0 \le b) \vee (a \le 0 \wedge b \le 0)$
by (*cases rule*: *ereal2-cases*[*of a b*]) (*simp-all add*: *zero-le-mult-iff*)

lemma *ereal-mult-less-0-iff*:
fixes *a b* :: *ereal*
shows $a * b < 0 \longleftrightarrow (0 < a \wedge b < 0) \vee (a < 0 \wedge 0 < b)$
by (*cases rule*: *ereal2-cases*[*of a b*]) (*simp-all add*: *mult-less-0-iff*)

lemma *ereal-zero-less-0-iff*:
fixes *a b* :: *ereal*
shows $0 < a * b \longleftrightarrow (0 < a \wedge 0 < b) \vee (a < 0 \wedge b < 0)$
by (*cases rule*: *ereal2-cases*[*of a b*]) (*simp-all add*: *zero-less-mult-iff*)

lemma *ereal-left-mult-cong*:
fixes *a b c* :: *ereal*
shows $c = d \Longrightarrow (d \ne 0 \Longrightarrow a = b) \Longrightarrow a * c = b * d$
by (*cases* $c = 0$) *simp-all*

lemma *ereal-right-mult-cong*:
fixes *a b c* :: *ereal*
shows $c = d \Longrightarrow (d \ne 0 \Longrightarrow a = b) \Longrightarrow c * a = d * b$
by (*cases* $c = 0$) *simp-all*

lemma *ereal-distrib*:
fixes *a b c* :: *ereal*

assumes $a \neq \infty \vee b \neq -\infty$
and $a \neq -\infty \vee b \neq \infty$
and $|c| \neq \infty$
shows $(a + b) * c = a * c + b * c$
using *assms*
by (*cases rule*: *ereal3-cases*[*of a b c*]) (*simp-all add*: *field-simps*)

lemma *numeral-eq-ereal* [*simp*]: *numeral w* = *ereal* (*numeral w*)
apply (*induct w rule*: *num-induct*)
apply (*simp only*: *numeral-One one-ereal-def*)
apply (*simp only*: *numeral-inc ereal-plus-1*)
done

lemma *distrib-left-ereal-nn*:
$c \geq 0 \Longrightarrow (x + y) * ereal\ c = x * ereal\ c + y * ereal\ c$
by(*cases x y rule*: *ereal2-cases*)(*simp-all add*: *ring-distribs*)

lemma *setsum-ereal-right-distrib*:
fixes $f :: 'a \Rightarrow ereal$
shows $(\bigwedge i.\ i \in A \Longrightarrow 0 \leq f\ i) \Longrightarrow r * setsum\ f\ A = (\sum n \in A.\ r * f\ n)$
by (*induct A rule*: *infinite-finite-induct*) (*auto simp*: *ereal-right-distrib setsum-nonneg*)

lemma *setsum-ereal-left-distrib*:
$(\bigwedge i.\ i \in A \Longrightarrow 0 \leq f\ i) \Longrightarrow setsum\ f\ A * r = (\sum n \in A.\ f\ n * r :: ereal)$
using *setsum-ereal-right-distrib*[*of A f r*] **by** (*simp add*: *mult-ac*)

lemma *setsum-left-distrib-ereal*:
$c \geq 0 \Longrightarrow setsum\ f\ A * ereal\ c = (\sum x \in A.\ f\ x * c :: ereal)$
by(*subst setsum-comp-morphism*[**where** $h = \lambda x.\ x * ereal\ c$, *symmetric*])(*simp-all add*: *distrib-left-ereal-nn*)

lemma *ereal-le-epsilon*:
fixes $x\ y :: ereal$
assumes $\forall e.\ 0 < e \longrightarrow x \leq y + e$
shows $x \leq y$
proof −
{
assume *a*: $\exists r.\ y = ereal\ r$
then obtain *r* **where** *r-def*: $y = ereal\ r$
by *auto*
{
assume $x = -\infty$
then have *?thesis* **by** *auto*
}
moreover
{
assume $x \neq -\infty$
then obtain *p* **where** *p-def*: $x = ereal\ p$
using *a assms*[*rule-format*, *of 1*]

```
      by (cases x) auto
     {
      fix e
      have 0 < e ⟶ p ≤ r + e
        using assms[rule-format, of ereal e] p-def r-def by auto
     }
     then have p ≤ r
      apply (subst field-le-epsilon)
      apply auto
      done
     then have ?thesis
      using r-def p-def by auto
    }
    ultimately have ?thesis
     by blast
  }
  moreover
  {
    assume y = −∞ | y = ∞
    then have ?thesis
     using assms[rule-format, of 1] by (cases x) auto
  }
  ultimately show ?thesis
    by (cases y) auto
qed

lemma ereal-le-epsilon2:
  fixes x y :: ereal
  assumes ∀ e. 0 < e ⟶ x ≤ y + ereal e
  shows x ≤ y
proof −
  {
    fix e :: ereal
    assume e > 0
    {
     assume e = ∞
     then have x ≤ y + e
      by auto
    }
    moreover
    {
     assume e ≠ ∞
     then obtain r where e = ereal r
      using ‹e > 0› by (cases e) auto
     then have x ≤ y + e
      using assms[rule-format, of r] ‹e>0› by auto
    }
    ultimately have x ≤ y + e
     by blast
```

```
  }
  then show ?thesis
    using ereal-le-epsilon by auto
qed

lemma ereal-le-real:
  fixes x y :: ereal
  assumes ∀ z. x ≤ ereal z ⟶ y ≤ ereal z
  shows y ≤ x
 by (metis assms ereal-bot ereal-cases ereal-infty-less-eq(2) ereal-less-eq(1) linorder-le-cases)

lemma setprod-ereal-0:
  fixes f :: 'a ⇒ ereal
  shows (∏ i∈A. f i) = 0 ⟷ finite A ∧ (∃ i∈A. f i = 0)
proof (cases finite A)
  case True
  then show ?thesis by (induct A) auto
next
  case False
  then show ?thesis by auto
qed

lemma setprod-ereal-pos:
  fixes f :: 'a ⇒ ereal
  assumes pos: ⋀i. i ∈ I ⟹ 0 ≤ f i
  shows 0 ≤ (∏ i∈I. f i)
proof (cases finite I)
  case True
  from this pos show ?thesis
    by induct auto
next
  case False
  then show ?thesis by simp
qed

lemma setprod-PInf:
  fixes f :: 'a ⇒ ereal
  assumes ⋀i. i ∈ I ⟹ 0 ≤ f i
  shows (∏ i∈I. f i) = ∞ ⟷ finite I ∧ (∃ i∈I. f i = ∞) ∧ (∀ i∈I. f i ≠ 0)
proof (cases finite I)
  case True
  from this assms show ?thesis
  proof (induct I)
    case (insert i I)
    then have pos: 0 ≤ f i 0 ≤ setprod f I
      by (auto intro!: setprod-ereal-pos)
    from insert have (∏ j∈insert i I. f j) = ∞ ⟷ setprod f I ∗ f i = ∞
      by auto
    also have ... ⟷ (setprod f I = ∞ ∨ f i = ∞) ∧ f i ≠ 0 ∧ setprod f I ≠ 0
```

using *setprod-ereal-pos*[*of I f*] *pos*
by (*cases rule*: *ereal2-cases*[*of f i setprod f I*]) *auto*
also have ... $\longleftrightarrow$ *finite* (*insert i I*) $\wedge$ ($\exists j \in insert\ i\ I.\ f\ j = \infty$) $\wedge$ ($\forall j \in insert\ i\ I.\ f\ j \neq 0$)
using *insert* **by** (*auto simp*: *setprod-ereal-0*)
finally show *?case* **.**
qed *simp*
next
case *False*
then show *?thesis* **by** *simp*
qed

lemma *setprod-ereal*: ($\prod i \in A.\ ereal\ (f\ i)$) = *ereal* (*setprod f A*)
proof (*cases finite A*)
case *True*
then show *?thesis*
by *induct* (*auto simp*: *one-ereal-def*)
next
case *False*
then show *?thesis*
by (*simp add*: *one-ereal-def*)
qed

33.1.4 Power

lemma *ereal-power*[*simp*]: (*ereal x*) ^ *n* = *ereal* (*x*^*n*)
by (*induct n*) (*auto simp*: *one-ereal-def*)

lemma *ereal-power-PInf*[*simp*]: (∞::*ereal*) ^ *n* = (*if n* = *0 then 1 else* ∞)
by (*induct n*) (*auto simp*: *one-ereal-def*)

lemma *ereal-power-uminus*[*simp*]:
fixes *x* :: *ereal*
shows (− *x*) ^ *n* = (*if even n then x* ^ *n else* − (*x*^*n*))
by (*induct n*) (*auto simp*: *one-ereal-def*)

lemma *ereal-power-numeral*[*simp*]:
(*numeral num* :: *ereal*) ^ *n* = *ereal* (*numeral num* ^ *n*)
by (*induct n*) (*auto simp*: *one-ereal-def*)

lemma *zero-le-power-ereal*[*simp*]:
fixes *a* :: *ereal*
assumes $0 \leq a$
shows $0 \leq a$ ^ *n*
using *assms* **by** (*induct n*) (*auto simp*: *ereal-zero-le-0-iff*)

33.1.5 Subtraction

lemma *ereal-minus-minus-image*[*simp*]:
fixes *S* :: *ereal set*

shows *uminus* ‘ *uminus* ‘ $S = S$
by (*auto simp*: *image-iff*)

lemma *ereal-uminus-lessThan*[*simp*]:
fixes a :: *ereal*
shows *uminus* ‘ $\{..<a\} = \{-a<..\}$
proof −
{
fix x
assume $-a < x$
then have $- x < - (- a)$
by (*simp del*: *ereal-uminus-uminus*)
then have $- x < a$
by *simp*
}
then show *?thesis*
by *force*
qed

lemma *ereal-uminus-greaterThan*[*simp*]: *uminus* ‘ $\{(a::ereal)<..\} = \{..<-a\}$
by (*metis ereal-uminus-lessThan ereal-uminus-uminus ereal-minus-minus-image*)

instantiation *ereal* :: *minus*
begin

definition $x - y = x + -(y::ereal)$
instance ..

end

lemma *ereal-minus*[*simp*]:
ereal r − *ereal* p = *ereal* $(r - p)$
$-\infty$ − *ereal* $r = -\infty$
ereal $r - \infty = -\infty$
$(\infty::ereal) - x = \infty$
$-(\infty::ereal) - \infty = -\infty$
$x - -y = x + y$
$x - 0 = x$
$0 - x = -x$
by (*simp-all add*: *minus-ereal-def*)

lemma *ereal-x-minus-x*[*simp*]: $x - x = ($*if* $|x| = \infty$ *then* ∞ *else* $0::ereal)$
by (*cases* x) *simp-all*

lemma *ereal-eq-minus-iff*:
fixes $x\ y\ z$ :: *ereal*
shows $x = z - y \longleftrightarrow$
$(|y| \neq \infty \longrightarrow x + y = z) \wedge$
$(y = -\infty \longrightarrow x = \infty) \wedge$

$(y = \infty \longrightarrow z = \infty \longrightarrow x = \infty) \wedge$
$(y = \infty \longrightarrow z \neq \infty \longrightarrow x = -\infty)$
by (*cases rule*: *ereal3-cases*[*of x y z*]) *auto*

lemma *ereal-eq-minus*:
fixes $x\ y\ z$ :: *ereal*
shows $|y| \neq \infty \Longrightarrow x = z - y \longleftrightarrow x + y = z$
by (*auto simp*: *ereal-eq-minus-iff*)

lemma *ereal-less-minus-iff*:
fixes $x\ y\ z$ :: *ereal*
shows $x < z - y \longleftrightarrow$
$(y = \infty \longrightarrow z = \infty \wedge x \neq \infty) \wedge$
$(y = -\infty \longrightarrow x \neq \infty) \wedge$
$(|y| \neq \infty \longrightarrow x + y < z)$
by (*cases rule*: *ereal3-cases*[*of x y z*]) *auto*

lemma *ereal-less-minus*:
fixes $x\ y\ z$ :: *ereal*
shows $|y| \neq \infty \Longrightarrow x < z - y \longleftrightarrow x + y < z$
by (*auto simp*: *ereal-less-minus-iff*)

lemma *ereal-le-minus-iff*:
fixes $x\ y\ z$ :: *ereal*
shows $x \leq z - y \longleftrightarrow (y = \infty \longrightarrow z \neq \infty \longrightarrow x = -\infty) \wedge (|y| \neq \infty \longrightarrow x + y \leq z)$
by (*cases rule*: *ereal3-cases*[*of x y z*]) *auto*

lemma *ereal-le-minus*:
fixes $x\ y\ z$ :: *ereal*
shows $|y| \neq \infty \Longrightarrow x \leq z - y \longleftrightarrow x + y \leq z$
by (*auto simp*: *ereal-le-minus-iff*)

lemma *ereal-minus-less-iff*:
fixes $x\ y\ z$ :: *ereal*
shows $x - y < z \longleftrightarrow y \neq -\infty \wedge (y = \infty \longrightarrow x \neq \infty \wedge z \neq -\infty) \wedge (y \neq \infty \longrightarrow x < z + y)$
by (*cases rule*: *ereal3-cases*[*of x y z*]) *auto*

lemma *ereal-minus-less*:
fixes $x\ y\ z$ :: *ereal*
shows $|y| \neq \infty \Longrightarrow x - y < z \longleftrightarrow x < z + y$
by (*auto simp*: *ereal-minus-less-iff*)

lemma *ereal-minus-le-iff*:
fixes $x\ y\ z$ :: *ereal*
shows $x - y \leq z \longleftrightarrow$
$(y = -\infty \longrightarrow z = \infty) \wedge$
$(y = \infty \longrightarrow x = \infty \longrightarrow z = \infty) \wedge$

$(|y| \neq \infty \longrightarrow x \leq z + y)$
by (*cases rule*: *ereal3-cases*[*of x y z*]) *auto*

lemma *ereal-minus-le*:
fixes $x\ y\ z$ :: *ereal*
shows $|y| \neq \infty \Longrightarrow x - y \leq z \longleftrightarrow x \leq z + y$
by (*auto simp*: *ereal-minus-le-iff*)

lemma *ereal-minus-eq-minus-iff*:
fixes $a\ b\ c$ :: *ereal*
shows $a - b = a - c \longleftrightarrow$
$b = c \lor a = \infty \lor (a = -\infty \land b \neq -\infty \land c \neq -\infty)$
by (*cases rule*: *ereal3-cases*[*of a b c*]) *auto*

lemma *ereal-add-le-add-iff*:
fixes $a\ b\ c$ :: *ereal*
shows $c + a \leq c + b \longleftrightarrow$
$a \leq b \lor c = \infty \lor (c = -\infty \land a \neq \infty \land b \neq \infty)$
by (*cases rule*: *ereal3-cases*[*of a b c*]) (*simp-all add*: *field-simps*)

lemma *ereal-add-le-add-iff2*:
fixes $a\ b\ c$ :: *ereal*
shows $a + c \leq b + c \longleftrightarrow a \leq b \lor c = \infty \lor (c = -\infty \land a \neq \infty \land b \neq \infty)$
by(*cases rule*: *ereal3-cases*[*of a b c*])(*simp-all add*: *field-simps*)

lemma *ereal-mult-le-mult-iff*:
fixes $a\ b\ c$ :: *ereal*
shows $|c| \neq \infty \Longrightarrow c * a \leq c * b \longleftrightarrow (0 < c \longrightarrow a \leq b) \land (c < 0 \longrightarrow b \leq a)$
by (*cases rule*: *ereal3-cases*[*of a b c*]) (*simp-all add*: *mult-le-cancel-left*)

lemma *ereal-minus-mono*:
fixes $A\ B\ C\ D$ :: *ereal* **assumes** $A \leq B\ D \leq C$
shows $A - C \leq B - D$
using *assms*
by (*cases rule*: *ereal3-cases*[*case-product ereal-cases*, *of A B C D*]) *simp-all*

lemma *ereal-mono-minus-cancel*:
fixes $a\ b\ c$ :: *ereal*
shows $c - a \leq c - b \Longrightarrow 0 \leq c \Longrightarrow c < \infty \Longrightarrow b \leq a$
by (*cases a b c rule*: *ereal3-cases*) *auto*

lemma *real-of-ereal-minus*:
fixes $a\ b$ :: *ereal*
shows *real-of-ereal* $(a - b) =$ (*if* $|a| = \infty \lor |b| = \infty$ *then 0 else real-of-ereal a* $-$ *real-of-ereal b*)
by (*cases rule*: *ereal2-cases*[*of a b*]) *auto*

lemma *real-of-ereal-minus'*: $|x| = \infty \longleftrightarrow |y| = \infty \Longrightarrow$ *real-of-ereal x* $-$ *real-of-ereal y* $=$ *real-of-ereal* $(x - y$ :: *ereal*$)$

by(*subst real-of-ereal-minus*) *auto*

lemma *ereal-diff-positive*:
fixes $a\ b$ :: *ereal* **shows** $a \leq b \Longrightarrow 0 \leq b - a$
by (*cases rule*: *ereal2-cases*[*of a b*]) *auto*

lemma *ereal-between*:
fixes $x\ e$ :: *ereal*
assumes $|x| \neq \infty$
and $0 < e$
shows $x - e < x$
and $x < x + e$
using *assms*
apply (*cases x*, *cases e*)
apply *auto*
using *assms*
apply (*cases x*, *cases e*)
apply *auto*
done

lemma *ereal-minus-eq-PInfty-iff*:
fixes $x\ y$ :: *ereal*
shows $x - y = \infty \longleftrightarrow y = -\infty \vee x = \infty$
by (*cases x y rule*: *ereal2-cases*) *simp-all*

lemma *ereal-diff-add-eq-diff-diff-swap*:
fixes $x\ y\ z$ :: *ereal*
shows $|y| \neq \infty \Longrightarrow x - (y + z) = x - y - z$
by(*cases x y z rule*: *ereal3-cases*) *simp-all*

lemma *ereal-diff-add-assoc2*:
fixes $x\ y\ z$ :: *ereal*
shows $x + y - z = x - z + y$
by(*cases x y z rule*: *ereal3-cases*) *simp-all*

lemma *ereal-add-uminus-conv-diff*: **fixes** $x\ y\ z$ :: *ereal* **shows** $- x + y = y - x$
by(*cases x y rule*: *ereal2-cases*) *simp-all*

lemma *ereal-minus-diff-eq*:
fixes $x\ y$ :: *ereal*
shows $[\![x = \infty \longrightarrow y \neq \infty; x = -\infty \longrightarrow y \neq - \infty]\!] \Longrightarrow - (x - y) = y - x$
by(*cases x y rule*: *ereal2-cases*) *simp-all*

lemma *ediff-le-self* [*simp*]: $x - y \leq (x :: enat)$
by(*cases x y rule*: *enat.exhaust*[*case-product enat.exhaust*]) *simp-all*

33.1.6 Division

instantiation *ereal* :: *inverse*

begin

function *inverse-ereal* **where**
inverse (*ereal r*) = (*if r* = *0 then* ∞ *else ereal* (*inverse r*))
| *inverse* (∞::*ereal*) = *0*
| *inverse* (−∞::*ereal*) = *0*
by (*auto intro*: *ereal-cases*)
termination by (*relation* {}) *simp*

definition *x div y* = *x* ∗ *inverse* (*y* :: *ereal*)

instance ..

end

lemma *real-of-ereal-inverse*[*simp*]:
fixes *a* :: *ereal*
shows *real-of-ereal* (*inverse a*) = *1* / *real-of-ereal a*
by (*cases a*) (*auto simp*: *inverse-eq-divide*)

lemma *ereal-inverse*[*simp*]:
inverse (*0*::*ereal*) = ∞
inverse (*1*::*ereal*) = *1*
by (*simp-all add*: *one-ereal-def zero-ereal-def*)

lemma *ereal-divide*[*simp*]:
ereal r / *ereal p* = (*if p* = *0 then ereal r* ∗ ∞ *else ereal* (*r* / *p*))
unfolding *divide-ereal-def* **by** (*auto simp*: *divide-real-def*)

lemma *ereal-divide-same*[*simp*]:
fixes *x* :: *ereal*
shows *x* / *x* = (*if* |*x*| = ∞ ∨ *x* = *0 then 0 else 1*)
by (*cases x*) (*simp-all add*: *divide-real-def divide-ereal-def one-ereal-def*)

lemma *ereal-inv-inv*[*simp*]:
fixes *x* :: *ereal*
shows *inverse* (*inverse x*) = (*if x* ≠ −∞ *then x else* ∞)
by (*cases x*) *auto*

lemma *ereal-inverse-minus*[*simp*]:
fixes *x* :: *ereal*
shows *inverse* (− *x*) = (*if x* = *0 then* ∞ *else* −*inverse x*)
by (*cases x*) *simp-all*

lemma *ereal-uminus-divide*[*simp*]:
fixes *x y* :: *ereal*
shows − *x* / *y* = − (*x* / *y*)
unfolding *divide-ereal-def* **by** *simp*

lemma *ereal-divide-Infty*[*simp*]:
fixes *x* :: *ereal*
shows $x\ /\ \infty = 0$ $x\ /\ -\infty = 0$
unfolding *divide-ereal-def* **by** *simp-all*

lemma *ereal-divide-one*[*simp*]: $x\ /\ 1 = (x{::}ereal)$
unfolding *divide-ereal-def* **by** *simp*

lemma *ereal-divide-ereal*[*simp*]: $\infty\ /\ ereal\ r = (if\ 0 \le r\ then\ \infty\ else\ -\infty)$
unfolding *divide-ereal-def* **by** *simp*

lemma *ereal-inverse-nonneg-iff*: $0 \le inverse\ (x :: ereal) \longleftrightarrow 0 \le x \lor x = -\infty$
by (*cases x*) *auto*

lemma *inverse-ereal-ge0I*: $0 \le (x :: ereal) \Longrightarrow 0 \le inverse\ x$
by(*cases x*) *simp-all*

lemma *zero-le-divide-ereal*[*simp*]:
fixes *a* :: *ereal*
assumes $0 \le a$
and $0 \le b$
shows $0 \le a\ /\ b$
using *assms* **by** (*cases rule*: *ereal2-cases*[*of a b*]) (*auto simp*: *zero-le-divide-iff*)

lemma *ereal-le-divide-pos*:
fixes *x y z* :: *ereal*
shows $x > 0 \Longrightarrow x \ne \infty \Longrightarrow y \le z\ /\ x \longleftrightarrow x * y \le z$
by (*cases rule*: *ereal3-cases*[*of x y z*]) (*auto simp*: *field-simps*)

lemma *ereal-divide-le-pos*:
fixes *x y z* :: *ereal*
shows $x > 0 \Longrightarrow x \ne \infty \Longrightarrow z\ /\ x \le y \longleftrightarrow z \le x * y$
by (*cases rule*: *ereal3-cases*[*of x y z*]) (*auto simp*: *field-simps*)

lemma *ereal-le-divide-neg*:
fixes *x y z* :: *ereal*
shows $x < 0 \Longrightarrow x \ne -\infty \Longrightarrow y \le z\ /\ x \longleftrightarrow z \le x * y$
by (*cases rule*: *ereal3-cases*[*of x y z*]) (*auto simp*: *field-simps*)

lemma *ereal-divide-le-neg*:
fixes *x y z* :: *ereal*
shows $x < 0 \Longrightarrow x \ne -\infty \Longrightarrow z\ /\ x \le y \longleftrightarrow x * y \le z$
by (*cases rule*: *ereal3-cases*[*of x y z*]) (*auto simp*: *field-simps*)

lemma *ereal-inverse-antimono-strict*:
fixes *x y* :: *ereal*
shows $0 \le x \Longrightarrow x < y \Longrightarrow inverse\ y < inverse\ x$
by (*cases rule*: *ereal2-cases*[*of x y*]) *auto*

lemma *ereal-inverse-antimono*:
fixes $x\ y$ :: *ereal*
shows $0 \leq x \Longrightarrow x \leq y \Longrightarrow$ *inverse* $y \leq$ *inverse* x
by (*cases rule*: *ereal2-cases*[*of x y*]) *auto*

lemma *inverse-inverse-Pinfty-iff* [*simp*]:
fixes x :: *ereal*
shows *inverse* $x = \infty \longleftrightarrow x = 0$
by (*cases* x) *auto*

lemma *ereal-inverse-eq-0*:
fixes x :: *ereal*
shows *inverse* $x = 0 \longleftrightarrow x = \infty \lor x = -\infty$
by (*cases* x) *auto*

lemma *ereal-0-gt-inverse*:
fixes x :: *ereal*
shows $0 <$ *inverse* $x \longleftrightarrow x \neq \infty \land 0 \leq x$
by (*cases* x) *auto*

lemma *ereal-inverse-le-0-iff*:
fixes x :: *ereal*
shows *inverse* $x \leq 0 \longleftrightarrow x < 0 \lor x = \infty$
by(*cases* x) *auto*

lemma *ereal-divide-eq-0-iff*: $x\ /\ y = 0 \longleftrightarrow x = 0 \lor |y$ :: *ereal*$| = \infty$
by(*cases x y rule*: *ereal2-cases*) *simp-all*

lemma *ereal-mult-less-right*:
fixes $a\ b\ c$ :: *ereal*
assumes $b * a < c * a$
and $0 < a$
and $a < \infty$
shows $b < c$
using *assms*
by (*cases rule*: *ereal3-cases*[*of a b c*])
(*auto split*: *if-split-asm simp*: *zero-less-mult-iff zero-le-mult-iff*)

lemma *ereal-mult-divide*: **fixes** $a\ b$ :: *ereal* **shows** $0 < b \Longrightarrow b < \infty \Longrightarrow b * (a\ /\ b) = a$
by (*cases a b rule*: *ereal2-cases*) *auto*

lemma *ereal-power-divide*:
fixes $x\ y$ :: *ereal*
shows $y \neq 0 \Longrightarrow (x\ /\ y)\ \hat{}\ n = x\hat{}n\ /\ y\hat{}n$
by (*cases rule*: *ereal2-cases* [*of x y*])
(*auto simp*: *one-ereal-def zero-ereal-def power-divide zero-le-power-eq*)

lemma *ereal-le-mult-one-interval*:

fixes $x\ y :: ereal$
assumes y: $y \neq -\infty$
assumes z: $\bigwedge z.\ 0 < z \Longrightarrow z < 1 \Longrightarrow z * x \leq y$
shows $x \leq y$
proof (*cases x*)
case *PInf*
with z[*of* $1\ /\ 2$] **show** $x \leq y$
by (*simp add*: *one-ereal-def*)
next
case (*real r*)
note $r = this$
show $x \leq y$
proof (*cases y*)
case (*real p*)
note $p = this$
have $r \leq p$
proof (*rule field-le-mult-one-interval*)
fix $z :: real$
assume $0 < z$ **and** $z < 1$
with z[*of ereal z*] **show** $z * r \leq p$
using *p r* **by** (*auto simp*: *zero-le-mult-iff one-ereal-def*)
qed
then show $x \leq y$
using *p r* **by** *simp*
qed (*insert y*, *simp-all*)
qed *simp*

lemma *ereal-divide-right-mono*[*simp*]:
fixes $x\ y\ z :: ereal$
assumes $x \leq y$
and $0 < z$
shows $x\ /\ z \leq y\ /\ z$
using *assms* **by** (*cases x y z rule*: *ereal3-cases*) (*auto intro*: *divide-right-mono*)

lemma *ereal-divide-left-mono*[*simp*]:
fixes $x\ y\ z :: ereal$
assumes $y \leq x$
and $0 < z$
and $0 < x * y$
shows $z\ /\ x \leq z\ /\ y$
using *assms*
by (*cases x y z rule*: *ereal3-cases*)
(*auto intro*: *divide-left-mono simp*: *field-simps zero-less-mult-iff mult-less-0-iff split*: *if-split-asm*)

lemma *ereal-divide-zero-left*[*simp*]:
fixes $a :: ereal$
shows $0\ /\ a = 0$
by (*cases a*) (*auto simp*: *zero-ereal-def*)

lemma *ereal-times-divide-eq-left*[*simp*]:
fixes *a b c* :: *ereal*
shows *b* / *c* ∗ *a* = *b* ∗ *a* / *c*
by (*cases a b c rule*: *ereal3-cases*) (*auto simp*: *field-simps zero-less-mult-iff mult-less-0-iff*)

lemma *ereal-times-divide-eq*: *a* ∗ (*b* / *c* :: *ereal*) = *a* ∗ *b* / *c*
by (*cases a b c rule*: *ereal3-cases*)
(*auto simp*: *field-simps zero-less-mult-iff*)

lemma *ereal-inverse-real*: $|z| \neq \infty \Longrightarrow z \neq 0 \Longrightarrow$ *ereal* (*inverse* (*real-of-ereal z*)) = *inverse z*
by (*cases z*) *simp-all*

lemma *ereal-inverse-mult*:
$a \neq 0 \Longrightarrow b \neq 0 \Longrightarrow$ *inverse* (*a* ∗ (*b*::*ereal*)) = *inverse a* ∗ *inverse b*
by (*cases a*; *cases b*) *auto*

33.2 Complete lattice

instantiation *ereal* :: *lattice*
begin

definition [*simp*]: *sup x y* = (*max x y* :: *ereal*)
definition [*simp*]: *inf x y* = (*min x y* :: *ereal*)
instance by *standard simp-all*

end

instantiation *ereal* :: *complete-lattice*
begin

definition *bot* = (−∞::*ereal*)
definition *top* = (∞::*ereal*)

definition *Sup S* = (*SOME x* :: *ereal*. (∀ *y*∈*S*. *y* ≤ *x*) ∧ (∀ *z*. (∀ *y*∈*S*. *y* ≤ *z*) ⟶ *x* ≤ *z*))
definition *Inf S* = (*SOME x* :: *ereal*. (∀ *y*∈*S*. *x* ≤ *y*) ∧ (∀ *z*. (∀ *y*∈*S*. *z* ≤ *y*) ⟶ *z* ≤ *x*))

lemma *ereal-complete-Sup*:
fixes *S* :: *ereal set*
shows ∃ *x*. (∀ *y*∈*S*. *y* ≤ *x*) ∧ (∀ *z*. (∀ *y*∈*S*. *y* ≤ *z*) ⟶ *x* ≤ *z*)
proof (*cases* ∃ *x*. ∀ *a*∈*S*. *a* ≤ *ereal x*)
case *True*
then obtain *y* **where** *y*: ⋀*a*. *a*∈*S* ⟹ *a* ≤ *ereal y*
by *auto*
then have ∞ ∉ *S*
by *force*

```
  show ?thesis
  proof (cases S ≠ {−∞} ∧ S ≠ {})
    case True
    with ‹∞ ∉ S› obtain x where x: x ∈ S |x| ≠ ∞
      by auto
    obtain s where s: ∀x∈ereal −‘ S. x ≤ s ⋀z. (∀x∈ereal −‘ S. x ≤ z) ⟹ s
≤ z
    proof (atomize-elim, rule complete-real)
      show ∃x. x ∈ ereal −‘ S
        using x by auto
      show ∃z. ∀x∈ereal −‘ S. x ≤ z
        by (auto dest: y intro!: exI[of - y])
    qed
    show ?thesis
    proof (safe intro!: exI[of - ereal s])
      fix y
      assume y ∈ S
      with s ‹∞ ∉ S› show y ≤ ereal s
        by (cases y) auto
    next
      fix z
      assume ∀y∈S. y ≤ z
      with ‹S ≠ {−∞} ∧ S ≠ {}› show ereal s ≤ z
        by (cases z) (auto intro!: s)
    qed
  next
    case False
    then show ?thesis
      by (auto intro!: exI[of - −∞])
  qed
next
  case False
  then show ?thesis
     by (fastforce intro!: exI[of - ∞] ereal-top intro: order-trans dest: less-imp-le
simp: not-le)
qed

lemma ereal-complete-uminus-eq:
  fixes S :: ereal set
  shows (∀y∈uminus‘S. y ≤ x) ∧ (∀z. (∀y∈uminus‘S. y ≤ z) ⟶ x ≤ z)
    ⟷ (∀y∈S. −x ≤ y) ∧ (∀z. (∀y∈S. z ≤ y) ⟶ z ≤ −x)
  by simp (metis ereal-minus-le-minus ereal-uminus-uminus)

lemma ereal-complete-Inf:
  ∃x. (∀y∈S::ereal set. x ≤ y) ∧ (∀z. (∀y∈S. z ≤ y) ⟶ z ≤ x)
  using ereal-complete-Sup[of uminus ‘ S]
  unfolding ereal-complete-uminus-eq
  by auto
```

instance
proof
  show *Sup* {} = (*bot*::*ereal*)
    apply (*auto simp*: *bot-ereal-def Sup-ereal-def*)
    apply (*rule some1-equality*)
    apply (*metis ereal-bot ereal-less-eq*(*2*))
    apply (*metis ereal-less-eq*(*2*))
    done
  show *Inf* {} = (*top*::*ereal*)
    apply (*auto simp*: *top-ereal-def Inf-ereal-def*)
    apply (*rule some1-equality*)
    apply (*metis ereal-top ereal-less-eq*(*1*))
    apply (*metis ereal-less-eq*(*1*))
    done
qed (*auto intro*: *someI2-ex ereal-complete-Sup ereal-complete-Inf*
  simp: *Sup-ereal-def Inf-ereal-def bot-ereal-def top-ereal-def*)

end

instance *ereal* :: *complete-linorder* **..**

instance *ereal* :: *linear-continuum*
proof
  show ∃ *a b*::*ereal*. *a* ≠ *b*
    using *zero-neq-one* **by** *blast*
qed

33.2.1 Topological space

instantiation *ereal* :: *linear-continuum-topology*
begin

definition *open-ereal* :: *ereal set* ⇒ *bool* **where**
  open-ereal-generated: *open-ereal* = *generate-topology* (*range lessThan* ∪ *range greaterThan*)

instance
  by *standard* (*simp add*: *open-ereal-generated*)

end

lemma *continuous-on-ereal*[*continuous-intros*]:
  assumes *f*: *continuous-on s f* **shows** *continuous-on s* (λ*x*. *ereal* (*f x*))
  by (*rule continuous-on-compose2* [*OF continuous-onI-mono*[*of ereal UNIV*] *f*]) *auto*

lemma *tendsto-ereal*[*tendsto-intros*, *simp*, *intro*]: (*f* ⟶ *x*) *F* ⟹ ((λ*x*. *ereal* (*f x*)) ⟶ *ereal x*) *F*
  using *isCont-tendsto-compose*[*of x ereal f F*] *continuous-on-ereal*[*of UNIV* λ*x*.

x]
  by (*simp add*: *continuous-on-eq-continuous-at*)

lemma *tendsto-uminus-ereal*[*tendsto-intros*, *simp*, *intro*]: (*f* ⟶ *x*) *F* ⟹ ((λ*x*. − *f x*::*ereal*) ⟶ − *x*) *F*
  apply (*rule tendsto-compose*[**where** *g*=*uminus*])
  apply (*auto intro*!: *order-tendstoI simp*: *eventually-at-topological*)
  apply (*rule-tac x*={..< −*a*} **in** *exI*)
  apply (*auto split*: *ereal.split simp*: *ereal-less-uminus-reorder*) []
  apply (*rule-tac x*={− *a* <..} **in** *exI*)
  apply (*auto split*: *ereal.split simp*: *ereal-uminus-reorder*) []
  done

lemma *at-infty-ereal-eq-at-top*: *at* ∞ = *filtermap ereal at-top*
  unfolding *filter-eq-iff eventually-at-filter eventually-at-top-linorder eventually-filtermap top-ereal-def*[*symmetric*]
  apply (*subst eventually-nhds-top*[*of 0*])
  apply (*auto simp*: *top-ereal-def less-le ereal-all-split ereal-ex-split*)
  apply (*metis PInfty-neq-ereal*(*2*) *ereal-less-eq*(*3*) *ereal-top le-cases order-trans*)
  done

lemma *ereal-Lim-uminus*: (*f* ⟶ *f0*) *net* ⟷ ((λ*x*. − *f x*::*ereal*) ⟶ − *f0*) *net*
  using *tendsto-uminus-ereal*[*of f f0 net*] *tendsto-uminus-ereal*[*of* λ*x*. − *f x* − *f0 net*]
  by *auto*

lemma *ereal-divide-less-iff*: *0* < (*c*::*ereal*) ⟹ *c* < ∞ ⟹ *a* / *c* < *b* ⟷ *a* < *b* ∗ *c*
  by (*cases a b c rule*: *ereal3-cases*) (*auto simp*: *field-simps*)

lemma *ereal-less-divide-iff*: *0* < (*c*::*ereal*) ⟹ *c* < ∞ ⟹ *a* < *b* / *c* ⟷ *a* ∗ *c* < *b*
  by (*cases a b c rule*: *ereal3-cases*) (*auto simp*: *field-simps*)

lemma *tendsto-cmult-ereal*[*tendsto-intros*, *simp*, *intro*]:
  assumes *c*: |*c*| ≠ ∞ **and** *f*: (*f* ⟶ *x*) *F* **shows** ((λ*x*. *c* ∗ *f x*::*ereal*) ⟶ *c* ∗ *x*) *F*
proof −
  { fix *c* :: *ereal* **assume** *0* < *c c* < ∞
    then have ((λ*x*. *c* ∗ *f x*::*ereal*) ⟶ *c* ∗ *x*) *F*
      apply (*intro tendsto-compose*[*OF - f*])
      apply (*auto intro*!: *order-tendstoI simp*: *eventually-at-topological*)
      apply (*rule-tac x*={*a*/*c* <..} **in** *exI*)
      apply (*auto split*: *ereal.split simp*: *ereal-divide-less-iff mult.commute*) []
      apply (*rule-tac x*={..< *a*/*c*} **in** *exI*)
      apply (*auto split*: *ereal.split simp*: *ereal-less-divide-iff mult.commute*) []
      done }
  note ∗ = *this*

have (($0 < c \wedge c < \infty$) ∨ ($-\infty < c \wedge c < 0$) ∨ $c = 0$)
using *c* **by** (*cases c*) *auto*
then show *?thesis*
proof (*elim disjE conjE*)
assume $-\infty < c$ $c < 0$
then have $0 < -c$ $-c < \infty$
by (*auto simp*: *ereal-uminus-reorder ereal-less-uminus-reorder*[*of 0*])
then have (($\lambda x.$ $(-c) * f\, x$) $\longrightarrow$ $(-c) * x$) *F*
by (*rule* ∗)
from *tendsto-uminus-ereal*[*OF this*] **show** *?thesis*
by *simp*
qed (*auto intro*!: ∗)
qed

lemma *tendsto-cmult-ereal-not-0*[*tendsto-intros*, *simp*, *intro*]:
assumes $x \neq 0$ **and** *f*: ($f \longrightarrow x$) *F* **shows** (($\lambda x.$ $c * f\, x$::*ereal*) $\longrightarrow$ $c * x$) *F*
proof *cases*
assume $|c| = \infty$
show *?thesis*
proof (*rule filterlim-cong*[*THEN iffD1*, *OF refl refl - tendsto-const*])
have $0 < x \vee x < 0$
using ‹$x \neq 0$› **by** (*auto simp add*: *neq-iff*)
then show *eventually* ($\lambda x'.$ $c * x = c * f\, x'$) *F*
proof
assume $0 < x$ **from** *order-tendstoD*(*1*)[*OF f this*] **show** *?thesis*
by *eventually-elim* (*insert* ‹$0<x$› ‹$|c| = \infty$›, *auto*)
next
assume $x < 0$ **from** *order-tendstoD*(*2*)[*OF f this*] **show** *?thesis*
by *eventually-elim* (*insert* ‹$x<0$› ‹$|c| = \infty$›, *auto*)
qed
qed
qed (*rule tendsto-cmult-ereal*[*OF - f*])

lemma *tendsto-cadd-ereal*[*tendsto-intros*, *simp*, *intro*]:
assumes *c*: $y \neq -\infty$ $x \neq -\infty$ **and** *f*: ($f \longrightarrow x$) *F* **shows** (($\lambda x.$ $f\, x + y$::*ereal*) $\longrightarrow$ $x + y$) *F*
apply (*intro tendsto-compose*[*OF - f*])
apply (*auto intro*!: *order-tendstoI simp*: *eventually-at-topological*)
apply (*rule-tac x*={$a - y$ <..} **in** *exI*)
apply (*auto split*: *ereal.split simp*: *ereal-minus-less-iff c*) []
apply (*rule-tac x*={..< $a - y$} **in** *exI*)
apply (*auto split*: *ereal.split simp*: *ereal-less-minus-iff c*) []
done

lemma *tendsto-add-left-ereal*[*tendsto-intros*, *simp*, *intro*]:
assumes *c*: $|y| \neq \infty$ **and** *f*: ($f \longrightarrow x$) *F* **shows** (($\lambda x.$ $f\, x + y$::*ereal*) $\longrightarrow$ $x + y$) *F*

apply (*intro tendsto-compose*[*OF* - *f*])
apply (*auto intro*!: *order-tendstoI simp*: *eventually-at-topological*)
apply (*rule-tac* x={$a - y$ <..} **in** *exI*)
apply (*insert c*, *auto split*: *ereal.split simp*: *ereal-minus-less-iff*) []
apply (*rule-tac* x={..< $a - y$} **in** *exI*)
apply (*auto split*: *ereal.split simp*: *ereal-less-minus-iff c*) []
done

lemma *continuous-at-ereal*[*continuous-intros*]: *continuous F f* $\Longrightarrow$ *continuous F* (λx. *ereal* (*f x*))
unfolding *continuous-def* **by** *auto*

lemma *ereal-Sup*:
assumes $*$: |*SUP a*:*A*. *ereal a*| $\neq \infty$
shows *ereal* (*Sup A*) = (*SUP a*:*A*. *ereal a*)
proof (*rule continuous-at-Sup-mono*)
obtain *r* **where** *r*: *ereal r* = (*SUP a*:*A*. *ereal a*) *A* $\neq$ {}
using $*$ **by** (*force simp*: *bot-ereal-def*)
then show *bdd-above A A* $\neq$ {}
by (*auto intro*!: *SUP-upper bdd-aboveI*[*of* - *r*] *simp add*: *ereal-less-eq*(*3*)[*symmetric*] *simp del*: *ereal-less-eq*)
qed (*auto simp*: *mono-def continuous-at-imp-continuous-at-within continuous-at-ereal*)

lemma *ereal-SUP*: |*SUP a*:*A*. *ereal* (*f a*)| $\neq \infty \Longrightarrow$ *ereal* (*SUP a*:*A*. *f a*) = (*SUP a*:*A*. *ereal* (*f a*))
using *ereal-Sup*[*of f'A*] **by** *auto*

lemma *ereal-Inf*:
assumes $*$: |*INF a*:*A*. *ereal a*| $\neq \infty$
shows *ereal* (*Inf A*) = (*INF a*:*A*. *ereal a*)
proof (*rule continuous-at-Inf-mono*)
obtain *r* **where** *r*: *ereal r* = (*INF a*:*A*. *ereal a*) *A* $\neq$ {}
using $*$ **by** (*force simp*: *top-ereal-def*)
then show *bdd-below A A* $\neq$ {}
by (*auto intro*!: *INF-lower bdd-belowI*[*of* - *r*] *simp add*: *ereal-less-eq*(*3*)[*symmetric*] *simp del*: *ereal-less-eq*)
qed (*auto simp*: *mono-def continuous-at-imp-continuous-at-within continuous-at-ereal*)

lemma *ereal-Inf'*:
assumes $*$: *bdd-below A A* $\neq$ {}
shows *ereal* (*Inf A*) = (*INF a*:*A*. *ereal a*)
proof (*rule ereal-Inf*)
from $*$ **obtain** *l u* **where** $\bigwedge x.\ x \in A \Longrightarrow l \leq x$ $u \in A$
by (*auto simp*: *bdd-below-def*)
then have $l \leq$ (*INF x*:*A*. *ereal x*) (*INF x*:*A*. *ereal x*) $\leq u$
by (*auto intro*!: *INF-greatest INF-lower*)
then show |*INF a*:*A*. *ereal a*| $\neq \infty$
by *auto*
qed

lemma *ereal-INF*: $|INF\ a{:}A.\ ereal\ (f\ a)| \neq \infty \Longrightarrow ereal\ (INF\ a{:}A.\ f\ a) = (INF\ a{:}A.\ ereal\ (f\ a))$
  using *ereal-Inf*[*of f'A*] **by** *auto*

lemma *ereal-Sup-uminus-image-eq*: *Sup* (*uminus* ' *S*::*ereal set*) = − *Inf S*
  by (*auto intro*!: *SUP-eqI*
        simp: *Ball-def*[*symmetric*] *ereal-uminus-le-reorder le-Inf-iff*
        intro!: *complete-lattice-class.Inf-lower2*)

lemma *ereal-SUP-uminus-eq*:
  fixes *f* :: *'a* ⇒ *ereal*
  shows (*SUP x*:*S*. *uminus* (*f x*)) = − (*INF x*:*S*. *f x*)
  using *ereal-Sup-uminus-image-eq* [*of f* ' *S*] **by** (*simp add*: *comp-def*)

lemma *ereal-inj-on-uminus*[*intro*, *simp*]: *inj-on uminus* (*A* :: *ereal set*)
  by (*auto intro*!: *inj-onI*)

lemma *ereal-Inf-uminus-image-eq*: *Inf* (*uminus* ' *S*::*ereal set*) = − *Sup S*
  using *ereal-Sup-uminus-image-eq*[*of uminus* ' *S*] **by** *simp*

lemma *ereal-INF-uminus-eq*:
  fixes *f* :: *'a* ⇒ *ereal*
  shows (*INF x*:*S*. − *f x*) = − (*SUP x*:*S*. *f x*)
  using *ereal-Inf-uminus-image-eq* [*of f* ' *S*] **by** (*simp add*: *comp-def*)

lemma *ereal-SUP-uminus*:
  fixes *f* :: *'a* ⇒ *ereal*
  shows (*SUP i* : *R*. − *f i*) = − (*INF i* : *R*. *f i*)
  using *ereal-Sup-uminus-image-eq*[*of f'R*]
  by (*simp add*: *image-image*)

lemma *ereal-SUP-not-infty*:
  fixes *f* :: - ⇒ *ereal*
  shows $A \neq \{\} \Longrightarrow l \neq -\infty \Longrightarrow u \neq \infty \Longrightarrow \forall a{\in}A.\ l \leq f\ a \wedge f\ a \leq u \Longrightarrow |SUPREMUM\ A\ f| \neq \infty$
  using *SUP-upper2*[*of* - *A l f*] *SUP-least*[*of A f u*]
  by (*cases SUPREMUM A f*) *auto*

lemma *ereal-INF-not-infty*:
  fixes *f* :: - ⇒ *ereal*
  shows $A \neq \{\} \Longrightarrow l \neq -\infty \Longrightarrow u \neq \infty \Longrightarrow \forall a{\in}A.\ l \leq f\ a \wedge f\ a \leq u \Longrightarrow |INFIMUM\ A\ f| \neq \infty$
  using *INF-lower2*[*of* - *A f u*] *INF-greatest*[*of A l f*]
  by (*cases INFIMUM A f*) *auto*

lemma *ereal-image-uminus-shift*:
  fixes *X Y* :: *ereal set*
  shows *uminus* ' *X* = *Y* ⟷ *X* = *uminus* ' *Y*

proof
  assume *uminus* ‘ $X = Y$
  then have *uminus* ‘ *uminus* ‘ $X =$ *uminus* ‘ Y
    by (*simp add*: *inj-image-eq-iff*)
  then show $X =$ *uminus* ‘ Y
    by (*simp add*: *image-image*)
qed (*simp add*: *image-image*)

lemma *Sup-eq-MInfty*:
  fixes S :: *ereal set*
  shows *Sup* $S = -\infty \longleftrightarrow S = \{\} \vee S = \{-\infty\}$
  unfolding *bot-ereal-def*[*symmetric*] **by** *auto*

lemma *Inf-eq-PInfty*:
  fixes S :: *ereal set*
  shows *Inf* $S = \infty \longleftrightarrow S = \{\} \vee S = \{\infty\}$
  using *Sup-eq-MInfty*[*of uminus*‘*S*]
  unfolding *ereal-Sup-uminus-image-eq ereal-image-uminus-shift* **by** *simp*

lemma *Inf-eq-MInfty*:
  fixes S :: *ereal set*
  shows $-\infty \in S \Longrightarrow$ *Inf* $S = -\infty$
  unfolding *bot-ereal-def*[*symmetric*] **by** *auto*

lemma *Sup-eq-PInfty*:
  fixes S :: *ereal set*
  shows $\infty \in S \Longrightarrow$ *Sup* $S = \infty$
  unfolding *top-ereal-def*[*symmetric*] **by** *auto*

lemma *not-MInfty-nonneg*[*simp*]: $0 \le (x$::*ereal*$) \Longrightarrow x \neq -\infty$
  by *auto*

lemma *Sup-ereal-close*:
  fixes e :: *ereal*
  assumes $0 < e$
    and S: $|$*Sup* $S| \neq \infty$ $S \neq \{\}$
  shows $\exists x \in S.$ *Sup* $S - e < x$
  using *assms* **by** (*cases e*) (*auto intro*!: *less-Sup-iff*[*THEN iffD1*])

lemma *Inf-ereal-close*:
  fixes e :: *ereal*
  assumes $|$*Inf* $X| \neq \infty$
    and $0 < e$
  shows $\exists x \in X.$ $x <$ *Inf* $X + e$
proof (*rule Inf-less-iff*[*THEN iffD1*])
  show *Inf* $X <$ *Inf* $X + e$
    using *assms* **by** (*cases e*) *auto*
qed

lemma *SUP-PInfty*:
($\bigwedge n$::*nat*. $\exists\, i \in A$. *ereal* (*real* n) $\leq f\ i$) $\Longrightarrow$ (*SUP* i:A. $f\ i$ :: *ereal*) $= \infty$
unfolding *top-ereal-def* [*symmetric*] *SUP-eq-top-iff*
by (*metis MInfty-neq-PInfty*(*2*) *PInfty-neq-ereal*(*2*) *less-PInf-Ex-of-nat less-ereal.elims*(*2*) *less-le-trans*)

lemma *SUP-nat-Infty*: (*SUP* i::*nat*. *ereal* (*real* i)) $= \infty$
by (*rule SUP-PInfty*) *auto*

lemma *SUP-ereal-add-left*:
assumes $I \neq \{\}$ $c \neq -\infty$
shows (*SUP* i:I. $f\ i$ + c :: *ereal*) = (*SUP* i:I. $f\ i$) + c
proof *cases*
assume (*SUP* i:I. $f\ i$) $= -\ \infty$
moreover then have $\bigwedge i.\ i \in I \Longrightarrow f\ i = -\infty$
unfolding *Sup-eq-MInfty* **by** *auto*
ultimately show *?thesis*
by (*cases* c) (*auto simp*: ‹$I \neq \{\}$›)
next
assume (*SUP* i:I. $f\ i$) $\neq -\ \infty$ **then show** *?thesis*
by (*subst continuous-at-Sup-mono*[**where** $f = \lambda x.\ x + c$])
(*auto simp*: *continuous-at-imp-continuous-at-within continuous-at mono-def ereal-add-mono* ‹$I \neq \{\}$› ‹$c \neq -\infty$›)
qed

lemma *SUP-ereal-add-right*:
fixes c :: *ereal*
shows $I \neq \{\} \Longrightarrow c \neq -\infty \Longrightarrow$ (*SUP* i:I. $c + f\ i$) $= c +$ (*SUP* i:I. $f\ i$)
using *SUP-ereal-add-left*[*of I c f*] **by** (*simp add*: *add.commute*)

lemma *SUP-ereal-minus-right*:
assumes $I \neq \{\}$ $c \neq -\infty$
shows (*SUP* i:I. $c - f\ i$ :: *ereal*) $= c -$ (*INF* i:I. $f\ i$)
using *SUP-ereal-add-right*[*OF assms*, *of* $\lambda i.\ -\ f\ i$]
by (*simp add*: *ereal-SUP-uminus minus-ereal-def*)

lemma *SUP-ereal-minus-left*:
assumes $I \neq \{\}$ $c \neq \infty$
shows (*SUP* i:I. $f\ i - c$:: *ereal*) = (*SUP* i:I. $f\ i$) $-\ c$
using *SUP-ereal-add-left*[*OF* ‹$I \neq \{\}$›, *of* $-c\ f$] **by** (*simp add*: ‹$c \neq \infty$› *minus-ereal-def*)

lemma *INF-ereal-minus-right*:
assumes $I \neq \{\}$ **and** $|c| \neq \infty$
shows (*INF* i:I. $c - f\ i$) $= c -$ (*SUP* i:I. $f\ i$::*ereal*)
proof −
{ fix b **have** $(-c) + b = -\ (c\ -\ b)$
using ‹$|c| \neq \infty$› **by** (*cases c b rule*: *ereal2-cases*) *auto* **}**
note $*$ = *this*
show *?thesis*

using *SUP-ereal-add-right*[*OF* ‹$I \neq \{\}$›, *of* $-c$ *f*] ‹$|c| \neq \infty$›
by (*auto simp add*: $*$ *ereal-SUP-uminus-eq*)
qed

lemma *SUP-ereal-le-addI*:
fixes $f :: 'i \Rightarrow ereal$
assumes $\bigwedge i.\ f\ i + y \leq z$ **and** $y \neq -\infty$
shows *SUPREMUM UNIV f* $+\ y \leq z$
unfolding *SUP-ereal-add-left*[*OF UNIV-not-empty* ‹$y \neq -\infty$›, *symmetric*]
by (*rule SUP-least assms*)+

lemma *SUP-combine*:
fixes f :: $'a$::*semilattice-sup* $\Rightarrow$ $'a$::*semilattice-sup* $\Rightarrow$ $'b$::*complete-lattice*
assumes *mono*: $\bigwedge a\ b\ c\ d.\ a \leq b \Longrightarrow c \leq d \Longrightarrow f\ a\ c \leq f\ b\ d$
shows (*SUP i*:*UNIV*. *SUP j*:*UNIV*. *f i j*) = (*SUP i*. *f i i*)
proof (*rule antisym*)
show (*SUP i j*. *f i j*) $\leq$ (*SUP i*. *f i i*)
by (*rule SUP-least SUP-upper2*[**where** *i*=*sup i j* **for** *i j*] *UNIV-I mono sup-ge1 sup-ge2*)+
show (*SUP i*. *f i i*) $\leq$ (*SUP i j*. *f i j*)
by (*rule SUP-least SUP-upper2 UNIV-I mono order-refl*)+
qed

lemma *SUP-ereal-add*:
fixes $f\ g :: nat \Rightarrow ereal$
assumes *inc*: *incseq f incseq g*
and *pos*: $\bigwedge i.\ f\ i \neq -\infty$ $\bigwedge i.\ g\ i \neq -\infty$
shows (*SUP i*. $f\ i + g\ i$) = *SUPREMUM UNIV f* + *SUPREMUM UNIV g*
apply (*subst SUP-ereal-add-left*[*symmetric*, *OF UNIV-not-empty*])
apply (*metis SUP-upper UNIV-I assms*(*4*) *ereal-infty-less-eq*(*2*))
apply (*subst* (*2*) *add.commute*)
apply (*subst SUP-ereal-add-left*[*symmetric*, *OF UNIV-not-empty assms*(*3*)])
apply (*subst* (*2*) *add.commute*)
apply (*rule SUP-combine*[*symmetric*] *ereal-add-mono inc*[*THEN monoD*] | *assumption*)+
done

lemma *INF-ereal-add*:
fixes $f :: nat \Rightarrow ereal$
assumes *decseq f decseq g*
and *fin*: $\bigwedge i.\ f\ i \neq \infty$ $\bigwedge i.\ g\ i \neq \infty$
shows (*INF i*. $f\ i + g\ i$) = *INFIMUM UNIV f* + *INFIMUM UNIV g*
proof −
have *INF-less*: (*INF i*. *f i*) $< \infty$ (*INF i*. *g i*) $< \infty$
using *assms* **unfolding** *INF-less-iff* **by** *auto*
{ fix *a b* :: *ereal* **assume** $a \neq \infty$ $b \neq \infty$
then have $-\ ((-\ a) + (-\ b)) = a + b$
by (*cases a b rule*: *ereal2-cases*) *auto* **}**
note $*$ = *this*

have (*INF i. f i + g i*) = (*INF i.* − ((− *f i*) + (− *g i*)))
by (*simp add*: *fin* ∗)
also have . . . = *INFIMUM UNIV f* + *INFIMUM UNIV g*
unfolding *ereal-INF-uminus-eq*
using *assms INF-less*
by (*subst SUP-ereal-add*) (*auto simp*: *ereal-SUP-uminus fin* ∗)
finally show *?thesis* **.**
qed

lemma *SUP-ereal-add-pos*:
fixes *f g* :: *nat* ⇒ *ereal*
assumes *inc*: *incseq f incseq g*
and *pos*: ⋀*i. 0* ≤ *f i* ⋀*i. 0* ≤ *g i*
shows (*SUP i. f i* + *g i*) = *SUPREMUM UNIV f* + *SUPREMUM UNIV g*
proof (*intro SUP-ereal-add inc*)
fix *i*
show *f i* ≠ −∞ *g i* ≠ −∞
using *pos*[*of i*] **by** *auto*
qed

lemma *SUP-ereal-setsum*:
fixes *f g* :: ′*a* ⇒ *nat* ⇒ *ereal*
assumes ⋀*n. n* ∈ *A* ⟹ *incseq* (*f n*)
and *pos*: ⋀*n i. n* ∈ *A* ⟹ *0* ≤ *f n i*
shows (*SUP i.* ∑*n*∈*A. f n i*) = (∑*n*∈*A. SUPREMUM UNIV* (*f n*))
proof (*cases finite A*)
case *True*
then show *?thesis* **using** *assms*
by *induct* (*auto simp*: *incseq-setsumI2 setsum-nonneg SUP-ereal-add-pos*)
next
case *False*
then show *?thesis* **by** *simp*
qed

lemma *SUP-ereal-mult-left*:
fixes *f* :: ′*a* ⇒ *ereal*
assumes *I* ≠ {}
assumes *f*: ⋀*i. i* ∈ *I* ⟹ *0* ≤ *f i* **and** *c*: *0* ≤ *c*
shows (*SUP i*:*I. c* ∗ *f i*) = *c* ∗ (*SUP i*:*I. f i*)
proof *cases*
assume (*SUP i*: *I. f i*) = *0*
moreover then have ⋀*i. i* ∈ *I* ⟹ *f i* = *0*
by (*metis SUP-upper f antisym*)
ultimately show *?thesis*
by *simp*
next
assume (*SUP i*:*I. f i*) ≠ *0* **then show** *?thesis*
by (*subst continuous-at-Sup-mono*[**where** *f*=λ*x. c* ∗ *x*])
(*auto simp*: *mono-def continuous-at continuous-at-imp-continuous-at-within*

‹$I \neq \{\}$›
intro!: ereal-mult-left-mono c)
qed

lemma *countable-approach*:
fixes x :: *ereal*
assumes $x \neq -\infty$
shows $\exists f.\ incseq\ f \wedge (\forall i{::}nat.\ f\ i < x) \wedge (f \longrightarrow x)$
proof (*cases x*)
case (*real r*)
moreover have $(\lambda n.\ r - inverse\ (real\ (Suc\ n))) \longrightarrow r - 0$
by (*intro tendsto-intros LIMSEQ-inverse-real-of-nat*)
ultimately show *?thesis*
by (*intro exI[of - λn. x − inverse (Suc n)]*) (*auto simp*: *incseq-def*)
next
case *PInf* **with** *LIMSEQ-SUP[of λn::nat. ereal (real n)]* **show** *?thesis*
by (*intro exI[of - λn. ereal (real n)]*) (*auto simp*: *incseq-def SUP-nat-Infty*)
qed (*simp add*: *assms*)

lemma *Sup-countable-SUP*:
assumes $A \neq \{\}$
shows $\exists f{::}nat \Rightarrow ereal.\ incseq\ f \wedge range\ f \subseteq A \wedge Sup\ A = (SUP\ i.\ f\ i)$
proof *cases*
assume $Sup\ A = -\infty$
with ‹$A \neq \{\}$› **have** $A = \{-\infty\}$
by (*auto simp*: *Sup-eq-MInfty*)
then show *?thesis*
by (*auto intro*!: *exI[of - λ-. −∞]* *simp*: *bot-ereal-def*)
next
assume $Sup\ A \neq -\infty$
then obtain l **where** *incseq l* **and** l: $\bigwedge i{::}nat.\ l\ i < Sup\ A$ **and** *l-Sup*: $l \longrightarrow Sup\ A$
by (*auto dest*: *countable-approach*)

have $\exists f.\ \forall n.\ (f\ n \in A \wedge l\ n \leq f\ n) \wedge (f\ n \leq f\ (Suc\ n))$
proof (*rule dependent-nat-choice*)
show $\exists x.\ x \in A \wedge l\ 0 \leq x$
using *l[of 0]* **by** (*auto simp*: *less-Sup-iff*)
next
fix $x\ n$ **assume** $x \in A \wedge l\ n \leq x$
moreover from *l[of Suc n]* **obtain** y **where** $y \in A\ l\ (Suc\ n) < y$
by (*auto simp*: *less-Sup-iff*)
ultimately show $\exists y.\ (y \in A \wedge l\ (Suc\ n) \leq y) \wedge x \leq y$
by (*auto intro*!: *exI[of - max x y]* *split*: *split-max*)
qed
then guess f .. **note** $f = this$
then have $range\ f \subseteq A\ incseq\ f$
by (*auto simp*: *incseq-Suc-iff*)
moreover

have (*SUP i. f i*) = *Sup A*
proof (*rule tendsto-unique*)
show *f* ⟶ (*SUP i. f i*)
by (*rule LIMSEQ-SUP* ‹*incseq f*›)+
show *f* ⟶ *Sup A*
using *l f*
by (*intro tendsto-sandwich*[*OF - - l-Sup tendsto-const*])
(*auto simp*: *Sup-upper*)
qed *simp*
ultimately show *?thesis*
by *auto*
qed

lemma *SUP-countable-SUP*:
A ≠ {} ⟹ ∃*f*::*nat* ⇒ *ereal*. *range f* ⊆ *g*‘*A* ∧ *SUPREMUM A g* = *SUPREMUM UNIV f*
using *Sup-countable-SUP* [*of g*‘*A*] **by** *auto*

33.3 Relation to *enat*

definition *ereal-of-enat n* = (*case n of enat n* ⇒ *ereal* (*real n*) | ∞ ⇒ ∞)

declare [[*coercion ereal-of-enat* :: *enat* ⇒ *ereal*]]
declare [[*coercion* (λ*n*. *ereal* (*real n*)) :: *nat* ⇒ *ereal*]]

lemma *ereal-of-enat-simps*[*simp*]:
ereal-of-enat (*enat n*) = *ereal n*
ereal-of-enat ∞ = ∞
by (*simp-all add*: *ereal-of-enat-def*)

lemma *ereal-of-enat-le-iff*[*simp*]: *ereal-of-enat m* ≤ *ereal-of-enat n* ⟷ *m* ≤ *n*
by (*cases m n rule*: *enat2-cases*) *auto*

lemma *ereal-of-enat-less-iff*[*simp*]: *ereal-of-enat m* < *ereal-of-enat n* ⟷ *m* < *n*
by (*cases m n rule*: *enat2-cases*) *auto*

lemma *numeral-le-ereal-of-enat-iff*[*simp*]: *numeral m* ≤ *ereal-of-enat n* ⟷ *numeral m* ≤ *n*
by (*cases n*) (*auto*)

lemma *numeral-less-ereal-of-enat-iff*[*simp*]: *numeral m* < *ereal-of-enat n* ⟷ *numeral m* < *n*
by (*cases n*) *auto*

lemma *ereal-of-enat-ge-zero-cancel-iff*[*simp*]: *0* ≤ *ereal-of-enat n* ⟷ *0* ≤ *n*
by (*cases n*) (*auto simp*: *enat-0*[*symmetric*])

lemma *ereal-of-enat-gt-zero-cancel-iff*[*simp*]: *0* < *ereal-of-enat n* ⟷ *0* < *n*
by (*cases n*) (*auto simp*: *enat-0*[*symmetric*])

lemma *ereal-of-enat-zero*[*simp*]: *ereal-of-enat 0* = *0*
  by (*auto simp*: *enat-0*[*symmetric*])

lemma *ereal-of-enat-inf*[*simp*]: *ereal-of-enat n* = ∞ ⟷ *n* = ∞
  by (*cases n*) *auto*

lemma *ereal-of-enat-add*: *ereal-of-enat* (*m* + *n*) = *ereal-of-enat m* + *ereal-of-enat n*
  by (*cases m n rule*: *enat2-cases*) *auto*

lemma *ereal-of-enat-sub*:
  assumes *n* ≤ *m*
  shows *ereal-of-enat* (*m* − *n*) = *ereal-of-enat m* − *ereal-of-enat n*
  using *assms* **by** (*cases m n rule*: *enat2-cases*) *auto*

lemma *ereal-of-enat-mult*:
  ereal-of-enat (*m* ∗ *n*) = *ereal-of-enat m* ∗ *ereal-of-enat n*
  by (*cases m n rule*: *enat2-cases*) *auto*

lemmas *ereal-of-enat-pushin* = *ereal-of-enat-add ereal-of-enat-sub ereal-of-enat-mult*
lemmas *ereal-of-enat-pushout* = *ereal-of-enat-pushin*[*symmetric*]

lemma *ereal-of-enat-nonneg*: *ereal-of-enat n* ≥ *0*
by(*cases n*) *simp-all*

lemma *ereal-of-enat-Sup*:
  assumes *A* ≠ {} **shows** *ereal-of-enat* (*Sup A*) = (*SUP a* : *A*. *ereal-of-enat a*)
proof (*intro antisym mono-Sup*)
  show *ereal-of-enat* (*Sup A*) ≤ (*SUP a* : *A*. *ereal-of-enat a*)
  proof *cases*
    assume *finite A*
    with ‹*A* ≠ {}› **obtain** *a* **where** *a* ∈ *A ereal-of-enat* (*Sup A*) = *ereal-of-enat a*
      using *Max-in*[*of A*] **by** (*auto simp*: *Sup-enat-def simp del*: *Max-in*)
    then show *?thesis*
      by (*auto intro*: *SUP-upper*)
  next
    assume ¬ *finite A*
    have [*simp*]: (*SUP a* : *A*. *ereal-of-enat a*) = *top*
      unfolding *SUP-eq-top-iff*
    proof *safe*
      fix *x* :: *ereal* **assume** *x* < *top*
      then obtain *n* :: *nat* **where** *x* < *n*
        using *less-PInf-Ex-of-nat top-ereal-def* **by** *auto*
      obtain *a* **where** *a* ∈ *A* − *enat* ‘ {.. *n*}
        by (*metis* ‹¬ *finite A*› *all-not-in-conv finite-Diff2 finite-atMost finite-imageI finite.emptyI*)
      then have *a* ∈ *A ereal n* ≤ *ereal-of-enat a*
        by (*auto simp*: *image-iff Ball-def*)

(*metis enat-iless enat-ord-simps*(*1*) *ereal-of-enat-less-iff ereal-of-enat-simps*(*1*) *less-le not-less*)
with ⟨$x < n$⟩ **show** $\exists\, i{\in}A.\ x <$ *ereal-of-enat i*
by (*auto intro*!: *bexI*[*of - a*])
qed
show *?thesis*
by *simp*
qed
qed (*simp add*: *mono-def*)

lemma *ereal-of-enat-SUP*:
$A \neq \{\} \Longrightarrow$ *ereal-of-enat* (*SUP a*:*A*. *f a*) = (*SUP a* : *A*. *ereal-of-enat* (*f a*))
using *ereal-of-enat-Sup*[*of f'A*] **by** *auto*

33.4 Limits on *ereal*

lemma *open-PInfty*: *open A* $\Longrightarrow \infty \in A \Longrightarrow (\exists\, x.\ \{$*ereal x*$<..\} \subseteq A)$
unfolding *open-ereal-generated*
proof (*induct rule*: *generate-topology.induct*)
case (*Int A B*)
then obtain *x z* **where** $\infty \in A \Longrightarrow \{$*ereal x* $<..\} \subseteq A$ $\infty \in B \Longrightarrow \{$*ereal z* $<..\} \subseteq B$
by *auto*
with *Int* **show** *?case*
by (*intro exI*[*of - max x z*]) *fastforce*
next
case (*Basis S*)
{
fix *x*
have $x \neq \infty \Longrightarrow \exists\, t.\ x \leq$ *ereal t*
by (*cases x*) *auto*
}
moreover note *Basis*
ultimately show *?case*
by (*auto split*: *ereal.split*)
qed (*fastforce simp add*: *vimage-Union*)+

lemma *open-MInfty*: *open A* $\Longrightarrow -\infty \in A \Longrightarrow (\exists\, x.\ \{..<$*ereal x*$\} \subseteq A)$
unfolding *open-ereal-generated*
proof (*induct rule*: *generate-topology.induct*)
case (*Int A B*)
then obtain *x z* **where** $-\infty \in A \Longrightarrow \{..<$ *ereal x*$\} \subseteq A$ $-\infty \in B \Longrightarrow \{..<$ *ereal z*$\} \subseteq B$
by *auto*
with *Int* **show** *?case*
by (*intro exI*[*of - min x z*]) *fastforce*
next
case (*Basis S*)
{

```
    fix x
    have x ≠ − ∞ ⟹ ∃ t. ereal t ≤ x
      by (cases x) auto
  }
  moreover note Basis
  ultimately show ?case
    by (auto split: ereal.split)
qed (fastforce simp add: vimage-Union)+

lemma open-ereal-vimage: open S ⟹ open (ereal −‘ S)
  by (intro open-vimage continuous-intros)

lemma open-ereal: open S ⟹ open (ereal ‘ S)
  unfolding open-generated-order[where 'a=real]
proof (induct rule: generate-topology.induct)
  case (Basis S)
  moreover {
    fix x
    have ereal ‘ {..< x} = { −∞ <..< ereal x }
      apply auto
      apply (case-tac xa)
      apply auto
      done
  }
  moreover {
    fix x
    have ereal ‘ {x <..} = { ereal x <..< ∞ }
      apply auto
      apply (case-tac xa)
      apply auto
      done
  }
  ultimately show ?case
     by auto
qed (auto simp add: image-Union image-Int)


lemma eventually-finite:
  fixes x :: ereal
  assumes |x| ≠ ∞ (f ⟶ x) F
  shows eventually (λx. |f x| ≠ ∞) F
proof −
  have (f ⟶ ereal (real-of-ereal x)) F
    using assms by (cases x) auto
  then have eventually (λx. f x ∈ ereal ‘ UNIV) F
    by (rule topological-tendstoD) (auto intro: open-ereal)
  also have (λx. f x ∈ ereal ‘ UNIV) = (λx. |f x| ≠ ∞)
    by auto
  finally show ?thesis .
```

qed

lemma *open-ereal-def*:
open A ⟷ *open* (*ereal* −' *A*) ∧ (∞ ∈ *A* ⟶ (∃ *x*. {*ereal x* <..} ⊆ *A*)) ∧ (−∞ ∈ *A* ⟶ (∃ *x*. {..<*ereal x*} ⊆ *A*))
(**is** *open A* ⟷ *?rhs*)
proof
assume *open A*
then show *?rhs*
using *open-PInfty open-MInfty open-ereal-vimage* **by** *auto*
next
assume *?rhs*
then obtain *x y* **where** *A*: *open* (*ereal* −' *A*) ∞ ∈ *A* ⟹ {*ereal x*<..} ⊆ *A* −∞ ∈ *A* ⟹ {..< *ereal y*} ⊆ *A*
by *auto*
have ∗: *A* = *ereal* ' (*ereal* −' *A*) ∪ (*if* ∞ ∈ *A then* {*ereal x*<..} *else* {}) ∪ (*if* −∞ ∈ *A then* {..< *ereal y*} *else* {})
using *A*(*2*,*3*) **by** *auto*
from *open-ereal*[*OF A*(*1*)] **show** *open A*
by (*subst* ∗) (*auto simp*: *open-Un*)
qed

lemma *open-PInfty2*:
assumes *open A*
and ∞ ∈ *A*
obtains *x* **where** {*ereal x*<..} ⊆ *A*
using *open-PInfty*[*OF assms*] **by** *auto*

lemma *open-MInfty2*:
assumes *open A*
and −∞ ∈ *A*
obtains *x* **where** {..<*ereal x*} ⊆ *A*
using *open-MInfty*[*OF assms*] **by** *auto*

lemma *ereal-openE*:
assumes *open A*
obtains *x y* **where** *open* (*ereal* −' *A*)
and ∞ ∈ *A* ⟹ {*ereal x*<..} ⊆ *A*
and −∞ ∈ *A* ⟹ {..<*ereal y*} ⊆ *A*
using *assms open-ereal-def* **by** *auto*

lemmas *open-ereal-lessThan* = *open-lessThan*[**where** '*a*=*ereal*]
lemmas *open-ereal-greaterThan* = *open-greaterThan*[**where** '*a*=*ereal*]
lemmas *ereal-open-greaterThanLessThan* = *open-greaterThanLessThan*[**where** '*a*=*ereal*]
lemmas *closed-ereal-atLeast* = *closed-atLeast*[**where** '*a*=*ereal*]
lemmas *closed-ereal-atMost* = *closed-atMost*[**where** '*a*=*ereal*]
lemmas *closed-ereal-atLeastAtMost* = *closed-atLeastAtMost*[**where** '*a*=*ereal*]
lemmas *closed-ereal-singleton* = *closed-singleton*[**where** '*a*=*ereal*]

lemma *ereal-open-cont-interval*:
  fixes S :: *ereal set*
  assumes *open* S
    and $x \in S$
    and $|x| \neq \infty$
  obtains e **where** $e > 0$ **and** $\{x-e <..< x+e\} \subseteq S$
proof −
  from ⟨*open* S⟩
  have *open* (*ereal* −' S)
    by (*rule ereal-openE*)
  then obtain e **where** $e > 0$ **and** e: $\bigwedge y.$ *dist* y (*real-of-ereal* x) $< e \Longrightarrow$ *ereal* $y \in S$
    using *assms* **unfolding** *open-dist* **by** *force*
  show *thesis*
  proof (*intro that subsetI*)
    show $0 <$ *ereal* e
      using ⟨$0 < e$⟩ **by** *auto*
    fix y
    assume $y \in \{x -$ *ereal* $e<..<x +$ *ereal* $e\}$
    with *assms* **obtain** t **where** $y =$ *ereal* t *dist* t (*real-of-ereal* x) $< e$
      by (*cases* y) (*auto simp*: *dist-real-def*)
    then show $y \in S$
      using e[*of* t] **by** *auto*
  qed
qed

lemma *ereal-open-cont-interval2*:
  fixes S :: *ereal set*
  assumes *open* S
    and $x \in S$
    and x: $|x| \neq \infty$
  obtains a b **where** $a < x$ **and** $x < b$ **and** $\{a <..< b\} \subseteq S$
proof −
  obtain e **where** $0 < e$ $\{x - e<..<x + e\} \subseteq S$
    using *assms* **by** (*rule ereal-open-cont-interval*)
  with *that*[*of* $x - e$ $x + e$] *ereal-between*[*OF* x, *of* e]
  show *thesis*
    by *auto*
qed

33.4.1 Convergent sequences

lemma *lim-real-of-ereal*[*simp*]:
  assumes *lim*: ($f \longrightarrow$ *ereal* x) *net*
  shows (($\lambda x.$ *real-of-ereal* (f x)) $\longrightarrow x$) *net*
proof (*intro topological-tendstoI*)
  fix S
  assume *open* S **and** $x \in S$

then have S: *open S ereal* $x \in$ *ereal* ' S
by (*simp-all add*: *inj-image-mem-iff*)
show *eventually* (λx. *real-of-ereal* ($f\ x$) $\in S$) *net*
by (*auto intro*: *eventually-mono* [*OF lim*[*THEN topological-tendstoD*, *OF open-ereal*, *OF S*]])
qed

lemma *lim-ereal*[*simp*]: ((λn. *ereal* ($f\ n$)) $\longrightarrow$ *ereal* x) *net* $\longleftrightarrow$ ($f \longrightarrow x$) *net*
by (*auto dest*!: *lim-real-of-ereal*)

lemma *convergent-real-imp-convergent-ereal*:
assumes *convergent a*
shows *convergent* (λn. *ereal* ($a\ n$)) **and** *lim* (λn. *ereal* ($a\ n$)) = *ereal* (*lim a*)
proof −
from *assms* **obtain** L **where** L: $a \longrightarrow L$ **unfolding** *convergent-def* **..**
hence *lim*: (λn. *ereal* ($a\ n$)) $\longrightarrow$ *ereal* L **using** *lim-ereal* **by** *auto*
thus *convergent* (λn. *ereal* ($a\ n$)) **unfolding** *convergent-def* **..**
thus *lim* (λn. *ereal* ($a\ n$)) = *ereal* (*lim a*) **using** *lim L limI* **by** *metis*
qed

lemma *tendsto-PInfty*: ($f \longrightarrow \infty$) $F \longleftrightarrow$ ($\forall r$. *eventually* (λx. *ereal* $r < f\ x$) F)
proof −
{
fix l :: *ereal*
assume $\forall r$. *eventually* (λx. *ereal* $r < f\ x$) F
from *this*[*THEN spec*, *of real-of-ereal l*] **have** $l \neq \infty \Longrightarrow$ *eventually* (λx. $l < f\ x$) F
by (*cases l*) (*auto elim*: *eventually-mono*)
}
then show *?thesis*
by (*auto simp*: *order-tendsto-iff*)
qed

lemma *tendsto-PInfty'*: ($f \longrightarrow \infty$) F = ($\forall r>c$. *eventually* (λx. *ereal* $r < f\ x$) F)
proof (*subst tendsto-PInfty*, *intro iffI allI impI*)
assume A: $\forall r>c$. *eventually* (λx. *ereal* $r < f\ x$) F
fix r :: *real*
from A **have** A: *eventually* (λx. *ereal* $r < f\ x$) F **if** $r > c$ **for** r **using** *that* **by** *blast*
show *eventually* (λx. *ereal* $r < f\ x$) F
proof (*cases* $r > c$)
case *False*
hence B: *ereal* $r \leq$ *ereal* ($c + 1$) **by** *simp*
have $c < c + 1$ **by** *simp*
from A[*OF this*] **show** *eventually* (λx. *ereal* $r < f\ x$) F
by *eventually-elim* (*rule le-less-trans*[*OF B*])
qed (*simp add*: A)
qed *simp*

lemma *tendsto-PInfty-eq-at-top*:
  $((\lambda z.\ ereal\ (f\ z)) \longrightarrow \infty)\ F \longleftrightarrow (LIM\ z\ F.\ f\ z :> at\text{-}top)$
  unfolding *tendsto-PInfty filterlim-at-top-dense* **by** *simp*

lemma *tendsto-MInfty*: $(f \longrightarrow -\infty)\ F \longleftrightarrow (\forall r.\ eventually\ (\lambda x.\ f\ x < ereal\ r)\ F)$
  unfolding *tendsto-def*
proof *safe*
  fix S :: *ereal set*
  assume *open* S $-\infty \in S$
  from *open-MInfty*[*OF this*] **obtain** B **where** $\{..<ereal\ B\} \subseteq S$ **..**
  moreover
  assume $\forall r{::}real.\ eventually\ (\lambda z.\ f\ z < r)\ F$
  then have $eventually\ (\lambda z.\ f\ z \in \{..< B\})\ F$
    by *auto*
  ultimately show $eventually\ (\lambda z.\ f\ z \in S)\ F$
    by (*auto elim*!: *eventually-mono*)
next
  fix x
  assume $\forall S.\ open\ S \longrightarrow -\infty \in S \longrightarrow eventually\ (\lambda x.\ f\ x \in S)\ F$
  from *this*[*rule-format*, *of* $\{..< ereal\ x\}$] **show** $eventually\ (\lambda y.\ f\ y < ereal\ x)\ F$
    by *auto*
qed

lemma *tendsto-MInfty'*: $(f \longrightarrow -\infty)\ F = (\forall r<c.\ eventually\ (\lambda x.\ ereal\ r > f\ x)\ F)$
proof (*subst tendsto-MInfty*, *intro iffI allI impI*)
  assume A: $\forall r<c.\ eventually\ (\lambda x.\ ereal\ r > f\ x)\ F$
  fix r :: *real*
  from A **have** A: $eventually\ (\lambda x.\ ereal\ r > f\ x)\ F$ **if** $r < c$ **for** r **using** *that* **by** *blast*
  show $eventually\ (\lambda x.\ ereal\ r > f\ x)\ F$
  proof (*cases* $r < c$)
    case *False*
    hence B: $ereal\ r \geq ereal\ (c - 1)$ **by** *simp*
    have $c > c - 1$ **by** *simp*
    from A[*OF this*] **show** $eventually\ (\lambda x.\ ereal\ r > f\ x)\ F$
      by *eventually-elim* (*erule less-le-trans*[*OF - B*])
  qed (*simp add*: A)
qed *simp*

lemma *Lim-PInfty*: $f \longrightarrow \infty \longleftrightarrow (\forall B.\ \exists N.\ \forall n\geq N.\ f\ n \geq ereal\ B)$
  unfolding *tendsto-PInfty eventually-sequentially*
proof *safe*
  fix r
  assume $\forall r.\ \exists N.\ \forall n\geq N.\ ereal\ r \leq f\ n$
  then obtain N **where** $\forall n\geq N.\ ereal\ (r + 1) \leq f\ n$
    by *blast*

moreover have *ereal r < ereal (r + 1)*
by *auto*
ultimately show $\exists N. \forall n \geq N.$ *ereal r < f n*
by (*blast intro*: *less-le-trans*)
qed (*blast intro*: *less-imp-le*)

lemma *Lim-MInfty*: $f \longrightarrow -\infty \longleftrightarrow (\forall B. \exists N. \forall n \geq N.$ *ereal* $B \geq f\, n)$
unfolding *tendsto-MInfty eventually-sequentially*
proof *safe*
fix *r*
assume $\forall r. \exists N. \forall n \geq N. f\, n \leq$ *ereal r*
then obtain *N* **where** $\forall n \geq N. f\, n \leq$ *ereal* $(r - 1)$
by *blast*
moreover have *ereal* $(r - 1) <$ *ereal r*
by *auto*
ultimately show $\exists N. \forall n \geq N. f\, n <$ *ereal r*
by (*blast intro*: *le-less-trans*)
qed (*blast intro*: *less-imp-le*)

lemma *Lim-bounded-PInfty*: $f \longrightarrow l \Longrightarrow (\bigwedge n. f\, n \leq$ *ereal B*$) \Longrightarrow l \neq \infty$
using *LIMSEQ-le-const2*[*of f l ereal B*] **by** *auto*

lemma *Lim-bounded-MInfty*: $f \longrightarrow l \Longrightarrow (\bigwedge n.$ *ereal* $B \leq f\, n) \Longrightarrow l \neq -\infty$
using *LIMSEQ-le-const*[*of f l ereal B*] **by** *auto*

lemma *tendsto-zero-erealI*:
assumes $\bigwedge e. e > 0 \Longrightarrow$ *eventually* $(\lambda x. |f\, x| <$ *ereal e*$)$ *F*
shows $(f \longrightarrow 0)$ *F*
proof (*subst filterlim-cong*[*OF refl refl*])
from *assms*[*OF zero-less-one*] **show** *eventually* $(\lambda x. f\, x =$ *ereal* (*real-of-ereal* $(f\, x))) F$
by *eventually-elim* (*auto simp*: *ereal-real*)
hence *eventually* $(\lambda x.$ *abs* (*real-of-ereal* $(f\, x)) < e)$ *F* **if** $e > 0$ **for** *e* **using** *assms*[*OF that*]
by *eventually-elim* (*simp add*: *real-less-ereal-iff that*)
hence $((\lambda x.$ *real-of-ereal* $(f\, x)) \longrightarrow 0)$ *F* **unfolding** *tendsto-iff*
by (*auto simp*: *tendsto-iff dist-real-def*)
thus $((\lambda x.$ *ereal* (*real-of-ereal* $(f\, x))) \longrightarrow 0)$ *F* **by** (*simp add*: *zero-ereal-def*)
qed

lemma *tendsto-explicit*:
$f \longrightarrow f0 \longleftrightarrow (\forall S.$ *open* $S \longrightarrow f0 \in S \longrightarrow (\exists N. \forall n \geq N. f\, n \in S))$
unfolding *tendsto-def eventually-sequentially* **by** *auto*

lemma *Lim-bounded-PInfty2*: $f \longrightarrow l \Longrightarrow \forall n \geq N. f\, n \leq$ *ereal* $B \Longrightarrow l \neq \infty$
using *LIMSEQ-le-const2*[*of f l ereal B*] **by** *fastforce*

lemma *Lim-bounded-ereal*: $f \longrightarrow (l :: 'a::$*linorder-topology*$) \Longrightarrow \forall n \geq M. f\, n \leq C \Longrightarrow l \leq C$

by (*intro LIMSEQ-le-const2*) *auto*

lemma *Lim-bounded2-ereal*:
assumes *lim*:$f \longrightarrow (l :: 'a::linorder\text{-}topology)$
and *ge*: $\forall n \geq N.\ f\ n \geq C$
shows $l \geq C$
using *ge*
by (*intro tendsto-le*[*OF trivial-limit-sequentially lim tendsto-const*])
(*auto simp*: *eventually-sequentially*)

lemma *real-of-ereal-mult*[*simp*]:
fixes *a b* :: *ereal*
shows *real-of-ereal* $(a * b)$ = *real-of-ereal* a * *real-of-ereal* b
by (*cases rule*: *ereal2-cases*[*of a b*]) *auto*

lemma *real-of-ereal-eq-0*:
fixes *x* :: *ereal*
shows *real-of-ereal* $x = 0 \longleftrightarrow x = \infty \lor x = -\infty \lor x = 0$
by (*cases x*) *auto*

lemma *tendsto-ereal-realD*:
fixes $f :: 'a \Rightarrow$ *ereal*
assumes $x \neq 0$
and *tendsto*: $((\lambda x.\ ereal\ (real\text{-}of\text{-}ereal\ (f\ x))) \longrightarrow x)\ net$
shows $(f \longrightarrow x)\ net$
proof (*intro topological-tendstoI*)
fix *S*
assume *S*: *open S* $x \in S$
with ‹$x \neq 0$› **have** *open* $(S - \{0\})$ $x \in S - \{0\}$
by *auto*
from *tendsto*[*THEN topological-tendstoD*, *OF this*]
show *eventually* $(\lambda x.\ f\ x \in S)$ *net*
by (*rule eventually-rev-mp*) (*auto simp*: *ereal-real*)
qed

lemma *tendsto-ereal-realI*:
fixes $f :: 'a \Rightarrow$ *ereal*
assumes *x*: $|x| \neq \infty$ **and** *tendsto*: $(f \longrightarrow x)$ *net*
shows $((\lambda x.\ ereal\ (real\text{-}of\text{-}ereal\ (f\ x))) \longrightarrow x)\ net$
proof (*intro topological-tendstoI*)
fix *S*
assume *open S* **and** $x \in S$
with *x* **have** *open* $(S - \{\infty, -\infty\})$ $x \in S - \{\infty, -\infty\}$
by *auto*
from *tendsto*[*THEN topological-tendstoD*, *OF this*]
show *eventually* $(\lambda x.\ ereal\ (real\text{-}of\text{-}ereal\ (f\ x)) \in S)$ *net*
by (*elim eventually-mono*) (*auto simp*: *ereal-real*)
qed

lemma *ereal-mult-cancel-left*:
fixes a b c :: *ereal*
shows $a * b = a * c \longleftrightarrow (|a| = \infty \wedge 0 < b * c) \vee a = 0 \vee b = c$
by (*cases rule*: *ereal3-cases*[*of* a b c]) (*simp-all add*: *zero-less-mult-iff*)

lemma *tendsto-add-ereal*:
fixes x y :: *ereal*
assumes x: $|x| \neq \infty$ **and** y: $|y| \neq \infty$
assumes f: $(f \longrightarrow x)$ F **and** g: $(g \longrightarrow y)$ F
shows $((\lambda x.\ f\ x + g\ x) \longrightarrow x + y)$ F
proof −
from x **obtain** r **where** x': $x =$ *ereal* r **by** (*cases* x) *auto*
with f **have** $((\lambda i.$ *real-of-ereal* $(f\ i)) \longrightarrow r)$ F **by** *simp*
moreover
from y **obtain** p **where** y': $y =$ *ereal* p **by** (*cases* y) *auto*
with g **have** $((\lambda i.$ *real-of-ereal* $(g\ i)) \longrightarrow p)$ F **by** *simp*
ultimately have $((\lambda i.$ *real-of-ereal* $(f\ i)$ + *real-of-ereal* $(g\ i)) \longrightarrow r + p)$ F
by (*rule tendsto-add*)
moreover
from *eventually-finite*[*OF* x f] *eventually-finite*[*OF* y g]
have *eventually* $(\lambda x.\ f\ x + g\ x =$ *ereal* (*real-of-ereal* $(f\ x)$ + *real-of-ereal* $(g\ x)))$
F
by *eventually-elim auto*
ultimately show *?thesis*
by (*simp add*: x' y' *cong*: *filterlim-cong*)
qed

lemma *tendsto-add-ereal-nonneg*:
fixes x y :: *ereal*
assumes $x \neq -\infty$ $y \neq -\infty$ $(f \longrightarrow x)$ F $(g \longrightarrow y)$ F
shows $((\lambda x.\ f\ x + g\ x) \longrightarrow x + y)$ F
proof *cases*
assume $x = \infty \vee y = \infty$
moreover
{ fix y :: *ereal* **and** f g :: $'a \Rightarrow$ *ereal* **assume** $y \neq -\infty$ $(f \longrightarrow \infty)$ F $(g \longrightarrow$
$y)$ F
then obtain y' **where** $-\infty < y'$ $y' < y$
using *dense*[*of* $-\infty$ y] **by** *auto*
have $((\lambda x.\ f\ x + g\ x) \longrightarrow \infty)$ F
proof (*rule tendsto-sandwich*)
have $\forall_F$ x *in* F. $y' < g\ x$
using *order-tendstoD*(*1*)[*OF* ‹$(g \longrightarrow y)$ F› ‹$y' < y$›] **by** *auto*
then show $\forall_F$ x *in* F. $f\ x + y' \leq f\ x + g\ x$
by *eventually-elim* (*auto intro*!: *add-mono*)
show $\forall_F$ n *in* F. $f\ n + g\ n \leq \infty$ $((\lambda n.\ \infty) \longrightarrow \infty)$ F
by *auto*
show $((\lambda x.\ f\ x + y') \longrightarrow \infty)$ F
using *tendsto-cadd-ereal*[*of* y' ∞ f F] ‹$(f \longrightarrow \infty)$ F› ‹$-\infty < y'$› **by** *auto*
qed }

note *this*[*of y f g*] *this*[*of x g f*]
ultimately show *?thesis*
using *assms* **by** (*auto simp*: *add-ac*)
next
assume $\neg$ ($x = \infty \vee y = \infty$)
with *assms tendsto-add-ereal*[*of x y f F g*]
show *?thesis*
by *auto*
qed

lemma *ereal-inj-affinity*:
fixes *m t* :: *ereal*
assumes $|m| \neq \infty$
and $m \neq 0$
and $|t| \neq \infty$
shows *inj-on* ($\lambda x.\ m * x + t$) *A*
using *assms*
by (*cases rule*: *ereal2-cases*[*of m t*])
(*auto intro*!: *inj-onI simp*: *ereal-add-cancel-right ereal-mult-cancel-left*)

lemma *ereal-PInfty-eq-plus*[*simp*]:
fixes *a b* :: *ereal*
shows $\infty = a + b \longleftrightarrow a = \infty \vee b = \infty$
by (*cases rule*: *ereal2-cases*[*of a b*]) *auto*

lemma *ereal-MInfty-eq-plus*[*simp*]:
fixes *a b* :: *ereal*
shows $-\infty = a + b \longleftrightarrow (a = -\infty \wedge b \neq \infty) \vee (b = -\infty \wedge a \neq \infty)$
by (*cases rule*: *ereal2-cases*[*of a b*]) *auto*

lemma *ereal-less-divide-pos*:
fixes *x y* :: *ereal*
shows $x > 0 \Longrightarrow x \neq \infty \Longrightarrow y < z\ /\ x \longleftrightarrow x * y < z$
by (*cases rule*: *ereal3-cases*[*of x y z*]) (*auto simp*: *field-simps*)

lemma *ereal-divide-less-pos*:
fixes *x y z* :: *ereal*
shows $x > 0 \Longrightarrow x \neq \infty \Longrightarrow y\ /\ x < z \longleftrightarrow y < x * z$
by (*cases rule*: *ereal3-cases*[*of x y z*]) (*auto simp*: *field-simps*)

lemma *ereal-divide-eq*:
fixes *a b c* :: *ereal*
shows $b \neq 0 \Longrightarrow |b| \neq \infty \Longrightarrow a\ /\ b = c \longleftrightarrow a = b * c$
by (*cases rule*: *ereal3-cases*[*of a b c*])
(*simp-all add*: *field-simps*)

lemma *ereal-inverse-not-MInfty*[*simp*]: *inverse* (*a*::*ereal*) $\neq -\infty$
by (*cases a*) *auto*

lemma *ereal-mult-m1*[*simp*]: $x * ereal\ (-1) = -x$
  by (*cases x*) *auto*

lemma *ereal-real'*:
  assumes $|x| \neq \infty$
  shows *ereal* (*real-of-ereal x*) = *x*
  using *assms* **by** *auto*

lemma *real-ereal-id*: *real-of-ereal* ∘ *ereal* = *id*
proof −
  {
    fix *x*
    have (*real-of-ereal o ereal*) *x* = *id x*
      by *auto*
  }
  then show *?thesis*
    using *ext* **by** *blast*
qed

lemma *open-image-ereal*: *open*(*UNIV* −{ ∞ , (−∞ :: *ereal*)})
  by (*metis range-ereal open-ereal open-UNIV*)

lemma *ereal-le-distrib*:
  fixes *a b c* :: *ereal*
  shows $c * (a + b) \leq c * a + c * b$
  by (*cases rule*: *ereal3-cases*[*of a b c*])
    (*auto simp add*: *field-simps not-le mult-le-0-iff mult-less-0-iff*)

lemma *ereal-pos-distrib*:
  fixes *a b c* :: *ereal*
  assumes $0 \leq c$
    and $c \neq \infty$
  shows $c * (a + b) = c * a + c * b$
  using *assms*
  by (*cases rule*: *ereal3-cases*[*of a b c*])
    (*auto simp add*: *field-simps not-le mult-le-0-iff mult-less-0-iff*)

lemma *ereal-max-mono*: $(a::ereal) \leq b \Longrightarrow c \leq d \Longrightarrow max\ a\ c \leq max\ b\ d$
  by (*metis sup-ereal-def sup-mono*)

lemma *ereal-max-least*: $(a::ereal) \leq x \Longrightarrow c \leq x \Longrightarrow max\ a\ c \leq x$
  by (*metis sup-ereal-def sup-least*)

lemma *ereal-LimI-finite*:
  fixes *x* :: *ereal*
  assumes $|x| \neq \infty$
    and $\bigwedge r.\ 0 < r \Longrightarrow \exists N.\ \forall n \geq N.\ u\ n < x + r \wedge x < u\ n + r$
  shows $u \longlongrightarrow x$
proof (*rule topological-tendstoI*, *unfold eventually-sequentially*)

obtain *rx* **where** *rx*: $x = ereal\ rx$
using *assms* **by** (*cases x*) *auto*
fix *S*
assume *open S* **and** $x \in S$
then have *open* (*ereal* −' *S*)
unfolding *open-ereal-def* **by** *auto*
with ‹$x \in S$› **obtain** *r* **where** $0 < r$ **and** *dist*: $\bigwedge y.\ dist\ y\ rx < r \Longrightarrow ereal\ y \in S$
unfolding *open-dist rx* **by** *auto*
then obtain *n* **where**
upper: $\bigwedge N.\ n \leq N \Longrightarrow u\ N < x + ereal\ r$ **and**
lower: $\bigwedge N.\ n \leq N \Longrightarrow x < u\ N + ereal\ r$
using *assms*(*2*)[*of ereal r*] **by** *auto*
show $\exists N.\ \forall n \geq N.\ u\ n \in S$
proof (*safe intro*!: *exI*[*of* - *n*])
fix *N*
assume $n \leq N$
from *upper*[*OF this*] *lower*[*OF this*] *assms* ‹$0 < r$›
have $u\ N \notin \{\infty,(-\infty)\}$
by *auto*
then obtain *ra* **where** *ra-def*: $(u\ N) = ereal\ ra$
by (*cases u N*) *auto*
then have $rx < ra + r$ **and** $ra < rx + r$
using *rx assms* ‹$0 < r$› *lower*[*OF* ‹$n \leq N$›] *upper*[*OF* ‹$n \leq N$›]
by *auto*
then have *dist* (*real-of-ereal* (*u N*)) $rx < r$
using *rx ra-def*
by (*auto simp*: *dist-real-def abs-diff-less-iff field-simps*)
from *dist*[*OF this*] **show** $u\ N \in S$
using ‹$u\ N \notin \{\infty, -\infty\}$›
by (*auto simp*: *ereal-real split*: *if-split-asm*)
qed
qed

lemma *tendsto-obtains-N*:
assumes $f \longrightarrow f0$
assumes *open S*
and $f0 \in S$
obtains *N* **where** $\forall n \geq N.\ f\ n \in S$
using *assms* **using** *tendsto-def*
using *tendsto-explicit*[*of f f0*] *assms* **by** *auto*

lemma *ereal-LimI-finite-iff*:
fixes *x* :: *ereal*
assumes $|x| \neq \infty$
shows $u \longrightarrow x \longleftrightarrow (\forall r.\ 0 < r \longrightarrow (\exists N.\ \forall n \geq N.\ u\ n < x + r \wedge x < u\ n + r))$
(**is** *?lhs* ⟷ *?rhs*)
proof

assume *lim*: $u \longrightarrow x$
{
fix r :: *ereal*
assume $r > 0$
then obtain N **where** $\forall n \geq N.\ u\ n \in \{x - r <..< x + r\}$
apply (*subst tendsto-obtains-N*[*of* $u\ x\ \{x - r <..< x + r\}$])
using *lim ereal-between*[*of* $x\ r$] *assms* ‹$r > 0$›
apply *auto*
done
then have $\exists N.\ \forall n \geq N.\ u\ n < x + r \wedge x < u\ n + r$
using *ereal-minus-less*[*of* $r\ x$]
by (*cases* r) *auto*
}
then show *?rhs*
by *auto*
next
assume *?rhs*
then show $u \longrightarrow x$
using *ereal-LimI-finite*[*of* x] *assms* **by** *auto*
qed

lemma *ereal-Limsup-uminus*:
fixes f :: *'a* ⇒ *ereal*
shows *Limsup net* $(\lambda x.\ - (f\ x)) = -$ *Liminf net f*
unfolding *Limsup-def Liminf-def ereal-SUP-uminus ereal-INF-uminus-eq* **..**

lemma *liminf-bounded-iff*:
fixes x :: *nat* ⇒ *ereal*
shows $C \leq$ *liminf* $x \longleftrightarrow (\forall B<C.\ \exists N.\ \forall n \geq N.\ B < x\ n)$
(**is** *?lhs* ⟷ *?rhs*)
unfolding *le-Liminf-iff eventually-sequentially* **..**

lemma *Liminf-add-le*:
fixes $f\ g$ :: - ⇒ *ereal*
assumes F: $F \neq$ *bot*
assumes *ev*: *eventually* $(\lambda x.\ 0 \leq f\ x)$ F *eventually* $(\lambda x.\ 0 \leq g\ x)$ F
shows *Liminf* $F\ f$ + *Liminf* $F\ g \leq$ *Liminf* F $(\lambda x.\ f\ x + g\ x)$
unfolding *Liminf-def*
proof (*subst SUP-ereal-add-left*[*symmetric*])
let *?F* = {*P*. *eventually P F*}
let *?INF* = $\lambda P\ g$. *INFIMUM* (*Collect P*) g
show *?F* ≠ {}
by (*auto intro*: *eventually-True*)
show (*SUP P*:*?F*. *?INF P g*) $\neq -\infty$
unfolding *bot-ereal-def*[*symmetric*] *SUP-bot-conv INF-eq-bot-iff*
by (*auto intro*!: *exI*[*of* - *0*] *ev simp*: *bot-ereal-def*)
have (*SUP P*:*?F*. *?INF P f* + (*SUP P*:*?F*. *?INF P g*)) ≤ (*SUP P*:*?F*. (*SUP P'*:*?F*. *?INF P f* + *?INF P' g*))
proof (*safe intro*!: *SUP-mono bexI*[*of* - $\lambda x.\ P\ x \wedge 0 \leq f\ x$ **for** P])

fix *P* **let** *?P'* = λ*x*. *P x* ∧ *0* ≤ *f x*
assume *eventually P F*
with *ev* **show** *eventually ?P' F*
by *eventually-elim auto*
have *?INF P f* + (*SUP P*:*?F*. *?INF P g*) ≤ *?INF ?P' f* + (*SUP P*:*?F*. *?INF P g*)
by (*intro ereal-add-mono INF-mono*) *auto*
also have ... = (*SUP P'*:*?F*. *?INF ?P' f* + *?INF P' g*)
proof (*rule SUP-ereal-add-right*[*symmetric*])
show *INFIMUM* {*x*. *P x* ∧ *0* ≤ *f x*} *f* ≠ − ∞
unfolding *bot-ereal-def*[*symmetric*] *INF-eq-bot-iff*
by (*auto intro*!: *exI*[*of* - *0*] *ev simp*: *bot-ereal-def*)
qed *fact*
finally show *?INF P f* + (*SUP P*:*?F*. *?INF P g*) ≤ (*SUP P'*:*?F*. *?INF ?P' f* + *?INF P' g*) **.**
qed
also have ... ≤ (*SUP P*:*?F*. *INF x*:*Collect P*. *f x* + *g x*)
proof (*safe intro*!: *SUP-least*)
fix *P Q* **assume** ∗: *eventually P F eventually Q F*
show *?INF P f* + *?INF Q g* ≤ (*SUP P*:*?F*. *INF x*:*Collect P*. *f x* + *g x*)
proof (*rule SUP-upper2*)
show (λ*x*. *P x* ∧ *Q x*) ∈ *?F*
using ∗ **by** (*auto simp*: *eventually-conj*)
show *?INF P f* + *?INF Q g* ≤ (*INF x*:{*x*. *P x* ∧ *Q x*}. *f x* + *g x*)
by (*intro INF-greatest ereal-add-mono*) (*auto intro*: *INF-lower*)
qed
qed
finally show (*SUP P*:*?F*. *?INF P f* + (*SUP P*:*?F*. *?INF P g*)) ≤ (*SUP P*:*?F*. *INF x*:*Collect P*. *f x* + *g x*) **.**
qed

lemma *Sup-ereal-mult-right'*:
assumes *nonempty*: *Y* ≠ {}
and *x*: *x* ≥ *0*
shows (*SUP i*:*Y*. *f i*) ∗ *ereal x* = (*SUP i*:*Y*. *f i* ∗ *ereal x*) (**is** *?lhs* = *?rhs*)
proof(*cases x* = *0*)
case *True* **thus** *?thesis* **by**(*auto simp add*: *nonempty zero-ereal-def*[*symmetric*])
next
case *False*
show *?thesis*
proof(*rule antisym*)
show *?rhs* ≤ *?lhs*
by(*rule SUP-least*)(*simp add*: *ereal-mult-right-mono SUP-upper x*)
next
have *?lhs* / *ereal x* = (*SUP i*:*Y*. *f i*) ∗ (*ereal x* / *ereal x*) **by**(*simp only*: *ereal-times-divide-eq*)
also have ... = (*SUP i*:*Y*. *f i*) **using** *False* **by** *simp*
also have ... ≤ *?rhs* / *x*
proof(*rule SUP-least*)

fix *i*
assume $i \in Y$
have $f\ i = f\ i * (ereal\ x\ /\ ereal\ x)$ **using** *False* **by** *simp*
also have $\ldots = f\ i * x\ /\ x$ **by**(*simp only*: *ereal-times-divide-eq*)
also from ‹$i \in Y$› **have** $f\ i * x \leq ?rhs$ **by**(*rule SUP-upper*)
hence $f\ i * x\ /\ x \leq ?rhs\ /\ x$ **using** *x False* **by** *simp*
finally show $f\ i \leq ?rhs\ /\ x$ **.**
qed
finally have $(?lhs\ /\ x) * x \leq (?rhs\ /\ x) * x$
by(*rule ereal-mult-right-mono*)(*simp add*: *x*)
also have $\ldots = ?rhs$ **using** *False ereal-divide-eq mult.commute* **by** *force*
also have $(?lhs\ /\ x) * x = ?lhs$ **using** *False ereal-divide-eq mult.commute* **by** *force*
finally show $?lhs \leq ?rhs$ **.**
qed
qed

lemma *Sup-ereal-mult-left'*:
$[\![\ Y \neq \{\};\ x \geq 0\ ]\!] \Longrightarrow ereal\ x * (SUP\ i{:}Y.\ f\ i) = (SUP\ i{:}Y.\ ereal\ x * f\ i)$
by(*subst* (*1 2*) *mult.commute*)(*rule Sup-ereal-mult-right'*)

lemma *sup-continuous-add*[*order-continuous-intros*]:
fixes $f\ g$ :: *'a*::*complete-lattice* ⇒ *ereal*
assumes *nn*: $\bigwedge x.\ 0 \leq f\ x$ $\bigwedge x.\ 0 \leq g\ x$ **and** *cont*: *sup-continuous f sup-continuous g*
shows *sup-continuous* $(\lambda x.\ f\ x + g\ x)$
unfolding *sup-continuous-def*
proof *safe*
fix *M* :: *nat* ⇒ *'a* **assume** *incseq M*
then show $f\ (SUP\ i.\ M\ i) + g\ (SUP\ i.\ M\ i) = (SUP\ i.\ f\ (M\ i) + g\ (M\ i))$
using *SUP-ereal-add-pos*[*of* $\lambda i.\ f\ (M\ i)$ $\lambda i.\ g\ (M\ i)$] *nn*
cont[*THEN sup-continuous-mono*] *cont*[*THEN sup-continuousD*]
by (*auto simp*: *mono-def*)
qed

lemma *sup-continuous-mult-right*[*order-continuous-intros*]:
$0 \leq c \Longrightarrow c < \infty \Longrightarrow$ *sup-continuous f* $\Longrightarrow$ *sup-continuous* $(\lambda x.\ f\ x * c :: ereal)$
by (*cases c*) (*auto simp*: *sup-continuous-def fun-eq-iff Sup-ereal-mult-right'*)

lemma *sup-continuous-mult-left*[*order-continuous-intros*]:
$0 \leq c \Longrightarrow c < \infty \Longrightarrow$ *sup-continuous f* $\Longrightarrow$ *sup-continuous* $(\lambda x.\ c * f\ x :: ereal)$
using *sup-continuous-mult-right*[*of c f*] **by** (*simp add*: *mult-ac*)

lemma *sup-continuous-ereal-of-enat*[*order-continuous-intros*]:
assumes *f*: *sup-continuous f* **shows** *sup-continuous* $(\lambda x.\ ereal\text{-}of\text{-}enat\ (f\ x))$
by (*rule sup-continuous-compose*[*OF - f*])
(*auto simp*: *sup-continuous-def ereal-of-enat-SUP*)

33.4.2 Sums

lemma *sums-ereal-positive*:
  fixes f :: *nat* ⇒ *ereal*
  assumes $\bigwedge i.\ 0 \leq f\ i$
  shows f *sums* $(SUP\ n.\ \sum i<n.\ f\ i)$
proof −
  have *incseq* $(\lambda i.\ \sum j=0..<i.\ f\ j)$
    using *ereal-add-mono*[*OF* - *assms*]
    by (*auto intro*!: *incseq-SucI*)
  from *LIMSEQ-SUP*[*OF this*]
  show *?thesis* **unfolding** *sums-def*
    by (*simp add*: *atLeast0LessThan*)
qed

lemma *summable-ereal-pos*:
  fixes f :: *nat* ⇒ *ereal*
  assumes $\bigwedge i.\ 0 \leq f\ i$
  shows *summable f*
  using *sums-ereal-positive*[*of f*, *OF assms*]
  unfolding *summable-def*
  by *auto*

lemma *sums-ereal*: $(\lambda x.\ ereal\ (f\ x))$ *sums ereal* $x \longleftrightarrow f$ *sums* x
  unfolding *sums-def* **by** *simp*

lemma *suminf-ereal-eq-SUP*:
  fixes f :: *nat* ⇒ *ereal*
  assumes $\bigwedge i.\ 0 \leq f\ i$
  shows $(\sum x.\ f\ x) = (SUP\ n.\ \sum i<n.\ f\ i)$
  using *sums-ereal-positive*[*of f*, *OF assms*, *THEN sums-unique*]
  by *simp*

lemma *suminf-bound*:
  fixes f :: *nat* ⇒ *ereal*
  assumes $\forall N.\ (\sum n<N.\ f\ n) \leq x$
    and *pos*: $\bigwedge n.\ 0 \leq f\ n$
  shows *suminf f* $\leq x$
proof (*rule Lim-bounded-ereal*)
  have *summable f* **using** *pos*[*THEN summable-ereal-pos*] **.**
  then show $(\lambda N.\ \sum n<N.\ f\ n) \longrightarrow$ *suminf f*
    by (*auto dest*!: *summable-sums simp*: *sums-def atLeast0LessThan*)
  show $\forall n\geq 0.$ *setsum f* $\{..<n\} \leq x$
    using *assms* **by** *auto*
qed

lemma *suminf-bound-add*:
  fixes f :: *nat* ⇒ *ereal*
  assumes $\forall N.\ (\sum n<N.\ f\ n) + y \leq x$
    and *pos*: $\bigwedge n.\ 0 \leq f\ n$

```
    and y ≠ −∞
  shows suminf f + y ≤ x
proof (cases y)
  case (real r)
  then have ∀ N. (∑ n<N. f n) ≤ x − y
    using assms by (simp add: ereal-le-minus)
  then have (∑ n. f n) ≤ x − y
    using pos by (rule suminf-bound)
  then show (∑ n. f n) + y ≤ x
    using assms real by (simp add: ereal-le-minus)
qed (insert assms, auto)

lemma suminf-upper:
  fixes f :: nat ⇒ ereal
  assumes ⋀n. 0 ≤ f n
  shows (∑ n<N. f n) ≤ (∑ n. f n)
  unfolding suminf-ereal-eq-SUP [OF assms]
  by (auto intro: complete-lattice-class.SUP-upper)

lemma suminf-0-le:
  fixes f :: nat ⇒ ereal
  assumes ⋀n. 0 ≤ f n
  shows 0 ≤ (∑ n. f n)
  using suminf-upper[of f 0, OF assms]
  by simp

lemma suminf-le-pos:
  fixes f g :: nat ⇒ ereal
  assumes ⋀N. f N ≤ g N
    and ⋀N. 0 ≤ f N
  shows suminf f ≤ suminf g
proof (safe intro!: suminf-bound)
  fix n
  {
    fix N
    have 0 ≤ g N
      using assms(2,1)[of N] by auto
  }
  have setsum f {..<n} ≤ setsum g {..<n}
    using assms by (auto intro: setsum-mono)
  also have ... ≤ suminf g
    using ‹⋀N. 0 ≤ g N›
    by (rule suminf-upper)
  finally show setsum f {..<n} ≤ suminf g .
qed (rule assms(2))

lemma suminf-half-series-ereal: (∑ n. (1/2 :: ereal) ^ Suc n) = 1
  using sums-ereal[THEN iffD2, OF power-half-series, THEN sums-unique, symmetric]
```

by (*simp add*: *one-ereal-def*)

lemma *suminf-add-ereal*:
fixes *f g* :: *nat* ⇒ *ereal*
assumes $\bigwedge i.\ 0 \leq f\ i$
and $\bigwedge i.\ 0 \leq g\ i$
shows $(\sum i.\ f\ i + g\ i) = suminf\ f + suminf\ g$
apply (*subst* (*1 2 3*) *suminf-ereal-eq-SUP*)
unfolding *setsum.distrib*
apply (*intro assms ereal-add-nonneg-nonneg SUP-ereal-add-pos incseq-setsumI setsum-nonneg ballI*)+
done

lemma *suminf-cmult-ereal*:
fixes *f g* :: *nat* ⇒ *ereal*
assumes $\bigwedge i.\ 0 \leq f\ i$
and $0 \leq a$
shows $(\sum i.\ a * f\ i) = a * suminf\ f$
by (*auto simp*: *setsum-ereal-right-distrib*[*symmetric*] *assms*
ereal-zero-le-0-iff setsum-nonneg suminf-ereal-eq-SUP
intro!: *SUP-ereal-mult-left*)

lemma *suminf-PInfty*:
fixes *f* :: *nat* ⇒ *ereal*
assumes $\bigwedge i.\ 0 \leq f\ i$
and $suminf\ f \neq \infty$
shows $f\ i \neq \infty$
proof −
from *suminf-upper*[*of f Suc i*, *OF assms*(*1*)] *assms*(*2*)
have $(\sum i<Suc\ i.\ f\ i) \neq \infty$
by *auto*
then show *?thesis*
unfolding *setsum-Pinfty* **by** *simp*
qed

lemma *suminf-PInfty-fun*:
assumes $\bigwedge i.\ 0 \leq f\ i$
and $suminf\ f \neq \infty$
shows $\exists f'.\ f = (\lambda x.\ ereal\ (f'\ x))$
proof −
have $\forall i.\ \exists r.\ f\ i = ereal\ r$
proof
fix *i*
show $\exists r.\ f\ i = ereal\ r$
using *suminf-PInfty*[*OF assms*] *assms*(*1*)[*of i*]
by (*cases f i*) *auto*
qed
from *choice*[*OF this*] **show** *?thesis*
by *auto*

qed

lemma *summable-ereal*:
assumes $\bigwedge i.\ 0 \leq f\ i$
and $(\sum i.\ ereal\ (f\ i)) \neq \infty$
shows *summable f*
proof −
have $0 \leq (\sum i.\ ereal\ (f\ i))$
using *assms* **by** (*intro suminf-0-le*) *auto*
with *assms* **obtain** *r* **where** *r*: $(\sum i.\ ereal\ (f\ i)) = ereal\ r$
by (*cases* $\sum i.\ ereal\ (f\ i)$) *auto*
from *summable-ereal-pos*[*of* $\lambda x.\ ereal\ (f\ x)$]
have *summable* $(\lambda x.\ ereal\ (f\ x))$
using *assms* **by** *auto*
from *summable-sums*[*OF this*]
have $(\lambda x.\ ereal\ (f\ x))$ *sums* $(\sum x.\ ereal\ (f\ x))$
by *auto*
then show *summable f*
unfolding *r sums-ereal summable-def* **..**
qed

lemma *suminf-ereal*:
assumes $\bigwedge i.\ 0 \leq f\ i$
and $(\sum i.\ ereal\ (f\ i)) \neq \infty$
shows $(\sum i.\ ereal\ (f\ i)) = ereal\ (suminf\ f)$
proof (*rule sums-unique*[*symmetric*])
from *summable-ereal*[*OF assms*]
show $(\lambda x.\ ereal\ (f\ x))$ *sums* $(ereal\ (suminf\ f))$
unfolding *sums-ereal*
using *assms*
by (*intro summable-sums summable-ereal*)
qed

lemma *suminf-ereal-minus*:
fixes *f g* :: *nat* ⇒ *ereal*
assumes *ord*: $\bigwedge i.\ g\ i \leq f\ i$ $\bigwedge i.\ 0 \leq g\ i$
and *fin*: $suminf\ f \neq \infty$ $suminf\ g \neq \infty$
shows $(\sum i.\ f\ i - g\ i) = suminf\ f - suminf\ g$
proof −
{
fix *i*
have $0 \leq f\ i$
using *ord*[*of i*] **by** *auto*
}
moreover
from *suminf-PInfty-fun*[*OF* ‹$\bigwedge i.\ 0 \leq f\ i$› *fin*(*1*)] **obtain** *f'* **where** [*simp*]: $f = (\lambda x.\ ereal\ (f'\ x))$ **..**
from *suminf-PInfty-fun*[*OF* ‹$\bigwedge i.\ 0 \leq g\ i$› *fin*(*2*)] **obtain** *g'* **where** [*simp*]: $g = (\lambda x.\ ereal\ (g'\ x))$ **..**

```
  {
    fix i
    have 0 ≤ f i − g i
      using ord[of i] by (auto simp: ereal-le-minus-iff)
  }
  moreover
  have suminf (λi. f i − g i) ≤ suminf f
    using assms by (auto intro!: suminf-le-pos simp: field-simps)
  then have suminf (λi. f i − g i) ≠ ∞
    using fin by auto
  ultimately show ?thesis
    using assms ‹⋀i. 0 ≤ f i›
    apply simp
    apply (subst (1 2 3) suminf-ereal)
    apply (auto intro!: suminf-diff[symmetric] summable-ereal)
    done
qed

lemma suminf-ereal-PInf [simp]: (∑x. ∞::ereal) = ∞
proof −
  have (∑i<Suc 0. ∞) ≤ (∑x. ∞::ereal)
    by (rule suminf-upper) auto
  then show ?thesis
    by simp
qed

lemma summable-real-of-ereal:
  fixes f :: nat ⇒ ereal
  assumes f: ⋀i. 0 ≤ f i
    and fin: (∑i. f i) ≠ ∞
  shows summable (λi. real-of-ereal (f i))
proof (rule summable-def[THEN iffD2])
  have 0 ≤ (∑i. f i)
    using assms by (auto intro: suminf-0-le)
  with fin obtain r where r: ereal r = (∑i. f i)
    by (cases (∑i. f i)) auto
  {
    fix i
    have f i ≠ ∞
      using f by (intro suminf-PInfty[OF - fin]) auto
    then have |f i| ≠ ∞
      using f[of i] by auto
  }
  note fin = this
  have (λi. ereal (real-of-ereal (f i))) sums (∑i. ereal (real-of-ereal (f i)))
    using f
    by (auto intro!: summable-ereal-pos simp: ereal-le-real-iff zero-ereal-def)
  also have . . . = ereal r
    using fin r by (auto simp: ereal-real)
```

finally show $\exists\, r.\ (\lambda i.\ \textit{real-of-ereal}\ (f\ i))\ \textit{sums}\ r$
by (*auto simp*: *sums-ereal*)
qed

lemma *suminf-SUP-eq*:
fixes $f :: \textit{nat} \Rightarrow \textit{nat} \Rightarrow \textit{ereal}$
assumes $\bigwedge i.\ \textit{incseq}\ (\lambda n.\ f\ n\ i)$
and $\bigwedge n\ i.\ 0 \leq f\ n\ i$
shows $(\sum i.\ \textit{SUP}\ n.\ f\ n\ i) = (\textit{SUP}\ n.\ \sum i.\ f\ n\ i)$
proof −
{
fix $n :: \textit{nat}$
have $(\sum i{<}n.\ \textit{SUP}\ k.\ f\ k\ i) = (\textit{SUP}\ k.\ \sum i{<}n.\ f\ k\ i)$
using *assms*
by (*auto intro*!: *SUP-ereal-setsum* [*symmetric*])
}
note ∗ = *this*
show *?thesis*
using *assms*
apply (*subst* (*1 2*) *suminf-ereal-eq-SUP*)
unfolding ∗
apply (*auto intro*!: *SUP-upper2*)
apply (*subst SUP-commute*)
apply *rule*
done
qed

lemma *suminf-setsum-ereal*:
fixes $f :: \text{-} \Rightarrow \text{-} \Rightarrow \textit{ereal}$
assumes *nonneg*: $\bigwedge i\ a.\ a \in A \Longrightarrow 0 \leq f\ i\ a$
shows $(\sum i.\ \sum a{\in}A.\ f\ i\ a) = (\sum a{\in}A.\ \sum i.\ f\ i\ a)$
proof (*cases finite A*)
case *True*
then show *?thesis*
using *nonneg*
by *induct* (*simp-all add*: *suminf-add-ereal setsum-nonneg*)
next
case *False*
then show *?thesis* **by** *simp*
qed

lemma *suminf-ereal-eq-0*:
fixes $f :: \textit{nat} \Rightarrow \textit{ereal}$
assumes *nneg*: $\bigwedge i.\ 0 \leq f\ i$
shows $(\sum i.\ f\ i) = 0 \longleftrightarrow (\forall\, i.\ f\ i = 0)$
proof
assume $(\sum i.\ f\ i) = 0$
{
fix i

assume $f\ i \neq 0$
with *nneg* **have** $0 < f\ i$
by (*auto simp*: *less-le*)
also have $f\ i = (\sum j.\ if\ j = i\ then\ f\ i\ else\ 0)$
by (*subst suminf-finite*[**where** $N=\{i\}$]) *auto*
also have $\ldots \leq (\sum i.\ f\ i)$
using *nneg*
by (*auto intro*!: *suminf-le-pos*)
finally have *False*
using ‹$(\sum i.\ f\ i) = 0$› **by** *auto*
}
then show $\forall i.\ f\ i = 0$
by *auto*
qed *simp*

lemma *suminf-ereal-offset-le*:
fixes $f :: nat \Rightarrow ereal$
assumes f: $\bigwedge i.\ 0 \leq f\ i$
shows $(\sum i.\ f\ (i + k)) \leq suminf\ f$
proof −
have $(\lambda n.\ \sum i<n.\ f\ (i + k)) \longrightarrow (\sum i.\ f\ (i + k))$
using *summable-sums*[*OF summable-ereal-pos*] **by** (*simp add*: *sums-def atLeast0LessThan f*)
moreover have $(\lambda n.\ \sum i<n.\ f\ i) \longrightarrow (\sum i.\ f\ i)$
using *summable-sums*[*OF summable-ereal-pos*] **by** (*simp add*: *sums-def atLeast0LessThan f*)
then have $(\lambda n.\ \sum i<n + k.\ f\ i) \longrightarrow (\sum i.\ f\ i)$
by (*rule LIMSEQ-ignore-initial-segment*)
ultimately show *?thesis*
proof (*rule LIMSEQ-le*, *safe intro*!: *exI*[*of - k*])
fix n **assume** $k \leq n$
have $(\sum i<n.\ f\ (i + k)) = (\sum i<n.\ (f \circ (\lambda i.\ i + k))\ i)$
by *simp*
also have $\ldots = (\sum i\in(\lambda i.\ i + k)\ `\ \{..<n\}.\ f\ i)$
by (*subst setsum.reindex*) *auto*
also have $\ldots \leq setsum\ f\ \{..<n + k\}$
by (*intro setsum-mono3*) (*auto simp*: *f*)
finally show $(\sum i<n.\ f\ (i + k)) \leq setsum\ f\ \{..<n + k\}$ **.**
qed
qed

lemma *sums-suminf-ereal*: $f\ sums\ x \Longrightarrow (\sum i.\ ereal\ (f\ i)) = ereal\ x$
by (*metis sums-ereal sums-unique*)

lemma *suminf-ereal'*: $summable\ f \Longrightarrow (\sum i.\ ereal\ (f\ i)) = ereal\ (\sum i.\ f\ i)$
by (*metis sums-ereal sums-unique summable-def*)

lemma *suminf-ereal-finite*: $summable\ f \Longrightarrow (\sum i.\ ereal\ (f\ i)) \neq \infty$
by (*auto simp*: *sums-ereal*[*symmetric*] *summable-def sums-unique*[*symmetric*])

lemma *suminf-ereal-finite-neg*:
  assumes *summable f*
  shows $(\sum x.\ ereal\ (f\ x)) \neq -\infty$
proof−
  from *assms* **obtain** *x* **where** *f sums x* **by** *blast*
  hence $(\lambda x.\ ereal\ (f\ x))$ *sums ereal x* **by** (*simp add*: *sums-ereal*)
  from *sums-unique*[*OF this*] **have** $(\sum x.\ ereal\ (f\ x)) = ereal\ x$ **..**
  thus $(\sum x.\ ereal\ (f\ x)) \neq -\infty$ **by** *simp-all*
qed

lemma *SUP-ereal-add-directed*:
  fixes $f\ g :: 'a \Rightarrow ereal$
  assumes *nonneg*: $\bigwedge i.\ i \in I \Longrightarrow 0 \leq f\ i$ $\bigwedge i.\ i \in I \Longrightarrow 0 \leq g\ i$
  assumes *directed*: $\bigwedge i\ j.\ i \in I \Longrightarrow j \in I \Longrightarrow \exists k{\in}I.\ f\ i + g\ j \leq f\ k + g\ k$
  shows $(SUP\ i{:}I.\ f\ i + g\ i) = (SUP\ i{:}I.\ f\ i) + (SUP\ i{:}I.\ g\ i)$
proof *cases*
  assume $I = \{\}$ **then show** *?thesis*
    by (*simp add*: *bot-ereal-def*)
next
  assume $I \neq \{\}$
  show *?thesis*
  proof (*rule antisym*)
    show $(SUP\ i{:}I.\ f\ i + g\ i) \leq (SUP\ i{:}I.\ f\ i) + (SUP\ i{:}I.\ g\ i)$
      by (*rule SUP-least*; *intro ereal-add-mono SUP-upper*)
  next
    have $bot < (SUP\ i{:}I.\ g\ i)$
      using ‹$I \neq \{\}$› *nonneg*(*2*) **by** (*auto simp*: *bot-ereal-def less-SUP-iff*)
    then have $(SUP\ i{:}I.\ f\ i) + (SUP\ i{:}I.\ g\ i) = (SUP\ i{:}I.\ f\ i + (SUP\ i{:}I.\ g\ i))$
      by (*intro SUP-ereal-add-left*[*symmetric*] ‹$I \neq \{\}$›) *auto*
    also have $\ldots = (SUP\ i{:}I.\ (SUP\ j{:}I.\ f\ i + g\ j))$
      using *nonneg*(*1*) **by** (*intro SUP-cong refl SUP-ereal-add-right*[*symmetric*] ‹I
$\neq \{\}$›) *auto*
    also have $\ldots \leq (SUP\ i{:}I.\ f\ i + g\ i)$
      using *directed* **by** (*intro SUP-least*) (*blast intro*: *SUP-upper2*)
    finally show $(SUP\ i{:}I.\ f\ i) + (SUP\ i{:}I.\ g\ i) \leq (SUP\ i{:}I.\ f\ i + g\ i)$ **.**
  qed
qed

lemma *SUP-ereal-setsum-directed*:
  fixes $f\ g :: 'a \Rightarrow 'b \Rightarrow ereal$
  assumes $I \neq \{\}$
  assumes *directed*: $\bigwedge N\ i\ j.\ N \subseteq A \Longrightarrow i \in I \Longrightarrow j \in I \Longrightarrow \exists k{\in}I.\ \forall n{\in}N.\ f\ n\ i$
$\leq f\ n\ k \wedge f\ n\ j \leq f\ n\ k$
  assumes *nonneg*: $\bigwedge n\ i.\ i \in I \Longrightarrow n \in A \Longrightarrow 0 \leq f\ n\ i$
  shows $(SUP\ i{:}I.\ \sum n{\in}A.\ f\ n\ i) = (\sum n{\in}A.\ SUP\ i{:}I.\ f\ n\ i)$
proof −
  have $N \subseteq A \Longrightarrow (SUP\ i{:}I.\ \sum n{\in}N.\ f\ n\ i) = (\sum n{\in}N.\ SUP\ i{:}I.\ f\ n\ i)$ **for** *N*
  proof (*induction N rule*: *infinite-finite-induct*)

case (*insert n N*)
moreover have ($SUP\ i{:}I.\ f\ n\ i + (\sum l{\in}N.\ f\ l\ i)) = (SUP\ i{:}I.\ f\ n\ i) + (SUP\ i{:}I.\ \sum l{\in}N.\ f\ l\ i)$
proof (*rule SUP-ereal-add-directed*)
fix *i* **assume** $i \in I$ **then show** $0 \le f\ n\ i$ $0 \le (\sum l{\in}N.\ f\ l\ i)$
using *insert* **by** (*auto intro*!: *setsum-nonneg nonneg*)
next
fix *i j* **assume** $i \in I$ $j \in I$
from *directed*[*OF* ‹*insert n N* ⊆ *A*› *this*] **guess** *k* ..
then show $\exists k{\in}I.\ f\ n\ i + (\sum l{\in}N.\ f\ l\ j) \le f\ n\ k + (\sum l{\in}N.\ f\ l\ k)$
by (*intro bexI*[*of - k*]) (*auto intro*!: *ereal-add-mono setsum-mono*)
qed
ultimately show *?case*
by *simp*
qed (*simp-all add*: *SUP-constant* ‹$I \ne \{\}$›)
from *this*[*of A*] **show** *?thesis* **by** *simp*
qed

lemma *suminf-SUP-eq-directed*:
fixes *f* :: - ⇒ *nat* ⇒ *ereal*
assumes $I \ne \{\}$
assumes *directed*: $\bigwedge N\ i\ j.\ i \in I \Longrightarrow j \in I \Longrightarrow finite\ N \Longrightarrow \exists k{\in}I.\ \forall n{\in}N.\ f\ i\ n \le f\ k\ n \wedge f\ j\ n \le f\ k\ n$
assumes *nonneg*: $\bigwedge n\ i.\ 0 \le f\ n\ i$
shows $(\sum i.\ SUP\ n{:}I.\ f\ n\ i) = (SUP\ n{:}I.\ \sum i.\ f\ n\ i)$
proof (*subst* (*1 2*) *suminf-ereal-eq-SUP*)
show $\bigwedge n\ i.\ 0 \le f\ n\ i$ $\bigwedge i.\ 0 \le (SUP\ n{:}I.\ f\ n\ i)$
using ‹$I \ne \{\}$› *nonneg* **by** (*auto intro*: *SUP-upper2*)
show $(SUP\ n.\ \sum i{<}n.\ SUP\ n{:}I.\ f\ n\ i) = (SUP\ n{:}I.\ SUP\ j.\ \sum i{<}j.\ f\ n\ i)$
apply (*subst SUP-commute*)
apply (*subst SUP-ereal-setsum-directed*)
apply (*auto intro*!: *assms simp*: *finite-subset*)
done
qed

lemma *ereal-dense3*:
fixes *x y* :: *ereal*
shows $x < y \Longrightarrow \exists r{::}rat.\ x < real\text{-}of\text{-}rat\ r \wedge real\text{-}of\text{-}rat\ r < y$
proof (*cases x y rule*: *ereal2-cases*, *simp-all*)
fix *r q* :: *real*
assume $r < q$
from *Rats-dense-in-real*[*OF this*] **show** $\exists x.\ r < real\text{-}of\text{-}rat\ x \wedge real\text{-}of\text{-}rat\ x < q$
by (*fastforce simp*: *Rats-def*)
next
fix *r* :: *real*
show $\exists x.\ r < real\text{-}of\text{-}rat\ x$ $\exists x.\ real\text{-}of\text{-}rat\ x < r$
using *gt-ex*[*of r*] *lt-ex*[*of r*] *Rats-dense-in-real*
by (*auto simp*: *Rats-def*)
qed

lemma *continuous-within-ereal*[*intro*, *simp*]: $x \in A \Longrightarrow$ *continuous* (*at x within A*) *ereal*
  using *continuous-on-eq-continuous-within*[*of A ereal*]
  by (*auto intro*: *continuous-on-ereal continuous-on-id*)

lemma *ereal-open-uminus*:
  fixes *S* :: *ereal set*
  assumes *open S*
  shows *open* (*uminus* ‘ *S*)
  using ‹*open S*›[*unfolded open-generated-order*]
proof *induct*
  have *range uminus* = (*UNIV* :: *ereal set*)
    by (*auto simp*: *image-iff ereal-uminus-eq-reorder*)
  then show *open* (*range uminus* :: *ereal set*)
    by *simp*
qed (*auto simp add*: *image-Union image-Int*)

lemma *ereal-uminus-complement*:
  fixes *S* :: *ereal set*
  shows *uminus* ‘ (− *S*) = − *uminus* ‘ *S*
  by (*auto intro*!: *bij-image-Compl-eq surjI*[*of - uminus*] *simp*: *bij-betw-def*)

lemma *ereal-closed-uminus*:
  fixes *S* :: *ereal set*
  assumes *closed S*
  shows *closed* (*uminus* ‘ *S*)
  using *assms*
  unfolding *closed-def ereal-uminus-complement*[*symmetric*]
  by (*rule ereal-open-uminus*)

lemma *ereal-open-affinity-pos*:
  fixes *S* :: *ereal set*
  assumes *open S*
    and *m*: $m \neq \infty$ $0 < m$
    and *t*: $|t| \neq \infty$
  shows *open* (($\lambda x.$ $m * x + t$) ‘ *S*)
proof −
  have *open* (($\lambda x.$ *inverse* $m * (x + -t)$) −‘ *S*)
    using *m t*
    apply (*intro open-vimage* ‹*open S*›)
   apply (*intro continuous-at-imp-continuous-on ballI tendsto-cmult-ereal continuous-at*[*THEN iffD2*]
        tendsto-ident-at tendsto-add-left-ereal)
    apply *auto*
    done
  also have ($\lambda x.$ *inverse* $m * (x + -t)$) −‘ *S* = ($\lambda x.$ $(x - t) / m$) −‘ *S*
   using *m t* **by** (*auto simp*: *divide-ereal-def mult.commute uminus-ereal.simps*[*symmetric*] *minus-ereal-def*

simp del: *uminus-ereal.simps*)
also have ($\lambda x.$ (x − t) / m) −' S = ($\lambda x.$ m ∗ x + t) ' S
using *m t*
by (*simp add*: *set-eq-iff image-iff*)
(*metis abs-ereal-less0 abs-ereal-uminus ereal-divide-eq ereal-eq-minus ereal-minus*(*7*,*8*)
ereal-minus-less-minus ereal-mult-eq-PInfty ereal-uminus-uminus
ereal-zero-mult)
finally show *?thesis* **.**
qed

lemma *ereal-open-affinity*:
fixes S :: *ereal set*
assumes *open S*
and m: $|m| \neq \infty$ $m \neq 0$
and t: $|t| \neq \infty$
shows *open* (($\lambda x.$ m ∗ x + t) ' S)
proof *cases*
assume $0 < m$
then show *?thesis*
using *ereal-open-affinity-pos*[*OF* ‹*open S*› - - *t*, *of m*] *m*
by *auto*
next
assume ¬ $0 < m$ **then**
have $0 < -m$
using ‹$m \neq 0$›
by (*cases m*) *auto*
then have m: $-m \neq \infty$ $0 < -m$
using ‹$|m| \neq \infty$›
by (*auto simp*: *ereal-uminus-eq-reorder*)
from *ereal-open-affinity-pos*[*OF ereal-open-uminus*[*OF* ‹*open S*›] *m t*] **show** *?thesis*
unfolding *image-image* **by** *simp*
qed

lemma *open-uminus-iff*:
fixes S :: *ereal set*
shows *open* (*uminus* ' S) ⟷ *open S*
using *ereal-open-uminus*[*of S*] *ereal-open-uminus*[*of uminus* ' S]
by *auto*

lemma *ereal-Liminf-uminus*:
fixes f :: $'a \Rightarrow$ *ereal*
shows *Liminf net* ($\lambda x.$ − (f x)) = − *Limsup net f*
using *ereal-Limsup-uminus*[*of* - ($\lambda x.$ − (f x))] **by** *auto*

lemma *Liminf-PInfty*:
fixes f :: $'a \Rightarrow$ *ereal*
assumes ¬ *trivial-limit net*
shows (f ⟶ ∞) *net* ⟷ *Liminf net f* = ∞
unfolding *tendsto-iff-Liminf-eq-Limsup*[*OF assms*]

using *Liminf-le-Limsup*[*OF assms*, *of f*]
by *auto*

lemma *Limsup-MInfty*:
fixes $f :: 'a \Rightarrow ereal$
assumes $\neg$ *trivial-limit net*
shows $(f \longrightarrow -\infty)$ *net* $\longleftrightarrow$ *Limsup net f* $= -\infty$
unfolding *tendsto-iff-Liminf-eq-Limsup*[*OF assms*]
using *Liminf-le-Limsup*[*OF assms*, *of f*]
by *auto*

lemma *convergent-ereal*: — RENAME
fixes $X :: nat \Rightarrow 'a :: \{complete\text{-}linorder, linorder\text{-}topology\}$
shows *convergent X* $\longleftrightarrow$ *limsup X* = *liminf X*
using *tendsto-iff-Liminf-eq-Limsup*[*of sequentially*]
by (*auto simp*: *convergent-def*)

lemma *limsup-le-liminf-real*:
fixes $X :: nat \Rightarrow real$ **and** $L :: real$
assumes *1*: *limsup X* $\leq L$ **and** *2*: $L \leq$ *liminf X*
shows $X \longrightarrow L$
proof −
from *1 2* **have** *limsup X* $\leq$ *liminf X* **by** *auto*
hence *3*: *limsup X* = *liminf X*
apply (*subst eq-iff*, *rule conjI*)
by (*rule Liminf-le-Limsup*, *auto*)
hence *4*: *convergent* (λn. *ereal* (*X n*))
by (*subst convergent-ereal*)
hence *limsup X* = *lim* (λn. *ereal*(*X n*))
by (*rule convergent-limsup-cl*)
also from *1 2 3* **have** *limsup X* = *L* **by** *auto*
finally have *lim* (λn. *ereal*(*X n*)) = *L* **..**
hence (λn. *ereal* (*X n*)) $\longrightarrow L$
apply (*elim subst*)
by (*subst convergent-LIMSEQ-iff* [*symmetric*], *rule 4*)
thus *?thesis* **by** *simp*
qed

lemma *liminf-PInfty*:
fixes $X :: nat \Rightarrow ereal$
shows $X \longrightarrow \infty \longleftrightarrow$ *liminf X* $= \infty$
by (*metis Liminf-PInfty trivial-limit-sequentially*)

lemma *limsup-MInfty*:
fixes $X :: nat \Rightarrow ereal$
shows $X \longrightarrow -\infty \longleftrightarrow$ *limsup X* $= -\infty$
by (*metis Limsup-MInfty trivial-limit-sequentially*)

lemma *ereal-lim-mono*:

fixes $X\ Y$ :: *nat* ⇒ *'a*::*linorder-topology*
assumes $\bigwedge n.\ N \leq n \Longrightarrow X\ n \leq Y\ n$
and $X \longrightarrow x$
and $Y \longrightarrow y$
shows $x \leq y$
using *assms*(*1*) **by** (*intro LIMSEQ-le*[*OF assms*(*2*,*3*)]) *auto*

lemma *incseq-le-ereal*:
fixes X :: *nat* ⇒ *'a*::*linorder-topology*
assumes *inc*: *incseq X*
and *lim*: $X \longrightarrow L$
shows $X\ N \leq L$
using *inc*
by (*intro ereal-lim-mono*[*of N*, *OF - tendsto-const lim*]) (*simp add*: *incseq-def*)

lemma *decseq-ge-ereal*:
assumes *dec*: *decseq X*
and *lim*: $X \longrightarrow$ (*L*::*'a*::*linorder-topology*)
shows $X\ N \geq L$
using *dec* **by** (*intro ereal-lim-mono*[*of N*, *OF - lim tendsto-const*]) (*simp add*: *decseq-def*)

lemma *bounded-abs*:
fixes *a* :: *real*
assumes $a \leq x$
and $x \leq b$
shows $|x| \leq max\ |a|\ |b|$
by (*metis abs-less-iff assms leI le-max-iff-disj less-eq-real-def less-le-not-le less-minus-iff minus-minus*)

lemma *ereal-Sup-lim*:
fixes *a* :: *'a*::{*complete-linorder*,*linorder-topology*}
assumes $\bigwedge n.\ b\ n \in s$
and $b \longrightarrow a$
shows $a \leq Sup\ s$
by (*metis Lim-bounded-ereal assms complete-lattice-class.Sup-upper*)

lemma *ereal-Inf-lim*:
fixes *a* :: *'a*::{*complete-linorder*,*linorder-topology*}
assumes $\bigwedge n.\ b\ n \in s$
and $b \longrightarrow a$
shows $Inf\ s \leq a$
by (*metis Lim-bounded2-ereal assms complete-lattice-class.Inf-lower*)

lemma *SUP-Lim-ereal*:
fixes X :: *nat* ⇒ *'a*::{*complete-linorder*,*linorder-topology*}
assumes *inc*: *incseq X*
and *l*: $X \longrightarrow l$
shows $(SUP\ n.\ X\ n) = l$

using *LIMSEQ-SUP*[*OF inc*] *tendsto-unique*[*OF trivial-limit-sequentially l*]
by *simp*

lemma *INF-Lim-ereal*:
fixes X :: *nat* ⇒ *'a*::{*complete-linorder*,*linorder-topology*}
assumes *dec*: *decseq X*
and *l*: $X \longrightarrow l$
shows (*INF n. X n*) = *l*
using *LIMSEQ-INF*[*OF dec*] *tendsto-unique*[*OF trivial-limit-sequentially l*]
by *simp*

lemma *SUP-eq-LIMSEQ*:
assumes *mono f*
shows (*SUP n. ereal* (*f n*)) = *ereal x* ⟷ *f* ⟶ *x*
proof
have *inc*: *incseq* (*λi. ereal* (*f i*))
using ‹*mono f*› **unfolding** *mono-def incseq-def* **by** *auto*
{
assume *f* ⟶ *x*
then have (*λi. ereal* (*f i*)) ⟶ *ereal x*
by *auto*
from *SUP-Lim-ereal*[*OF inc this*] **show** (*SUP n. ereal* (*f n*)) = *ereal x* **.**
next
assume (*SUP n. ereal* (*f n*)) = *ereal x*
with *LIMSEQ-SUP*[*OF inc*] **show** *f* ⟶ *x* **by** *auto*
}
qed

lemma *liminf-ereal-cminus*:
fixes *f* :: *nat* ⇒ *ereal*
assumes $c \neq -\infty$
shows *liminf* (*λx. c − f x*) = *c − limsup f*
proof (*cases c*)
case *PInf*
then show *?thesis*
by (*simp add*: *Liminf-const*)
next
case (*real r*)
then show *?thesis*
unfolding *liminf-SUP-INF limsup-INF-SUP*
apply (*subst INF-ereal-minus-right*)
apply *auto*
apply (*subst SUP-ereal-minus-right*)
apply *auto*
done
qed (*insert* ‹$c \neq -\infty$›, *simp*)

33.4.3 Continuity

lemma *continuous-at-of-ereal*:
$|x0 :: ereal| \neq \infty \Longrightarrow$ *continuous* (*at x0*) *real-of-ereal*
unfolding *continuous-at*
by (*rule lim-real-of-ereal*) (*simp add*: *ereal-real*)

lemma *nhds-ereal*: *nhds* (*ereal r*) = *filtermap ereal* (*nhds r*)
by (*simp add*: *filtermap-nhds-open-map open-ereal continuous-at-of-ereal*)

lemma *at-ereal*: *at* (*ereal r*) = *filtermap ereal* (*at r*)
by (*simp add*: *filter-eq-iff eventually-at-filter nhds-ereal eventually-filtermap*)

lemma *at-left-ereal*: *at-left* (*ereal r*) = *filtermap ereal* (*at-left r*)
by (*simp add*: *filter-eq-iff eventually-at-filter nhds-ereal eventually-filtermap*)

lemma *at-right-ereal*: *at-right* (*ereal r*) = *filtermap ereal* (*at-right r*)
by (*simp add*: *filter-eq-iff eventually-at-filter nhds-ereal eventually-filtermap*)

lemma
shows *at-left-PInf*: *at-left* ∞ = *filtermap ereal at-top*
and *at-right-MInf*: *at-right* $(-\infty)$ = *filtermap ereal at-bot*
unfolding *filter-eq-iff eventually-filtermap eventually-at-top-dense eventually-at-bot-dense eventually-at-left*[*OF ereal-less*(*5*)] *eventually-at-right*[*OF ereal-less*(*6*)]
by (*auto simp add*: *ereal-all-split ereal-ex-split*)

lemma *ereal-tendsto-simps1*:
$((f \circ real\text{-}of\text{-}ereal) \longrightarrow y)$ (*at-left* (*ereal x*)) $\longleftrightarrow$ $(f \longrightarrow y)$ (*at-left x*)
$((f \circ real\text{-}of\text{-}ereal) \longrightarrow y)$ (*at-right* (*ereal x*)) $\longleftrightarrow$ $(f \longrightarrow y)$ (*at-right x*)
$((f \circ real\text{-}of\text{-}ereal) \longrightarrow y)$ (*at-left* (∞::*ereal*)) $\longleftrightarrow$ $(f \longrightarrow y)$ *at-top*
$((f \circ real\text{-}of\text{-}ereal) \longrightarrow y)$ (*at-right* ($-\infty$::*ereal*)) $\longleftrightarrow$ $(f \longrightarrow y)$ *at-bot*
unfolding *tendsto-compose-filtermap at-left-ereal at-right-ereal at-left-PInf at-right-MInf*
by (*auto simp*: *filtermap-filtermap filtermap-ident*)

lemma *ereal-tendsto-simps2*:
$((ereal \circ f) \longrightarrow ereal\ a)\ F \longleftrightarrow (f \longrightarrow a)\ F$
$((ereal \circ f) \longrightarrow \infty)\ F \longleftrightarrow$ (*LIM x F. f x :> at-top*)
$((ereal \circ f) \longrightarrow -\infty)\ F \longleftrightarrow$ (*LIM x F. f x :> at-bot*)
unfolding *tendsto-PInfty filterlim-at-top-dense tendsto-MInfty filterlim-at-bot-dense*
using *lim-ereal* **by** (*simp-all add*: *comp-def*)

lemma *inverse-infty-ereal-tendsto-0*: *inverse* $-\infty\rightarrow$ (*0*::*ereal*)
proof −
have ∗∗: ((λx. *ereal* (*inverse x*)) $\longrightarrow$ *ereal 0*) *at-infinity*
by (*intro tendsto-intros tendsto-inverse-0*)

show *?thesis*
by (*simp add*: *at-infty-ereal-eq-at-top tendsto-compose-filtermap*[*symmetric*]
comp-def)
(*auto simp*: *eventually-at-top-linorder exI*[*of - 1*] *zero-ereal-def at-top-le-at-infinity*

intro!: *filterlim-mono-eventually*[*OF* ∗∗])
qed

lemma *inverse-ereal-tendsto-pos*:
fixes x :: *ereal* **assumes** $0 < x$
shows *inverse* $-x\rightarrow$ *inverse* x
proof (*cases* x)
case (*real* r)
with ‹$0 < x$› **have** ∗∗: ($\lambda x.$ *ereal* (*inverse* x)) $-r\rightarrow$ *ereal* (*inverse* r)
by (*auto intro*!: *tendsto-inverse*)
from *real* ‹$0 < x$› **show** *?thesis*
by (*auto simp*: *at-ereal tendsto-compose-filtermap*[*symmetric*] *eventually-at-filter*
intro!: *Lim-transform-eventually*[*OF* - ∗∗] *t1-space-nhds*)
qed (*insert* ‹$0 < x$›, *auto intro*!: *inverse-infty-ereal-tendsto-0*)

lemma *inverse-ereal-tendsto-at-right-0*: (*inverse* $\longrightarrow \infty$) (*at-right* (*0*::*ereal*))
unfolding *tendsto-compose-filtermap*[*symmetric*] *at-right-ereal zero-ereal-def*
by (*subst filterlim-cong*[*OF refl refl*, **where** g=$\lambda x.$ *ereal* (*inverse* x)])
(*auto simp*: *eventually-at-filter tendsto-PInfty-eq-at-top filterlim-inverse-at-top-right*)

lemmas *ereal-tendsto-simps* = *ereal-tendsto-simps1 ereal-tendsto-simps2*

lemma *continuous-at-iff-ereal*:
fixes f :: ′*a*::*t2-space* ⇒ *real*
shows *continuous* (*at x0 within s*) $f \longleftrightarrow$ *continuous* (*at x0 within s*) (*ereal* ∘ f)
unfolding *continuous-within comp-def lim-ereal* **..**

lemma *continuous-on-iff-ereal*:
fixes f :: ′*a*::*t2-space* => *real*
assumes *open A*
shows *continuous-on A f* $\longleftrightarrow$ *continuous-on A* (*ereal* ∘ f)
unfolding *continuous-on-def comp-def lim-ereal* **..**

lemma *continuous-on-real*: *continuous-on* (*UNIV* − {∞, −∞::*ereal*}) *real-of-ereal*
using *continuous-at-of-ereal continuous-on-eq-continuous-at open-image-ereal*
by *auto*

lemma *continuous-on-iff-real*:
fixes f :: ′*a*::*t2-space* ⇒ *ereal*
assumes ∗: $\bigwedge x.\ x \in A \Longrightarrow |f\,x| \neq \infty$
shows *continuous-on A f* $\longleftrightarrow$ *continuous-on A* (*real-of-ereal* ∘ f)
proof −
have f ‘ $A \subseteq$ *UNIV* − {∞, −∞}
using *assms* **by** *force*
then have ∗: *continuous-on* (f ‘ A) *real-of-ereal*
using *continuous-on-real* **by** (*simp add*: *continuous-on-subset*)
have ∗∗: *continuous-on* ((*real-of-ereal* ∘ f) ‘ A) *ereal*
by (*intro continuous-on-ereal continuous-on-id*)
{

```
    assume continuous-on A f
    then have continuous-on A (real-of-ereal ∘ f)
      apply (subst continuous-on-compose)
      using ∗
      apply auto
      done
  }
  moreover
  {
    assume continuous-on A (real-of-ereal ∘ f)
    then have continuous-on A (ereal ∘ (real-of-ereal ∘ f))
      apply (subst continuous-on-compose)
      using ∗∗
      apply auto
      done
    then have continuous-on A f
      apply (subst continuous-on-cong[of - A - ereal ∘ (real-of-ereal ∘ f)])
      using assms ereal-real
      apply auto
      done
  }
  ultimately show ?thesis
    by auto
qed

lemma continuous-uminus-ereal [continuous-intros]: continuous-on (A :: ereal set)
uminus
  unfolding continuous-on-def
  by (intro ballI tendsto-uminus-ereal[of λx. x::ereal]) simp

lemma ereal-uminus-atMost [simp]: uminus ‘ {..(a::ereal)} = {−a..}
proof (intro equalityI subsetI)
  fix x :: ereal assume x ∈ {−a..}
  hence −(−x) ∈ uminus ‘ {..a} by (intro imageI) (simp add: ereal-uminus-le-reorder)
  thus x ∈ uminus ‘ {..a} by simp
qed auto

lemma continuous-on-inverse-ereal [continuous-intros]:
  continuous-on {0::ereal ..} inverse
  unfolding continuous-on-def
proof clarsimp
  fix x :: ereal assume 0 ≤ x
  moreover have at 0 within {0 ..} = at-right (0::ereal)
    by (auto simp: filter-eq-iff eventually-at-filter le-less)
  moreover have at x within {0 ..} = at x if 0 < x
    using that by (intro at-within-nhd[of - {0<..}]) auto
  ultimately show (inverse ⟶ inverse x) (at x within {0..})
    by (auto simp: le-less inverse-ereal-tendsto-at-right-0 inverse-ereal-tendsto-pos)
qed
```

lemma *continuous-inverse-ereal-nonpos*: *continuous-on* ({..<0} :: *ereal set*) *inverse*
proof (*subst continuous-on-cong*[*OF refl*])
  have *continuous-on* {(0::*ereal*)<..} *inverse*
    by (*rule continuous-on-subset*[*OF continuous-on-inverse-ereal*]) *auto*
  thus *continuous-on* {..<(0::*ereal*)} (*uminus* ∘ *inverse* ∘ *uminus*)
    by (*intro continuous-intros*) *simp-all*
qed *simp*

lemma *tendsto-inverse-ereal*:
  assumes (f ⟶ (c :: *ereal*)) F
  assumes *eventually* ($\lambda x.\ f\ x \geq 0$) F
  shows (($\lambda x.$ *inverse* ($f\ x$)) ⟶ *inverse* c) F
  by (*cases* F = *bot*)
    (*auto intro*!: *tendsto-le-const*[*of* F] *assms*
        continuous-on-tendsto-compose[*OF continuous-on-inverse-ereal*])

33.4.4 liminf and limsup

lemma *Limsup-ereal-mult-right*:
  assumes $F \neq$ *bot* (c::*real*) ≥ 0
  shows *Limsup* F ($\lambda n.\ f\ n$ ∗ *ereal* c) = *Limsup* $F\ f$ ∗ *ereal* c
proof (*rule Limsup-compose-continuous-mono*)
  from *assms* **show** *continuous-on UNIV* ($\lambda a.\ a$ ∗ *ereal* c)
    using *tendsto-cmult-ereal*[*of ereal* c $\lambda x.\ x$]
    by (*force simp*: *continuous-on-def mult-ac*)
qed (*insert assms*, *auto simp*: *mono-def ereal-mult-right-mono*)

lemma *Liminf-ereal-mult-right*:
  assumes $F \neq$ *bot* (c::*real*) ≥ 0
  shows *Liminf* F ($\lambda n.\ f\ n$ ∗ *ereal* c) = *Liminf* $F\ f$ ∗ *ereal* c
proof (*rule Liminf-compose-continuous-mono*)
  from *assms* **show** *continuous-on UNIV* ($\lambda a.\ a$ ∗ *ereal* c)
    using *tendsto-cmult-ereal*[*of ereal* c $\lambda x.\ x$]
    by (*force simp*: *continuous-on-def mult-ac*)
qed (*insert assms*, *auto simp*: *mono-def ereal-mult-right-mono*)

lemma *Limsup-ereal-mult-left*:
  assumes $F \neq$ *bot* (c::*real*) ≥ 0
  shows *Limsup* F ($\lambda n.$ *ereal* c ∗ $f\ n$) = *ereal* c ∗ *Limsup* $F\ f$
  using *Limsup-ereal-mult-right*[*OF assms*] **by** (*subst* (*1 2*) *mult.commute*)

lemma *limsup-ereal-mult-right*:
  (c::*real*) ≥ 0 ⟹ *limsup* ($\lambda n.\ f\ n$ ∗ *ereal* c) = *limsup* f ∗ *ereal* c
  by (*rule Limsup-ereal-mult-right*) *simp-all*

lemma *limsup-ereal-mult-left*:
  (c::*real*) ≥ 0 ⟹ *limsup* ($\lambda n.$ *ereal* c ∗ $f\ n$) = *ereal* c ∗ *limsup* f
  by (*subst* (*1 2*) *mult.commute*, *rule limsup-ereal-mult-right*) *simp-all*

lemma *Limsup-add-ereal-right*:
  $F \neq bot \Longrightarrow abs\ c \neq \infty \Longrightarrow Limsup\ F\ (\lambda n.\ g\ n + (c :: ereal)) = Limsup\ F\ g + c$
  by (*rule Limsup-compose-continuous-mono*) (*auto simp*: *mono-def ereal-add-mono continuous-on-def*)

lemma *Limsup-add-ereal-left*:
  $F \neq bot \Longrightarrow abs\ c \neq \infty \Longrightarrow Limsup\ F\ (\lambda n.\ (c :: ereal) + g\ n) = c + Limsup\ F\ g$
  by (*subst* (*1 2*) *add.commute*) (*rule Limsup-add-ereal-right*)

lemma *Liminf-add-ereal-right*:
  $F \neq bot \Longrightarrow abs\ c \neq \infty \Longrightarrow Liminf\ F\ (\lambda n.\ g\ n + (c :: ereal)) = Liminf\ F\ g + c$
  by (*rule Liminf-compose-continuous-mono*) (*auto simp*: *mono-def ereal-add-mono continuous-on-def*)

lemma *Liminf-add-ereal-left*:
  $F \neq bot \Longrightarrow abs\ c \neq \infty \Longrightarrow Liminf\ F\ (\lambda n.\ (c :: ereal) + g\ n) = c + Liminf\ F\ g$
  by (*subst* (*1 2*) *add.commute*) (*rule Liminf-add-ereal-right*)

lemma
  assumes $F \neq bot$
  assumes *nonneg*: *eventually* $(\lambda x.\ f\ x \geq (0::ereal))\ F$
  shows *Liminf-inverse-ereal*: $Liminf\ F\ (\lambda x.\ inverse\ (f\ x)) = inverse\ (Limsup\ F\ f)$
  and *Limsup-inverse-ereal*: $Limsup\ F\ (\lambda x.\ inverse\ (f\ x)) = inverse\ (Liminf\ F\ f)$
proof −
  def $inv \equiv \lambda x.\ if\ x \leq 0\ then\ \infty\ else\ inverse\ x :: ereal$
  have *continuous-on* $(\{..0\} \cup \{0..\})$ *inv* **unfolding** *inv-def*
    by (*intro continuous-on-If*) (*auto intro*!: *continuous-intros*)
  also have $\{..0\} \cup \{0..\} = (UNIV :: ereal\ set)$ **by** *auto*
  finally have *cont*: *continuous-on UNIV inv* **.**
  have *antimono*: *antimono inv* **unfolding** *inv-def antimono-def*
    by (*auto intro*!: *ereal-inverse-antimono*)

  have $Liminf\ F\ (\lambda x.\ inverse\ (f\ x)) = Liminf\ F\ (\lambda x.\ inv\ (f\ x))$ **using** *nonneg*
    by (*auto intro*!: *Liminf-eq elim*!: *eventually-mono simp*: *inv-def*)
  also have ... $= inv\ (Limsup\ F\ f)$
    by (*simp add*: *assms*(*1*) *Liminf-compose-continuous-antimono*[*OF cont antimono*])
  also from *assms* **have** $Limsup\ F\ f \geq 0$ **by** (*intro le-Limsup*) *simp-all*
  hence $inv\ (Limsup\ F\ f) = inverse\ (Limsup\ F\ f)$ **by** (*simp add*: *inv-def*)
  finally show $Liminf\ F\ (\lambda x.\ inverse\ (f\ x)) = inverse\ (Limsup\ F\ f)$ **.**

  have $Limsup\ F\ (\lambda x.\ inverse\ (f\ x)) = Limsup\ F\ (\lambda x.\ inv\ (f\ x))$ **using** *nonneg*
    by (*auto intro*!: *Limsup-eq elim*!: *eventually-mono simp*: *inv-def*)
  also have ... $= inv\ (Liminf\ F\ f)$

by (*simp add*: *assms*(*1*) *Limsup-compose-continuous-antimono*[*OF cont anti-mono*])
also from *assms* **have** *Liminf F f* $\geq$ *0* **by** (*intro Liminf-bounded*) *simp-all*
hence *inv* (*Liminf F f*) = *inverse* (*Liminf F f*) **by** (*simp add*: *inv-def*)
finally show *Limsup F* (λx. *inverse* (*f x*)) = *inverse* (*Liminf F f*) **.**
qed

33.4.5 Tests for code generator

value $- \infty$ :: *ereal*
value $|-\infty|$ :: *ereal*
value *4* + *5* / *4* − *ereal 2* :: *ereal*
value *ereal 3* < ∞
value *real-of-ereal* (∞::*ereal*) = *0*

end

34 Indicator Function

theory *Indicator-Function*
imports *Complex-Main*
begin

definition *indicator S x* = (*if x* ∈ *S then 1 else 0*)

lemma *indicator-simps*[*simp*]:
x ∈ *S* ⟹ *indicator S x* = *1*
x ∉ *S* ⟹ *indicator S x* = *0*
unfolding *indicator-def* **by** *auto*

lemma *indicator-pos-le*[*intro*, *simp*]: (*0*::′*a*::*linordered-semidom*) ≤ *indicator S x*
and *indicator-le-1*[*intro*, *simp*]: *indicator S x* ≤ (*1*::′*a*::*linordered-semidom*)
unfolding *indicator-def* **by** *auto*

lemma *indicator-abs-le-1*: |*indicator S x*| ≤ (*1*::′*a*::*linordered-idom*)
unfolding *indicator-def* **by** *auto*

lemma *indicator-eq-0-iff*: *indicator A x* = (*0*::-::*zero-neq-one*) ⟷ *x* ∉ *A*
by (*auto simp*: *indicator-def*)

lemma *indicator-eq-1-iff*: *indicator A x* = (*1*::-::*zero-neq-one*) ⟷ *x* ∈ *A*
by (*auto simp*: *indicator-def*)

lemma *split-indicator*: *P* (*indicator S x*) ⟷ ((*x* ∈ *S* ⟶ *P 1*) ∧ (*x* ∉ *S* ⟶ *P 0*))
unfolding *indicator-def* **by** *auto*

lemma *split-indicator-asm*: *P* (*indicator S x*) ⟷ (¬ (*x* ∈ *S* ∧ ¬ *P 1* ∨ *x* ∉ *S* ∧ ¬ *P 0*))

unfolding *indicator-def* **by** *auto*

lemma *indicator-inter-arith*: *indicator* $(A \cap B)$ x = *indicator* A x $*$ (*indicator* B x::$'a$::*semiring-1*)
unfolding *indicator-def* **by** (*auto simp*: *min-def max-def*)

lemma *indicator-union-arith*: *indicator* $(A \cup B)$ x = *indicator* A x + *indicator* B x − *indicator* A x $*$ (*indicator* B x::$'a$::*ring-1*)
unfolding *indicator-def* **by** (*auto simp*: *min-def max-def*)

lemma *indicator-inter-min*: *indicator* $(A \cap B)$ x = *min* (*indicator* A x) (*indicator* B x::$'a$::*linordered-semidom*)
and *indicator-union-max*: *indicator* $(A \cup B)$ x = *max* (*indicator* A x) (*indicator* B x::$'a$::*linordered-semidom*)
unfolding *indicator-def* **by** (*auto simp*: *min-def max-def*)

lemma *indicator-disj-union*: $A \cap B = \{\} \Longrightarrow$ *indicator* $(A \cup B)$ x = (*indicator* A x + *indicator* B x::$'a$::*linordered-semidom*)
by (*auto split*: *split-indicator*)

lemma *indicator-compl*: *indicator* $(- A)$ x = *1* − (*indicator* A x::$'a$::*ring-1*)
and *indicator-diff*: *indicator* $(A - B)$ x = *indicator* A x $*$ (*1* − *indicator* B x::$'a$::*ring-1*)
unfolding *indicator-def* **by** (*auto simp*: *min-def max-def*)

lemma *indicator-times*: *indicator* $(A \times B)$ x = *indicator* A (*fst* x) $*$ (*indicator* B (*snd* x)::$'a$::*semiring-1*)
unfolding *indicator-def* **by** (*cases* x) *auto*

lemma *indicator-sum*: *indicator* (A <+> B) x = (*case* x *of* *Inl* $x \Rightarrow$ *indicator* A x | *Inr* $x \Rightarrow$ *indicator* B x)
unfolding *indicator-def* **by** (*cases* x) *auto*

lemma *indicator-image*: *inj* $f \Longrightarrow$ *indicator* (f ‘ X) (f x) = (*indicator* X x::-::*zero-neq-one*)
by (*auto simp*: *indicator-def inj-on-def*)

lemma *indicator-vimage*: *indicator* (f −‘ A) x = *indicator* A (f x)
by(*auto split*: *split-indicator*)

lemma
fixes f :: $'a \Rightarrow 'b$::*semiring-1* **assumes** *finite* A
shows *setsum-mult-indicator*[*simp*]: $(\sum x \in A.\ f\ x\ *$ *indicator* $B\ x) = (\sum x \in A \cap B.\ f\ x)$
and *setsum-indicator-mult*[*simp*]: $(\sum x \in A.$ *indicator* $B\ x * f\ x) = (\sum x \in A \cap B.\ f\ x)$
unfolding *indicator-def*
using *assms* **by** (*auto intro*!: *setsum.mono-neutral-cong-right split*: *if-split-asm*)

lemma *setsum-indicator-eq-card*:

assumes *finite A*
shows $(\sum x \in A.\ indicator\ B\ x) = card\ (A\ Int\ B)$
using *setsum-mult-indicator*[*OF assms*, *of %x. 1::nat*]
unfolding *card-eq-setsum* **by** *simp*

lemma *setsum-indicator-scaleR*[*simp*]:
finite A $\Longrightarrow$
$(\sum x \in A.\ indicator\ (B\ x)\ (g\ x) *_R f\ x) = (\sum x \in \{x{\in}A.\ g\ x \in B\ x\}.\ f\ x{::}'a{::}real\text{-}vector)$
using *assms* **by** (*auto intro*!: *setsum.mono-neutral-cong-right split*: *if-split-asm simp*: *indicator-def*)

lemma *LIMSEQ-indicator-incseq*:
assumes *incseq A*
shows $(\lambda i.\ indicator\ (A\ i)\ x :: 'a :: \{topological\text{-}space, one, zero\}) \longlonglongrightarrow indicator\ (\bigcup i.\ A\ i)\ x$
proof *cases*
assume $\exists i.\ x \in A\ i$
then obtain *i* **where** $x \in A\ i$
by *auto*
then have
$\bigwedge n.\ (indicator\ (A\ (n + i))\ x :: 'a) = 1$
$(indicator\ (\bigcup i.\ A\ i)\ x :: 'a) = 1$
using *incseqD*[*OF* ‹*incseq A*›, *of i n + i* **for** *n*] ‹$x \in A\ i$› **by** (*auto simp*: *indicator-def*)
then show *?thesis*
by (*rule-tac LIMSEQ-offset*[*of - i*]) *simp*
qed (*auto simp*: *indicator-def*)

lemma *LIMSEQ-indicator-UN*:
$(\lambda k.\ indicator\ (\bigcup i{<}k.\ A\ i)\ x :: 'a :: \{topological\text{-}space, one, zero\}) \longlonglongrightarrow indicator\ (\bigcup i.\ A\ i)\ x$
proof –
have $(\lambda k.\ indicator\ (\bigcup i{<}k.\ A\ i)\ x{::}'a) \longlonglongrightarrow indicator\ (\bigcup k.\ \bigcup i{<}k.\ A\ i)\ x$
by (*intro LIMSEQ-indicator-incseq*) (*auto simp*: *incseq-def intro*: *less-le-trans*)
also have $(\bigcup k.\ \bigcup i{<}k.\ A\ i) = (\bigcup i.\ A\ i)$
by *auto*
finally show *?thesis* **.**
qed

lemma *LIMSEQ-indicator-decseq*:
assumes *decseq A*
shows $(\lambda i.\ indicator\ (A\ i)\ x :: 'a :: \{topological\text{-}space, one, zero\}) \longlonglongrightarrow indicator\ (\bigcap i.\ A\ i)\ x$
proof *cases*
assume $\exists i.\ x \notin A\ i$
then obtain *i* **where** $x \notin A\ i$
by *auto*
then have

$\bigwedge n.$ (*indicator* (A (n + i)) x :: $'a$) = 0
(*indicator* ($\bigcap i.$ A i) x :: $'a$) = 0
using *decseqD*[*OF* ‹*decseq A*›, *of i n* + *i* **for** *n*] ‹$x \notin A\ i$› **by** (*auto simp*: *indicator-def*)
then show *?thesis*
by (*rule-tac LIMSEQ-offset*[*of* - *i*]) *simp*
qed (*auto simp*: *indicator-def*)

lemma *LIMSEQ-indicator-INT*:
($\lambda k.$ *indicator* ($\bigcap i<k.$ A i) x :: $'a$ :: {*topological-space*, *one*, *zero*}) $\longrightarrow$ *indicator* ($\bigcap i.$ A i) x
proof −
have ($\lambda k.$ *indicator* ($\bigcap i<k.$ A i) x::$'a$) $\longrightarrow$ *indicator* ($\bigcap k.$ $\bigcap i<k.$ A i) x
by (*intro LIMSEQ-indicator-decseq*) (*auto simp*: *decseq-def intro*: *less-le-trans*)
also have ($\bigcap k.$ $\bigcap i<k.$ A i) = ($\bigcap i.$ A i)
by *auto*
finally show *?thesis* **.**
qed

lemma *indicator-add*:
$A \cap B = \{\} \Longrightarrow$ (*indicator A x*::-::*monoid-add*) + *indicator B x* = *indicator* (A $\cup$ B) x
unfolding *indicator-def* **by** *auto*

lemma *of-real-indicator*: *of-real* (*indicator A x*) = *indicator A x*
by (*simp split*: *split-indicator*)

lemma *real-of-nat-indicator*: *real* (*indicator A x* :: *nat*) = *indicator A x*
by (*simp split*: *split-indicator*)

lemma *abs-indicator*: |*indicator A x* :: $'a$::*linordered-idom*| = *indicator A x*
by (*simp split*: *split-indicator*)

lemma *mult-indicator-subset*:
$A \subseteq B \Longrightarrow$ *indicator A x* $*$ *indicator B x* = (*indicator A x* :: $'a$::{*comm-semiring-1*})
by (*auto split*: *split-indicator simp*: *fun-eq-iff*)

lemma *indicator-sums*:
assumes $\bigwedge i\ j.\ i \neq j \Longrightarrow A\ i \cap A\ j = \{\}$
shows ($\lambda i.$ *indicator* (A i) x::*real*) *sums indicator* ($\bigcup i.$ A i) x
proof *cases*
assume $\exists i.\ x \in A\ i$
then obtain i **where** i: $x \in A\ i$ **..**
with *assms* **have** ($\lambda i.$ *indicator* (A i) x::*real*) *sums* ($\sum i{\in}\{i\}.$ *indicator* (A i) x)
by (*intro sums-finite*) (*auto split*: *split-indicator*)
also have ($\sum i{\in}\{i\}.$ *indicator* (A i) x) = *indicator* ($\bigcup i.$ A i) x
using i **by** (*auto split*: *split-indicator*)
finally show *?thesis* **.**

qed *simp*

end

34.1 The type of non-negative extended real numbers

theory *Extended-Nonnegative-Real*
imports *Extended-Real Indicator-Function*
begin

lemma *ereal-ineq-diff-add*:
assumes $b \neq (-\infty::ereal)$ $a \geq b$
shows $a = b + (a-b)$
by (*metis add.commute assms(1) assms(2) ereal-eq-minus-iff ereal-minus-le-iff ereal-plus-eq-PInfty*)

lemma *Limsup-const-add*:
fixes *c* :: *'a*::{*complete-linorder*, *linorder-topology*, *topological-monoid-add*, *ordered-ab-semigroup-add*}
shows $F \neq bot \Longrightarrow Limsup\ F\ (\lambda x.\ c + f\ x) = c + Limsup\ F\ f$
by (*rule Limsup-compose-continuous-mono*)
(*auto intro*!: *monoI add-mono continuous-on-add continuous-on-id continuous-on-const*)

lemma *Liminf-const-add*:
fixes *c* :: *'a*::{*complete-linorder*, *linorder-topology*, *topological-monoid-add*, *ordered-ab-semigroup-add*}
shows $F \neq bot \Longrightarrow Liminf\ F\ (\lambda x.\ c + f\ x) = c + Liminf\ F\ f$
by (*rule Liminf-compose-continuous-mono*)
(*auto intro*!: *monoI add-mono continuous-on-add continuous-on-id continuous-on-const*)

lemma *Liminf-add-const*:
fixes *c* :: *'a*::{*complete-linorder*, *linorder-topology*, *topological-monoid-add*, *ordered-ab-semigroup-add*}
shows $F \neq bot \Longrightarrow Liminf\ F\ (\lambda x.\ f\ x + c) = Liminf\ F\ f + c$
by (*rule Liminf-compose-continuous-mono*)
(*auto intro*!: *monoI add-mono continuous-on-add continuous-on-id continuous-on-const*)

lemma *sums-offset*:
fixes *f g* :: *nat* ⇒ *'a* :: {*t2-space*, *topological-comm-monoid-add*}
assumes $(\lambda n.\ f\ (n + i))$ *sums* *l* **shows** *f sums* $(l + (\sum j<i.\ f\ j))$
proof −
have $(\lambda k.\ (\sum n<k.\ f\ (n + i)) + (\sum j<i.\ f\ j)) \longrightarrow l + (\sum j<i.\ f\ j)$
using *assms* **by** (*auto intro*!: *tendsto-add simp*: *sums-def*)
moreover
{ fix *k* :: *nat*
have $(\sum j<k + i.\ f\ j) = (\sum j=i..<k + i.\ f\ j) + (\sum j=0..<i.\ f\ j)$
by (*subst setsum.union-disjoint*[*symmetric*]) (*auto intro*!: *setsum.cong*)
also have $(\sum j=i..<k + i.\ f\ j) = (\sum j\in(\lambda n.\ n + i)`\{0..<k\}.\ f\ j)$
unfolding *image-add-atLeastLessThan* **by** *simp*
finally have $(\sum j<k + i.\ f\ j) = (\sum n<k.\ f\ (n + i)) + (\sum j<i.\ f\ j)$
by (*auto simp*: *inj-on-def atLeast0LessThan setsum.reindex*) **}**
ultimately have $(\lambda k.\ (\sum n<k + i.\ f\ n)) \longrightarrow l + (\sum j<i.\ f\ j)$

by *simp*
then show *?thesis*
unfolding *sums-def* **by** (*rule LIMSEQ-offset*)
qed

lemma *suminf-offset*:
fixes *f g* :: *nat* ⇒ *'a* :: {*t2-space*, *topological-comm-monoid-add*}
shows *summable* (λ*j*. *f* (*j* + *i*)) ⟹ *suminf f* = (∑*j*. *f* (*j* + *i*)) + (∑*j*<*i*. *f j*)
by (*intro sums-unique*[*symmetric*] *sums-offset summable-sums*)

lemma *eventually-at-left-1*: (⋀*z*::*real*. *0* < *z* ⟹ *z* < *1* ⟹ *P z*) ⟹ *eventually P* (*at-left 1*)
by (*subst eventually-at-left*[*of 0*]) (*auto intro*: *exI*[*of - 0*])

lemma *mult-eq-1*:
fixes *a b* :: *'a* :: {*ordered-semiring*, *comm-monoid-mult*}
shows *0* ≤ *a* ⟹ *a* ≤ *1* ⟹ *b* ≤ *1* ⟹ *a* ∗ *b* = *1* ⟷ (*a* = *1* ∧ *b* = *1*)
by (*metis mult.left-neutral eq-iff mult.commute mult-right-mono*)

lemma *ereal-add-diff-cancel*:
fixes *a b* :: *ereal*
shows |*b*| ≠ ∞ ⟹ (*a* + *b*) − *b* = *a*
by (*cases a b rule*: *ereal2-cases*) *auto*

lemma *add-top*:
fixes *x* :: *'a*::{*order-top*, *ordered-comm-monoid-add*}
shows *0* ≤ *x* ⟹ *x* + *top* = *top*
by (*intro top-le add-increasing order-refl*)

lemma *top-add*:
fixes *x* :: *'a*::{*order-top*, *ordered-comm-monoid-add*}
shows *0* ≤ *x* ⟹ *top* + *x* = *top*
by (*intro top-le add-increasing2 order-refl*)

lemma *le-lfp*: *mono f* ⟹ *x* ≤ *lfp f* ⟹ *f x* ≤ *lfp f*
by (*subst lfp-unfold*) (*auto dest*: *monoD*)

lemma *lfp-transfer*:
assumes α: *sup-continuous* α **and** *f*: *sup-continuous f* **and** *mg*: *mono g*
assumes *bot*: α *bot* ≤ *lfp g* **and** *eq*: ⋀*x*. *x* ≤ *lfp f* ⟹ α (*f x*) = *g* (α *x*)
shows α (*lfp f*) = *lfp g*
proof (*rule antisym*)
note *mf* = *sup-continuous-mono*[*OF f*]
have *f-le-lfp*: (*f* ^^ *i*) *bot* ≤ *lfp f* **for** *i*
by (*induction i*) (*auto intro*: *le-lfp mf*)

have α ((*f* ^^ *i*) *bot*) ≤ *lfp g* **for** *i*
by (*induction i*) (*auto simp*: *bot eq f-le-lfp intro*!: *le-lfp mg*)
then show α (*lfp f*) ≤ *lfp g*

unfolding *sup-continuous-lfp*[*OF f*]
by (*subst* α[*THEN sup-continuousD*])
(*auto intro*!: *mono-funpow sup-continuous-mono*[*OF f*] *SUP-least*)

show *lfp g* $\leq$ α (*lfp f*)
by (*rule lfp-lowerbound*) (*simp add*: *eq*[*symmetric*] *lfp-unfold*[*OF mf*, *symmetric*])
qed

lemma *sup-continuous-applyD*: *sup-continuous f* $\Longrightarrow$ *sup-continuous* (λx. *f x h*)
using *sup-continuous-apply*[*THEN sup-continuous-compose*] **.**

lemma *sup-continuous-SUP*[*order-continuous-intros*]:
fixes *M* :: - $\Rightarrow$ - $\Rightarrow$ *'a*::*complete-lattice*
assumes *M*: $\bigwedge i$. $i \in I \Longrightarrow$ *sup-continuous* (*M i*)
shows *sup-continuous* (*SUP i*:*I*. *M i*)
unfolding *sup-continuous-def* **by** (*auto simp add*: *sup-continuousD*[*OF M*] *intro*: *SUP-commute*)

lemma *sup-continuous-apply-SUP*[*order-continuous-intros*]:
fixes *M* :: - $\Rightarrow$ - $\Rightarrow$ *'a*::*complete-lattice*
shows ($\bigwedge i$. $i \in I \Longrightarrow$ *sup-continuous* (*M i*)) $\Longrightarrow$ *sup-continuous* (λx. *SUP i*:*I*. *M i x*)
unfolding *SUP-apply*[*symmetric*] **by** (*rule sup-continuous-SUP*)

lemma *sup-continuous-lfp'*[*order-continuous-intros*]:
assumes *1*: *sup-continuous f*
assumes *2*: $\bigwedge g$. *sup-continuous g* $\Longrightarrow$ *sup-continuous* (*f g*)
shows *sup-continuous* (*lfp f*)
proof −
have *sup-continuous* ((*f* ^^ *i*) *bot*) **for** *i*
proof (*induction i*)
case (*Suc i*) **then show** *?case*
by (*auto intro*!: *2*)
qed (*simp add*: *bot-fun-def sup-continuous-const*)
then show *?thesis*
unfolding *sup-continuous-lfp*[*OF 1*] **by** (*intro order-continuous-intros*)
qed

lemma *sup-continuous-lfp''*[*order-continuous-intros*]:
assumes *1*: $\bigwedge s$. *sup-continuous* (*f s*)
assumes *2*: $\bigwedge g$. *sup-continuous g* $\Longrightarrow$ *sup-continuous* (λs. *f s* (*g s*))
shows *sup-continuous* (λx. *lfp* (*f x*))
proof −
have *sup-continuous* (λx. (*f x* ^^ *i*) *bot*) **for** *i*
proof (*induction i*)
case (*Suc i*) **then show** *?case*
by (*auto intro*!: *2*)
qed (*simp add*: *bot-fun-def sup-continuous-const*)

then show *?thesis*
unfolding *sup-continuous-lfp*[*OF 1*] **by** (*intro order-continuous-intros*)
qed

lemma *mono-INF-fun*:
$(\bigwedge x\ y.\ mono\ (F\ x\ y)) \Longrightarrow mono\ (\lambda z\ x.\ INF\ y : X\ x.\ F\ x\ y\ z :: 'a ::$ *complete-lattice*)
by (*auto intro*!: *INF-mono*[*OF bexI*] *simp*: *le-fun-def mono-def*)

lemma *continuous-on-max*:
fixes *f g* :: *'a*::*topological-space* ⇒ *'b*::*linorder-topology*
shows *continuous-on A f* ⟹ *continuous-on A g* ⟹ *continuous-on A* (λ*x*. *max* (*f x*) (*g x*))
by (*auto simp*: *continuous-on-def intro*!: *tendsto-max*)

lemma *continuous-on-cmult-ereal*:
|*c*::*ereal*| ≠ ∞ ⟹ *continuous-on A f* ⟹ *continuous-on A* (λ*x*. *c* ∗ *f x*)
using *tendsto-cmult-ereal*[*of c f f x at x within A* **for** *x*]
by (*auto simp*: *continuous-on-def simp del*: *tendsto-cmult-ereal*)

context *linordered-nonzero-semiring*
begin

lemma *of-nat-nonneg* [*simp*]: *0* ≤ *of-nat n*
by (*induct n*) *simp-all*

lemma *of-nat-mono*[*simp*]: *i* ≤ *j* ⟹ *of-nat i* ≤ *of-nat j*
by (*auto simp add*: *le-iff-add intro*!: *add-increasing2*)

end

lemma *real-of-nat-Sup*:
assumes *A* ≠ {} *bdd-above A*
shows *of-nat* (*Sup A*) = (*SUP a*:*A*. *of-nat a* :: *real*)
proof (*intro antisym*)
show (*SUP a*:*A*. *of-nat a*::*real*) ≤ *of-nat* (*Sup A*)
using *assms* **by** (*intro cSUP-least of-nat-mono*) (*auto intro*: *cSup-upper*)
have *Sup A* ∈ *A*
unfolding *Sup-nat-def* **using** *assms* **by** (*intro Max-in*) (*auto simp*: *bdd-above-nat*)
then show *of-nat* (*Sup A*) ≤ (*SUP a*:*A*. *of-nat a*::*real*)
by (*intro cSUP-upper bdd-above-image-mono assms*) (*auto simp*: *mono-def*)
qed

lemma *of-nat-less*[*simp*]:
i < *j* ⟹ *of-nat i* < (*of-nat j*::*'a*::{*linordered-nonzero-semiring*, *semiring-char-0*})
by (*auto simp*: *less-le*)

lemma *of-nat-le-iff* [*simp*]:
of-nat i ≤ (*of-nat j*::*'a*::{*linordered-nonzero-semiring*, *semiring-char-0*}) ⟷ *i*

$\leq j$
proof (*safe intro*!: *of-nat-mono*)
  assume *of-nat* $i \leq$ (*of-nat* j::'*a*) **then show** $i \leq j$
  proof (*intro leI notI*)
    assume $j < i$ **from** *less-le-trans*[*OF of-nat-less*[*OF this*] ‹*of-nat* $i \leq$ *of-nat* j›]
show *False*
      by *blast*
  qed
qed

lemma (**in** *complete-lattice*) *SUP-sup-const1*:
  $I \neq \{\} \Longrightarrow$ (*SUP* i:I. *sup* c (f i)) = *sup* c (*SUP* i:I. f i)
  using *SUP-sup-distrib*[*of* λ-. c I f] **by** *simp*

lemma (**in** *complete-lattice*) *SUP-sup-const2*:
  $I \neq \{\} \Longrightarrow$ (*SUP* i:I. *sup* (f i) c) = *sup* (*SUP* i:I. f i) c
  using *SUP-sup-distrib*[*of* f I λ-. c] **by** *simp*

lemma *one-less-of-natD*:
  (*1*::'*a*::*linordered-semidom*) < *of-nat* $n \Longrightarrow 1 < n$
  using *zero-le-one*[**where** '*a*='*a*]
  apply (*cases* n)
  apply *simp*
  subgoal for n'
    apply (*cases* n')
    apply *simp*
    apply *simp*
    done
  done

lemma *setsum-le-suminf*:
  fixes f :: *nat* ⇒ '*a*::{*ordered-comm-monoid-add*, *linorder-topology*}
  shows *summable* $f \Longrightarrow$ *finite* $I \Longrightarrow \forall m \in - I.\ 0 \leq f\ m \Longrightarrow$ *setsum* f $I \leq$ *suminf*
f
  by (*rule sums-le*[*OF* - *sums-If-finite-set summable-sums*]) *auto*

34.2 Defining the extended non-negative reals

Basic definitions and type class setup

typedef *ennreal* = {x :: *ereal*. $0 \leq x$}
  morphisms *enn2ereal e2ennreal'*
  by *auto*

definition *e2ennreal* x = *e2ennreal'* (*max* 0 x)

lemma *enn2ereal-range*: *e2ennreal* ‘ {*0*..} = *UNIV*
proof −
  have $\exists y \geq 0.\ x =$ *e2ennreal* y **for** x
    by (*cases* x) (*auto simp*: *e2ennreal-def max-absorb2*)

then show *?thesis*
by (*auto simp*: *image-iff Bex-def*)
qed

lemma *type-definition-ennreal′*: *type-definition enn2ereal e2ennreal* $\{x. \ 0 \leq x\}$
using *type-definition-ennreal*
by (*auto simp*: *type-definition-def e2ennreal-def max-absorb2*)

setup-lifting *type-definition-ennreal′*

declare [[*coercion e2ennreal*]]

instantiation *ennreal* :: *complete-linorder*
begin

lift-definition *top-ennreal* :: *ennreal* **is** *top* **by** (*rule top-greatest*)
lift-definition *bot-ennreal* :: *ennreal* **is** *0* **by** (*rule order-refl*)
lift-definition *sup-ennreal* :: *ennreal* ⇒ *ennreal* ⇒ *ennreal* **is** *sup* **by** (*rule le-supI1*)
lift-definition *inf-ennreal* :: *ennreal* ⇒ *ennreal* ⇒ *ennreal* **is** *inf* **by** (*rule le-infI*)

lift-definition *Inf-ennreal* :: *ennreal set* ⇒ *ennreal* **is** *Inf*
by (*rule Inf-greatest*)

lift-definition *Sup-ennreal* :: *ennreal set* ⇒ *ennreal* **is** *sup 0* ∘ *Sup*
by *auto*

lift-definition *less-eq-ennreal* :: *ennreal* ⇒ *ennreal* ⇒ *bool* **is** *op* ≤ **.**
lift-definition *less-ennreal* :: *ennreal* ⇒ *ennreal* ⇒ *bool* **is** *op* < **.**

instance
by *standard*
(*transfer* ; *auto simp*: *Inf-lower Inf-greatest Sup-upper Sup-least le-max-iff-disj max.absorb1*)+

end

lemma *pcr-ennreal-enn2ereal*[*simp*]: *pcr-ennreal* (*enn2ereal x*) *x*
by (*simp add*: *ennreal.pcr-cr-eq cr-ennreal-def*)

lemma *rel-fun-eq-pcr-ennreal*: *rel-fun op* = *pcr-ennreal f g* ⟷ *f* = *enn2ereal* ∘ *g*
by (*auto simp*: *rel-fun-def ennreal.pcr-cr-eq cr-ennreal-def*)

instantiation *ennreal* :: *infinity*
begin

definition *infinity-ennreal* :: *ennreal*
where
[*simp*]: ∞ = (*top*::*ennreal*)

instance ..

end

instantiation *ennreal* :: {*semiring-1-no-zero-divisors*, *comm-semiring-1*}
begin

lift-definition *one-ennreal* :: *ennreal* **is** *1* **by** *simp*
lift-definition *zero-ennreal* :: *ennreal* **is** *0* **by** *simp*
lift-definition *plus-ennreal* :: *ennreal* ⇒ *ennreal* ⇒ *ennreal* **is** *op* + **by** *simp*
lift-definition *times-ennreal* :: *ennreal* ⇒ *ennreal* ⇒ *ennreal* **is** *op* ∗ **by** *simp*

instance
  by *standard* (*transfer*; *auto simp*: *field-simps ereal-right-distrib*)+

end

instantiation *ennreal* :: *minus*
begin

lift-definition *minus-ennreal* :: *ennreal* ⇒ *ennreal* ⇒ *ennreal* **is** λ*a b*. *max 0* (*a* − *b*)
  by *simp*

instance ..

end

instance *ennreal* :: *numeral* **..**

instantiation *ennreal* :: *inverse*
begin

lift-definition *inverse-ennreal* :: *ennreal* ⇒ *ennreal* **is** *inverse*
  by (*rule inverse-ereal-ge0I*)

definition *divide-ennreal* :: *ennreal* ⇒ *ennreal* ⇒ *ennreal*
  where *x div y* = *x* ∗ *inverse* (*y* :: *ennreal*)

instance ..

end

lemma *ennreal-zero-less-one*: *0* < (*1*::*ennreal*) — TODO: remove
  by *transfer auto*

instance *ennreal* :: *dioid*
proof (*standard*; *transfer*)
  fix *a b* :: *ereal* **assume** *0* ≤ *a 0* ≤ *b* **then show** (*a* ≤ *b*) = (∃ *c*∈*Collect* (*op* ≤

0). *b* = *a* + *c*)
unfolding *ereal-ex-split Bex-def*
by (*cases a b rule*: *ereal2-cases*) (*auto intro*!: *exI*[*of* - *real-of-ereal* (*b* − *a*)])
qed

instance *ennreal* :: *ordered-comm-semiring*
by *standard*
(*transfer* ; *auto intro*: *add-mono mult-mono mult-ac ereal-left-distrib ereal-mult-left-mono*)+

instance *ennreal* :: *linordered-nonzero-semiring*
proof qed (*transfer*; *simp*)

instance *ennreal* :: *strict-ordered-ab-semigroup-add*
proof
fix *a b c d* :: *ennreal* **show** *a* < *b* ⟹ *c* < *d* ⟹ *a* + *c* < *b* + *d*
by *transfer* (*auto intro*!: *ereal-add-strict-mono*)
qed

declare [[*coercion of-nat* :: *nat* ⇒ *ennreal*]]

lemma *e2ennreal-neg*: *x* ≤ *0* ⟹ *e2ennreal x* = *0*
unfolding *zero-ennreal-def e2ennreal-def* **by** (*simp add*: *max-absorb1*)

lemma *e2ennreal-mono*: *x* ≤ *y* ⟹ *e2ennreal x* ≤ *e2ennreal y*
by (*cases 0* ≤ *x 0* ≤ *y rule*: *bool.exhaust*[*case-product bool.exhaust*])
(*auto simp*: *e2ennreal-neg less-eq-ennreal.abs-eq eq-onp-def*)

lemma *enn2ereal-nonneg*[*simp*]: *0* ≤ *enn2ereal x*
using *ennreal.enn2ereal*[*of x*] **by** *simp*

lemma *ereal-ennreal-cases*:
obtains *b* **where** *0* ≤ *a a* = *enn2ereal b* | *a* < *0*
using *e2ennreal′-inverse*[*of a*, *symmetric*] **by** (*cases 0* ≤ *a*) (*auto intro*: *enn2ereal-nonneg*)

lemma *rel-fun-liminf*[*transfer-rule*]: *rel-fun* (*rel-fun op* = *pcr-ennreal*) *pcr-ennreal liminf liminf*
proof −
have *rel-fun* (*rel-fun op* = *pcr-ennreal*) *pcr-ennreal* (λ*x*. *sup 0* (*liminf x*)) *liminf*
unfolding *liminf-SUP-INF*[*abs-def*] **by** (*transfer-prover-start*, *transfer-step*+; *simp*)
then show *?thesis*
apply (*subst* (*asm*) (*2*) *rel-fun-def*)
apply (*subst* (*2*) *rel-fun-def*)
apply (*auto simp*: *comp-def max.absorb2 Liminf-bounded rel-fun-eq-pcr-ennreal*)
done
qed

lemma *rel-fun-limsup*[*transfer-rule*]: *rel-fun* (*rel-fun op* = *pcr-ennreal*) *pcr-ennreal limsup limsup*

proof −
have *rel-fun* (*rel-fun op* = *pcr-ennreal*) *pcr-ennreal* (λx. *INF n. sup 0* (*SUP i*:{*n*..}. *x i*)) *limsup*
unfolding *limsup-INF-SUP*[*abs-def*] **by** (*transfer-prover-start*, *transfer-step*+; *simp*)
then show *?thesis*
unfolding *limsup-INF-SUP*[*abs-def*]
apply (*subst* (*asm*) (*2*) *rel-fun-def*)
apply (*subst* (*2*) *rel-fun-def*)
apply (*auto simp*: *comp-def max.absorb2 Sup-upper2 rel-fun-eq-pcr-ennreal*)
apply (*subst* (*asm*) *max.absorb2*)
apply (*rule SUP-upper2*)
apply *auto*
done
qed

lemma *setsum-enn2ereal*[*simp*]: ($\bigwedge i.\ i \in I \Longrightarrow 0 \le f\ i$) $\Longrightarrow$ ($\sum i{\in}I.$ *enn2ereal* (*f i*)) = *enn2ereal* (*setsum f I*)
by (*induction I rule*: *infinite-finite-induct*) (*auto simp*: *setsum-nonneg zero-ennreal.rep-eq plus-ennreal.rep-eq*)

lemma *transfer-e2ennreal-setsum* [*transfer-rule*]:
rel-fun (*rel-fun op* = *pcr-ennreal*) (*rel-fun op* = *pcr-ennreal*) *setsum setsum*
by (*auto intro*!: *rel-funI simp*: *rel-fun-eq-pcr-ennreal comp-def*)

lemma *enn2ereal-of-nat*[*simp*]: *enn2ereal* (*of-nat n*) = *ereal n*
by (*induction n*) (*auto simp*: *zero-ennreal.rep-eq one-ennreal.rep-eq plus-ennreal.rep-eq*)

lemma *enn2ereal-numeral*[*simp*]: *enn2ereal* (*numeral a*) = *numeral a*
apply (*subst of-nat-numeral*[*of a*, *symmetric*])
apply (*subst enn2ereal-of-nat*)
apply *simp*
done

lemma *transfer-numeral*[*transfer-rule*]: *pcr-ennreal* (*numeral a*) (*numeral a*)
unfolding *cr-ennreal-def pcr-ennreal-def* **by** *auto*

34.3 Cancellation simprocs

lemma *ennreal-add-left-cancel*: $a + b = a + c \longleftrightarrow a = (\infty{::}ennreal) \vee b = c$
unfolding *infinity-ennreal-def* **by** *transfer* (*simp add*: *top-ereal-def ereal-add-cancel-left*)

lemma *ennreal-add-left-cancel-le*: $a + b \le a + c \longleftrightarrow a = (\infty{::}ennreal) \vee b \le c$
unfolding *infinity-ennreal-def* **by** *transfer* (*simp add*: *ereal-add-le-add-iff top-ereal-def disj-commute*)

lemma *ereal-add-left-cancel-less*:
fixes *a b c* :: *ereal*
shows $0 \le a \Longrightarrow 0 \le b \Longrightarrow a + b < a + c \longleftrightarrow a \ne \infty \wedge b < c$

by (*cases a b c rule*: *ereal3-cases*) *auto*

lemma *ennreal-add-left-cancel-less*: $a + b < a + c \longleftrightarrow a \neq (\infty::ennreal) \wedge b < c$
unfolding *infinity-ennreal-def*
by *transfer* (*simp add*: *top-ereal-def ereal-add-left-cancel-less*)

ML ‹
```
structure Cancel-Ennreal-Common =
struct
  (* copied from src/HOL/Tools/nat-numeral-simprocs.ML *)
  fun find-first-t -    - []         = raise TERM(find-first-t, [])
    | find-first-t past u (t::terms) =
          if u aconv t then (rev past @ terms)
          else find-first-t (t::past) u terms

  fun dest-summing (Const (@{const-name Groups.plus}, -) $ t $ u, ts) =
        dest-summing (t, dest-summing (u, ts))
    | dest-summing (t, ts) = t :: ts

  val mk-sum = Arith-Data.long-mk-sum
  fun dest-sum t = dest-summing (t, [])
  val find-first = find-first-t []
  val trans-tac = Numeral-Simprocs.trans-tac
  val norm-ss =
    simpset-of (put-simpset HOL-basic-ss @{context}
      addsimps @{thms ac-simps add-0-left add-0-right})
  fun norm-tac ctxt = ALLGOALS (simp-tac (put-simpset norm-ss ctxt))
  fun simplify-meta-eq ctxt cancel-th th =
    Arith-Data.simplify-meta-eq [] ctxt
      ([th, cancel-th] MRS trans)
  fun mk-eq (a, b) = HOLogic.mk-Trueprop (HOLogic.mk-eq (a, b))
end

structure Eq-Ennreal-Cancel = ExtractCommonTermFun
(open Cancel-Ennreal-Common
  val mk-bal = HOLogic.mk-eq
  val dest-bal = HOLogic.dest-bin @{const-name HOL.eq} @{typ ennreal}
  fun simp-conv - - = SOME @{thm ennreal-add-left-cancel}
)

structure Le-Ennreal-Cancel = ExtractCommonTermFun
(open Cancel-Ennreal-Common
  val mk-bal = HOLogic.mk-binrel @{const-name Orderings.less-eq}
  val dest-bal = HOLogic.dest-bin @{const-name Orderings.less-eq} @{typ ennreal}
  fun simp-conv - - = SOME @{thm ennreal-add-left-cancel-le}
)

structure Less-Ennreal-Cancel = ExtractCommonTermFun
```

(*open Cancel-Ennreal-Common*
val mk-bal = *HOLogic.mk-binrel* @{*const-name Orderings.less*}
val dest-bal = *HOLogic.dest-bin* @{*const-name Orderings.less*} @{*typ ennreal*}
fun simp-conv - - = *SOME* @{*thm ennreal-add-left-cancel-less*}
)
›

simproc-setup *ennreal-eq-cancel*
((*l*::*ennreal*) + *m* = *n* | (*l*::*ennreal*) = *m* + *n*) =
‹*fn phi* => *fn ctxt* => *fn ct* => *Eq-Ennreal-Cancel.proc ctxt* (*Thm.term-of ct*)›

simproc-setup *ennreal-le-cancel*
((*l*::*ennreal*) + *m* ≤ *n* | (*l*::*ennreal*) ≤ *m* + *n*) =
‹*fn phi* => *fn ctxt* => *fn ct* => *Le-Ennreal-Cancel.proc ctxt* (*Thm.term-of ct*)›

simproc-setup *ennreal-less-cancel*
((*l*::*ennreal*) + *m* < *n* | (*l*::*ennreal*) < *m* + *n*) =
‹*fn phi* => *fn ctxt* => *fn ct* => *Less-Ennreal-Cancel.proc ctxt* (*Thm.term-of ct*)›

34.4 Order with top

lemma *ennreal-zero-less-top*[*simp*]: *0* < (*top*::*ennreal*)
by *transfer* (*simp add*: *top-ereal-def*)

lemma *ennreal-one-less-top*[*simp*]: *1* < (*top*::*ennreal*)
by *transfer* (*simp add*: *top-ereal-def*)

lemma *ennreal-zero-neq-top*[*simp*]: *0* ≠ (*top*::*ennreal*)
by *transfer* (*simp add*: *top-ereal-def*)

lemma *ennreal-top-neq-zero*[*simp*]: (*top*::*ennreal*) ≠ *0*
by *transfer* (*simp add*: *top-ereal-def*)

lemma *ennreal-top-neq-one*[*simp*]: *top* ≠ (*1*::*ennreal*)
by *transfer* (*simp add*: *top-ereal-def one-ereal-def ereal-max*[*symmetric*] *del*: *ereal-max*)

lemma *ennreal-one-neq-top*[*simp*]: *1* ≠ (*top*::*ennreal*)
by *transfer* (*simp add*: *top-ereal-def one-ereal-def ereal-max*[*symmetric*] *del*: *ereal-max*)

lemma *ennreal-add-less-top*[*simp*]:
fixes *a b* :: *ennreal*
shows *a* + *b* < *top* ⟷ *a* < *top* ∧ *b* < *top*
by *transfer* (*auto simp*: *top-ereal-def*)

lemma *ennreal-add-eq-top*[*simp*]:
fixes *a b* :: *ennreal*
shows *a* + *b* = *top* ⟷ *a* = *top* ∨ *b* = *top*
by *transfer* (*auto simp*: *top-ereal-def*)

lemma *ennreal-setsum-less-top*[*simp*]:
  fixes f :: $'a \Rightarrow ennreal$
  shows *finite I* $\Longrightarrow (\sum i{\in}I.\ f\ i) < top \longleftrightarrow (\forall\, i{\in}I.\ f\ i < top)$
  by (*induction I rule*: *finite-induct*) *auto*

lemma *ennreal-setsum-eq-top*[*simp*]:
  fixes f :: $'a \Rightarrow ennreal$
  shows *finite I* $\Longrightarrow (\sum i{\in}I.\ f\ i) = top \longleftrightarrow (\exists\, i{\in}I.\ f\ i = top)$
  by (*induction I rule*: *finite-induct*) *auto*

lemma *ennreal-mult-eq-top-iff*:
  fixes $a\ b$ :: *ennreal*
  shows $a * b = top \longleftrightarrow (a = top \wedge b \neq 0) \vee (b = top \wedge a \neq 0)$
  by *transfer* (*auto simp*: *top-ereal-def*)

lemma *ennreal-top-eq-mult-iff*:
  fixes $a\ b$ :: *ennreal*
  shows $top = a * b \longleftrightarrow (a = top \wedge b \neq 0) \vee (b = top \wedge a \neq 0)$
  using *ennreal-mult-eq-top-iff*[*of a b*] **by** *auto*

lemma *ennreal-mult-less-top*:
  fixes $a\ b$ :: *ennreal*
  shows $a * b < top \longleftrightarrow (a = 0 \vee b = 0 \vee (a < top \wedge b < top))$
  by *transfer* (*auto simp add*: *top-ereal-def*)

lemma *top-power-ennreal*: $top\ \hat{}\ n = (if\ n = 0\ then\ 1\ else\ top :: ennreal)$
  by (*induction n*) (*simp-all add*: *ennreal-mult-eq-top-iff*)

lemma *ennreal-setprod-eq-0*[*simp*]:
  fixes f :: $'a \Rightarrow ennreal$
  shows $(setprod\ f\ A = 0) = (finite\ A \wedge (\exists\, i{\in}A.\ f\ i = 0))$
  by (*induction A rule*: *infinite-finite-induct*) *auto*

lemma *ennreal-setprod-eq-top*:
  fixes f :: $'a \Rightarrow ennreal$
  shows $(\prod i{\in}I.\ f\ i) = top \longleftrightarrow (finite\ I \wedge ((\forall\, i{\in}I.\ f\ i \neq 0) \wedge (\exists\, i{\in}I.\ f\ i = top)))$
  by (*induction I rule*: *infinite-finite-induct*) (*auto simp*: *ennreal-mult-eq-top-iff*)

lemma *ennreal-top-mult*: $top * a = (if\ a = 0\ then\ 0\ else\ top :: ennreal)$
  by (*simp add*: *ennreal-mult-eq-top-iff*)

lemma *ennreal-mult-top*: $a * top = (if\ a = 0\ then\ 0\ else\ top :: ennreal)$
  by (*simp add*: *ennreal-mult-eq-top-iff*)

lemma *enn2ereal-eq-top-iff*[*simp*]: $enn2ereal\ x = \infty \longleftrightarrow x = top$
  by *transfer* (*simp add*: *top-ereal-def*)

lemma *enn2ereal-top*: $enn2ereal\ top = \infty$

by *transfer* (*simp add*: *top-ereal-def*)

lemma *e2ennreal-infty*: *e2ennreal* ∞ = *top*
 by (*simp add*: *top-ennreal.abs-eq top-ereal-def*)

lemma *ennreal-top-minus*[*simp*]: *top* − *x* = (*top*::*ennreal*)
 by *transfer* (*auto simp*: *top-ereal-def max-def*)

lemma *minus-top-ennreal*: *x* − *top* = (*if x* = *top then top else 0*:: *ennreal*)
 apply *transfer*
 subgoal for *x*
  by (*cases x*) (*auto simp*: *top-ereal-def max-def*)
 done

lemma *bot-ennreal*: *bot* = (*0*::*ennreal*)
 by *transfer rule*

lemma *ennreal-of-nat-neq-top*[*simp*]: *of-nat i* ≠ (*top*::*ennreal*)
 by (*induction i*) *auto*

lemma *numeral-eq-of-nat*: (*numeral a*::*ennreal*) = *of-nat* (*numeral a*)
 by *simp*

lemma *of-nat-less-top*: *of-nat i* < (*top*::*ennreal*)
 using *less-le-trans*[*of of-nat i of-nat* (*Suc i*) *top*::*ennreal*]
 by *simp*

lemma *top-neq-numeral*[*simp*]: *top* ≠ (*numeral i*::*ennreal*)
 using *of-nat-less-top*[*of numeral i*] **by** *simp*

lemma *ennreal-numeral-less-top*[*simp*]: *numeral i* < (*top*::*ennreal*)
 using *of-nat-less-top*[*of numeral i*] **by** *simp*

lemma *ennreal-add-bot*[*simp*]: *bot* + *x* = (*x*::*ennreal*)
 by *transfer simp*

instance *ennreal* :: *semiring-char-0*
proof (*standard*, *safe intro*!: *linorder-injI*)
 have ∗: *1* + *of-nat k* ≠ (*0*::*ennreal*) **for** *k*
  using *add-pos-nonneg*[*OF zero-less-one*, *of of-nat k* :: *ennreal*] **by** *auto*
 fix *x y* :: *nat* **assume** *x* < *y of-nat x* = (*of-nat y*::*ennreal*) **then show** *False*
  by (*auto simp add*: *less-iff-Suc-add* ∗)
qed

34.5 Arithmetic

lemma *ennreal-minus-zero*[*simp*]: *a* − (*0*::*ennreal*) = *a*
 by *transfer* (*auto simp*: *max-def*)

lemma *ennreal-add-diff-cancel-right*[*simp*]:
  fixes *x y z* :: *ennreal* **shows** $y \neq top \Longrightarrow (x + y) - y = x$
  apply *transfer*
  subgoal for *x y*
    apply (*cases x y rule*: *ereal2-cases*)
    apply (*auto split*: *split-max simp*: *top-ereal-def*)
    done
  done

lemma *ennreal-add-diff-cancel-left*[*simp*]:
  fixes *x y z* :: *ennreal* **shows** $y \neq top \Longrightarrow (y + x) - y = x$
  by (*simp add*: *add.commute*)

lemma
  fixes *a b* :: *ennreal*
  shows $a - b = 0 \Longrightarrow a \leq b$
  apply *transfer*
  subgoal for *a b*
    apply (*cases a b rule*: *ereal2-cases*)
    apply (*auto simp*: *not-le max-def split*: *if-splits*)
    done
  done

lemma *ennreal-minus-cancel*:
  fixes *a b c* :: *ennreal*
  shows $c \neq top \Longrightarrow a \leq c \Longrightarrow b \leq c \Longrightarrow c - a = c - b \Longrightarrow a = b$
  apply *transfer*
  subgoal for *a b c*
    by (*cases a b c rule*: *ereal3-cases*)
      (*auto simp*: *top-ereal-def max-def split*: *if-splits*)
  done

lemma *sup-const-add-ennreal*:
  fixes *a b c* :: *ennreal*
  shows *sup* $(c + a)$ $(c + b) = c +$ *sup a b*
  apply *transfer*
  subgoal for *a b c*
    apply (*cases a b c rule*: *ereal3-cases*)
    apply (*auto simp*: *ereal-max*[*symmetric*] *simp del*: *ereal-max*)
    apply (*auto simp*: *top-ereal-def*[*symmetric*] *sup-ereal-def*[*symmetric*]
          simp del: *sup-ereal-def*)
    apply (*auto simp add*: *top-ereal-def*)
    done
  done

lemma *ennreal-diff-add-assoc*:
  fixes *a b c* :: *ennreal*
  shows $a \leq b \Longrightarrow c + b - a = c + (b - a)$
  apply *transfer*

subgoal for *a b c*
by (*cases a b c rule*: *ereal3-cases*) (*auto simp*: *field-simps max-absorb2*)
done

lemma *mult-divide-eq-ennreal*:
fixes *a b* :: *ennreal*
shows $b \neq 0 \Longrightarrow b \neq top \Longrightarrow (a * b) / b = a$
unfolding *divide-ennreal-def*
apply *transfer*
apply (*subst mult.assoc*)
apply (*simp add*: *top-ereal-def divide-ereal-def*[*symmetric*])
done

lemma *divide-mult-eq*: $a \neq 0 \Longrightarrow a \neq \infty \Longrightarrow x * a / (b * a) = x / (b::ennreal)$
unfolding *divide-ennreal-def infinity-ennreal-def*
apply *transfer*
subgoal for *a b c*
apply (*cases a b c rule*: *ereal3-cases*)
apply (*auto simp*: *top-ereal-def*)
done
done

lemma *ennreal-mult-divide-eq*:
fixes *a b* :: *ennreal*
shows $b \neq 0 \Longrightarrow b \neq top \Longrightarrow (a * b) / b = a$
unfolding *divide-ennreal-def*
apply *transfer*
apply (*subst mult.assoc*)
apply (*simp add*: *top-ereal-def divide-ereal-def*[*symmetric*])
done

lemma *ennreal-add-diff-cancel*:
fixes *a b* :: *ennreal*
shows $b \neq \infty \Longrightarrow (a + b) - b = a$
unfolding *infinity-ennreal-def*
by *transfer* (*simp add*: *max-absorb2 top-ereal-def ereal-add-diff-cancel*)

lemma *ennreal-minus-eq-0*:
$a - b = 0 \Longrightarrow a \leq (b::ennreal)$
apply *transfer*
subgoal for *a b*
apply (*cases a b rule*: *ereal2-cases*)
apply (*auto simp*: *zero-ereal-def ereal-max*[*symmetric*] *max.absorb2 simp del*: *ereal-max*)
done
done

lemma *ennreal-mono-minus-cancel*:
fixes *a b c* :: *ennreal*

shows $a - b \le a - c \Longrightarrow a < top \Longrightarrow b \le a \Longrightarrow c \le a \Longrightarrow c \le b$
by *transfer*
(*auto simp add*: *max.absorb2 ereal-diff-positive top-ereal-def dest*: *ereal-mono-minus-cancel*)

lemma *ennreal-mono-minus*:
fixes *a b c* :: *ennreal*
shows $c \le b \Longrightarrow a - b \le a - c$
apply *transfer*
apply (*rule max.mono*)
apply *simp*
subgoal for *a b c*
by (*cases a b c rule*: *ereal3-cases*) *auto*
done

lemma *ennreal-minus-pos-iff*:
fixes *a b* :: *ennreal*
shows $a < top \lor b < top \Longrightarrow 0 < a - b \Longrightarrow b < a$
apply *transfer*
subgoal for *a b*
by (*cases a b rule*: *ereal2-cases*) (*auto simp*: *less-max-iff-disj*)
done

lemma *ennreal-inverse-top*[*simp*]: *inverse top* = (*0*::*ennreal*)
by *transfer* (*simp add*: *top-ereal-def ereal-inverse-eq-0*)

lemma *ennreal-inverse-zero*[*simp*]: *inverse 0* = (*top*::*ennreal*)
by *transfer* (*simp add*: *top-ereal-def ereal-inverse-eq-0*)

lemma *ennreal-top-divide*: *top* / (*x*::*ennreal*) = (*if x* = *top then 0 else top*)
unfolding *divide-ennreal-def*
by *transfer* (*simp add*: *top-ereal-def ereal-inverse-eq-0 ereal-0-gt-inverse*)

lemma *ennreal-zero-divide*[*simp*]: *0* / (*x*::*ennreal*) = *0*
by (*simp add*: *divide-ennreal-def*)

lemma *ennreal-divide-zero*[*simp*]: *x* / (*0*::*ennreal*) = (*if x* = *0 then 0 else top*)
by (*simp add*: *divide-ennreal-def ennreal-mult-top*)

lemma *ennreal-divide-top*[*simp*]: *x* / (*top*::*ennreal*) = *0*
by (*simp add*: *divide-ennreal-def ennreal-top-mult*)

lemma *ennreal-times-divide*: *a* ∗ (*b* / *c*) = *a* ∗ *b* / (*c*::*ennreal*)
unfolding *divide-ennreal-def*
by *transfer* (*simp add*: *divide-ereal-def*[*symmetric*] *ereal-times-divide-eq*)

lemma *ennreal-zero-less-divide*: $0 < a / b \longleftrightarrow (0 < a \land b < (top{::}ennreal))$
unfolding *divide-ennreal-def*
by *transfer* (*auto simp*: *ereal-zero-less-0-iff top-ereal-def ereal-0-gt-inverse*)

lemma *divide-right-mono-ennreal*:
fixes *a b c* :: *ennreal*
shows $a \le b \Longrightarrow a \ / \ c \le b \ / \ c$
unfolding *divide-ennreal-def* **by** (*intro mult-mono*) *auto*

lemma *ennreal-mult-strict-right-mono*: $(a::ennreal) < c \Longrightarrow 0 < b \Longrightarrow b < top \Longrightarrow a * b < c * b$
by *transfer* (*auto intro*!: *ereal-mult-strict-right-mono*)

lemma *ennreal-indicator-less*[*simp*]:
$indicator\ A\ x \le (indicator\ B\ x::ennreal) \longleftrightarrow (x \in A \longrightarrow x \in B)$
by (*simp add*: *indicator-def not-le*)

lemma *ennreal-inverse-positive*: $0 < inverse\ x \longleftrightarrow (x::ennreal) \neq top$
by *transfer* (*simp add*: *ereal-0-gt-inverse top-ereal-def*)

lemma *ennreal-inverse-mult'*: $((0 < b \lor a < top) \land (0 < a \lor b < top)) \Longrightarrow inverse\ (a * b::ennreal) = inverse\ a * inverse\ b$
apply *transfer*
subgoal for *a b*
by (*cases a b rule*: *ereal2-cases*) (*auto simp*: *top-ereal-def*)
done

lemma *ennreal-inverse-mult*: $a < top \Longrightarrow b < top \Longrightarrow inverse\ (a * b::ennreal) = inverse\ a * inverse\ b$
apply *transfer*
subgoal for *a b*
by (*cases a b rule*: *ereal2-cases*) (*auto simp*: *top-ereal-def*)
done

lemma *ennreal-inverse-1*[*simp*]: $inverse\ (1::ennreal) = 1$
by *transfer simp*

lemma *ennreal-inverse-eq-0-iff*[*simp*]: $inverse\ (a::ennreal) = 0 \longleftrightarrow a = top$
by *transfer* (*simp add*: *ereal-inverse-eq-0 top-ereal-def*)

lemma *ennreal-inverse-eq-top-iff*[*simp*]: $inverse\ (a::ennreal) = top \longleftrightarrow a = 0$
by *transfer* (*simp add*: *top-ereal-def*)

lemma *ennreal-divide-eq-0-iff*[*simp*]: $(a::ennreal) \ / \ b = 0 \longleftrightarrow (a = 0 \lor b = top)$
by (*simp add*: *divide-ennreal-def*)

lemma *ennreal-divide-eq-top-iff*: $(a::ennreal) \ / \ b = top \longleftrightarrow ((a \neq 0 \land b = 0) \lor (a = top \land b \neq top))$
by (*auto simp add*: *divide-ennreal-def ennreal-mult-eq-top-iff*)

lemma *one-divide-one-divide-ennreal*[*simp*]: $1 \ / \ (1 \ / \ c) = (c::ennreal)$
including *ennreal.lifting*
unfolding *divide-ennreal-def*

by *transfer auto*

lemma *ennreal-mult-left-cong*:
$((a{::}ennreal) \neq 0 \Longrightarrow b = c) \Longrightarrow a * b = a * c$
by (*cases* $a = 0$) *simp-all*

lemma *ennreal-mult-right-cong*:
$((a{::}ennreal) \neq 0 \Longrightarrow b = c) \Longrightarrow b * a = c * a$
by (*cases* $a = 0$) *simp-all*

lemma *ennreal-zero-less-mult-iff*: $0 < a * b \longleftrightarrow 0 < a \wedge 0 < (b{::}ennreal)$
by *transfer* (*auto simp add*: *ereal-zero-less-0-iff le-less*)

lemma *less-diff-eq-ennreal*:
fixes *a b c* :: *ennreal*
shows $b < top \vee c < top \Longrightarrow a < b - c \longleftrightarrow a + c < b$
apply *transfer*
subgoal for *a b c*
by (*cases a b c rule*: *ereal3-cases*) (*auto split*: *split-max*)
done

lemma *diff-add-cancel-ennreal*:
fixes *a b* :: *ennreal* **shows** $a \leq b \Longrightarrow b - a + a = b$
unfolding *infinity-ennreal-def*
apply *transfer*
subgoal for *a b*
by (*cases a b rule*: *ereal2-cases*) (*auto simp*: *max-absorb2*)
done

lemma *ennreal-diff-self*[*simp*]: $a \neq top \Longrightarrow a - a = (0{::}ennreal)$
by *transfer* (*simp add*: *top-ereal-def*)

lemma *ennreal-minus-mono*:
fixes *a b c* :: *ennreal*
shows $a \leq c \Longrightarrow d \leq b \Longrightarrow a - b \leq c - d$
apply *transfer*
apply (*rule max.mono*)
apply *simp*
subgoal for *a b c d*
by (*cases a b c d rule*: *ereal3-cases*[*case-product ereal-cases*]) *auto*
done

lemma *ennreal-minus-eq-top*[*simp*]: $a - (b{::}ennreal) = top \longleftrightarrow a = top$
by *transfer* (*auto simp*: *top-ereal-def max.absorb2 ereal-minus-eq-PInfty-iff split*: *split-max*)

lemma *ennreal-divide-self*[*simp*]: $a \neq 0 \Longrightarrow a < top \Longrightarrow a / a = (1{::}ennreal)$
unfolding *divide-ennreal-def*
apply *transfer*

subgoal for *a*
by (*cases a*) (*auto simp*: *top-ereal-def*)
done

34.6 Coercion from *real* to *ennreal*

lift-definition *ennreal* :: *real* ⇒ *ennreal* **is** *sup 0* ∘ *ereal*
by *simp*

declare [[*coercion ennreal*]]

lemma *ennreal-cases*[*cases type*: *ennreal*]:
fixes *x* :: *ennreal*
obtains (*real*) *r* :: *real* **where** *0* ≤ *r x* = *ennreal r* | (*top*) *x* = *top*
apply *transfer*
subgoal for *x thesis*
by (*cases x*) (*auto simp*: *max.absorb2 top-ereal-def*)
done

lemmas *ennreal2-cases* = *ennreal-cases*[*case-product ennreal-cases*]
lemmas *ennreal3-cases* = *ennreal-cases*[*case-product ennreal2-cases*]

lemma *ennreal-neq-top*[*simp*]: *ennreal r* ≠ *top*
by *transfer* (*simp add*: *top-ereal-def zero-ereal-def ereal-max*[*symmetric*] *del*: *ereal-max*)

lemma *top-neq-ennreal*[*simp*]: *top* ≠ *ennreal r*
using *ennreal-neq-top*[*of r*] **by** (*auto simp del*: *ennreal-neq-top*)

lemma *ennreal-less-top*[*simp*]: *ennreal x* < *top*
by *transfer* (*simp add*: *top-ereal-def max-def*)

lemma *ennreal-neg*: *x* ≤ *0* ⟹ *ennreal x* = *0*
by *transfer* (*simp add*: *max.absorb1*)

lemma *ennreal-inj*[*simp*]:
0 ≤ *a* ⟹ *0* ≤ *b* ⟹ *ennreal a* = *ennreal b* ⟷ *a* = *b*
by (*transfer fixing*: *a b*) (*auto simp*: *max-absorb2*)

lemma *ennreal-le-iff*[*simp*]: *0* ≤ *y* ⟹ *ennreal x* ≤ *ennreal y* ⟷ *x* ≤ *y*
by (*auto simp*: *ennreal-def zero-ereal-def less-eq-ennreal.abs-eq eq-onp-def split*: *split-max*)

lemma *le-ennreal-iff*: *0* ≤ *r* ⟹ *x* ≤ *ennreal r* ⟷ (∃ *q*≥*0*. *x* = *ennreal q* ∧ *q* ≤ *r*)
by (*cases x*) (*auto simp*: *top-unique*)

lemma *ennreal-less-iff*: *0* ≤ *r* ⟹ *ennreal r* < *ennreal q* ⟷ *r* < *q*
unfolding *not-le*[*symmetric*] **by** *auto*

lemma *ennreal-eq-zero-iff* [*simp*]: $0 \leq x \Longrightarrow ennreal\ x = 0 \longleftrightarrow x = 0$
by *transfer* (*auto simp*: *max-absorb2*)

lemma *ennreal-less-zero-iff* [*simp*]: $0 < ennreal\ x \longleftrightarrow 0 < x$
by *transfer* (*auto simp*: *max-def*)

lemma *ennreal-lessI*: $0 < q \Longrightarrow r < q \Longrightarrow ennreal\ r < ennreal\ q$
by (*cases* $0 \leq r$) (*auto simp*: *ennreal-less-iff ennreal-neg*)

lemma *ennreal-leI*: $x \leq y \Longrightarrow ennreal\ x \leq ennreal\ y$
by (*cases* $0 \leq y$) (*auto simp*: *ennreal-neg*)

lemma *enn2ereal-ennreal*[*simp*]: $0 \leq x \Longrightarrow enn2ereal\ (ennreal\ x) = x$
by *transfer* (*simp add*: *max-absorb2*)

lemma *e2ennreal-enn2ereal*[*simp*]: $e2ennreal\ (enn2ereal\ x) = x$
by (*simp add*: *e2ennreal-def max-absorb2 ennreal.enn2ereal-inverse*)

lemma *ennreal-0*[*simp*]: $ennreal\ 0 = 0$
by (*simp add*: *ennreal-def max.absorb1 zero-ennreal.abs-eq*)

lemma *ennreal-1*[*simp*]: $ennreal\ 1 = 1$
by *transfer* (*simp add*: *max-absorb2*)

lemma *ennreal-eq-0-iff*: $ennreal\ x = 0 \longleftrightarrow x \leq 0$
by (*cases* $0 \leq x$) (*auto simp*: *ennreal-neg*)

lemma *ennreal-le-iff2*: $ennreal\ x \leq ennreal\ y \longleftrightarrow ((0 \leq y \wedge x \leq y) \vee (x \leq 0 \wedge y \leq 0))$
by (*cases* $0 \leq y$) (*auto simp*: *ennreal-eq-0-iff ennreal-neg*)

lemma *ennreal-eq-1*[*simp*]: $ennreal\ x = 1 \longleftrightarrow x = 1$
by (*cases* $0 \leq x$)
(*auto simp*: *ennreal-neg ennreal-1*[*symmetric*] *simp del*: *ennreal-1*)

lemma *ennreal-le-1*[*simp*]: $ennreal\ x \leq 1 \longleftrightarrow x \leq 1$
by (*cases* $0 \leq x$)
(*auto simp*: *ennreal-neg ennreal-1*[*symmetric*] *simp del*: *ennreal-1*)

lemma *ennreal-ge-1*[*simp*]: $ennreal\ x \geq 1 \longleftrightarrow x \geq 1$
by (*cases* $0 \leq x$)
(*auto simp*: *ennreal-neg ennreal-1*[*symmetric*] *simp del*: *ennreal-1*)

lemma *ennreal-plus*[*simp*]:
$0 \leq a \Longrightarrow 0 \leq b \Longrightarrow ennreal\ (a + b) = ennreal\ a + ennreal\ b$
by (*transfer fixing*: *a b*) (*auto simp*: *max-absorb2*)

lemma *setsum-ennreal*[*simp*]: $(\bigwedge i.\ i \in I \Longrightarrow 0 \leq f\ i) \Longrightarrow (\sum i{\in}I.\ ennreal\ (f\ i))$

= *ennreal* (*setsum f I*)
by (*induction I rule*: *infinite-finite-induct*) (*auto simp*: *setsum-nonneg*)

lemma *ennreal-of-nat-eq-real-of-nat*: *of-nat i* = *ennreal* (*of-nat i*)
by (*induction i*) *simp-all*

lemma *of-nat-le-ennreal-iff* [*simp*]: $0 \leq r \Longrightarrow$ *of-nat i* $\leq$ *ennreal r* $\longleftrightarrow$ *of-nat i* $\leq$ *r*
by (*simp add*: *ennreal-of-nat-eq-real-of-nat*)

lemma *ennreal-le-of-nat-iff* [*simp*]: *ennreal r* $\leq$ *of-nat i* $\longleftrightarrow$ $r \leq$ *of-nat i*
by (*simp add*: *ennreal-of-nat-eq-real-of-nat*)

lemma *ennreal-indicator*: *ennreal* (*indicator A x*) = *indicator A x*
by (*auto split*: *split-indicator*)

lemma *ennreal-numeral*[*simp*]: *ennreal* (*numeral n*) = *numeral n*
using *ennreal-of-nat-eq-real-of-nat*[*of numeral n*] **by** *simp*

lemma *min-ennreal*: $0 \leq x \Longrightarrow 0 \leq y \Longrightarrow$ *min* (*ennreal x*) (*ennreal y*) = *ennreal* (*min x y*)
by (*auto split*: *split-min*)

lemma *ennreal-half* [*simp*]: *ennreal* (*1/2*) = *inverse 2*
by *transfer* (*simp add*: *max.absorb2*)

lemma *ennreal-minus*: $0 \leq q \Longrightarrow$ *ennreal r* − *ennreal q* = *ennreal* (*r* − *q*)
by *transfer*
(*simp add*: *max.absorb2 zero-ereal-def ereal-max*[*symmetric*] *del*: *ereal-max*)

lemma *ennreal-minus-top*[*simp*]: *ennreal a* − *top* = *0*
by (*simp add*: *minus-top-ennreal*)

lemma *ennreal-mult*: $0 \leq a \Longrightarrow 0 \leq b \Longrightarrow$ *ennreal* (*a* ∗ *b*) = *ennreal a* ∗ *ennreal b*
by *transfer* (*simp add*: *max-absorb2*)

lemma *ennreal-mult'*: $0 \leq a \Longrightarrow$ *ennreal* (*a* ∗ *b*) = *ennreal a* ∗ *ennreal b*
by (*cases 0* ≤ *b*) (*auto simp*: *ennreal-mult ennreal-neg mult-nonneg-nonpos*)

lemma *indicator-mult-ennreal*: *indicator A x* ∗ *ennreal r* = *ennreal* (*indicator A x* ∗ *r*)
by (*simp split*: *split-indicator*)

lemma *ennreal-mult''*: $0 \leq b \Longrightarrow$ *ennreal* (*a* ∗ *b*) = *ennreal a* ∗ *ennreal b*
by (*cases 0* ≤ *a*) (*auto simp*: *ennreal-mult ennreal-neg mult-nonpos-nonneg*)

lemma *numeral-mult-ennreal*: $0 \leq x \Longrightarrow$ *numeral b* ∗ *ennreal x* = *ennreal* (*numeral b* ∗ *x*)

by (*simp add*: *ennreal-mult*)

lemma *ennreal-power*: $0 \leq r \Longrightarrow ennreal\ r\ \hat{}\ n = ennreal\ (r\ \hat{}\ n)$
by (*induction n*) (*auto simp*: *ennreal-mult*)

lemma *power-eq-top-ennreal*: $x\ \hat{}\ n = top \longleftrightarrow (n \neq 0 \wedge (x::ennreal) = top)$
by (*cases x rule*: *ennreal-cases*)
(*auto simp*: *ennreal-power top-power-ennreal*)

lemma *inverse-ennreal*: $0 < r \Longrightarrow inverse\ (ennreal\ r) = ennreal\ (inverse\ r)$
by *transfer* (*simp add*: *max.absorb2*)

lemma *divide-ennreal*: $0 \leq r \Longrightarrow 0 < q \Longrightarrow ennreal\ r\ /\ ennreal\ q = ennreal\ (r\ /\ q)$
by (*simp add*: *divide-ennreal-def inverse-ennreal ennreal-mult*[*symmetric*] *inverse-eq-divide*)

lemma *ennreal-inverse-power*: $inverse\ (x\ \hat{}\ n :: ennreal) = inverse\ x\ \hat{}\ n$
proof (*cases x rule*: *ennreal-cases*)
case *top* **with** *power-eq-top-ennreal*[*of x n*] **show** *?thesis*
by (*cases* $n = 0$) *auto*
next
case (*real r*) **then show** *?thesis*
proof *cases*
assume $x = 0$ **then show** *?thesis*
using *power-eq-top-ennreal*[*of top* $n - 1$]
by (*cases n*) (*auto simp*: *ennreal-top-mult*)
next
assume $x \neq 0$
with *real* **have** $0 < r$ **by** *auto*
with *real* **show** *?thesis*
by (*induction n*)
(*auto simp add*: *ennreal-power ennreal-mult*[*symmetric*] *inverse-ennreal*)
qed
qed

lemma *ennreal-divide-numeral*: $0 \leq x \Longrightarrow ennreal\ x\ /\ numeral\ b = ennreal\ (x\ /\ numeral\ b)$
by (*subst divide-ennreal*[*symmetric*]) *auto*

lemma *setprod-ennreal*: $(\bigwedge i.\ i \in A \Longrightarrow 0 \leq f\ i) \Longrightarrow (\prod i \in A.\ ennreal\ (f\ i)) = ennreal\ (setprod\ f\ A)$
by (*induction A rule*: *infinite-finite-induct*)
(*auto simp*: *ennreal-mult setprod-nonneg*)

lemma *mult-right-ennreal-cancel*: $a * ennreal\ c = b * ennreal\ c \longleftrightarrow (a = b \vee c \leq 0)$
apply (*cases* $0 \leq c$)
apply (*cases a b rule*: *ennreal2-cases*)
apply (*auto simp*: *ennreal-mult*[*symmetric*] *ennreal-neg ennreal-top-mult*)

done

lemma *ennreal-le-epsilon*:
($\bigwedge e$::*real*. $y < top \Longrightarrow 0 < e \Longrightarrow x \leq y + ennreal\ e) \Longrightarrow x \leq y$
apply (*cases y rule*: *ennreal-cases*)
apply (*cases x rule*: *ennreal-cases*)
apply (*auto simp del*: *ennreal-plus simp add*: *top-unique ennreal-plus*[*symmetric*]
intro: *zero-less-one field-le-epsilon*)
done

lemma *ennreal-rat-dense*:
fixes *x y* :: *ennreal*
shows $x < y \Longrightarrow \exists r$::*rat*. $x <$ *real-of-rat r* $\wedge$ *real-of-rat r* $< y$
proof *transfer*
fix *x y* :: *ereal* **assume** *xy*: $0 \leq x$ $0 \leq y$ $x < y$
moreover
from *ereal-dense3*[*OF* ‹$x < y$›]
obtain *r* **where** $x <$ *ereal* (*real-of-rat r*) *ereal* (*real-of-rat r*) $< y$
by *auto*
moreover then have $0 \leq r$
using *le-less-trans*[*OF* ‹$0 \leq x$› ‹$x <$ *ereal* (*real-of-rat r*)›] **by** *auto*
ultimately show $\exists r$. $x <$ (*sup 0* $\circ$ *ereal*) (*real-of-rat r*) $\wedge$ (*sup 0* $\circ$ *ereal*) (*real-of-rat r*) $< y$
by (*intro exI*[*of - r*]) (*auto simp*: *max-absorb2*)
qed

lemma *ennreal-Ex-less-of-nat*: (*x*::*ennreal*) $<$ *top* $\Longrightarrow \exists n$. $x <$ *of-nat n*
by (*cases x rule*: *ennreal-cases*)
(*auto simp*: *ennreal-of-nat-eq-real-of-nat ennreal-less-iff reals-Archimedean2*)

34.7 Coercion from *ennreal* to *real*

definition *enn2real x* = *real-of-ereal* (*enn2ereal x*)

lemma *enn2real-nonneg*[*simp*]: $0 \leq$ *enn2real x*
by (*auto simp*: *enn2real-def intro*!: *real-of-ereal-pos enn2ereal-nonneg*)

lemma *enn2real-mono*: $a \leq b \Longrightarrow b <$ *top* $\Longrightarrow$ *enn2real a* $\leq$ *enn2real b*
by (*auto simp add*: *enn2real-def less-eq-ennreal.rep-eq intro*!: *real-of-ereal-positive-mono enn2ereal-nonneg*)

lemma *enn2real-of-nat*[*simp*]: *enn2real* (*of-nat n*) = *n*
by (*auto simp*: *enn2real-def*)

lemma *enn2real-ennreal*[*simp*]: $0 \leq r \Longrightarrow$ *enn2real* (*ennreal r*) = *r*
by (*simp add*: *enn2real-def*)

lemma *ennreal-enn2real*[*simp*]: $r <$ *top* $\Longrightarrow$ *ennreal* (*enn2real r*) = *r*
by (*cases r rule*: *ennreal-cases*) *auto*

lemma *real-of-ereal-enn2ereal*[*simp*]: *real-of-ereal* (*enn2ereal x*) = *enn2real x*
by (*simp add*: *enn2real-def*)

lemma *enn2real-top*[*simp*]: *enn2real top* = *0*
unfolding *enn2real-def top-ennreal.rep-eq top-ereal-def* **by** *simp*

lemma *enn2real-0*[*simp*]: *enn2real 0* = *0*
unfolding *enn2real-def zero-ennreal.rep-eq* **by** *simp*

lemma *enn2real-1*[*simp*]: *enn2real 1* = *1*
unfolding *enn2real-def one-ennreal.rep-eq* **by** *simp*

lemma *enn2real-numeral*[*simp*]: *enn2real* (*numeral n*) = (*numeral n*)
unfolding *enn2real-def* **by** *simp*

lemma *enn2real-mult*: *enn2real* (*a* ∗ *b*) = *enn2real a* ∗ *enn2real b*
unfolding *enn2real-def*
by (*simp del*: *real-of-ereal-enn2ereal add*: *times-ennreal.rep-eq*)

lemma *enn2real-leI*: *0* ≤ *B* ⟹ *x* ≤ *ennreal B* ⟹ *enn2real x* ≤ *B*
by (*cases x rule*: *ennreal-cases*) (*auto simp*: *top-unique*)

lemma *enn2real-positive-iff*: *0* < *enn2real x* ⟷ (*0* < *x* ∧ *x* < *top*)
by (*cases x rule*: *ennreal-cases*) *auto*

34.8 Coercion from *enat* to *ennreal*

definition *ennreal-of-enat* :: *enat* ⇒ *ennreal*
where
ennreal-of-enat n = (*case n of* ∞ ⇒ *top* | *enat n* ⇒ *of-nat n*)

declare [[*coercion ennreal-of-enat*]]
declare [[*coercion of-nat* :: *nat* ⇒ *ennreal*]]

lemma *ennreal-of-enat-infty*[*simp*]: *ennreal-of-enat* ∞ = ∞
by (*simp add*: *ennreal-of-enat-def*)

lemma *ennreal-of-enat-enat*[*simp*]: *ennreal-of-enat* (*enat n*) = *of-nat n*
by (*simp add*: *ennreal-of-enat-def*)

lemma *ennreal-of-enat-0*[*simp*]: *ennreal-of-enat 0* = *0*
using *ennreal-of-enat-enat*[*of 0*] **unfolding** *enat-0* **by** *simp*

lemma *ennreal-of-enat-1*[*simp*]: *ennreal-of-enat 1* = *1*
using *ennreal-of-enat-enat*[*of 1*] **unfolding** *enat-1* **by** *simp*

lemma *ennreal-top-neq-of-nat*[*simp*]: (*top*::*ennreal*) ≠ *of-nat i*
using *ennreal-of-nat-neq-top*[*of i*] **by** *metis*

lemma *ennreal-of-enat-inj*[*simp*]: *ennreal-of-enat* i = *ennreal-of-enat* $j \longleftrightarrow i = j$
  by (*cases* i j *rule*: *enat.exhaust*[*case-product enat.exhaust*]) *auto*

lemma *ennreal-of-enat-le-iff* [*simp*]: *ennreal-of-enat* $m \leq$ *ennreal-of-enat* $n \longleftrightarrow m \leq n$
  by (*auto simp*: *ennreal-of-enat-def top-unique split*: *enat.split*)

lemma *of-nat-less-ennreal-of-nat*[*simp*]: *of-nat* $n \leq$ *ennreal-of-enat* $x \longleftrightarrow$ *of-nat* $n \leq x$
  by (*cases* x) (*auto simp*: *of-nat-eq-enat*)

lemma *ennreal-of-enat-Sup*: *ennreal-of-enat* (*Sup* X) = (*SUP* x:X. *ennreal-of-enat* x)
proof −
  have *ennreal-of-enat* (*Sup* X) ≤ (*SUP* x : X. *ennreal-of-enat* x)
    unfolding *Sup-enat-def*
  proof (*clarsimp*, *intro conjI impI*)
    fix x **assume** *finite* X $X \neq \{\}$
    then show *ennreal-of-enat* (*Max* X) ≤ (*SUP* x : X. *ennreal-of-enat* x)
      by (*intro SUP-upper Max-in*)
  next
    assume *infinite* X $X \neq \{\}$
    have $\exists y \in X$. $r <$ *ennreal-of-enat* y **if** r: $r <$ *top* **for** r
    proof −
      from *ennreal-Ex-less-of-nat*[*OF* r] **guess** n **..** **note** n = *this*
      have ¬ ($X \subseteq$ *enat* ‘ {.. n})
        using ‹*infinite* X› **by** (*auto dest*: *finite-subset*)
      then obtain x **where** $x \in X$ $x \notin$ *enat* ‘ {..n}
        by *blast*
      moreover then have *of-nat* $n \leq x$
        by (*cases* x) (*auto simp*: *of-nat-eq-enat*)
      ultimately show *?thesis*
        by (*auto intro*!: *bexI*[*of - x*] *less-le-trans*[*OF n*])
    qed
    then have (*SUP* x : X. *ennreal-of-enat* x) = *top*
      by *simp*
    then show *top* ≤ (*SUP* x : X. *ennreal-of-enat* x)
      unfolding *top-unique* **by** *simp*
  qed
  then show *?thesis*
    by (*auto intro*!: *antisym Sup-least intro*: *Sup-upper*)
qed

lemma *ennreal-of-enat-eSuc*[*simp*]: *ennreal-of-enat* (*eSuc* x) = *1* + *ennreal-of-enat* x
  by (*cases* x) (*auto simp*: *eSuc-enat*)

34.9 Topology on *ennreal*

lemma *enn2ereal-Iio*: *enn2ereal* −' {..<*a*} = (*if 0* ≤ *a then* {..< *e2ennreal a*} *else* {})
  using *enn2ereal-nonneg*
  by (*cases a rule*: *ereal-ennreal-cases*)
    (*auto simp add*: *vimage-def set-eq-iff ennreal.enn2ereal-inverse less-ennreal.rep-eq e2ennreal-def max-absorb2*
      simp del: *enn2ereal-nonneg*
      intro: *le-less-trans less-imp-le*)

lemma *enn2ereal-Ioi*: *enn2ereal* −' {*a* <..} = (*if 0* ≤ *a then* {*e2ennreal a* <..} *else UNIV*)
  by (*cases a rule*: *ereal-ennreal-cases*)
    (*auto simp add*: *vimage-def set-eq-iff ennreal.enn2ereal-inverse less-ennreal.rep-eq e2ennreal-def max-absorb2*
      intro: *less-le-trans*)

instantiation *ennreal* :: *linear-continuum-topology*
begin

definition *open-ennreal* :: *ennreal set* ⇒ *bool*
  where (*open* :: *ennreal set* ⇒ *bool*) = *generate-topology* (*range lessThan* ∪ *range greaterThan*)

instance
proof
  show ∃ *a b*::*ennreal*. *a* ≠ *b*
    using *zero-neq-one* **by** (*intro exI*)
  show ⋀*x y*::*ennreal*. *x* < *y* ⟹ ∃ *z*>*x*. *z* < *y*
  proof *transfer*
    fix *x y* :: *ereal* **assume** *0* ≤ *x x* < *y*
    moreover from *dense*[*OF this*(*2*)] **guess** *z* **..**
    ultimately show ∃ *z*∈*Collect* (*op* ≤ *0*). *x* < *z* ∧ *z* < *y*
      by (*intro bexI*[*of - z*]) *auto*
  qed
qed (*rule open-ennreal-def*)

end

lemma *continuous-on-e2ennreal*: *continuous-on A e2ennreal*
proof (*rule continuous-on-subset*)
  show *continuous-on* ({*0*..} ∪ {..*0*}) *e2ennreal*
  proof (*rule continuous-on-closed-Un*)
    show *continuous-on* {*0* ..} *e2ennreal*
      by (*rule continuous-onI-mono*)
        (*auto simp add*: *less-eq-ennreal.abs-eq eq-onp-def enn2ereal-range*)
    show *continuous-on* {.. *0*} *e2ennreal*
      by (*subst continuous-on-cong*[*OF refl*, *of - - λ-. 0*])
        (*auto simp add*: *e2ennreal-neg continuous-on-const*)

qed *auto*
show $A \subseteq \{0..\} \cup \{..0::ereal\}$
by *auto*
qed

lemma *continuous-at-e2ennreal*: *continuous* (*at x within A*) *e2ennreal*
by (*rule continuous-on-imp-continuous-within*[*OF continuous-on-e2ennreal*, *of - UNIV*]) *auto*

lemma *continuous-on-enn2ereal*: *continuous-on UNIV enn2ereal*
by (*rule continuous-on-generate-topology*[*OF open-generated-order*])
(*auto simp add*: *enn2ereal-Iio enn2ereal-Ioi*)

lemma *continuous-at-enn2ereal*: *continuous* (*at x within A*) *enn2ereal*
by (*rule continuous-on-imp-continuous-within*[*OF continuous-on-enn2ereal*]) *auto*

lemma *sup-continuous-e2ennreal*[*order-continuous-intros*]:
assumes *f*: *sup-continuous f* **shows** *sup-continuous* ($\lambda x.$ *e2ennreal* (*f x*))
apply (*rule sup-continuous-compose*[*OF - f*])
apply (*rule continuous-at-left-imp-sup-continuous*)
apply (*auto simp*: *mono-def e2ennreal-mono continuous-at-e2ennreal*)
done

lemma *sup-continuous-enn2ereal*[*order-continuous-intros*]:
assumes *f*: *sup-continuous f* **shows** *sup-continuous* ($\lambda x.$ *enn2ereal* (*f x*))
apply (*rule sup-continuous-compose*[*OF - f*])
apply (*rule continuous-at-left-imp-sup-continuous*)
apply (*simp-all add*: *mono-def less-eq-ennreal.rep-eq continuous-at-enn2ereal*)
done

lemma *sup-continuous-mult-left-ennreal'*:
fixes *c* :: *ennreal*
shows *sup-continuous* ($\lambda x.\ c * x$)
unfolding *sup-continuous-def*
by *transfer* (*auto simp*: *SUP-ereal-mult-left max.absorb2 SUP-upper2*)

lemma *sup-continuous-mult-left-ennreal*[*order-continuous-intros*]:
sup-continuous f $\Longrightarrow$ *sup-continuous* ($\lambda x.\ c * f\ x$ :: *ennreal*)
by (*rule sup-continuous-compose*[*OF sup-continuous-mult-left-ennreal'*])

lemma *sup-continuous-mult-right-ennreal*[*order-continuous-intros*]:
sup-continuous f $\Longrightarrow$ *sup-continuous* ($\lambda x.\ f\ x * c$ :: *ennreal*)
using *sup-continuous-mult-left-ennreal*[*of f c*] **by** (*simp add*: *mult.commute*)

lemma *sup-continuous-divide-ennreal*[*order-continuous-intros*]:
fixes *f g* :: *'a::complete-lattice* $\Rightarrow$ *ennreal*
shows *sup-continuous f* $\Longrightarrow$ *sup-continuous* ($\lambda x.\ f\ x\ /\ c$)
unfolding *divide-ennreal-def* **by** (*rule sup-continuous-mult-right-ennreal*)

lemma *transfer-enn2ereal-continuous-on* [*transfer-rule*]:
rel-fun (*op* =) (*rel-fun* (*rel-fun op* = *pcr-ennreal*) *op* =) *continuous-on continuous-on*
proof –
have *continuous-on A f* **if** *continuous-on A* (λ*x*. *enn2ereal* (*f x*)) **for** *A* **and** *f* :: ′*a* ⇒ *ennreal*
using *continuous-on-compose2*[*OF continuous-on-e2ennreal*[*of* {*0*..}] *that*]
by (*auto simp*: *ennreal.enn2ereal-inverse subset-eq e2ennreal-def max-absorb2*)
moreover
have *continuous-on A* (λ*x*. *enn2ereal* (*f x*)) **if** *continuous-on A f* **for** *A* **and** *f* :: ′*a* ⇒ *ennreal*
using *continuous-on-compose2*[*OF continuous-on-enn2ereal that*] **by** *auto*
ultimately
show *?thesis*
by (*auto simp add*: *rel-fun-def ennreal.pcr-cr-eq cr-ennreal-def*)
qed

lemma *transfer-sup-continuous*[*transfer-rule*]:
(*rel-fun* (*rel-fun* (*op* =) *pcr-ennreal*) *op* =) *sup-continuous sup-continuous*
proof (*safe intro*!: *rel-funI dest*!: *rel-fun-eq-pcr-ennreal*[*THEN iffD1*])
show *sup-continuous* (*enn2ereal* ∘ *f*) ⟹ *sup-continuous f* **for** *f* :: ′*a* ⇒ -
using *sup-continuous-e2ennreal*[*of enn2ereal* ∘ *f*] **by** *simp*
show *sup-continuous f* ⟹ *sup-continuous* (*enn2ereal* ∘ *f*) **for** *f* :: ′*a* ⇒ -
using *sup-continuous-enn2ereal*[*of f*] **by** (*simp add*: *comp-def*)
qed

lemma *continuous-on-ennreal*[*tendsto-intros*]:
continuous-on A f ⟹ *continuous-on A* (λ*x*. *ennreal* (*f x*))
by *transfer* (*auto intro*!: *continuous-on-max continuous-on-const continuous-on-ereal*)

lemma *tendsto-ennrealD*:
assumes *lim*: ((λ*x*. *ennreal* (*f x*)) ⟶ *ennreal x*) *F*
assumes ∗: ∀$_F$ *x in F*. *0* ≤ *f x* **and** *x*: *0* ≤ *x*
shows (*f* ⟶ *x*) *F*
using *continuous-on-tendsto-compose*[*OF continuous-on-enn2ereal lim*]
apply *simp*
apply (*subst* (*asm*) *tendsto-cong*)
using ∗
apply *eventually-elim*
apply (*auto simp*: *max-absorb2* ‹*0* ≤ *x*›)
done

lemma *tendsto-ennreal-iff* [*simp*]:
∀$_F$ *x in F*. *0* ≤ *f x* ⟹ *0* ≤ *x* ⟹ ((λ*x*. *ennreal* (*f x*)) ⟶ *ennreal x*) *F* ⟷ (*f* ⟶ *x*) *F*
by (*auto dest*: *tendsto-ennrealD*)
(*auto simp*: *ennreal-def*
intro!: *continuous-on-tendsto-compose*[*OF continuous-on-e2ennreal*[*of UNIV*]] *tendsto-max*)

lemma *tendsto-enn2ereal-iff* [*simp*]: $((\lambda i.\ enn2ereal\ (f\ i)) \longrightarrow enn2ereal\ x)\ F \longleftrightarrow (f \longrightarrow x)\ F$
using *continuous-on-enn2ereal*[*THEN continuous-on-tendsto-compose*, *of f x F*]
continuous-on-e2ennreal[*THEN continuous-on-tendsto-compose*, *of* $\lambda x.$ *enn2ereal* (*f x*) *enn2ereal x F UNIV*]
by *auto*

lemma *continuous-on-add-ennreal*:
fixes *f g* :: *'a::topological-space* ⇒ *ennreal*
shows *continuous-on A f* ⟹ *continuous-on A g* ⟹ *continuous-on A* ($\lambda x.$ *f x* + *g x*)
by (*transfer fixing*: *A*) (*auto intro*!: *tendsto-add-ereal-nonneg simp*: *continuous-on-def*)

lemma *continuous-on-inverse-ennreal*[*continuous-intros*]:
fixes *f* :: *'a::topological-space* ⇒ *ennreal*
shows *continuous-on A f* ⟹ *continuous-on A* ($\lambda x.$ *inverse* (*f x*))
proof (*transfer fixing*: *A*)
show *pred-fun* (λ-. *True*) (*op* ≤ *0*) *f* ⟹ *continuous-on A* ($\lambda x.$ *inverse* (*f x*)) **if** *continuous-on A f*
for *f* :: *'a* ⇒ *ereal*
using *continuous-on-compose2*[*OF continuous-on-inverse-ereal that*] **by** (*auto simp*: *subset-eq*)
qed

instance *ennreal* :: *topological-comm-monoid-add*
proof
show (($\lambda x.$ *fst x* + *snd x*) ⟶ *a* + *b*) (*nhds a* $\times_F$ *nhds b*) **for** *a b* :: *ennreal*
using *continuous-on-add-ennreal*[*of UNIV fst snd*]
using *tendsto-at-iff-tendsto-nhds*[*symmetric*, *of* λx::(*ennreal* × *ennreal*). *fst x* + *snd x*]
by (*auto simp*: *continuous-on-eq-continuous-at*)
(*simp add*: *isCont-def nhds-prod*[*symmetric*])
qed

lemma *sup-continuous-add-ennreal*[*order-continuous-intros*]:
fixes *f g* :: *'a::complete-lattice* ⇒ *ennreal*
shows *sup-continuous f* ⟹ *sup-continuous g* ⟹ *sup-continuous* ($\lambda x.$ *f x* + *g x*)
by *transfer* (*auto intro*!: *sup-continuous-add*)

lemma *ennreal-suminf-lessD*: ($\sum$ *i*. *f i* :: *ennreal*) < *x* ⟹ *f i* < *x*
using *le-less-trans*[*OF setsum-le-suminf*[*OF summableI*, *of* {*i*} *f*]] **by** *simp*

lemma *sums-ennreal*[*simp*]: ($\bigwedge i.\ 0 \le f\ i$) ⟹ *0* ≤ *x* ⟹ ($\lambda i.$ *ennreal* (*f i*)) *sums ennreal x* ⟷ *f sums x*
unfolding *sums-def* **by** (*simp add*: *always-eventually setsum-nonneg*)

lemma *summable-suminf-not-top*: ($\bigwedge i.\ 0 \le f\ i$) ⟹ ($\sum$ *i*. *ennreal* (*f i*)) ≠ *top* ⟹ *summable f*

using *summable-sums*[*OF summableI*, *of* $\lambda i.$ *ennreal* (*f i*)]
by (*cases* $\sum i.$ *ennreal* (*f i*) *rule*: *ennreal-cases*)
(*auto simp*: *summable-def*)

lemma *suminf-ennreal*[*simp*]:
$(\bigwedge i.\ 0 \le f\ i) \Longrightarrow (\sum i.\ ennreal\ (f\ i)) \neq top \Longrightarrow (\sum i.\ ennreal\ (f\ i)) = ennreal\ (\sum i.\ f\ i)$
by (*rule sums-unique*[*symmetric*]) (*simp add*: *summable-suminf-not-top suminf-nonneg summable-sums*)

lemma *sums-enn2ereal*[*simp*]: $(\lambda i.\ enn2ereal\ (f\ i))$ *sums enn2ereal x* $\longleftrightarrow$ *f sums x*
unfolding *sums-def* **by** (*simp add*: *always-eventually setsum-nonneg*)

lemma *suminf-enn2ereal*[*simp*]: $(\sum i.\ enn2ereal\ (f\ i)) = enn2ereal\ (suminf\ f)$
by (*rule sums-unique*[*symmetric*]) (*simp add*: *summable-sums*)

lemma *transfer-e2ennreal-suminf* [*transfer-rule*]: *rel-fun* (*rel-fun op* = *pcr-ennreal*) *pcr-ennreal suminf suminf*
by (*auto simp*: *rel-funI rel-fun-eq-pcr-ennreal comp-def*)

lemma *ennreal-suminf-cmult*[*simp*]: $(\sum i.\ r * f\ i) = r * (\sum i.\ f\ i::ennreal)$
by *transfer* (*auto intro*!: *suminf-cmult-ereal*)

lemma *ennreal-suminf-multc*[*simp*]: $(\sum i.\ f\ i * r) = (\sum i.\ f\ i::ennreal) * r$
using *ennreal-suminf-cmult*[*of r f*] **by** (*simp add*: *ac-simps*)

lemma *ennreal-suminf-divide*[*simp*]: $(\sum i.\ f\ i\ /\ r) = (\sum i.\ f\ i::ennreal)\ /\ r$
by (*simp add*: *divide-ennreal-def*)

lemma *ennreal-suminf-neq-top*: *summable f* $\Longrightarrow (\bigwedge i.\ 0 \le f\ i) \Longrightarrow (\sum i.\ ennreal\ (f\ i)) \neq top$
using *sums-ennreal*[*of f suminf f*]
by (*simp add*: *suminf-nonneg sums-unique*[*symmetric*] *summable-sums-iff*[*symmetric*] *del*: *sums-ennreal*)

lemma *suminf-ennreal-eq*:
$(\bigwedge i.\ 0 \le f\ i) \Longrightarrow$ *f sums x* $\Longrightarrow (\sum i.\ ennreal\ (f\ i)) = ennreal\ x$
using *suminf-nonneg*[*of f*] *sums-unique*[*of f x*]
by (*intro sums-unique*[*symmetric*]) (*auto simp*: *summable-sums-iff*)

lemma *ennreal-suminf-bound-add*:
fixes $f :: nat \Rightarrow ennreal$
shows $(\bigwedge N.\ (\sum n<N.\ f\ n) + y \le x) \Longrightarrow suminf\ f + y \le x$
by *transfer* (*auto intro*!: *suminf-bound-add*)

lemma *ennreal-suminf-SUP-eq-directed*:
fixes $f :: 'a \Rightarrow nat \Rightarrow ennreal$
assumes $*$: $\bigwedge N\ i\ j.\ i \in I \Longrightarrow j \in I \Longrightarrow finite\ N \Longrightarrow \exists\, k{\in}I.\ \forall\, n{\in}N.\ f\ i\ n \le f\ k$

$n \wedge f\ j\ n \leq f\ k\ n$
shows $(\sum n.\ SUP\ i{:}I.\ f\ i\ n) = (SUP\ i{:}I.\ \sum n.\ f\ i\ n)$
proof *cases*
assume $I \neq \{\}$
then obtain i **where** $i \in I$ **by** *auto*
from $*$ **show** *?thesis*
by (*transfer fixing*: I)
(*auto simp*: *max-absorb2 SUP-upper2*[*OF* ‹$i \in I$›] *suminf-nonneg summable-ereal-pos* ‹$I \neq \{\}$›
intro!: *suminf-SUP-eq-directed*)
qed (*simp add*: *bot-ennreal*)

lemma *INF-ennreal-add-const*:
fixes *f g* :: *nat* ⇒ *ennreal*
shows $(INF\ i.\ f\ i + c) = (INF\ i.\ f\ i) + c$
using *continuous-at-Inf-mono*[*of* $\lambda x.\ x + c$ *f'UNIV*]
using *continuous-add*[*of at-right* (*Inf* (*range f*)), *of* $\lambda x.\ x\ \lambda x.\ c$]
by (*auto simp*: *mono-def*)

lemma *INF-ennreal-const-add*:
fixes *f g* :: *nat* ⇒ *ennreal*
shows $(INF\ i.\ c + f\ i) = c + (INF\ i.\ f\ i)$
using *INF-ennreal-add-const*[*of f c*] **by** (*simp add*: *ac-simps*)

lemma *SUP-mult-left-ennreal*: $c * (SUP\ i{:}I.\ f\ i) = (SUP\ i{:}I.\ c * f\ i\ ::ennreal)$
proof *cases*
assume $I \neq \{\}$ **then show** *?thesis*
by *transfer* (*auto simp add*: *SUP-ereal-mult-left max-absorb2 SUP-upper2*)
qed (*simp add*: *bot-ennreal*)

lemma *SUP-mult-right-ennreal*: $(SUP\ i{:}I.\ f\ i) * c = (SUP\ i{:}I.\ f\ i * c\ ::ennreal)$
using *SUP-mult-left-ennreal* **by** (*simp add*: *mult.commute*)

lemma *SUP-divide-ennreal*: $(SUP\ i{:}I.\ f\ i)\ /\ c = (SUP\ i{:}I.\ f\ i\ /\ c\ ::ennreal)$
using *SUP-mult-right-ennreal* **by** (*simp add*: *divide-ennreal-def*)

lemma *ennreal-SUP-of-nat-eq-top*: $(SUP\ x.\ of\text{-}nat\ x\ ::\ ennreal) = top$
proof (*intro antisym top-greatest le-SUP-iff*[*THEN iffD2*] *allI impI*)
fix *y* :: *ennreal* **assume** $y < top$
then obtain *r* **where** $y = ennreal\ r$
by (*cases y rule*: *ennreal-cases*) *auto*
then show $\exists\, i{\in}UNIV.\ y < of\text{-}nat\ i$
using *reals-Archimedean2*[*of max 1 r*] *zero-less-one*
by (*auto simp*: *ennreal-of-nat-eq-real-of-nat ennreal-def less-ennreal.abs-eq eq-onp-def max.absorb2*
dest!: *one-less-of-natD intro*: *less-trans*)
qed

lemma *ennreal-SUP-eq-top*:

fixes f :: $'a \Rightarrow$ *ennreal*
assumes $\bigwedge n.\ \exists\, i{\in}I.$ *of-nat* $n \leq f\ i$
shows $(SUP\ i : I.\ f\ i) = top$
proof −
have $(SUP\ x.\ \textit{of-nat}\ x :: \textit{ennreal}) \leq (SUP\ i : I.\ f\ i)$
using *assms* **by** (*auto intro*!: *SUP-least intro*: *SUP-upper2*)
then show *?thesis*
by (*auto simp*: *ennreal-SUP-of-nat-eq-top top-unique*)
qed

lemma *ennreal-INF-const-minus*:
fixes f :: $'a \Rightarrow$ *ennreal*
shows $I \neq \{\} \Longrightarrow (SUP\ x{:}I.\ c - f\ x) = c - (INF\ x{:}I.\ f\ x)$
by (*transfer fixing*: I)
(*simp add*: *sup-max*[*symmetric*] *SUP-sup-const1 SUP-ereal-minus-right del*: *sup-ereal-def*)

lemma *of-nat-Sup-ennreal*:
assumes $A \neq \{\}$ *bdd-above* A
shows *of-nat* (*Sup* A) = (*SUP* $a{:}A$. *of-nat* a :: *ennreal*)
proof (*intro antisym*)
show (*SUP* $a{:}A$. *of-nat* a::*ennreal*) $\leq$ *of-nat* (*Sup* A)
by (*intro SUP-least of-nat-mono*) (*auto intro*: *cSup-upper assms*)
have *Sup* $A \in A$
unfolding *Sup-nat-def* **using** *assms* **by** (*intro Max-in*) (*auto simp*: *bdd-above-nat*)
then show *of-nat* (*Sup* A) $\leq$ (*SUP* $a{:}A$. *of-nat* a::*ennreal*)
by (*intro SUP-upper*)
qed

lemma *ennreal-tendsto-const-minus*:
fixes g :: $'a \Rightarrow$ *ennreal*
assumes *ae*: $\forall_F\ x$ *in* $F.\ g\ x \leq c$
assumes g: $((\lambda x.\ c - g\ x) \longrightarrow 0)\ F$
shows $(g \longrightarrow c)\ F$
proof (*cases c rule*: *ennreal-cases*)
case *top* **with** *tendsto-unique*[*OF* - g, *of top*] **show** *?thesis*
by (*cases* $F = bot$) *auto*
next
case (*real* r)
then have $\forall\, x.\ \exists\, q{\geq}0.\ g\ x \leq c \longrightarrow (g\ x = \textit{ennreal}\ q \wedge q \leq r)$
by (*auto simp*: *le-ennreal-iff*)
then obtain f **where** $*$: $\bigwedge x.\ g\ x \leq c \Longrightarrow 0 \leq f\ x\ \bigwedge x.\ g\ x \leq c \Longrightarrow g\ x = \textit{ennreal}\ (f\ x)\ \bigwedge x.\ g\ x \leq c \Longrightarrow f\ x \leq r$
by *metis*
from *ae* **have** *ae2*: $\forall_F\ x$ *in* $F.\ c - g\ x = \textit{ennreal}\ (r - f\ x) \wedge f\ x \leq r \wedge g\ x = \textit{ennreal}\ (f\ x) \wedge 0 \leq f\ x$
proof *eventually-elim*
fix x **assume** $g\ x \leq c$ **with** $*$[*of* x] ‹$0 \leq r$› **show** $c - g\ x = \textit{ennreal}\ (r - f\ x) \wedge f\ x \leq r \wedge g\ x = \textit{ennreal}\ (f\ x) \wedge 0 \leq f\ x$

by (*auto simp*: *real ennreal-minus*)
qed
with *g* **have** $((\lambda x.\ ennreal\ (r - f\ x)) \longlongrightarrow ennreal\ 0)\ F$
by (*auto simp add*: *tendsto-cong eventually-conj-iff*)
with *ae2* **have** $((\lambda x.\ r - f\ x) \longrightarrow 0)\ F$
by (*subst* (*asm*) *tendsto-ennreal-iff*) (*auto elim*: *eventually-mono*)
then have $(f \longrightarrow r)\ F$
by (*rule Lim-transform2*[*OF tendsto-const*])
with *ae2* **have** $((\lambda x.\ ennreal\ (f\ x)) \longrightarrow ennreal\ r)\ F$
by (*subst tendsto-ennreal-iff*) (*auto elim*: *eventually-mono simp*: *real*)
with *ae2* **show** *?thesis*
by (*auto simp*: *real tendsto-cong eventually-conj-iff*)
qed

lemma *ennreal-SUP-add*:
fixes $f\ g :: nat \Rightarrow ennreal$
shows $incseq\ f \Longrightarrow incseq\ g \Longrightarrow (SUP\ i.\ f\ i + g\ i) = SUPREMUM\ UNIV\ f + SUPREMUM\ UNIV\ g$
unfolding *incseq-def le-fun-def*
by *transfer*
(*simp add*: *SUP-ereal-add incseq-def le-fun-def max-absorb2 SUP-upper2*)

lemma *ennreal-SUP-setsum*:
fixes $f :: 'a \Rightarrow nat \Rightarrow ennreal$
shows $(\bigwedge i.\ i \in I \Longrightarrow incseq\ (f\ i)) \Longrightarrow (SUP\ n.\ \sum i\in I.\ f\ i\ n) = (\sum i\in I.\ SUP\ n.\ f\ i\ n)$
unfolding *incseq-def*
by *transfer*
(*simp add*: *SUP-ereal-setsum incseq-def SUP-upper2 max-absorb2 setsum-nonneg*)

lemma *ennreal-liminf-minus*:
fixes $f :: nat \Rightarrow ennreal$
shows $(\bigwedge n.\ f\ n \le c) \Longrightarrow liminf\ (\lambda n.\ c - f\ n) = c - limsup\ f$
apply *transfer*
apply (*simp add*: *ereal-diff-positive max.absorb2 liminf-ereal-cminus*)
apply (*subst max.absorb2*)
apply (*rule ereal-diff-positive*)
apply (*rule Limsup-bounded*)
apply *auto*
done

lemma *ennreal-continuous-on-cmult*:
$(c::ennreal) < top \Longrightarrow continuous\text{-}on\ A\ f \Longrightarrow continuous\text{-}on\ A\ (\lambda x.\ c * f\ x)$
by (*transfer fixing*: *A*) (*auto intro*: *continuous-on-cmult-ereal*)

lemma *ennreal-tendsto-cmult*:
$(c::ennreal) < top \Longrightarrow (f \longrightarrow x)\ F \Longrightarrow ((\lambda x.\ c * f\ x) \longrightarrow c * x)\ F$
by (*rule continuous-on-tendsto-compose*[**where** *g*=*f*, *OF ennreal-continuous-on-cmult*, **where** *s*=*UNIV*])

(*auto simp*: *continuous-on-id*)

lemma *tendsto-ennrealI*[*intro*, *simp*]:
$(f \longrightarrow x)\ F \Longrightarrow ((\lambda x.\ ennreal\ (f\ x)) \longrightarrow ennreal\ x)\ F$
by (*auto simp*: *ennreal-def*
intro!: *continuous-on-tendsto-compose*[*OF continuous-on-e2ennreal*[*of UNIV*]] *tendsto-max*)

lemma *ennreal-suminf-minus*:
fixes $f\ g :: nat \Rightarrow ennreal$
shows $(\bigwedge i.\ g\ i \leq f\ i) \Longrightarrow suminf\ f \neq top \Longrightarrow suminf\ g \neq top \Longrightarrow (\sum i.\ f\ i - g\ i) = suminf\ f - suminf\ g$
by *transfer*
(*auto simp add*: *max.absorb2 ereal-diff-positive suminf-le-pos top-ereal-def intro*!: *suminf-ereal-minus*)

lemma *ennreal-Sup-countable-SUP*:
$A \neq \{\} \Longrightarrow \exists f::nat \Rightarrow ennreal.\ incseq\ f \wedge range\ f \subseteq A \wedge Sup\ A = (SUP\ i.\ f\ i)$
unfolding *incseq-def*
apply *transfer*
subgoal for *A*
using *Sup-countable-SUP*[*of A*]
apply (*clarsimp simp add*: *incseq-def*[*symmetric*] *SUP-upper2 max.absorb2 image-subset-iff Sup-upper2 cong*: *conj-cong*)
subgoal for *f*
by (*intro exI*[*of - f*]) *auto*
done
done

lemma *ennreal-SUP-countable-SUP*:
$A \neq \{\} \Longrightarrow \exists f::nat \Rightarrow ennreal.\ range\ f \subseteq g`A \wedge SUPREMUM\ A\ g = SUPREMUM\ UNIV\ f$
using *ennreal-Sup-countable-SUP* [*of g'A*] **by** *auto*

lemma *of-nat-tendsto-top-ennreal*: $(\lambda n::nat.\ of\text{-}nat\ n :: ennreal) \longrightarrow top$
using *LIMSEQ-SUP*[*of of-nat* :: *nat* ⇒ *ennreal*]
by (*simp add*: *ennreal-SUP-of-nat-eq-top incseq-def*)

lemma *SUP-sup-continuous-ennreal*:
fixes $f :: ennreal \Rightarrow 'a::complete\text{-}lattice$
assumes *f*: *sup-continuous f* **and** $I \neq \{\}$
shows $(SUP\ i{:}I.\ f\ (g\ i)) = f\ (SUP\ i{:}I.\ g\ i)$
proof (*rule antisym*)
show $(SUP\ i{:}I.\ f\ (g\ i)) \leq f\ (SUP\ i{:}I.\ g\ i)$
by (*rule mono-SUP*[*OF sup-continuous-mono*[*OF f*]])
from *ennreal-Sup-countable-SUP*[*of g'I*] ‹$I \neq \{\}$›
obtain $M :: nat \Rightarrow ennreal$ **where** *incseq M* **and** *M*: $range\ M \subseteq g\ `\ I$ **and** *eq*: $(SUP\ i : I.\ g\ i) = (SUP\ i.\ M\ i)$
by *auto*

have *f* (*SUP i* : *I*. *g i*) = (*SUP i* : *range M*. *f i*)
  unfolding *eq sup-continuousD*[*OF f* ‹*mono M*›] **by** *simp*
also have ... ≤ (*SUP i* : *I*. *f* (*g i*))
  by (*insert M*, *drule SUP-subset-mono*) *auto*
finally show *f* (*SUP i* : *I*. *g i*) ≤ (*SUP i* : *I*. *f* (*g i*)) **.**
qed

lemma *ennreal-suminf-SUP-eq*:
  fixes *f* :: *nat* ⇒ *nat* ⇒ *ennreal*
  shows (⋀*i*. *incseq* (λ*n*. *f n i*)) ⟹ (∑ *i*. *SUP n*. *f n i*) = (*SUP n*. ∑ *i*. *f n i*)
  apply (*rule ennreal-suminf-SUP-eq-directed*)
  subgoal for *N n j*
    by (*auto simp*: *incseq-def intro*!:*exI*[*of - max n j*])
  done

lemma *ennreal-SUP-add-left*:
  fixes *c* :: *ennreal*
  shows *I* ≠ {} ⟹ (*SUP i*:*I*. *f i* + *c*) = (*SUP i*:*I*. *f i*) + *c*
  apply *transfer*
  apply (*simp add*: *SUP-ereal-add-left*)
  apply (*subst* (*1 2*) *max.absorb2*)
  apply (*auto intro*: *SUP-upper2 ereal-add-nonneg-nonneg*)
  done

lemma *ennreal-SUP-const-minus*:
  fixes *f* :: ′*a* ⇒ *ennreal*
  shows *I* ≠ {} ⟹ *c* < *top* ⟹ (*INF x*:*I*. *c* − *f x*) = *c* − (*SUP x*:*I*. *f x*)
  apply (*transfer fixing*: *I*)
  unfolding *ex-in-conv*[*symmetric*]
  apply (*auto simp add*: *sup-max*[*symmetric*] *SUP-upper2 sup-absorb2*
      simp del: *sup-ereal-def*)
  apply (*subst INF-ereal-minus-right*[*symmetric*])
  apply (*auto simp del*: *sup-ereal-def simp add*: *sup-INF*)
  done

34.10 Approximation lemmas

lemma *INF-approx-ennreal*:
  fixes *x*::*ennreal* **and** *e*::*real*
  assumes *e* > *0*
  assumes *INF*: *x* = (*INF i* : *A*. *f i*)
  assumes *x* ≠ ∞
  shows ∃ *i* ∈ *A*. *f i* < *x* + *e*
proof −
  have (*INF i* : *A*. *f i*) < *x* + *e*
    unfolding *INF*[*symmetric*] **using** ‹*0*<*e*› ‹*x* ≠ ∞› **by** (*cases x*) *auto*
  then show *?thesis*
    unfolding *INF-less-iff* **.**
qed

lemma *SUP-approx-ennreal*:
  fixes *x*::*ennreal* **and** *e*::*real*
  assumes $e > 0$ $A \neq \{\}$
  assumes *SUP*: $x = (SUP\ i : A.\ f\ i)$
  assumes $x \neq \infty$
  shows $\exists\, i \in A.\ x < f\ i + e$
proof −
  have $x < x + e$
    using ‹$0<e$› ‹$x \neq \infty$› **by** (*cases x*) *auto*
  also have $x + e = (SUP\ i : A.\ f\ i + e)$
    unfolding *SUP ennreal-SUP-add-left*[*OF* ‹$A \neq \{\}$›] **..**
  finally show *?thesis*
    unfolding *less-SUP-iff* **.**
qed

lemma *ennreal-approx-SUP*:
  fixes *x*::*ennreal*
  assumes *f-bound*: $\bigwedge i.\ i \in A \Longrightarrow f\ i \leq x$
  assumes *approx*: $\bigwedge e.\ (e{::}real) > 0 \Longrightarrow \exists\, i \in A.\ x \leq f\ i + e$
  shows $x = (SUP\ i : A.\ f\ i)$
proof (*rule antisym*)
  show $x \leq (SUP\ i{:}A.\ f\ i)$
  proof (*rule ennreal-le-epsilon*)
    fix *e* :: *real* **assume** $0 < e$
    from *approx*[*OF this*] **guess** *i* **..**
    then have $x \leq f\ i + e$
      by *simp*
    also have $\ldots \leq (SUP\ i{:}A.\ f\ i) + e$
      by (*intro add-mono* ‹$i \in A$› *SUP-upper order-refl*)
    finally show $x \leq (SUP\ i{:}A.\ f\ i) + e$ **.**
  qed
qed (*intro SUP-least f-bound*)

lemma *ennreal-approx-INF*:
  fixes *x*::*ennreal*
  assumes *f-bound*: $\bigwedge i.\ i \in A \Longrightarrow x \leq f\ i$
  assumes *approx*: $\bigwedge e.\ (e{::}real) > 0 \Longrightarrow \exists\, i \in A.\ f\ i \leq x + e$
  shows $x = (INF\ i : A.\ f\ i)$
proof (*rule antisym*)
  show $(INF\ i{:}A.\ f\ i) \leq x$
  proof (*rule ennreal-le-epsilon*)
    fix *e* :: *real* **assume** $0 < e$
    from *approx*[*OF this*] **guess** *i* **.. note** *i* = *this*
    then have $(INF\ i{:}A.\ f\ i) \leq f\ i$
      by (*intro INF-lower*)
    also have $\ldots \leq x + e$
      by *fact*
    finally show $(INF\ i{:}A.\ f\ i) \leq x + e$ **.**

qed
qed (*intro INF-greatest f-bound*)

lemma *ennreal-approx-unit*:
$(\bigwedge a::ennreal.\ 0 < a \Longrightarrow a < 1 \Longrightarrow a * z \leq y) \Longrightarrow z \leq y$
apply (*subst SUP-mult-right-ennreal*[*of* $\lambda x.\ x$ $\{0 <..< 1\}$ *z*, *simplified*])
apply (*rule SUP-least*)
apply *auto*
done

lemma *suminf-ennreal2*:
$(\bigwedge i.\ 0 \leq f\ i) \Longrightarrow summable\ f \Longrightarrow (\sum i.\ ennreal\ (f\ i)) = ennreal\ (\sum i.\ f\ i)$
using *suminf-ennreal-eq* **by** *blast*

lemma *less-top-ennreal*: $x < top \longleftrightarrow (\exists r \geq 0.\ x = ennreal\ r)$
by (*cases x*) *auto*

lemma *tendsto-top-iff-ennreal*:
fixes $f :: 'a \Rightarrow ennreal$
shows $(f \longrightarrow top)\ F \longleftrightarrow (\forall l \geq 0.\ eventually\ (\lambda x.\ ennreal\ l < f\ x)\ F)$
by (*auto simp*: *less-top-ennreal order-tendsto-iff*)

lemma *ennreal-tendsto-top-eq-at-top*:
$((\lambda z.\ ennreal\ (f\ z)) \longrightarrow top)\ F \longleftrightarrow (LIM\ z\ F.\ f\ z :> at\text{-}top)$
unfolding *filterlim-at-top-dense tendsto-top-iff-ennreal*
apply (*auto simp*: *ennreal-less-iff*)
subgoal for *y*
by (*auto elim*!: *eventually-mono allE*[*of - max 0 y*])
done

lemma *tendsto-0-if-Limsup-eq-0-ennreal*:
fixes $f :: - \Rightarrow ennreal$
shows $Limsup\ F\ f = 0 \Longrightarrow (f \longrightarrow 0)\ F$
using *Liminf-le-Limsup*[*of F f*] *tendsto-iff-Liminf-eq-Limsup*[*of F f 0*]
by (*cases F = bot*) *auto*

lemma *diff-le-self-ennreal*[*simp*]: $a - b \leq (a::ennreal)$
by (*cases a b rule*: *ennreal2-cases*) (*auto simp*: *ennreal-minus*)

lemma *ennreal-ineq-diff-add*: $b \leq a \Longrightarrow a = b + (a - b::ennreal)$
by *transfer* (*auto simp*: *ereal-diff-positive max.absorb2 ereal-ineq-diff-add*)

lemma *ennreal-mult-strict-left-mono*: $(a::ennreal) < c \Longrightarrow 0 < b \Longrightarrow b < top \Longrightarrow b * a < b * c$
by *transfer* (*auto intro*!: *ereal-mult-strict-left-mono*)

lemma *ennreal-between*: $0 < e \Longrightarrow 0 < x \Longrightarrow x < top \Longrightarrow x - e < (x::ennreal)$
by *transfer* (*auto intro*!: *ereal-between*)

lemma *minus-less-iff-ennreal*: $b < top \Longrightarrow b \leq a \Longrightarrow a - b < c \longleftrightarrow a < c + (b{::}ennreal)$
  by *transfer*
    (*auto simp*: *top-ereal-def ereal-minus-less le-less*)

lemma *tendsto-zero-ennreal*:
  assumes *ev*: $\bigwedge r.\ 0 < r \Longrightarrow \forall_F\ x\ in\ F.\ f\ x < ennreal\ r$
  shows $(f \longrightarrow 0)\ F$
proof (*rule order-tendstoI*)
  fix *e*::*ennreal* **assume** $e > 0$
  obtain *e'*::*real* **where** $e' > 0$ $ennreal\ e' < e$
    using ‹$0 < e$› *dense*[*of 0 if e = top then 1 else* (*enn2real e*)]
    by (*cases e*) (*auto simp*: *ennreal-less-iff*)
  from *ev*[*OF* ‹$e' > 0$›] **show** $\forall_F\ x\ in\ F.\ f\ x < e$
    by *eventually-elim* (*insert* ‹*ennreal e' < e*›, *auto*)
qed *simp*

lifting-update *ennreal.lifting*
lifting-forget *ennreal.lifting*

end

35 A generic phantom type

theory *Phantom-Type*
imports *Main*
begin

datatype (*'a*, *'b*) *phantom* = *phantom* (*of-phantom*: *'b*)

lemma *type-definition-phantom'*: *type-definition of-phantom phantom UNIV*
by(*unfold-locales*) *simp-all*

lemma *phantom-comp-of-phantom* [*simp*]: *phantom* ∘ *of-phantom* = *id*
  and *of-phantom-comp-phantom* [*simp*]: *of-phantom* ∘ *phantom* = *id*
by(*simp-all add*: *o-def id-def*)

syntax *-Phantom* :: *type* ⇒ *logic* ((*1Phantom*/(*1'*(-'))))
translations
  Phantom(*'t*) => *CONST phantom* :: - ⇒ (*'t*, -) *phantom*

typed-print-translation ‹
  let
    fun phantom-tr' ctxt (*Type* (@{*type-name fun*}, [-, *Type* (@{*type-name phantom*}, [*T*, -])])) *ts* =
        list-comb
          (*Syntax.const* @{*syntax-const -Phantom*} $ *Syntax-Phases.term-of-typ ctxt T*, *ts*)
      | *phantom-tr'* - - - = *raise Match*;

in [(@{*const-syntax phantom*}, *phantom-tr'*)] *end*
⟩

lemma *of-phantom-inject* [*simp*]:
of-phantom x = *of-phantom y* ⟷ *x* = *y*
by(*cases x y rule*: *phantom.exhaust*[*case-product phantom.exhaust*]) *simp*

end

36 Cardinality of types

theory *Cardinality*
imports *Phantom-Type*
begin

36.1 Preliminary lemmas

lemma (**in** *type-definition*) *univ*:
UNIV = *Abs* ' *A*
proof
show *Abs* ' *A* ⊆ *UNIV* **by** (*rule subset-UNIV*)
show *UNIV* ⊆ *Abs* ' *A*
proof
fix *x* :: *'b*
have *x* = *Abs* (*Rep x*) **by** (*rule Rep-inverse* [*symmetric*])
moreover have *Rep x* ∈ *A* **by** (*rule Rep*)
ultimately show *x* ∈ *Abs* ' *A* **by** (*rule image-eqI*)
qed
qed

lemma (**in** *type-definition*) *card*: *card* (*UNIV* :: *'b set*) = *card A*
by (*simp add*: *univ card-image inj-on-def Abs-inject*)

lemma *finite-range-Some*: *finite* (*range* (*Some* :: *'a* ⇒ *'a option*)) = *finite* (*UNIV* :: *'a set*)
by(*auto dest*: *finite-imageD intro*: *inj-Some*)

lemma *infinite-literal*: ¬ *finite* (*UNIV* :: *String.literal set*)
proof −
have *inj STR* **by**(*auto intro*: *injI*)
thus *?thesis*
by(*auto simp add*: *type-definition.univ*[*OF type-definition-literal*] *infinite-UNIV-listI dest*: *finite-imageD*)
qed

36.2 Cardinalities of types

syntax *-type-card* :: *type* => *nat* ((*1CARD*/(*1'*(-'))))

translations *CARD('t) => CONST card (CONST UNIV :: 't set)*

print-translation ⟨
let
fun card-univ-tr' ctxt [Const (@{const-syntax UNIV}, Type (-, [T]))] =
Syntax.const @{syntax-const -type-card} $ Syntax-Phases.term-of-typ ctxt T
in [(@{const-syntax card}, card-univ-tr')] end
⟩

lemma *card-prod* [*simp*]: *CARD('a × 'b) = CARD('a) ∗ CARD('b)*
unfolding *UNIV-Times-UNIV* [*symmetric*] **by** (*simp only*: *card-cartesian-product*)

lemma *card-UNIV-sum*: *CARD('a + 'b) = (if CARD('a) ≠ 0 ∧ CARD('b) ≠ 0 then CARD('a) + CARD('b) else 0)*
unfolding *UNIV-Plus-UNIV*[*symmetric*]
by(*auto simp add*: *card-eq-0-iff card-Plus simp del*: *UNIV-Plus-UNIV*)

lemma *card-sum* [*simp*]: *CARD('a + 'b) = CARD('a::finite) + CARD('b::finite)*
by(*simp add*: *card-UNIV-sum*)

lemma *card-UNIV-option*: *CARD('a option) = (if CARD('a) = 0 then 0 else CARD('a) + 1)*
proof −
have (*None :: 'a option*) ∉ *range Some* **by** *clarsimp*
thus *?thesis*
by (*simp add*: *UNIV-option-conv card-eq-0-iff finite-range-Some card-image*)
qed

lemma *card-option* [*simp*]: *CARD('a option) = Suc CARD('a::finite)*
by(*simp add*: *card-UNIV-option*)

lemma *card-UNIV-set*: *CARD('a set) = (if CARD('a) = 0 then 0 else 2 ^ CARD('a))*
by(*simp add*: *Pow-UNIV*[*symmetric*] *card-eq-0-iff card-Pow del*: *Pow-UNIV*)

lemma *card-set* [*simp*]: *CARD('a set) = 2 ^ CARD('a::finite)*
by(*simp add*: *card-UNIV-set*)

lemma *card-nat* [*simp*]: *CARD(nat) = 0*
by (*simp add*: *card-eq-0-iff*)

lemma *card-fun*: *CARD('a ⇒ 'b) = (if CARD('a) ≠ 0 ∧ CARD('b) ≠ 0 ∨ CARD('b) = 1 then CARD('b) ^ CARD('a) else 0)*
proof −
{ **assume** *0 < CARD('a)* **and** *0 < CARD('b)*
hence *fina*: *finite (UNIV :: 'a set)* **and** *finb*: *finite (UNIV :: 'b set)*
by(*simp-all only*: *card-ge-0-finite*)
from *finite-distinct-list*[*OF finb*] **obtain** *bs*
where *bs*: *set bs = (UNIV :: 'b set)* **and** *distb*: *distinct bs* **by** *blast*
from *finite-distinct-list*[*OF fina*] **obtain** *as*

where *as*: *set as* = (*UNIV* :: *'a set*) **and** *dista*: *distinct as* **by** *blast*
have *cb*: *CARD('b)* = *length bs*
unfolding *bs*[*symmetric*] *distinct-card*[*OF distb*] **..**
have *ca*: *CARD('a)* = *length as*
unfolding *as*[*symmetric*] *distinct-card*[*OF dista*] **..**
let *?xs* = *map* (λ*ys*. *the o map-of* (*zip as ys*)) (*List.n-lists* (*length as*) *bs*)
have *UNIV* = *set ?xs*
proof(*rule UNIV-eq-I*)
fix *f* :: *'a* ⇒ *'b*
from *as* **have** *f* = *the* ∘ *map-of* (*zip as* (*map f as*))
by(*auto simp add*: *map-of-zip-map*)
thus *f* ∈ *set ?xs* **using** *bs* **by**(*auto simp add*: *set-n-lists*)
qed
moreover have *distinct ?xs* **unfolding** *distinct-map*
proof(*intro conjI distinct-n-lists distb inj-onI*)
fix *xs ys* :: *'b list*
assume *xs*: *xs* ∈ *set* (*List.n-lists* (*length as*) *bs*)
and *ys*: *ys* ∈ *set* (*List.n-lists* (*length as*) *bs*)
and *eq*: *the* ∘ *map-of* (*zip as xs*) = *the* ∘ *map-of* (*zip as ys*)
from *xs ys* **have** [*simp*]: *length xs* = *length as length ys* = *length as*
by(*simp-all add*: *length-n-lists-elem*)
have *map-of* (*zip as xs*) = *map-of* (*zip as ys*)
proof
fix *x*
from *as bs* **have** ∃ *y*. *map-of* (*zip as xs*) *x* = *Some y* ∃ *y*. *map-of* (*zip as ys*) *x* = *Some y*
by(*simp-all add*: *map-of-zip-is-Some*[*symmetric*])
with *eq* **show** *map-of* (*zip as xs*) *x* = *map-of* (*zip as ys*) *x*
by(*auto dest*: *fun-cong*[**where** *x*=*x*])
qed
with *dista* **show** *xs* = *ys* **by**(*simp add*: *map-of-zip-inject*)
qed
hence *card* (*set ?xs*) = *length ?xs* **by**(*simp only*: *distinct-card*)
moreover have *length ?xs* = *length bs* ˆ *length as* **by**(*simp add*: *length-n-lists*)
ultimately have *CARD('a ⇒ 'b)* = *CARD('b)* ˆ *CARD('a)* **using** *cb ca* **by** *simp* **}**
moreover {
assume *cb*: *CARD('b)* = *1*
then obtain *b* **where** *b*: *UNIV* = {*b* :: *'b*} **by**(*auto simp add*: *card-Suc-eq*)
have *eq*: *UNIV* = {λ*x* :: *'a*. *b* ::*'b*}
proof(*rule UNIV-eq-I*)
fix *x* :: *'a* ⇒ *'b*
{ fix *y*
have *x y* ∈ *UNIV* **..**
hence *x y* = *b* **unfolding** *b* **by** *simp* **}**
thus *x* ∈ {λ*x*. *b*} **by**(*auto*)
qed
have *CARD('a ⇒ 'b)* = *1* **unfolding** *eq* **by** *simp* **}**
ultimately show *?thesis*

by(*auto simp del*: *One-nat-def*)(*auto simp add*: *card-eq-0-iff dest*: *finite-fun-UNIVD2 finite-fun-UNIVD1*)
qed

corollary *finite-UNIV-fun*:
finite (*UNIV* :: ('*a* ⇒ '*b*) *set*) ⟷
finite (*UNIV* :: '*a set*) ∧ *finite* (*UNIV* :: '*b set*) ∨ *CARD*('*b*) = *1*
(**is** *?lhs* ⟷ *?rhs*)
proof −
have *?lhs* ⟷ *CARD*('*a* ⇒ '*b*) > *0* **by**(*simp add*: *card-gt-0-iff*)
also have ... ⟷ *CARD*('*a*) > *0* ∧ *CARD*('*b*) > *0* ∨ *CARD*('*b*) = *1*
by(*simp add*: *card-fun*)
also have ... = *?rhs* **by**(*simp add*: *card-gt-0-iff*)
finally show *?thesis* **.**
qed

lemma *card-literal*: *CARD*(*String.literal*) = *0*
by(*simp add*: *card-eq-0-iff infinite-literal*)

36.3 Classes with at least 1 and 2

Class finite already captures "at least 1"

lemma *zero-less-card-finite* [*simp*]: *0* < *CARD*('*a*::*finite*)
unfolding *neq0-conv* [*symmetric*] **by** *simp*

lemma *one-le-card-finite* [*simp*]: *Suc 0* ≤ *CARD*('*a*::*finite*)
by (*simp add*: *less-Suc-eq-le* [*symmetric*])

Class for cardinality "at least 2"

class *card2* = *finite* +
assumes *two-le-card*: *2* ≤ *CARD*('*a*)

lemma *one-less-card*: *Suc 0* < *CARD*('*a*::*card2*)
using *two-le-card* [**where** '*a*='*a*] **by** *simp*

lemma *one-less-int-card*: *1* < *int CARD*('*a*::*card2*)
using *one-less-card* [**where** '*a*='*a*] **by** *simp*

36.4 A type class for deciding finiteness of types

type-synonym '*a finite-UNIV* = ('*a*, *bool*) *phantom*

class *finite-UNIV* =
fixes *finite-UNIV* :: ('*a*, *bool*) *phantom*
assumes *finite-UNIV*: *finite-UNIV* = *Phantom*('*a*) (*finite* (*UNIV* :: '*a set*))

lemma *finite-UNIV-code* [*code-unfold*]:
finite (*UNIV* :: '*a* :: *finite-UNIV set*)
⟷ *of-phantom* (*finite-UNIV* :: '*a finite-UNIV*)

by(*simp add*: *finite-UNIV*)

36.5 A type class for computing the cardinality of types

definition *is-list-UNIV* :: *'a list* ⇒ *bool*
where *is-list-UNIV xs* = (*let c* = *CARD*(*'a*) *in if c* = *0 then False else size* (*remdups xs*) = *c*)

lemma *is-list-UNIV-iff*: *is-list-UNIV xs* ⟷ *set xs* = *UNIV*
by(*auto simp add*: *is-list-UNIV-def Let-def card-eq-0-iff List.card-set*[*symmetric*]
dest: *subst*[**where** *P*=*finite*, *OF* - *finite-set*] *card-eq-UNIV-imp-eq-UNIV*)

type-synonym *'a card-UNIV* = (*'a*, *nat*) *phantom*

class *card-UNIV* = *finite-UNIV* +
fixes *card-UNIV* :: *'a card-UNIV*
assumes *card-UNIV*: *card-UNIV* = *Phantom*(*'a*) *CARD*(*'a*)

36.6 Instantiations for *card-UNIV*

instantiation *nat* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*nat*) *False*
definition *card-UNIV* = *Phantom*(*nat*) *0*
instance by *intro-classes* (*simp-all add*: *finite-UNIV-nat-def card-UNIV-nat-def*)
end

instantiation *int* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*int*) *False*
definition *card-UNIV* = *Phantom*(*int*) *0*
instance by *intro-classes* (*simp-all add*: *card-UNIV-int-def finite-UNIV-int-def infinite-UNIV-int*)
end

instantiation *natural* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*natural*) *False*
definition *card-UNIV* = *Phantom*(*natural*) *0*
instance
by *standard*
(*auto simp add*: *finite-UNIV-natural-def card-UNIV-natural-def card-eq-0-iff*
type-definition.univ [*OF type-definition-natural*] *natural-eq-iff*
dest!: *finite-imageD intro*: *inj-onI*)
end

instantiation *integer* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*integer*) *False*
definition *card-UNIV* = *Phantom*(*integer*) *0*
instance
by *standard*
(*auto simp add*: *finite-UNIV-integer-def card-UNIV-integer-def card-eq-0-iff*
type-definition.univ [*OF type-definition-integer*] *infinite-UNIV-int*

dest!: *finite-imageD intro*: *inj-onI*)
end

instantiation *list* :: (*type*) *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*'a list*) *False*
definition *card-UNIV* = *Phantom*(*'a list*) *0*
instance by *intro-classes* (*simp-all add*: *card-UNIV-list-def finite-UNIV-list-def infinite-UNIV-listI*)
end

instantiation *unit* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*unit*) *True*
definition *card-UNIV* = *Phantom*(*unit*) *1*
instance by *intro-classes* (*simp-all add*: *card-UNIV-unit-def finite-UNIV-unit-def*)
end

instantiation *bool* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*bool*) *True*
definition *card-UNIV* = *Phantom*(*bool*) *2*
instance by(*intro-classes*)(*simp-all add*: *card-UNIV-bool-def finite-UNIV-bool-def*)
end

instantiation *char* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*char*) *True*
definition *card-UNIV* = *Phantom*(*char*) *256*
instance by *intro-classes* (*simp-all add*: *card-UNIV-char-def card-UNIV-char finite-UNIV-char-def*)
end

instantiation *prod* :: (*finite-UNIV*, *finite-UNIV*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*'a* × *'b*)
(*of-phantom* (*finite-UNIV* :: *'a finite-UNIV*) ∧ *of-phantom* (*finite-UNIV* :: *'b finite-UNIV*))
instance by *intro-classes* (*simp add*: *finite-UNIV-prod-def finite-UNIV finite-prod*)
end

instantiation *prod* :: (*card-UNIV*, *card-UNIV*) *card-UNIV* **begin**
definition *card-UNIV* = *Phantom*(*'a* × *'b*)
(*of-phantom* (*card-UNIV* :: *'a card-UNIV*) ∗ *of-phantom* (*card-UNIV* :: *'b card-UNIV*))
instance by *intro-classes* (*simp add*: *card-UNIV-prod-def card-UNIV*)
end

instantiation *sum* :: (*finite-UNIV*, *finite-UNIV*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*'a* + *'b*)
(*of-phantom* (*finite-UNIV* :: *'a finite-UNIV*) ∧ *of-phantom* (*finite-UNIV* :: *'b finite-UNIV*))
instance
by *intro-classes* (*simp add*: *UNIV-Plus-UNIV*[*symmetric*] *finite-UNIV-sum-def finite-UNIV del*: *UNIV-Plus-UNIV*)
end

instantiation *sum* :: (*card-UNIV*, *card-UNIV*) *card-UNIV* **begin**
definition *card-UNIV* = *Phantom*('*a* + '*b*)
(*let ca* = *of-phantom* (*card-UNIV* :: '*a card-UNIV*);
cb = *of-phantom* (*card-UNIV* :: '*b card-UNIV*)
in if ca ≠ *0* ∧ *cb* ≠ *0 then ca* + *cb else 0*)
instance by *intro-classes* (*auto simp add*: *card-UNIV-sum-def card-UNIV card-UNIV-sum*)
end

instantiation *fun* :: (*finite-UNIV*, *card-UNIV*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*('*a* ⇒ '*b*)
(*let cb* = *of-phantom* (*card-UNIV* :: '*b card-UNIV*)
in cb = *1* ∨ *of-phantom* (*finite-UNIV* :: '*a finite-UNIV*) ∧ *cb* ≠ *0*)
instance
by *intro-classes* (*auto simp add*: *finite-UNIV-fun-def Let-def card-UNIV finite-UNIV finite-UNIV-fun card-gt-0-iff*)
end

instantiation *fun* :: (*card-UNIV*, *card-UNIV*) *card-UNIV* **begin**
definition *card-UNIV* = *Phantom*('*a* ⇒ '*b*)
(*let ca* = *of-phantom* (*card-UNIV* :: '*a card-UNIV*);
cb = *of-phantom* (*card-UNIV* :: '*b card-UNIV*)
in if ca ≠ *0* ∧ *cb* ≠ *0* ∨ *cb* = *1 then cb* ^ *ca else 0*)
instance by *intro-classes* (*simp add*: *card-UNIV-fun-def card-UNIV Let-def card-fun*)
end

instantiation *option* :: (*finite-UNIV*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*('*a option*) (*of-phantom* (*finite-UNIV* :: '*a finite-UNIV*))
instance by *intro-classes* (*simp add*: *finite-UNIV-option-def finite-UNIV*)
end

instantiation *option* :: (*card-UNIV*) *card-UNIV* **begin**
definition *card-UNIV* = *Phantom*('*a option*)
(*let c* = *of-phantom* (*card-UNIV* :: '*a card-UNIV*) *in if c* ≠ *0 then Suc c else 0*)
instance by *intro-classes* (*simp add*: *card-UNIV-option-def card-UNIV card-UNIV-option*)
end

instantiation *String.literal* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*String.literal*) *False*
definition *card-UNIV* = *Phantom*(*String.literal*) *0*
instance
by *intro-classes* (*simp-all add*: *card-UNIV-literal-def finite-UNIV-literal-def infinite-literal card-literal*)
end

instantiation *set* :: (*finite-UNIV*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*('*a set*) (*of-phantom* (*finite-UNIV* :: '*a finite-UNIV*))
instance by *intro-classes* (*simp add*: *finite-UNIV-set-def finite-UNIV Finite-Set.finite-set*)

end

instantiation *set* :: (*card-UNIV*) *card-UNIV* **begin**
definition *card-UNIV* = *Phantom*(*'a set*)
(*let c* = *of-phantom* (*card-UNIV* :: *'a card-UNIV*) *in if c* = *0 then 0 else 2* ^ *c*)
instance by *intro-classes* (*simp add*: *card-UNIV-set-def card-UNIV-set card-UNIV*)
end

lemma *UNIV-finite-1*: *UNIV* = *set* [*finite-1*.a_1]
by(*auto intro*: *finite-1*.*exhaust*)

lemma *UNIV-finite-2*: *UNIV* = *set* [*finite-2*.a_1, *finite-2*.a_2]
by(*auto intro*: *finite-2*.*exhaust*)

lemma *UNIV-finite-3*: *UNIV* = *set* [*finite-3*.a_1, *finite-3*.a_2, *finite-3*.a_3]
by(*auto intro*: *finite-3*.*exhaust*)

lemma *UNIV-finite-4*: *UNIV* = *set* [*finite-4*.a_1, *finite-4*.a_2, *finite-4*.a_3, *finite-4*.a_4]
by(*auto intro*: *finite-4*.*exhaust*)

lemma *UNIV-finite-5*:
UNIV = *set* [*finite-5*.a_1, *finite-5*.a_2, *finite-5*.a_3, *finite-5*.a_4, *finite-5*.a_5]
by(*auto intro*: *finite-5*.*exhaust*)

instantiation *Enum.finite-1* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*Enum.finite-1*) *True*
definition *card-UNIV* = *Phantom*(*Enum.finite-1*) *1*
instance
by *intro-classes* (*simp-all add*: *UNIV-finite-1 card-UNIV-finite-1-def finite-UNIV-finite-1-def*)
end

instantiation *Enum.finite-2* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*Enum.finite-2*) *True*
definition *card-UNIV* = *Phantom*(*Enum.finite-2*) *2*
instance
by *intro-classes* (*simp-all add*: *UNIV-finite-2 card-UNIV-finite-2-def finite-UNIV-finite-2-def*)
end

instantiation *Enum.finite-3* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*Enum.finite-3*) *True*
definition *card-UNIV* = *Phantom*(*Enum.finite-3*) *3*
instance
by *intro-classes* (*simp-all add*: *UNIV-finite-3 card-UNIV-finite-3-def finite-UNIV-finite-3-def*)
end

instantiation *Enum.finite-4* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*Enum.finite-4*) *True*
definition *card-UNIV* = *Phantom*(*Enum.finite-4*) *4*
instance

by *intro-classes* (*simp-all add*: *UNIV-finite-4 card-UNIV-finite-4-def finite-UNIV-finite-4-def*)
end

instantiation *Enum.finite-5* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*Enum.finite-5*) *True*
definition *card-UNIV* = *Phantom*(*Enum.finite-5*) *5*
instance
by *intro-classes* (*simp-all add*: *UNIV-finite-5 card-UNIV-finite-5-def finite-UNIV-finite-5-def*)
end

36.7 Code setup for sets

Implement *CARD*(*'a*) via *card-UNIV-class.card-UNIV* and provide implementations for *finite*, *card*, *op* ⊆, and *op* =if the calling context already provides *finite-UNIV* and *card-UNIV* instances. If we implemented the latter always via *card-UNIV-class.card-UNIV*, we would require instances of essentially all element types, i.e., a lot of instantiation proofs and – at run time – possibly slow dictionary constructions.

context
begin

qualified definition *card-UNIV'* :: *'a card-UNIV*
where [*code del*]: *card-UNIV'* = *Phantom*(*'a*) *CARD*(*'a*)

lemma *CARD-code* [*code-unfold*]:
CARD(*'a*) = *of-phantom* (*card-UNIV'* :: *'a card-UNIV*)
by(*simp add*: *card-UNIV'-def*)

lemma *card-UNIV'-code* [*code*]:
card-UNIV' = *card-UNIV*
by(*simp add*: *card-UNIV card-UNIV'-def*)

end

lemma *card-Compl*:
finite A ⟹ *card* (− *A*) = *card* (*UNIV* :: *'a set*) − *card* (*A* :: *'a set*)
by (*metis Compl-eq-Diff-UNIV card-Diff-subset top-greatest*)

context fixes *xs* :: *'a* :: *finite-UNIV list*
begin

qualified definition *finite'* :: *'a set* ⇒ *bool*
where [*simp*, *code del*, *code-abbrev*]: *finite'* = *finite*

lemma *finite'-code* [*code*]:
finite' (*set xs*) ⟷ *True*
finite' (*List.coset xs*) ⟷ *of-phantom* (*finite-UNIV* :: *'a finite-UNIV*)
by(*simp-all add*: *card-gt-0-iff finite-UNIV*)

end

context fixes *xs* :: *'a* :: *card-UNIV list*
begin

qualified definition *card'* :: *'a set* ⇒ *nat*
where [*simp*, *code del*, *code-abbrev*]: *card'* = *card*

lemma *card'-code* [*code*]:
  card' (*set xs*) = *length* (*remdups xs*)
  card' (*List.coset xs*) = *of-phantom* (*card-UNIV* :: *'a card-UNIV*) − *length* (*remdups xs*)
by(*simp-all add*: *List.card-set card-Compl card-UNIV*)

qualified definition *subset'* :: *'a set* ⇒ *'a set* ⇒ *bool*
where [*simp*, *code del*, *code-abbrev*]: *subset'* = *op* ⊆

lemma *subset'-code* [*code*]:
  subset' *A* (*List.coset ys*) ⟷ (∀ *y* ∈ *set ys*. *y* ∉ *A*)
  subset' (*set ys*) *B* ⟷ (∀ *y* ∈ *set ys*. *y* ∈ *B*)
  subset' (*List.coset xs*) (*set ys*) ⟷ (*let n* = *CARD*(*'a*) *in n* > *0* ∧ *card*(*set* (*xs* @ *ys*)) = *n*)
by(*auto simp add*: *Let-def card-gt-0-iff dest*: *card-eq-UNIV-imp-eq-UNIV intro*: *arg-cong*[**where** *f*=*card*])
  (*metis finite-compl finite-set rev-finite-subset*)

qualified definition *eq-set* :: *'a set* ⇒ *'a set* ⇒ *bool*
where [*simp*, *code del*, *code-abbrev*]: *eq-set* = *op* =

lemma *eq-set-code* [*code*]:
  fixes *ys*
  defines *rhs* ≡
  let n = *CARD*(*'a*)
  in if n = *0 then False else*
    let xs' = *remdups xs*; *ys'* = *remdups ys*
    in length xs' + *length ys'* = *n* ∧ (∀ *x* ∈ *set xs'*. *x* ∉ *set ys'*) ∧ (∀ *y* ∈ *set ys'*. *y* ∉ *set xs'*)
  shows *eq-set* (*List.coset xs*) (*set ys*) ⟷ *rhs*
  and *eq-set* (*set ys*) (*List.coset xs*) ⟷ *rhs*
  and *eq-set* (*set xs*) (*set ys*) ⟷ (∀ *x* ∈ *set xs*. *x* ∈ *set ys*) ∧ (∀ *y* ∈ *set ys*. *y* ∈ *set xs*)
  and *eq-set* (*List.coset xs*) (*List.coset ys*) ⟷ (∀ *x* ∈ *set xs*. *x* ∈ *set ys*) ∧ (∀ *y* ∈ *set ys*. *y* ∈ *set xs*)
proof *goal-cases*
  {
    case *1*
    show *?case* (**is** *?lhs* ⟷ *?rhs*)

```
proof
  show ?rhs if ?lhs
    using that
    by (auto simp add: rhs-def Let-def List.card-set[symmetric]
      card-Un-Int[where A=set xs and B=− set xs] card-UNIV
      Compl-partition card-gt-0-iff dest: sym)(metis finite-compl finite-set)
  show ?lhs if ?rhs
  proof −
    have ⟦ ∀ y∈set xs. y ∉ set ys; ∀ x∈set ys. x ∉ set xs ⟧ ⟹ set xs ∩ set ys
= {} by blast
    with that show ?thesis
      by (auto simp add: rhs-def Let-def List.card-set[symmetric]
        card-UNIV card-gt-0-iff card-Un-Int[where A=set xs and B=set ys]
        dest: card-eq-UNIV-imp-eq-UNIV split: if-split-asm)
  qed
qed
}
moreover
case 2
ultimately show ?case unfolding eq-set-def by blast
next
case 3
show ?case unfolding eq-set-def List.coset-def by blast
next
case 4
show ?case unfolding eq-set-def List.coset-def by blast
qed

end
```

Provide more informative exceptions than Match for non-rewritten cases. If generated code raises one these exceptions, then a code equation calls the mentioned operator for an element type that is not an instance of *card-UNIV* and is therefore not implemented via *card-UNIV-class.card-UNIV*. Constrain the element type with sort *card-UNIV* to change this.

```
lemma card-coset-error [code]:
  card (List.coset xs) =
   Code.abort (STR ''card (List.coset -) requires type class instance card-UNIV'')
    (λ-. card (List.coset xs))
by(simp)

lemma coset-subseteq-set-code [code]:
  List.coset xs ⊆ set ys ⟷
  (if xs = [] ∧ ys = [] then False
   else Code.abort
   (STR ''subset-eq (List.coset -) (List.set -) requires type class instance card-UNIV'')
    (λ-. List.coset xs ⊆ set ys))
by simp
```

notepad begin — test code setup
have *List.coset* [*True*] = *set* [*False*] ∧
List.coset [] ⊆ *List.set* [*True*, *False*] ∧
finite (*List.coset* [*True*])
by *eval*
end

end

37 Almost everywhere constant functions

theory *FinFun*
imports *Cardinality*
begin

This theory defines functions which are constant except for finitely many points (FinFun) and introduces a type finfin along with a number of operators for them. The code generator is set up such that such functions can be represented as data in the generated code and all operators are executable.

For details, see Formalising FinFuns - Generating Code for Functions as Data by A. Lochbihler in TPHOLs 2009.

37.1 The *map-default* operation

definition *map-default* :: $'b \Rightarrow ('a \rightharpoonup 'b) \Rightarrow 'a \Rightarrow 'b$
where *map-default b f a* ≡ *case f a of None* ⇒ *b* | *Some b′* ⇒ *b′*

lemma *map-default-delete* [*simp*]:
map-default b (*f*(*a* := *None*)) = (*map-default b f*)(*a* := *b*)
by(*simp add*: *map-default-def fun-eq-iff*)

lemma *map-default-insert*:
map-default b (*f*(*a* ↦ *b′*)) = (*map-default b f*)(*a* := *b′*)
by(*simp add*: *map-default-def fun-eq-iff*)

lemma *map-default-empty* [*simp*]: *map-default b empty* = (λ*a*. *b*)
by(*simp add*: *fun-eq-iff map-default-def*)

lemma *map-default-inject*:
fixes *g g′* :: $'a \rightharpoonup 'b$
assumes *infin-eq*: ¬ *finite* (*UNIV* :: *'a set*) ∨ *b* = *b′*
and *fin*: *finite* (*dom g*) **and** *b*: *b* ∉ *ran g*
and *fin′*: *finite* (*dom g′*) **and** *b′*: *b′* ∉ *ran g′*
and *eq′*: *map-default b g* = *map-default b′ g′*
shows *b* = *b′ g* = *g′*
proof −
from *infin-eq* **show** *bb′*: *b* = *b′*
proof

assume *infin*: ¬ *finite* (*UNIV* :: *'a set*)
from *fin fin'* **have** *finite* (*dom g* ∪ *dom g'*) **by** *auto*
with *infin* **have** *UNIV* − (*dom g* ∪ *dom g'*) ≠ {} **by**(*auto dest*: *finite-subset*)
then obtain *a* **where** *a*: *a* ∉ *dom g* ∪ *dom g'* **by** *auto*
hence *map-default b g a* = *b map-default b' g' a* = *b'* **by**(*auto simp add*: *map-default-def*)
with *eq'* **show** *b* = *b'* **by** *simp*
qed

show *g* = *g'*
proof
fix *x*
show *g x* = *g' x*
proof(*cases g x*)
case *None*
hence *map-default b g x* = *b* **by**(*simp add*: *map-default-def*)
with *bb' eq'* **have** *map-default b' g' x* = *b'* **by** *simp*
with *b'* **have** *g' x* = *None* **by**(*simp add*: *map-default-def ran-def split*: *option.split-asm*)
with *None* **show** *?thesis* **by** *simp*
next
case (*Some c*)
with *b* **have** *cb*: *c* ≠ *b* **by**(*auto simp add*: *ran-def*)
moreover from *Some* **have** *map-default b g x* = *c* **by**(*simp add*: *map-default-def*)
with *eq'* **have** *map-default b' g' x* = *c* **by** *simp*
ultimately have *g' x* = *Some c* **using** *b' bb'* **by**(*auto simp add*: *map-default-def split*: *option.splits*)
with *Some* **show** *?thesis* **by** *simp*
qed
qed
qed

37.2 The finfun type

definition *finfun* = {*f*::*'a*⇒*'b*. ∃ *b*. *finite* {*a*. *f a* ≠ *b*}}

typedef (*'a*,*'b*) *finfun* ((- ⇒*f* /-) [*22*, *21*] *21*) = *finfun* :: (*'a* => *'b*) *set*
morphisms *finfun-apply Abs-finfun*
proof −
have ∃*f*. *finite* {*x*. *f x* ≠ *undefined*}
proof
show *finite* {*x*. (λ*y*. *undefined*) *x* ≠ *undefined*} **by** *auto*
qed
then show *?thesis* **unfolding** *finfun-def* **by** *auto*
qed

type-notation *finfun* ((- ⇒*f* /-) [*22*, *21*] *21*)

setup-lifting *type-definition-finfun*

lemma *fun-upd-finfun*: $y(a := b) \in finfun \longleftrightarrow y \in finfun$
proof −
  { fix b'
    have $finite\ \{a'.\ (y(a := b))\ a' \neq b'\} = finite\ \{a'.\ y\ a' \neq b'\}$
    proof(*cases* $b = b'$)
      case *True*
      hence $\{a'.\ (y(a := b))\ a' \neq b'\} = \{a'.\ y\ a' \neq b'\} - \{a\}$ **by** *auto*
      thus *?thesis* **by** *simp*
    next
      case *False*
      hence $\{a'.\ (y(a := b))\ a' \neq b'\} = insert\ a\ \{a'.\ y\ a' \neq b'\}$ **by** *auto*
      thus *?thesis* **by** *simp*
    qed }
  thus *?thesis* **unfolding** *finfun-def* **by** *blast*
qed

lemma *const-finfun*: $(\lambda x.\ a) \in finfun$
by(*auto simp add*: *finfun-def*)

lemma *finfun-left-compose*:
  assumes $y \in finfun$
  shows $g \circ y \in finfun$
proof −
  from *assms* **obtain** b **where** $finite\ \{a.\ y\ a \neq b\}$
    unfolding *finfun-def* **by** *blast*
  hence $finite\ \{c.\ g\ (y\ c) \neq g\ b\}$
  proof(*induct* $\{a.\ y\ a \neq b\}$ *arbitrary*: y)
    case *empty*
    hence $y = (\lambda a.\ b)$ **by**(*auto*)
    thus *?case* **by**(*simp*)
  next
    case (*insert* x F)
    note $IH = \langle \bigwedge y.\ F = \{a.\ y\ a \neq b\} \Longrightarrow finite\ \{c.\ g\ (y\ c) \neq g\ b\}\rangle$
    from $\langle insert\ x\ F = \{a.\ y\ a \neq b\}\rangle$ $\langle x \notin F\rangle$
    have F: $F = \{a.\ (y(x := b))\ a \neq b\}$ **by**(*auto*)
    show *?case*
    proof(*cases* $g\ (y\ x) = g\ b$)
      case *True*
      hence $\{c.\ g\ ((y(x := b))\ c) \neq g\ b\} = \{c.\ g\ (y\ c) \neq g\ b\}$ **by** *auto*
      with *IH*[*OF F*] **show** *?thesis* **by** *simp*
    next
      case *False*
      hence $\{c.\ g\ (y\ c) \neq g\ b\} = insert\ x\ \{c.\ g\ ((y(x := b))\ c) \neq g\ b\}$ **by** *auto*
      with *IH*[*OF F*] **show** *?thesis* **by**(*simp*)
    qed
  qed
  thus *?thesis* **unfolding** *finfun-def* **by** *auto*
qed

lemma assumes $y \in \mathit{finfun}$
  shows *fst-finfun*: $\mathit{fst} \circ y \in \mathit{finfun}$
  and *snd-finfun*: $\mathit{snd} \circ y \in \mathit{finfun}$
proof −
  from *assms* **obtain** b c **where** *bc*: $\mathit{finite}\ \{a.\ y\ a \neq (b,\ c)\}$
    unfolding *finfun-def* **by** *auto*
  have $\{a.\ \mathit{fst}\ (y\ a) \neq b\} \subseteq \{a.\ y\ a \neq (b,\ c)\}$
    and $\{a.\ \mathit{snd}\ (y\ a) \neq c\} \subseteq \{a.\ y\ a \neq (b,\ c)\}$ **by** *auto*
  hence $\mathit{finite}\ \{a.\ \mathit{fst}\ (y\ a) \neq b\}$
    and $\mathit{finite}\ \{a.\ \mathit{snd}\ (y\ a) \neq c\}$ **using** *bc* **by**(*auto intro*: *finite-subset*)
  thus $\mathit{fst} \circ y \in \mathit{finfun}$ $\mathit{snd} \circ y \in \mathit{finfun}$
    unfolding *finfun-def* **by** *auto*
qed

lemma *map-of-finfun*: *map-of xs* $\in$ *finfun*
unfolding *finfun-def*
by(*induct xs*)(*auto simp add*: *Collect-neg-eq Collect-conj-eq Collect-imp-eq intro*: *finite-subset*)

lemma *Diag-finfun*: $(\lambda x.\ (f\ x,\ g\ x)) \in \mathit{finfun} \longleftrightarrow f \in \mathit{finfun} \wedge g \in \mathit{finfun}$
by(*auto intro*: *finite-subset simp add*: *Collect-neg-eq Collect-imp-eq Collect-conj-eq finfun-def*)

lemma *finfun-right-compose*:
  assumes *g*: $g \in \mathit{finfun}$ **and** *inj*: *inj f*
  shows $g\ o\ f \in \mathit{finfun}$
proof −
  from *g* **obtain** b **where** *b*: $\mathit{finite}\ \{a.\ g\ a \neq b\}$ **unfolding** *finfun-def* **by** *blast*
  moreover have f ‘ $\{a.\ g\ (f\ a) \neq b\} \subseteq \{a.\ g\ a \neq b\}$ **by** *auto*
  moreover from *inj* **have** *inj-on f* $\{a.\ \ g\ (f\ a) \neq b\}$ **by**(*rule subset-inj-on*) *blast*
  ultimately have $\mathit{finite}\ \{a.\ g\ (f\ a) \neq b\}$
    by(*blast intro*: *finite-imageD*[**where** *f*=*f*] *finite-subset*)
  thus *?thesis* **unfolding** *finfun-def* **by** *auto*
qed

lemma *finfun-curry*:
  assumes *fin*: $f \in \mathit{finfun}$
  shows *curry f* $\in$ *finfun* *curry f a* $\in$ *finfun*
proof −
  from *fin* **obtain** c **where** *c*: $\mathit{finite}\ \{ab.\ f\ ab \neq c\}$ **unfolding** *finfun-def* **by** *blast*
  moreover have $\{a.\ \exists\ b.\ f\ (a,\ b) \neq c\} = \mathit{fst}$ ‘ $\{ab.\ f\ ab \neq c\}$ **by**(*force*)
  hence $\{a.\ \mathit{curry}\ f\ a \neq (\lambda b.\ c)\} = \mathit{fst}$ ‘ $\{ab.\ f\ ab \neq c\}$
    by(*auto simp add*: *curry-def fun-eq-iff*)
  ultimately have $\mathit{finite}\ \{a.\ \mathit{curry}\ f\ a \neq (\lambda b.\ c)\}$ **by** *simp*
  thus *curry f* $\in$ *finfun* **unfolding** *finfun-def* **by** *blast*

  have *snd* ‘ $\{ab.\ f\ ab \neq c\} = \{b.\ \exists\ a.\ f\ (a,\ b) \neq c\}$ **by**(*force*)
  hence $\{b.\ f\ (a,\ b) \neq c\} \subseteq \mathit{snd}$ ‘ $\{ab.\ f\ ab \neq c\}$ **by** *auto*

hence *finite* {*b*. *f* (*a*, *b*) ≠ *c*} **by**(*rule finite-subset*)(*rule finite-imageI*[*OF c*])
thus *curry f a* ∈ *finfun* **unfolding** *finfun-def* **by** *auto*
qed

bundle *finfun* =
fst-finfun[*simp*] *snd-finfun*[*simp*] *Abs-finfun-inverse*[*simp*]
finfun-apply-inverse[*simp*] *Abs-finfun-inject*[*simp*] *finfun-apply-inject*[*simp*]
Diag-finfun[*simp*] *finfun-curry*[*simp*]
const-finfun[*iff*] *fun-upd-finfun*[*iff*] *finfun-apply*[*iff*] *map-of-finfun*[*iff*]
finfun-left-compose[*intro*] *fst-finfun*[*intro*] *snd-finfun*[*intro*]

lemma *Abs-finfun-inject-finite*:
fixes *x y* :: *'a* ⇒ *'b*
assumes *fin*: *finite* (*UNIV* :: *'a set*)
shows *Abs-finfun x* = *Abs-finfun y* ⟷ *x* = *y*
proof
assume *Abs-finfun x* = *Abs-finfun y*
moreover have *x* ∈ *finfun y* ∈ *finfun* **unfolding** *finfun-def*
by(*auto intro*: *finite-subset*[*OF - fin*])
ultimately show *x* = *y* **by**(*simp add*: *Abs-finfun-inject*)
qed *simp*

lemma *Abs-finfun-inject-finite-class*:
fixes *x y* :: (*'a* :: *finite*) ⇒ *'b*
shows *Abs-finfun x* = *Abs-finfun y* ⟷ *x* = *y*
using *finite-UNIV*
by(*simp add*: *Abs-finfun-inject-finite*)

lemma *Abs-finfun-inj-finite*:
assumes *fin*: *finite* (*UNIV* :: *'a set*)
shows *inj* (*Abs-finfun* :: (*'a* ⇒ *'b*) ⇒ *'a* ⇒*f* *'b*)
proof(*rule inj-onI*)
fix *x y* :: *'a* ⇒ *'b*
assume *Abs-finfun x* = *Abs-finfun y*
moreover have *x* ∈ *finfun y* ∈ *finfun* **unfolding** *finfun-def*
by(*auto intro*: *finite-subset*[*OF - fin*])
ultimately show *x* = *y* **by**(*simp add*: *Abs-finfun-inject*)
qed

lemma *Abs-finfun-inverse-finite*:
fixes *x* :: *'a* ⇒ *'b*
assumes *fin*: *finite* (*UNIV* :: *'a set*)
shows *finfun-apply* (*Abs-finfun x*) = *x*
including *finfun*
proof −
from *fin* **have** *x* ∈ *finfun*
by(*auto simp add*: *finfun-def intro*: *finite-subset*)
thus *?thesis* **by** *simp*
qed

lemma *Abs-finfun-inverse-finite-class*:
  fixes x :: (*'a* :: *finite*) ⇒ *'b*
  shows *finfun-apply* (*Abs-finfun* x) = x
using *finite-UNIV* **by**(*simp add*: *Abs-finfun-inverse-finite*)

lemma *finfun-eq-finite-UNIV*: *finite* (*UNIV* :: *'a set*) ⟹ (*finfun* :: (*'a* ⇒ *'b*) *set*) = *UNIV*
unfolding *finfun-def* **by**(*auto intro*: *finite-subset*)

lemma *finfun-finite-UNIV-class*: *finfun* = (*UNIV* :: (*'a* :: *finite* ⇒ *'b*) *set*)
by(*simp add*: *finfun-eq-finite-UNIV*)

lemma *map-default-in-finfun*:
  assumes *fin*: *finite* (*dom f*)
  shows *map-default b f* ∈ *finfun*
unfolding *finfun-def*
proof(*intro CollectI exI*)
  from *fin* **show** *finite* {*a*. *map-default b f a* ≠ *b*}
    by(*auto simp add*: *map-default-def dom-def Collect-conj-eq split*: *option.splits*)
qed

lemma *finfun-cases-map-default*:
  obtains *b g* **where** *f* = *Abs-finfun* (*map-default b g*) *finite* (*dom g*) *b* ∉ *ran g*
proof −
  obtain *y* **where** *f*: *f* = *Abs-finfun y* **and** *y*: *y* ∈ *finfun* **by**(*cases f*)
  from *y* **obtain** *b* **where** *b*: *finite* {*a*. *y a* ≠ *b*} **unfolding** *finfun-def* **by** *auto*
  let *?g* = (λ*a*. *if y a* = *b then None else Some* (*y a*))
  have *map-default b ?g* = *y* **by**(*simp add*: *fun-eq-iff map-default-def*)
  with *f* **have** *f* = *Abs-finfun* (*map-default b ?g*) **by** *simp*
  moreover from *b* **have** *finite* (*dom ?g*) **by**(*auto simp add*: *dom-def*)
  moreover have *b* ∉ *ran ?g* **by**(*auto simp add*: *ran-def*)
  ultimately show *?thesis* **by**(*rule that*)
qed

37.3 Kernel functions for type 'a ⇒f 'b

lift-definition *finfun-const* :: *'b* ⇒ *'a* ⇒f *'b* (*K*$/ - [*0*] *1*)
is λ *b x*. *b* **by** (*rule const-finfun*)

lift-definition *finfun-update* :: *'a* ⇒f *'b* ⇒ *'a* ⇒ *'b* ⇒ *'a* ⇒f *'b* (-'(- $:= -')
[*1000*,*0*,*0*] *1000*) **is** *fun-upd*
by (*simp add*: *fun-upd-finfun*)

lemma *finfun-update-twist*: *a* ≠ *a'* ⟹ *f*(*a* $:= *b*)(*a'* $:= *b'*) = *f*(*a'* $:= *b'*)(*a* $:= *b*)
by *transfer* (*simp add*: *fun-upd-twist*)

lemma *finfun-update-twice* [*simp*]:

$f(a \$:= b)(a \$:= b') = f(a \$:= b')$
by *transfer simp*

lemma *finfun-update-const-same*: $(K\$ \ b)(a \$:= b) = (K\$ \ b)$
by *transfer* (*simp add*: *fun-eq-iff*)

37.4 Code generator setup

definition *finfun-update-code* :: $'a \Rightarrow f \ 'b \Rightarrow \ 'a \Rightarrow \ 'b \Rightarrow \ 'a \Rightarrow f \ 'b$
where [*simp*, *code del*]: *finfun-update-code* = *finfun-update*

code-datatype *finfun-const finfun-update-code*

lemma *finfun-update-const-code* [*code*]:
$(K\$ \ b)(a \$:= b') = (\textit{if } b = b' \textit{ then } (K\$ \ b) \textit{ else finfun-update-code } (K\$ \ b) \ a \ b')$
by(*simp add*: *finfun-update-const-same*)

lemma *finfun-update-update-code* [*code*]:
$(\textit{finfun-update-code } f \ a \ b)(a' \$:= b') = (\textit{if } a = a' \textit{ then } f(a \$:= b') \textit{ else finfun-update-code } (f(a' \$:= b')) \ a \ b)$
by(*simp add*: *finfun-update-twist*)

37.5 Setup for quickcheck

quickcheck-generator *finfun constructors*: *finfun-update-code*, *finfun-const* :: $'b \Rightarrow \ 'a \Rightarrow f \ 'b$

37.6 *finfun-update* as instance of *comp-fun-commute*

interpretation *finfun-update*: *comp-fun-commute* $\lambda a \ f. \ f(a :: \ 'a \$:= b')$
  including *finfun*
proof
  fix $a \ a' :: \ 'a$
  show $(\lambda f. \ f(a \$:= b')) \circ (\lambda f. \ f(a' \$:= b')) = (\lambda f. \ f(a' \$:= b')) \circ (\lambda f. \ f(a \$:= b'))$
  proof
    fix b
    have $(\textit{finfun-apply } b)(a := b', a' := b') = (\textit{finfun-apply } b)(a' := b', a := b')$
      by(*cases* $a = a'$)(*auto simp add*: *fun-upd-twist*)
    then have $b(a \$:= b')(a' \$:= b') = b(a' \$:= b')(a \$:= b')$
      by(*auto simp add*: *finfun-update-def fun-upd-twist*)
    then show $((\lambda f. \ f(a \$:= b')) \circ (\lambda f. \ f(a' \$:= b'))) \ b = ((\lambda f. \ f(a' \$:= b')) \circ (\lambda f. \ f(a \$:= b'))) \ b$
      by (*simp add*: *fun-eq-iff*)
  qed
qed

lemma *fold-finfun-update-finite-univ*:
  assumes *fin*: *finite* $(UNIV :: \ 'a \ set)$
  shows $\textit{Finite-Set.fold } (\lambda a \ f. \ f(a \$:= b')) \ (K\$ \ b) \ (UNIV :: \ 'a \ set) = (K\$ \ b')$

proof −
 { **fix** *A* :: *'a set*
  from *fin* **have** *finite A* **by**(*auto intro*: *finite-subset*)
  hence *Finite-Set.fold* (λ*a f*. *f*(*a* $:= *b'*)) (*K$ b*) *A* = *Abs-finfun* (λ*a*. *if a* ∈ *A then b' else b*)
  proof(*induct*)
   case (*insert x F*)
   have (λ*a*. *if a* = *x then b' else* (*if a* ∈ *F then b' else b*)) = (λ*a*. *if a* = *x* ∨ *a* ∈ *F then b' else b*)
    by(*auto*)
   with *insert* **show** *?case*
    by(*simp add*: *finfun-const-def fun-upd-def*)(*simp add*: *finfun-update-def Abs-finfun-inverse-finite*[*OF fin*] *fun-upd-def*)
  qed(*simp add*: *finfun-const-def*) }
 thus *?thesis* **by**(*simp add*: *finfun-const-def*)
qed

37.7 Default value for FinFuns

definition *finfun-default-aux* :: (*'a* ⇒ *'b*) ⇒ *'b*
where [*code del*]: *finfun-default-aux f* = (*if finite* (*UNIV* :: *'a set*) *then undefined else THE b*. *finite* {*a*. *f a* ≠ *b*})

lemma *finfun-default-aux-infinite*:
 fixes *f* :: *'a* ⇒ *'b*
 assumes *infin*: ¬ *finite* (*UNIV* :: *'a set*)
 and *fin*: *finite* {*a*. *f a* ≠ *b*}
 shows *finfun-default-aux f* = *b*
proof −
 let *?B* = {*a*. *f a* ≠ *b*}
 from *fin* **have** (*THE b*. *finite* {*a*. *f a* ≠ *b*}) = *b*
 proof(*rule the-equality*)
  fix *b'*
  assume *finite* {*a*. *f a* ≠ *b'*} (**is** *finite ?B'*)
  with *infin fin* **have** *UNIV* − (*?B'* ∪ *?B*) ≠ {} **by**(*auto dest*: *finite-subset*)
  then obtain *a* **where** *a*: *a* ∉ *?B'* ∪ *?B* **by** *auto*
  thus *b'* = *b* **by** *auto*
 qed
 thus *?thesis* **using** *infin* **by**(*simp add*: *finfun-default-aux-def*)
qed

lemma *finite-finfun-default-aux*:
 fixes *f* :: *'a* ⇒ *'b*
 assumes *fin*: *f* ∈ *finfun*
 shows *finite* {*a*. *f a* ≠ *finfun-default-aux f*}
proof(*cases finite* (*UNIV* :: *'a set*))
 case *True* **thus** *?thesis* **using** *fin*
  by(*auto simp add*: *finfun-def finfun-default-aux-def intro*: *finite-subset*)

next
  case *False*
  from *fin* **obtain** b **where** b: *finite* $\{a.\ f\ a \neq b\}$ (**is** *finite ?B*)
    unfolding *finfun-def* **by** *blast*
  with *False* **show** *?thesis* **by**(*simp add*: *finfun-default-aux-infinite*)
qed

lemma *finfun-default-aux-update-const*:
  fixes $f :: 'a \Rightarrow 'b$
  assumes *fin*: $f \in$ *finfun*
  shows *finfun-default-aux* $(f(a := b))$ = *finfun-default-aux f*
proof(*cases finite* (*UNIV* :: $'a$ *set*))
  case *False*
  from *fin* **obtain** b' **where** b': *finite* $\{a.\ f\ a \neq b'\}$ **unfolding** *finfun-def* **by** *blast*
  hence *finite* $\{a'.\ (f(a := b))\ a' \neq b'\}$
  proof(*cases* $b = b' \wedge f\ a \neq b'$)
    case *True*
    hence $\{a.\ f\ a \neq b'\}$ = *insert* a $\{a'.\ (f(a := b))\ a' \neq b'\}$ **by** *auto*
    thus *?thesis* **using** b' **by** *simp*
  next
    case *False*
    moreover
    { assume $b \neq b'$
      hence $\{a'.\ (f(a := b))\ a' \neq b'\}$ = *insert* a $\{a.\ f\ a \neq b'\}$ **by** *auto*
      hence *?thesis* **using** b' **by** *simp* **}**
    moreover
    { assume $b = b'\ f\ a = b'$
      hence $\{a'.\ (f(a := b))\ a' \neq b'\} = \{a.\ f\ a \neq b'\}$ **by** *auto*
      hence *?thesis* **using** b' **by** *simp* **}**
    ultimately show *?thesis* **by** *blast*
  qed
  with *False* b' **show** *?thesis* **by**(*auto simp del*: *fun-upd-apply simp add*: *finfun-default-aux-infinite*)
next
  case *True* **thus** *?thesis* **by**(*simp add*: *finfun-default-aux-def*)
qed

lift-definition *finfun-default* :: $'a \Rightarrow f\ 'b \Rightarrow 'b$
is *finfun-default-aux* **.**

lemma *finite-finfun-default*: *finite* $\{a.$ *finfun-apply f a* $\neq$ *finfun-default f*$\}$
by *transfer* (*erule finite-finfun-default-aux*)

lemma *finfun-default-const*: *finfun-default* $((K\$\ b) :: 'a \Rightarrow f\ 'b)$ = (*if finite* (*UNIV* :: $'a$ *set*) *then undefined else* b)
by(*transfer*)(*auto simp add*: *finfun-default-aux-infinite finfun-default-aux-def*)

lemma *finfun-default-update-const*:
  finfun-default $(f(a\ \$:= b))$ = *finfun-default f*
by *transfer* (*simp add*: *finfun-default-aux-update-const*)

lemma *finfun-default-const-code* [*code*]:
finfun-default ((*K$ c*) :: *'a* :: *card-UNIV* ⇒*f* *'b*) = (*if CARD*(*'a*) = *0 then c else undefined*)
by(*simp add*: *finfun-default-const*)

lemma *finfun-default-update-code* [*code*]:
finfun-default (*finfun-update-code f a b*) = *finfun-default f*
by(*simp add*: *finfun-default-update-const*)

37.8 Recursion combinator and well-formedness conditions

definition *finfun-rec* :: (*'b* ⇒ *'c*) ⇒ (*'a* ⇒ *'b* ⇒ *'c* ⇒ *'c*) ⇒ (*'a* ⇒*f* *'b*) ⇒ *'c*
where [*code del*]:
finfun-rec cnst upd f ≡
let b = *finfun-default f*;
g = *THE g*. *f* = *Abs-finfun* (*map-default b g*) ∧ *finite* (*dom g*) ∧ *b* ∉ *ran g*
in Finite-Set.fold (λ*a*. *upd a* (*map-default b g a*)) (*cnst b*) (*dom g*)

locale *finfun-rec-wf-aux* =
fixes *cnst* :: *'b* ⇒ *'c*
and *upd* :: *'a* ⇒ *'b* ⇒ *'c* ⇒ *'c*
assumes *upd-const-same*: *upd a b* (*cnst b*) = *cnst b*
and *upd-commute*: *a* ≠ *a'* ⟹ *upd a b* (*upd a' b' c*) = *upd a' b'* (*upd a b c*)
and *upd-idemp*: *b* ≠ *b'* ⟹ *upd a b''* (*upd a b'* (*cnst b*)) = *upd a b''* (*cnst b*)
begin

lemma *upd-left-comm*: *comp-fun-commute* (λ*a*. *upd a* (*f a*))
by(*unfold-locales*)(*auto intro*: *upd-commute simp add*: *fun-eq-iff*)

lemma *upd-upd-twice*: *upd a b''* (*upd a b'* (*cnst b*)) = *upd a b''* (*cnst b*)
by(*cases b* ≠ *b'*)(*auto simp add*: *fun-upd-def upd-const-same upd-idemp*)

lemma *map-default-update-const*:
assumes *fin*: *finite* (*dom f*)
and *anf*: *a* ∉ *dom f*
and *fg*: *f* $\subseteq_m$ *g*
shows *upd a d* (*Finite-Set.fold* (λ*a*. *upd a* (*map-default d g a*)) (*cnst d*) (*dom f*)) =
Finite-Set.fold (λ*a*. *upd a* (*map-default d g a*)) (*cnst d*) (*dom f*)
proof −
let *?upd* = λ*a*. *upd a* (*map-default d g a*)
let *?fr* = λ*A*. *Finite-Set.fold ?upd* (*cnst d*) *A*
interpret *gwf*: *comp-fun-commute ?upd* **by**(*rule upd-left-comm*)

from *fin anf fg* **show** *?thesis*
proof(*induct dom f arbitrary*: *f*)
case *empty*

from ‹{} = *dom f*› **have** *f* = *empty* **by**(*auto simp add*: *dom-def*)
thus *?case* **by**(*simp add*: *finfun-const-def upd-const-same*)
next
case (*insert a′ A*)
note *IH* = ‹⋀*f*. ⟦ *A* = *dom f*; *a* ∉ *dom f*; *f* ⊆*m* *g* ⟧ ⟹ *upd a d* (*?fr* (*dom f*)) = *?fr* (*dom f*)›
note *fin* = ‹*finite A*› **note** *anf* = ‹*a* ∉ *dom f*› **note** *a′nA* = ‹*a′* ∉ *A*›
note *domf* = ‹*insert a′ A* = *dom f*› **note** *fg* = ‹*f* ⊆*m* *g*›

from *domf* **obtain** *b* **where** *b*: *f a′* = *Some b* **by** *auto*
let *?f′* = *f*(*a′* := *None*)
have *upd a d* (*?fr* (*insert a′ A*)) = *upd a d* (*upd a′* (*map-default d g a′*) (*?fr A*))
by(*subst gwf.fold-insert*[*OF fin a′nA*]) *rule*
also from *b fg* **have** *g a′* = *f a′* **by**(*auto simp add*: *map-le-def intro*: *domI dest*: *bspec*)
hence *ga′*: *map-default d g a′* = *map-default d f a′* **by**(*simp add*: *map-default-def*)
also from *anf domf* **have** *a* ≠ *a′* **by** *auto* **note** *upd-commute*[*OF this*]
also from *domf a′nA anf fg* **have** *a* ∉ *dom ?f′ ?f′* ⊆*m* *g* **and** *A*: *A* = *dom ?f′* **by**(*auto simp add*: *ran-def map-le-def*)
note *A* **also note** *IH*[*OF A* ‹*a* ∉ *dom ?f′*› ‹*?f′* ⊆*m* *g*›]
also have *upd a′* (*map-default d f a′*) (*?fr* (*dom* (*f*(*a′* := *None*)))) = *?fr* (*dom f*)
unfolding *domf*[*symmetric*] *gwf.fold-insert*[*OF fin a′nA*] *ga′* **unfolding** *A* **..**
also have *insert a′* (*dom ?f′*) = *dom f* **using** *domf* **by** *auto*
finally show *?case* **.**
qed
qed

lemma *map-default-update-twice*:
assumes *fin*: *finite* (*dom f*)
and *anf*: *a* ∉ *dom f*
and *fg*: *f* ⊆*m* *g*
shows *upd a d″* (*upd a d′* (*Finite-Set.fold* (λ*a*. *upd a* (*map-default d g a*)) (*cnst d*) (*dom f*))) =
upd a d″ (*Finite-Set.fold* (λ*a*. *upd a* (*map-default d g a*)) (*cnst d*) (*dom f*))
proof −
let *?upd* = λ*a*. *upd a* (*map-default d g a*)
let *?fr* = λ*A*. *Finite-Set.fold ?upd* (*cnst d*) *A*
interpret *gwf*: *comp-fun-commute ?upd* **by**(*rule upd-left-comm*)

from *fin anf fg* **show** *?thesis*
proof(*induct dom f arbitrary*: *f*)
case *empty*
from ‹{} = *dom f*› **have** *f* = *empty* **by**(*auto simp add*: *dom-def*)
thus *?case* **by**(*auto simp add*: *finfun-const-def finfun-update-def upd-upd-twice*)
next
case (*insert a′ A*)
note *IH* = ‹⋀*f*. ⟦*A* = *dom f*; *a* ∉ *dom f*; *f* ⊆*m* *g*⟧ ⟹ *upd a d″* (*upd a d′* (*?fr*

(*dom f*))) = *upd a d''* (*?fr* (*dom f*))›
note *fin* = ‹*finite A*› **note** *anf* = ‹$a \notin$ *dom f*› **note** *a'nA* = ‹$a' \notin A$›
note *domf* = ‹*insert a' A* = *dom f*› **note** *fg* = ‹$f \subseteq_m g$›

from *domf* **obtain** *b* **where** *b*: *f a'* = *Some b* **by** *auto*
let *?f'* = *f*(*a'* := *None*)
let *?b'* = *case f a' of None* ⇒ *d* | *Some b* ⇒ *b*
from *domf* **have** *upd a d''* (*upd a d'* (*?fr* (*dom f*))) = *upd a d''* (*upd a d'* (*?fr* (*insert a' A*))) **by** *simp*
also note *gwf.fold-insert*[*OF fin a'nA*]
also from *b fg* **have** *g a'* = *f a'* **by**(*auto simp add*: *map-le-def intro*: *domI dest*: *bspec*)
hence *ga'*: *map-default d g a'* = *map-default d f a'* **by**(*simp add*: *map-default-def*)
also from *anf domf* **have** *ana'*: $a \neq a'$ **by** *auto* **note** *upd-commute*[*OF this*]
also note *upd-commute*[*OF ana'*]
also from *domf a'nA anf fg* **have** $a \notin$ *dom ?f'* *?f'* $\subseteq_m$ *g* **and** *A*: *A* = *dom ?f'* **by**(*auto simp add*: *ran-def map-le-def*)
note *A* **also note** *IH*[*OF A* ‹$a \notin$ *dom ?f'*› ‹*?f'* $\subseteq_m$ *g*›]
also note *upd-commute*[*OF ana'*[*symmetric*]] **also note** *ga'*[*symmetric*] **also note** *A*[*symmetric*]
also note *gwf.fold-insert*[*symmetric*, *OF fin a'nA*] **also note** *domf*
finally show *?case* **.**
qed
qed

lemma *map-default-eq-id* [*simp*]: *map-default d* ((λ*a*. *Some* (*f a*)) |‘ {*a*. *f a* ≠ *d*}) = *f*
by(*auto simp add*: *map-default-def restrict-map-def*)

lemma *finite-rec-cong1*:
assumes *f*: *comp-fun-commute f* **and** *g*: *comp-fun-commute g*
and *fin*: *finite A*
and *eq*: $\bigwedge a.\ a \in A \Longrightarrow f\ a = g\ a$
shows *Finite-Set.fold f z A* = *Finite-Set.fold g z A*
proof −
interpret *f*: *comp-fun-commute f* **by**(*rule f*)
interpret *g*: *comp-fun-commute g* **by**(*rule g*)
{ fix *B*
assume *BsubA*: $B \subseteq A$
with *fin* **have** *finite B* **by**(*blast intro*: *finite-subset*)
hence $B \subseteq A \Longrightarrow$ *Finite-Set.fold f z B* = *Finite-Set.fold g z B*
proof(*induct*)
case *empty* **thus** *?case* **by** *simp*
next
case (*insert a B*)
note *finB* = ‹*finite B*› **note** *anB* = ‹$a \notin B$› **note** *sub* = ‹*insert a B* $\subseteq A$›
note *IH* = ‹$B \subseteq A \Longrightarrow$ *Finite-Set.fold f z B* = *Finite-Set.fold g z B*›
from *sub anB* **have** *BpsubA*: $B \subset A$ **and** *BsubA*: $B \subseteq A$ **and** *aA*: $a \in A$ **by** *auto*

from *IH*[*OF BsubA*] *eq*[*OF aA*] *finB anB*
show *?case* **by**(*auto*)
qed
with *BsubA* **have** *Finite-Set.fold f z B* = *Finite-Set.fold g z B* **by** *blast* **}**
thus *?thesis* **by** *blast*
qed

lemma *finfun-rec-upd* [*simp*]:
finfun-rec cnst upd ($f(a'$ \$:= b')) = *upd* a' b' (*finfun-rec cnst upd f*)
including *finfun*
proof −
obtain *b* **where** *b*: *b* = *finfun-default f* **by** *auto*
let *?the* = $\lambda f\ g$. *f* = *Abs-finfun* (*map-default b g*) $\wedge$ *finite* (*dom g*) $\wedge$ $b \notin$ *ran g*
obtain *g* **where** *g*: *g* = *The* (*?the f*) **by** *blast*
obtain *y* **where** *f*: *f* = *Abs-finfun y* **and** *y*: $y \in$ *finfun* **by** (*cases f*)
from *f y b* **have** *bfin*: *finite* {*a*. *y a* $\neq$ *b*} **by**(*simp add*: *finfun-default-def finite-finfun-default-aux*)

let *?g* = (λa. *Some* (*y a*)) |' {*a*. *y a* $\neq$ *b*}
from *bfin* **have** *fing*: *finite* (*dom ?g*) **by** *auto*
have *bran*: $b \notin$ *ran ?g* **by**(*auto simp add*: *ran-def restrict-map-def*)
have *yg*: *y* = *map-default b ?g* **by** *simp*
have *gg*: *g* = *?g* **unfolding** *g*
proof(*rule the-equality*)
from *f y bfin* **show** *?the f ?g*
by(*auto*)(*simp add*: *restrict-map-def ran-def split*: *if-split-asm*)
next
fix g'
assume *?the f* g'
hence *fin'*: *finite* (*dom* g') **and** *ran'*: $b \notin$ *ran* g'
and *eq*: *Abs-finfun* (*map-default b ?g*) = *Abs-finfun* (*map-default b* g') **using** *f yg* **by** *auto*
from *fin' fing* **have** *map-default b ?g* $\in$ *finfun map-default b* g' $\in$ *finfun* **by**(*blast intro*: *map-default-in-finfun*)+
with *eq* **have** *map-default b ?g* = *map-default b* g' **by** *simp*
with *fing bran fin' ran'* **show** g' = *?g* **by**(*rule map-default-inject*[*OF disjI2*[*OF refl*], *THEN sym*])
qed

show *?thesis*
proof(*cases* $b' = b$)
case *True*
note *b'b* = *True*

let *?g'* = (λa. *Some* (($y(a'$:= b)) *a*)) |' {*a*. ($y(a'$:= b)) *a* $\neq$ *b*}
from *bfin b'b* **have** *fing'*: *finite* (*dom ?g'*)
by(*auto simp add*: *Collect-conj-eq Collect-imp-eq intro*: *finite-subset*)
have *brang'*: $b \notin$ *ran ?g'* **by**(*auto simp add*: *ran-def restrict-map-def*)

let *?b'* = *λa. case ?g' a of None ⇒ b | Some b ⇒ b*
let *?b* = *map-default b ?g*
from *upd-left-comm upd-left-comm fing'*
have *Finite-Set.fold* (*λa. upd a* (*?b' a*)) (*cnst b*) (*dom ?g'*) = *Finite-Set.fold* (*λa. upd a* (*?b a*)) (*cnst b*) (*dom ?g'*)
by(*rule finite-rec-cong1*)(*auto simp add*: *restrict-map-def b'b b map-default-def*)
also interpret *gwf*: *comp-fun-commute λa. upd a* (*?b a*) **by**(*rule upd-left-comm*)
have *Finite-Set.fold* (*λa. upd a* (*?b a*)) (*cnst b*) (*dom ?g'*) = *upd a' b'* (*Finite-Set.fold* (*λa. upd a* (*?b a*)) (*cnst b*) (*dom ?g*))
proof(*cases y a'* = *b*)
case *True*
with *b'b* **have** *g'*: *?g'* = *?g* **by**(*auto simp add*: *restrict-map-def*)
from *True* **have** *a'ndomg*: *a'* ∉ *dom ?g* **by** *auto*
from *f b'b b* **show** *?thesis* **unfolding** *g'*
by(*subst map-default-update-const*[*OF fing a'ndomg map-le-refl*, *symmetric*]) *simp*
next
case *False*
hence *domg*: *dom ?g* = *insert a'* (*dom ?g'*) **by** *auto*
from *False b'b* **have** *a'ndomg'*: *a'* ∉ *dom ?g'* **by** *auto*
have *Finite-Set.fold* (*λa. upd a* (*?b a*)) (*cnst b*) (*insert a'* (*dom ?g'*)) =
upd a' (*?b a'*) (*Finite-Set.fold* (*λa. upd a* (*?b a*)) (*cnst b*) (*dom ?g'*))
using *fing' a'ndomg'* **unfolding** *b'b* **by**(*rule gwf.fold-insert*)
hence *upd a' b* (*Finite-Set.fold* (*λa. upd a* (*?b a*)) (*cnst b*) (*insert a'* (*dom ?g'*))) =
upd a' b (*upd a'* (*?b a'*) (*Finite-Set.fold* (*λa. upd a* (*?b a*)) (*cnst b*) (*dom ?g'*))) **by** *simp*
also from *b'b* **have** *g'leg*: *?g'* $\subseteq_m$ *?g* **by**(*auto simp add*: *restrict-map-def map-le-def*)
note *map-default-update-twice*[*OF fing' a'ndomg' this*, *of b ?b a' b*]
also note *map-default-update-const*[*OF fing' a'ndomg' g'leg*, *of b*]
finally show *?thesis* **unfolding** *b'b domg*[*unfolded b'b*] **by**(*rule sym*)
qed
also have *The* (*?the* (*f*(*a'* $:= *b'*))) = *?g'*
proof(*rule the-equality*)
from *f y b b'b brang' fing'* **show** *?the* (*f*(*a'* $:= *b'*)) *?g'*
by(*auto simp del*: *fun-upd-apply simp add*: *finfun-update-def*)
next
fix *g'*
assume *?the* (*f*(*a'* $:= *b'*)) *g'*
hence *fin'*: *finite* (*dom g'*) **and** *ran'*: *b* ∉ *ran g'*
and *eq*: *f*(*a'* $:= *b'*) = *Abs-finfun* (*map-default b g'*)
by(*auto simp del*: *fun-upd-apply*)
from *fin' fing'* **have** *map-default b g'* ∈ *finfun map-default b ?g'* ∈ *finfun*
by(*blast intro*: *map-default-in-finfun*)+
with *eq f b'b b* **have** *map-default b ?g'* = *map-default b g'*
by(*simp del*: *fun-upd-apply add*: *finfun-update-def*)
with *fing' brang' fin' ran'* **show** *g'* = *?g'*
by(*rule map-default-inject*[*OF disjI2*[*OF refl*], *THEN sym*])

qed
ultimately show *?thesis* **unfolding** *finfun-rec-def Let-def b gg*[*unfolded g b*] **using** *bfin b'b b*
by(*simp only*: *finfun-default-update-const map-default-def*)
next
case *False*
note $b'b = this$
let $?g' = ?g(a' \mapsto b')$
let $?b' = map\text{-}default\ b\ ?g'$
let $?b = map\text{-}default\ b\ ?g$
from *fing* **have** *fing'*: *finite* (*dom ?g'*) **by** *auto*
from *bran b'b* **have** *bnrang'*: $b \notin ran\ ?g'$ **by**(*auto simp add*: *ran-def*)
have *ffmg'*: $map\text{-}default\ b\ ?g' = y(a' := b')$ **by**(*auto simp add*: *map-default-def restrict-map-def*)
with *f y* **have** *f-Abs*: $f(a'\ \${:=}\ b') = Abs\text{-}finfun\ (map\text{-}default\ b\ ?g')$ **by**(*auto simp add*: *finfun-update-def*)
have *g'*: $The\ (?the\ (f(a'\ \${:=}\ b'))) = ?g'$
proof (*rule the-equality*)
from *fing' bnrang' f-Abs* **show** $?the\ (f(a'\ \${:=}\ b'))\ ?g'$
by(*auto simp add*: *finfun-update-def restrict-map-def*)
next
fix *g'* **assume** $?the\ (f(a'\ \${:=}\ b'))\ g'$
hence *f'*: $f(a'\ \${:=}\ b') = Abs\text{-}finfun\ (map\text{-}default\ b\ g')$
and *fin'*: *finite* (*dom g'*) **and** *brang'*: $b \notin ran\ g'$ **by** *auto*
from *fing' fin'* **have** $map\text{-}default\ b\ ?g' \in finfun$ $map\text{-}default\ b\ g' \in finfun$
by(*auto intro*: *map-default-in-finfun*)
with *f' f-Abs* **have** $map\text{-}default\ b\ g' = map\text{-}default\ b\ ?g'$ **by** *simp*
with *fin' brang' fing' bnrang'* **show** $g' = ?g'$
by(*rule map-default-inject*[*OF disjI2*[*OF refl*]])
qed
have *dom*: $dom\ (((\lambda a.\ Some\ (y\ a))\ |`\ \{a.\ y\ a \neq b\})(a' \mapsto b')) = insert\ a'\ (dom\ ((\lambda a.\ Some\ (y\ a))\ |`\ \{a.\ y\ a \neq b\}))$
by *auto*
show *?thesis*
proof(*cases* $y\ a' = b$)
case *True*
hence *a'ndomg*: $a' \notin dom\ ?g$ **by** *auto*
from *f y b'b True* **have** *yff*: $y = map\text{-}default\ b\ (?g'\ |`\ dom\ ?g)$
by(*auto simp add*: *restrict-map-def map-default-def intro!*: *ext*)
hence *f'*: $f = Abs\text{-}finfun\ (map\text{-}default\ b\ (?g'\ |`\ dom\ ?g))$ **using** *f* **by** *simp*
interpret *g'wf*: *comp-fun-commute* $\lambda a.\ upd\ a\ (?b'\ a)$ **by**(*rule upd-left-comm*)
from *upd-left-comm upd-left-comm fing*
have $Finite\text{-}Set.fold\ (\lambda a.\ upd\ a\ (?b\ a))\ (cnst\ b)\ (dom\ ?g) = Finite\text{-}Set.fold\ (\lambda a.\ upd\ a\ (?b'\ a))\ (cnst\ b)\ (dom\ ?g)$
by(*rule finite-rec-cong1*)(*auto simp add*: *restrict-map-def b'b True map-default-def*)
thus *?thesis* **unfolding** *finfun-rec-def Let-def finfun-default-update-const b*[*symmetric*]
unfolding *g' g*[*symmetric*] *gg g'wf.fold-insert*[*OF fing a'ndomg*, *of cnst b*, *folded dom*]

by −(*rule arg-cong2*[**where** *f*=*upd a'*], *simp-all add*: *map-default-def*)
next
case *False*
hence *insert a'* (*dom ?g*) = *dom ?g* **by** *auto*
moreover {
let *?g''* = *?g*(*a'* := *None*)
let *?b''* = *map-default b ?g''*
from *False* **have** *domg*: *dom ?g* = *insert a'* (*dom ?g''*) **by** *auto*
from *False* **have** *a'ndomg''*: *a'* ∉ *dom ?g''* **by** *auto*
have *fing''*: *finite* (*dom ?g''*) **by**(*rule finite-subset*[*OF - fing*]) *auto*
have *bnrang''*: *b* ∉ *ran ?g''* **by**(*auto simp add*: *ran-def restrict-map-def*)
interpret *gwf*: *comp-fun-commute* λ*a*. *upd a* (*?b a*) **by**(*rule upd-left-comm*)
interpret *g'wf*: *comp-fun-commute* λ*a*. *upd a* (*?b' a*) **by**(*rule upd-left-comm*)
have *upd a' b'* (*Finite-Set.fold* (λ*a*. *upd a* (*?b a*)) (*cnst b*) (*insert a'* (*dom ?g''*))) =
upd a' b' (*upd a'* (*?b a'*) (*Finite-Set.fold* (λ*a*. *upd a* (*?b a*)) (*cnst b*) (*dom ?g''*)))
unfolding *gwf.fold-insert*[*OF fing'' a'ndomg''*] *f* **..**
also have *g''leg*: *?g* |' *dom ?g''* $\subseteq_m$ *?g* **by**(*auto simp add*: *map-le-def*)
have *dom* (*?g* |' *dom ?g''*) = *dom ?g''* **by** *auto*
note *map-default-update-twice*[**where** *d*=*b* **and** *f* = *?g* |' *dom ?g''* **and** *a*=*a'* **and** *d'*=*?b a'* **and** *d''*=*b'* **and** *g*=*?g*,
unfolded this, *OF fing'' a'ndomg'' g''leg*]
also have *b'*: *b'* = *?b' a'* **by**(*auto simp add*: *map-default-def*)
from *upd-left-comm upd-left-comm fing''*
have *Finite-Set.fold* (λ*a*. *upd a* (*?b a*)) (*cnst b*) (*dom ?g''*) =
Finite-Set.fold (λ*a*. *upd a* (*?b' a*)) (*cnst b*) (*dom ?g''*)
by(*rule finite-rec-cong1*)(*auto simp add*: *restrict-map-def b'b map-default-def*)
with *b'* **have** *upd a' b'* (*Finite-Set.fold* (λ*a*. *upd a* (*?b a*)) (*cnst b*) (*dom ?g''*)) =
upd a' (*?b' a'*) (*Finite-Set.fold* (λ*a*. *upd a* (*?b' a*)) (*cnst b*) (*dom ?g''*)) **by** *simp*
also note *g'wf.fold-insert*[*OF fing'' a'ndomg''*, *symmetric*]
finally have *upd a' b'* (*Finite-Set.fold* (λ*a*. *upd a* (*?b a*)) (*cnst b*) (*dom ?g*)) =
Finite-Set.fold (λ*a*. *upd a* (*?b' a*)) (*cnst b*) (*dom ?g*)
unfolding *domg* **.** }
ultimately have *Finite-Set.fold* (λ*a*. *upd a* (*?b' a*)) (*cnst b*) (*insert a'* (*dom ?g*)) =
upd a' b' (*Finite-Set.fold* (λ*a*. *upd a* (*?b a*)) (*cnst b*) (*dom ?g*)) **by** *simp*
thus *?thesis* **unfolding** *finfun-rec-def Let-def finfun-default-update-const b*[*symmetric*] *g*[*symmetric*] *g' dom*[*symmetric*]
using *b'b gg* **by**(*simp add*: *map-default-insert*)
qed
qed
qed

end

locale *finfun-rec-wf* = *finfun-rec-wf-aux* +
assumes *const-update-all*:
finite (*UNIV* :: *'a set*) ⟹ *Finite-Set.fold* (*λa. upd a b'*) (*cnst b*) (*UNIV* :: *'a set*) = *cnst b'*
begin

lemma *finfun-rec-const* [*simp*]: **includes** *finfun* **shows**
finfun-rec cnst upd (*K$ c*) = *cnst c*
proof(*cases finite* (*UNIV* :: *'a set*))
case *False*
hence *finfun-default* ((*K$ c*) :: *'a ⇒f 'b*) = *c* **by**(*simp add*: *finfun-default-const*)
moreover have (*THE g* :: *'a ⇀ 'b*. (*K$ c*) = *Abs-finfun* (*map-default c g*) ∧ *finite* (*dom g*) ∧ *c* ∉ *ran g*) = *empty*
proof (*rule the-equality*)
show (*K$ c*) = *Abs-finfun* (*map-default c empty*) ∧ *finite* (*dom empty*) ∧ *c* ∉ *ran empty*
by(*auto simp add*: *finfun-const-def*)
next
fix *g* :: *'a ⇀ 'b*
assume (*K$ c*) = *Abs-finfun* (*map-default c g*) ∧ *finite* (*dom g*) ∧ *c* ∉ *ran g*
hence *g*: (*K$ c*) = *Abs-finfun* (*map-default c g*) **and** *fin*: *finite* (*dom g*) **and** *ran*: *c* ∉ *ran g* **by** *blast+*
from *g map-default-in-finfun*[*OF fin*, *of c*] **have** *map-default c g* = (*λa. c*)
by(*simp add*: *finfun-const-def*)
moreover have *map-default c empty* = (*λa. c*) **by** *simp*
ultimately show *g* = *empty* **by**−(*rule map-default-inject*[*OF disjI2*[*OF refl*] *fin ran*], *auto*)
qed
ultimately show *?thesis* **by**(*simp add*: *finfun-rec-def*)
next
case *True*
hence *default*: *finfun-default* ((*K$ c*) :: *'a ⇒f 'b*) = *undefined* **by**(*simp add*: *finfun-default-const*)
let *?the* = *λg* :: *'a ⇀ 'b*. (*K$ c*) = *Abs-finfun* (*map-default undefined g*) ∧ *finite* (*dom g*) ∧ *undefined* ∉ *ran g*
show *?thesis*
proof(*cases c* = *undefined*)
case *True*
have *the*: *The ?the* = *empty*
proof (*rule the-equality*)
from *True* **show** *?the empty* **by**(*auto simp add*: *finfun-const-def*)
next
fix *g'*
assume *?the g'*
hence *fg*: (*K$ c*) = *Abs-finfun* (*map-default undefined g'*)
and *fin*: *finite* (*dom g'*) **and** *g*: *undefined* ∉ *ran g'* **by** *simp-all*
from *fin* **have** *map-default undefined g'* ∈ *finfun* **by**(*rule map-default-in-finfun*)
with *fg* **have** *map-default undefined g'* = (*λa. c*)

by(*auto simp add*: *finfun-const-def intro*: *Abs-finfun-inject*[*THEN iffD1*, *symmetric*])
with *True* **show** $g' = empty$
by −(*rule map-default-inject*(*2*)[*OF - fin g*], *auto*)
qed
show *?thesis* **unfolding** *finfun-rec-def* **using** ⟨*finite UNIV*⟩ *True*
unfolding *Let-def the default* **by**(*simp*)
next
case *False*
have *the*: *The ?the* = (λa :: $'a$. *Some c*)
proof (*rule the-equality*)
from *False True* **show** *?the* (λa :: $'a$. *Some c*)
by(*auto simp add*: *map-default-def* [*abs-def*] *finfun-const-def dom-def ran-def*)
next
fix g' :: $'a \rightharpoonup 'b$
assume *?the* g'
hence *fg*: (*K$ c*) = *Abs-finfun* (*map-default undefined* g')
and *fin*: *finite* (*dom* g') **and** *g*: *undefined* $\notin$ *ran* g' **by** *simp-all*
from *fin* **have** *map-default undefined* g' $\in$ *finfun* **by**(*rule map-default-in-finfun*)
with *fg* **have** *map-default undefined* g' = (λa. *c*)
by(*auto simp add*: *finfun-const-def intro*: *Abs-finfun-inject*[*THEN iffD1*])
with *True False* **show** g' = (λa::$'a$. *Some c*)
by − (*rule map-default-inject*(*2*)[*OF - fin g*],
auto simp add: *dom-def ran-def map-default-def* [*abs-def*])
qed
show *?thesis* **unfolding** *finfun-rec-def* **using** *True False*
unfolding *Let-def the default* **by**(*simp add*: *dom-def map-default-def const-update-all*)
qed
qed

end

37.9 Weak induction rule and case analysis for FinFuns

lemma *finfun-weak-induct* [*consumes 0*, *case-names const update*]:
assumes *const*: $\bigwedge b$. *P* (*K$ b*)
and *update*: $\bigwedge f\ a\ b$. $P\ f \Longrightarrow P\ (f(a\ \$:= b))$
shows *P x*
including *finfun*
proof(*induct x rule*: *Abs-finfun-induct*)
case (*Abs-finfun y*)
then obtain *b* **where** *finite* {*a*. *y a* $\neq$ *b*} **unfolding** *finfun-def* **by** *blast*
thus *?case* **using** ⟨$y \in$ *finfun*⟩
proof(*induct* {*a*. *y a* $\neq$ *b*} *arbitrary*: *y rule*: *finite-induct*)
case *empty*
hence $\bigwedge a$. *y a* = *b* **by** *blast*
hence *y* = (λa. *b*) **by**(*auto*)
hence *Abs-finfun y* = *finfun-const b* **unfolding** *finfun-const-def* **by** *simp*
thus *?case* **by**(*simp add*: *const*)

next
case (*insert a A*)
note *IH* = ‹$\bigwedge y$. ⟦ $A = \{a.\ y\ a \neq b\}$; $y \in$ *finfun* ⟧ $\Longrightarrow P$ (*Abs-finfun y*)›
note *y* = ‹$y \in$ *finfun*›
with ‹*insert a A* = $\{a.\ y\ a \neq b\}$› ‹$a \notin A$›
have $A = \{a'.\ (y(a := b))\ a' \neq b\}$ $y(a := b) \in$ *finfun* **by** *auto*
from *IH*[*OF this*] **have** *P* (*finfun-update* (*Abs-finfun* ($y(a := b)$)) *a* (*y a*))
by(*rule update*)
thus *?case* **using** *y* **unfolding** *finfun-update-def* **by** *simp*
qed
qed

lemma *finfun-exhaust-disj*: ($\exists b.\ x =$ *finfun-const b*) $\vee$ ($\exists f\ a\ b.\ x =$ *finfun-update f a b*)
by(*induct x rule*: *finfun-weak-induct*) *blast+*

lemma *finfun-exhaust*:
obtains *b* **where** $x = (K\$\ b)$
| *f a b* **where** $x = f(a\ \$:= b)$
by(*atomize-elim*)(*rule finfun-exhaust-disj*)

lemma *finfun-rec-unique*:
fixes $f :: 'a \Rightarrow f\ 'b \Rightarrow 'c$
assumes *c*: $\bigwedge c.\ f\ (K\$\ c) =$ *cnst c*
and *u*: $\bigwedge g\ a\ b.\ f\ (g(a\ \$:= b)) =$ *upd g a b* $(f\ g)$
and *c'*: $\bigwedge c.\ f'\ (K\$\ c) =$ *cnst c*
and *u'*: $\bigwedge g\ a\ b.\ f'\ (g(a\ \$:= b)) =$ *upd g a b* $(f'\ g)$
shows $f = f'$
proof
fix $g :: 'a \Rightarrow f\ 'b$
show $f\ g = f'\ g$
by(*induct g rule*: *finfun-weak-induct*)(*auto simp add*: *c u c' u'*)
qed

37.10 Function application

notation *finfun-apply* (**infixl** $ *999*)

interpretation *finfun-apply-aux*: *finfun-rec-wf-aux* $\lambda b.\ b\ \lambda a'\ b\ c.$ *if* $(a = a')$ *then b else c*
by(*unfold-locales*) *auto*

interpretation *finfun-apply*: *finfun-rec-wf* $\lambda b.\ b\ \lambda a'\ b\ c.$ *if* $(a = a')$ *then b else c*
proof(*unfold-locales*)
fix $b'\ b :: 'a$
assume *fin*: *finite* (*UNIV* :: *'b set*)
{ **fix** *A* :: *'b set*
interpret *comp-fun-commute* $\lambda a'.$ *If* $(a = a')\ b'$ **by**(*rule finfun-apply-aux.upd-left-comm*)
from *fin* **have** *finite A* **by**(*auto intro*: *finite-subset*)

hence *Finite-Set.fold* $(\lambda a'. \ If \ (a = a') \ b') \ b \ A = (if \ a \in A \ then \ b' \ else \ b)$
by *induct auto* **}**
from *this*[*of UNIV*] **show** *Finite-Set.fold* $(\lambda a'. \ If \ (a = a') \ b') \ b \ UNIV = b'$ **by** *simp*
qed

lemma *finfun-apply-def*: *op* \$ $= (\lambda f \ a. \ finfun\text{-}rec \ (\lambda b. \ b) \ (\lambda a' \ b \ c. \ if \ (a = a') \ then \ b \ else \ c) \ f)$
proof(*rule finfun-rec-unique*)
fix *c* **show** *op* \$ $(K\$ \ c) = (\lambda a. \ c)$ **by**(*simp add*: *finfun-const.rep-eq*)
next
fix *g a b* **show** *op* \$ $g(a \ \$:= \ b) = (\lambda c. \ if \ c = a \ then \ b \ else \ g \ \$ \ c)$
by(*auto simp add*: *finfun-update-def fun-upd-finfun Abs-finfun-inverse finfun-apply*)
qed *auto*

lemma *finfun-upd-apply*: $f(a \ \$:= \ b) \ \$ \ a' = (if \ a = a' \ then \ b \ else \ f \ \$ \ a')$
and *finfun-upd-apply-code* [*code*]: $(finfun\text{-}update\text{-}code \ f \ a \ b) \ \$ \ a' = (if \ a = a' \ then \ b \ else \ f \ \$ \ a')$
by(*simp-all add*: *finfun-apply-def*)

lemma *finfun-const-apply* [*simp*, *code*]: $(K\$ \ b) \ \$ \ a = b$
by(*simp add*: *finfun-apply-def*)

lemma *finfun-upd-apply-same* [*simp*]:
$f(a \ \$:= \ b) \ \$ \ a = b$
by(*simp add*: *finfun-upd-apply*)

lemma *finfun-upd-apply-other* [*simp*]:
$a \neq a' \Longrightarrow f(a \ \$:= \ b) \ \$ \ a' = f \ \$ \ a'$
by(*simp add*: *finfun-upd-apply*)

lemma *finfun-ext*: $(\bigwedge a. \ f \ \$ \ a = g \ \$ \ a) \Longrightarrow f = g$
by(*auto simp add*: *finfun-apply-inject*[*symmetric*])

lemma *expand-finfun-eq*: $(f = g) = (op \ \$ \ f = op \ \$ \ g)$
by(*auto intro*: *finfun-ext*)

lemma *finfun-upd-triv* [*simp*]: $f(x \ \$:= \ f \ \$ \ x) = f$
by(*simp add*: *expand-finfun-eq fun-eq-iff finfun-upd-apply*)

lemma *finfun-const-inject* [*simp*]: $(K\$ \ b) = (K\$ \ b') \equiv b = b'$
by(*simp add*: *expand-finfun-eq fun-eq-iff*)

lemma *finfun-const-eq-update*:
$((K\$ \ b) = f(a \ \$:= \ b')) = (b = b' \wedge (\forall a'. \ a \neq a' \longrightarrow f \ \$ \ a' = b))$
by(*auto simp add*: *expand-finfun-eq fun-eq-iff finfun-upd-apply*)

37.11 Function composition

definition *finfun-comp* :: ('a ⇒ 'b) ⇒ 'c ⇒f 'a ⇒ 'c ⇒f 'b (**infixr** ∘$ *55*)
where [*code del*]: *g* ∘$ *f* = *finfun-rec* (λ*b*. (*K$ g b*)) (λ*a b c*. *c*(*a* $:= *g b*)) *f*

notation (*ASCII*)
finfun-comp (**infixr** *o$ 55*)

interpretation *finfun-comp-aux*: *finfun-rec-wf-aux* (λ*b*. (*K$ g b*)) (λ*a b c*. *c*(*a* $:= *g b*))
by(*unfold-locales*)(*auto simp add*: *finfun-upd-apply intro*: *finfun-ext*)

interpretation *finfun-comp*: *finfun-rec-wf* (λ*b*. (*K$ g b*)) (λ*a b c*. *c*(*a* $:= *g b*))
proof
 fix *b' b* :: *'a*
 assume *fin*: *finite* (*UNIV* :: *'c set*)
 { **fix** *A* :: *'c set*
  from *fin* **have** *finite A* **by**(*auto intro*: *finite-subset*)
  hence *Finite-Set.fold* (λ(*a* :: *'c*) *c*. *c*(*a* $:= *g b'*)) (*K$ g b*) *A* =
   Abs-finfun (λ*a*. *if a* ∈ *A then g b' else g b*)
   by *induct* (*simp-all add*: *finfun-const-def*, *auto simp add*: *finfun-update-def Abs-finfun-inverse-finite fun-upd-def Abs-finfun-inject-finite fun-eq-iff fin*) }
 from *this*[*of UNIV*] **show** *Finite-Set.fold* (λ(*a* :: *'c*) *c*. *c*(*a* $:= *g b'*)) (*K$ g b*) *UNIV* = (*K$ g b'*)
  by(*simp add*: *finfun-const-def*)
qed

lemma *finfun-comp-const* [*simp*, *code*]:
 g ∘$ (*K$ c*) = (*K$ g c*)
by(*simp add*: *finfun-comp-def*)

lemma *finfun-comp-update* [*simp*]: *g* ∘$ (*f*(*a* $:= *b*)) = (*g* ∘$ *f*)(*a* $:= *g b*)
 and *finfun-comp-update-code* [*code*]:
 g ∘$ (*finfun-update-code f a b*) = *finfun-update-code* (*g* ∘$ *f*) *a* (*g b*)
by(*simp-all add*: *finfun-comp-def*)

lemma *finfun-comp-apply* [*simp*]:
 op $ (*g* ∘$ *f*) = *g* ∘ *op* $ *f*
by(*induct f rule*: *finfun-weak-induct*)(*auto simp add*: *finfun-upd-apply*)

lemma *finfun-comp-comp-collapse* [*simp*]: *f* ∘$ *g* ∘$ *h* = (*f* ∘ *g*) ∘$ *h*
by(*induct h rule*: *finfun-weak-induct*) *simp-all*

lemma *finfun-comp-const1* [*simp*]: (λ*x*. *c*) ∘$ *f* = (*K$ c*)
by(*induct f rule*: *finfun-weak-induct*)(*auto intro*: *finfun-ext simp add*: *finfun-upd-apply*)

lemma *finfun-comp-id1* [*simp*]: (λ*x*. *x*) ∘$ *f* = *f id* ∘$ *f* = *f*
by(*induct f rule*: *finfun-weak-induct*) *auto*

lemma *finfun-comp-conv-comp*: *g* ∘$ *f* = *Abs-finfun* (*g* ∘ *op* $ *f*)

including *finfun*
proof –
have ($\lambda f.\ g \circ\$\ f$) = ($\lambda f.$ *Abs-finfun* ($g \circ op\ \$\ f$))
proof(*rule finfun-rec-unique*)
{ **fix** *c* **show** *Abs-finfun* ($g \circ op\ \$\ (K\$\ c)$) = ($K\$\ g\ c$)
by(*simp add*: *finfun-comp-def o-def*)(*simp add*: *finfun-const-def*) }
{ **fix** g' *a b* **show** *Abs-finfun* ($g \circ op\ \$\ g'(a\ \$:= b)$) = (*Abs-finfun* ($g \circ op\ \$\ g'$))($a\ \$:= g\ b$)
proof –
obtain *y* **where** *y*: $y \in$ *finfun* **and** g': $g' =$ *Abs-finfun y* **by**(*cases* g')
moreover from g' **have** ($g \circ op\ \$\ g'$) $\in$ *finfun* **by**(*simp add*: *finfun-left-compose*)
moreover have $g \circ y(a := b) = (g \circ y)(a := g\ b)$ **by**(*auto*)
ultimately show *?thesis* **by**(*simp add*: *finfun-comp-def finfun-update-def*)
qed }
qed *auto*
thus *?thesis* **by**(*auto simp add*: *fun-eq-iff*)
qed

definition *finfun-comp2* :: $'b \Rightarrow f\ 'c \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow f\ 'c$ (**infixr** $\$\circ$ *55*)
where [*code del*]: $g\ \$\circ\ f$ = *Abs-finfun* ($op\ \$\ g \circ f$)

notation (*ASCII*)
finfun-comp2 (**infixr** $\$o$ *55*)

lemma *finfun-comp2-const* [*code*, *simp*]: *finfun-comp2* ($K\$\ c$) f = ($K\$\ c$)
including *finfun*
by(*simp add*: *finfun-comp2-def finfun-const-def comp-def*)

lemma *finfun-comp2-update*:
includes *finfun*
assumes *inj*: *inj f*
shows *finfun-comp2* ($g(b\ \$:= c)$) f = (*if* $b \in$ *range f then* (*finfun-comp2 g f*)(*inv f b* $\$:= c$) *else finfun-comp2 g f*)
proof(*cases* $b \in$ *range f*)
case *True*
from *inj* **have** $\bigwedge x.\ (op\ \$\ g)(f\ x := c) \circ f = (op\ \$\ g \circ f)(x := c)$ **by**(*auto intro*!: *ext dest*: *injD*)
with *inj True* **show** *?thesis* **by**(*auto simp add*: *finfun-comp2-def finfun-update-def finfun-right-compose*)
next
case *False*
hence $(op\ \$\ g)(b := c) \circ f = op\ \$\ g \circ f$ **by**(*auto simp add*: *fun-eq-iff*)
with *False* **show** *?thesis* **by**(*auto simp add*: *finfun-comp2-def finfun-update-def*)
qed

37.12 Universal quantification

definition *finfun-All-except* :: $'a\ list \Rightarrow 'a \Rightarrow f\ bool \Rightarrow bool$
where [*code del*]: *finfun-All-except A P* $\equiv \forall a.\ a \in set\ A \vee P\ \$\ a$

lemma *finfun-All-except-const*: *finfun-All-except A* (*K*\$ *b*) ⟷ *b* ∨ *set A* = *UNIV*
by(*auto simp add*: *finfun-All-except-def*)

lemma *finfun-All-except-const-finfun-UNIV-code* [*code*]:
finfun-All-except A (*K*\$ *b*) = (*b* ∨ *is-list-UNIV A*)
by(*simp add*: *finfun-All-except-const is-list-UNIV-iff*)

lemma *finfun-All-except-update*:
finfun-All-except A f(*a* \$:= *b*) = ((*a* ∈ *set A* ∨ *b*) ∧ *finfun-All-except* (*a* # *A*) *f*)
by(*fastforce simp add*: *finfun-All-except-def finfun-upd-apply*)

lemma *finfun-All-except-update-code* [*code*]:
fixes *a* :: *'a* :: *card-UNIV*
shows *finfun-All-except A* (*finfun-update-code f a b*) = ((*a* ∈ *set A* ∨ *b*) ∧ *finfun-All-except* (*a* # *A*) *f*)
by(*simp add*: *finfun-All-except-update*)

definition *finfun-All* :: *'a* ⇒*f bool* ⇒ *bool*
where *finfun-All* = *finfun-All-except* []

lemma *finfun-All-const* [*simp*]: *finfun-All* (*K*\$ *b*) = *b*
by(*simp add*: *finfun-All-def finfun-All-except-def*)

lemma *finfun-All-update*: *finfun-All f*(*a* \$:= *b*) = (*b* ∧ *finfun-All-except* [*a*] *f*)
by(*simp add*: *finfun-All-def finfun-All-except-update*)

lemma *finfun-All-All*: *finfun-All P* = *All* (*op* \$ *P*)
by(*simp add*: *finfun-All-def finfun-All-except-def*)

definition *finfun-Ex* :: *'a* ⇒*f bool* ⇒ *bool*
where *finfun-Ex P* = *Not* (*finfun-All* (*Not* ∘\$ *P*))

lemma *finfun-Ex-Ex*: *finfun-Ex P* = *Ex* (*op* \$ *P*)
unfolding *finfun-Ex-def finfun-All-All* **by** *simp*

lemma *finfun-Ex-const* [*simp*]: *finfun-Ex* (*K*\$ *b*) = *b*
by(*simp add*: *finfun-Ex-def*)

37.13 A diagonal operator for FinFuns

definition *finfun-Diag* :: *'a* ⇒*f 'b* ⇒ *'a* ⇒*f 'c* ⇒ *'a* ⇒*f* (*'b* × *'c*) ((*1'*(\$-,/ -\$')) [*0*, *0*] *1000*)
where [*code del*]: (\$*f*, *g*\$) = *finfun-rec* (λ*b*. *Pair b* ∘\$ *g*) (λ*a b c*. *c*(*a* \$:= (*b*, *g* \$ *a*))) *f*

interpretation *finfun-Diag-aux*: *finfun-rec-wf-aux* λ*b*. *Pair b* ∘\$ *g* λ*a b c*. *c*(*a* \$:=

(*b*, *g* $ *a*))
by(*unfold-locales*)(*simp-all add*: *expand-finfun-eq fun-eq-iff finfun-upd-apply*)

interpretation *finfun-Diag*: *finfun-rec-wf* λ*b*. *Pair b* ∘$ *g* λ*a b c*. *c*(*a* $:= (*b*, *g* $ *a*))
proof
  fix *b′ b* :: ′*a*
  assume *fin*: *finite* (*UNIV* :: ′*c set*)
  { fix *A* :: ′*c set*
   interpret *comp-fun-commute* λ*a c*. *c*(*a* $:= (*b′*, *g* $ *a*)) **by**(*rule finfun-Diag-aux.upd-left-comm*)
    from *fin* **have** *finite A* **by**(*auto intro*: *finite-subset*)
    hence *Finite-Set.fold* (λ*a c*. *c*(*a* $:= (*b′*, *g* $ *a*))) (*Pair b* ∘$ *g*) *A* =
      Abs-finfun (λ*a*. (*if a* ∈ *A then b′ else b*, *g* $ *a*))
      by(*induct*)(*simp-all add*: *finfun-const-def finfun-comp-conv-comp o-def*,
            auto simp add: *finfun-update-def Abs-finfun-inverse-finite fun-upd-def Abs-finfun-inject-finite fun-eq-iff fin*) **}**
  from *this*[*of UNIV*] **show** *Finite-Set.fold* (λ*a c*. *c*(*a* $:= (*b′*, *g* $ *a*))) (*Pair b* ∘$ *g*) *UNIV* = *Pair b′* ∘$ *g*
    by(*simp add*: *finfun-const-def finfun-comp-conv-comp o-def*)
qed

lemma *finfun-Diag-const1*: ($*K*$ *b*, *g*$) = *Pair b* ∘$ *g*
by(*simp add*: *finfun-Diag-def*)

Do not use ($*K*$ *?b*, *?g*$) = *Pair ?b* ∘$ *?g* for the code generator because *Pair b* is injective, i.e. if *g* is free of redundant updates, there is no need to check for redundant updates as is done for *op* ∘$.

lemma *finfun-Diag-const-code* [*code*]:
  ($*K*$ *b*, *K*$ *c*$) = (*K*$ (*b*, *c*))
  ($*K*$ *b*, *finfun-update-code g a c*$) = *finfun-update-code* ($*K*$ *b*, *g*$) *a* (*b*, *c*)
by(*simp-all add*: *finfun-Diag-const1*)

lemma *finfun-Diag-update1*: ($*f*(*a* $:= *b*), *g*$) = ($*f*, *g*$)(*a* $:= (*b*, *g* $ *a*))
  and *finfun-Diag-update1-code* [*code*]: ($*finfun-update-code f a b*, *g*$) = ($*f*, *g*$)(*a* $:= (*b*, *g* $ *a*))
by(*simp-all add*: *finfun-Diag-def*)

lemma *finfun-Diag-const2*: ($*f*, *K*$ *c*$) = (λ*b*. (*b*, *c*)) ∘$ *f*
by(*induct f rule*: *finfun-weak-induct*)(*auto intro*!: *finfun-ext simp add*: *finfun-upd-apply finfun-Diag-const1 finfun-Diag-update1*)

lemma *finfun-Diag-update2*: ($*f*, *g*(*a* $:= *c*)$) = ($*f*, *g*$)(*a* $:= (*f* $ *a*, *c*))
by(*induct f rule*: *finfun-weak-induct*)(*auto intro*!: *finfun-ext simp add*: *finfun-upd-apply finfun-Diag-const1 finfun-Diag-update1*)

lemma *finfun-Diag-const-const* [*simp*]: ($*K*$ *b*, *K*$ *c*$) = (*K*$ (*b*, *c*))
by(*simp add*: *finfun-Diag-const1*)

lemma *finfun-Diag-const-update*:

(K b, g(a $:= c)$) = (K b, g$)(a $:= (b, c))
by(*simp add*: *finfun-Diag-const1*)

lemma *finfun-Diag-update-const*:
($f(a $:= b), K$ c$) = ($f, K$ c$)(a $:= (b, c))
by(*simp add*: *finfun-Diag-def*)

lemma *finfun-Diag-update-update*:
($f(a $:= b), g(a′ $:= c)$) = (*if* a = a′ *then* (f, g)(a $:= (b, c)) *else* ($f, g$)(a $:= (b, g $ a))(a′ $:= (f $ a′, c)))
by(*auto simp add*: *finfun-Diag-update1 finfun-Diag-update2*)

lemma *finfun-Diag-apply* [*simp*]: *op* $ (f, g) = (λx. (f $ x, g $ x))
by(*induct f rule*: *finfun-weak-induct*)(*auto simp add*: *finfun-Diag-const1 finfun-Diag-update1 finfun-upd-apply*)

lemma *finfun-Diag-conv-Abs-finfun*:
(f, g) = *Abs-finfun* ((λx. (f $ x, g $ x)))
including *finfun*
proof −
have (λf :: ′a ⇒f ′b. (f, g)) = (λf. *Abs-finfun* ((λx. (f $ x, g $ x))))
proof(*rule finfun-rec-unique*)
{ fix c **show** *Abs-finfun* (λx. ((K$ c) $ x, g $ x)) = *Pair* c ∘$ g
by(*simp add*: *finfun-comp-conv-comp o-def finfun-const-def*) **}**
{ fix g′ a b
show *Abs-finfun* (λx. (g′(a $:= b) $ x, g $ x)) =
(*Abs-finfun* (λx. (g′ $ x, g $ x)))(a $:= (b, g $ a))
by(*auto simp add*: *finfun-update-def fun-eq-iff simp del*: *fun-upd-apply*) *simp*
}
qed(*simp-all add*: *finfun-Diag-const1 finfun-Diag-update1*)
thus *?thesis* **by**(*auto simp add*: *fun-eq-iff*)
qed

lemma *finfun-Diag-eq*: (f, g) = ($f′, g′$) ⟷ f = f′ ∧ g = g′
by(*auto simp add*: *expand-finfun-eq fun-eq-iff*)

definition *finfun-fst* :: ′a ⇒f (′b × ′c) ⇒ ′a ⇒f ′b
where [*code*]: *finfun-fst* f = *fst* ∘$ f

lemma *finfun-fst-const*: *finfun-fst* (K$ bc) = (K$ *fst* bc)
by(*simp add*: *finfun-fst-def*)

lemma *finfun-fst-update*: *finfun-fst* (f(a $:= bc)) = (*finfun-fst* f)(a $:= *fst* bc)
and *finfun-fst-update-code*: *finfun-fst* (*finfun-update-code* f a bc) = (*finfun-fst* f)(a $:= *fst* bc)
by(*simp-all add*: *finfun-fst-def*)

lemma *finfun-fst-comp-conv*: *finfun-fst* (f ∘$ g) = (*fst* ∘ f) ∘$ g
by(*simp add*: *finfun-fst-def*)

lemma *finfun-fst-conv* [*simp*]: *finfun-fst* (\$*f*, *g*\$) = *f*
by(*induct f rule*: *finfun-weak-induct*)(*simp-all add*: *finfun-Diag-const1 finfun-fst-comp-conv o-def finfun-Diag-update1 finfun-fst-update*)

lemma *finfun-fst-conv-Abs-finfun*: *finfun-fst* = (λ*f*. *Abs-finfun* (*fst* ∘ *op* \$ *f*))
by(*simp add*: *finfun-fst-def* [*abs-def*] *finfun-comp-conv-comp*)

definition *finfun-snd* :: ′*a* ⇒*f* (′*b* × ′*c*) ⇒ ′*a* ⇒*f* ′*c*
where [*code*]: *finfun-snd f* = *snd* ∘\$ *f*

lemma *finfun-snd-const*: *finfun-snd* (*K*\$ *bc*) = (*K*\$ *snd bc*)
by(*simp add*: *finfun-snd-def*)

lemma *finfun-snd-update*: *finfun-snd* (*f*(*a* \$:= *bc*)) = (*finfun-snd f*)(*a* \$:= *snd bc*)
and *finfun-snd-update-code* [*code*]: *finfun-snd* (*finfun-update-code f a bc*) = (*finfun-snd f*)(*a* \$:= *snd bc*)
by(*simp-all add*: *finfun-snd-def*)

lemma *finfun-snd-comp-conv*: *finfun-snd* (*f* ∘\$ *g*) = (*snd* ∘ *f*) ∘\$ *g*
by(*simp add*: *finfun-snd-def*)

lemma *finfun-snd-conv* [*simp*]: *finfun-snd* (\$*f*, *g*\$) = *g*
apply(*induct f rule*: *finfun-weak-induct*)
apply(*auto simp add*: *finfun-Diag-const1 finfun-snd-comp-conv o-def finfun-Diag-update1 finfun-snd-update finfun-upd-apply intro*: *finfun-ext*)
done

lemma *finfun-snd-conv-Abs-finfun*: *finfun-snd* = (λ*f*. *Abs-finfun* (*snd* ∘ *op* \$ *f*))
by(*simp add*: *finfun-snd-def* [*abs-def*] *finfun-comp-conv-comp*)

lemma *finfun-Diag-collapse* [*simp*]: (\$*finfun-fst f*, *finfun-snd f*\$) = *f*
by(*induct f rule*: *finfun-weak-induct*)(*simp-all add*: *finfun-fst-const finfun-snd-const finfun-fst-update finfun-snd-update finfun-Diag-update-update*)

37.14 Currying for FinFuns

definition *finfun-curry* :: (′*a* × ′*b*) ⇒*f* ′*c* ⇒ ′*a* ⇒*f* ′*b* ⇒*f* ′*c*
where [*code del*]: *finfun-curry* = *finfun-rec* (*finfun-const* ∘ *finfun-const*) (λ(*a*, *b*) *c f*. *f*(*a* \$:= (*f* \$ *a*)(*b* \$:= *c*)))

interpretation *finfun-curry-aux*: *finfun-rec-wf-aux finfun-const* ∘ *finfun-const* λ(*a*, *b*) *c f*. *f*(*a* \$:= (*f* \$ *a*)(*b* \$:= *c*))
apply(*unfold-locales*)
apply(*auto simp add*: *split-def finfun-update-twist finfun-upd-apply split-paired-all finfun-update-const-same*)
done

interpretation *finfun-curry*: *finfun-rec-wf finfun-const* $\circ$ *finfun-const* $\lambda(a,\ b)\ c\ f.\ f(a\ \$:=\ (f\ \$\ a)(b\ \$:=\ c))$
proof(*unfold-locales*)
fix $b'\ b$ :: $'b$
assume *fin*: *finite* (*UNIV* :: ($'c \times 'a$) *set*)
hence *fin1*: *finite* (*UNIV* :: $'c$ *set*) **and** *fin2*: *finite* (*UNIV* :: $'a$ *set*)
unfolding *UNIV-Times-UNIV*[*symmetric*]
by(*fastforce dest*: *finite-cartesian-productD1 finite-cartesian-productD2*)+
note [*simp*] = *Abs-finfun-inverse-finite*[*OF fin*] *Abs-finfun-inverse-finite*[*OF fin1*] *Abs-finfun-inverse-finite*[*OF fin2*]
{ fix *A* :: ($'c \times 'a$) *set*
interpret *comp-fun-commute* λa :: $'c \times 'a$. $(\lambda(a,\ b)\ c\ f.\ f(a\ \$:=\ (f\ \$\ a)(b\ \$:=\ c)))\ a\ b'$
by(*rule finfun-curry-aux.upd-left-comm*)
from *fin* **have** *finite A* **by**(*auto intro*: *finite-subset*)
hence *Finite-Set.fold* (λa :: $'c \times 'a$. $(\lambda(a,\ b)\ c\ f.\ f(a\ \$:=\ (f\ \$\ a)(b\ \$:=\ c)))\ a\ b'$) ((*finfun-const* $\circ$ *finfun-const*) *b*) *A* = *Abs-finfun* (λa. *Abs-finfun* ($\lambda b''$. *if* $(a,\ b'') \in A$ *then* b' *else* *b*))
by *induct* (*simp-all*, *auto simp add*: *finfun-update-def finfun-const-def split-def intro*!: *arg-cong*[**where** *f*=*Abs-finfun*] *ext*) **}**
from *this*[*of UNIV*]
show *Finite-Set.fold* (λa :: $'c \times 'a$. $(\lambda(a,\ b)\ c\ f.\ f(a\ \$:=\ (f\ \$\ a)(b\ \$:=\ c)))\ a\ b'$) ((*finfun-const* $\circ$ *finfun-const*) *b*) *UNIV* = (*finfun-const* $\circ$ *finfun-const*) b'
by(*simp add*: *finfun-const-def*)
qed

lemma *finfun-curry-const* [*simp*, *code*]: *finfun-curry* (*K*\$ *c*) = (*K*\$ *K*\$ *c*)
by(*simp add*: *finfun-curry-def*)

lemma *finfun-curry-update* [*simp*]:
finfun-curry (*f*((*a*, *b*) \$:= *c*)) = (*finfun-curry f*)(*a* \$:= (*finfun-curry f* \$ *a*)(*b* \$:= *c*))
and *finfun-curry-update-code* [*code*]:
finfun-curry (*finfun-update-code f* (*a*, *b*) *c*) = (*finfun-curry f*)(*a* \$:= (*finfun-curry f* \$ *a*)(*b* \$:= *c*))
by(*simp-all add*: *finfun-curry-def*)

lemma *finfun-Abs-finfun-curry*: **assumes** *fin*: $f \in$ *finfun*
shows (λa. *Abs-finfun* (*curry f a*)) $\in$ *finfun*
including *finfun*
proof −
from *fin* **obtain** *c* **where** *c*: *finite* $\{ab.\ f\ ab \neq c\}$ **unfolding** *finfun-def* **by** *blast*
have $\{a.\ \exists b.\ f\ (a,\ b) \neq c\}$ = *fst* ‘ $\{ab.\ f\ ab \neq c\}$ **by**(*force*)
hence $\{a.\ curry\ f\ a \neq (\lambda x.\ c)\}$ = *fst* ‘ $\{ab.\ f\ ab \neq c\}$
by(*auto simp add*: *curry-def fun-eq-iff*)
with *fin c* **have** *finite* {*a*. *Abs-finfun* (*curry f a*) $\neq$ (*K*\$ *c*)}
by(*simp add*: *finfun-const-def finfun-curry*)
thus *?thesis* **unfolding** *finfun-def* **by** *auto*
qed

lemma *finfun-curry-conv-curry*:
  fixes f :: $('a \times 'b) \Rightarrow f\ 'c$
  shows *finfun-curry f* = *Abs-finfun* (λa. *Abs-finfun* (*curry* (*finfun-apply f*) *a*))
  including *finfun*
proof −
  have *finfun-curry* = (λf :: $('a \times 'b) \Rightarrow f\ 'c$. *Abs-finfun* ($\lambda a$. *Abs-finfun* (*curry* (*finfun-apply f*) *a*)))
  proof(*rule finfun-rec-unique*)
    fix *c* **show** *finfun-curry* (*K$ c*) = (*K$ K$ c*) **by** *simp*
    fix *f a*
    show *finfun-curry* (*f*(*a* $:= *c*)) = (*finfun-curry f*)(*fst a* $:= (*finfun-curry f* $ (*fst a*))(*snd a* $:= *c*))
      by(*cases a*) *simp*
    show *Abs-finfun* (λa. *Abs-finfun* (*curry* (*finfun-apply* (*K$ c*)) *a*)) = (*K$ K$ c*)
      by(*simp add*: *finfun-curry-def finfun-const-def curry-def*)
    fix *g b*
    show *Abs-finfun* (λaa. *Abs-finfun* (*curry* (*op* $ *g*(*a* $:= *b*)) *aa*)) =
      (*Abs-finfun* (λa. *Abs-finfun* (*curry* (*op* $ *g*) *a*)))(
       fst a $:= ((*Abs-finfun* ($\lambda a$. *Abs-finfun* (*curry* (*op* $ *g*) *a*))) $ (*fst a*))(*snd a* $:= *b*))
    by(*cases a*)(*auto intro*!: *ext arg-cong*[**where** *f*=*Abs-finfun*] *simp add*: *finfun-curry-def finfun-update-def finfun-Abs-finfun-curry*)
  qed
  thus *?thesis* **by**(*auto simp add*: *fun-eq-iff*)
qed

37.15 Executable equality for FinFuns

lemma *eq-finfun-All-ext*: ($f = g$) $\longleftrightarrow$ *finfun-All* (($\lambda(x, y)$. $x = y$) ∘$ (f, g))
by(*simp add*: *expand-finfun-eq fun-eq-iff finfun-All-All o-def*)

instantiation *finfun* :: ({*card-UNIV*,*equal*},*equal*) *equal* **begin**
definition *eq-finfun-def* [*code*]: *HOL.equal f g* $\longleftrightarrow$ *finfun-All* (($\lambda(x, y)$. $x = y$) ∘$ (f, g))
instance by(*intro-classes*)(*simp add*: *eq-finfun-All-ext eq-finfun-def*)
end

lemma [*code nbe*]:
  HOL.equal (f :: - $\Rightarrow f$ -) f $\longleftrightarrow$ *True*
  by (*fact equal-refl*)

37.16 An operator that explicitly removes all redundant updates in the generated representations

definition *finfun-clearjunk* :: $'a \Rightarrow f\ 'b \Rightarrow 'a \Rightarrow f\ 'b$
where [*simp*, *code del*]: *finfun-clearjunk* = *id*

lemma *finfun-clearjunk-const* [*code*]: *finfun-clearjunk* (*K$ b*) = (*K$ b*)
by *simp*

lemma *finfun-clearjunk-update* [*code*]:
finfun-clearjunk (*finfun-update-code f a b*) = *f*(*a* $:= *b*)
by *simp*

37.17 The domain of a FinFun as a FinFun

definition *finfun-dom* :: (*'a* ⇒*f 'b*) ⇒ (*'a* ⇒*f bool*)
where [*code del*]: *finfun-dom f* = *Abs-finfun* (λ*a*. *f* $ *a* ≠ *finfun-default f*)

lemma *finfun-dom-const*:
finfun-dom ((*K$ c*) :: *'a* ⇒*f 'b*) = (*K$ finite* (*UNIV* :: *'a set*) ∧ *c* ≠ *undefined*)
unfolding *finfun-dom-def finfun-default-const*
by(*auto*)(*simp-all add*: *finfun-const-def*)

finfun-dom raises an exception when called on a FinFun whose domain is a finite type. For such FinFuns, the default value (and as such the domain) is undefined.

lemma *finfun-dom-const-code* [*code*]:
finfun-dom ((*K$ c*) :: (*'a* :: *card-UNIV*) ⇒*f 'b*) =
(*if CARD*(*'a*) = *0 then* (*K$ False*) *else Code.abort* (*STR ''finfun-dom called on finite type''*) (λ-. *finfun-dom* (*K$ c*)))
by(*simp add*: *finfun-dom-const card-UNIV card-eq-0-iff*)

lemma *finfun-dom-finfunI*: (λ*a*. *f* $ *a* ≠ *finfun-default f*) ∈ *finfun*
using *finite-finfun-default*[*of f*]
by(*simp add*: *finfun-def exI*[**where** *x*=*False*])

lemma *finfun-dom-update* [*simp*]:
finfun-dom (*f*(*a* $:= *b*)) = (*finfun-dom f*)(*a* $:= (*b* ≠ *finfun-default f*))
including *finfun* **unfolding** *finfun-dom-def finfun-update-def*
apply(*simp add*: *finfun-default-update-const finfun-dom-finfunI*)
apply(*fold finfun-update.rep-eq*)
apply(*simp add*: *finfun-upd-apply fun-eq-iff fun-upd-def finfun-default-update-const*)
done

lemma *finfun-dom-update-code* [*code*]:
finfun-dom (*finfun-update-code f a b*) = *finfun-update-code* (*finfun-dom f*) *a* (*b* ≠ *finfun-default f*)
by(*simp*)

lemma *finite-finfun-dom*: *finite* {*x*. *finfun-dom f* $ *x*}
proof(*induct f rule*: *finfun-weak-induct*)
case (*const b*)
thus *?case*
by (*cases finite* (*UNIV* :: *'a set*) ∧ *b* ≠ *undefined*)
(*auto simp add*: *finfun-dom-const UNIV-def* [*symmetric*] *Set.empty-def* [*symmetric*])

next
case (*update f a b*)
have {*x. finfun-dom f*(*a* $:= *b*) $ *x*} =
(*if b = finfun-default f then* {*x. finfun-dom f* $ *x*} − {*a*} *else insert a* {*x. finfun-dom f* $ *x*})
by (*auto simp add*: *finfun-upd-apply split*: *if-split-asm*)
thus *?case* **using** *update* **by** *simp*
qed

37.18 The domain of a FinFun as a sorted list

definition *finfun-to-list* :: (*'a* :: *linorder*) ⇒*f 'b* ⇒ *'a list*
where
finfun-to-list f = (*THE xs. set xs* = {*x. finfun-dom f* $ *x*} ∧ *sorted xs* ∧ *distinct xs*)

lemma *set-finfun-to-list* [*simp*]: *set* (*finfun-to-list f*) = {*x. finfun-dom f* $ *x*} (**is** *?thesis1*)
and *sorted-finfun-to-list*: *sorted* (*finfun-to-list f*) (**is** *?thesis2*)
and *distinct-finfun-to-list*: *distinct* (*finfun-to-list f*) (**is** *?thesis3*)
proof (*atomize* (*full*))
show *?thesis1* ∧ *?thesis2* ∧ *?thesis3*
unfolding *finfun-to-list-def*
by(*rule theI'*)(*rule finite-sorted-distinct-unique finite-finfun-dom*)+
qed

lemma *finfun-const-False-conv-bot*: *op* $ (*K*$ *False*) = *bot*
by *auto*

lemma *finfun-const-True-conv-top*: *op* $ (*K*$ *True*) = *top*
by *auto*

lemma *finfun-to-list-const*:
finfun-to-list ((*K*$ *c*) :: (*'a* :: {*linorder*} ⇒*f 'b*)) =
(*if* ¬ *finite* (*UNIV* :: *'a set*) ∨ *c* = *undefined then* [] *else THE xs. set xs* = *UNIV* ∧ *sorted xs* ∧ *distinct xs*)
by(*auto simp add*: *finfun-to-list-def finfun-const-False-conv-bot finfun-const-True-conv-top finfun-dom-const*)

lemma *finfun-to-list-const-code* [*code*]:
finfun-to-list ((*K*$ *c*) :: (*'a* :: {*linorder*, *card-UNIV*} ⇒*f 'b*)) =
(*if CARD*(*'a*) = *0 then* [] *else Code.abort* (*STR ''finfun-to-list called on finite type''*) (λ-. *finfun-to-list* ((*K*$ *c*) :: (*'a* ⇒*f 'b*))))
by(*auto simp add*: *finfun-to-list-const card-UNIV card-eq-0-iff*)

lemma *remove1-insort-insert-same*:
x ∉ *set xs* ⟹ *remove1 x* (*insort-insert x xs*) = *xs*
by (*metis insort-insert-insort remove1-insort*)

lemma *finfun-dom-conv*:
finfun-dom f $ x $\longleftrightarrow$ *f $ x* $\neq$ *finfun-default f*
by(*induct f rule*: *finfun-weak-induct*)(*auto simp add*: *finfun-dom-const finfun-default-const finfun-default-update-const finfun-upd-apply*)

lemma *finfun-to-list-update*:
finfun-to-list (*f*(*a* $:= *b*)) =
(*if b* = *finfun-default f then List.remove1 a* (*finfun-to-list f*) *else List.insort-insert a* (*finfun-to-list f*))
proof(*subst finfun-to-list-def*, *rule the-equality*)
fix *xs*
assume *set xs* = {*x*. *finfun-dom f*(*a* $:= *b*) $ *x*} ∧ *sorted xs* ∧ *distinct xs*
hence *eq*: *set xs* = {*x*. *finfun-dom f*(*a* $:= *b*) $ *x*}
and [*simp*]: *sorted xs distinct xs* **by** *simp-all*
show *xs* = (*if b* = *finfun-default f then remove1 a* (*finfun-to-list f*) *else insort-insert a* (*finfun-to-list f*))
proof(*cases b* = *finfun-default f*)
case [*simp*]: *True*
show *?thesis*
proof(*cases finfun-dom f* $ *a*)
case *True*
have *finfun-to-list f* = *insort-insert a xs*
unfolding *finfun-to-list-def*
proof(*rule the-equality*)
have *set* (*insort-insert a xs*) = *insert a* (*set xs*) **by**(*simp add*: *set-insort-insert*)
also note *eq* **also**
have *insert a* {*x*. *finfun-dom f*(*a* $:= *b*) $ *x*} = {*x*. *finfun-dom f* $ *x*} **using** *True*
by(*auto simp add*: *finfun-upd-apply split*: *if-split-asm*)
finally show *1*: *set* (*insort-insert a xs*) = {*x*. *finfun-dom f* $ *x*} ∧ *sorted* (*insort-insert a xs*) ∧ *distinct* (*insort-insert a xs*)
by(*simp add*: *sorted-insort-insert distinct-insort-insert*)

fix *xs'*
assume *set xs'* = {*x*. *finfun-dom f* $ *x*} ∧ *sorted xs'* ∧ *distinct xs'*
thus *xs'* = *insort-insert a xs* **using** *1* **by**(*auto dest*: *sorted-distinct-set-unique*)
qed
with *eq True* **show** *?thesis* **by**(*simp add*: *remove1-insort-insert-same*)
next
case *False*
hence *f* $ *a* = *b* **by**(*auto simp add*: *finfun-dom-conv*)
hence *f*: *f*(*a* $:= *b*) = *f* **by**(*simp add*: *expand-finfun-eq fun-eq-iff finfun-upd-apply*)
from *eq* **have** *finfun-to-list f* = *xs* **unfolding** *f finfun-to-list-def*
by(*auto elim*: *sorted-distinct-set-unique intro*!: *the-equality*)
with *eq False* **show** *?thesis* **unfolding** *f* **by**(*simp add*: *remove1-idem*)
qed
next
case *False*
show *?thesis*

proof(*cases finfun-dom f $ a*)
case *True*
have *finfun-to-list f = xs*
unfolding *finfun-to-list-def*
proof(*rule the-equality*)
have *finfun-dom f = finfun-dom f*(*a $:= b*) **using** *False True*
by(*simp add*: *expand-finfun-eq fun-eq-iff finfun-upd-apply*)
with *eq* **show** *1*: *set xs* = {*x*. *finfun-dom f $ x*} ∧ *sorted xs* ∧ *distinct xs*
by(*simp del*: *finfun-dom-update*)

fix *xs'*
assume *set xs'* = {*x*. *finfun-dom f $ x*} ∧ *sorted xs'* ∧ *distinct xs'*
thus *xs'* = *xs* **using** *1* **by**(*auto elim*: *sorted-distinct-set-unique*)
qed
thus *?thesis* **using** *False True eq* **by**(*simp add*: *insort-insert-triv*)
next
case *False*
have *finfun-to-list f = remove1 a xs*
unfolding *finfun-to-list-def*
proof(*rule the-equality*)
have *set* (*remove1 a xs*) = *set xs* − {*a*} **by** *simp*
also note *eq* **also**
have {*x*. *finfun-dom f*(*a $:= b*) *$ x*} − {*a*} = {*x*. *finfun-dom f $ x*} **using** *False*
by(*auto simp add*: *finfun-upd-apply split*: *if-split-asm*)
finally show *1*: *set* (*remove1 a xs*) = {*x*. *finfun-dom f $ x*} ∧ *sorted* (*remove1 a xs*) ∧ *distinct* (*remove1 a xs*)
by(*simp add*: *sorted-remove1*)

fix *xs'*
assume *set xs'* = {*x*. *finfun-dom f $ x*} ∧ *sorted xs'* ∧ *distinct xs'*
thus *xs'* = *remove1 a xs* **using** *1* **by**(*blast intro*: *sorted-distinct-set-unique*)
qed
thus *?thesis* **using** *False eq* ‹*b* ≠ *finfun-default f*›
by (*simp add*: *insort-insert-insort insort-remove1*)
qed
qed
qed (*auto simp add*: *distinct-finfun-to-list sorted-finfun-to-list sorted-remove1 set-insort-insert sorted-insort-insert distinct-insort-insert finfun-upd-apply split*: *if-split-asm*)

lemma *finfun-to-list-update-code* [*code*]:
finfun-to-list (*finfun-update-code f a b*) =
(*if b* = *finfun-default f then List.remove1 a* (*finfun-to-list f*) *else List.insort-insert a* (*finfun-to-list f*))
by(*simp add*: *finfun-to-list-update*)

More type class instantiations

lemma *card-eq-1-iff*: *card A* = *1* ⟷ *A* ≠ {} ∧ (∀ *x*∈*A*. ∀ *y*∈*A*. *x* = *y*)
(**is** *?lhs* ⟷ *?rhs*)

proof
  assume *?lhs*
  moreover **{**
    fix *x y*
    assume *A*: $x \in A$ $y \in A$ **and** *neq*: $x \neq y$
    have *finite A* **using** ‹*?lhs*› **by**(*simp add*: *card-ge-0-finite*)
    from *neq* **have** *2* = *card* {*x*, *y*} **by** *simp*
    also have ... ≤ *card A* **using** *A* ‹*finite A*›
      by(*auto intro*: *card-mono*)
    finally have *False* **using** ‹*?lhs*› **by** *simp* **}**
  ultimately show *?rhs* **by** *auto*
next
  assume *?rhs*
  hence *A* = {*THE x*. *x* ∈ *A*}
    by *safe* (*auto intro*: *theI the-equality*[*symmetric*])
  also have *card* ... = *1* **by** *simp*
  finally show *?lhs* **.**
qed

lemma *card-UNIV-finfun*:
  defines *F* == *finfun* :: (*'a* ⇒ *'b*) *set*
  shows *CARD*(*'a* ⇒*f* *'b*) = (*if CARD*(*'a*) ≠ *0* ∧ *CARD*(*'b*) ≠ *0* ∨ *CARD*(*'b*)
= *1 then CARD*(*'b*) ^ *CARD*(*'a*) *else 0*)
proof(*cases 0* < *CARD*(*'a*) ∧ *0* < *CARD*(*'b*) ∨ *CARD*(*'b*) = *1*)
  case *True*
  from *True* **have** *F* = *UNIV*
  proof
    assume *b*: *CARD*(*'b*) = *1*
    hence ∀ *x* :: *'b*. *x* = *undefined*
      by(*auto simp add*: *card-eq-1-iff simp del*: *One-nat-def*)
   thus *?thesis* **by**(*auto simp add*: *finfun-def F-def intro*: *exI*[**where** *x*=*undefined*])
 qed(*auto simp add*: *finfun-def card-gt-0-iff F-def intro*: *finite-subset*[**where** *B*=*UNIV*])
  moreover have *CARD*(*'a* ⇒*f* *'b*) = *card F*
    unfolding *type-definition.Abs-image*[*OF type-definition-finfun*, *symmetric*]
    by(*auto intro*!: *card-image inj-onI simp add*: *Abs-finfun-inject F-def*)
  ultimately show *?thesis* **by**(*simp add*: *card-fun*)
next
  case *False*
  hence *infinite*: ¬ (*finite* (*UNIV* :: *'a set*) ∧ *finite* (*UNIV* :: *'b set*))
    and *b*: *CARD*(*'b*) ≠ *1* **by**(*simp-all add*: *card-eq-0-iff*)
  from *b* **obtain** *b1 b2* :: *'b* **where** *b1* ≠ *b2*
    by(*auto simp add*: *card-eq-1-iff simp del*: *One-nat-def*)
  let *?f* = λ*a a'* :: *'a*. *if a* = *a' then b1 else b2*
  from *infinite* **have** ¬ *finite* (*UNIV* :: (*'a* ⇒*f* *'b*) *set*)
  proof(*rule contrapos-nn*[*OF - conjI*])
    assume *finite*: *finite* (*UNIV* :: (*'a* ⇒*f* *'b*) *set*)
    hence *finite F*
      unfolding *type-definition.Abs-image*[*OF type-definition-finfun*, *symmetric*]
F-def

by(*rule finite-imageD*)(*auto intro*: *inj-onI simp add*: *Abs-finfun-inject*)
hence *finite* (*range ?f*)
by(*rule finite-subset*[*rotated 1*])(*auto simp add*: *F-def finfun-def* ‹*b1* ≠ *b2*› *intro*!: *exI*[**where** *x*=*b2*])
thus *finite* (*UNIV* :: *'a set*)
by(*rule finite-imageD*)(*auto intro*: *inj-onI simp add*: *fun-eq-iff* ‹*b1* ≠ *b2*› *split*: *if-split-asm*)

from *finite* **have** *finite* (*range* (λ*b*. ((*K$ b*) :: *'a* ⇒*f* *'b*)))
by(*rule finite-subset*[*rotated 1*]) *simp*
thus *finite* (*UNIV* :: *'b set*)
by(*rule finite-imageD*)(*auto intro*!: *inj-onI*)
qed
with *False* **show** *?thesis* **by** *auto*
qed

lemma *finite-UNIV-finfun*:
finite (*UNIV* :: (*'a* ⇒*f* *'b*) *set*) ⟷
(*finite* (*UNIV* :: *'a set*) ∧ *finite* (*UNIV* :: *'b set*) ∨ *CARD*(*'b*) = *1*)
(**is** *?lhs* ⟷ *?rhs*)
proof −
have *?lhs* ⟷ *CARD*(*'a* ⇒*f* *'b*) > *0* **by**(*simp add*: *card-gt-0-iff*)
also have ... ⟷ *CARD*(*'a*) > *0* ∧ *CARD*(*'b*) > *0* ∨ *CARD*(*'b*) = *1*
by(*simp add*: *card-UNIV-finfun*)
also have ... = *?rhs* **by**(*simp add*: *card-gt-0-iff*)
finally show *?thesis* **.**
qed

instantiation *finfun* :: (*finite-UNIV*, *card-UNIV*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*'a* ⇒*f* *'b*)
(*let cb* = *of-phantom* (*card-UNIV* :: *'b card-UNIV*)
in cb = *1* ∨ *of-phantom* (*finite-UNIV* :: *'a finite-UNIV*) ∧ *cb* ≠ *0*)
instance
by *intro-classes* (*auto simp add*: *finite-UNIV-finfun-def Let-def card-UNIV finite-UNIV finite-UNIV-finfun card-gt-0-iff*)
end

instantiation *finfun* :: (*card-UNIV*, *card-UNIV*) *card-UNIV* **begin**
definition *card-UNIV* = *Phantom*(*'a* ⇒*f* *'b*)
(*let ca* = *of-phantom* (*card-UNIV* :: *'a card-UNIV*);
cb = *of-phantom* (*card-UNIV* :: *'b card-UNIV*)
in if ca ≠ *0* ∧ *cb* ≠ *0* ∨ *cb* = *1 then cb ^ ca else 0*)
instance by *intro-classes* (*simp add*: *card-UNIV-finfun-def card-UNIV Let-def card-UNIV-finfun*)
end

Deactivate syntax again. Import theory *FinFun-Syntax* to reactivate it again

no-type-notation

finfun ((- ⇒*f* /-) [*22*, *21*] *21*)

no-notation
finfun-const (*K*$/ - [*0*] *1*) **and**
finfun-update (-′(- $:= -′) [*1000*,*0*,*0*] *1000*) **and**
finfun-apply (**infixl** $ *999*) **and**
finfun-comp (**infixr** ∘$ *55*) **and**
finfun-comp2 (**infixr** $∘ *55*) **and**
finfun-Diag ((*1*′($-,/ -$′)) [*0*, *0*] *1000*)

no-notation (*ASCII*)
finfun-comp (**infixr** *o*$ *55*) **and**
finfun-comp2 (**infixr** $*o* *55*)

end

38 Various algebraic structures combined with a lattice

theory *Lattice-Algebras*
imports *Complex-Main*
begin

class *semilattice-inf-ab-group-add* = *ordered-ab-group-add* + *semilattice-inf*
begin

lemma *add-inf-distrib-left*: *a* + *inf b c* = *inf* (*a* + *b*) (*a* + *c*)
apply (*rule antisym*)
apply (*simp-all add*: *le-infI*)
apply (*rule add-le-imp-le-left* [*of uminus a*])
apply (*simp only*: *add.assoc* [*symmetric*], *simp add*: *diff-le-eq add.commute*)
done

lemma *add-inf-distrib-right*: *inf a b* + *c* = *inf* (*a* + *c*) (*b* + *c*)
proof −
have *c* + *inf a b* = *inf* (*c* + *a*) (*c* + *b*)
by (*simp add*: *add-inf-distrib-left*)
then show *?thesis*
by (*simp add*: *add.commute*)
qed

end

class *semilattice-sup-ab-group-add* = *ordered-ab-group-add* + *semilattice-sup*
begin

lemma *add-sup-distrib-left*: *a* + *sup b c* = *sup* (*a* + *b*) (*a* + *c*)
apply (*rule antisym*)

```
  apply (rule add-le-imp-le-left [of uminus a])
  apply (simp only: add.assoc [symmetric], simp)
  apply (simp add: le-diff-eq add.commute)
  apply (rule le-supI)
  apply (rule add-le-imp-le-left [of a], simp only: add.assoc[symmetric], simp)+
  done

lemma add-sup-distrib-right: sup a b + c = sup (a + c) (b + c)
proof −
  have c + sup a b = sup (c+a) (c+b)
    by (simp add: add-sup-distrib-left)
  then show ?thesis
    by (simp add: add.commute)
qed

end

class lattice-ab-group-add = ordered-ab-group-add + lattice
begin

subclass semilattice-inf-ab-group-add ..
subclass semilattice-sup-ab-group-add ..

lemmas add-sup-inf-distribs =
  add-inf-distrib-right add-inf-distrib-left add-sup-distrib-right add-sup-distrib-left

lemma inf-eq-neg-sup: inf a b = − sup (− a) (− b)
proof (rule inf-unique)
  fix a b c :: 'a
  show − sup (− a) (− b) ≤ a
    by (rule add-le-imp-le-right [of - sup (uminus a) (uminus b)])
      (simp, simp add: add-sup-distrib-left)
  show − sup (−a) (−b) ≤ b
    by (rule add-le-imp-le-right [of - sup (uminus a) (uminus b)])
      (simp, simp add: add-sup-distrib-left)
  assume a ≤ b a ≤ c
  then show a ≤ − sup (−b) (−c)
    by (subst neg-le-iff-le [symmetric]) (simp add: le-supI)
qed

lemma sup-eq-neg-inf: sup a b = − inf (− a) (− b)
proof (rule sup-unique)
  fix a b c :: 'a
  show a ≤ − inf (− a) (− b)
    by (rule add-le-imp-le-right [of - inf (uminus a) (uminus b)])
      (simp, simp add: add-inf-distrib-left)
  show b ≤ − inf (− a) (− b)
    by (rule add-le-imp-le-right [of - inf (uminus a) (uminus b)])
      (simp, simp add: add-inf-distrib-left)
```

```
  assume a ≤ c b ≤ c
  then show − inf (− a) (− b) ≤ c
    by (subst neg-le-iff-le [symmetric]) (simp add: le-infI)
qed

lemma neg-inf-eq-sup: − inf a b = sup (− a) (− b)
  by (simp add: inf-eq-neg-sup)

lemma diff-inf-eq-sup: a − inf b c = a + sup (− b) (− c)
  using neg-inf-eq-sup [of b c, symmetric] by simp

lemma neg-sup-eq-inf: − sup a b = inf (− a) (− b)
  by (simp add: sup-eq-neg-inf)

lemma diff-sup-eq-inf: a − sup b c = a + inf (− b) (− c)
  using neg-sup-eq-inf [of b c, symmetric] by simp

lemma add-eq-inf-sup: a + b = sup a b + inf a b
proof −
  have 0 = − inf 0 (a − b) + inf (a − b) 0
    by (simp add: inf-commute)
  then have 0 = sup 0 (b − a) + inf (a − b) 0
    by (simp add: inf-eq-neg-sup)
  then have 0 = (− a + sup a b) + (inf a b + (− b))
    by (simp only: add-sup-distrib-left add-inf-distrib-right) simp
  then show ?thesis
    by (simp add: algebra-simps)
qed
```

38.1 Positive Part, Negative Part, Absolute Value

```
definition nprt :: 'a ⇒ 'a
  where nprt x = inf x 0

definition pprt :: 'a ⇒ 'a
  where pprt x = sup x 0

lemma pprt-neg: pprt (− x) = − nprt x
proof −
  have sup (− x) 0 = sup (− x) (− 0)
    unfolding minus-zero ..
  also have ... = − inf x 0
    unfolding neg-inf-eq-sup ..
  finally have sup (− x) 0 = − inf x 0 .
  then show ?thesis
    unfolding pprt-def nprt-def .
qed

lemma nprt-neg: nprt (− x) = − pprt x
```

proof –
  from *pprt-neg* **have** *pprt* (– (– *x*)) = – *nprt* (– *x*) **.**
  then have *pprt x* = – *nprt* (– *x*) **by** *simp*
  then show *?thesis* **by** *simp*
qed

lemma *prts*: *a* = *pprt a* + *nprt a*
  by (*simp add*: *pprt-def nprt-def add-eq-inf-sup*[*symmetric*])

lemma *zero-le-pprt*[*simp*]: *0* ≤ *pprt a*
  by (*simp add*: *pprt-def*)

lemma *nprt-le-zero*[*simp*]: *nprt a* ≤ *0*
  by (*simp add*: *nprt-def*)

lemma *le-eq-neg*: *a* ≤ – *b* ⟷ *a* + *b* ≤ *0*
  (**is** *?l* = *?r*)
proof
  assume *?l*
  then show *?r*
    apply –
    apply (*rule add-le-imp-le-right*[*of* - *uminus b* -])
    apply (*simp add*: *add.assoc*)
    done
next
  assume *?r*
  then show *?l*
    apply –
    apply (*rule add-le-imp-le-right*[*of* - *b* -])
    apply *simp*
    done
qed

lemma *pprt-0*[*simp*]: *pprt 0* = *0* **by** (*simp add*: *pprt-def*)
lemma *nprt-0*[*simp*]: *nprt 0* = *0* **by** (*simp add*: *nprt-def*)

lemma *pprt-eq-id* [*simp*, *no-atp*]: *0* ≤ *x* ⟹ *pprt x* = *x*
  by (*simp add*: *pprt-def sup-absorb1*)

lemma *nprt-eq-id* [*simp*, *no-atp*]: *x* ≤ *0* ⟹ *nprt x* = *x*
  by (*simp add*: *nprt-def inf-absorb1*)

lemma *pprt-eq-0* [*simp*, *no-atp*]: *x* ≤ *0* ⟹ *pprt x* = *0*
  by (*simp add*: *pprt-def sup-absorb2*)

lemma *nprt-eq-0* [*simp*, *no-atp*]: *0* ≤ *x* ⟹ *nprt x* = *0*
  by (*simp add*: *nprt-def inf-absorb2*)

lemma *sup-0-imp-0*:

```
  assumes sup a (− a) = 0
  shows a = 0
proof −
  have p: 0 ≤ a if sup a (− a) = 0 for a :: 'a
  proof −
    from that have sup a (− a) + a = a
      by simp
    then have sup (a + a) 0 = a
      by (simp add: add-sup-distrib-right)
    then have sup (a + a) 0 ≤ a
      by simp
    then show ?thesis
      by (blast intro: order-trans inf-sup-ord)
  qed
  from assms have **: sup (−a) (−(−a)) = 0
    by (simp add: sup-commute)
  from p[OF assms] p[OF **] show a = 0
    by simp
qed

lemma inf-0-imp-0: inf a (− a) = 0 ⟹ a = 0
  apply (simp add: inf-eq-neg-sup)
  apply (simp add: sup-commute)
  apply (erule sup-0-imp-0)
  done

lemma inf-0-eq-0 [simp, no-atp]: inf a (− a) = 0 ⟷ a = 0
  apply rule
  apply (erule inf-0-imp-0)
  apply simp
  done

lemma sup-0-eq-0 [simp, no-atp]: sup a (− a) = 0 ⟷ a = 0
  apply rule
  apply (erule sup-0-imp-0)
  apply simp
  done

lemma zero-le-double-add-iff-zero-le-single-add [simp]: 0 ≤ a + a ⟷ 0 ≤ a
  (is ?lhs ⟷ ?rhs)
proof
  show ?rhs if ?lhs
  proof −
    from that have a: inf (a + a) 0 = 0
      by (simp add: inf-commute inf-absorb1)
    have inf a 0 + inf a 0 = inf (inf (a + a) 0) a  (is ?l = -)
      by (simp add: add-sup-inf-distribs inf-aci)
    then have ?l = 0 + inf a 0
      by (simp add: a, simp add: inf-commute)
```

then have *inf a 0 = 0*
by (*simp only*: *add-right-cancel*)
then show *?thesis*
unfolding *le-iff-inf* **by** (*simp add*: *inf-commute*)
qed
show *?lhs* **if** *?rhs*
by (*simp add*: *add-mono*[*OF that that*, *simplified*])
qed

lemma *double-zero* [*simp*]: *a + a = 0 ⟷ a = 0*
(**is** *?lhs ⟷ ?rhs*)
proof
show *?rhs* **if** *?lhs*
proof −
from *that* **have** *a + a + − a = − a*
by *simp*
then have *a + (a + − a) = − a*
by (*simp only*: *add.assoc*)
then have *a*: *− a = a*
by *simp*
show *?thesis*
apply (*rule antisym*)
apply (*unfold neg-le-iff-le* [*symmetric*, *of a*])
unfolding *a*
apply *simp*
unfolding *zero-le-double-add-iff-zero-le-single-add* [*symmetric*, *of a*]
unfolding *that*
unfolding *le-less*
apply *simp-all*
done
qed
show *?lhs* **if** *?rhs*
using *that* **by** *simp*
qed

lemma *zero-less-double-add-iff-zero-less-single-add* [*simp*]: *0 < a + a ⟷ 0 < a*
proof (*cases a = 0*)
case *True*
then show *?thesis* **by** *auto*
next
case *False*
then show *?thesis*
unfolding *less-le*
apply *simp*
apply *rule*
apply *clarify*
apply *rule*
apply *assumption*
apply (*rule notI*)

unfolding *double-zero* [*symmetric*, *of a*]
apply *blast*
done
qed

lemma *double-add-le-zero-iff-single-add-le-zero* [*simp*]: $a + a \leq 0 \longleftrightarrow a \leq 0$
proof −
have $a + a \leq 0 \longleftrightarrow 0 \leq - (a + a)$
by (*subst le-minus-iff*) *simp*
moreover have $\ldots \longleftrightarrow a \leq 0$
by (*simp only*: *minus-add-distrib zero-le-double-add-iff-zero-le-single-add*) *simp*
ultimately show *?thesis*
by *blast*
qed

lemma *double-add-less-zero-iff-single-less-zero* [*simp*]: $a + a < 0 \longleftrightarrow a < 0$
proof −
have $a + a < 0 \longleftrightarrow 0 < - (a + a)$
by (*subst less-minus-iff*) *simp*
moreover have $\ldots \longleftrightarrow a < 0$
by (*simp only*: *minus-add-distrib zero-less-double-add-iff-zero-less-single-add*) *simp*
ultimately show *?thesis*
by *blast*
qed

declare *neg-inf-eq-sup* [*simp*] *neg-sup-eq-inf* [*simp*] *diff-inf-eq-sup* [*simp*] *diff-sup-eq-inf* [*simp*]

lemma *le-minus-self-iff*: $a \leq - a \longleftrightarrow a \leq 0$
proof −
from *add-le-cancel-left* [*of uminus a plus a a zero*]
have $a \leq - a \longleftrightarrow a + a \leq 0$
by (*simp add*: *add.assoc*[*symmetric*])
then show *?thesis*
by *simp*
qed

lemma *minus-le-self-iff*: $- a \leq a \longleftrightarrow 0 \leq a$
proof −
have $- a \leq a \longleftrightarrow 0 \leq a + a$
using *add-le-cancel-left* [*of uminus a zero plus a a*]
by (*simp add*: *add.assoc*[*symmetric*])
then show *?thesis*
by *simp*
qed

lemma *zero-le-iff-zero-nprt*: $0 \leq a \longleftrightarrow nprt\ a = 0$
unfolding *le-iff-inf* **by** (*simp add*: *nprt-def inf-commute*)

lemma *le-zero-iff-zero-pprt*: $a \leq 0 \longleftrightarrow pprt\ a = 0$
unfolding *le-iff-sup* **by** (*simp add*: *pprt-def sup-commute*)

lemma *le-zero-iff-pprt-id*: $0 \leq a \longleftrightarrow pprt\ a = a$
unfolding *le-iff-sup* **by** (*simp add*: *pprt-def sup-commute*)

lemma *zero-le-iff-nprt-id*: $a \leq 0 \longleftrightarrow nprt\ a = a$
unfolding *le-iff-inf* **by** (*simp add*: *nprt-def inf-commute*)

lemma *pprt-mono* [*simp*, *no-atp*]: $a \leq b \Longrightarrow pprt\ a \leq pprt\ b$
unfolding *le-iff-sup* **by** (*simp add*: *pprt-def sup-aci sup-assoc* [*symmetric*, *of a*])

lemma *nprt-mono* [*simp*, *no-atp*]: $a \leq b \Longrightarrow nprt\ a \leq nprt\ b$
unfolding *le-iff-inf* **by** (*simp add*: *nprt-def inf-aci inf-assoc* [*symmetric*, *of a*])

end

lemmas *add-sup-inf-distribs* =
add-inf-distrib-right add-inf-distrib-left add-sup-distrib-right add-sup-distrib-left

class *lattice-ab-group-add-abs* = *lattice-ab-group-add* + *abs* +
assumes *abs-lattice*: $|a| = sup\ a\ (-\ a)$
begin

lemma *abs-prts*: $|a| = pprt\ a - nprt\ a$
proof −
have $0 \leq |a|$
proof −
have *a*: $a \leq |a|$ **and** *b*: $-\ a \leq |a|$
by (*auto simp add*: *abs-lattice*)
show *?thesis*
by (*rule add-mono* [*OF a b*, *simplified*])
qed
then have $0 \leq sup\ a\ (-\ a)$
unfolding *abs-lattice* **.**
then have $sup\ (sup\ a\ (-\ a))\ 0 = sup\ a\ (-\ a)$
by (*rule sup-absorb1*)
then show *?thesis*
by (*simp add*: *add-sup-inf-distribs ac-simps pprt-def nprt-def abs-lattice*)
qed

subclass *ordered-ab-group-add-abs*
proof
have *abs-ge-zero* [*simp*]: $0 \leq |a|$ **for** *a*
proof −
have *a*: $a \leq |a|$ **and** *b*: $-\ a \leq |a|$
by (*auto simp add*: *abs-lattice*)

```
    show 0 ≤ |a|
      by (rule add-mono [OF a b, simplified])
  qed
  have abs-leI: a ≤ b ⟹ − a ≤ b ⟹ |a| ≤ b for a b
    by (simp add: abs-lattice le-supI)
  fix a b
  show 0 ≤ |a|
    by simp
  show a ≤ |a|
    by (auto simp add: abs-lattice)
  show |−a| = |a|
    by (simp add: abs-lattice sup-commute)
  show − a ≤ b ⟹ |a| ≤ b if a ≤ b
    using that by (rule abs-leI)
  show |a + b| ≤ |a| + |b|
  proof −
    have g: |a| + |b| = sup (a + b) (sup (− a − b) (sup (− a + b) (a + (− b))))
      (is - = sup ?m ?n)
      by (simp add: abs-lattice add-sup-inf-distribs ac-simps)
    have a: a + b ≤ sup ?m ?n
      by simp
    have b: − a − b ≤ ?n
      by simp
    have c: ?n ≤ sup ?m ?n
      by simp
    from b c have d: − a − b ≤ sup ?m ?n
      by (rule order-trans)
    have e: − a − b = − (a + b)
      by simp
    from a d e have |a + b| ≤ sup ?m ?n
      apply −
      apply (drule abs-leI)
      apply (simp-all only: algebra-simps minus-add)
      apply (metis add-uminus-conv-diff d sup-commute uminus-add-conv-diff)
      done
    with g[symmetric] show ?thesis by simp
  qed
qed

end

lemma sup-eq-if:
  fixes a :: 'a::{lattice-ab-group-add,linorder}
  shows sup a (− a) = (if a < 0 then − a else a)
  using add-le-cancel-right [of a a − a, symmetric, simplified]
    and add-le-cancel-right [of −a a a, symmetric, simplified]
  by (auto simp: sup-max max.absorb1 max.absorb2)

lemma abs-if-lattice:
```

fixes *a* :: *'a*::{*lattice-ab-group-add-abs*,*linorder*}
shows |*a*| = (*if a* < *0 then* − *a else a*)
by *auto*

lemma *estimate-by-abs*:
fixes *a b c* :: *'a*::*lattice-ab-group-add-abs*
assumes *a* + *b* ≤ *c*
shows *a* ≤ *c* + |*b*|
proof −
from *assms* **have** *a* ≤ *c* + (− *b*)
by (*simp add*: *algebra-simps*)
have − *b* ≤ |*b*|
by (*rule abs-ge-minus-self*)
then have *c* + (− *b*) ≤ *c* + |*b*|
by (*rule add-left-mono*)
with ‹*a* ≤ *c* + (− *b*)› **show** *?thesis*
by (*rule order-trans*)
qed

class *lattice-ring* = *ordered-ring* + *lattice-ab-group-add-abs*
begin

subclass *semilattice-inf-ab-group-add* **..**
subclass *semilattice-sup-ab-group-add* **..**

end

lemma *abs-le-mult*:
fixes *a b* :: *'a*::*lattice-ring*
shows |*a* ∗ *b*| ≤ |*a*| ∗ |*b*|
proof −
let *?x* = *pprt a* ∗ *pprt b* − *pprt a* ∗ *nprt b* − *nprt a* ∗ *pprt b* + *nprt a* ∗ *nprt b*
let *?y* = *pprt a* ∗ *pprt b* + *pprt a* ∗ *nprt b* + *nprt a* ∗ *pprt b* + *nprt a* ∗ *nprt b*
have *a*: |*a*| ∗ |*b*| = *?x*
by (*simp only*: *abs-prts*[*of a*] *abs-prts*[*of b*] *algebra-simps*)
have *bh*: *u* = *a* ⟹ *v* = *b* ⟹
u ∗ *v* = *pprt a* ∗ *pprt b* + *pprt a* ∗ *nprt b* +
nprt a ∗ *pprt b* + *nprt a* ∗ *nprt b* **for** *u v* :: *'a*
apply (*subst prts*[*of u*], *subst prts*[*of v*])
apply (*simp add*: *algebra-simps*)
done
note *b* = *this*[*OF refl*[*of a*] *refl*[*of b*]]
have *xy*: − *?x* ≤ *?y*
apply *simp*
apply (*metis* (*full-types*) *add-increasing add-uminus-conv-diff*
lattice-ab-group-add-class.minus-le-self-iff minus-add-distrib mult-nonneg-nonneg
mult-nonpos-nonpos nprt-le-zero zero-le-pprt)
done
have *yx*: *?y* ≤ *?x*

```
    apply simp
    apply (metis (full-types) add-nonpos-nonpos add-uminus-conv-diff
     lattice-ab-group-add-class.le-minus-self-iff minus-add-distrib mult-nonneg-nonpos
      mult-nonpos-nonneg nprt-le-zero zero-le-pprt)
    done
  have i1: a ∗ b ≤ |a| ∗ |b|
    by (simp only: a b yx)
  have i2: − (|a| ∗ |b|) ≤ a ∗ b
    by (simp only: a b xy)
  show ?thesis
    apply (rule abs-leI)
    apply (simp add: i1)
    apply (simp add: i2[simplified minus-le-iff])
    done
qed

instance lattice-ring ⊆ ordered-ring-abs
proof
  fix a b :: 'a::lattice-ring
  assume a: (0 ≤ a ∨ a ≤ 0) ∧ (0 ≤ b ∨ b ≤ 0)
  show |a ∗ b| = |a| ∗ |b|
  proof −
    have s: (0 ≤ a ∗ b) ∨ (a ∗ b ≤ 0)
      apply auto
      apply (rule-tac split-mult-pos-le)
      apply (rule-tac contrapos-np[of a ∗ b ≤ 0])
      apply simp
      apply (rule-tac split-mult-neg-le)
      using a
      apply blast
      done
    have mulprts: a ∗ b = (pprt a + nprt a) ∗ (pprt b + nprt b)
      by (simp add: prts[symmetric])
    show ?thesis
    proof (cases 0 ≤ a ∗ b)
      case True
      then show ?thesis
        apply (simp-all add: mulprts abs-prts)
        using a
        apply (auto simp add:
          algebra-simps
          iffD1[OF zero-le-iff-zero-nprt] iffD1[OF le-zero-iff-zero-pprt]
          iffD1[OF le-zero-iff-pprt-id] iffD1[OF zero-le-iff-nprt-id])
        apply(drule (1) mult-nonneg-nonpos[of a b], simp)
        apply(drule (1) mult-nonneg-nonpos2[of b a], simp)
        done
    next
      case False
      with s have a ∗ b ≤ 0
```

by *simp*
then show *?thesis*
apply (*simp-all add*: *mulprts abs-prts*)
apply (*insert a*)
apply (*auto simp add*: *algebra-simps*)
apply(*drule* (*1*) *mult-nonneg-nonneg*[*of a b*],*simp*)
apply(*drule* (*1*) *mult-nonpos-nonpos*[*of a b*],*simp*)
done
qed
qed
qed

lemma *mult-le-prts*:
fixes *a b* :: ′*a*::*lattice-ring*
assumes *a1* ≤ *a*
and *a* ≤ *a2*
and *b1* ≤ *b*
and *b* ≤ *b2*
shows *a* ∗ *b* ≤
pprt a2 ∗ *pprt b2* + *pprt a1* ∗ *nprt b2* + *nprt a2* ∗ *pprt b1* + *nprt a1* ∗ *nprt b1*
proof −
have *a* ∗ *b* = (*pprt a* + *nprt a*) ∗ (*pprt b* + *nprt b*)
by (*subst prts*[*symmetric*])+ *simp*
then have *a* ∗ *b* = *pprt a* ∗ *pprt b* + *pprt a* ∗ *nprt b* + *nprt a* ∗ *pprt b* + *nprt a* ∗ *nprt b*
by (*simp add*: *algebra-simps*)
moreover have *pprt a* ∗ *pprt b* ≤ *pprt a2* ∗ *pprt b2*
by (*simp-all add*: *assms mult-mono*)
moreover have *pprt a* ∗ *nprt b* ≤ *pprt a1* ∗ *nprt b2*
proof −
have *pprt a* ∗ *nprt b* ≤ *pprt a* ∗ *nprt b2*
by (*simp add*: *mult-left-mono assms*)
moreover have *pprt a* ∗ *nprt b2* ≤ *pprt a1* ∗ *nprt b2*
by (*simp add*: *mult-right-mono-neg assms*)
ultimately show *?thesis*
by *simp*
qed
moreover have *nprt a* ∗ *pprt b* ≤ *nprt a2* ∗ *pprt b1*
proof −
have *nprt a* ∗ *pprt b* ≤ *nprt a2* ∗ *pprt b*
by (*simp add*: *mult-right-mono assms*)
moreover have *nprt a2* ∗ *pprt b* ≤ *nprt a2* ∗ *pprt b1*
by (*simp add*: *mult-left-mono-neg assms*)
ultimately show *?thesis*
by *simp*
qed
moreover have *nprt a* ∗ *nprt b* ≤ *nprt a1* ∗ *nprt b1*
proof −

```
    have nprt a ∗ nprt b ≤ nprt a ∗ nprt b1
      by (simp add: mult-left-mono-neg assms)
    moreover have nprt a ∗ nprt b1 ≤ nprt a1 ∗ nprt b1
      by (simp add: mult-right-mono-neg assms)
    ultimately show ?thesis
      by simp
  qed
  ultimately show ?thesis
    by − (rule add-mono | simp)+
qed

lemma mult-ge-prts:
  fixes a b :: 'a::lattice-ring
  assumes a1 ≤ a
    and a ≤ a2
    and b1 ≤ b
    and b ≤ b2
  shows a ∗ b ≥
    nprt a1 ∗ pprt b2 + nprt a2 ∗ nprt b2 + pprt a1 ∗ pprt b1 + pprt a2 ∗ nprt
b1
proof −
  from assms have a1: − a2 ≤ −a
    by auto
  from assms have a2: − a ≤ −a1
    by auto
  from mult-le-prts[of − a2 − a − a1 b1 b b2,
    OF a1 a2 assms(3) assms(4), simplified nprt-neg pprt-neg]
  have le: − (a ∗ b) ≤
    − nprt a1 ∗ pprt b2 + − nprt a2 ∗ nprt b2 +
    − pprt a1 ∗ pprt b1 + − pprt a2 ∗ nprt b1
    by simp
  then have − (− nprt a1 ∗ pprt b2 + − nprt a2 ∗ nprt b2 +
      − pprt a1 ∗ pprt b1 + − pprt a2 ∗ nprt b1) ≤ a ∗ b
    by (simp only: minus-le-iff)
  then show ?thesis
    by (simp add: algebra-simps)
qed

instance int :: lattice-ring
proof
  fix k :: int
  show |k| = sup k (− k)
    by (auto simp add: sup-int-def)
qed

instance real :: lattice-ring
proof
  fix a :: real
  show |a| = sup a (− a)
```

by (*auto simp add*: *sup-real-def*)
qed

end

39 Floating-Point Numbers

theory *Float*
imports *Complex-Main Lattice-Algebras*
begin

definition *float* = {*m* ∗ *2 powr e* | (*m* :: *int*) (*e* :: *int*). *True*}

typedef *float* = *float*
morphisms *real-of-float float-of*
unfolding *float-def* **by** *auto*

setup-lifting *type-definition-float*

declare *real-of-float* [*code-unfold*]

lemmas *float-of-inject*[*simp*]

declare [[*coercion real-of-float* :: *float* ⇒ *real*]]

lemma *real-of-float-eq*:
fixes *f1 f2* :: *float*
shows *f1* = *f2* ⟷ *real-of-float f1* = *real-of-float f2*
unfolding *real-of-float-inject* **..**

declare *real-of-float-inverse*[*simp*] *float-of-inverse* [*simp*]
declare *real-of-float* [*simp*]

39.1 Real operations preserving the representation as floating point number

lemma *floatI*: **fixes** *m e* :: *int* **shows** *m* ∗ *2 powr e* = *x* ⟹ *x* ∈ *float*
by (*auto simp*: *float-def*)

lemma *zero-float*[*simp*]: *0* ∈ *float*
by (*auto simp*: *float-def*)
lemma *one-float*[*simp*]: *1* ∈ *float*
by (*intro floatI*[*of 1 0*]) *simp*
lemma *numeral-float*[*simp*]: *numeral i* ∈ *float*
by (*intro floatI*[*of numeral i 0*]) *simp*
lemma *neg-numeral-float*[*simp*]: − *numeral i* ∈ *float*
by (*intro floatI*[*of* − *numeral i 0*]) *simp*
lemma *real-of-int-float*[*simp*]: *real-of-int* (*x* :: *int*) ∈ *float*
by (*intro floatI*[*of x 0*]) *simp*

lemma *real-of-nat-float*[*simp*]: *real* (*x* :: *nat*) ∈ *float*
  by (*intro floatI*[*of x 0*]) *simp*
lemma *two-powr-int-float*[*simp*]: *2 powr* (*real-of-int* (*i*::*int*)) ∈ *float*
  by (*intro floatI*[*of 1 i*]) *simp*
lemma *two-powr-nat-float*[*simp*]: *2 powr* (*real* (*i*::*nat*)) ∈ *float*
  by (*intro floatI*[*of 1 i*]) *simp*
lemma *two-powr-minus-int-float*[*simp*]: *2 powr* − (*real-of-int* (*i*::*int*)) ∈ *float*
  by (*intro floatI*[*of 1* −*i*]) *simp*
lemma *two-powr-minus-nat-float*[*simp*]: *2 powr* − (*real* (*i*::*nat*)) ∈ *float*
  by (*intro floatI*[*of 1* −*i*]) *simp*
lemma *two-powr-numeral-float*[*simp*]: *2 powr numeral i* ∈ *float*
  by (*intro floatI*[*of 1 numeral i*]) *simp*
lemma *two-powr-neg-numeral-float*[*simp*]: *2 powr* − *numeral i* ∈ *float*
  by (*intro floatI*[*of 1* − *numeral i*]) *simp*
lemma *two-pow-float*[*simp*]: *2* ˆ *n* ∈ *float*
  by (*intro floatI*[*of 1 n*]) (*simp add*: *powr-realpow*)

lemma *plus-float*[*simp*]: *r* ∈ *float* ⟹ *p* ∈ *float* ⟹ *r* + *p* ∈ *float*
  unfolding *float-def*
proof (*safe*, *simp*)
  have ∗: ∃(*m*::*int*) (*e*::*int*). *m1* ∗ *2 powr e1* + *m2* ∗ *2 powr e2* = *m* ∗ *2 powr e*
    if *e1* ≤ *e2* **for** *e1 m1 e2 m2* :: *int*
  proof −
    from *that* **have** *m1* ∗ *2 powr e1* + *m2* ∗ *2 powr e2* = (*m1* + *m2* ∗ *2* ˆ *nat*
(*e2* − *e1*)) ∗ *2 powr e1*
      by (*simp add*: *powr-realpow*[*symmetric*] *powr-divide2*[*symmetric*] *field-simps*)
    then show *?thesis*
      by *blast*
  qed
  fix *e1 m1 e2 m2* :: *int*
  consider *e2* ≤ *e1* | *e1* ≤ *e2* **by** (*rule linorder-le-cases*)
  then show ∃(*m*::*int*) (*e*::*int*). *m1* ∗ *2 powr e1* + *m2* ∗ *2 powr e2* = *m* ∗ *2 powr*
e
  proof *cases*
    case *1*
    from ∗[*OF this*, *of m2 m1*] **show** *?thesis*
      by (*simp add*: *ac-simps*)
  next
    case *2*
    then show *?thesis* **by** (*rule* ∗)
  qed
qed

lemma *uminus-float*[*simp*]: *x* ∈ *float* ⟹ −*x* ∈ *float*
  apply (*auto simp*: *float-def*)
  apply *hypsubst-thin*
  apply (*rename-tac m e*)

```
  apply (rule-tac x=−m in exI)
  apply (rule-tac x=e in exI)
  apply (simp add: field-simps)
  done

lemma times-float[simp]: x ∈ float ⟹ y ∈ float ⟹ x ∗ y ∈ float
  apply (auto simp: float-def)
  apply hypsubst-thin
  apply (rename-tac mx my ex ey)
  apply (rule-tac x=mx ∗ my in exI)
  apply (rule-tac x=ex + ey in exI)
  apply (simp add: powr-add)
  done

lemma minus-float[simp]: x ∈ float ⟹ y ∈ float ⟹ x − y ∈ float
  using plus-float [of x − y] by simp

lemma abs-float[simp]: x ∈ float ⟹ |x| ∈ float
  by (cases x rule: linorder-cases[of 0]) auto

lemma sgn-of-float[simp]: x ∈ float ⟹ sgn x ∈ float
  by (cases x rule: linorder-cases[of 0]) (auto intro!: uminus-float)

lemma div-power-2-float[simp]: x ∈ float ⟹ x / 2^d ∈ float
  apply (auto simp add: float-def)
  apply hypsubst-thin
  apply (rename-tac m e)
  apply (rule-tac x=m in exI)
  apply (rule-tac x=e − d in exI)
  apply (simp add: powr-realpow[symmetric] field-simps powr-add[symmetric])
  done

lemma div-power-2-int-float[simp]: x ∈ float ⟹ x / (2::int)^d ∈ float
  apply (auto simp add: float-def)
  apply hypsubst-thin
  apply (rename-tac m e)
  apply (rule-tac x=m in exI)
  apply (rule-tac x=e − d in exI)
  apply (simp add: powr-realpow[symmetric] field-simps powr-add[symmetric])
  done

lemma div-numeral-Bit0-float[simp]:
  assumes x: x / numeral n ∈ float
  shows x / (numeral (Num.Bit0 n)) ∈ float
proof −
  have (x / numeral n) / 2^1 ∈ float
    by (intro x div-power-2-float)
  also have (x / numeral n) / 2^1 = x / (numeral (Num.Bit0 n))
    by (induct n) auto
```

finally show *?thesis* **.**
qed

lemma *div-neg-numeral-Bit0-float*[*simp*]:
assumes *x*: *x* / *numeral n* ∈ *float*
shows *x* / (− *numeral* (*Num.Bit0 n*)) ∈ *float*
proof −
have − (*x* / *numeral* (*Num.Bit0 n*)) ∈ *float*
using *x* **by** *simp*
also have − (*x* / *numeral* (*Num.Bit0 n*)) = *x* / − *numeral* (*Num.Bit0 n*)
by *simp*
finally show *?thesis* **.**
qed

lemma *power-float*[*simp*]:
assumes *a* ∈ *float*
shows *a* ˆ *b* ∈ *float*
proof −
from *assms* **obtain** *m e* :: *int* **where** *a* = *m* ∗ *2 powr e*
by (*auto simp*: *float-def*)
then show *?thesis*
by (*auto intro*!: *floatI*[**where** *m*=*m*ˆ*b* **and** *e* = *e*∗*b*]
simp: *power-mult-distrib powr-realpow*[*symmetric*] *powr-powr*)
qed

lift-definition *Float* :: *int* ⇒ *int* ⇒ *float* **is** λ(*m*::*int*) (*e*::*int*). *m* ∗ *2 powr e*
by *simp*
declare *Float.rep-eq*[*simp*]

code-datatype *Float*

lemma *compute-real-of-float*[*code*]:
real-of-float (*Float m e*) = (*if e* ≥ *0 then m* ∗ *2* ˆ *nat e else m* / *2* ˆ (*nat* (−*e*)))
by (*simp add*: *powr-int*)

39.2 Arithmetic operations on floating point numbers

instantiation *float* :: {*ring-1*, *linorder*, *linordered-ring*, *linordered-idom*, *numeral*, *equal*}
begin

lift-definition *zero-float* :: *float* **is** *0* **by** *simp*
declare *zero-float.rep-eq*[*simp*]
lift-definition *one-float* :: *float* **is** *1* **by** *simp*
declare *one-float.rep-eq*[*simp*]
lift-definition *plus-float* :: *float* ⇒ *float* ⇒ *float* **is** *op* + **by** *simp*
declare *plus-float.rep-eq*[*simp*]
lift-definition *times-float* :: *float* ⇒ *float* ⇒ *float* **is** *op* ∗ **by** *simp*
declare *times-float.rep-eq*[*simp*]

lift-definition *minus-float* :: *float* ⇒ *float* ⇒ *float* **is** *op* − **by** *simp*
declare *minus-float.rep-eq*[*simp*]
lift-definition *uminus-float* :: *float* ⇒ *float* **is** *uminus* **by** *simp*
declare *uminus-float.rep-eq*[*simp*]

lift-definition *abs-float* :: *float* ⇒ *float* **is** *abs* **by** *simp*
declare *abs-float.rep-eq*[*simp*]
lift-definition *sgn-float* :: *float* ⇒ *float* **is** *sgn* **by** *simp*
declare *sgn-float.rep-eq*[*simp*]

lift-definition *equal-float* :: *float* ⇒ *float* ⇒ *bool* **is** *op* = :: *real* ⇒ *real* ⇒ *bool* **.**

lift-definition *less-eq-float* :: *float* ⇒ *float* ⇒ *bool* **is** *op* ≤ **.**
declare *less-eq-float.rep-eq*[*simp*]
lift-definition *less-float* :: *float* ⇒ *float* ⇒ *bool* **is** *op* < **.**
declare *less-float.rep-eq*[*simp*]

instance
by (*standard*; *transfer*; *fastforce simp add*: *field-simps intro*: *mult-left-mono mult-right-mono*)+

end

lemma *real-of-float* [*simp*]: *real-of-float* (*of-nat n*) = *of-nat n*
by (*induct n*) *simp-all*

lemma *real-of-float-of-int-eq* [*simp*]: *real-of-float* (*of-int z*) = *of-int z*
by (*cases z rule*: *int-diff-cases*) (*simp-all add*: *of-rat-diff*)

lemma *Float-0-eq-0*[*simp*]: *Float 0 e* = *0*
by *transfer simp*

lemma *real-of-float-power*[*simp*]:
fixes *f* :: *float*
shows *real-of-float* (*f*^*n*) = *real-of-float f*^*n*
by (*induct n*) *simp-all*

lemma
fixes *x y* :: *float*
shows *real-of-float-min*: *real-of-float* (*min x y*) = *min* (*real-of-float x*) (*real-of-float y*)
and *real-of-float-max*: *real-of-float* (*max x y*) = *max* (*real-of-float x*) (*real-of-float y*)
by (*simp-all add*: *min-def max-def*)

instance *float* :: *unbounded-dense-linorder*
proof
fix *a b* :: *float*
show ∃ *c*. *a* < *c*

apply (*intro exI*[*of* - *a* + *1*])
apply *transfer*
apply *simp*
done
show ∃ *c*. *c* < *a*
apply (*intro exI*[*of* - *a* − *1*])
apply *transfer*
apply *simp*
done
show ∃ *c*. *a* < *c* ∧ *c* < *b* **if** *a* < *b*
apply (*rule exI*[*of* - (*a* + *b*) ∗ *Float 1* (− *1*)])
using *that*
apply *transfer*
apply (*simp add*: *powr-minus*)
done
qed

instantiation *float* :: *lattice-ab-group-add*
begin

definition *inf-float* :: *float* ⇒ *float* ⇒ *float*
where *inf-float a b* = *min a b*

definition *sup-float* :: *float* ⇒ *float* ⇒ *float*
where *sup-float a b* = *max a b*

instance
by (*standard*; *transfer*; *simp add*: *inf-float-def sup-float-def real-of-float-min real-of-float-max*)

end

lemma *float-numeral*[*simp*]: *real-of-float* (*numeral x* :: *float*) = *numeral x*
apply (*induct x*)
apply *simp*
apply (*simp-all only*: *numeral-Bit0 numeral-Bit1 real-of-float-eq float-of-inverse plus-float.rep-eq one-float.rep-eq plus-float numeral-float one-float*)
done

lemma *transfer-numeral* [*transfer-rule*]:
rel-fun (*op* =) *pcr-float* (*numeral* :: - ⇒ *real*) (*numeral* :: - ⇒ *float*)
by (*simp add*: *rel-fun-def float.pcr-cr-eq cr-float-def*)

lemma *float-neg-numeral*[*simp*]: *real-of-float* (− *numeral x* :: *float*) = − *numeral x*
by *simp*

lemma *transfer-neg-numeral* [*transfer-rule*]:
rel-fun (*op* =) *pcr-float* (− *numeral* :: - ⇒ *real*) (− *numeral* :: - ⇒ *float*)
by (*simp add*: *rel-fun-def float.pcr-cr-eq cr-float-def*)

lemma
shows *float-of-numeral*[*simp*]: *numeral k* = *float-of* (*numeral k*)
and *float-of-neg-numeral*[*simp*]: − *numeral k* = *float-of* (− *numeral k*)
unfolding *real-of-float-eq* **by** *simp-all*

39.3 Quickcheck

instantiation *float* :: *exhaustive*
begin

definition *exhaustive-float* **where**
exhaustive-float f d =
Quickcheck-Exhaustive.exhaustive (%*x*. *Quickcheck-Exhaustive.exhaustive* (%*y*. *f* (*Float x y*)) *d*) *d*

instance ..

end

definition (**in** *term-syntax*) [*code-unfold*]:
valtermify-float x y = *Code-Evaluation.valtermify Float* {·} *x* {·} *y*

instantiation *float* :: *full-exhaustive*
begin

definition *full-exhaustive-float* **where**
full-exhaustive-float f d =
Quickcheck-Exhaustive.full-exhaustive
(λ*x*. *Quickcheck-Exhaustive.full-exhaustive* (λ*y*. *f* (*valtermify-float x y*)) *d*) *d*

instance ..

end

instantiation *float* :: *random*
begin

definition *Quickcheck-Random.random i* =
scomp (*Quickcheck-Random.random* (*2* ˆ *nat-of-natural i*))
(λ*man*. *scomp* (*Quickcheck-Random.random i*) (λ*exp*. *Pair* (*valtermify-float man exp*)))

instance ..

end

39.4 Represent floats as unique mantissa and exponent

lemma *int-induct-abs*[*case-names less*]:

```
  fixes j :: int
  assumes H: ⋀n. (⋀i. |i| < |n| ⟹ P i) ⟹ P n
  shows P j
proof (induct nat |j| arbitrary: j rule: less-induct)
  case less
  show ?case by (rule H[OF less]) simp
qed

lemma int-cancel-factors:
  fixes n :: int
  assumes 1 < r
  shows n = 0 ∨ (∃ k i. n = k ∗ r ˆ i ∧ ¬ r dvd k)
proof (induct n rule: int-induct-abs)
  case (less n)
  have ∃ k i. n = k ∗ r ˆ Suc i ∧ ¬ r dvd k if n ≠ 0 n = m ∗ r for m
  proof −
    from that have |m | < |n|
      using ‹1 < r› by (simp add: abs-mult)
    from less[OF this] that show ?thesis by auto
  qed
  then show ?case
   by (metis dvd-def monoid-mult-class.mult.right-neutral mult.commute power-0)
qed

lemma mult-powr-eq-mult-powr-iff-asym:
  fixes m1 m2 e1 e2 :: int
  assumes m1: ¬ 2 dvd m1
    and e1 ≤ e2
  shows m1 ∗ 2 powr e1 = m2 ∗ 2 powr e2 ⟷ m1 = m2 ∧ e1 = e2
  (is ?lhs ⟷ ?rhs)
proof
  show ?rhs if eq: ?lhs
  proof −
    have m1 ≠ 0
      using m1 unfolding dvd-def by auto
    from ‹e1 ≤ e2› eq have m1 = m2 ∗ 2 powr nat (e2 − e1)
      by (simp add: powr-divide2[symmetric] field-simps)
    also have ... = m2 ∗ 2ˆnat (e2 − e1)
      by (simp add: powr-realpow)
    finally have m1-eq: m1 = m2 ∗ 2ˆnat (e2 − e1)
      by linarith
    with m1 have m1 = m2
      by (cases nat (e2 − e1)) (auto simp add: dvd-def)
    then show ?thesis
      using eq ‹m1 ≠ 0› by (simp add: powr-inj)
  qed
  show ?lhs if ?rhs
    using that by simp
qed
```

lemma *mult-powr-eq-mult-powr-iff*:
fixes *m1 m2 e1 e2* :: *int*
shows $\neg$ *2 dvd m1* $\Longrightarrow$ $\neg$ *2 dvd m2* $\Longrightarrow$ *m1* $*$ *2 powr e1* = *m2* $*$ *2 powr e2* $\longleftrightarrow$ *m1* = *m2* $\wedge$ *e1* = *e2*
using *mult-powr-eq-mult-powr-iff-asym*[*of m1 e1 e2 m2*]
using *mult-powr-eq-mult-powr-iff-asym*[*of m2 e2 e1 m1*]
by (*cases e1 e2 rule*: *linorder-le-cases*) *auto*

lemma *floatE-normed*:
assumes *x*: *x* $\in$ *float*
obtains (*zero*) *x* = *0*
| (*powr*) *m e* :: *int* **where** *x* = *m* $*$ *2 powr e* $\neg$ *2 dvd m x* $\neq$ *0*
proof −
{
assume *x* $\neq$ *0*
from *x* **obtain** *m e* :: *int* **where** *x*: *x* = *m* $*$ *2 powr e*
by (*auto simp*: *float-def*)
with ‹*x* $\neq$ *0*› *int-cancel-factors*[*of 2 m*] **obtain** *k i* **where** *m* = *k* $*$ *2* ^ *i* $\neg$ *2 dvd k*
by *auto*
with ‹$\neg$ *2 dvd k*› *x* **have** $\exists$(*m*::*int*) (*e*::*int*). *x* = *m* $*$ *2 powr e* $\wedge$ $\neg$ (*2*::*int*) *dvd m*
by (*rule-tac exI*[*of - k*], *rule-tac exI*[*of - e* + *int i*])
(*simp add*: *powr-add powr-realpow*)
}
with *that* **show** *thesis* **by** *blast*
qed

lemma *float-normed-cases*:
fixes *f* :: *float*
obtains (*zero*) *f* = *0*
| (*powr*) *m e* :: *int* **where** *real-of-float f* = *m* $*$ *2 powr e* $\neg$ *2 dvd m f* $\neq$ *0*
proof (*atomize-elim*, *induct f*)
case (*float-of y*)
then show *?case*
by (*cases rule*: *floatE-normed*) (*auto simp*: *zero-float-def*)
qed

definition *mantissa* :: *float* $\Rightarrow$ *int* **where**
mantissa f = *fst* (*SOME p*::*int* $\times$ *int*. (*f* = *0* $\wedge$ *fst p* = *0* $\wedge$ *snd p* = *0*)
$\vee$ (*f* $\neq$ *0* $\wedge$ *real-of-float f* = *real-of-int* (*fst p*) $*$ *2 powr real-of-int* (*snd p*) $\wedge$ $\neg$ *2 dvd fst p*))

definition *exponent* :: *float* $\Rightarrow$ *int* **where**
exponent f = *snd* (*SOME p*::*int* $\times$ *int*. (*f* = *0* $\wedge$ *fst p* = *0* $\wedge$ *snd p* = *0*)
$\vee$ (*f* $\neq$ *0* $\wedge$ *real-of-float f* = *real-of-int* (*fst p*) $*$ *2 powr real-of-int* (*snd p*) $\wedge$ $\neg$ *2 dvd fst p*))

lemma
shows *exponent-0*[*simp*]: *exponent* (*float-of 0*) = *0* (**is** *?E*)
and *mantissa-0*[*simp*]: *mantissa* (*float-of 0*) = *0* (**is** *?M*)
proof −
have $\bigwedge p$::*int* × *int*. *fst p* = *0* ∧ *snd p* = *0* ⟷ *p* = (*0*, *0*)
by *auto*
then show *?E ?M*
by (*auto simp add*: *mantissa-def exponent-def zero-float-def*)
qed

lemma
shows *mantissa-exponent*: *real-of-float f* = *mantissa f* ∗ *2 powr exponent f* (**is** *?E*)
and *mantissa-not-dvd*: *f* ≠ (*float-of 0*) ⟹ ¬ *2 dvd mantissa f* (**is** - ⟹ *?D*)
proof *cases*
assume [*simp*]: *f* ≠ *float-of 0*
have *f* = *mantissa f* ∗ *2 powr exponent f* ∧ ¬ *2 dvd mantissa f*
proof (*cases f rule*: *float-normed-cases*)
case *zero*
then show *?thesis* **by** (*simp add*: *zero-float-def*)
next
case (*powr m e*)
then have ∃ *p*::*int* × *int*. (*f* = *0* ∧ *fst p* = *0* ∧ *snd p* = *0*) ∨
(*f* ≠ *0* ∧ *real-of-float f* = *real-of-int* (*fst p*) ∗ *2 powr real-of-int* (*snd p*) ∧ ¬ *2 dvd fst p*)
by *auto*
then show *?thesis*
unfolding *exponent-def mantissa-def*
by (*rule someI2-ex*) (*simp add*: *zero-float-def*)
qed
then show *?E ?D* **by** *auto*
qed *simp*

lemma *mantissa-noteq-0*: *f* ≠ *float-of 0* ⟹ *mantissa f* ≠ *0*
using *mantissa-not-dvd*[*of f*] **by** *auto*

lemma
fixes *m e* :: *int*
defines *f* ≡ *float-of* (*m* ∗ *2 powr e*)
assumes *dvd*: ¬ *2 dvd m*
shows *mantissa-float*: *mantissa f* = *m* (**is** *?M*)
and *exponent-float*: *m* ≠ *0* ⟹ *exponent f* = *e* (**is** - ⟹ *?E*)
proof *cases*
assume *m* = *0*
with *dvd* **show** *mantissa f* = *m* **by** *auto*
next
assume *m* ≠ *0*
then have *f-not-0*: *f* ≠ *float-of 0* **by** (*simp add*: *f-def*)
from *mantissa-exponent*[*of f*] **have** *m* ∗ *2 powr e* = *mantissa f* ∗ *2 powr exponent*

```
f
    by (auto simp add: f-def)
  then show ?M ?E
    using mantissa-not-dvd[OF f-not-0] dvd
    by (auto simp: mult-powr-eq-mult-powr-iff)
qed
```

39.5 Compute arithmetic operations

```
lemma Float-mantissa-exponent: Float (mantissa f) (exponent f) = f
  unfolding real-of-float-eq mantissa-exponent[of f] by simp

lemma Float-cases [cases type: float]:
  fixes f :: float
  obtains (Float) m e :: int where f = Float m e
  using Float-mantissa-exponent[symmetric]
  by (atomize-elim) auto

lemma denormalize-shift:
  assumes f-def: f ≡ Float m e
    and not-0: f ≠ float-of 0
  obtains i where m = mantissa f * 2 ^ i e = exponent f − i
proof
  from mantissa-exponent[of f] f-def
  have m * 2 powr e = mantissa f * 2 powr exponent f
    by simp
  then have eq: m = mantissa f * 2 powr (exponent f − e)
    by (simp add: powr-divide2[symmetric] field-simps)
  moreover
  have e ≤ exponent f
  proof (rule ccontr)
    assume ¬ e ≤ exponent f
    then have pos: exponent f < e by simp
    then have 2 powr (exponent f − e) = 2 powr − real-of-int (e − exponent f)
      by simp
    also have ... = 1 / 2^nat (e − exponent f)
      using pos by (simp add: powr-realpow[symmetric] powr-divide2[symmetric])
    finally have m * 2^nat (e − exponent f) = real-of-int (mantissa f)
      using eq by simp
    then have mantissa f = m * 2^nat (e − exponent f)
      by linarith
    with ‹exponent f < e› have 2 dvd mantissa f
      apply (intro dvdI[where k=m * 2^(nat (e−exponent f)) div 2])
      apply (cases nat (e − exponent f))
      apply auto
      done
    then show False using mantissa-not-dvd[OF not-0] by simp
  qed
  ultimately have real-of-int m = mantissa f * 2^nat (exponent f − e)
```

by (*simp add*: *powr-realpow*[*symmetric*])
with ‹*e* ≤ *exponent f*›
show *m* = *mantissa f* ∗ *2* ˆ *nat* (*exponent f* − *e*)
by *linarith*
show *e* = *exponent f* − *nat* (*exponent f* − *e*)
using ‹*e* ≤ *exponent f*› **by** *auto*
qed

context
begin

qualified lemma *compute-float-zero*[*code-unfold*, *code*]: *0* = *Float 0 0*
by *transfer simp*

qualified lemma *compute-float-one*[*code-unfold*, *code*]: *1* = *Float 1 0*
by *transfer simp*

lift-definition *normfloat* :: *float* ⇒ *float* **is** λ*x*. *x* **.**
lemma *normloat-id*[*simp*]: *normfloat x* = *x* **by** *transfer rule*

qualified lemma *compute-normfloat*[*code*]: *normfloat* (*Float m e*) =
(*if m mod 2* = *0* ∧ *m* ≠ *0 then normfloat* (*Float* (*m div 2*) (*e* + *1*))
else if m = *0 then 0 else Float m e*)
by *transfer* (*auto simp add*: *powr-add zmod-eq-0-iff*)

qualified lemma *compute-float-numeral*[*code-abbrev*]: *Float* (*numeral k*) *0* = *numeral k*
by *transfer simp*

qualified lemma *compute-float-neg-numeral*[*code-abbrev*]: *Float* (− *numeral k*) *0* = − *numeral k*
by *transfer simp*

qualified lemma *compute-float-uminus*[*code*]: − *Float m1 e1* = *Float* (− *m1*) *e1*
by *transfer simp*

qualified lemma *compute-float-times*[*code*]: *Float m1 e1* ∗ *Float m2 e2* = *Float* (*m1* ∗ *m2*) (*e1* + *e2*)
by *transfer* (*simp add*: *field-simps powr-add*)

qualified lemma *compute-float-plus*[*code*]: *Float m1 e1* + *Float m2 e2* =
(*if m1* = *0 then Float m2 e2 else if m2* = *0 then Float m1 e1 else*
if e1 ≤ *e2 then Float* (*m1* + *m2* ∗ *2*ˆ*nat* (*e2* − *e1*)) *e1*
else Float (*m2* + *m1* ∗ *2*ˆ*nat* (*e1* − *e2*)) *e2*)
by *transfer* (*simp add*: *field-simps powr-realpow*[*symmetric*] *powr-divide2*[*symmetric*])

qualified lemma *compute-float-minus*[*code*]: **fixes** *f g*::*float* **shows** *f* − *g* = *f* + (−*g*)
by *simp*

qualified lemma *compute-float-sgn*[*code*]: *sgn* (*Float m1 e1*) = (*if 0 < m1 then 1 else if m1 < 0 then −1 else 0*)
by *transfer* (*simp add*: *sgn-times*)

lift-definition *is-float-pos* :: *float* ⇒ *bool* **is** *op* < *0* :: *real* ⇒ *bool* **.**

qualified lemma *compute-is-float-pos*[*code*]: *is-float-pos* (*Float m e*) ⟷ *0* < *m*
by *transfer* (*auto simp add*: *zero-less-mult-iff not-le*[*symmetric*, *of - 0*])

lift-definition *is-float-nonneg* :: *float* ⇒ *bool* **is** *op* ≤ *0* :: *real* ⇒ *bool* **.**

qualified lemma *compute-is-float-nonneg*[*code*]: *is-float-nonneg* (*Float m e*) ⟷ *0* ≤ *m*
by *transfer* (*auto simp add*: *zero-le-mult-iff not-less*[*symmetric*, *of - 0*])

lift-definition *is-float-zero* :: *float* ⇒ *bool* **is** *op* = *0* :: *real* ⇒ *bool* **.**

qualified lemma *compute-is-float-zero*[*code*]: *is-float-zero* (*Float m e*) ⟷ *0* = *m*
by *transfer* (*auto simp add*: *is-float-zero-def*)

qualified lemma *compute-float-abs*[*code*]: |*Float m e*| = *Float* |*m*| *e*
by *transfer* (*simp add*: *abs-mult*)

qualified lemma *compute-float-eq*[*code*]: *equal-class.equal f g* = *is-float-zero* (*f* − *g*)
by *transfer simp*

end

39.6 Lemmas for types *real*, *nat*, *int*

lemmas *real-of-ints* =
of-int-add
of-int-minus
of-int-diff
of-int-mult
of-int-power
of-int-numeral of-int-neg-numeral

lemmas *int-of-reals* = *real-of-ints*[*symmetric*]

39.7 Rounding Real Numbers

definition *round-down* :: *int* ⇒ *real* ⇒ *real*
where *round-down prec x* = ⌊*x* ∗ *2 powr prec*⌋ ∗ *2 powr* −*prec*

definition *round-up* :: *int* ⇒ *real* ⇒ *real*
where *round-up prec x* = ⌈*x* ∗ *2 powr prec*⌉ ∗ *2 powr* −*prec*

lemma *round-down-float*[*simp*]: *round-down prec x* ∈ *float*
unfolding *round-down-def*
by (*auto intro*!: *times-float simp*: *of-int-minus*[*symmetric*] *simp del*: *of-int-minus*)

lemma *round-up-float*[*simp*]: *round-up prec x* ∈ *float*
unfolding *round-up-def*
by (*auto intro*!: *times-float simp*: *of-int-minus*[*symmetric*] *simp del*: *of-int-minus*)

lemma *round-up*: $x \leq$ *round-up prec x*
by (*simp add*: *powr-minus-divide le-divide-eq round-up-def ceiling-correct*)

lemma *round-down*: *round-down prec x* $\leq x$
by (*simp add*: *powr-minus-divide divide-le-eq round-down-def*)

lemma *round-up-0*[*simp*]: *round-up p 0* = *0*
unfolding *round-up-def* **by** *simp*

lemma *round-down-0*[*simp*]: *round-down p 0* = *0*
unfolding *round-down-def* **by** *simp*

lemma *round-up-diff-round-down*:
round-up prec x − *round-down prec x* ≤ *2 powr* −*prec*
proof −
have *round-up prec x* − *round-down prec x* =
(⌈*x* ∗ *2 powr prec*⌉ − ⌊*x* ∗ *2 powr prec*⌋) ∗ *2 powr* −*prec*
by (*simp add*: *round-up-def round-down-def field-simps*)
also have ... ≤ *1* ∗ *2 powr* −*prec*
by (*rule mult-mono*)
(*auto simp del*: *of-int-diff*
simp: *of-int-diff*[*symmetric*] *ceiling-diff-floor-le-1*)
finally show *?thesis* **by** *simp*
qed

lemma *round-down-shift*: *round-down p* (*x* ∗ *2 powr k*) = *2 powr k* ∗ *round-down* (*p* + *k*) *x*
unfolding *round-down-def*
by (*simp add*: *powr-add powr-mult field-simps powr-divide2*[*symmetric*])
(*simp add*: *powr-add*[*symmetric*])

lemma *round-up-shift*: *round-up p* (*x* ∗ *2 powr k*) = *2 powr k* ∗ *round-up* (*p* + *k*) *x*
unfolding *round-up-def*
by (*simp add*: *powr-add powr-mult field-simps powr-divide2*[*symmetric*])
(*simp add*: *powr-add*[*symmetric*])

lemma *round-up-uminus-eq*: *round-up p* (−*x*) = − *round-down p x*
and *round-down-uminus-eq*: *round-down p* (−*x*) = − *round-up p x*
by (*auto simp*: *round-up-def round-down-def ceiling-def*)

lemma *round-up-mono*: $x \leq y \Longrightarrow$ *round-up p x* $\leq$ *round-up p y*
  by (*auto intro*!: *ceiling-mono simp*: *round-up-def*)

lemma *round-up-le1*:
  assumes $x \leq 1$ *prec* ≥ 0
  shows *round-up prec x* ≤ 1
proof −
  have *real-of-int* $\lceil x * 2$ *powr prec*$\rceil \leq$ *real-of-int* $\lceil 2$ *powr real-of-int prec*$\rceil$
    using *assms* **by** (*auto intro*!: *ceiling-mono*)
  also have ... = *2 powr prec* **using** *assms* **by** (*auto simp*: *powr-int intro*!: *exI*[**where** *x=2^nat prec*])
  finally show *?thesis*
    by (*simp add*: *round-up-def*) (*simp add*: *powr-minus inverse-eq-divide*)
qed

lemma *round-up-less1*:
  assumes $x < 1 / 2$ $p > 0$
  shows *round-up p x* < 1
proof −
  have $x * 2$ *powr p* $< 1 / 2 * 2$ *powr p*
    using *assms* **by** *simp*
  also have ... ≤ 2 *powr p* $- 1$ **using** ‹$p > 0$›
    by (*auto simp*: *powr-divide2*[*symmetric*] *powr-int field-simps self-le-power*)
  finally show *?thesis* **using** ‹$p > 0$›
    by (*simp add*: *round-up-def field-simps powr-minus powr-int ceiling-less-iff*)
qed

lemma *round-down-ge1*:
  assumes *x*: $x \geq 1$
  assumes *prec*: $p \geq -$ *log 2 x*
  shows $1 \leq$ *round-down p x*
proof *cases*
  assume *nonneg*: $0 \leq p$
  have *2 powr p* = *real-of-int* $\lfloor 2$ *powr real-of-int p*$\rfloor$
    using *nonneg* **by** (*auto simp*: *powr-int*)
  also have ... $\leq$ *real-of-int* $\lfloor x * 2$ *powr p*$\rfloor$
    using *assms* **by** (*auto intro*!: *floor-mono*)
  finally show *?thesis*
    by (*simp add*: *round-down-def*) (*simp add*: *powr-minus inverse-eq-divide*)
next
  assume *neg*: $\neg\ 0 \leq p$
  have *x* = *2 powr* (*log 2 x*)
    using *x* **by** *simp*
  also have *2 powr* (*log 2 x*) ≥ 2 *powr* $-$ *p*
    using *prec* **by** *auto*
  finally have *x-le*: $x \geq 2$ *powr* $-p$ **.**

  from *neg* **have** *2 powr real-of-int p* ≤ 2 *powr 0*

by (*intro powr-mono*) *auto*
also have ... $\leq \lfloor 2\ powr\ 0::real \rfloor$ **by** *simp*
also have ... $\leq \lfloor x * 2\ powr\ (real\text{-}of\text{-}int\ p) \rfloor$
unfolding *of-int-le-iff*
using *x x-le* **by** (*intro floor-mono*) (*simp add*: *powr-minus-divide field-simps*)
finally show *?thesis*
using *prec x*
by (*simp add*: *round-down-def powr-minus-divide pos-le-divide-eq*)
qed

lemma *round-up-le0*: $x \leq 0 \Longrightarrow round\text{-}up\ p\ x \leq 0$
unfolding *round-up-def*
by (*auto simp*: *field-simps mult-le-0-iff zero-le-mult-iff*)

39.8 Rounding Floats

definition *div-twopow* :: *int* ⇒ *nat* ⇒ *int*
where [*simp*]: *div-twopow x n* = *x div* (*2* ˆ *n*)

definition *mod-twopow* :: *int* ⇒ *nat* ⇒ *int*
where [*simp*]: *mod-twopow x n* = *x mod* (*2* ˆ *n*)

lemma *compute-div-twopow*[*code*]:
div-twopow x n = (*if x* = *0* ∨ *x* = −*1* ∨ *n* = *0 then x else div-twopow* (*x div 2*) (*n* − *1*))
by (*cases n*) (*auto simp*: *zdiv-zmult2-eq div-eq-minus1*)

lemma *compute-mod-twopow*[*code*]:
mod-twopow x n = (*if n* = *0 then 0 else x mod 2* + *2* ∗ *mod-twopow* (*x div 2*) (*n* − *1*))
by (*cases n*) (*auto simp*: *zmod-zmult2-eq*)

lift-definition *float-up* :: *int* ⇒ *float* ⇒ *float* **is** *round-up* **by** *simp*
declare *float-up.rep-eq*[*simp*]

lemma *round-up-correct*: *round-up e f* − *f* ∈ {*0..2 powr* −*e*}
unfolding *atLeastAtMost-iff*
proof
have *round-up e f* − *f* ≤ *round-up e f* − *round-down e f*
using *round-down* **by** *simp*
also have ... ≤ *2 powr* −*e*
using *round-up-diff-round-down* **by** *simp*
finally show *round-up e f* − *f* ≤ *2 powr* − (*real-of-int e*)
by *simp*
qed (*simp add*: *algebra-simps round-up*)

lemma *float-up-correct*: *real-of-float* (*float-up e f*) − *real-of-float f* ∈ {*0..2 powr* −*e*}
by *transfer* (*rule round-up-correct*)

lift-definition *float-down* :: *int* ⇒ *float* ⇒ *float* **is** *round-down* **by** *simp*
declare *float-down.rep-eq*[*simp*]

lemma *round-down-correct*: $f - (round\text{-}down\ e\ f) \in \{0..2\ powr\ -e\}$
  unfolding *atLeastAtMost-iff*
proof
  have $f - round\text{-}down\ e\ f \leq round\text{-}up\ e\ f - round\text{-}down\ e\ f$
    using *round-up* **by** *simp*
  also have $\ldots \leq 2\ powr\ -e$
    using *round-up-diff-round-down* **by** *simp*
  finally show $f - round\text{-}down\ e\ f \leq 2\ powr - (real\text{-}of\text{-}int\ e)$
    by *simp*
qed (*simp add*: *algebra-simps round-down*)

lemma *float-down-correct*: $real\text{-}of\text{-}float\ f - real\text{-}of\text{-}float\ (float\text{-}down\ e\ f) \in \{0..2\ powr\ -e\}$
  by *transfer* (*rule round-down-correct*)

context
begin

qualified lemma *compute-float-down*[*code*]:
  $float\text{-}down\ p\ (Float\ m\ e) =$
    $(if\ p + e < 0\ then\ Float\ (div\text{-}twopow\ m\ (nat\ (-(p + e))))\ (-p)\ else\ Float\ m\ e)$
proof $(cases\ p + e < 0)$
  case *True*
  then have $real\text{-}of\text{-}int\ ((2::int)\ \hat{}\ nat\ (-(p + e))) = 2\ powr\ (-(p + e))$
    using *powr-realpow*[*of 2 nat* $(-(p + e))$] **by** *simp*
  also have $\ldots = 1\ /\ 2\ powr\ p\ /\ 2\ powr\ e$
    unfolding *powr-minus-divide of-int-minus* **by** (*simp add*: *powr-add*)
  finally show *?thesis*
    using ‹$p + e < 0$›
    apply *transfer*
    apply (*simp add*: *ac-simps round-down-def floor-divide-of-int-eq*[*symmetric*])
    proof −
      fix *pa* :: *int* **and** *ea* :: *int* **and** *ma* :: *int*
       assume *a1*: $2\ \hat{}\ nat\ (-\ pa - ea) = 1\ /\ (2\ powr\ real\text{-}of\text{-}int\ pa * 2\ powr\ real\text{-}of\text{-}int\ ea)$
      assume $pa + ea < 0$
      have $\lfloor real\text{-}of\text{-}int\ ma\ /\ real\text{-}of\text{-}int\ (int\ 2\ \hat{}\ nat\ (-\ (pa + ea)))\rfloor = \lfloor real\text{-}of\text{-}float\ (Float\ ma\ (pa + ea))\rfloor$
        using *a1* **by** (*simp add*: *powr-add*)
      thus $\lfloor real\text{-}of\text{-}int\ ma * (2\ powr\ real\text{-}of\text{-}int\ pa * 2\ powr\ real\text{-}of\text{-}int\ ea)\rfloor = ma\ div\ 2\ \hat{}\ nat\ (-\ pa - ea)$
      by (*metis Float.rep-eq add-uminus-conv-diff floor-divide-of-int-eq minus-add-distrib of-int-simps*(*3*) *of-nat-numeral powr-add*)
    qed

next
  case *False*
  then have *r*: *real-of-int e* + *real-of-int p* = *real* (*nat* (*e* + *p*)) **by** *simp*
  have *r*: ⌊(*m* ∗ *2 powr e*) ∗ *2 powr real-of-int p*⌋ = (*m* ∗ *2 powr e*) ∗ *2 powr real-of-int p*
    by (*auto intro*: *exI*[**where** *x*=*m*∗*2*ˆ*nat* (*e*+*p*)]
          simp add: *ac-simps powr-add*[*symmetric*] *r powr-realpow*)
  with ‹¬ *p* + *e* < *0*› **show** *?thesis*
    by *transfer* (*auto simp add*: *round-down-def field-simps powr-add powr-minus*)
qed

lemma *abs-round-down-le*: |*f* − (*round-down e f*)| ≤ *2 powr* −*e*
  using *round-down-correct*[*of f e*] **by** *simp*

lemma *abs-round-up-le*: |*f* − (*round-up e f*)| ≤ *2 powr* −*e*
  using *round-up-correct*[*of e f*] **by** *simp*

lemma *round-down-nonneg*: *0* ≤ *s* ⟹ *0* ≤ *round-down p s*
  by (*auto simp*: *round-down-def*)

lemma *ceil-divide-floor-conv*:
  assumes *b* ≠ *0*
  shows ⌈*real-of-int a* / *real-of-int b*⌉ = (*if b dvd a then a div b else* ⌊*real-of-int a* / *real-of-int b*⌋ + *1*)
proof (*cases b dvd a*)
  case *True*
  then show *?thesis*
    by (*simp add*: *ceiling-def of-int-minus*[*symmetric*] *divide-minus-left*[*symmetric*]
      floor-divide-of-int-eq dvd-neg-div del: *divide-minus-left of-int-minus*)
next
  case *False*
  then have *a mod b* ≠ *0*
    by *auto*
  then have *ne*: *real-of-int* (*a mod b*) / *real-of-int b* ≠ *0*
    using ‹*b* ≠ *0*› **by** *auto*
  have ⌈*real-of-int a* / *real-of-int b*⌉ = ⌊*real-of-int a* / *real-of-int b*⌋ + *1*
    apply (*rule ceiling-eq*)
    apply (*auto simp*: *floor-divide-of-int-eq*[*symmetric*])
  proof −
    have *real-of-int* ⌊*real-of-int a* / *real-of-int b*⌋ ≤ *real-of-int a* / *real-of-int b*
      by *simp*
     moreover have *real-of-int* ⌊*real-of-int a* / *real-of-int b*⌋ ≠ *real-of-int a* / *real-of-int b*
      apply (*subst* (*2*) *real-of-int-div-aux*)
      unfolding *floor-divide-of-int-eq*
      using *ne* ‹*b* ≠ *0*› **apply** *auto*
      done
     ultimately show *real-of-int* ⌊*real-of-int a* / *real-of-int b*⌋ < *real-of-int a* / *real-of-int b* **by** *arith*

qed
then show *?thesis*
using ‹¬ *b dvd a*› **by** *simp*
qed

qualified lemma *compute-float-up*[*code*]: *float-up p x* = − *float-down p* (−*x*)
by *transfer* (*simp add*: *round-down-uminus-eq*)

end

39.9 Compute bitlen of integers

definition *bitlen* :: *int* ⇒ *int*
where *bitlen a* = (*if a* > *0 then* ⌊*log 2 a*⌋ + *1 else 0*)

lemma *bitlen-nonneg*: *0* ≤ *bitlen x*
proof −
have −*1* < *log 2* (−*x*) **if** *0* > *x*
proof −
have −*1* = *log 2* (*inverse 2*)
by (*subst log-inverse*) *simp-all*
also have ... < *log 2* (−*x*)
using ‹*0* > *x*› **by** *auto*
finally show *?thesis* **.**
qed
then show *?thesis*
unfolding *bitlen-def* **by** (*auto intro*!: *add-nonneg-nonneg*)
qed

lemma *bitlen-bounds*:
assumes *x* > *0*
shows *2* ^ *nat* (*bitlen x* − *1*) ≤ *x* ∧ *x* < *2* ^ *nat* (*bitlen x*)
proof
show *2* ^ *nat* (*bitlen x* − *1*) ≤ *x*
proof −
have (*2*::*real*) ^ *nat* ⌊*log 2* (*real-of-int x*)⌋ = *2 powr real-of-int* ⌊*log 2* (*real-of-int x*)⌋
using *powr-realpow*[*symmetric*, *of 2 nat* ⌊*log 2* (*real-of-int x*)⌋] ‹*x* > *0*›
by *simp*
also have ... ≤ *2 powr log 2* (*real-of-int x*)
by *simp*
also have ... = *real-of-int x*
using ‹*0* < *x*› **by** *simp*
finally have *2* ^ *nat* ⌊*log 2* (*real-of-int x*)⌋ ≤ *real-of-int x*
by *simp*
then show *?thesis*
using ‹*0* < *x*› **by** (*simp add*: *bitlen-def*)
qed
show *x* < *2* ^ *nat* (*bitlen x*)

proof −
have *x* ≤ *2 powr* (*log 2 x*)
using ‹*x* > *0*› **by** *simp*
also have . . . < *2* ˆ *nat* (⌊*log 2* (*real-of-int x*)⌋ + *1*)
apply (*simp add*: *powr-realpow*[*symmetric*])
using ‹*x* > *0*› **apply** *simp*
done
finally show *?thesis*
using ‹*x* > *0*› **by** (*simp add*: *bitlen-def ac-simps*)
qed
qed

lemma *bitlen-pow2*[*simp*]:
assumes *b* > *0*
shows *bitlen* (*b* ∗ *2* ˆ *c*) = *bitlen b* + *c*
proof −
from *assms* **have** *b* ∗ *2* ˆ *c* > *0*
by *auto*
then show *?thesis*
using *floor-add*[*of log 2 b c*] *assms*
apply (*auto simp add*: *log-mult log-nat-power bitlen-def*)
by (*metis add.right-neutral frac-lt-1 frac-of-int of-int-of-nat-eq*)
qed

lemma *bitlen-Float*:
fixes *m e*
defines *f* ≡ *Float m e*
shows *bitlen* (|*mantissa f*|) + *exponent f* = (*if m* = *0 then 0 else bitlen* |*m*| + *e*)
proof (*cases m* = *0*)
case *True*
then show *?thesis* **by** (*simp add*: *f-def bitlen-def Float-def*)
next
case *False*
then have *f* ≠ *float-of 0*
unfolding *real-of-float-eq* **by** (*simp add*: *f-def*)
then have *mantissa f* ≠ *0*
by (*simp add*: *mantissa-noteq-0*)
moreover
obtain *i* **where** *m* = *mantissa f* ∗ *2* ˆ *i e* = *exponent f* − *int i*
by (*rule f-def*[*THEN denormalize-shift*, *OF* ‹*f* ≠ *float-of 0*›])
ultimately show *?thesis* **by** (*simp add*: *abs-mult*)
qed

context
begin

qualified lemma *compute-bitlen*[*code*]: *bitlen x* = (*if x* > *0 then bitlen* (*x div 2*) + *1 else 0*)

proof –
  { **assume** $2 \le x$
    then have $\lfloor log\ 2\ (x\ div\ 2)\rfloor + 1 = \lfloor log\ 2\ (x - x\ mod\ 2)\rfloor$
      by (*simp add*: *log-mult zmod-zdiv-equality'*)
    also have $\ldots = \lfloor log\ 2\ (real\text{-}of\text{-}int\ x)\rfloor$
    proof (*cases* $x\ mod\ 2 = 0$)
      case *True*
      then show *?thesis* **by** *simp*
    next
      case *False*
      def $n \equiv \lfloor log\ 2\ (real\text{-}of\text{-}int\ x)\rfloor$
      then have $0 \le n$
        using ‹$2 \le x$› **by** *simp*
      from ‹$2 \le x$› *False* **have** $x\ mod\ 2 = 1\ \neg\ 2\ dvd\ x$
        by (*auto simp add*: *dvd-eq-mod-eq-0*)
      with ‹$2 \le x$› **have** $x \ne 2\ \hat{}\ nat\ n$
        by (*cases nat n*) *auto*
      moreover
      { **have** $real\text{-}of\text{-}int\ (2\hat{}nat\ n :: int) = 2\ powr\ (nat\ n)$
          by (*simp add*: *powr-realpow*)
        also have $\ldots \le 2\ powr\ (log\ 2\ x)$
          using ‹$2 \le x$› **by** (*simp add*: *n-def del*: *powr-log-cancel*)
        finally have $2\hat{}nat\ n \le x$ **using** ‹$2 \le x$› **by** *simp* }
      ultimately have $2\hat{}nat\ n \le x - 1$ **by** *simp*
      then have $2\hat{}nat\ n \le real\text{-}of\text{-}int\ (x - 1)$
        using *numeral-power-le-real-of-int-cancel-iff* **by** *blast*
      { **have** $n = \lfloor log\ 2\ (2\hat{}nat\ n)\rfloor$
          using ‹$0 \le n$› **by** (*simp add*: *log-nat-power*)
        also have $\ldots \le \lfloor log\ 2\ (x - 1)\rfloor$
          using ‹$2\hat{}nat\ n \le real\text{-}of\text{-}int\ (x - 1)$› ‹$0 \le n$› ‹$2 \le x$› **by** (*auto intro*: *floor-mono*)
        finally have $n \le \lfloor log\ 2\ (x - 1)\rfloor$ **.** }
      moreover have $\lfloor log\ 2\ (x - 1)\rfloor \le n$
        using ‹$2 \le x$› **by** (*auto simp add*: *n-def intro*!: *floor-mono*)
      ultimately show $\lfloor log\ 2\ (x - x\ mod\ 2)\rfloor = \lfloor log\ 2\ x\rfloor$
        unfolding *n-def* ‹$x\ mod\ 2 = 1$› **by** *auto*
    qed
    finally have $\lfloor log\ 2\ (x\ div\ 2)\rfloor + 1 = \lfloor log\ 2\ x\rfloor$ **.** }
  moreover
  { **assume** $x < 2\ 0 < x$
    then have $x = 1$ **by** *simp*
    then have $\lfloor log\ 2\ (real\text{-}of\text{-}int\ x)\rfloor = 0$ **by** *simp* }
  ultimately show *?thesis*
    unfolding *bitlen-def*
    by (*auto simp*: *pos-imp-zdiv-pos-iff not-le*)
qed

end

lemma *float-gt1-scale*: **assumes** *1* ≤ *Float m e*
  shows *0* ≤ *e* + (*bitlen m* − *1*)
proof −
  have *0* < *Float m e* **using** *assms* **by** *auto*
  then have *0* < *m* **using** *powr-gt-zero*[*of 2 e*]
    apply (*auto simp*: *zero-less-mult-iff*)
    using *not-le powr-ge-pzero* **apply** *blast*
    done
  then have *m* ≠ *0* **by** *auto*
  show *?thesis*
  proof (*cases 0* ≤ *e*)
    case *True*
    then show *?thesis*
      using ‹*0* < *m*› **by** (*simp add*: *bitlen-def*)
  next
    case *False*
    have (*1*::*int*) < *2* **by** *simp*
    let *?S* = *2*ˆ(*nat* (−*e*))
    have *inverse* (*2* ˆ *nat* (− *e*)) = *2 powr e*
      using *assms False powr-realpow*[*of 2 nat* (−*e*)]
      by (*auto simp*: *powr-minus field-simps*)
    then have *1* ≤ *real-of-int m* ∗ *inverse ?S*
      using *assms False powr-realpow*[*of 2 nat* (−*e*)]
      by (*auto simp*: *powr-minus*)
    then have *1* ∗ *?S* ≤ *real-of-int m* ∗ *inverse ?S* ∗ *?S*
      by (*rule mult-right-mono*) *auto*
    then have *?S* ≤ *real-of-int m*
      unfolding *mult.assoc* **by** *auto*
    then have *?S* ≤ *m*
      unfolding *of-int-le-iff*[*symmetric*] **by** *auto*
    from *this bitlen-bounds*[*OF* ‹*0* < *m*›, *THEN conjunct2*]
    have *nat* (−*e*) < (*nat* (*bitlen m*))
      unfolding *power-strict-increasing-iff*[*OF* ‹*1* < *2*›, *symmetric*]
      by (*rule order-le-less-trans*)
    then have −*e* < *bitlen m*
      using *False* **by** *auto*
    then show *?thesis*
      by *auto*
  qed
qed

lemma *bitlen-div*:
  assumes *0* < *m*
  shows *1* ≤ *real-of-int m* / *2*ˆ*nat* (*bitlen m* − *1*)
    and *real-of-int m* / *2*ˆ*nat* (*bitlen m* − *1*) < *2*
proof −
  let *?B* = *2*ˆ*nat*(*bitlen m* − *1*)

  have *?B* ≤ *m* **using** *bitlen-bounds*[*OF* ‹*0* <*m*›] **..**

then have *1 ∗ ?B ≤ real-of-int m*
unfolding *of-int-le-iff* [*symmetric*] **by** *auto*
then show *1 ≤ real-of-int m / ?B*
by *auto*

have *m ≠ 0*
using *assms* **by** *auto*
have *0 ≤ bitlen m − 1*
using ‹*0 < m*› **by** (*auto simp*: *bitlen-def*)

have *m < 2^nat*(*bitlen m*)
using *bitlen-bounds*[*OF* ‹*0 <m*›] **..**
also have ... = *2^nat*(*bitlen m − 1 + 1*)
using ‹*0 < m*› **by** (*auto simp*: *bitlen-def*)
also have ... = *?B ∗ 2*
unfolding *nat-add-distrib*[*OF* ‹*0 ≤ bitlen m − 1*› *zero-le-one*] **by** *auto*
finally have *real-of-int m < 2 ∗ ?B*
by (*metis* (*full-types*) *mult.commute power.simps*(*2*) *real-of-int-less-numeral-power-cancel-iff*)
then have *real-of-int m / ?B < 2 ∗ ?B / ?B*
by (*rule divide-strict-right-mono*) *auto*
then show *real-of-int m / ?B < 2*
by *auto*
qed

39.10 Truncating Real Numbers

definition *truncate-down*::*nat ⇒ real ⇒ real*
where *truncate-down prec x = round-down* (*prec −* ⌊*log 2* |*x*|⌋) *x*

lemma *truncate-down*: *truncate-down prec x ≤ x*
using *round-down* **by** (*simp add*: *truncate-down-def*)

lemma *truncate-down-le*: *x ≤ y ⟹ truncate-down prec x ≤ y*
by (*rule order-trans*[*OF truncate-down*])

lemma *truncate-down-zero*[*simp*]: *truncate-down prec 0 = 0*
by (*simp add*: *truncate-down-def*)

lemma *truncate-down-float*[*simp*]: *truncate-down p x ∈ float*
by (*auto simp*: *truncate-down-def*)

definition *truncate-up*::*nat ⇒ real ⇒ real*
where *truncate-up prec x = round-up* (*prec −* ⌊*log 2* |*x*|⌋) *x*

lemma *truncate-up*: *x ≤ truncate-up prec x*
using *round-up* **by** (*simp add*: *truncate-up-def*)

lemma *truncate-up-le*: *x ≤ y ⟹ x ≤ truncate-up prec y*
by (*rule order-trans*[*OF - truncate-up*])

lemma *truncate-up-zero*[*simp*]: *truncate-up prec 0* = *0*
 by (*simp add*: *truncate-up-def*)

lemma *truncate-up-uminus-eq*: *truncate-up prec* (−*x*) = − *truncate-down prec x*
 and *truncate-down-uminus-eq*: *truncate-down prec* (−*x*) = − *truncate-up prec x*
 by (*auto simp*: *truncate-up-def round-up-def truncate-down-def round-down-def ceiling-def*)

lemma *truncate-up-float*[*simp*]: *truncate-up p x* ∈ *float*
 by (*auto simp*: *truncate-up-def*)

lemma *mult-powr-eq*: *0* < *b* ⟹ *b* ≠ *1* ⟹ *0* < *x* ⟹ *x* ∗ *b powr y* = *b powr* (*y* + *log b x*)
 by (*simp-all add*: *powr-add*)

lemma *truncate-down-pos*:
 assumes *x* > *0*
 shows *truncate-down p x* > *0*
proof −
 have *0* ≤ *log 2 x* − *real-of-int* ⌊*log 2 x*⌋
  by (*simp add*: *algebra-simps*)
 with *assms*
 show *?thesis*
  apply (*auto simp*: *truncate-down-def round-down-def mult-powr-eq*
   intro!: *ge-one-powr-ge-zero mult-pos-pos*)
  by *linarith*
qed

lemma *truncate-down-nonneg*: *0* ≤ *y* ⟹ *0* ≤ *truncate-down prec y*
 by (*auto simp*: *truncate-down-def round-down-def*)

lemma *truncate-down-ge1*: *1* ≤ *x* ⟹ *1* ≤ *truncate-down p x*
 apply (*auto simp*: *truncate-down-def algebra-simps intro*!: *round-down-ge1*)
 apply *linarith*
 done

lemma *truncate-up-nonpos*: *x* ≤ *0* ⟹ *truncate-up prec x* ≤ *0*
 by (*auto simp*: *truncate-up-def round-up-def intro*!: *mult-nonpos-nonneg*)

lemma *truncate-up-le1*:
 assumes *x* ≤ *1*
 shows *truncate-up p x* ≤ *1*
proof −
 consider *x* ≤ *0* | *x* > *0*
  by *arith*
 then show *?thesis*
 proof *cases*
  case *1*

with *truncate-up-nonpos*[*OF this*, *of p*] **show** *?thesis*
by *simp*
next
case *2*
then have *le*: ⌊*log 2* |*x*|⌋ ≤ *0*
using *assms* **by** (*auto simp*: *log-less-iff*)
from *assms* **have** *0* ≤ *int p* **by** *simp*
from *add-mono*[*OF this le*]
show *?thesis*
using *assms* **by** (*simp add*: *truncate-up-def round-up-le1 add-mono*)
qed
qed

lemma *truncate-down-shift-int*: *truncate-down p* (*x* ∗ *2 powr real-of-int k*) = *truncate-down p x* ∗ *2 powr k*
by (*cases x* = *0*)
(*simp-all add*: *algebra-simps abs-mult log-mult truncate-down-def round-down-shift*[*of - - k*, *simplified*])

lemma *truncate-down-shift-nat*: *truncate-down p* (*x* ∗ *2 powr real k*) = *truncate-down p x* ∗ *2 powr k*
by (*metis of-int-of-nat-eq truncate-down-shift-int*)

lemma *truncate-up-shift-int*: *truncate-up p* (*x* ∗ *2 powr real-of-int k*) = *truncate-up p x* ∗ *2 powr k*
by (*cases x* = *0*)
(*simp-all add*: *algebra-simps abs-mult log-mult truncate-up-def round-up-shift*[*of - - k*, *simplified*])

lemma *truncate-up-shift-nat*: *truncate-up p* (*x* ∗ *2 powr real k*) = *truncate-up p x* ∗ *2 powr k*
by (*metis of-int-of-nat-eq truncate-up-shift-int*)

39.11 Truncating Floats

lift-definition *float-round-up* :: *nat* ⇒ *float* ⇒ *float* **is** *truncate-up*
by (*simp add*: *truncate-up-def*)

lemma *float-round-up*: *real-of-float x* ≤ *real-of-float* (*float-round-up prec x*)
using *truncate-up* **by** *transfer simp*

lemma *float-round-up-zero*[*simp*]: *float-round-up prec 0* = *0*
by *transfer simp*

lift-definition *float-round-down* :: *nat* ⇒ *float* ⇒ *float* **is** *truncate-down*
by (*simp add*: *truncate-down-def*)

lemma *float-round-down*: *real-of-float* (*float-round-down prec x*) ≤ *real-of-float x*
using *truncate-down* **by** *transfer simp*

lemma *float-round-down-zero*[*simp*]: *float-round-down prec 0 = 0*
by *transfer simp*

lemmas *float-round-up-le = order-trans*[*OF - float-round-up*]
and *float-round-down-le = order-trans*[*OF float-round-down*]

lemma *minus-float-round-up-eq*: − *float-round-up prec x = float-round-down prec* (− *x*)
and *minus-float-round-down-eq*: − *float-round-down prec x = float-round-up prec* (− *x*)
by (*transfer*, *simp add*: *truncate-down-uminus-eq truncate-up-uminus-eq*)+

context
begin

qualified lemma *compute-float-round-down*[*code*]:
float-round-down prec (*Float m e*) = (*let d = bitlen* |*m*| − *int prec* − *1 in*
if 0 < *d then Float* (*div-twopow m* (*nat d*)) (*e* + *d*)
else Float m e)
using *Float.compute-float-down*[*of Suc prec* − *bitlen* |*m*| − *e m e*, *symmetric*]
by *transfer*
(*simp add*: *field-simps abs-mult log-mult bitlen-def truncate-down-def*
cong del: *if-weak-cong*)

qualified lemma *compute-float-round-up*[*code*]:
float-round-up prec x = − *float-round-down prec* (−*x*)
by *transfer* (*simp add*: *truncate-down-uminus-eq*)

end

39.12 Approximation of positive rationals

lemma *div-mult-twopow-eq*:
fixes *a b* :: *nat*
shows *a div* ((*2*::*nat*) ˆ *n*) *div b = a div* (*b* ∗ *2* ˆ *n*)
by (*cases b = 0*) (*simp-all add*: *div-mult2-eq*[*symmetric*] *ac-simps*)

lemma *real-div-nat-eq-floor-of-divide*:
fixes *a b* :: *nat*
shows *a div b = real-of-int* ⌊*a* / *b*⌋
by (*simp add*: *floor-divide-of-nat-eq* [*of a b*])

definition *rat-precision prec x y* =
(*let d = bitlen x* − *bitlen y in int prec* − *d* +
(*if Float* (*abs x*) *0* < *Float* (*abs y*) *d then 1 else 0*))

lemma *floor-log-divide-eq*:
assumes *i* > *0 j* > *0 p* > *1*

shows ⌊*log p* (*i* / *j*)⌋ = *floor* (*log p i*) − *floor* (*log p j*) −
(*if i* ≥ *j* ∗ *p powr* (*floor* (*log p i*) − *floor* (*log p j*)) *then 0 else 1*)
proof −
let *?l* = *log p*
let *?fl* = λ*x*. *floor* (*?l x*)
have ⌊*?l* (*i* / *j*)⌋ = ⌊*?l i* − *?l j*⌋ **using** *assms*
by (*auto simp*: *log-divide*)
also have . . . = *floor* (*real-of-int* (*?fl i* − *?fl j*) + (*?l i* − *?fl i* − (*?l j* − *?fl j*)))
(**is** - = *floor* (- + *?r*))
by (*simp add*: *algebra-simps*)
also note *floor-add2*
also note ‹*p* > *1*›
note *powr* = *powr-le-cancel-iff* [*symmetric*, *OF* ‹*1* < *p*›, *THEN iffD2*]
note *powr-strict* = *powr-less-cancel-iff* [*symmetric*, *OF* ‹*1* < *p*›, *THEN iffD2*]
have *floor ?r* = (*if i* ≥ *j* ∗ *p powr* (*?fl i* − *?fl j*) *then 0 else* −*1*) (**is** - = *?if*)
using *assms*
by (*linarith* |
auto
intro!: *floor-eq2*
intro: *powr-strict powr*
simp: *powr-divide2* [*symmetric*] *powr-add divide-simps algebra-simps bitlen-def*)+
finally
show *?thesis* **by** *simp*
qed

lemma *truncate-down-rat-precision*:
truncate-down prec (*real x* / *real y*) = *round-down* (*rat-precision prec x y*) (*real x* / *real y*)
and *truncate-up-rat-precision*:
truncate-up prec (*real x* / *real y*) = *round-up* (*rat-precision prec x y*) (*real x* / *real y*)
unfolding *truncate-down-def truncate-up-def rat-precision-def*
by (*cases x*; *cases y*) (*auto simp*: *floor-log-divide-eq algebra-simps bitlen-def*)

lift-definition *lapprox-posrat* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *float*
is λ*prec* (*x*::*nat*) (*y*::*nat*). *truncate-down prec* (*x* / *y*)
by *simp*

context
begin

qualified lemma *compute-lapprox-posrat*[*code*]:
fixes *prec x y*
shows *lapprox-posrat prec x y* =
(*let*
l = *rat-precision prec x y*;
d = *if 0* ≤ *l then x* ∗ *2*ˆ*nat l div y else x div 2*ˆ*nat* (− *l*) *div y*
in normfloat (*Float d* (− *l*)))
unfolding *div-mult-twopow-eq*

```
    by transfer
      (simp add: round-down-def powr-int real-div-nat-eq-floor-of-divide field-simps
Let-def
        truncate-down-rat-precision del: two-powr-minus-int-float)

end

lift-definition rapprox-posrat :: nat ⇒ nat ⇒ nat ⇒ float
  is λprec (x::nat) (y::nat). truncate-up prec (x / y)
  by simp

context
begin

qualified lemma compute-rapprox-posrat[code]:
  fixes prec x y
  defines l ≡ rat-precision prec x y
  shows rapprox-posrat prec x y = (let
     l = l ;
     (r, s) = if 0 ≤ l then (x * 2^nat l, y) else (x, y * 2^nat(−l)) ;
     d = r div s ;
     m = r mod s
   in normfloat (Float (d + (if m = 0 ∨ y = 0 then 0 else 1)) (− l)))
proof (cases y = 0)
  assume y = 0
  then show ?thesis by transfer simp
next
  assume y ≠ 0
  show ?thesis
  proof (cases 0 ≤ l)
    case True
    def x' ≡ x * 2 ^ nat l
    have int x * 2 ^ nat l = x'
      by (simp add: x'-def of-nat-mult of-nat-power)
    moreover have real x * 2 powr l = real x'
      by (simp add: powr-realpow[symmetric] ‹0 ≤ l› x'-def)
    ultimately show ?thesis
      using ceil-divide-floor-conv[of y x'] powr-realpow[of 2 nat l] ‹0 ≤ l› ‹y ≠ 0›
        l-def[symmetric, THEN meta-eq-to-obj-eq]
      apply transfer
      apply (auto simp add: round-up-def truncate-up-rat-precision)
      by (metis floor-divide-of-int-eq of-int-of-nat-eq)
   next
    case False
    def y' ≡ y * 2 ^ nat (− l)
    from ‹y ≠ 0› have y' ≠ 0 by (simp add: y'-def)
   have int y * 2 ^ nat (− l) = y' by (simp add: y'-def of-nat-mult of-nat-power)
    moreover have real x * real-of-int (2::int) powr real-of-int l / real y = x /
real y'
```

using ‹¬ *0* ≤ *l*›
by (*simp add*: *powr-realpow*[*symmetric*] *powr-minus y′-def field-simps*)
ultimately show *?thesis*
using *ceil-divide-floor-conv*[*of y′ x*] ‹¬ *0* ≤ *l*› ‹*y′* ≠ *0*› ‹*y* ≠ *0*›
l-def[*symmetric*, *THEN meta-eq-to-obj-eq*]
apply *transfer*
apply (*auto simp add*: *round-up-def ceil-divide-floor-conv truncate-up-rat-precision*)
by (*metis floor-divide-of-int-eq of-int-of-nat-eq*)
qed
qed

end

lemma *rat-precision-pos*:
assumes *0* ≤ *x*
and *0* < *y*
and *2* ∗ *x* < *y*
shows *rat-precision n* (*int x*) (*int y*) > *0*
proof −
have *0* < *x* ⟹ *log 2 x* + *1* = *log 2* (*2* ∗ *x*)
by (*simp add*: *log-mult*)
then have *bitlen* (*int x*) < *bitlen* (*int y*)
using *assms*
by (*simp add*: *bitlen-def del*: *floor-add-one*)
(*auto intro*!: *floor-mono simp add*: *floor-add-one*[*symmetric*] *simp del*: *floor-add floor-add-one*)
then show *?thesis*
using *assms*
by (*auto intro*!: *pos-add-strict simp add*: *field-simps rat-precision-def*)
qed

lemma *rapprox-posrat-less1*:
0 ≤ *x* ⟹ *0* < *y* ⟹ *2* ∗ *x* < *y* ⟹ *real-of-float* (*rapprox-posrat n x y*) < *1*
by *transfer* (*simp add*: *rat-precision-pos round-up-less1 truncate-up-rat-precision*)

lift-definition *lapprox-rat* :: *nat* ⇒ *int* ⇒ *int* ⇒ *float* **is**
λprec (*x*::*int*) (*y*::*int*). *truncate-down prec* (*x* / *y*)
by *simp*

context
begin

qualified lemma *compute-lapprox-rat*[*code*]:
lapprox-rat prec x y =
(*if y* = *0 then 0*
else if 0 ≤ *x then*
(*if 0* < *y then lapprox-posrat prec* (*nat x*) (*nat y*)
else − (*rapprox-posrat prec* (*nat x*) (*nat* (−*y*))))
else (*if 0* < *y*

then − (*rapprox-posrat prec* (*nat* (−*x*)) (*nat y*))
else lapprox-posrat prec (*nat* (−*x*)) (*nat* (−*y*))))
by *transfer* (*simp add*: *truncate-up-uminus-eq*)

lift-definition *rapprox-rat* :: *nat* ⇒ *int* ⇒ *int* ⇒ *float* **is**
λ*prec* (*x*::*int*) (*y*::*int*). *truncate-up prec* (*x* / *y*)
by *simp*

lemma *rapprox-rat* = *rapprox-posrat*
by *transfer auto*

lemma *lapprox-rat* = *lapprox-posrat*
by *transfer auto*

qualified lemma *compute-rapprox-rat*[*code*]:
rapprox-rat prec x y = − *lapprox-rat prec* (−*x*) *y*
by *transfer* (*simp add*: *truncate-down-uminus-eq*)

qualified lemma *compute-truncate-down*[*code*]: *truncate-down p* (*Ratreal r*) = (*let* (*a*, *b*) = *quotient-of r in lapprox-rat p a b*)
by *transfer* (*auto split*: *prod.split simp*: *of-rat-divide dest*!: *quotient-of-div*)

qualified lemma *compute-truncate-up*[*code*]: *truncate-up p* (*Ratreal r*) = (*let* (*a*, *b*) = *quotient-of r in rapprox-rat p a b*)
by *transfer* (*auto split*: *prod.split simp*: *of-rat-divide dest*!: *quotient-of-div*)

end

39.13 Division

definition *real-divl prec a b* = *truncate-down prec* (*a* / *b*)

definition *real-divr prec a b* = *truncate-up prec* (*a* / *b*)

lift-definition *float-divl* :: *nat* ⇒ *float* ⇒ *float* ⇒ *float* **is** *real-divl*
by (*simp add*: *real-divl-def*)

context
begin

qualified lemma *compute-float-divl*[*code*]:
float-divl prec (*Float m1 s1*) (*Float m2 s2*) = *lapprox-rat prec m1 m2* ∗ *Float 1* (*s1* − *s2*)
apply *transfer*
unfolding *real-divl-def of-int-1 mult-1 truncate-down-shift-int*[*symmetric*]
apply (*simp add*: *powr-divide2*[*symmetric*] *powr-minus*)
done

lift-definition *float-divr* :: *nat* ⇒ *float* ⇒ *float* ⇒ *float* **is** *real-divr*

by (*simp add*: *real-divr-def*)

qualified lemma *compute-float-divr*[*code*]:
float-divr prec x y = − *float-divl prec* (−*x*) *y*
by *transfer* (*simp add*: *real-divr-def real-divl-def truncate-down-uminus-eq*)

end

39.14 Approximate Power

lemma *div2-less-self*[*termination-simp*]:
fixes *n* :: *nat*
shows *odd n* ⟹ *n div 2* < *n*
by (*simp add*: *odd-pos*)

fun *power-down* :: *nat* ⇒ *real* ⇒ *nat* ⇒ *real*
where
power-down p x 0 = *1*
| *power-down p x* (*Suc n*) =
(*if odd n then truncate-down* (*Suc p*) ((*power-down p x* (*Suc n div 2*))^2)
else truncate-down (*Suc p*) (*x* ∗ *power-down p x n*))

fun *power-up* :: *nat* ⇒ *real* ⇒ *nat* ⇒ *real*
where
power-up p x 0 = *1*
| *power-up p x* (*Suc n*) =
(*if odd n then truncate-up p* ((*power-up p x* (*Suc n div 2*))^2)
else truncate-up p (*x* ∗ *power-up p x n*))

lift-definition *power-up-fl* :: *nat* ⇒ *float* ⇒ *nat* ⇒ *float* **is** *power-up*
by (*induct-tac rule*: *power-up.induct*) *simp-all*

lift-definition *power-down-fl* :: *nat* ⇒ *float* ⇒ *nat* ⇒ *float* **is** *power-down*
by (*induct-tac rule*: *power-down.induct*) *simp-all*

lemma *power-float-transfer*[*transfer-rule*]:
(*rel-fun pcr-float* (*rel-fun op* = *pcr-float*)) *op* ˆ *op* ˆ
unfolding *power-def*
by *transfer-prover*

lemma *compute-power-up-fl*[*code*]:
power-up-fl p x 0 = *1*
power-up-fl p x (*Suc n*) =
(*if odd n then float-round-up p* ((*power-up-fl p x* (*Suc n div 2*))^2)
else float-round-up p (*x* ∗ *power-up-fl p x n*))
and *compute-power-down-fl*[*code*]:
power-down-fl p x 0 = *1*
power-down-fl p x (*Suc n*) =
(*if odd n then float-round-down* (*Suc p*) ((*power-down-fl p x* (*Suc n div 2*))^2)

else float-round-down (*Suc p*) (*x* ∗ *power-down-fl p x n*))
unfolding *atomize-conj*
by *transfer simp*

lemma *power-down-pos*: *0* < *x* ⟹ *0* < *power-down p x n*
by (*induct p x n rule*: *power-down.induct*)
(*auto simp del*: *odd-Suc-div-two intro*!: *truncate-down-pos*)

lemma *power-down-nonneg*: *0* ≤ *x* ⟹ *0* ≤ *power-down p x n*
by (*induct p x n rule*: *power-down.induct*)
(*auto simp del*: *odd-Suc-div-two intro*!: *truncate-down-nonneg mult-nonneg-nonneg*)

lemma *power-down*: *0* ≤ *x* ⟹ *power-down p x n* ≤ *x* ˆ *n*
proof (*induct p x n rule*: *power-down.induct*)
case (*2 p x n*)
{
assume *odd n*
then have (*power-down p x* (*Suc n div 2*)) ˆ *2* ≤ (*x* ˆ (*Suc n div 2*)) ˆ *2*
using *2*
by (*auto intro*: *power-mono power-down-nonneg simp del*: *odd-Suc-div-two*)
also have . . . = *x* ˆ (*Suc n div 2* ∗ *2*)
by (*simp add*: *power-mult*[*symmetric*])
also have *Suc n div 2* ∗ *2* = *Suc n*
using ‹*odd n*› **by** *presburger*
finally have *?case*
using ‹*odd n*›
by (*auto intro*!: *truncate-down-le simp del*: *odd-Suc-div-two*)
}
then show *?case*
by (*auto intro*!: *truncate-down-le mult-left-mono 2 mult-nonneg-nonneg power-down-nonneg*)
qed *simp*

lemma *power-up*: *0* ≤ *x* ⟹ *x* ˆ *n* ≤ *power-up p x n*
proof (*induct p x n rule*: *power-up.induct*)
case (*2 p x n*)
{
assume *odd n*
then have *Suc n* = *Suc n div 2* ∗ *2*
using ‹*odd n*› *even-Suc* **by** *presburger*
then have *x* ˆ *Suc n* ≤ (*x* ˆ (*Suc n div 2*))^2
by (*simp add*: *power-mult*[*symmetric*])
also have . . . ≤ (*power-up p x* (*Suc n div 2*))^2
using *2* ‹*odd n*›
by (*auto intro*: *power-mono simp del*: *odd-Suc-div-two*)
finally have *?case*
using ‹*odd n*›
by (*auto intro*!: *truncate-up-le simp del*: *odd-Suc-div-two*)
}
then show *?case*

by (*auto intro*!: *truncate-up-le mult-left-mono 2*)
qed *simp*

lemmas *power-up-le* = *order-trans*[*OF - power-up*]
and *power-up-less* = *less-le-trans*[*OF - power-up*]
and *power-down-le* = *order-trans*[*OF power-down*]

lemma *power-down-fl*: $0 \leq x \Longrightarrow$ *power-down-fl p x n* $\leq x$ ^ n
by *transfer* (*rule power-down*)

lemma *power-up-fl*: $0 \leq x \Longrightarrow x$ ^ $n \leq$ *power-up-fl p x n*
by *transfer* (*rule power-up*)

lemma *real-power-up-fl*: *real-of-float* (*power-up-fl p x n*) = *power-up p x n*
by *transfer simp*

lemma *real-power-down-fl*: *real-of-float* (*power-down-fl p x n*) = *power-down p x n*
by *transfer simp*

39.15 Approximate Addition

definition *plus-down prec x y* = *truncate-down prec* (*x* + *y*)

definition *plus-up prec x y* = *truncate-up prec* (*x* + *y*)

lemma *float-plus-down-float*[*intro*, *simp*]: $x \in$ *float* $\Longrightarrow y \in$ *float* $\Longrightarrow$ *plus-down p x y* $\in$ *float*
by (*simp add*: *plus-down-def*)

lemma *float-plus-up-float*[*intro*, *simp*]: $x \in$ *float* $\Longrightarrow y \in$ *float* $\Longrightarrow$ *plus-up p x y* $\in$ *float*
by (*simp add*: *plus-up-def*)

lift-definition *float-plus-down*::*nat* ⇒ *float* ⇒ *float* ⇒ *float* **is** *plus-down* **..**

lift-definition *float-plus-up*::*nat* ⇒ *float* ⇒ *float* ⇒ *float* **is** *plus-up* **..**

lemma *plus-down*: *plus-down prec x y* $\leq x + y$
and *plus-up*: $x + y \leq$ *plus-up prec x y*
by (*auto simp*: *plus-down-def truncate-down plus-up-def truncate-up*)

lemma *float-plus-down*: *real-of-float* (*float-plus-down prec x y*) $\leq x + y$
and *float-plus-up*: $x + y \leq$ *real-of-float* (*float-plus-up prec x y*)
by (*transfer*, *rule plus-down plus-up*)+

lemmas *plus-down-le* = *order-trans*[*OF plus-down*]
and *plus-up-le* = *order-trans*[*OF - plus-up*]
and *float-plus-down-le* = *order-trans*[*OF float-plus-down*]

and *float-plus-up-le* = *order-trans*[*OF* - *float-plus-up*]

lemma *compute-plus-up*[*code*]: *plus-up p x y* = − *plus-down p* (−*x*) (−*y*)
using *truncate-down-uminus-eq*[*of p x* + *y*]
by (*auto simp*: *plus-down-def plus-up-def*)

lemma *truncate-down-log2-eqI*:
assumes ⌊*log 2* |*x*|⌋ = ⌊*log 2* |*y*|⌋
assumes ⌊*x* ∗ *2 powr* (*p* − ⌊*log 2* |*x*|⌋)⌋ = ⌊*y* ∗ *2 powr* (*p* − ⌊*log 2* |*x*|⌋)⌋
shows *truncate-down p x* = *truncate-down p y*
using *assms* **by** (*auto simp*: *truncate-down-def round-down-def*)

lemma *bitlen-eq-zero-iff*: *bitlen x* = *0* ⟷ *x* ≤ *0*
by (*clarsimp simp add*: *bitlen-def*)
(*metis Float.compute-bitlen add.commute bitlen-def bitlen-nonneg less-add-same-cancel2 not-less*
zero-less-one)

lemma *sum-neq-zeroI*:
fixes *a k* :: *real*
shows |*a*| ≥ *k* ⟹ |*b*| < *k* ⟹ *a* + *b* ≠ *0*
and |*a*| > *k* ⟹ |*b*| ≤ *k* ⟹ *a* + *b* ≠ *0*
by *auto*

lemma *abs-real-le-2-powr-bitlen*[*simp*]: |*real-of-int m2*| < *2 powr real-of-int* (*bitlen* |*m2*|)
proof (*cases m2* = *0*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
then have |*m2*| < *2* ^ *nat* (*bitlen* |*m2*|)
using *bitlen-bounds*[*of* |*m2*|]
by (*auto simp*: *powr-add bitlen-nonneg*)
then show *?thesis*
by (*metis bitlen-nonneg powr-int of-int-abs real-of-int-less-numeral-power-cancel-iff zero-less-numeral*)
qed

lemma *floor-sum-times-2-powr-sgn-eq*:
fixes *ai p q* :: *int*
and *a b* :: *real*
assumes *a* ∗ *2 powr p* = *ai*
and *b-le-1*: |*b* ∗ *2 powr* (*p* + *1*)| ≤ *1*
and *leqp*: *q* ≤ *p*
shows ⌊(*a* + *b*) ∗ *2 powr q*⌋ = ⌊(*2* ∗ *ai* + *sgn b*) ∗ *2 powr* (*q* − *p* − *1*)⌋
proof −
consider *b* = *0* | *b* > *0* | *b* < *0* **by** *arith*
then show *?thesis*

proof *cases*
case *1*
then show *?thesis*
by (*simp add*: *assms*(*1*)[*symmetric*] *powr-add*[*symmetric*] *algebra-simps powr-mult-base*)
next
case *2*
then have $b * 2\ powr\ p < |b * 2\ powr\ (p + 1)|$
by *simp*
also note *b-le-1*
finally have *b-less-1*: $b * 2\ powr\ real\text{-}of\text{-}int\ p < 1$ **.**

from *b-less-1* ‹$b > 0$› **have** *floor-eq*: $\lfloor b * 2\ powr\ real\text{-}of\text{-}int\ p\rfloor = 0$ $\lfloor sgn\ b\ /\ 2\rfloor = 0$
by (*simp-all add*: *floor-eq-iff*)

have $\lfloor(a + b) * 2\ powr\ q\rfloor = \lfloor(a + b) * 2\ powr\ p * 2\ powr\ (q - p)\rfloor$
by (*simp add*: *algebra-simps powr-realpow*[*symmetric*] *powr-add*[*symmetric*])
also have ... $= \lfloor(ai + b * 2\ powr\ p) * 2\ powr\ (q - p)\rfloor$
by (*simp add*: *assms algebra-simps*)
also have ... $= \lfloor(ai + b * 2\ powr\ p)\ /\ real\text{-}of\text{-}int\ ((2::int)\ \hat{}\ nat\ (p - q))\rfloor$
using *assms*
by (*simp add*: *algebra-simps powr-realpow*[*symmetric*] *divide-powr-uminus powr-add*[*symmetric*])
also have ... $= \lfloor ai\ /\ real\text{-}of\text{-}int\ ((2::int)\ \hat{}\ nat\ (p - q))\rfloor$
by (*simp del*: *of-int-power add*: *floor-divide-real-eq-div floor-eq*)
finally have $\lfloor(a + b) * 2\ powr\ real\text{-}of\text{-}int\ q\rfloor = \lfloor real\text{-}of\text{-}int\ ai\ /\ real\text{-}of\text{-}int\ ((2::int)\ \hat{}\ nat\ (p - q))\rfloor$ **.**
moreover
{
have $\lfloor(2 * ai + sgn\ b) * 2\ powr\ (real\text{-}of\text{-}int\ (q - p) - 1)\rfloor = \lfloor(ai + sgn\ b\ /\ 2) * 2\ powr\ (q - p)\rfloor$
by (*subst powr-divide2*[*symmetric*]) (*simp add*: *field-simps*)
also have ... $= \lfloor(ai + sgn\ b\ /\ 2)\ /\ real\text{-}of\text{-}int\ ((2::int)\ \hat{}\ nat\ (p - q))\rfloor$
using *leqp* **by** (*simp add*: *powr-realpow*[*symmetric*] *powr-divide2*[*symmetric*])
also have ... $= \lfloor ai\ /\ real\text{-}of\text{-}int\ ((2::int)\ \hat{}\ nat\ (p - q))\rfloor$
by (*simp del*: *of-int-power add*: *floor-divide-real-eq-div floor-eq*)
finally
have $\lfloor(2 * ai + (sgn\ b)) * 2\ powr\ (real\text{-}of\text{-}int\ (q - p) - 1)\rfloor =$
$\lfloor real\text{-}of\text{-}int\ ai\ /\ real\text{-}of\text{-}int\ ((2::int)\ \hat{}\ nat\ (p - q))\rfloor$ **.**
}
ultimately show *?thesis* **by** *simp*
next
case *3*
then have *floor-eq*: $\lfloor b * 2\ powr\ (real\text{-}of\text{-}int\ p + 1)\rfloor = -1$
using *b-le-1*
by (*auto simp*: *floor-eq-iff algebra-simps pos-divide-le-eq*[*symmetric*] *abs-if divide-powr-uminus*
intro!: *mult-neg-pos split*: *if-split-asm*)
have $\lfloor(a + b) * 2\ powr\ q\rfloor = \lfloor(2{*}a + 2{*}b) * 2\ powr\ p * 2\ powr\ (q - p - 1)\rfloor$

by (*simp add*: *algebra-simps powr-realpow*[*symmetric*] *powr-add*[*symmetric*] *powr-mult-base*)
also have ... = ⌊(*2* ∗ (*a* ∗ *2 powr p*) + *2* ∗ *b* ∗ *2 powr p*) ∗ *2 powr* (*q* − *p* − *1*)⌋
by (*simp add*: *algebra-simps*)
also have ... = ⌊(*2* ∗ *ai* + *b* ∗ *2 powr* (*p* + *1*)) / *2 powr* (*1* − *q* + *p*)⌋
using *assms* **by** (*simp add*: *algebra-simps powr-mult-base divide-powr-uminus*)
also have ... = ⌊(*2* ∗ *ai* + *b* ∗ *2 powr* (*p* + *1*)) / *real-of-int* ((*2*::*int*) ˆ *nat* (*p* − *q* + *1*))⌋
using *assms* **by** (*simp add*: *algebra-simps powr-realpow*[*symmetric*])
also have ... = ⌊(*2* ∗ *ai* − *1*) / *real-of-int* ((*2*::*int*) ˆ *nat* (*p* − *q* + *1*))⌋
using ‹*b* < *0*› *assms*
by (*simp add*: *floor-divide-of-int-eq floor-eq floor-divide-real-eq-div*
del: *of-int-mult of-int-power of-int-diff*)
also have ... = ⌊(*2* ∗ *ai* − *1*) ∗ *2 powr* (*q* − *p* − *1*)⌋
using *assms* **by** (*simp add*: *algebra-simps divide-powr-uminus powr-realpow*[*symmetric*])
finally show *?thesis*
using ‹*b* < *0*› **by** *simp*
qed
qed

lemma *log2-abs-int-add-less-half-sgn-eq*:
fixes *ai* :: *int*
and *b* :: *real*
assumes |*b*| ≤ *1*/*2*
and *ai* ≠ *0*
shows ⌊*log 2* |*real-of-int ai* + *b*|⌋ = ⌊*log 2* |*ai* + *sgn b* / *2*|⌋
proof (*cases b* = *0*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
def *k* ≡ ⌊*log 2* |*ai*|⌋
then have ⌊*log 2* |*ai*|⌋ = *k*
by *simp*
then have *k*: *2 powr k* ≤ |*ai*| |*ai*| < *2 powr* (*k* + *1*)
by (*simp-all add*: *floor-log-eq-powr-iff* ‹*ai* ≠ *0*›)
have *k* ≥ *0*
using *assms* **by** (*auto simp*: *k-def*)
def *r* ≡ |*ai*| − *2* ˆ *nat k*
have *r*: *0* ≤ *r r* < *2 powr k*
using ‹*k* ≥ *0*› *k*
by (*auto simp*: *r-def k-def algebra-simps powr-add abs-if powr-int*)
then have *r* ≤ (*2*::*int*) ˆ *nat k* − *1*
using ‹*k* ≥ *0*› **by** (*auto simp*: *powr-int*)
from *this*[*simplified of-int-le-iff*[*symmetric*]] ‹*0* ≤ *k*›
have *r-le*: *r* ≤ *2 powr k* − *1*
apply (*auto simp*: *algebra-simps powr-int*)
by (*metis of-int-1 of-int-add real-of-int-le-numeral-power-cancel-iff*)

have |*ai*| = *2 powr k* + *r*
  using ‹*k* ≥ *0*› **by** (*auto simp*: *k-def r-def powr-realpow*[*symmetric*])

have *pos*: |*b*| < *1* ⟹ *0* < *2 powr k* + (*r* + *b*) **for** *b* :: *real*
  using ‹*0* ≤ *k*› ‹*ai* ≠ *0*›
  by (*auto simp add*: *r-def powr-realpow*[*symmetric*] *abs-if sgn-if algebra-simps*
    split: *if-split-asm*)
have *less*: |*sgn ai* ∗ *b*| < *1*
  and *less'*: |*sgn* (*sgn ai* ∗ *b*) / *2*| < *1*
  using ‹|*b*| ≤ -› **by** (*auto simp*: *abs-if sgn-if split*: *if-split-asm*)

have *floor-eq*: ⋀*b*::*real*. |*b*| ≤ *1* / *2* ⟹
    ⌊*log 2* (*1* + (*r* + *b*) / *2 powr k*)⌋ = (*if r* = *0* ∧ *b* < *0 then* −*1 else 0*)
  using ‹*k* ≥ *0*› *r r-le*
  by (*auto simp*: *floor-log-eq-powr-iff powr-minus-divide field-simps sgn-if*)

from ‹*real-of-int* |*ai*| = -› **have** |*ai* + *b*| = *2 powr k* + (*r* + *sgn ai* ∗ *b*)
  using ‹|*b*| <= -› ‹*0* ≤ *k*› *r*
  by (*auto simp add*: *sgn-if abs-if*)
also have ⌊*log 2* ...⌋ = ⌊*log 2* (*2 powr k* + *r* + *sgn* (*sgn ai* ∗ *b*) / *2*)⌋
proof −
  have *2 powr k* + (*r* + (*sgn ai*) ∗ *b*) = *2 powr k* ∗ (*1* + (*r* + *sgn ai* ∗ *b*) / *2 powr k*)
    by (*simp add*: *field-simps*)
  also have ⌊*log 2* ...⌋ = *k* + ⌊*log 2* (*1* + (*r* + *sgn ai* ∗ *b*) / *2 powr k*)⌋
    using *pos*[*OF less*]
    by (*subst log-mult*) (*simp-all add*: *log-mult powr-mult field-simps*)
  also
  let *?if* = *if r* = *0* ∧ *sgn ai* ∗ *b* < *0 then* −*1 else 0*
  have ⌊*log 2* (*1* + (*r* + *sgn ai* ∗ *b*) / *2 powr k*)⌋ = *?if*
    using ‹|*b*| <= -›
    by (*intro floor-eq*) (*auto simp*: *abs-mult sgn-if*)
  also
  have ... = ⌊*log 2* (*1* + (*r* + *sgn* (*sgn ai* ∗ *b*) / *2*) / *2 powr k*)⌋
    by (*subst floor-eq*) (*auto simp*: *sgn-if*)
  also have *k* + ... = ⌊*log 2* (*2 powr k* ∗ (*1* + (*r* + *sgn* (*sgn ai* ∗ *b*) / *2*) / *2 powr k*))⌋
    unfolding *floor-add2*[*symmetric*]
    using *pos*[*OF less'*] ‹|*b*| ≤ -›
    by (*simp add*: *field-simps add-log-eq-powr*)
  also have *2 powr k* ∗ (*1* + (*r* + *sgn* (*sgn ai* ∗ *b*) / *2*) / *2 powr k*) =
      2 powr k + *r* + *sgn* (*sgn ai* ∗ *b*) / *2*
    by (*simp add*: *sgn-if field-simps*)
  finally show *?thesis* **.**
qed
also have *2 powr k* + *r* + *sgn* (*sgn ai* ∗ *b*) / *2* = |*ai* + *sgn b* / *2*|
  unfolding ‹*real-of-int* |*ai*| = -›[*symmetric*] **using** ‹*ai* ≠ *0*›
  by (*auto simp*: *abs-if sgn-if algebra-simps*)

finally show *?thesis* **.**
qed

context
begin

qualified lemma *compute-far-float-plus-down*:
fixes *m1 e1 m2 e2* :: *int*
and *p* :: *nat*
defines *k1* ≡ *Suc p* − *nat* (*bitlen* |*m1*|)
assumes *H*: *bitlen* |*m2*| ≤ *e1* − *e2* − *k1* − *2 m1* ≠ *0 m2* ≠ *0 e1* ≥ *e2*
shows *float-plus-down p* (*Float m1 e1*) (*Float m2 e2*) =
float-round-down p (*Float* (*m1* ∗ *2* ˆ (*Suc* (*Suc k1*)) + *sgn m2*) (*e1* − *int k1* − *2*))
proof −
let *?a* = *real-of-float* (*Float m1 e1*)
let *?b* = *real-of-float* (*Float m2 e2*)
let *?sum* = *?a* + *?b*
let *?shift* = *real-of-int e2* − *real-of-int e1* + *real k1* + *1*
let *?m1* = *m1* ∗ *2* ˆ *Suc k1*
let *?m2* = *m2* ∗ *2 powr ?shift*
let *?m2'* = *sgn m2* / *2*
let *?e* = *e1* − *int k1* − *1*

have *sum-eq*: *?sum* = (*?m1* + *?m2*) ∗ *2 powr ?e*
by (*auto simp*: *powr-add*[*symmetric*] *powr-mult*[*symmetric*] *algebra-simps powr-realpow*[*symmetric*] *powr-mult-base*)

have |*?m2*| ∗ *2* < *2 powr* (*bitlen* |*m2*| + *?shift* + *1*)
by (*auto simp*: *field-simps powr-add powr-mult-base powr-numeral powr-divide2*[*symmetric*] *abs-mult*)
also have ... ≤ *2 powr 0*
using *H* **by** (*intro powr-mono*) *auto*
finally have *abs-m2-less-half*: |*?m2*| < *1* / *2*
by *simp*

then have |*real-of-int m2*| < *2 powr* −(*?shift* + *1*)
unfolding *powr-minus-divide* **by** (*auto simp*: *bitlen-def field-simps powr-mult-base abs-mult*)
also have ... ≤ *2 powr real-of-int* (*e1* − *e2* − *2*)
by *simp*
finally have *b-less-quarter*: |*?b*| < *1/4* ∗ *2 powr real-of-int e1*
by (*simp add*: *powr-add field-simps powr-divide2*[*symmetric*] *powr-numeral abs-mult*)
also have *1/4* < |*real-of-int m1*| / *2* **using** ‹*m1* ≠ *0*› **by** *simp*
finally have *b-less-half-a*: |*?b*| < *1/2* ∗ |*?a*|
by (*simp add*: *algebra-simps powr-mult-base abs-mult*)
then have *a-half-less-sum*: |*?a*| / *2* < |*?sum*|
by (*auto simp*: *field-simps abs-if split*: *if-split-asm*)

from *b-less-half-a* **have** $|?b| < |?a|$ $|?b| \leq |?a|$
by *simp-all*

have $|real\text{-}of\text{-}float\ (Float\ m1\ e1)| \geq 1/4 * 2\ powr\ real\text{-}of\text{-}int\ e1$
using ‹$m1 \neq 0$›
by (*auto simp*: *powr-add powr-int bitlen-nonneg divide-right-mono abs-mult*)
then have $?sum \neq 0$ **using** *b-less-quarter*
by (*rule sum-neq-zeroI*)
then have $?m1 + ?m2 \neq 0$
unfolding *sum-eq* **by** (*simp add*: *abs-mult zero-less-mult-iff*)

have $|real\text{-}of\text{-}int\ ?m1| \geq 2\ \hat{}\ Suc\ k1$ $|?m2'| < 2\ \hat{}\ Suc\ k1$
using ‹$m1 \neq 0$› ‹$m2 \neq 0$› **by** (*auto simp*: *sgn-if less-1-mult abs-mult simp del*: *power.simps*)
then have *sum'-nz*: $?m1 + ?m2' \neq 0$
by (*intro sum-neq-zeroI*)

have $\lfloor log\ 2\ |real\text{-}of\text{-}float\ (Float\ m1\ e1) + real\text{-}of\text{-}float\ (Float\ m2\ e2)|\rfloor = \lfloor log\ 2\ |?m1 + ?m2|\rfloor + ?e$
using ‹$?m1 + ?m2 \neq 0$›
unfolding *floor-add*[*symmetric*] *sum-eq*
by (*simp add*: *abs-mult log-mult*) *linarith*
also have $\lfloor log\ 2\ |?m1 + ?m2|\rfloor = \lfloor log\ 2\ |?m1 + sgn\ (real\text{-}of\text{-}int\ m2 * 2\ powr\ ?shift) / 2|\rfloor$
using *abs-m2-less-half* ‹$m1 \neq 0$›
by (*intro log2-abs-int-add-less-half-sgn-eq*) (*auto simp*: *abs-mult*)
also have $sgn\ (real\text{-}of\text{-}int\ m2 * 2\ powr\ ?shift) = sgn\ m2$
by (*auto simp*: *sgn-if zero-less-mult-iff less-not-sym*)
also
have $|?m1 + ?m2'| * 2\ powr\ ?e = |?m1 * 2 + sgn\ m2| * 2\ powr\ (?e - 1)$
by (*auto simp*: *field-simps powr-minus*[*symmetric*] *powr-divide2*[*symmetric*] *powr-mult-base*)
then have $\lfloor log\ 2\ |?m1 + ?m2'|\rfloor + ?e = \lfloor log\ 2\ |real\text{-}of\text{-}float\ (Float\ (?m1 * 2 + sgn\ m2)\ (?e - 1))|\rfloor$
using ‹$?m1 + ?m2' \neq 0$›
unfolding *floor-add-of-int*[*symmetric*]
by (*simp add*: *log-add-eq-powr abs-mult-pos*)
finally
have $\lfloor log\ 2\ |?sum|\rfloor = \lfloor log\ 2\ |real\text{-}of\text{-}float\ (Float\ (?m1*2 + sgn\ m2)\ (?e - 1))|\rfloor$ **.**
then have $plus\text{-}down\ p\ (Float\ m1\ e1)\ (Float\ m2\ e2) =$
$truncate\text{-}down\ p\ (Float\ (?m1*2 + sgn\ m2)\ (?e - 1))$
unfolding *plus-down-def*
proof (*rule truncate-down-log2-eqI*)
let $?f = (int\ p - \lfloor log\ 2\ |real\text{-}of\text{-}float\ (Float\ m1\ e1) + real\text{-}of\text{-}float\ (Float\ m2\ e2)|\rfloor)$
let $?ai = m1 * 2\ \hat{}\ (Suc\ k1)$
have $\lfloor(?a + ?b) * 2\ powr\ real\text{-}of\text{-}int\ ?f\rfloor = \lfloor(real\text{-}of\text{-}int\ (2 * ?ai) + sgn\ ?b) *$

2 powr real-of-int (?f − − ?e − 1)⌋
  proof (*rule floor-sum-times-2-powr-sgn-eq*)
   show *?a ∗ 2 powr real-of-int (−?e) = real-of-int ?ai*
    by (*simp add*: *powr-add powr-realpow*[*symmetric*] *powr-divide2*[*symmetric*])
   show |*?b ∗ 2 powr real-of-int (−?e + 1)*| ≤ *1*
    using *abs-m2-less-half*
    by (*simp add*: *abs-mult powr-add*[*symmetric*] *algebra-simps powr-mult-base*)
  next
   have *e1* + ⌊*log 2* |*real-of-int m1*|⌋ − *1* = ⌊*log 2* |*?a*|⌋ − *1*
    using ‹*m1* ≠ *0*›
     by (*simp add*: *floor-add2*[*symmetric*] *algebra-simps log-mult abs-mult del*:
floor-add2)
   also have ... ≤ ⌊*log 2* |*?a* + *?b*|⌋
    using *a-half-less-sum* ‹*m1* ≠ *0*› ‹*?sum* ≠ *0*›
    unfolding *floor-diff-of-int*[*symmetric*]
    by (*auto simp add*: *log-minus-eq-powr powr-minus-divide intro*!: *floor-mono*)
   finally
   have *int p* − ⌊*log 2* |*?a* + *?b*|⌋ ≤ *p* − (*bitlen* |*m1*|) − *e1* + *2*
    by (*auto simp*: *algebra-simps bitlen-def* ‹*m1* ≠ *0*›)
   also have ... ≤ − *?e*
    using *bitlen-nonneg*[*of* |*m1*|] **by** (*simp add*: *k1-def*)
   finally show *?f* ≤ − *?e* **by** *simp*
  qed
  also have *sgn ?b* = *sgn m2*
   using *powr-gt-zero*[*of 2 e2*]
   by (*auto simp add*: *sgn-if zero-less-mult-iff simp del*: *powr-gt-zero*)
  also have ⌊(*real-of-int* (*2* ∗ *?m1*) + *real-of-int* (*sgn m2*)) ∗ *2 powr real-of-int*
(*?f* − − *?e* − *1*)⌋ =
    ⌊*Float* (*?m1* ∗ *2* + *sgn m2*) (*?e* − *1*) ∗ *2 powr ?f*⌋
   by (*simp add*: *powr-add*[*symmetric*] *algebra-simps powr-realpow*[*symmetric*])
  finally
  show ⌊(*?a* + *?b*) ∗ *2 powr ?f*⌋ = ⌊*real-of-float* (*Float* (*?m1* ∗ *2* + *sgn m2*) (*?e*
− *1*)) ∗ *2 powr ?f*⌋ **.**
 qed
 then show *?thesis*
  by *transfer* (*simp add*: *plus-down-def ac-simps Let-def*)
qed

lemma *compute-float-plus-down-naive*[*code*]: *float-plus-down p x y* = *float-round-down p* (*x* + *y*)
 by *transfer* (*auto simp*: *plus-down-def*)

qualified lemma *compute-float-plus-down*[*code*]:
 fixes *p*::*nat* **and** *m1 e1 m2 e2*::*int*
 shows *float-plus-down p* (*Float m1 e1*) (*Float m2 e2*) =
  (*if m1* = *0 then float-round-down p* (*Float m2 e2*)
  else if m2 = *0 then float-round-down p* (*Float m1 e1*)
  else (*if e1* ≥ *e2 then*
   (*let*

k1 = Suc p − nat (bitlen |m1|)
in
if bitlen |m2| > e1 − e2 − k1 − 2 then float-round-down p ((Float m1 e1) + (Float m2 e2))
else float-round-down p (Float (m1 ∗ 2 ˆ (Suc (Suc k1)) + sgn m2) (e1 − int k1 − 2)))
else float-plus-down p (Float m2 e2) (Float m1 e1)))
proof −
{
assume *bitlen |m2| ≤ e1 − e2 − (Suc p − nat (bitlen |m1|)) − 2 m1 ≠ 0 m2 ≠ 0 e1 ≥ e2*
note *compute-far-float-plus-down*[*OF this*]
}
then show *?thesis*
by *transfer* (*simp add*: *Let-def plus-down-def ac-simps*)
qed

qualified lemma *compute-float-plus-up*[*code*]: *float-plus-up p x y* = − *float-plus-down p* (−*x*) (−*y*)
using *truncate-down-uminus-eq*[*of p x + y*]
by *transfer* (*simp add*: *plus-down-def plus-up-def ac-simps*)

lemma *mantissa-zero*[*simp*]: *mantissa 0 = 0*
by (*metis mantissa-0 zero-float.abs-eq*)

qualified lemma *compute-float-less*[*code*]: *a < b* ⟷ *is-float-pos* (*float-plus-down 0 b* (− *a*))
using *truncate-down*[*of 0 b − a*] *truncate-down-pos*[*of b − a 0*]
by *transfer* (*auto simp*: *plus-down-def*)

qualified lemma *compute-float-le*[*code*]: *a ≤ b* ⟷ *is-float-nonneg* (*float-plus-down 0 b* (− *a*))
using *truncate-down*[*of 0 b − a*] *truncate-down-nonneg*[*of b − a 0*]
by *transfer* (*auto simp*: *plus-down-def*)

end

39.16 Lemmas needed by Approximate

lemma *Float-num*[*simp*]:
real-of-float (*Float 1 0*) = *1*
real-of-float (*Float 1 1*) = *2*
real-of-float (*Float 1 2*) = *4*
real-of-float (*Float 1* (− *1*)) = *1/2*
real-of-float (*Float 1* (− *2*)) = *1/4*
real-of-float (*Float 1* (− *3*)) = *1/8*
real-of-float (*Float* (− *1*) *0*) = −*1*
real-of-float (*Float* (*numeral n*) *0*) = *numeral n*
real-of-float (*Float* (− *numeral n*) *0*) = − *numeral n*

using *two-powr-int-float*[*of* 2] *two-powr-int-float*[*of* −1] *two-powr-int-float*[*of* −2]
two-powr-int-float[*of* −3]
using *powr-realpow*[*of* 2 2] *powr-realpow*[*of* 2 3]
using *powr-minus*[*of* 2 1] *powr-minus*[*of* 2 2] *powr-minus*[*of* 2 3]
by *auto*

lemma *real-of-Float-int*[*simp*]: *real-of-float* (*Float n 0*) = *real n*
by *simp*

lemma *float-zero*[*simp*]: *real-of-float* (*Float 0 e*) = *0*
by *simp*

lemma *abs-div-2-less*: $a \neq 0 \Longrightarrow a \neq -1 \Longrightarrow |(a::int)\ div\ 2| < |a|$
by *arith*

lemma *lapprox-rat*: *real-of-float* (*lapprox-rat prec x y*) ≤ *real-of-int x* / *real-of-int y*
by (*simp add*: *lapprox-rat.rep-eq truncate-down*)

lemma *mult-div-le*:
fixes *a b* :: *int*
assumes $b > 0$
shows $a \geq b * (a\ div\ b)$
proof −
from *zmod-zdiv-equality'*[*of a b*] **have** $a = b * (a\ div\ b) + a\ mod\ b$
by *simp*
also have $\ldots \geq b * (a\ div\ b) + 0$
apply (*rule add-left-mono*)
apply (*rule pos-mod-sign*)
using *assms* **apply** *simp*
done
finally show *?thesis*
by *simp*
qed

lemma *lapprox-rat-nonneg*:
fixes *n x y*
assumes $0 \leq x$ **and** $0 \leq y$
shows 0 ≤ *real-of-float* (*lapprox-rat n x y*)
using *assms*
by *transfer* (*simp add*: *truncate-down-nonneg*)

lemma *rapprox-rat*: *real-of-int x* / *real-of-int y* ≤ *real-of-float* (*rapprox-rat prec x y*)
by *transfer* (*simp add*: *truncate-up*)

lemma *rapprox-rat-le1*:
fixes *n x y*

assumes *xy*: $0 \le x$ $0 < y$ $x \le y$
shows *real-of-float* (*rapprox-rat n x y*) $\le$ *1*
using *assms*
by *transfer* (*simp add*: *truncate-up-le1*)

lemma *rapprox-rat-nonneg-nonpos*: $0 \le x \Longrightarrow y \le 0 \Longrightarrow$ *real-of-float* (*rapprox-rat n x y*) ≤ 0
by *transfer* (*simp add*: *truncate-up-nonpos divide-nonneg-nonpos*)

lemma *rapprox-rat-nonpos-nonneg*: $x \le 0 \Longrightarrow 0 \le y \Longrightarrow$ *real-of-float* (*rapprox-rat n x y*) ≤ 0
by *transfer* (*simp add*: *truncate-up-nonpos divide-nonpos-nonneg*)

lemma *real-divl*: *real-divl prec x y* $\le x / y$
by (*simp add*: *real-divl-def truncate-down*)

lemma *real-divr*: $x / y \le$ *real-divr prec x y*
by (*simp add*: *real-divr-def truncate-up*)

lemma *float-divl*: *real-of-float* (*float-divl prec x y*) $\le x / y$
by *transfer* (*rule real-divl*)

lemma *real-divl-lower-bound*:
$0 \le x \Longrightarrow 0 \le y \Longrightarrow 0 \le$ *real-divl prec x y*
by (*simp add*: *real-divl-def truncate-down-nonneg*)

lemma *float-divl-lower-bound*:
$0 \le x \Longrightarrow 0 \le y \Longrightarrow 0 \le$ *real-of-float* (*float-divl prec x y*)
by *transfer* (*rule real-divl-lower-bound*)

lemma *exponent-1*: *exponent 1 = 0*
using *exponent-float*[*of 1 0*] **by** (*simp add*: *one-float-def*)

lemma *mantissa-1*: *mantissa 1 = 1*
using *mantissa-float*[*of 1 0*] **by** (*simp add*: *one-float-def*)

lemma *bitlen-1*: *bitlen 1 = 1*
by (*simp add*: *bitlen-def*)

lemma *mantissa-eq-zero-iff*: *mantissa x = 0* $\longleftrightarrow$ *x = 0*
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
show *?rhs* **if** *?lhs*
proof −
from *that* **have** *z*: *0 = real-of-float x*
using *mantissa-exponent* **by** *simp*
show *?thesis*
by (*simp add*: *zero-float-def z*)
qed

show *?lhs* **if** *?rhs*
using *that* **by** (*simp add*: *zero-float-def*)
qed

lemma *float-upper-bound*: $x \leq 2$ *powr* (*bitlen* |*mantissa x*| + *exponent x*)
proof (*cases* $x = 0$)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
then have *mantissa* $x \neq 0$
using *mantissa-eq-zero-iff* **by** *auto*
have $x =$ *mantissa* $x * 2$ *powr* (*exponent x*)
by (*rule mantissa-exponent*)
also have *mantissa* $x \leq$ |*mantissa x*|
by *simp*
also have ... ≤ 2 *powr* (*bitlen* |*mantissa x*|)
using *bitlen-bounds*[*of* |*mantissa x*|] *bitlen-nonneg* ‹*mantissa* $x \neq 0$›
by (*auto simp del*: *of-int-abs simp add*: *powr-int*)
finally show *?thesis* **by** (*simp add*: *powr-add*)
qed

lemma *real-divl-pos-less1-bound*:
assumes $0 < x$ $x \leq 1$
shows $1 \leq$ *real-divl prec 1 x*
using *assms*
by (*auto intro*!: *truncate-down-ge1 simp*: *real-divl-def*)

lemma *float-divl-pos-less1-bound*:
$0 <$ *real-of-float* $x \Longrightarrow$ *real-of-float* $x \leq 1 \Longrightarrow$ *prec* $\geq 1 \Longrightarrow 1 \leq$ *real-of-float* (*float-divl prec 1 x*)
by *transfer* (*rule real-divl-pos-less1-bound*)

lemma *float-divr*: *real-of-float x* / *real-of-float y* $\leq$ *real-of-float* (*float-divr prec x y*)
by *transfer* (*rule real-divr*)

lemma *real-divr-pos-less1-lower-bound*:
assumes $0 < x$
and $x \leq 1$
shows $1 \leq$ *real-divr prec 1 x*
proof −
have $1 \leq 1 / x$
using ‹$0 < x$› **and** ‹$x <= 1$› **by** *auto*
also have ... $\leq$ *real-divr prec 1 x*
using *real-divr*[**where** x=*1* **and** y=x] **by** *auto*
finally show *?thesis* **by** *auto*
qed

lemma *float-divr-pos-less1-lower-bound*: $0 < x \Longrightarrow x \leq 1 \Longrightarrow 1 \leq$ *float-divr prec 1 x*
by *transfer* (*rule real-divr-pos-less1-lower-bound*)

lemma *real-divr-nonpos-pos-upper-bound*:
$x \leq 0 \Longrightarrow 0 \leq y \Longrightarrow$ *real-divr prec x y* ≤ 0
by (*simp add*: *real-divr-def truncate-up-nonpos divide-le-0-iff*)

lemma *float-divr-nonpos-pos-upper-bound*:
real-of-float x $\leq 0 \Longrightarrow 0 \leq$ *real-of-float y* $\Longrightarrow$ *real-of-float* (*float-divr prec x y*) ≤ 0
by *transfer* (*rule real-divr-nonpos-pos-upper-bound*)

lemma *real-divr-nonneg-neg-upper-bound*:
$0 \leq x \Longrightarrow y \leq 0 \Longrightarrow$ *real-divr prec x y* ≤ 0
by (*simp add*: *real-divr-def truncate-up-nonpos divide-le-0-iff*)

lemma *float-divr-nonneg-neg-upper-bound*:
$0 \leq$ *real-of-float x* $\Longrightarrow$ *real-of-float y* $\leq 0 \Longrightarrow$ *real-of-float* (*float-divr prec x y*) ≤ 0
by *transfer* (*rule real-divr-nonneg-neg-upper-bound*)

lemma *truncate-up-nonneg-mono*:
assumes $0 \leq x$ $x \leq y$
shows *truncate-up prec x* $\leq$ *truncate-up prec y*
proof −
consider $\lfloor log\ 2\ x \rfloor = \lfloor log\ 2\ y \rfloor$ | $\lfloor log\ 2\ x \rfloor \neq \lfloor log\ 2\ y \rfloor$ $0 < x$ | $x \leq 0$
by *arith*
then show *?thesis*
proof *cases*
case *1*
then show *?thesis*
using *assms*
by (*auto simp*: *truncate-up-def round-up-def intro*!: *ceiling-mono*)
next
case *2*
from *assms* ‹$0 < x$› **have** $log\ 2\ x \leq log\ 2\ y$
by *auto*
with ‹$\lfloor log\ 2\ x \rfloor \neq \lfloor log\ 2\ y \rfloor$›
have *logless*: $log\ 2\ x < log\ 2\ y$ **and** *flogless*: $\lfloor log\ 2\ x \rfloor < \lfloor log\ 2\ y \rfloor$
by (*metis floor-less-cancel linorder-cases not-le*)+
have *truncate-up prec x* =
real-of-int $\lceil x * 2$ *powr real-of-int* (*int prec* − $\lfloor log\ 2\ x \rfloor$)$\rceil * 2$ *powr* − *real-of-int* (*int prec* − $\lfloor log\ 2\ x \rfloor$)
using *assms* **by** (*simp add*: *truncate-up-def round-up-def*)
also have $\lceil x * 2$ *powr real-of-int* (*int prec* − $\lfloor log\ 2\ x \rfloor$)$\rceil \leq$ (2 ˆ (*Suc prec*))
proof (*unfold ceiling-le-iff*)
have $x * 2$ *powr real-of-int* (*int prec* − $\lfloor log\ 2\ x \rfloor$) $\leq x *$ (2 *powr real* (*Suc prec*) / (2 *powr log* 2 *x*))

using *real-of-int-floor-add-one-ge*[*of log 2 x*] *assms*
by (*auto simp add*: *algebra-simps powr-divide2 intro*!: *mult-left-mono*)
then show $x * 2$ *powr real-of-int* (*int prec* − ⌊*log 2 x*⌋) ≤ *real-of-int* ((*2*::*int*) ˆ (*Suc prec*))
using ‹*0* < *x*› **by** (*simp add*: *powr-realpow powr-add*)
qed
then have *real-of-int* ⌈*x* ∗ *2 powr real-of-int* (*int prec* − ⌊*log 2 x*⌋)⌉ ≤ *2 powr int* (*Suc prec*)
apply (*auto simp*: *powr-realpow powr-add*)
by (*metis power-Suc real-of-int-le-numeral-power-cancel-iff*)
also
have *2 powr* − *real-of-int* (*int prec* − ⌊*log 2 x*⌋) ≤ *2 powr* − *real-of-int* (*int prec* − ⌊*log 2 y*⌋ + *1*)
using *logless flogless* **by** (*auto intro*!: *floor-mono*)
also have *2 powr real-of-int* (*int* (*Suc prec*)) ≤ *2 powr* (*log 2 y* + *real-of-int* (*int prec* − ⌊*log 2 y*⌋ + *1*))
using *assms* ‹*0* < *x*›
by (*auto simp*: *algebra-simps*)
finally have *truncate-up prec x* ≤ *2 powr* (*log 2 y* + *real-of-int* (*int prec* − ⌊*log 2 y*⌋ + *1*)) ∗ *2 powr* − *real-of-int* (*int prec* − ⌊*log 2 y*⌋ + *1*)
by *simp*
also have ... = *2 powr* (*log 2 y* + *real-of-int* (*int prec* − ⌊*log 2 y*⌋) − *real-of-int* (*int prec* − ⌊*log 2 y*⌋))
by (*subst powr-add*[*symmetric*]) *simp*
also have ... = *y*
using ‹*0* < *x*› *assms*
by (*simp add*: *powr-add*)
also have ... ≤ *truncate-up prec y*
by (*rule truncate-up*)
finally show *?thesis* **.**
next
case *3*
then show *?thesis*
using *assms*
by (*auto intro*!: *truncate-up-le*)
qed
qed

lemma *truncate-up-switch-sign-mono*:
assumes *x* ≤ *0 0* ≤ *y*
shows *truncate-up prec x* ≤ *truncate-up prec y*
proof −
note *truncate-up-nonpos*[*OF* ‹*x* ≤ *0*›]
also note *truncate-up-le*[*OF* ‹*0* ≤ *y*›]
finally show *?thesis* **.**
qed

lemma *truncate-down-switch-sign-mono*:
assumes *x* ≤ *0*

and $0 \le y$
and $x \le y$
shows *truncate-down prec* $x \le$ *truncate-down prec* y
proof −
note *truncate-down-le*[*OF* ‹$x \le 0$›]
also note *truncate-down-nonneg*[*OF* ‹$0 \le y$›]
finally show *?thesis* **.**
qed

lemma *truncate-down-nonneg-mono*:
assumes $0 \le x$ $x \le y$
shows *truncate-down prec* $x \le$ *truncate-down prec* y
proof −
consider $x \le 0$ | $\lfloor log\ 2\ |x|\rfloor = \lfloor log\ 2\ |y|\rfloor$ |
$0 < x$ $\lfloor log\ 2\ |x|\rfloor \neq \lfloor log\ 2\ |y|\rfloor$
by *arith*
then show *?thesis*
proof *cases*
case *1*
with *assms* **have** $x = 0$ $0 \le y$ **by** *simp-all*
then show *?thesis*
by (*auto intro*!: *truncate-down-nonneg*)
next
case *2*
then show *?thesis*
using *assms*
by (*auto simp*: *truncate-down-def round-down-def intro*!: *floor-mono*)
next
case *3*
from ‹$0 < x$› **have** *log 2 x* $\le$ *log 2 y* $0 < y$ $0 \le y$
using *assms* **by** *auto*
with ‹$\lfloor log\ 2\ |x|\rfloor \neq \lfloor log\ 2\ |y|\rfloor$›
have *logless*: *log 2 x* < *log 2 y* **and** *flogless*: $\lfloor log\ 2\ x\rfloor < \lfloor log\ 2\ y\rfloor$
unfolding *atomize-conj abs-of-pos*[*OF* ‹$0 < x$›] *abs-of-pos*[*OF* ‹$0 < y$›]
by (*metis floor-less-cancel linorder-cases not-le*)
have *2 powr prec* $\le y * 2$ *powr real prec* / (*2 powr log 2 y*)
using ‹$0 < y$› **by** *simp*
also have ... $\le y * 2$ *powr real* (*Suc prec*) / (*2 powr* (*real-of-int* $\lfloor log\ 2\ y\rfloor$ + *1*))
using ‹$0 \le y$› ‹$0 \le x$› *assms*(*2*)
by (*auto intro*!: *powr-mono divide-left-mono*
simp: *of-nat-diff powr-add*
powr-divide2[*symmetric*])
also have ... $= y * 2$ *powr real* (*Suc prec*) / (*2 powr real-of-int* $\lfloor log\ 2\ y\rfloor * 2$)
by (*auto simp*: *powr-add*)
finally have (*2* ^ *prec*) $\le \lfloor y * 2$ *powr real-of-int* (*int* (*Suc prec*) − $\lfloor log\ 2\ |y|\rfloor$ − *1*)$\rfloor$
using ‹$0 \le y$›
by (*auto simp*: *powr-divide2*[*symmetric*] *le-floor-iff powr-realpow powr-add*)

then have (*2* ˆ (*prec*)) ∗ *2 powr* − *real-of-int* (*int prec* − ⌊*log 2* |*y*|⌋) ≤ *truncate-down prec y*
by (*auto simp*: *truncate-down-def round-down-def*)
moreover
{
have *x* = *2 powr* (*log 2* |*x*|) **using** ‹*0* < *x*› **by** *simp*
also have . . . ≤ (*2* ˆ (*Suc prec*)) ∗ *2 powr* − *real-of-int* (*int prec* − ⌊*log 2* |*x*|⌋)
using *real-of-int-floor-add-one-ge*[*of log 2* |*x*|] ‹*0* < *x*›
by (*auto simp*: *powr-realpow*[*symmetric*] *powr-add*[*symmetric*] *algebra-simps* *powr-mult-base le-powr-iff*)
also
have *2 powr* − *real-of-int* (*int prec* − ⌊*log 2* |*x*|⌋) ≤ *2 powr* − *real-of-int* (*int prec* − ⌊*log 2* |*y*|⌋ + *1*)
using *logless flogless* ‹*x* > *0*› ‹*y* > *0*›
by (*auto intro*!: *floor-mono*)
finally have *x* ≤ (*2* ˆ *prec*) ∗ *2 powr* − *real-of-int* (*int prec* − ⌊*log 2* |*y*|⌋)
by (*auto simp*: *powr-realpow*[*symmetric*] *powr-divide2*[*symmetric*] *assms* *of-nat-diff*)
}
ultimately show *?thesis*
by (*metis dual-order.trans truncate-down*)
qed
qed

lemma *truncate-down-eq-truncate-up*: *truncate-down p x* = − *truncate-up p* (−*x*)
and *truncate-up-eq-truncate-down*: *truncate-up p x* = − *truncate-down p* (−*x*)
by (*auto simp*: *truncate-up-uminus-eq truncate-down-uminus-eq*)

lemma *truncate-down-mono*: *x* ≤ *y* =⇒ *truncate-down p x* ≤ *truncate-down p y*
apply (*cases 0* ≤ *x*)
apply (*rule truncate-down-nonneg-mono*, *assumption*+)
apply (*simp add*: *truncate-down-eq-truncate-up*)
apply (*cases 0* ≤ *y*)
apply (*auto intro*: *truncate-up-nonneg-mono truncate-up-switch-sign-mono*)
done

lemma *truncate-up-mono*: *x* ≤ *y* =⇒ *truncate-up p x* ≤ *truncate-up p y*
by (*simp add*: *truncate-up-eq-truncate-down truncate-down-mono*)

lemma *Float-le-zero-iff* : *Float a b* ≤ *0* ←→ *a* ≤ *0*
by (*auto simp*: *zero-float-def mult-le-0-iff*) (*simp add*: *not-less* [*symmetric*])

lemma *real-of-float-pprt*[*simp*]:
fixes *a* :: *float*
shows *real-of-float* (*pprt a*) = *pprt* (*real-of-float a*)
unfolding *pprt-def sup-float-def max-def sup-real-def* **by** *auto*

lemma *real-of-float-nprt*[*simp*]:

fixes *a* :: *float*
shows *real-of-float* (*nprt a*) = *nprt* (*real-of-float a*)
unfolding *nprt-def inf-float-def min-def inf-real-def* **by** *auto*

context
begin

lift-definition *int-floor-fl* :: *float* ⇒ *int* **is** *floor* **.**

qualified lemma *compute-int-floor-fl*[*code*]:
int-floor-fl (*Float m e*) = (*if 0* ≤ *e then m* ∗ *2* ˆ *nat e else m div* (*2* ˆ (*nat* (−*e*))))
apply *transfer*
apply (*simp add*: *powr-int floor-divide-of-int-eq*)
apply (*metis* (*no-types*, *hide-lams*)*floor-divide-of-int-eq of-int-numeral of-int-power floor-of-int of-int-mult*)
done

lift-definition *floor-fl* :: *float* ⇒ *float* **is** λ*x*. *real-of-int* ⌊*x*⌋
by *simp*

qualified lemma *compute-floor-fl*[*code*]:
floor-fl (*Float m e*) = (*if 0* ≤ *e then Float m e else Float* (*m div* (*2* ˆ (*nat* (−*e*)))) *0*)
apply *transfer*
apply (*simp add*: *powr-int floor-divide-of-int-eq*)
apply (*metis* (*no-types*, *hide-lams*)*floor-divide-of-int-eq of-int-numeral of-int-power of-int-mult*)
done

end

lemma *floor-fl*: *real-of-float* (*floor-fl x*) ≤ *real-of-float x*
by *transfer simp*

lemma *int-floor-fl*: *real-of-int* (*int-floor-fl x*) ≤ *real-of-float x*
by *transfer simp*

lemma *floor-pos-exp*: *exponent* (*floor-fl x*) ≥ *0*
proof (*cases floor-fl x* = *float-of 0*)
case *True*
then show *?thesis*
by (*simp add*: *floor-fl-def*)
next
case *False*
have *eq*: *floor-fl x* = *Float* ⌊*real-of-float x*⌋ *0*
by *transfer simp*
obtain *i* **where** ⌊*real-of-float x*⌋ = *mantissa* (*floor-fl x*) ∗ *2* ˆ *i 0* = *exponent* (*floor-fl x*) − *int i*

by (*rule denormalize-shift*[*OF eq*[*THEN eq-reflection*] *False*])
then show *?thesis*
by *simp*
qed

lemma *compute-mantissa*[*code*]:
mantissa (*Float m e*) =
(*if m* = *0 then 0 else if 2 dvd m then mantissa* (*normfloat* (*Float m e*)) *else m*)
by (*auto simp*: *mantissa-float Float.abs-eq*)

lemma *compute-exponent*[*code*]:
exponent (*Float m e*) =
(*if m* = *0 then 0 else if 2 dvd m then exponent* (*normfloat* (*Float m e*)) *else e*)
by (*auto simp*: *exponent-float Float.abs-eq*)

end

40 Less common functions on lists

theory *More-List*
imports *Main*
begin

definition *strip-while* :: ($'a \Rightarrow$ *bool*) $\Rightarrow$ $'a$ *list* $\Rightarrow$ $'a$ *list*
where
strip-while P = *rev* ∘ *dropWhile P* ∘ *rev*

lemma *strip-while-rev* [*simp*]:
strip-while P (*rev xs*) = *rev* (*dropWhile P xs*)
by (*simp add*: *strip-while-def*)

lemma *strip-while-Nil* [*simp*]:
strip-while P [] = []
by (*simp add*: *strip-while-def*)

lemma *strip-while-append* [*simp*]:
¬ *P x* ⟹ *strip-while P* (*xs* @ [*x*]) = *xs* @ [*x*]
by (*simp add*: *strip-while-def*)

lemma *strip-while-append-rec* [*simp*]:
P x ⟹ *strip-while P* (*xs* @ [*x*]) = *strip-while P xs*
by (*simp add*: *strip-while-def*)

lemma *strip-while-Cons* [*simp*]:
¬ *P x* ⟹ *strip-while P* (*x* # *xs*) = *x* # *strip-while P xs*
by (*induct xs rule*: *rev-induct*) (*simp-all add*: *strip-while-def*)

lemma *strip-while-eq-Nil* [*simp*]:
strip-while P xs = [] ⟷ (∀ *x*∈*set xs*. *P x*)

by (*simp add*: *strip-while-def*)

lemma *strip-while-eq-Cons-rec*:
strip-while P (*x* # *xs*) = *x* # *strip-while P xs* ⟷ ¬ (*P x* ∧ (∀ *x*∈*set xs*. *P x*))
by (*induct xs rule*: *rev-induct*) (*simp-all add*: *strip-while-def*)

lemma *strip-while-not-last* [*simp*]:
¬ *P* (*last xs*) ⟹ *strip-while P xs* = *xs*
by (*cases xs rule*: *rev-cases*) *simp-all*

lemma *split-strip-while-append*:
fixes *xs* :: *'a list*
obtains *ys zs* :: *'a list*
where *strip-while P xs* = *ys* **and** ∀ *x*∈*set zs*. *P x* **and** *xs* = *ys* @ *zs*
proof (*rule that*)
show *strip-while P xs* = *strip-while P xs* **..**
show ∀ *x*∈*set* (*rev* (*takeWhile P* (*rev xs*))). *P x* **by** (*simp add*: *takeWhile-eq-all-conv* [*symmetric*])
have *rev xs* = *rev* (*strip-while P xs* @ *rev* (*takeWhile P* (*rev xs*)))
by (*simp add*: *strip-while-def*)
then show *xs* = *strip-while P xs* @ *rev* (*takeWhile P* (*rev xs*))
by (*simp only*: *rev-is-rev-conv*)
qed

lemma *strip-while-snoc* [*simp*]:
strip-while P (*xs* @ [*x*]) = (*if P x then strip-while P xs else xs* @ [*x*])
by (*simp add*: *strip-while-def*)

lemma *strip-while-map*:
strip-while P (*map f xs*) = *map f* (*strip-while* (*P* ∘ *f*) *xs*)
by (*simp add*: *strip-while-def rev-map dropWhile-map*)

definition *no-leading* :: (*'a* ⇒ *bool*) ⇒ *'a list* ⇒ *bool*
where
no-leading P xs ⟷ (*xs* ≠ [] ⟶ ¬ *P* (*hd xs*))

lemma *no-leading-Nil* [*simp*, *intro*!]:
no-leading P []
by (*simp add*: *no-leading-def*)

lemma *no-leading-Cons* [*simp*, *intro*!]:
no-leading P (*x* # *xs*) ⟷ ¬ *P x*
by (*simp add*: *no-leading-def*)

lemma *no-leading-append* [*simp*]:
no-leading P (*xs* @ *ys*) ⟷ *no-leading P xs* ∧ (*xs* = [] ⟶ *no-leading P ys*)
by (*induct xs*) *simp-all*

```
lemma no-leading-dropWhile [simp]:
  no-leading P (dropWhile P xs)
  by (induct xs) simp-all

lemma dropWhile-eq-obtain-leading:
  assumes dropWhile P xs = ys
  obtains zs where xs = zs @ ys and ⋀z. z ∈ set zs ⟹ P z and no-leading P
ys
proof –
  from assms have ∃ zs. xs = zs @ ys ∧ (∀ z ∈ set zs. P z) ∧ no-leading P ys
  proof (induct xs arbitrary: ys)
    case Nil then show ?case by simp
  next
    case (Cons x xs ys)
    show ?case proof (cases P x)
      case True with Cons.hyps [of ys] Cons.prems
      have ∃ zs. xs = zs @ ys ∧ (∀ a∈set zs. P a) ∧ no-leading P ys
        by simp
      then obtain zs where xs = zs @ ys and ⋀z. z ∈ set zs ⟹ P z
        and *: no-leading P ys
        by blast
      with True have x # xs = (x # zs) @ ys and ⋀z. z ∈ set (x # zs) ⟹ P z
        by auto
      with * show ?thesis
        by blast   next
      case False
      with Cons show ?thesis by (cases ys) simp-all
    qed
  qed
  with that show thesis
    by blast
qed

lemma dropWhile-idem-iff:
  dropWhile P xs = xs ⟷ no-leading P xs
  by (cases xs) (auto elim: dropWhile-eq-obtain-leading)


abbreviation no-trailing :: ('a ⇒ bool) ⇒ 'a list ⇒ bool
where
  no-trailing P xs ≡ no-leading P (rev xs)

lemma no-trailing-unfold:
  no-trailing P xs ⟷ (xs ≠ [] ⟶ ¬ P (last xs))
  by (induct xs) simp-all

lemma no-trailing-Nil [simp, intro!]:
  no-trailing P []
  by simp
```

lemma *no-trailing-Cons* [*simp*]:
no-trailing P (*x* # *xs*) ⟷ *no-trailing P xs* ∧ (*xs* = [] ⟶ ¬ *P x*)
by *simp*

lemma *no-trailing-append-Cons* [*simp*]:
no-trailing P (*xs* @ *y* # *ys*) ⟷ *no-trailing P* (*y* # *ys*)
by *simp*

lemma *no-trailing-strip-while* [*simp*]:
no-trailing P (*strip-while P xs*)
by (*induct xs rule*: *rev-induct*) *simp-all*

lemma *strip-while-eq-obtain-trailing*:
assumes *strip-while P xs* = *ys*
obtains *zs* **where** *xs* = *ys* @ *zs* **and** ⋀*z*. *z* ∈ *set zs* ⟹ *P z* **and** *no-trailing P ys*
proof −
from *assms* **have** *rev* (*rev* (*dropWhile P* (*rev xs*))) = *rev ys*
by (*simp add*: *strip-while-def*)
then have *dropWhile P* (*rev xs*) = *rev ys*
by *simp*
then obtain *zs* **where** *A*: *rev xs* = *zs* @ *rev ys* **and** *B*: ⋀*z*. *z* ∈ *set zs* ⟹ *P z*
and *C*: *no-trailing P ys*
using *dropWhile-eq-obtain-leading* **by** *blast*
from *A* **have** *rev* (*rev xs*) = *rev* (*zs* @ *rev ys*)
by *simp*
then have *xs* = *ys* @ *rev zs*
by *simp*
moreover from *B* **have** ⋀*z*. *z* ∈ *set* (*rev zs*) ⟹ *P z*
by *simp*
ultimately show *thesis* **using** *that C* **by** *blast*
qed

lemma *strip-while-idem-iff*:
strip-while P xs = *xs* ⟷ *no-trailing P xs*
proof −
def *ys* ≡ *rev xs*
moreover have *strip-while P* (*rev ys*) = *rev ys* ⟷ *no-trailing P* (*rev ys*)
by (*simp add*: *dropWhile-idem-iff*)
ultimately show *?thesis* **by** *simp*
qed

lemma *no-trailing-map*:
no-trailing P (*map f xs*) = *no-trailing* (*P* ∘ *f*) *xs*
by (*simp add*: *last-map no-trailing-unfold*)

lemma *no-trailing-upt* [*simp*]:
no-trailing P [*n*..<*m*] ⟷ (*n* < *m* ⟶ ¬ *P* (*m* − *1*))

by (*auto simp add*: *no-trailing-unfold*)

definition *nth-default* :: *'a* ⇒ *'a list* ⇒ *nat* ⇒ *'a*
where
nth-default dflt xs n = (*if n* < *length xs then xs* ! *n else dflt*)

lemma *nth-default-nth*:
n < *length xs* ⟹ *nth-default dflt xs n* = *xs* ! *n*
by (*simp add*: *nth-default-def*)

lemma *nth-default-beyond*:
length xs ≤ *n* ⟹ *nth-default dflt xs n* = *dflt*
by (*simp add*: *nth-default-def*)

lemma *nth-default-Nil* [*simp*]:
nth-default dflt [] *n* = *dflt*
by (*simp add*: *nth-default-def*)

lemma *nth-default-Cons*:
nth-default dflt (*x* # *xs*) *n* = (*case n of 0* ⇒ *x* | *Suc n'* ⇒ *nth-default dflt xs n'*)
by (*simp add*: *nth-default-def split*: *nat.split*)

lemma *nth-default-Cons-0* [*simp*]:
nth-default dflt (*x* # *xs*) *0* = *x*
by (*simp add*: *nth-default-Cons*)

lemma *nth-default-Cons-Suc* [*simp*]:
nth-default dflt (*x* # *xs*) (*Suc n*) = *nth-default dflt xs n*
by (*simp add*: *nth-default-Cons*)

lemma *nth-default-replicate-dflt* [*simp*]:
nth-default dflt (*replicate n dflt*) *m* = *dflt*
by (*simp add*: *nth-default-def*)

lemma *nth-default-append*:
nth-default dflt (*xs* @ *ys*) *n* =
(*if n* < *length xs then nth xs n else nth-default dflt ys* (*n* − *length xs*))
by (*auto simp add*: *nth-default-def nth-append*)

lemma *nth-default-append-trailing* [*simp*]:
nth-default dflt (*xs* @ *replicate n dflt*) = *nth-default dflt xs*
by (*simp add*: *fun-eq-iff nth-default-append*) (*simp add*: *nth-default-def*)

lemma *nth-default-snoc-default* [*simp*]:
nth-default dflt (*xs* @ [*dflt*]) = *nth-default dflt xs*
by (*auto simp add*: *nth-default-def fun-eq-iff nth-append*)

lemma *nth-default-eq-dflt-iff*:

nth-default dflt xs k = *dflt* ⟷ (*k* < *length xs* ⟶ *xs* ! *k* = *dflt*)
by (*simp add*: *nth-default-def*)

lemma *in-enumerate-iff-nth-default-eq*:
x ≠ *dflt* ⟹ (*n*, *x*) ∈ *set* (*enumerate 0 xs*) ⟷ *nth-default dflt xs n* = *x*
by (*auto simp add*: *nth-default-def in-set-conv-nth enumerate-eq-zip*)

lemma *last-conv-nth-default*:
assumes *xs* ≠ []
shows *last xs* = *nth-default dflt xs* (*length xs* − *1*)
using *assms* **by** (*simp add*: *nth-default-def last-conv-nth*)

lemma *nth-default-map-eq*:
f dflt' = *dflt* ⟹ *nth-default dflt* (*map f xs*) *n* = *f* (*nth-default dflt' xs n*)
by (*simp add*: *nth-default-def*)

lemma *finite-nth-default-neq-default* [*simp*]:
finite {*k*. *nth-default dflt xs k* ≠ *dflt*}
by (*simp add*: *nth-default-def*)

lemma *sorted-list-of-set-nth-default*:
sorted-list-of-set {*k*. *nth-default dflt xs k* ≠ *dflt*} = *map fst* (*filter* (λ(-, *x*). *x* ≠ *dflt*) (*enumerate 0 xs*))
by (*rule sorted-distinct-set-unique*) (*auto simp add*: *nth-default-def in-set-conv-nth sorted-filter distinct-map-filter enumerate-eq-zip intro*: *rev-image-eqI*)

lemma *map-nth-default*:
map (*nth-default x xs*) [*0*..<*length xs*] = *xs*
proof −
have ∗: *map* (*nth-default x xs*) [*0*..<*length xs*] = *map* (*List.nth xs*) [*0*..<*length xs*]
by (*rule map-cong*) (*simp-all add*: *nth-default-nth*)
show *?thesis* **by** (*simp add*: ∗ *map-nth*)
qed

lemma *range-nth-default* [*simp*]:
range (*nth-default dflt xs*) = *insert dflt* (*set xs*)
by (*auto simp add*: *nth-default-def* [*abs-def*] *in-set-conv-nth*)

lemma *nth-strip-while*:
assumes *n* < *length* (*strip-while P xs*)
shows *strip-while P xs* ! *n* = *xs* ! *n*
proof −
have *length* (*dropWhile P* (*rev xs*)) + *length* (*takeWhile P* (*rev xs*)) = *length xs*
by (*subst add.commute*)
(*simp add*: *arg-cong* [**where** *f*=*length*, *OF takeWhile-dropWhile-id*, *unfolded length-append*])
then show *?thesis* **using** *assms*
by (*simp add*: *strip-while-def rev-nth dropWhile-nth*)

qed

lemma *length-strip-while-le*:
length (*strip-while P xs*) ≤ *length xs*
unfolding *strip-while-def o-def length-rev*
by (*subst* (*2*) *length-rev*[*symmetric*])
(*simp add*: *strip-while-def length-dropWhile-le del*: *length-rev*)

lemma *nth-default-strip-while-dflt* [*simp*]:
nth-default dflt (*strip-while* (*op* = *dflt*) *xs*) = *nth-default dflt xs*
by (*induct xs rule*: *rev-induct*) *auto*

lemma *nth-default-eq-iff*:
nth-default dflt xs = *nth-default dflt ys*
⟷ *strip-while* (*HOL.eq dflt*) *xs* = *strip-while* (*HOL.eq dflt*) *ys* (**is** *?P* ⟷ *?Q*)
proof
let *?xs* = *strip-while* (*HOL.eq dflt*) *xs* **and** *?ys* = *strip-while* (*HOL.eq dflt*) *ys*
assume *?P*
then have *eq*: *nth-default dflt ?xs* = *nth-default dflt ?ys*
by *simp*
have *len*: *length ?xs* = *length ?ys*
proof (*rule ccontr*)
assume *len*: *length ?xs* ≠ *length ?ys*
{ **fix** *xs ys* :: *'a list*
let *?xs* = *strip-while* (*HOL.eq dflt*) *xs* **and** *?ys* = *strip-while* (*HOL.eq dflt*) *ys*
assume *eq*: *nth-default dflt ?xs* = *nth-default dflt ?ys*
assume *len*: *length ?xs* < *length ?ys*
then have *length ?ys* > *0* **by** *arith*
then have *?ys* ≠ [] **by** *simp*
with *last-conv-nth-default* [*of ?ys dflt*]
have *last ?ys* = *nth-default dflt ?ys* (*length ?ys* − *1*)
by *auto*
moreover from ‹*?ys* ≠ []› *no-trailing-strip-while* [*of HOL.eq dflt ys*]
have *last ?ys* ≠ *dflt* **by** (*simp add*: *no-trailing-unfold*)
ultimately have *nth-default dflt ?xs* (*length ?ys* − *1*) ≠ *dflt*
using *eq* **by** *simp*
moreover from *len* **have** *length ?ys* − *1* ≥ *length ?xs* **by** *simp*
ultimately have *False* **by** (*simp only*: *nth-default-beyond*) *simp*
}
from *this* [*of xs ys*] *this* [*of ys xs*] *len eq* **show** *False*
by (*auto simp only*: *linorder-class.neq-iff*)
qed
then show *?Q*
proof (*rule nth-equalityI* [*rule-format*])
fix *n*
assume *n* < *length ?xs*
moreover with *len* **have** *n* < *length ?ys*
by *simp*

ultimately have *xs*: *nth-default dflt ?xs n* = *?xs* ! *n*
and *ys*: *nth-default dflt ?ys n* = *?ys* ! *n*
by (*simp-all only*: *nth-default-nth*)
with *eq* **show** *?xs* ! *n* = *?ys* ! *n*
by *simp*
qed
next
assume *?Q*
then have *nth-default dflt* (*strip-while* (*HOL.eq dflt*) *xs*) = *nth-default dflt* (*strip-while* (*HOL.eq dflt*) *ys*)
by *simp*
then show *?P*
by *simp*
qed

end

41 Polynomials as type over a ring structure

theory *Polynomial*
imports *Main* ~~/*src*/*HOL*/*Deriv* ~~/*src*/*HOL*/*Library*/*More-List*
~~/*src*/*HOL*/*Library*/*Infinite-Set*
begin

41.1 Auxiliary: operations for lists (later) representing coefficients

definition *cCons* :: *'a::zero* ⇒ *'a list* ⇒ *'a list* (**infixr** *## 65*)
where
x ## xs = (*if xs* = [] ∧ *x* = *0 then* [] *else x # xs*)

lemma *cCons-0-Nil-eq* [*simp*]:
0 ## [] = []
by (*simp add*: *cCons-def*)

lemma *cCons-Cons-eq* [*simp*]:
x ## y # ys = *x # y # ys*
by (*simp add*: *cCons-def*)

lemma *cCons-append-Cons-eq* [*simp*]:
x ## xs @ y # ys = *x # xs @ y # ys*
by (*simp add*: *cCons-def*)

lemma *cCons-not-0-eq* [*simp*]:
x ≠ *0* ⟹ *x ## xs* = *x # xs*
by (*simp add*: *cCons-def*)

lemma *strip-while-not-0-Cons-eq* [*simp*]:
strip-while (λ*x*. *x* = *0*) (*x # xs*) = *x ## strip-while* (λ*x*. *x* = *0*) *xs*

proof (*cases x* = *0*)
  case *False* **then show** *?thesis* **by** *simp*
next
  case *True* **show** *?thesis*
  proof (*induct xs rule*: *rev-induct*)
    case *Nil* **with** *True* **show** *?case* **by** *simp*
  next
    case (*snoc y ys*) **then show** *?case*
      by (*cases y* = *0*) (*simp-all add*: *append-Cons* [*symmetric*] *del*: *append-Cons*)
  qed
qed

lemma *tl-cCons* [*simp*]:
  tl (*x* ## *xs*) = *xs*
  by (*simp add*: *cCons-def*)

41.2 Definition of type *poly*

typedef (**overloaded**) $'a$ *poly* = $\{f :: nat \Rightarrow 'a::zero.\ \forall_\infty\ n.\ f\ n = 0\}$
  morphisms *coeff Abs-poly* **by** (*auto intro*!: *ALL-MOST*)

setup-lifting *type-definition-poly*

lemma *poly-eq-iff*: $p = q \longleftrightarrow (\forall n.\ coeff\ p\ n = coeff\ q\ n)$
  by (*simp add*: *coeff-inject* [*symmetric*] *fun-eq-iff*)

lemma *poly-eqI*: $(\bigwedge n.\ coeff\ p\ n = coeff\ q\ n) \Longrightarrow p = q$
  by (*simp add*: *poly-eq-iff*)

lemma *MOST-coeff-eq-0*: $\forall_\infty\ n.\ coeff\ p\ n = 0$
  using *coeff* [*of p*] **by** *simp*

41.3 Degree of a polynomial

definition *degree* :: $'a::zero\ poly \Rightarrow nat$
where
  degree p = (*LEAST n*. $\forall i>n.\ coeff\ p\ i = 0$)

lemma *coeff-eq-0*:
  assumes *degree p* < *n*
  shows *coeff p n* = *0*
proof −
  have $\exists n.\ \forall i>n.\ coeff\ p\ i = 0$
    using *MOST-coeff-eq-0* **by** (*simp add*: *MOST-nat*)
  then have $\forall i>degree\ p.\ coeff\ p\ i = 0$
    unfolding *degree-def* **by** (*rule LeastI-ex*)
  with *assms* **show** *?thesis* **by** *simp*
qed

lemma *le-degree*: $coeff\ p\ n \neq 0 \Longrightarrow n \le degree\ p$

by (*erule contrapos-np*, *rule coeff-eq-0*, *simp*)

lemma *degree-le*: $\forall\, i>n.$ *coeff p i* $= 0 \Longrightarrow$ *degree p* $\leq n$
unfolding *degree-def* **by** (*erule Least-le*)

lemma *less-degree-imp*: $n <$ *degree p* $\Longrightarrow \exists\, i>n.$ *coeff p i* $\neq 0$
unfolding *degree-def* **by** (*drule not-less-Least*, *simp*)

41.4 The zero polynomial

instantiation *poly* :: (*zero*) *zero*
begin

lift-definition *zero-poly* :: *'a poly*
is λ-. *0* **by** (*rule MOST-I*) *simp*

instance ..

end

lemma *coeff-0* [*simp*]:
coeff 0 n = *0*
by *transfer rule*

lemma *degree-0* [*simp*]:
degree 0 = *0*
by (*rule order-antisym* [*OF degree-le le0*]) *simp*

lemma *leading-coeff-neq-0*:
assumes $p \neq 0$
shows *coeff p* (*degree p*) $\neq 0$
proof (*cases degree p*)
case *0*
from ‹$p \neq 0$› **have** $\exists\, n.$ *coeff p n* $\neq 0$
by (*simp add*: *poly-eq-iff*)
then obtain *n* **where** *coeff p n* $\neq 0$ **..**
hence $n \leq$ *degree p* **by** (*rule le-degree*)
with ‹*coeff p n* $\neq 0$› **and** ‹*degree p* = *0*›
show *coeff p* (*degree p*) $\neq 0$ **by** *simp*
next
case (*Suc n*)
from ‹*degree p* = *Suc n*› **have** $n <$ *degree p* **by** *simp*
hence $\exists\, i>n.$ *coeff p i* $\neq 0$ **by** (*rule less-degree-imp*)
then obtain *i* **where** $n < i$ **and** *coeff p i* $\neq 0$ **by** *fast*
from ‹*degree p* = *Suc n*› **and** ‹$n < i$› **have** *degree p* $\leq i$ **by** *simp*
also from ‹*coeff p i* $\neq 0$› **have** $i \leq$ *degree p* **by** (*rule le-degree*)
finally have *degree p* = *i* **.**
with ‹*coeff p i* $\neq 0$› **show** *coeff p* (*degree p*) $\neq 0$ **by** *simp*
qed

lemma *leading-coeff-0-iff* [*simp*]:
coeff p (*degree p*) = *0* ⟷ *p* = *0*
by (*cases p* = *0*, *simp*, *simp add*: *leading-coeff-neq-0*)

41.5 List-style constructor for polynomials

lift-definition *pCons* :: *'a*::*zero* ⇒ *'a poly* ⇒ *'a poly*
is λ*a p*. *case-nat a* (*coeff p*)
by (*rule MOST-SucD*) (*simp add*: *MOST-coeff-eq-0*)

lemmas *coeff-pCons* = *pCons.rep-eq*

lemma *coeff-pCons-0* [*simp*]:
coeff (*pCons a p*) *0* = *a*
by *transfer simp*

lemma *coeff-pCons-Suc* [*simp*]:
coeff (*pCons a p*) (*Suc n*) = *coeff p n*
by (*simp add*: *coeff-pCons*)

lemma *degree-pCons-le*:
degree (*pCons a p*) ≤ *Suc* (*degree p*)
by (*rule degree-le*) (*simp add*: *coeff-eq-0 coeff-pCons split*: *nat.split*)

lemma *degree-pCons-eq*:
p ≠ *0* ⟹ *degree* (*pCons a p*) = *Suc* (*degree p*)
apply (*rule order-antisym* [*OF degree-pCons-le*])
apply (*rule le-degree*, *simp*)
done

lemma *degree-pCons-0*:
degree (*pCons a 0*) = *0*
apply (*rule order-antisym* [*OF - le0*])
apply (*rule degree-le*, *simp add*: *coeff-pCons split*: *nat.split*)
done

lemma *degree-pCons-eq-if* [*simp*]:
degree (*pCons a p*) = (*if p* = *0 then 0 else Suc* (*degree p*))
apply (*cases p* = *0*, *simp-all*)
apply (*rule order-antisym* [*OF - le0*])
apply (*rule degree-le*, *simp add*: *coeff-pCons split*: *nat.split*)
apply (*rule order-antisym* [*OF degree-pCons-le*])
apply (*rule le-degree*, *simp*)
done

lemma *pCons-0-0* [*simp*]:
pCons 0 0 = *0*
by (*rule poly-eqI*) (*simp add*: *coeff-pCons split*: *nat.split*)

lemma *pCons-eq-iff* [*simp*]:
  pCons a p = *pCons b q* ⟷ *a* = *b* ∧ *p* = *q*
proof *safe*
  assume *pCons a p* = *pCons b q*
  then have *coeff* (*pCons a p*) *0* = *coeff* (*pCons b q*) *0* **by** *simp*
  then show *a* = *b* **by** *simp*
next
  assume *pCons a p* = *pCons b q*
  then have ∀ *n*. *coeff* (*pCons a p*) (*Suc n*) =
                coeff (*pCons b q*) (*Suc n*) **by** *simp*
  then show *p* = *q* **by** (*simp add*: *poly-eq-iff*)
qed

lemma *pCons-eq-0-iff* [*simp*]:
  pCons a p = *0* ⟷ *a* = *0* ∧ *p* = *0*
  using *pCons-eq-iff* [*of a p 0 0*] **by** *simp*

lemma *pCons-cases* [*cases type*: *poly*]:
  obtains (*pCons*) *a q* **where** *p* = *pCons a q*
proof
  show *p* = *pCons* (*coeff p 0*) (*Abs-poly* (λ*n*. *coeff p* (*Suc n*)))
    by *transfer*
      (*simp-all add*: *MOST-inj*[**where** *f*=*Suc* **and** *P*=λ*n*. *p n* = *0* **for** *p*] *fun-eq-iff*
Abs-poly-inverse
                split: *nat.split*)
qed

lemma *pCons-induct* [*case-names 0 pCons*, *induct type*: *poly*]:
  assumes *zero*: *P 0*
  assumes *pCons*: ⋀*a p*. *a* ≠ *0* ∨ *p* ≠ *0* ⟹ *P p* ⟹ *P* (*pCons a p*)
  shows *P p*
proof (*induct p rule*: *measure-induct-rule* [**where** *f*=*degree*])
  case (*less p*)
  obtain *a q* **where** *p* = *pCons a q* **by** (*rule pCons-cases*)
  have *P q*
  proof (*cases q* = *0*)
    case *True*
    then show *P q* **by** (*simp add*: *zero*)
  next
    case *False*
    then have *degree* (*pCons a q*) = *Suc* (*degree q*)
      by (*rule degree-pCons-eq*)
    then have *degree q* < *degree p*
      using ‹*p* = *pCons a q*› **by** *simp*
    then show *P q*
      by (*rule less.hyps*)
  qed
  have *P* (*pCons a q*)

proof (*cases* $a \neq 0 \vee q \neq 0$)
  case *True*
  with ⟨$P\ q$⟩ **show** *?thesis* **by** (*auto intro*: *pCons*)
next
  case *False*
  with *zero* **show** *?thesis* **by** *simp*
qed
then show *?case*
  using ⟨$p = pCons\ a\ q$⟩ **by** *simp*
qed

lemma *degree-eq-zeroE*:
fixes p :: $'a$::*zero poly*
assumes *degree* $p = 0$
obtains a **where** $p = pCons\ a\ 0$
proof −
obtain $a\ q$ **where** p: $p = pCons\ a\ q$ **by** (*cases* p)
with *assms* **have** $q = 0$ **by** (*cases* $q = 0$) *simp-all*
with p **have** $p = pCons\ a\ 0$ **by** *simp*
with *that* **show** *thesis* **.**
qed

41.6 Quickcheck generator for polynomials

quickcheck-generator *poly constructors*: *0* :: *- poly*, *pCons*

41.7 List-style syntax for polynomials

syntax
-poly :: *args* ⇒ $'a$ *poly* ([:(-):])

translations
[:x, xs:] == *CONST pCons* x [:xs:]
[:x:] == *CONST pCons* x *0*
[:x:] <= *CONST pCons* x (*-constrain 0* t)

41.8 Representation of polynomials by lists of coefficients

primrec *Poly* :: $'a$::*zero list* ⇒ $'a$ *poly*
where
[*code-post*]: *Poly* [] = *0*
| [*code-post*]: *Poly* (a # *as*) = *pCons* a (*Poly as*)

lemma *Poly-replicate-0* [*simp*]:
Poly (*replicate* n *0*) = *0*
by (*induct* n) *simp-all*

lemma *Poly-eq-0*:
Poly as = *0* ⟷ (∃ n. *as* = *replicate* n *0*)
by (*induct as*) (*auto simp add*: *Cons-replicate-eq*)

lemma *degree-Poly*: *degree* (*Poly xs*) ≤ *length xs*
  by (*induction xs*) *simp-all*

definition *coeffs* :: *'a poly* ⇒ *'a::zero list*
where
  coeffs p = (*if p* = *0 then* [] *else map* (λ*i*. *coeff p i*) [*0* ..< *Suc* (*degree p*)])

lemma *coeffs-eq-Nil* [*simp*]:
  coeffs p = [] ⟷ *p* = *0*
  by (*simp add*: *coeffs-def*)

lemma *not-0-coeffs-not-Nil*:
  p ≠ *0* ⟹ *coeffs p* ≠ []
  by *simp*

lemma *coeffs-0-eq-Nil* [*simp*]:
  coeffs 0 = []
  by *simp*

lemma *coeffs-pCons-eq-cCons* [*simp*]:
  coeffs (*pCons a p*) = *a* ## *coeffs p*
proof −
  { **fix** *ms* :: *nat list* **and** *f* :: *nat* ⇒ *'a* **and** *x* :: *'a*
    assume ∀ *m*∈*set ms*. *m* > *0*
    then have *map* (*case-nat x f*) *ms* = *map f* (*map* (λ*n*. *n* − *1*) *ms*)
      by (*induct ms*) (*auto split*: *nat.split*)
  }
  note ∗ = *this*
  show *?thesis*
   by (*simp add*: *coeffs-def* ∗ *upt-conv-Cons coeff-pCons map-decr-upt del*: *upt-Suc*)
qed

lemma *length-coeffs*: *p* ≠ *0* ⟹ *length* (*coeffs p*) = *degree p* + *1*
  by (*simp add*: *coeffs-def*)

lemma *coeffs-nth*:
  assumes *p* ≠ *0 n* ≤ *degree p*
  shows   *coeffs p* ! *n* = *coeff p n*
  using *assms* **unfolding** *coeffs-def* **by** (*auto simp del*: *upt-Suc*)

lemma *not-0-cCons-eq* [*simp*]:
  p ≠ *0* ⟹ *a* ## *coeffs p* = *a* # *coeffs p*
  by (*simp add*: *cCons-def*)

lemma *Poly-coeffs* [*simp*, *code abstype*]:
  Poly (*coeffs p*) = *p*
  by (*induct p*) *auto*

lemma *coeffs-Poly* [*simp*]:
coeffs (*Poly as*) = *strip-while* (*HOL.eq 0*) *as*
proof (*induct as*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons a as*)
have (∀ *n*. *as* ≠ *replicate n 0*) ⟷ (∃ *a*∈*set as*. *a* ≠ *0*)
using *replicate-length-same* [*of as 0*] **by** (*auto dest*: *sym* [*of - as*])
with *Cons* **show** *?case* **by** *auto*
qed

lemma *last-coeffs-not-0*:
p ≠ *0* ⟹ *last* (*coeffs p*) ≠ *0*
by (*induct p*) (*auto simp add*: *cCons-def*)

lemma *strip-while-coeffs* [*simp*]:
strip-while (*HOL.eq 0*) (*coeffs p*) = *coeffs p*
by (*cases p* = *0*) (*auto dest*: *last-coeffs-not-0 intro*: *strip-while-not-last*)

lemma *coeffs-eq-iff*:
p = *q* ⟷ *coeffs p* = *coeffs q* (**is** *?P* ⟷ *?Q*)
proof
assume *?P* **then show** *?Q* **by** *simp*
next
assume *?Q*
then have *Poly* (*coeffs p*) = *Poly* (*coeffs q*) **by** *simp*
then show *?P* **by** *simp*
qed

lemma *coeff-Poly-eq*:
coeff (*Poly xs*) *n* = *nth-default 0 xs n*
apply (*induct xs arbitrary*: *n*) **apply** *simp-all*
by (*metis nat.case not0-implies-Suc nth-default-Cons-0 nth-default-Cons-Suc pCons.rep-eq*)

lemma *nth-default-coeffs-eq*:
nth-default 0 (*coeffs p*) = *coeff p*
by (*simp add*: *fun-eq-iff coeff-Poly-eq* [*symmetric*])

lemma [*code*]:
coeff p = *nth-default 0* (*coeffs p*)
by (*simp add*: *nth-default-coeffs-eq*)

lemma *coeffs-eqI*:
assumes *coeff*: ⋀*n*. *coeff p n* = *nth-default 0 xs n*
assumes *zero*: *xs* ≠ [] ⟹ *last xs* ≠ *0*
shows *coeffs p* = *xs*
proof −
from *coeff* **have** *p* = *Poly xs* **by** (*simp add*: *poly-eq-iff coeff-Poly-eq*)
with *zero* **show** *?thesis* **by** *simp* (*cases xs*, *simp-all*)

qed

lemma *degree-eq-length-coeffs* [*code*]:
degree p = *length* (*coeffs p*) − *1*
by (*simp add*: *coeffs-def*)

lemma *length-coeffs-degree*:
$p \neq 0 \Longrightarrow$ *length* (*coeffs p*) = *Suc* (*degree p*)
by (*induct p*) (*auto simp add*: *cCons-def*)

lemma [*code abstract*]:
coeffs 0 = []
by (*fact coeffs-0-eq-Nil*)

lemma [*code abstract*]:
coeffs (*pCons a p*) = *a* ## *coeffs p*
by (*fact coeffs-pCons-eq-cCons*)

instantiation *poly* :: ({*zero*, *equal*}) *equal*
begin

definition
[*code*]: *HOL.equal* (*p*::'*a poly*) *q* ⟷ *HOL.equal* (*coeffs p*) (*coeffs q*)

instance
by *standard* (*simp add*: *equal equal-poly-def coeffs-eq-iff*)

end

lemma [*code nbe*]: *HOL.equal* (*p* :: - *poly*) *p* ⟷ *True*
by (*fact equal-refl*)

definition *is-zero* :: '*a*::*zero poly* ⇒ *bool*
where
[*code*]: *is-zero p* ⟷ *List.null* (*coeffs p*)

lemma *is-zero-null* [*code-abbrev*]:
is-zero p ⟷ *p* = *0*
by (*simp add*: *is-zero-def null-def*)

41.9 Fold combinator for polynomials

definition *fold-coeffs* :: ('*a*::*zero* ⇒ '*b* ⇒ '*b*) ⇒ '*a poly* ⇒ '*b* ⇒ '*b*
where
fold-coeffs f p = *foldr f* (*coeffs p*)

lemma *fold-coeffs-0-eq* [*simp*]:
fold-coeffs f 0 = *id*
by (*simp add*: *fold-coeffs-def*)

lemma *fold-coeffs-pCons-eq* [*simp*]:
$f\ 0 = id \Longrightarrow$ *fold-coeffs f* (*pCons a p*) = *f a* ∘ *fold-coeffs f p*
by (*simp add*: *fold-coeffs-def cCons-def fun-eq-iff*)

lemma *fold-coeffs-pCons-0-0-eq* [*simp*]:
fold-coeffs f (*pCons 0 0*) = *id*
by (*simp add*: *fold-coeffs-def*)

lemma *fold-coeffs-pCons-coeff-not-0-eq* [*simp*]:
$a \neq 0 \Longrightarrow$ *fold-coeffs f* (*pCons a p*) = *f a* ∘ *fold-coeffs f p*
by (*simp add*: *fold-coeffs-def*)

lemma *fold-coeffs-pCons-not-0-0-eq* [*simp*]:
$p \neq 0 \Longrightarrow$ *fold-coeffs f* (*pCons a p*) = *f a* ∘ *fold-coeffs f p*
by (*simp add*: *fold-coeffs-def*)

41.10 Canonical morphism on polynomials – evaluation

definition *poly* :: *'a::comm-semiring-0 poly* ⇒ *'a* ⇒ *'a*
where
poly p = *fold-coeffs* (λ*a f x*. *a* + *x* ∗ *f x*) *p* (λ*x*. *0*) — The Horner Schema

lemma *poly-0* [*simp*]:
poly 0 x = *0*
by (*simp add*: *poly-def*)

lemma *poly-pCons* [*simp*]:
poly (*pCons a p*) *x* = *a* + *x* ∗ *poly p x*
by (*cases p* = *0* ∧ *a* = *0*) (*auto simp add*: *poly-def*)

lemma *poly-altdef*:
poly p (*x* :: *'a* :: {*comm-semiring-0*, *semiring-1*}) = (∑ *i*≤*degree p*. *coeff p i* ∗ *x* ^ *i*)
proof (*induction p rule*: *pCons-induct*)
case (*pCons a p*)
show *?case*
proof (*cases p* = *0*)
case *False*
let *?p'* = *pCons a p*
note *poly-pCons*[*of a p x*]
also note *pCons.IH*
also have *a* + *x* ∗ (∑ *i*≤*degree p*. *coeff p i* ∗ *x* ^ *i*) =
coeff ?p' 0 ∗ *x*^*0* + (∑ *i*≤*degree p*. *coeff ?p'* (*Suc i*) ∗ *x*^*Suc i*)
by (*simp add*: *field-simps setsum-right-distrib coeff-pCons*)
also note *setsum-atMost-Suc-shift*[*symmetric*]
also note *degree-pCons-eq*[*OF* ‹*p* ≠ *0*›, *of a*, *symmetric*]
finally show *?thesis* **.**
qed *simp*

qed *simp*

lemma *poly-0-coeff-0*: *poly p 0 = coeff p 0*
by (*cases p*) (*auto simp*: *poly-altdef*)

41.11 Monomials

lift-definition *monom* :: *'a ⇒ nat ⇒ 'a*::*zero poly*
is *λa m n. if m = n then a else 0*
by (*simp add*: *MOST-iff-cofinite*)

lemma *coeff-monom* [*simp*]:
coeff (*monom a m*) *n* = (*if m = n then a else 0*)
by *transfer rule*

lemma *monom-0*:
monom a 0 = pCons a 0
by (*rule poly-eqI*) (*simp add*: *coeff-pCons split*: *nat.split*)

lemma *monom-Suc*:
monom a (*Suc n*) = *pCons 0* (*monom a n*)
by (*rule poly-eqI*) (*simp add*: *coeff-pCons split*: *nat.split*)

lemma *monom-eq-0* [*simp*]: *monom 0 n = 0*
by (*rule poly-eqI*) *simp*

lemma *monom-eq-0-iff* [*simp*]: *monom a n = 0 ⟷ a = 0*
by (*simp add*: *poly-eq-iff*)

lemma *monom-eq-iff* [*simp*]: *monom a n = monom b n ⟷ a = b*
by (*simp add*: *poly-eq-iff*)

lemma *degree-monom-le*: *degree* (*monom a n*) ≤ *n*
by (*rule degree-le*, *simp*)

lemma *degree-monom-eq*: *a ≠ 0 ⟹ degree* (*monom a n*) = *n*
apply (*rule order-antisym* [*OF degree-monom-le*])
apply (*rule le-degree*, *simp*)
done

lemma *coeffs-monom* [*code abstract*]:
coeffs (*monom a n*) = (*if a = 0 then* [] *else replicate n 0* @ [*a*])
by (*induct n*) (*simp-all add*: *monom-0 monom-Suc*)

lemma *fold-coeffs-monom* [*simp*]:
a ≠ 0 ⟹ fold-coeffs f (*monom a n*) = *f 0 ^^ n ∘ f a*
by (*simp add*: *fold-coeffs-def coeffs-monom fun-eq-iff*)

lemma *poly-monom*:

fixes *a x* :: *'a*::{*comm-semiring-1*}
shows *poly* (*monom a n*) *x* = *a* ∗ *x* ˆ *n*
by (*cases a* = *0*, *simp-all*)
(*induct n*, *simp-all add*: *mult.left-commute poly-def*)

41.12 Addition and subtraction

instantiation *poly* :: (*comm-monoid-add*) *comm-monoid-add*
begin

lift-definition *plus-poly* :: *'a poly* ⇒ *'a poly* ⇒ *'a poly*
is *λp q n. coeff p n* + *coeff q n*
proof −
fix *q p* :: *'a poly*
show $\forall_\infty$*n. coeff p n* + *coeff q n* = *0*
using *MOST-coeff-eq-0*[*of p*] *MOST-coeff-eq-0*[*of q*] **by** *eventually-elim simp*
qed

lemma *coeff-add* [*simp*]: *coeff* (*p* + *q*) *n* = *coeff p n* + *coeff q n*
by (*simp add*: *plus-poly.rep-eq*)

instance
proof
fix *p q r* :: *'a poly*
show (*p* + *q*) + *r* = *p* + (*q* + *r*)
by (*simp add*: *poly-eq-iff add.assoc*)
show *p* + *q* = *q* + *p*
by (*simp add*: *poly-eq-iff add.commute*)
show *0* + *p* = *p*
by (*simp add*: *poly-eq-iff*)
qed

end

instantiation *poly* :: (*cancel-comm-monoid-add*) *cancel-comm-monoid-add*
begin

lift-definition *minus-poly* :: *'a poly* ⇒ *'a poly* ⇒ *'a poly*
is *λp q n. coeff p n* − *coeff q n*
proof −
fix *q p* :: *'a poly*
show $\forall_\infty$*n. coeff p n* − *coeff q n* = *0*
using *MOST-coeff-eq-0*[*of p*] *MOST-coeff-eq-0*[*of q*] **by** *eventually-elim simp*
qed

lemma *coeff-diff* [*simp*]: *coeff* (*p* − *q*) *n* = *coeff p n* − *coeff q n*
by (*simp add*: *minus-poly.rep-eq*)

instance

proof
  fix *p q r* :: *'a poly*
  show *p* + *q* − *p* = *q*
    by (*simp add*: *poly-eq-iff*)
  show *p* − *q* − *r* = *p* − (*q* + *r*)
    by (*simp add*: *poly-eq-iff diff-diff-eq*)
qed

end

instantiation *poly* :: (*ab-group-add*) *ab-group-add*
begin

lift-definition *uminus-poly* :: *'a poly* ⇒ *'a poly*
  is λ*p n*. − *coeff p n*
proof −
  fix *p* :: *'a poly*
  show $\forall_\infty$*n*. − *coeff p n* = *0*
    using *MOST-coeff-eq-0* **by** *simp*
qed

lemma *coeff-minus* [*simp*]: *coeff* (− *p*) *n* = − *coeff p n*
  by (*simp add*: *uminus-poly.rep-eq*)

instance
proof
  fix *p q* :: *'a poly*
  show − *p* + *p* = *0*
    by (*simp add*: *poly-eq-iff*)
  show *p* − *q* = *p* + − *q*
    by (*simp add*: *poly-eq-iff*)
qed

end

lemma *add-pCons* [*simp*]:
  pCons a p + *pCons b q* = *pCons* (*a* + *b*) (*p* + *q*)
  by (*rule poly-eqI*, *simp add*: *coeff-pCons split*: *nat.split*)

lemma *minus-pCons* [*simp*]:
  − *pCons a p* = *pCons* (− *a*) (− *p*)
  by (*rule poly-eqI*, *simp add*: *coeff-pCons split*: *nat.split*)

lemma *diff-pCons* [*simp*]:
  pCons a p − *pCons b q* = *pCons* (*a* − *b*) (*p* − *q*)
  by (*rule poly-eqI*, *simp add*: *coeff-pCons split*: *nat.split*)

lemma *degree-add-le-max*: *degree* (*p* + *q*) ≤ *max* (*degree p*) (*degree q*)
  by (*rule degree-le*, *auto simp add*: *coeff-eq-0*)

lemma *degree-add-le*:
$[\![$*degree p* $\leq$ *n*; *degree q* $\leq$ *n*$]\!] \Longrightarrow$ *degree* (*p* + *q*) $\leq$ *n*
by (*auto intro*: *order-trans degree-add-le-max*)

lemma *degree-add-less*:
$[\![$*degree p* < *n*; *degree q* < *n*$]\!] \Longrightarrow$ *degree* (*p* + *q*) < *n*
by (*auto intro*: *le-less-trans degree-add-le-max*)

lemma *degree-add-eq-right*:
degree p < *degree q* $\Longrightarrow$ *degree* (*p* + *q*) = *degree q*
apply (*cases q* = *0*, *simp*)
apply (*rule order-antisym*)
apply (*simp add*: *degree-add-le*)
apply (*rule le-degree*)
apply (*simp add*: *coeff-eq-0*)
done

lemma *degree-add-eq-left*:
degree q < *degree p* $\Longrightarrow$ *degree* (*p* + *q*) = *degree p*
using *degree-add-eq-right* [*of q p*]
by (*simp add*: *add.commute*)

lemma *degree-minus* [*simp*]:
degree (− *p*) = *degree p*
unfolding *degree-def* **by** *simp*

lemma *degree-diff-le-max*:
fixes *p q* :: *'a* :: *ab-group-add poly*
shows *degree* (*p* − *q*) $\leq$ *max* (*degree p*) (*degree q*)
using *degree-add-le* [**where** *p*=*p* **and** *q*=−*q*]
by *simp*

lemma *degree-diff-le*:
fixes *p q* :: *'a* :: *ab-group-add poly*
assumes *degree p* $\leq$ *n* **and** *degree q* $\leq$ *n*
shows *degree* (*p* − *q*) $\leq$ *n*
using *assms degree-add-le* [*of p n* − *q*] **by** *simp*

lemma *degree-diff-less*:
fixes *p q* :: *'a* :: *ab-group-add poly*
assumes *degree p* < *n* **and** *degree q* < *n*
shows *degree* (*p* − *q*) < *n*
using *assms degree-add-less* [*of p n* − *q*] **by** *simp*

lemma *add-monom*: *monom a n* + *monom b n* = *monom* (*a* + *b*) *n*
by (*rule poly-eqI*) *simp*

lemma *diff-monom*: *monom a n* − *monom b n* = *monom* (*a* − *b*) *n*

by (*rule poly-eqI*) *simp*

lemma *minus-monom*: $-$ *monom a n* $=$ *monom* $(-a)$ *n*
by (*rule poly-eqI*) *simp*

lemma *coeff-setsum*: *coeff* $(\sum x\in A.\ p\ x)\ i = (\sum x\in A.\ coeff\ (p\ x)\ i)$
by (*cases finite A*, *induct set*: *finite*, *simp-all*)

lemma *monom-setsum*: *monom* $(\sum x\in A.\ a\ x)\ n = (\sum x\in A.\ monom\ (a\ x)\ n)$
by (*rule poly-eqI*) (*simp add*: *coeff-setsum*)

fun *plus-coeffs* :: *'a::comm-monoid-add list* ⇒ *'a list* ⇒ *'a list*
where
plus-coeffs xs [] = *xs*
| *plus-coeffs* [] *ys* = *ys*
| *plus-coeffs* (*x* # *xs*) (*y* # *ys*) = (*x* + *y*) ## *plus-coeffs xs ys*

lemma *coeffs-plus-eq-plus-coeffs* [*code abstract*]:
coeffs (*p* + *q*) = *plus-coeffs* (*coeffs p*) (*coeffs q*)
proof −
{ **fix** *xs ys* :: *'a list* **and** *n*
have *nth-default 0* (*plus-coeffs xs ys*) *n* = *nth-default 0 xs n* + *nth-default 0 ys n*
proof (*induct xs ys arbitrary*: *n rule*: *plus-coeffs.induct*)
case (*3 x xs y ys n*)
then show *?case* **by** (*cases n*) (*auto simp add*: *cCons-def*)
qed *simp-all* }
note ∗ = *this*
{ **fix** *xs ys* :: *'a list*
assume *xs* ≠ [] ⟹ *last xs* ≠ *0* **and** *ys* ≠ [] ⟹ *last ys* ≠ *0*
moreover assume *plus-coeffs xs ys* ≠ []
ultimately have *last* (*plus-coeffs xs ys*) ≠ *0*
proof (*induct xs ys rule*: *plus-coeffs.induct*)
case (*3 x xs y ys*) **then show** *?case* **by** (*auto simp add*: *cCons-def*) *metis*
qed *simp-all* }
note ∗∗ = *this*
show *?thesis*
apply (*rule coeffs-eqI*)
apply (*simp add*: ∗ *nth-default-coeffs-eq*)
apply (*rule* ∗∗)
apply (*auto dest*: *last-coeffs-not-0*)
done
qed

lemma *coeffs-uminus* [*code abstract*]:
coeffs (− *p*) = *map* (λ*a*. − *a*) (*coeffs p*)
by (*rule coeffs-eqI*)
(*simp-all add*: *not-0-coeffs-not-Nil last-map last-coeffs-not-0 nth-default-map-eq nth-default-coeffs-eq*)

lemma [*code*]:
  fixes *p q* :: *'a*::*ab-group-add poly*
  shows $p - q = p + - q$
  by (*fact diff-conv-add-uminus*)

lemma *poly-add* [*simp*]: *poly* (*p* + *q*) *x* = *poly p x* + *poly q x*
  apply (*induct p arbitrary*: *q*, *simp*)
  apply (*case-tac q*, *simp*, *simp add*: *algebra-simps*)
  done

lemma *poly-minus* [*simp*]:
  fixes *x* :: *'a*::*comm-ring*
  shows *poly* (− *p*) *x* = − *poly p x*
  by (*induct p*) *simp-all*

lemma *poly-diff* [*simp*]:
  fixes *x* :: *'a*::*comm-ring*
  shows *poly* (*p* − *q*) *x* = *poly p x* − *poly q x*
  using *poly-add* [*of p* − *q x*] **by** *simp*

lemma *poly-setsum*: *poly* ($\sum$ *k*∈*A*. *p k*) *x* = ($\sum$ *k*∈*A*. *poly* (*p k*) *x*)
  by (*induct A rule*: *infinite-finite-induct*) *simp-all*

lemma *degree-setsum-le*: *finite S* ⟹ ($\bigwedge$ *p* . *p* ∈ *S* ⟹ *degree* (*f p*) ≤ *n*)
  ⟹ *degree* (*setsum f S*) ≤ *n*
proof (*induct S rule*: *finite-induct*)
  case (*insert p S*)
  hence *degree* (*setsum f S*) ≤ *n degree* (*f p*) ≤ *n* **by** *auto*
  thus *?case* **unfolding** *setsum.insert*[*OF insert*(*1*−*2*)] **by** (*metis degree-add-le*)
qed *simp*

lemma *poly-as-sum-of-monoms'*:
  assumes *n*: *degree p* ≤ *n*
  shows ($\sum$ *i*≤*n*. *monom* (*coeff p i*) *i*) = *p*
proof −
  have *eq*: $\bigwedge$*i*. {..*n*} ∩ {*i*} = (*if i* ≤ *n then* {*i*} *else* {})
    by *auto*
  show *?thesis*
    using *n* **by** (*simp add*: *poly-eq-iff coeff-setsum coeff-eq-0 setsum.If-cases eq*
        if-distrib[**where** *f*=λ*x*. *x* ∗ *a* **for** *a*])
qed

lemma *poly-as-sum-of-monoms*: ($\sum$ *i*≤*degree p*. *monom* (*coeff p i*) *i*) = *p*
  by (*intro poly-as-sum-of-monoms' order-refl*)

lemma *Poly-snoc*: *Poly* (*xs* @ [*x*]) = *Poly xs* + *monom x* (*length xs*)
  by (*induction xs*) (*simp-all add*: *monom-0 monom-Suc*)

41.13 Multiplication by a constant, polynomial multiplication and the unit polynomial

lift-definition *smult* :: *'a::comm-semiring-0* ⇒ *'a poly* ⇒ *'a poly*
  is *λa p n. a ∗ coeff p n*
proof −
  fix *a* :: *'a* **and** *p* :: *'a poly* **show** *∀∞ i. a ∗ coeff p i = 0*
    using *MOST-coeff-eq-0[of p]* **by** *eventually-elim simp*
qed

lemma *coeff-smult* [*simp*]:
  coeff (smult a p) n = a ∗ coeff p n
  by (*simp add*: *smult.rep-eq*)

lemma *degree-smult-le*: *degree (smult a p) ≤ degree p*
  by (*rule degree-le*, *simp add*: *coeff-eq-0*)

lemma *smult-smult* [*simp*]: *smult a (smult b p) = smult (a ∗ b) p*
  by (*rule poly-eqI*, *simp add*: *mult.assoc*)

lemma *smult-0-right* [*simp*]: *smult a 0 = 0*
  by (*rule poly-eqI*, *simp*)

lemma *smult-0-left* [*simp*]: *smult 0 p = 0*
  by (*rule poly-eqI*, *simp*)

lemma *smult-1-left* [*simp*]: *smult (1::'a::comm-semiring-1) p = p*
  by (*rule poly-eqI*, *simp*)

lemma *smult-add-right*:
  smult a (p + q) = smult a p + smult a q
  by (*rule poly-eqI*, *simp add*: *algebra-simps*)

lemma *smult-add-left*:
  smult (a + b) p = smult a p + smult b p
  by (*rule poly-eqI*, *simp add*: *algebra-simps*)

lemma *smult-minus-right* [*simp*]:
  smult (a::'a::comm-ring) (− p) = − smult a p
  by (*rule poly-eqI*, *simp*)

lemma *smult-minus-left* [*simp*]:
  smult (− a::'a::comm-ring) p = − smult a p
  by (*rule poly-eqI*, *simp*)

lemma *smult-diff-right*:
  smult (a::'a::comm-ring) (p − q) = smult a p − smult a q
  by (*rule poly-eqI*, *simp add*: *algebra-simps*)

lemma *smult-diff-left*:

smult (*a* − *b*::′*a*::*comm-ring*) *p* = *smult a p* − *smult b p*
by (*rule poly-eqI*, *simp add*: *algebra-simps*)

lemmas *smult-distribs* =
smult-add-left smult-add-right
smult-diff-left smult-diff-right

lemma *smult-pCons* [*simp*]:
smult a (*pCons b p*) = *pCons* (*a* ∗ *b*) (*smult a p*)
by (*rule poly-eqI*, *simp add*: *coeff-pCons split*: *nat.split*)

lemma *smult-monom*: *smult a* (*monom b n*) = *monom* (*a* ∗ *b*) *n*
by (*induct n*, *simp add*: *monom-0*, *simp add*: *monom-Suc*)

lemma *degree-smult-eq* [*simp*]:
fixes *a* :: ′*a*::*idom*
shows *degree* (*smult a p*) = (*if a* = *0 then 0 else degree p*)
by (*cases a* = *0*, *simp*, *simp add*: *degree-def*)

lemma *smult-eq-0-iff* [*simp*]:
fixes *a* :: ′*a*::*idom*
shows *smult a p* = *0* ⟷ *a* = *0* ∨ *p* = *0*
by (*simp add*: *poly-eq-iff*)

lemma *coeffs-smult* [*code abstract*]:
fixes *p* :: ′*a*::*idom poly*
shows *coeffs* (*smult a p*) = (*if a* = *0 then* [] *else map* (*Groups.times a*) (*coeffs p*))
by (*rule coeffs-eqI*)
(*auto simp add*: *not-0-coeffs-not-Nil last-map last-coeffs-not-0 nth-default-map-eq nth-default-coeffs-eq*)

instantiation *poly* :: (*comm-semiring-0*) *comm-semiring-0*
begin

definition
p ∗ *q* = *fold-coeffs* (λ*a p*. *smult a q* + *pCons 0 p*) *p 0*

lemma *mult-poly-0-left*: (*0*::′*a poly*) ∗ *q* = *0*
by (*simp add*: *times-poly-def*)

lemma *mult-pCons-left* [*simp*]:
pCons a p ∗ *q* = *smult a q* + *pCons 0* (*p* ∗ *q*)
by (*cases p* = *0* ∧ *a* = *0*) (*auto simp add*: *times-poly-def*)

lemma *mult-poly-0-right*: *p* ∗ (*0*::′*a poly*) = *0*
by (*induct p*) (*simp add*: *mult-poly-0-left*, *simp*)

lemma *mult-pCons-right* [*simp*]:

```
  p * pCons a q = smult a p + pCons 0 (p * q)
  by (induct p) (simp add: mult-poly-0-left, simp add: algebra-simps)

lemmas mult-poly-0 = mult-poly-0-left mult-poly-0-right

lemma mult-smult-left [simp]:
  smult a p * q = smult a (p * q)
  by (induct p) (simp add: mult-poly-0, simp add: smult-add-right)

lemma mult-smult-right [simp]:
  p * smult a q = smult a (p * q)
  by (induct q) (simp add: mult-poly-0, simp add: smult-add-right)

lemma mult-poly-add-left:
  fixes p q r :: 'a poly
  shows (p + q) * r = p * r + q * r
  by (induct r) (simp add: mult-poly-0, simp add: smult-distribs algebra-simps)

instance
proof
  fix p q r :: 'a poly
  show 0: 0 * p = 0
    by (rule mult-poly-0-left)
  show p * 0 = 0
    by (rule mult-poly-0-right)
  show (p + q) * r = p * r + q * r
    by (rule mult-poly-add-left)
  show (p * q) * r = p * (q * r)
    by (induct p, simp add: mult-poly-0, simp add: mult-poly-add-left)
  show p * q = q * p
    by (induct p, simp add: mult-poly-0, simp)
qed

end

instance poly :: (comm-semiring-0-cancel) comm-semiring-0-cancel ..

lemma coeff-mult:
  coeff (p * q) n = (∑ i≤n. coeff p i * coeff q (n−i))
proof (induct p arbitrary: n)
  case 0 show ?case by simp
next
  case (pCons a p n) thus ?case
    by (cases n, simp, simp add: setsum-atMost-Suc-shift
                      del: setsum-atMost-Suc)
qed

lemma degree-mult-le: degree (p * q) ≤ degree p + degree q
apply (rule degree-le)
```

apply (*induct p*)
apply *simp*
apply (*simp add*: *coeff-eq-0 coeff-pCons split*: *nat.split*)
done

lemma *mult-monom*: *monom a m* ∗ *monom b n* = *monom* (*a* ∗ *b*) (*m* + *n*)
by (*induct m*) (*simp add*: *monom-0 smult-monom*, *simp add*: *monom-Suc*)

instantiation *poly* :: (*comm-semiring-1*) *comm-semiring-1*
begin

definition *one-poly-def*: *1* = *pCons 1 0*

instance
proof
show *1* ∗ *p* = *p* **for** *p* :: *'a poly*
unfolding *one-poly-def* **by** *simp*
show *0* ≠ (*1*::*'a poly*)
unfolding *one-poly-def* **by** *simp*
qed

end

instance *poly* :: (*comm-ring*) *comm-ring* **..**

instance *poly* :: (*comm-ring-1*) *comm-ring-1* **..**

lemma *coeff-1* [*simp*]: *coeff 1 n* = (*if n* = *0 then 1 else 0*)
unfolding *one-poly-def*
by (*simp add*: *coeff-pCons split*: *nat.split*)

lemma *monom-eq-1* [*simp*]:
monom 1 0 = *1*
by (*simp add*: *monom-0 one-poly-def*)

lemma *degree-1* [*simp*]: *degree 1* = *0*
unfolding *one-poly-def*
by (*rule degree-pCons-0*)

lemma *coeffs-1-eq* [*simp*, *code abstract*]:
coeffs 1 = [*1*]
by (*simp add*: *one-poly-def*)

lemma *degree-power-le*:
degree (*p* ^ *n*) ≤ *degree p* ∗ *n*
by (*induct n*) (*auto intro*: *order-trans degree-mult-le*)

lemma *poly-smult* [*simp*]:
poly (*smult a p*) *x* = *a* ∗ *poly p x*

by (*induct p*, *simp*, *simp add*: *algebra-simps*)

lemma *poly-mult* [*simp*]:
poly (*p* ∗ *q*) *x* = *poly p x* ∗ *poly q x*
by (*induct p*, *simp-all*, *simp add*: *algebra-simps*)

lemma *poly-1* [*simp*]:
poly 1 x = *1*
by (*simp add*: *one-poly-def*)

lemma *poly-power* [*simp*]:
fixes *p* :: *'a*::{*comm-semiring-1*} *poly*
shows *poly* (*p* ˆ *n*) *x* = *poly p x* ˆ *n*
by (*induct n*) *simp-all*

lemma *poly-setprod*: *poly* (∏ *k*∈*A*. *p k*) *x* = (∏ *k*∈*A*. *poly* (*p k*) *x*)
by (*induct A rule*: *infinite-finite-induct*) *simp-all*

lemma *degree-setprod-setsum-le*: *finite S* ⟹ *degree* (*setprod f S*) ≤ *setsum* (*degree o f*) *S*
proof (*induct S rule*: *finite-induct*)
case (*insert a S*)
show *?case* **unfolding** *setprod.insert*[*OF insert*(*1*−*2*)] *setsum.insert*[*OF insert*(*1*−*2*)]
by (*rule le-trans*[*OF degree-mult-le*], *insert insert*, *auto*)
qed *simp*

41.14 Conversions from natural numbers

lemma *of-nat-poly*: *of-nat n* = [:*of-nat n* :: *'a* :: *comm-semiring-1*:]
proof (*induction n*)
case (*Suc n*)
hence *of-nat* (*Suc n*) = *1* + (*of-nat n* :: *'a poly*)
by *simp*
also have (*of-nat n* :: *'a poly*) = [: *of-nat n* :]
by (*subst Suc*) (*rule refl*)
also have *1* = [:*1*:] **by** (*simp add*: *one-poly-def*)
finally show *?case* **by** (*subst* (*asm*) *add-pCons*) *simp*
qed *simp*

lemma *degree-of-nat* [*simp*]: *degree* (*of-nat n*) = *0*
by (*simp add*: *of-nat-poly*)

lemma *degree-numeral* [*simp*]: *degree* (*numeral n*) = *0*
by (*subst of-nat-numeral* [*symmetric*], *subst of-nat-poly*) *simp*

lemma *numeral-poly*: *numeral n* = [:*numeral n*:]
by (*subst of-nat-numeral* [*symmetric*], *subst of-nat-poly*) *simp*

41.15 Lemmas about divisibility

lemma *dvd-smult*: *p dvd q* $\Longrightarrow$ *p dvd smult a q*
proof –
  assume *p dvd q*
  then obtain *k* **where** $q = p * k$ **..**
  then have *smult a q* = *p* ∗ *smult a k* **by** *simp*
  then show *p dvd smult a q* **..**
qed

lemma *dvd-smult-cancel*:
  fixes *a* :: *'a* :: *field*
  shows *p dvd smult a q* $\Longrightarrow$ $a \neq 0$ $\Longrightarrow$ *p dvd q*
  by (*drule dvd-smult* [**where** *a*=*inverse a*]) *simp*

lemma *dvd-smult-iff*:
  fixes *a* :: *'a*::*field*
  shows $a \neq 0$ $\Longrightarrow$ *p dvd smult a q* $\longleftrightarrow$ *p dvd q*
  by (*safe elim*!: *dvd-smult dvd-smult-cancel*)

lemma *smult-dvd-cancel*:
  smult a p dvd q $\Longrightarrow$ *p dvd q*
proof –
  assume *smult a p dvd q*
  then obtain *k* **where** *q* = *smult a p* ∗ *k* **..**
  then have *q* = *p* ∗ *smult a k* **by** *simp*
  then show *p dvd q* **..**
qed

lemma *smult-dvd*:
  fixes *a* :: *'a*::*field*
  shows *p dvd q* $\Longrightarrow$ $a \neq 0$ $\Longrightarrow$ *smult a p dvd q*
  by (*rule smult-dvd-cancel* [**where** *a*=*inverse a*]) *simp*

lemma *smult-dvd-iff*:
  fixes *a* :: *'a*::*field*
  shows *smult a p dvd q* $\longleftrightarrow$ (*if a* = *0 then q* = *0 else p dvd q*)
  by (*auto elim*: *smult-dvd smult-dvd-cancel*)

41.16 Polynomials form an integral domain

lemma *coeff-mult-degree-sum*:
  coeff (*p* ∗ *q*) (*degree p* + *degree q*) =
   coeff p (*degree p*) ∗ *coeff q* (*degree q*)
  by (*induct p*, *simp*, *simp add*: *coeff-eq-0*)

instance *poly* :: (*idom*) *idom*
proof
  fix *p q* :: *'a poly*
  assume $p \neq 0$ **and** $q \neq 0$

have *coeff* $(p * q)$ $(degree\ p + degree\ q) =$
  coeff p $(degree\ p) * coeff\ q\ (degree\ q)$
 by (*rule coeff-mult-degree-sum*)
also have $coeff\ p\ (degree\ p) * coeff\ q\ (degree\ q) \neq 0$
 using $\langle p \neq 0 \rangle$ **and** $\langle q \neq 0 \rangle$ **by** *simp*
finally have $\exists\, n.\ coeff\ (p * q)\ n \neq 0$ **..**
thus $p * q \neq 0$ **by** (*simp add*: *poly-eq-iff*)
qed

lemma *degree-mult-eq*:
 fixes $p\ q$ :: *'a::semidom poly*
 shows $[\![p \neq 0;\ q \neq 0]\!] \Longrightarrow degree\ (p * q) = degree\ p + degree\ q$
apply (*rule order-antisym* [*OF degree-mult-le le-degree*])
apply (*simp add*: *coeff-mult-degree-sum*)
done

lemma *degree-mult-right-le*:
 fixes $p\ q$ :: *'a::semidom poly*
 assumes $q \neq 0$
 shows $degree\ p \leq degree\ (p * q)$
 using *assms* **by** (*cases* $p = 0$) (*simp-all add*: *degree-mult-eq*)

lemma *coeff-degree-mult*:
 fixes $p\ q$ :: *'a::semidom poly*
 shows $coeff\ (p * q)\ (degree\ (p * q)) =$
  $coeff\ q\ (degree\ q) * coeff\ p\ (degree\ p)$
 by (*cases* $p = 0 \lor q = 0$) (*auto simp add*: *degree-mult-eq coeff-mult-degree-sum mult-ac*)

lemma *dvd-imp-degree-le*:
 fixes $p\ q$ :: *'a::semidom poly*
 shows $[\![p\ dvd\ q;\ q \neq 0]\!] \Longrightarrow degree\ p \leq degree\ q$
 by (*erule dvdE*, *hypsubst*, *subst degree-mult-eq*) *auto*

lemma *divides-degree*:
 assumes *pq*: $p\ dvd\ (q :: 'a :: semidom\ poly)$
 shows $degree\ p \leq degree\ q \lor q = 0$
 by (*metis dvd-imp-degree-le pq*)

41.17 Polynomials form an ordered integral domain

definition *pos-poly* :: *'a::linordered-idom poly* $\Rightarrow$ *bool*
where
 $pos\text{-}poly\ p \longleftrightarrow 0 < coeff\ p\ (degree\ p)$

lemma *pos-poly-pCons*:
 $pos\text{-}poly\ (pCons\ a\ p) \longleftrightarrow pos\text{-}poly\ p \lor (p = 0 \land 0 < a)$
 unfolding *pos-poly-def* **by** *simp*

lemma *not-pos-poly-0* [*simp*]: ¬ *pos-poly 0*
  unfolding *pos-poly-def* **by** *simp*

lemma *pos-poly-add*: ⟦*pos-poly p*; *pos-poly q*⟧ ⟹ *pos-poly* (*p* + *q*)
  apply (*induct p arbitrary*: *q*, *simp*)
  apply (*case-tac q*, *force simp add*: *pos-poly-pCons add-pos-pos*)
  done

lemma *pos-poly-mult*: ⟦*pos-poly p*; *pos-poly q*⟧ ⟹ *pos-poly* (*p* ∗ *q*)
  unfolding *pos-poly-def*
  apply (*subgoal-tac p* ≠ *0* ∧ *q* ≠ *0*)
  apply (*simp add*: *degree-mult-eq coeff-mult-degree-sum*)
  apply *auto*
  done

lemma *pos-poly-total*: *p* = *0* ∨ *pos-poly p* ∨ *pos-poly* (− *p*)
by (*induct p*) (*auto simp add*: *pos-poly-pCons*)

lemma *last-coeffs-eq-coeff-degree*:
  p ≠ *0* ⟹ *last* (*coeffs p*) = *coeff p* (*degree p*)
  by (*simp add*: *coeffs-def*)

lemma *pos-poly-coeffs* [*code*]:
  pos-poly p ⟷ (*let as* = *coeffs p in as* ≠ [] ∧ *last as* > *0*) (**is** *?P* ⟷ *?Q*)
proof
 assume *?Q* **then show** *?P* **by** (*auto simp add*: *pos-poly-def last-coeffs-eq-coeff-degree*)
next
  assume *?P* **then have** ∗: *0* < *coeff p* (*degree p*) **by** (*simp add*: *pos-poly-def*)
  then have *p* ≠ *0* **by** *auto*
  with ∗ **show** *?Q* **by** (*simp add*: *last-coeffs-eq-coeff-degree*)
qed

instantiation *poly* :: (*linordered-idom*) *linordered-idom*
begin

definition
  x < *y* ⟷ *pos-poly* (*y* − *x*)

definition
  x ≤ *y* ⟷ *x* = *y* ∨ *pos-poly* (*y* − *x*)

definition
  |*x*::*'a poly*| = (*if x* < *0 then* − *x else x*)

definition
  sgn (*x*::*'a poly*) = (*if x* = *0 then 0 else if 0* < *x then 1 else* − *1*)

instance
proof

fix *x y z* :: *'a poly*
show $x < y \longleftrightarrow x \leq y \land \neg\, y \leq x$
unfolding *less-eq-poly-def less-poly-def*
apply *safe*
apply *simp*
apply (*drule* (*1*) *pos-poly-add*)
apply *simp*
done
show $x \leq x$
unfolding *less-eq-poly-def* **by** *simp*
show $x \leq y \Longrightarrow y \leq z \Longrightarrow x \leq z$
unfolding *less-eq-poly-def*
apply *safe*
apply (*drule* (*1*) *pos-poly-add*)
apply (*simp add*: *algebra-simps*)
done
show $x \leq y \Longrightarrow y \leq x \Longrightarrow x = y$
unfolding *less-eq-poly-def*
apply *safe*
apply (*drule* (*1*) *pos-poly-add*)
apply *simp*
done
show $x \leq y \Longrightarrow z + x \leq z + y$
unfolding *less-eq-poly-def*
apply *safe*
apply (*simp add*: *algebra-simps*)
done
show $x \leq y \lor y \leq x$
unfolding *less-eq-poly-def*
using *pos-poly-total* [*of* $x - y$]
by *auto*
show $x < y \Longrightarrow 0 < z \Longrightarrow z * x < z * y$
unfolding *less-poly-def*
by (*simp add*: *right-diff-distrib* [*symmetric*] *pos-poly-mult*)
show $|x| = (\textit{if } x < 0 \textit{ then } - x \textit{ else } x)$
by (*rule abs-poly-def*)
show $\textit{sgn } x = (\textit{if } x = 0 \textit{ then } 0 \textit{ else if } 0 < x \textit{ then } 1 \textit{ else } - 1)$
by (*rule sgn-poly-def*)
qed

end

TODO: Simplification rules for comparisons

41.18 Synthetic division and polynomial roots

Synthetic division is simply division by the linear polynomial $x - c$.

definition *synthetic-divmod* :: *'a::comm-semiring-0 poly* ⇒ *'a* ⇒ *'a poly* × *'a*
where

synthetic-divmod p c = *fold-coeffs* (λ*a* (*q*, *r*). (*pCons r q*, *a* + *c* ∗ *r*)) *p* (*0*, *0*)

definition *synthetic-div* :: *'a::comm-semiring-0 poly* ⇒ *'a* ⇒ *'a poly*
where
synthetic-div p c = *fst* (*synthetic-divmod p c*)

lemma *synthetic-divmod-0* [*simp*]:
synthetic-divmod 0 c = (*0*, *0*)
by (*simp add*: *synthetic-divmod-def*)

lemma *synthetic-divmod-pCons* [*simp*]:
synthetic-divmod (*pCons a p*) *c* = (λ(*q*, *r*). (*pCons r q*, *a* + *c* ∗ *r*)) (*synthetic-divmod p c*)
by (*cases p* = *0* ∧ *a* = *0*) (*auto simp add*: *synthetic-divmod-def*)

lemma *synthetic-div-0* [*simp*]:
synthetic-div 0 c = *0*
unfolding *synthetic-div-def* **by** *simp*

lemma *synthetic-div-unique-lemma*: *smult c p* = *pCons a p* =⇒ *p* = *0*
by (*induct p arbitrary*: *a*) *simp-all*

lemma *snd-synthetic-divmod*:
snd (*synthetic-divmod p c*) = *poly p c*
by (*induct p*, *simp*, *simp add*: *split-def*)

lemma *synthetic-div-pCons* [*simp*]:
synthetic-div (*pCons a p*) *c* = *pCons* (*poly p c*) (*synthetic-div p c*)
unfolding *synthetic-div-def*
by (*simp add*: *split-def snd-synthetic-divmod*)

lemma *synthetic-div-eq-0-iff*:
synthetic-div p c = *0* ←→ *degree p* = *0*
by (*induct p*, *simp*, *case-tac p*, *simp*)

lemma *degree-synthetic-div*:
degree (*synthetic-div p c*) = *degree p* − *1*
by (*induct p*, *simp*, *simp add*: *synthetic-div-eq-0-iff*)

lemma *synthetic-div-correct*:
p + *smult c* (*synthetic-div p c*) = *pCons* (*poly p c*) (*synthetic-div p c*)
by (*induct p*) *simp-all*

lemma *synthetic-div-unique*:
p + *smult c q* = *pCons r q* =⇒ *r* = *poly p c* ∧ *q* = *synthetic-div p c*
apply (*induct p arbitrary*: *q r*)
apply (*simp*, *frule synthetic-div-unique-lemma*, *simp*)
apply (*case-tac q*, *force*)
done

lemma *synthetic-div-correct'*:
  fixes *c* :: *'a::comm-ring-1*
  shows [:−*c*, *1*:] ∗ *synthetic-div p c* + [:*poly p c*:] = *p*
  using *synthetic-div-correct* [*of p c*]
  by (*simp add*: *algebra-simps*)

lemma *poly-eq-0-iff-dvd*:
  fixes *c* :: *'a::idom*
  shows *poly p c* = *0* ⟷ [:−*c*, *1*:] *dvd p*
proof
  assume *poly p c* = *0*
  with *synthetic-div-correct'* [*of c p*]
  have *p* = [:−*c*, *1*:] ∗ *synthetic-div p c* **by** *simp*
  then show [:−*c*, *1*:] *dvd p* **..**
next
  assume [:−*c*, *1*:] *dvd p*
  then obtain *k* **where** *p* = [:−*c*, *1*:] ∗ *k* **by** (*rule dvdE*)
  then show *poly p c* = *0* **by** *simp*
qed

lemma *dvd-iff-poly-eq-0*:
  fixes *c* :: *'a::idom*
  shows [:*c*, *1*:] *dvd p* ⟷ *poly p* (−*c*) = *0*
  by (*simp add*: *poly-eq-0-iff-dvd*)

lemma *poly-roots-finite*:
  fixes *p* :: *'a::idom poly*
  shows *p* ≠ *0* ⟹ *finite* {*x*. *poly p x* = *0*}
proof (*induct n* ≡ *degree p arbitrary*: *p*)
  case (*0 p*)
  then obtain *a* **where** *a* ≠ *0* **and** *p* = [:*a*:]
    by (*cases p*, *simp split*: *if-splits*)
  then show *finite* {*x*. *poly p x* = *0*} **by** *simp*
next
  case (*Suc n p*)
  show *finite* {*x*. *poly p x* = *0*}
  proof (*cases* ∃ *x*. *poly p x* = *0*)
    case *False*
    then show *finite* {*x*. *poly p x* = *0*} **by** *simp*
  next
    case *True*
    then obtain *a* **where** *poly p a* = *0* **..**
    then have [:−*a*, *1*:] *dvd p* **by** (*simp only*: *poly-eq-0-iff-dvd*)
    then obtain *k* **where** *k*: *p* = [:−*a*, *1*:] ∗ *k* **..**
    with ‹*p* ≠ *0*› **have** *k* ≠ *0* **by** *auto*
    with *k* **have** *degree p* = *Suc* (*degree k*)
      by (*simp add*: *degree-mult-eq del*: *mult-pCons-left*)
    with ‹*Suc n* = *degree p*› **have** *n* = *degree k* **by** *simp*

then have *finite* {*x*. *poly k x* = *0*} **using** ‹*k* ≠ *0*› **by** (*rule Suc.hyps*)
then have *finite* (*insert a* {*x*. *poly k x* = *0*}) **by** *simp*
then show *finite* {*x*. *poly p x* = *0*}
by (*simp add*: *k Collect-disj-eq del*: *mult-pCons-left*)
qed
qed

lemma *poly-eq-poly-eq-iff*:
fixes *p q* :: *'a*::{*idom*,*ring-char-0*} *poly*
shows *poly p* = *poly q* ⟷ *p* = *q* (**is** *?P* ⟷ *?Q*)
proof
assume *?Q* **then show** *?P* **by** *simp*
next
{ **fix** *p* :: *'a*::{*idom*,*ring-char-0*} *poly*
have *poly p* = *poly 0* ⟷ *p* = *0*
apply (*cases p* = *0*, *simp-all*)
apply (*drule poly-roots-finite*)
apply (*auto simp add*: *infinite-UNIV-char-0*)
done
} **note** *this* [*of p* − *q*]
moreover assume *?P*
ultimately show *?Q* **by** *auto*
qed

lemma *poly-all-0-iff-0*:
fixes *p* :: *'a*::{*ring-char-0*, *idom*} *poly*
shows (∀ *x*. *poly p x* = *0*) ⟷ *p* = *0*
by (*auto simp add*: *poly-eq-poly-eq-iff* [*symmetric*])

41.19 Long division of polynomials

definition *pdivmod-rel* :: *'a*::*field poly* ⇒ *'a poly* ⇒ *'a poly* ⇒ *'a poly* ⇒ *bool*
where
pdivmod-rel x y q r ⟷
x = *q* ∗ *y* + *r* ∧ (*if y* = *0 then q* = *0 else r* = *0* ∨ *degree r* < *degree y*)

lemma *pdivmod-rel-0*:
pdivmod-rel 0 y 0 0
unfolding *pdivmod-rel-def* **by** *simp*

lemma *pdivmod-rel-by-0*:
pdivmod-rel x 0 0 x
unfolding *pdivmod-rel-def* **by** *simp*

lemma *eq-zero-or-degree-less*:
assumes *degree p* ≤ *n* **and** *coeff p n* = *0*
shows *p* = *0* ∨ *degree p* < *n*
proof (*cases n*)
case *0*

```
  with ‹degree p ≤ n› and ‹coeff p n = 0›
  have coeff p (degree p) = 0 by simp
  then have p = 0 by simp
  then show ?thesis ..
next
  case (Suc m)
  have ∀ i>n. coeff p i = 0
    using ‹degree p ≤ n› by (simp add: coeff-eq-0)
  then have ∀ i≥n. coeff p i = 0
    using ‹coeff p n = 0› by (simp add: le-less)
  then have ∀ i>m. coeff p i = 0
    using ‹n = Suc m› by (simp add: less-eq-Suc-le)
  then have degree p ≤ m
    by (rule degree-le)
  then have degree p < n
    using ‹n = Suc m› by (simp add: less-Suc-eq-le)
  then show ?thesis ..
qed

lemma pdivmod-rel-pCons:
  assumes rel: pdivmod-rel x y q r
  assumes y: y ≠ 0
  assumes b: b = coeff (pCons a r) (degree y) / coeff y (degree y)
  shows pdivmod-rel (pCons a x) y (pCons b q) (pCons a r − smult b y)
    (is pdivmod-rel ?x y ?q ?r)
proof −
  have x: x = q ∗ y + r and r: r = 0 ∨ degree r < degree y
    using assms unfolding pdivmod-rel-def by simp-all

  have 1: ?x = ?q ∗ y + ?r
    using b x by simp

  have 2: ?r = 0 ∨ degree ?r < degree y
  proof (rule eq-zero-or-degree-less)
    show degree ?r ≤ degree y
    proof (rule degree-diff-le)
      show degree (pCons a r) ≤ degree y
        using r by auto
      show degree (smult b y) ≤ degree y
        by (rule degree-smult-le)
    qed
  next
    show coeff ?r (degree y) = 0
      using ‹y ≠ 0› unfolding b by simp
  qed

  from 1 2 show ?thesis
    unfolding pdivmod-rel-def
    using ‹y ≠ 0› by simp
```

qed

lemma *pdivmod-rel-exists*: ∃ *q r*. *pdivmod-rel x y q r*
apply (*cases y* = *0*)
apply (*fast intro*!: *pdivmod-rel-by-0*)
apply (*induct x*)
apply (*fast intro*!: *pdivmod-rel-0*)
apply (*fast intro*!: *pdivmod-rel-pCons*)
done

lemma *pdivmod-rel-unique*:
  assumes *1*: *pdivmod-rel x y q1 r1*
  assumes *2*: *pdivmod-rel x y q2 r2*
  shows *q1* = *q2* ∧ *r1* = *r2*
proof (*cases y* = *0*)
  assume *y* = *0* **with** *assms* **show** *?thesis*
    by (*simp add*: *pdivmod-rel-def*)
next
  assume [*simp*]: *y* ≠ *0*
  from *1* **have** *q1*: *x* = *q1* ∗ *y* + *r1* **and** *r1*: *r1* = *0* ∨ *degree r1* < *degree y*
    unfolding *pdivmod-rel-def* **by** *simp-all*
  from *2* **have** *q2*: *x* = *q2* ∗ *y* + *r2* **and** *r2*: *r2* = *0* ∨ *degree r2* < *degree y*
    unfolding *pdivmod-rel-def* **by** *simp-all*
  from *q1 q2* **have** *q3*: (*q1* − *q2*) ∗ *y* = *r2* − *r1*
    by (*simp add*: *algebra-simps*)
  from *r1 r2* **have** *r3*: (*r2* − *r1*) = *0* ∨ *degree* (*r2* − *r1*) < *degree y*
    by (*auto intro*: *degree-diff-less*)

  show *q1* = *q2* ∧ *r1* = *r2*
  proof (*rule ccontr*)
    assume ¬ (*q1* = *q2* ∧ *r1* = *r2*)
    with *q3* **have** *q1* ≠ *q2* **and** *r1* ≠ *r2* **by** *auto*
    with *r3* **have** *degree* (*r2* − *r1*) < *degree y* **by** *simp*
    also have *degree y* ≤ *degree* (*q1* − *q2*) + *degree y* **by** *simp*
    also have ... = *degree* ((*q1* − *q2*) ∗ *y*)
      using ‹*q1* ≠ *q2*› **by** (*simp add*: *degree-mult-eq*)
    also have ... = *degree* (*r2* − *r1*)
      using *q3* **by** *simp*
    finally have *degree* (*r2* − *r1*) < *degree* (*r2* − *r1*) **.**
    then show *False* **by** *simp*
  qed
qed

lemma *pdivmod-rel-0-iff*: *pdivmod-rel 0 y q r* ⟷ *q* = *0* ∧ *r* = *0*
by (*auto dest*: *pdivmod-rel-unique intro*: *pdivmod-rel-0*)

lemma *pdivmod-rel-by-0-iff*: *pdivmod-rel x 0 q r* ⟷ *q* = *0* ∧ *r* = *x*
by (*auto dest*: *pdivmod-rel-unique intro*: *pdivmod-rel-by-0*)

lemmas *pdivmod-rel-unique-div* = *pdivmod-rel-unique* [*THEN conjunct1*]

lemmas *pdivmod-rel-unique-mod* = *pdivmod-rel-unique* [*THEN conjunct2*]

instantiation *poly* :: (*field*) *ring-div*
begin

definition *divide-poly* **where**
  div-poly-def: *x div y* = (*THE q*. ∃ *r*. *pdivmod-rel x y q r*)

definition *mod-poly* **where**
  x mod y = (*THE r*. ∃ *q*. *pdivmod-rel x y q r*)

lemma *div-poly-eq*:
  pdivmod-rel x y q r ⟹ *x div y* = *q*
unfolding *div-poly-def*
by (*fast elim*: *pdivmod-rel-unique-div*)

lemma *mod-poly-eq*:
  pdivmod-rel x y q r ⟹ *x mod y* = *r*
unfolding *mod-poly-def*
by (*fast elim*: *pdivmod-rel-unique-mod*)

lemma *pdivmod-rel*:
  pdivmod-rel x y (*x div y*) (*x mod y*)
proof −
  from *pdivmod-rel-exists*
    obtain *q r* **where** *pdivmod-rel x y q r* **by** *fast*
  thus *?thesis*
    by (*simp add*: *div-poly-eq mod-poly-eq*)
qed

instance
proof
  fix *x y* :: *'a poly*
  show *x div y* ∗ *y* + *x mod y* = *x*
    using *pdivmod-rel* [*of x y*]
    by (*simp add*: *pdivmod-rel-def*)
next
  fix *x* :: *'a poly*
  have *pdivmod-rel x 0 0 x*
    by (*rule pdivmod-rel-by-0*)
  thus *x div 0* = *0*
    by (*rule div-poly-eq*)
next
  fix *y* :: *'a poly*
  have *pdivmod-rel 0 y 0 0*
    by (*rule pdivmod-rel-0*)
  thus *0 div y* = *0*

by (*rule div-poly-eq*)
next
fix *x y z* :: *'a poly*
assume $y \neq 0$
hence *pdivmod-rel* (*x* + *z* ∗ *y*) *y* (*z* + *x div y*) (*x mod y*)
using *pdivmod-rel* [*of x y*]
by (*simp add*: *pdivmod-rel-def distrib-right*)
thus (*x* + *z* ∗ *y*) *div y* = *z* + *x div y*
by (*rule div-poly-eq*)
next
fix *x y z* :: *'a poly*
assume $x \neq 0$
show (*x* ∗ *y*) *div* (*x* ∗ *z*) = *y div z*
proof (*cases* $y \neq 0 \wedge z \neq 0$)
have ⋀*x*::*'a poly*. *pdivmod-rel x 0 0 x*
by (*rule pdivmod-rel-by-0*)
then have [*simp*]: ⋀*x*::*'a poly*. *x div 0* = *0*
by (*rule div-poly-eq*)
have ⋀*x*::*'a poly*. *pdivmod-rel 0 x 0 0*
by (*rule pdivmod-rel-0*)
then have [*simp*]: ⋀*x*::*'a poly*. *0 div x* = *0*
by (*rule div-poly-eq*)
case *False* **then show** *?thesis* **by** *auto*
next
case *True* **then have** $y \neq 0$ **and** $z \neq 0$ **by** *auto*
with ‹$x \neq 0$›
have ⋀*q r*. *pdivmod-rel y z q r* ⟹ *pdivmod-rel* (*x* ∗ *y*) (*x* ∗ *z*) *q* (*x* ∗ *r*)
by (*auto simp add*: *pdivmod-rel-def algebra-simps*)
(*rule classical*, *simp add*: *degree-mult-eq*)
moreover from *pdivmod-rel* **have** *pdivmod-rel y z* (*y div z*) (*y mod z*) **.**
ultimately have *pdivmod-rel* (*x* ∗ *y*) (*x* ∗ *z*) (*y div z*) (*x* ∗ (*y mod z*)) **.**
then show *?thesis* **by** (*simp add*: *div-poly-eq*)
qed
qed

end

lemma *is-unit-monom-0*:
fixes *a* :: *'a*::*field*
assumes $a \neq 0$
shows *is-unit* (*monom a 0*)
proof
from *assms* **show** *1* = *monom a 0* ∗ *monom* (*inverse a*) *0*
by (*simp add*: *mult-monom*)
qed

lemma *is-unit-triv*:
fixes *a* :: *'a*::*field*
assumes $a \neq 0$

shows *is-unit* [:*a*:]
using *assms* **by** (*simp add*: *is-unit-monom-0 monom-0* [*symmetric*])

lemma *is-unit-iff-degree*:
assumes $p \neq 0$
shows *is-unit p* ⟷ *degree p* = *0* (**is** *?P* ⟷ *?Q*)
proof
assume *?Q*
then obtain *a* **where** *p* = [:*a*:] **by** (*rule degree-eq-zeroE*)
with *assms* **show** *?P* **by** (*simp add*: *is-unit-triv*)
next
assume *?P*
then obtain *q* **where** $q \neq 0$ $p * q = 1$ **..**
then have *degree* (*p* ∗ *q*) = *degree 1*
by *simp*
with ‹$p \neq 0$› ‹$q \neq 0$› **have** *degree p* + *degree q* = *0*
by (*simp add*: *degree-mult-eq*)
then show *?Q* **by** *simp*
qed

lemma *is-unit-pCons-iff*:
is-unit (*pCons a p*) ⟷ *p* = *0* ∧ $a \neq 0$ (**is** *?P* ⟷ *?Q*)
by (*cases p* = *0*) (*auto simp add*: *is-unit-triv is-unit-iff-degree*)

lemma *is-unit-monom-trival*:
fixes *p* :: *'a*::*field poly*
assumes *is-unit p*
shows *monom* (*coeff p* (*degree p*)) *0* = *p*
using *assms* **by** (*cases p*) (*simp-all add*: *monom-0 is-unit-pCons-iff*)

lemma *is-unit-polyE*:
assumes *is-unit p*
obtains *a* **where** *p* = *monom a 0* **and** $a \neq 0$
proof −
obtain *a q* **where** *p* = *pCons a q* **by** (*cases p*)
with *assms* **have** *p* = [:*a*:] **and** $a \neq 0$
by (*simp-all add*: *is-unit-pCons-iff*)
with *that* **show** *thesis* **by** (*simp add*: *monom-0*)
qed

instantiation *poly* :: (*field*) *normalization-semidom*
begin

definition *normalize-poly* :: *'a poly* ⇒ *'a poly*
where *normalize-poly p* = *smult* (*inverse* (*coeff p* (*degree p*))) *p*

definition *unit-factor-poly* :: *'a poly* ⇒ *'a poly*
where *unit-factor-poly p* = *monom* (*coeff p* (*degree p*)) *0*

instance
proof
  fix *p* :: *'a poly*
  show *unit-factor p ∗ normalize p = p*
    by (*cases p = 0*)
      (*simp-all add*: *normalize-poly-def unit-factor-poly-def*,
      simp only: *mult-smult-left* [*symmetric*] *smult-monom*, *simp*)
next
  show *normalize 0* = (*0*::*'a poly*)
    by (*simp add*: *normalize-poly-def*)
next
  show *unit-factor 0* = (*0*::*'a poly*)
    by (*simp add*: *unit-factor-poly-def*)
next
  fix *p* :: *'a poly*
  assume *is-unit p*
  then obtain *a* **where** *p = monom a 0* **and** *a ≠ 0*
    by (*rule is-unit-polyE*)
  then show *normalize p = 1*
    by (*auto simp add*: *normalize-poly-def smult-monom degree-monom-eq*)
next
  fix *p q* :: *'a poly*
  assume *q ≠ 0*
  from ‹*q ≠ 0*› **have** *is-unit* (*monom* (*coeff q* (*degree q*)) *0*)
    by (*auto intro*: *is-unit-monom-0*)
  then show *is-unit* (*unit-factor q*)
    by (*simp add*: *unit-factor-poly-def*)
next
  fix *p q* :: *'a poly*
  have *monom* (*coeff* (*p ∗ q*) (*degree* (*p ∗ q*))) *0* =
    monom (*coeff p* (*degree p*)) *0 ∗ monom* (*coeff q* (*degree q*)) *0*
    by (*simp add*: *monom-0 coeff-degree-mult*)
  then show *unit-factor* (*p ∗ q*) =
    unit-factor p ∗ unit-factor q
    by (*simp add*: *unit-factor-poly-def*)
qed

end

lemma *unit-factor-monom* [*simp*]:
  unit-factor (*monom a n*) =
    (*if a = 0 then 0 else monom a 0*)
  by (*simp add*: *unit-factor-poly-def degree-monom-eq*)

lemma *unit-factor-pCons* [*simp*]:
  unit-factor (*pCons a p*) =
    (*if p = 0 then monom a 0 else unit-factor p*)
  by (*simp add*: *unit-factor-poly-def*)

lemma *normalize-monom* [*simp*]:
normalize (*monom a n*) =
(*if a* = *0 then 0 else monom 1 n*)
by (*simp add*: *normalize-poly-def degree-monom-eq smult-monom*)

lemma *degree-mod-less*:
$y \neq 0 \Longrightarrow x \; mod \; y = 0 \lor degree \; (x \; mod \; y) < degree \; y$
using *pdivmod-rel* [*of x y*]
unfolding *pdivmod-rel-def* **by** *simp*

lemma *div-poly-less*: *degree x* < *degree y* ⟹ *x div y* = *0*
proof −
assume *degree x* < *degree y*
hence *pdivmod-rel x y 0 x*
by (*simp add*: *pdivmod-rel-def*)
thus *x div y* = *0* **by** (*rule div-poly-eq*)
qed

lemma *mod-poly-less*: *degree x* < *degree y* ⟹ *x mod y* = *x*
proof −
assume *degree x* < *degree y*
hence *pdivmod-rel x y 0 x*
by (*simp add*: *pdivmod-rel-def*)
thus *x mod y* = *x* **by** (*rule mod-poly-eq*)
qed

lemma *pdivmod-rel-smult-left*:
pdivmod-rel x y q r
⟹ *pdivmod-rel* (*smult a x*) *y* (*smult a q*) (*smult a r*)
unfolding *pdivmod-rel-def* **by** (*simp add*: *smult-add-right*)

lemma *div-smult-left*: (*smult a x*) *div y* = *smult a* (*x div y*)
by (*rule div-poly-eq*, *rule pdivmod-rel-smult-left*, *rule pdivmod-rel*)

lemma *mod-smult-left*: (*smult a x*) *mod y* = *smult a* (*x mod y*)
by (*rule mod-poly-eq*, *rule pdivmod-rel-smult-left*, *rule pdivmod-rel*)

lemma *poly-div-minus-left* [*simp*]:
fixes *x y* :: *'a::field poly*
shows (− *x*) *div y* = − (*x div y*)
using *div-smult-left* [*of* − *1::'a*] **by** *simp*

lemma *poly-mod-minus-left* [*simp*]:
fixes *x y* :: *'a::field poly*
shows (− *x*) *mod y* = − (*x mod y*)
using *mod-smult-left* [*of* − *1::'a*] **by** *simp*

lemma *pdivmod-rel-add-left*:
assumes *pdivmod-rel x y q r*

assumes *pdivmod-rel* x' y q' r'
shows *pdivmod-rel* $(x + x')$ y $(q + q')$ $(r + r')$
using *assms* **unfolding** *pdivmod-rel-def*
by (*auto simp add*: *algebra-simps degree-add-less*)

lemma *poly-div-add-left*:
fixes *x y z* :: *'a::field poly*
shows $(x + y)$ *div* $z = x$ *div* $z + y$ *div* z
using *pdivmod-rel-add-left* [*OF pdivmod-rel pdivmod-rel*]
by (*rule div-poly-eq*)

lemma *poly-mod-add-left*:
fixes *x y z* :: *'a::field poly*
shows $(x + y)$ *mod* $z = x$ *mod* $z + y$ *mod* z
using *pdivmod-rel-add-left* [*OF pdivmod-rel pdivmod-rel*]
by (*rule mod-poly-eq*)

lemma *poly-div-diff-left*:
fixes *x y z* :: *'a::field poly*
shows $(x - y)$ *div* $z = x$ *div* $z - y$ *div* z
by (*simp only*: *diff-conv-add-uminus poly-div-add-left poly-div-minus-left*)

lemma *poly-mod-diff-left*:
fixes *x y z* :: *'a::field poly*
shows $(x - y)$ *mod* $z = x$ *mod* $z - y$ *mod* z
by (*simp only*: *diff-conv-add-uminus poly-mod-add-left poly-mod-minus-left*)

lemma *pdivmod-rel-smult-right*:
⟦$a \neq 0$; *pdivmod-rel x y q r*⟧
⟹ *pdivmod-rel x* (*smult a y*) (*smult* (*inverse a*) *q*) *r*
unfolding *pdivmod-rel-def* **by** *simp*

lemma *div-smult-right*:
$a \neq 0 \Longrightarrow x$ *div* (*smult a y*) = *smult* (*inverse a*) (x *div* y)
by (*rule div-poly-eq*, *erule pdivmod-rel-smult-right*, *rule pdivmod-rel*)

lemma *mod-smult-right*: $a \neq 0 \Longrightarrow x$ *mod* (*smult a y*) = x *mod* y
by (*rule mod-poly-eq*, *erule pdivmod-rel-smult-right*, *rule pdivmod-rel*)

lemma *poly-div-minus-right* [*simp*]:
fixes *x y* :: *'a::field poly*
shows x *div* $(- y) = - (x$ *div* $y)$
using *div-smult-right* [*of* $-$ *1::'a*] **by** (*simp add*: *nonzero-inverse-minus-eq*)

lemma *poly-mod-minus-right* [*simp*]:
fixes *x y* :: *'a::field poly*
shows x *mod* $(- y) = x$ *mod* y
using *mod-smult-right* [*of* $-$ *1::'a*] **by** *simp*

lemma *pdivmod-rel-mult*:
⟦*pdivmod-rel x y q r*; *pdivmod-rel q z q′ r′*⟧
⟹ *pdivmod-rel x* (*y* ∗ *z*) *q′* (*y* ∗ *r′* + *r*)
apply (*cases z* = *0*, *simp add*: *pdivmod-rel-def*)
apply (*cases y* = *0*, *simp add*: *pdivmod-rel-by-0-iff pdivmod-rel-0-iff*)
apply (*cases r* = *0*)
apply (*cases r′* = *0*)
apply (*simp add*: *pdivmod-rel-def*)
apply (*simp add*: *pdivmod-rel-def field-simps degree-mult-eq*)
apply (*cases r′* = *0*)
apply (*simp add*: *pdivmod-rel-def degree-mult-eq*)
apply (*simp add*: *pdivmod-rel-def field-simps*)
apply (*simp add*: *degree-mult-eq degree-add-less*)
done

lemma *poly-div-mult-right*:
fixes *x y z* :: *′a*::*field poly*
shows *x div* (*y* ∗ *z*) = (*x div y*) *div z*
by (*rule div-poly-eq*, *rule pdivmod-rel-mult*, (*rule pdivmod-rel*)+)

lemma *poly-mod-mult-right*:
fixes *x y z* :: *′a*::*field poly*
shows *x mod* (*y* ∗ *z*) = *y* ∗ (*x div y mod z*) + *x mod y*
by (*rule mod-poly-eq*, *rule pdivmod-rel-mult*, (*rule pdivmod-rel*)+)

lemma *mod-pCons*:
fixes *a* **and** *x*
assumes *y*: *y* ≠ *0*
defines *b*: *b* ≡ *coeff* (*pCons a* (*x mod y*)) (*degree y*) / *coeff y* (*degree y*)
shows (*pCons a x*) *mod y* = (*pCons a* (*x mod y*) − *smult b y*)
unfolding *b*
apply (*rule mod-poly-eq*)
apply (*rule pdivmod-rel-pCons* [*OF pdivmod-rel y refl*])
done

definition *pdivmod* :: *′a*::*field poly* ⇒ *′a poly* ⇒ *′a poly* × *′a poly*
where
pdivmod p q = (*p div q*, *p mod q*)

lemma *div-poly-code* [*code*]:
p div q = *fst* (*pdivmod p q*)
by (*simp add*: *pdivmod-def*)

lemma *mod-poly-code* [*code*]:
p mod q = *snd* (*pdivmod p q*)
by (*simp add*: *pdivmod-def*)

lemma *pdivmod-0*:
pdivmod 0 q = (*0*, *0*)

by (*simp add*: *pdivmod-def*)

lemma *pdivmod-pCons*:
pdivmod (*pCons a p*) *q* =
(*if q* = *0 then* (*0*, *pCons a p*) *else*
(*let* (*s*, *r*) = *pdivmod p q*;
b = *coeff* (*pCons a r*) (*degree q*) / *coeff q* (*degree q*)
in (*pCons b s*, *pCons a r* − *smult b q*)))
apply (*simp add*: *pdivmod-def Let-def*, *safe*)
apply (*rule div-poly-eq*)
apply (*erule pdivmod-rel-pCons* [*OF pdivmod-rel - refl*])
apply (*rule mod-poly-eq*)
apply (*erule pdivmod-rel-pCons* [*OF pdivmod-rel - refl*])
done

lemma *pdivmod-fold-coeffs* [*code*]:
pdivmod p q = (*if q* = *0 then* (*0*, *p*)
else fold-coeffs (λ*a* (*s*, *r*).
let b = *coeff* (*pCons a r*) (*degree q*) / *coeff q* (*degree q*)
in (*pCons b s*, *pCons a r* − *smult b q*)
) *p* (*0*, *0*))
apply (*cases q* = *0*)
apply (*simp add*: *pdivmod-def*)
apply (*rule sym*)
apply (*induct p*)
apply (*simp-all add*: *pdivmod-0 pdivmod-pCons*)
apply (*case-tac a* = *0* ∧ *p* = *0*)
apply (*auto simp add*: *pdivmod-def*)
done

41.20 Order of polynomial roots

definition *order* :: *'a*::*idom* ⇒ *'a poly* ⇒ *nat*
where
order a p = (*LEAST n*. ¬ [:−*a*, *1*:] ^ *Suc n dvd p*)

lemma *coeff-linear-power*:
fixes *a* :: *'a*::*comm-semiring-1*
shows *coeff* ([:*a*, *1*:] ^ *n*) *n* = *1*
apply (*induct n*, *simp-all*)
apply (*subst coeff-eq-0*)
apply (*auto intro*: *le-less-trans degree-power-le*)
done

lemma *degree-linear-power*:
fixes *a* :: *'a*::*comm-semiring-1*
shows *degree* ([:*a*, *1*:] ^ *n*) = *n*
apply (*rule order-antisym*)
apply (*rule ord-le-eq-trans* [*OF degree-power-le*], *simp*)

apply (*rule le-degree*, *simp add*: *coeff-linear-power*)
done

lemma *order-1*: [:−*a*, *1*:] ^ *order a p dvd p*
apply (*cases p* = *0*, *simp*)
apply (*cases order a p*, *simp*)
apply (*subgoal-tac nat* < (*LEAST n*. ¬ [:−*a*, *1*:] ^ *Suc n dvd p*))
apply (*drule not-less-Least*, *simp*)
apply (*fold order-def* , *simp*)
done

lemma *order-2*: *p* ≠ *0* ⟹ ¬ [:−*a*, *1*:] ^ *Suc* (*order a p*) *dvd p*
unfolding *order-def*
apply (*rule LeastI-ex*)
apply (*rule-tac x*=*degree p* **in** *exI*)
apply (*rule notI*)
apply (*drule* (*1*) *dvd-imp-degree-le*)
apply (*simp only*: *degree-linear-power*)
done

lemma *order*:
  p ≠ *0* ⟹ [:−*a*, *1*:] ^ *order a p dvd p* ∧ ¬ [:−*a*, *1*:] ^ *Suc* (*order a p*) *dvd p*
by (*rule conjI* [*OF order-1 order-2*])

lemma *order-degree*:
  assumes *p*: *p* ≠ *0*
  shows *order a p* ≤ *degree p*
proof −
  have *order a p* = *degree* ([:−*a*, *1*:] ^ *order a p*)
    by (*simp only*: *degree-linear-power*)
  also have ... ≤ *degree p*
    using *order-1 p* **by** (*rule dvd-imp-degree-le*)
  finally show *?thesis* **.**
qed

lemma *order-root*: *poly p a* = *0* ⟷ *p* = *0* ∨ *order a p* ≠ *0*
apply (*cases p* = *0*, *simp-all*)
apply (*rule iffI*)
apply (*metis order-2 not-gr0 poly-eq-0-iff-dvd power-0 power-Suc-0 power-one-right*)
unfolding *poly-eq-0-iff-dvd*
apply (*metis dvd-power dvd-trans order-1*)
done

lemma *order-0I*: *poly p a* ≠ *0* ⟹ *order a p* = *0*
  by (*subst* (*asm*) *order-root*) *auto*

41.21 Additional induction rules on polynomials

An induction rule for induction over the roots of a polynomial with a certain property. (e.g. all positive roots)

lemma *poly-root-induct* [*case-names 0 no-roots root*]:
  fixes p :: $'a$ :: *idom poly*
  assumes $Q\ 0$
  assumes $\bigwedge p.\ (\bigwedge a.\ P\ a \Longrightarrow poly\ p\ a \neq 0) \Longrightarrow Q\ p$
  assumes $\bigwedge a\ p.\ P\ a \Longrightarrow Q\ p \Longrightarrow Q\ ([:a,\ -1:] * p)$
  shows $Q\ p$
proof (*induction degree p arbitrary*: *p rule*: *less-induct*)
  case (*less p*)
  show *?case*
  proof (*cases* $p = 0$)
    assume *nz*: $p \neq 0$
    show *?case*
    proof (*cases* $\exists\, a.\ P\ a \wedge poly\ p\ a = 0$)
      case *False*
      thus *?thesis* **by** (*intro assms*(*2*)) *blast*
    next
      case *True*
      then obtain *a* **where** *a*: $P\ a\ poly\ p\ a = 0$
        by *blast*
      hence $-[:-a,\ 1:]$ *dvd p*
        by (*subst minus-dvd-iff*) (*simp add*: *poly-eq-0-iff-dvd*)
      then obtain *q* **where** *q*: $p = [:a,\ -1:] * q$ **by** (*elim dvdE*) *simp*
      with *nz* **have** *q-nz*: $q \neq 0$ **by** *auto*
      have *degree p* = *Suc* (*degree q*)
        by (*subst q*, *subst degree-mult-eq*) (*simp-all add*: *q-nz*)
      hence $Q\ q$ **by** (*intro less*) *simp*
      from *a*(*1*) **and** *this* **have** $Q\ ([:a,\ -1:] * q)$
        by (*rule assms*(*3*))
      with *q* **show** *?thesis* **by** *simp*
    qed
  qed (*simp add*: *assms*(*1*))
qed

lemma *dropWhile-replicate-append*:
  dropWhile (*op=* *a*) (*replicate n a* @ *ys*) = *dropWhile* (*op=* *a*) *ys*
  by (*induction n*) *simp-all*

lemma *Poly-append-replicate-0*: *Poly* (*xs* @ *replicate n 0*) = *Poly xs*
  by (*subst coeffs-eq-iff*) (*simp-all add*: *strip-while-def dropWhile-replicate-append*)

An induction rule for simultaneous induction over two polynomials, prepending one coefficient in each step.

lemma *poly-induct2* [*case-names 0 pCons*]:
  assumes $P\ 0\ 0\ \bigwedge a\ p\ b\ q.\ P\ p\ q \Longrightarrow P\ (pCons\ a\ p)\ (pCons\ b\ q)$
  shows $P\ p\ q$

proof −
  def *n* ≡ *max* (*length* (*coeffs p*)) (*length* (*coeffs q*))
  def *xs* ≡ *coeffs p* @ (*replicate* (*n* − *length* (*coeffs p*)) *0*)
  def *ys* ≡ *coeffs q* @ (*replicate* (*n* − *length* (*coeffs q*)) *0*)
  have *length xs* = *length ys*
    by (*simp add*: *xs-def ys-def n-def*)
  hence *P* (*Poly xs*) (*Poly ys*)
    by (*induction rule*: *list-induct2*) (*simp-all add*: *assms*)
  also have *Poly xs* = *p*
    by (*simp add*: *xs-def Poly-append-replicate-0*)
  also have *Poly ys* = *q*
    by (*simp add*: *ys-def Poly-append-replicate-0*)
  finally show *?thesis* **.**
qed

41.22 Composition of polynomials

definition *pcompose* :: *'a::comm-semiring-0 poly* ⇒ *'a poly* ⇒ *'a poly*
where
  pcompose p q = *fold-coeffs* (λ*a c*. [:*a*:] + *q* ∗ *c*) *p 0*

notation *pcompose* (**infixl** $\circ_p$ *71*)

lemma *pcompose-0* [*simp*]:
  pcompose 0 q = *0*
  by (*simp add*: *pcompose-def*)

lemma *pcompose-pCons*:
  pcompose (*pCons a p*) *q* = [:*a*:] + *q* ∗ *pcompose p q*
  by (*cases p* = *0* ∧ *a* = *0*) (*auto simp add*: *pcompose-def*)

lemma *pcompose-1*:
  fixes *p* :: *'a* :: *comm-semiring-1 poly*
  shows *pcompose 1 p* = *1*
  unfolding *one-poly-def* **by** (*auto simp*: *pcompose-pCons*)

lemma *poly-pcompose*:
  poly (*pcompose p q*) *x* = *poly p* (*poly q x*)
  by (*induct p*) (*simp-all add*: *pcompose-pCons*)

lemma *degree-pcompose-le*:
  degree (*pcompose p q*) ≤ *degree p* ∗ *degree q*
apply (*induct p*, *simp*)
apply (*simp add*: *pcompose-pCons*, *clarify*)
apply (*rule degree-add-le*, *simp*)
apply (*rule order-trans* [*OF degree-mult-le*], *simp*)
done

lemma *pcompose-add*:

fixes *p q r* :: *'a* :: {*comm-semiring-0*, *ab-semigroup-add*} *poly*
shows *pcompose* (*p* + *q*) *r* = *pcompose p r* + *pcompose q r*
proof (*induction p q rule*: *poly-induct2*)
case (*pCons a p b q*)
have *pcompose* (*pCons a p* + *pCons b q*) *r* =
[:*a* + *b*:] + *r* ∗ *pcompose p r* + *r* ∗ *pcompose q r*
by (*simp-all add*: *pcompose-pCons pCons.IH algebra-simps*)
also have [:*a* + *b*:] = [:*a*:] + [:*b*:] **by** *simp*
also have ... + *r* ∗ *pcompose p r* + *r* ∗ *pcompose q r* =
pcompose (*pCons a p*) *r* + *pcompose* (*pCons b q*) *r*
by (*simp only*: *pcompose-pCons add-ac*)
finally show *?case* **.**
qed *simp*

lemma *pcompose-uminus*:
fixes *p r* :: *'a* :: *comm-ring poly*
shows *pcompose* (−*p*) *r* = −*pcompose p r*
by (*induction p*) (*simp-all add*: *pcompose-pCons*)

lemma *pcompose-diff*:
fixes *p q r* :: *'a* :: *comm-ring poly*
shows *pcompose* (*p* − *q*) *r* = *pcompose p r* − *pcompose q r*
using *pcompose-add*[*of p* −*q*] **by** (*simp add*: *pcompose-uminus*)

lemma *pcompose-smult*:
fixes *p r* :: *'a* :: *comm-semiring-0 poly*
shows *pcompose* (*smult a p*) *r* = *smult a* (*pcompose p r*)
by (*induction p*)
(*simp-all add*: *pcompose-pCons pcompose-add smult-add-right*)

lemma *pcompose-mult*:
fixes *p q r* :: *'a* :: *comm-semiring-0 poly*
shows *pcompose* (*p* ∗ *q*) *r* = *pcompose p r* ∗ *pcompose q r*
by (*induction p arbitrary*: *q*)
(*simp-all add*: *pcompose-add pcompose-smult pcompose-pCons algebra-simps*)

lemma *pcompose-assoc*:
pcompose p (*pcompose q r* :: *'a* :: *comm-semiring-0 poly*) =
pcompose (*pcompose p q*) *r*
by (*induction p arbitrary*: *q*)
(*simp-all add*: *pcompose-pCons pcompose-add pcompose-mult*)

lemma *pcompose-idR*[*simp*]:
fixes *p* :: *'a* :: *comm-semiring-1 poly*
shows *pcompose p* [: *0*, *1* :] = *p*
by (*induct p*; *simp add*: *pcompose-pCons*)

lemma *degree-mult-eq-0*:
fixes *p q*:: *'a* :: *semidom poly*
shows *degree* (*p*∗*q*) = *0* ⟷ *p*=*0* ∨ *q*=*0* ∨ (*p*≠*0* ∧ *q*≠*0* ∧ *degree p* =*0* ∧ *degree q* =*0*)
by (*auto simp add*:*degree-mult-eq*)

lemma *pcompose-const*[*simp*]:*pcompose* [:*a*:] *q* = [:*a*:] **by** (*subst pcompose-pCons*,*simp*)

lemma *pcompose-0'*: *pcompose p 0* = [:*coeff p 0*:]
by (*induct p*) (*auto simp add*:*pcompose-pCons*)

lemma *degree-pcompose*:
fixes *p q*:: *'a*::*semidom poly*
shows *degree* (*pcompose p q*) = *degree p* ∗ *degree q*
proof (*induct p*)
case *0*
thus *?case* **by** *auto*
next
case (*pCons a p*)
have *degree* (*q* ∗ *pcompose p q*) = *0* ⟹ *?case*
proof (*cases p*=*0*)
case *True*
thus *?thesis* **by** *auto*
next
case *False* **assume** *degree* (*q* ∗ *pcompose p q*) = *0*
hence *degree q*=*0* ∨ *pcompose p q*=*0* **by** (*auto simp add*: *degree-mult-eq-0*)
moreover have ⟦*pcompose p q*=*0*;*degree q*≠*0*⟧ ⟹ *False* **using** *pCons.hyps*(*2*) ‹*p*≠*0*›
proof −
assume *pcompose p q*=*0 degree q*≠*0*
hence *degree p*=*0* **using** *pCons.hyps*(*2*) **by** *auto*
then obtain *a1* **where** *p*=[:*a1*:]
by (*metis degree-pCons-eq-if old.nat.distinct*(*2*) *pCons-cases*)
thus *False* **using** ‹*pcompose p q*=*0*› ‹*p*≠*0*› **by** *auto*
qed
ultimately have *degree* (*pCons a p*) ∗ *degree q*=*0* **by** *auto*
moreover have *degree* (*pcompose* (*pCons a p*) *q*) = *0*
proof −
have *0* = *max* (*degree* [:*a*:]) (*degree* (*q*∗*pcompose p q*))
using ‹*degree* (*q* ∗ *pcompose p q*) = *0*› **by** *simp*
also have ... ≥ *degree* ([:*a*:] + *q* ∗ *pcompose p q*)
by (*rule degree-add-le-max*)
finally show *?thesis* **by** (*auto simp add*:*pcompose-pCons*)
qed
ultimately show *?thesis* **by** *simp*
qed
moreover have *degree* (*q* ∗ *pcompose p q*)>*0* ⟹ *?case*

proof –
assume *asm*:*0* < *degree* (*q* ∗ *pcompose p q*)
hence *p*≠*0 q*≠*0 pcompose p q*≠*0* **by** *auto*
have *degree* (*pcompose* (*pCons a p*) *q*) = *degree* (*q* ∗ *pcompose p q*)
unfolding *pcompose-pCons*
using *degree-add-eq-right*[*of* [:*a*:]] *asm* **by** *auto*
thus *?thesis*
using *pCons.hyps*(*2*) *degree-mult-eq*[*OF* ‹*q*≠*0*› ‹*pcompose p q*≠*0*›] **by** *auto*
qed
ultimately show *?case* **by** *blast*
qed

lemma *pcompose-eq-0*:
fixes *p q*:: *'a* :: *semidom poly*
assumes *pcompose p q* = *0 degree q* > *0*
shows *p* = *0*
proof –
have *degree p*=*0* **using** *assms degree-pcompose*[*of p q*] **by** *auto*
then obtain *a* **where** *p*=[:*a*:]
by (*metis degree-pCons-eq-if gr0-conv-Suc neq0-conv pCons-cases*)
hence *a*=*0* **using** *assms*(*1*) **by** *auto*
thus *?thesis* **using** ‹*p*=[:*a*:]› **by** *simp*
qed

41.23 Leading coefficient

definition *lead-coeff* :: *'a*::*zero poly* ⇒ *'a* **where**
lead-coeff p= *coeff p* (*degree p*)

lemma *lead-coeff-pCons*[*simp*]:
p≠*0* ⟹*lead-coeff* (*pCons a p*) = *lead-coeff p*
p=*0* ⟹ *lead-coeff* (*pCons a p*) = *a*
unfolding *lead-coeff-def* **by** *auto*

lemma *lead-coeff-0*[*simp*]:*lead-coeff 0* =*0*
unfolding *lead-coeff-def* **by** *auto*

lemma *lead-coeff-mult*:
fixes *p q*::*'a* ::*idom poly*
shows *lead-coeff* (*p* ∗ *q*) = *lead-coeff p* ∗ *lead-coeff q*
by (*unfold lead-coeff-def*,*cases p*=*0* ∨ *q*=*0*,*auto simp add*:*coeff-mult-degree-sum degree-mult-eq*)

lemma *lead-coeff-add-le*:
assumes *degree p* < *degree q*
shows *lead-coeff* (*p*+*q*) = *lead-coeff q*
using *assms* **unfolding** *lead-coeff-def*
by (*metis coeff-add coeff-eq-0 monoid-add-class.add.left-neutral degree-add-eq-right*)

lemma *lead-coeff-minus*:
  lead-coeff (−*p*) = − *lead-coeff p*
by (*metis coeff-minus degree-minus lead-coeff-def*)

lemma *lead-coeff-comp*:
  fixes *p q*:: *'a*::*idom poly*
  assumes *degree q* > *0*
  shows *lead-coeff* (*pcompose p q*) = *lead-coeff p* ∗ *lead-coeff q* ˆ (*degree p*)
proof (*induct p*)
  case *0*
  thus *?case* **unfolding** *lead-coeff-def* **by** *auto*
next
  case (*pCons a p*)
  have *degree* (*q* ∗ *pcompose p q*) = *0* ⟹ *?case*
    proof −
      assume *degree* (*q* ∗ *pcompose p q*) = *0*
     hence *pcompose p q* = *0* **by** (*metis assms degree-0 degree-mult-eq-0 neq0-conv*)
      hence *p*=*0* **using** *pcompose-eq-0*[*OF* - ‹*degree q* > *0*›] **by** *simp*
      thus *?thesis* **by** *auto*
    qed
  moreover have *degree* (*q* ∗ *pcompose p q*) > *0* ⟹ *?case*
    proof −
      assume *degree* (*q* ∗ *pcompose p q*) > *0*
      hence *lead-coeff* (*pcompose* (*pCons a p*) *q*) =*lead-coeff* (*q* ∗ *pcompose p q*)
        by (*auto simp add*:*pcompose-pCons lead-coeff-add-le*)
      also have ... = *lead-coeff q* ∗ (*lead-coeff p* ∗ *lead-coeff q* ˆ *degree p*)
        using *pCons.hyps*(*2*) *lead-coeff-mult*[*of q pcompose p q*] **by** *simp*
      also have ... = *lead-coeff p* ∗ *lead-coeff q* ˆ (*degree p* + *1*)
        by *auto*
      finally show *?thesis* **by** *auto*
    qed
  ultimately show *?case* **by** *blast*
qed

lemma *lead-coeff-smult*:
  lead-coeff (*smult c p* :: *'a* :: *idom poly*) = *c* ∗ *lead-coeff p*
proof −
  have *smult c p* = [:*c*:] ∗ *p* **by** *simp*
  also have *lead-coeff* . . . = *c* ∗ *lead-coeff p*
    by (*subst lead-coeff-mult*) *simp-all*
  finally show *?thesis* **.**
qed

lemma *lead-coeff-1* [*simp*]: *lead-coeff 1* = *1*
  by (*simp add*: *lead-coeff-def*)

lemma *lead-coeff-of-nat* [*simp*]:
  lead-coeff (*of-nat n*) = (*of-nat n* :: *'a* :: {*comm-semiring-1*,*semiring-char-0*})

by (*induction n*) (*simp-all add*: *lead-coeff-def of-nat-poly*)

lemma *lead-coeff-numeral* [*simp*]:
lead-coeff (*numeral n*) = *numeral n*
unfolding *lead-coeff-def*
by (*subst of-nat-numeral* [*symmetric*], *subst of-nat-poly*) *simp*

lemma *lead-coeff-power*:
lead-coeff (*p* ^ *n* :: *'a* :: *idom poly*) = *lead-coeff p* ^ *n*
by (*induction n*) (*simp-all add*: *lead-coeff-mult*)

lemma *lead-coeff-nonzero*: *p* ≠ *0* ⟹ *lead-coeff p* ≠ *0*
by (*simp add*: *lead-coeff-def*)

41.24 Derivatives of univariate polynomials

function *pderiv* :: (*'a* :: *semidom*) *poly* ⇒ *'a poly*
where
[*simp del*]: *pderiv* (*pCons a p*) = (*if p* = *0 then 0 else p* + *pCons 0* (*pderiv p*))
by (*auto intro*: *pCons-cases*)

termination *pderiv*
by (*relation measure degree*) *simp-all*

lemma *pderiv-0* [*simp*]:
pderiv 0 = *0*
using *pderiv.simps* [*of 0 0*] **by** *simp*

lemma *pderiv-pCons*:
pderiv (*pCons a p*) = *p* + *pCons 0* (*pderiv p*)
by (*simp add*: *pderiv.simps*)

lemma *pderiv-1* [*simp*]: *pderiv 1* = *0*
unfolding *one-poly-def* **by** (*simp add*: *pderiv-pCons*)

lemma *pderiv-of-nat* [*simp*]: *pderiv* (*of-nat n*) = *0*
and *pderiv-numeral* [*simp*]: *pderiv* (*numeral m*) = *0*
by (*simp-all add*: *of-nat-poly numeral-poly pderiv-pCons*)

lemma *coeff-pderiv*: *coeff* (*pderiv p*) *n* = *of-nat* (*Suc n*) ∗ *coeff p* (*Suc n*)
by (*induct p arbitrary*: *n*)
(*auto simp add*: *pderiv-pCons coeff-pCons algebra-simps split*: *nat.split*)

fun *pderiv-coeffs-code* :: (*'a* :: *semidom*) ⇒ *'a list* ⇒ *'a list* **where**
pderiv-coeffs-code f (*x* # *xs*) = *cCons* (*f* ∗ *x*) (*pderiv-coeffs-code* (*f*+*1*) *xs*)
| *pderiv-coeffs-code f* [] = []

definition *pderiv-coeffs* :: (*'a* :: *semidom*) *list* ⇒ *'a list* **where**
pderiv-coeffs xs = *pderiv-coeffs-code 1* (*tl xs*)

lemma *pderiv-coeffs-code*:
  nth-default 0 (*pderiv-coeffs-code f xs*) *n* = (*f* + *of-nat n*) ∗ (*nth-default 0 xs n*)
proof (*induct xs arbitrary*: *f n*)
  case (*Cons x xs f n*)
  show *?case*
  proof (*cases n*)
    case *0*
    thus *?thesis* **by** (*cases pderiv-coeffs-code* (*f* + *1*) *xs* = [] ∧ *f* ∗ *x* = *0*, *auto simp*: *cCons-def*)
  next
    case (*Suc m*) **note** *n* = *this*
    show *?thesis*
    proof (*cases pderiv-coeffs-code* (*f* + *1*) *xs* = [] ∧ *f* ∗ *x* = *0*)
      case *False*
      hence *nth-default 0* (*pderiv-coeffs-code f* (*x* # *xs*)) *n* =
              nth-default 0 (*pderiv-coeffs-code* (*f* + *1*) *xs*) *m*
        by (*auto simp*: *cCons-def n*)
      also have ... = (*f* + *of-nat n*) ∗ (*nth-default 0 xs m*)
        unfolding *Cons* **by** (*simp add*: *n add-ac*)
      finally show *?thesis* **by** (*simp add*: *n*)
    next
      case *True*
      {
        fix *g*
        have *pderiv-coeffs-code g xs* = [] ⟹ *g* + *of-nat m* = *0* ∨ *nth-default 0 xs m* = *0*
        proof (*induct xs arbitrary*: *g m*)
          case (*Cons x xs g*)
          from *Cons*(*2*) **have** *empty*: *pderiv-coeffs-code* (*g* + *1*) *xs* = []
                            and *g*: (*g* = *0* ∨ *x* = *0*)
            by (*auto simp*: *cCons-def split*: *if-splits*)
          note *IH* = *Cons*(*1*)[*OF empty*]
          from *IH*[*of m*] *IH*[*of m* − *1*] *g*
          show *?case* **by** (*cases m*, *auto simp*: *field-simps*)
        qed *simp*
      } **note** *empty* = *this*
      from *True* **have** *nth-default 0* (*pderiv-coeffs-code f* (*x* # *xs*)) *n* = *0*
        by (*auto simp*: *cCons-def n*)
      moreover have (*f* + *of-nat n*) ∗ *nth-default 0* (*x* # *xs*) *n* = *0* **using** *True*
        by (*simp add*: *n*, *insert empty*[*of f*+*1*], *auto simp*: *field-simps*)
      ultimately show *?thesis* **by** *simp*
    qed
  qed
qed *simp*

lemma *map-upt-Suc*: *map f* [*0* ..< *Suc n*] = *f 0* # *map* (λ *i*. *f* (*Suc i*)) [*0* ..< *n*]
  by (*induct n arbitrary*: *f*, *auto*)

```
lemma coeffs-pderiv-code [code abstract]:
  coeffs (pderiv p) = pderiv-coeffs (coeffs p) unfolding pderiv-coeffs-def
proof (rule coeffs-eqI, unfold pderiv-coeffs-code coeff-pderiv, goal-cases)
  case (1 n)
  have id: coeff p (Suc n) = nth-default 0 (map (λi. coeff p (Suc i)) [0..<degree
p]) n
    by (cases n < degree p, auto simp: nth-default-def coeff-eq-0)
  show ?case unfolding coeffs-def map-upt-Suc by (auto simp: id)
next
  case 2
  obtain n xs where id: tl (coeffs p) = xs (1 :: 'a) = n by auto
  from 2 show ?case
    unfolding id by (induct xs arbitrary: n, auto simp: cCons-def)
qed

context
  assumes SORT-CONSTRAINT('a::{semidom, semiring-char-0})
begin

lemma pderiv-eq-0-iff:
  pderiv (p :: 'a poly) = 0 ⟷ degree p = 0
  apply (rule iffI)
  apply (cases p, simp)
  apply (simp add: poly-eq-iff coeff-pderiv del: of-nat-Suc)
  apply (simp add: poly-eq-iff coeff-pderiv coeff-eq-0)
  done

lemma degree-pderiv: degree (pderiv (p :: 'a poly)) = degree p − 1
  apply (rule order-antisym [OF degree-le])
  apply (simp add: coeff-pderiv coeff-eq-0)
  apply (cases degree p, simp)
  apply (rule le-degree)
  apply (simp add: coeff-pderiv del: of-nat-Suc)
  apply (metis degree-0 leading-coeff-0-iff nat.distinct(1))
  done

lemma not-dvd-pderiv:
  assumes degree (p :: 'a poly) ≠ 0
  shows ¬ p dvd pderiv p
proof
  assume dvd: p dvd pderiv p
  then obtain q where p: pderiv p = p * q unfolding dvd-def by auto
  from dvd have le: degree p ≤ degree (pderiv p)
    by (simp add: assms dvd-imp-degree-le pderiv-eq-0-iff)
  from this[unfolded degree-pderiv] assms show False by auto
qed

lemma dvd-pderiv-iff [simp]: (p :: 'a poly) dvd pderiv p ⟷ degree p = 0
```

using *not-dvd-pderiv*[*of p*] **by** (*auto simp*: *pderiv-eq-0-iff* [*symmetric*])

end

lemma *pderiv-singleton* [*simp*]: *pderiv* [:*a*:] = *0*
by (*simp add*: *pderiv-pCons*)

lemma *pderiv-add*: *pderiv* (*p* + *q*) = *pderiv p* + *pderiv q*
by (*rule poly-eqI*, *simp add*: *coeff-pderiv algebra-simps*)

lemma *pderiv-minus*: *pderiv* (− *p* :: *'a* :: *idom poly*) = − *pderiv p*
by (*rule poly-eqI*, *simp add*: *coeff-pderiv algebra-simps*)

lemma *pderiv-diff*: *pderiv* (*p* − *q*) = *pderiv p* − *pderiv q*
by (*rule poly-eqI*, *simp add*: *coeff-pderiv algebra-simps*)

lemma *pderiv-smult*: *pderiv* (*smult a p*) = *smult a* (*pderiv p*)
by (*rule poly-eqI*, *simp add*: *coeff-pderiv algebra-simps*)

lemma *pderiv-mult*: *pderiv* (*p* ∗ *q*) = *p* ∗ *pderiv q* + *q* ∗ *pderiv p*
by (*induct p*) (*auto simp*: *pderiv-add pderiv-smult pderiv-pCons algebra-simps*)

lemma *pderiv-power-Suc*:
pderiv (*p* ^ *Suc n*) = *smult* (*of-nat* (*Suc n*)) (*p* ^ *n*) ∗ *pderiv p*
apply (*induct n*)
apply *simp*
apply (*subst power-Suc*)
apply (*subst pderiv-mult*)
apply (*erule ssubst*)
apply (*simp only*: *of-nat-Suc smult-add-left smult-1-left*)
apply (*simp add*: *algebra-simps*)
done

lemma *pderiv-setprod*: *pderiv* (*setprod f* (*as*)) =
(∑ *a* ∈ *as*. *setprod f* (*as* − {*a*}) ∗ *pderiv* (*f a*))
proof (*induct as rule*: *infinite-finite-induct*)
case (*insert a as*)
hence *id*: *setprod f* (*insert a as*) = *f a* ∗ *setprod f as*
⋀ *g*. *setsum g* (*insert a as*) = *g a* + *setsum g as*
insert a as − {*a*} = *as*
by *auto*
{
fix *b*
assume *b* ∈ *as*
hence *id2*: *insert a as* − {*b*} = *insert a* (*as* − {*b*}) **using** ‹*a* ∉ *as*› **by** *auto*
have *setprod f* (*insert a as* − {*b*}) = *f a* ∗ *setprod f* (*as* − {*b*})
unfolding *id2*
by (*subst setprod.insert*, *insert insert*, *auto*)
} **note** *id2* = *this*

show *?case*
unfolding *id pderiv-mult insert*(*3*) *setsum-right-distrib*
by (*auto simp add*: *ac-simps id2 intro*!: *setsum.cong*)
qed *auto*

lemma *DERIV-pow2*: *DERIV* (%*x*. *x* ^ *Suc n*) *x* :> *real* (*Suc n*) ∗ (*x* ^ *n*)
by (*rule DERIV-cong*, *rule DERIV-pow*, *simp*)
declare *DERIV-pow2* [*simp*] *DERIV-pow* [*simp*]

lemma *DERIV-add-const*: *DERIV f x* :> *D* ==> *DERIV* (%*x*. *a* + *f x* :: *'a*::*real-normed-field*) *x* :> *D*
by (*rule DERIV-cong*, *rule DERIV-add*, *auto*)

lemma *poly-DERIV* [*simp*]: *DERIV* (%*x*. *poly p x*) *x* :> *poly* (*pderiv p*) *x*
by (*induct p*, *auto intro*!: *derivative-eq-intros simp add*: *pderiv-pCons*)

lemma *continuous-on-poly* [*continuous-intros*]:
fixes *p* :: *'a* :: {*real-normed-field*} *poly*
assumes *continuous-on A f*
shows *continuous-on A* (λ*x*. *poly p* (*f x*))
proof −
have *continuous-on A* (λ*x*. (∑ *i*≤*degree p*. (*f x*) ^ *i* ∗ *coeff p i*))
by (*intro continuous-intros assms*)
also have ... = (λ*x*. *poly p* (*f x*)) **by** (*intro ext*) (*simp add*: *poly-altdef mult-ac*)
finally show *?thesis* **.**
qed

Consequences of the derivative theorem above

lemma *poly-differentiable*[*simp*]: (%*x*. *poly p x*) *differentiable* (*at x*::*real filter*)
apply (*simp add*: *real-differentiable-def*)
apply (*blast intro*: *poly-DERIV*)
done

lemma *poly-isCont*[*simp*]: *isCont* (%*x*. *poly p x*) (*x*::*real*)
by (*rule poly-DERIV* [*THEN DERIV-isCont*])

lemma *poly-IVT-pos*: [| *a* < *b*; *poly p* (*a*::*real*) < *0*; *0* < *poly p b* |]
==> ∃ *x*. *a* < *x* & *x* < *b* & (*poly p x* = *0*)
using *IVT-objl* [*of poly p a 0 b*]
by (*auto simp add*: *order-le-less*)

lemma *poly-IVT-neg*: [| (*a*::*real*) < *b*; *0* < *poly p a*; *poly p b* < *0* |]
==> ∃ *x*. *a* < *x* & *x* < *b* & (*poly p x* = *0*)
by (*insert poly-IVT-pos* [**where** *p* = − *p*]) *simp*

lemma *poly-IVT*:
fixes *p*::*real poly*
assumes *a*<*b* **and** *poly p a* ∗ *poly p b* < *0*
shows ∃ *x*>*a*. *x* < *b* ∧ *poly p x* = *0*

by (*metis assms*(*1*) *assms*(*2*) *less-not-sym mult-less-0-iff poly-IVT-neg poly-IVT-pos*)

lemma *poly-MVT*: (*a*::*real*) < *b* ==>
  $\exists x.\ a < x\ \&\ x < b\ \&\ (poly\ p\ b - poly\ p\ a = (b - a) * poly\ (pderiv\ p)\ x)$
using *MVT* [*of a b poly p*]
apply *auto*
apply (*rule-tac x* = *z* **in** *exI*)
apply (*auto simp add*: *mult-left-cancel poly-DERIV* [*THEN DERIV-unique*])
done

lemma *poly-MVT'*:
  assumes $\{min\ a\ b..max\ a\ b\} \subseteq A$
  shows $\exists x{\in}A.\ poly\ p\ b - poly\ p\ a = (b - a) * poly\ (pderiv\ p)\ (x{::}real)$
proof (*cases a b rule*: *linorder-cases*)
  case *less*
  from *poly-MVT*[*OF less*, *of p*] **guess** *x* **by** (*elim exE conjE*)
  thus *?thesis* **by** (*intro bexI*[*of - x*]) (*auto intro*!: *subsetD*[*OF assms*])

next
  case *greater*
  from *poly-MVT*[*OF greater*, *of p*] **guess** *x* **by** (*elim exE conjE*)
  thus *?thesis* **by** (*intro bexI*[*of - x*]) (*auto simp*: *algebra-simps intro*!: *subsetD*[*OF assms*])
qed (*insert assms*, *auto*)

lemma *poly-pinfty-gt-lc*:
  fixes *p*:: *real poly*
  assumes *lead-coeff p* > *0*
  shows $\exists\ n.\ \forall\ x \geq n.\ poly\ p\ x \geq lead\text{-}coeff\ p$ **using** *assms*
proof (*induct p*)
  case *0*
  thus *?case* **by** *auto*
next
  case (*pCons a p*)
  have $[\![a\neq 0;p=0]\!] \Longrightarrow ?case$ **by** *auto*
  moreover have $p\neq 0 \Longrightarrow ?case$
    proof –
      assume $p\neq 0$
      then obtain *n1* **where** *gte-lcoeff*:$\forall x{\geq}n1.\ lead\text{-}coeff\ p \leq poly\ p\ x$ **using** *that pCons* **by** *auto*
      have *gt-0*:*lead-coeff p* >*0* **using** *pCons*(*3*) ‹$p\neq 0$› **by** *auto*
      def $n \equiv max\ n1\ (1+ |a|/(lead\text{-}coeff\ p))$
      show *?thesis*
        proof (*rule-tac x*=*n* **in** *exI*,*rule*,*rule*)
          fix *x* **assume** $n \leq x$
          hence $lead\text{-}coeff\ p \leq poly\ p\ x$
            using *gte-lcoeff* **unfolding** *n-def* **by** *auto*
          hence $|a|/(lead\text{-}coeff\ p) \geq |a|/(poly\ p\ x)$ **and** $poly\ p\ x>0$ **using** *gt-0*
            by (*intro frac-le*,*auto*)

```
        hence x≥1+ |a|/(poly p x) using ‹n≤x›[unfolded n-def] by auto
        thus lead-coeff (pCons a p) ≤ poly (pCons a p) x
          using ‹lead-coeff p ≤ poly p x› ‹poly p x>0› ‹p≠0›
          by (auto simp add:field-simps)
      qed
    qed
  ultimately show ?case by fastforce
qed
```

41.25 Algebraic numbers

Algebraic numbers can be defined in two equivalent ways: all real numbers that are roots of rational polynomials or of integer polynomials. The Algebraic-Numbers AFP entry uses the rational definition, but we need the integer definition.

The equivalence is obvious since any rational polynomial can be multiplied with the LCM of its coefficients, yielding an integer polynomial with the same roots.

41.26 Algebraic numbers

```
definition algebraic :: 'a :: field-char-0 ⇒ bool where
  algebraic x ⟷ (∃ p. (∀ i. coeff p i ∈ ℤ) ∧ p ≠ 0 ∧ poly p x = 0)

lemma algebraicI:
  assumes ⋀i. coeff p i ∈ ℤ p ≠ 0 poly p x = 0
  shows   algebraic x
  using assms unfolding algebraic-def by blast

lemma algebraicE:
  assumes algebraic x
  obtains p where ⋀i. coeff p i ∈ ℤ p ≠ 0 poly p x = 0
  using assms unfolding algebraic-def by blast

lemma quotient-of-denom-pos': snd (quotient-of x) > 0
  using quotient-of-denom-pos[OF surjective-pairing] .

lemma of-int-div-in-Ints:
  b dvd a ⟹ of-int a div of-int b ∈ (ℤ :: 'a :: ring-div set)
proof (cases of-int b = (0 :: 'a))
  assume b dvd a of-int b ≠ (0::'a)
  then obtain c where a = b ∗ c by (elim dvdE)
  with ‹of-int b ≠ (0::'a)› show ?thesis by simp
qed auto

lemma of-int-divide-in-Ints:
  b dvd a ⟹ of-int a / of-int b ∈ (ℤ :: 'a :: field set)
proof (cases of-int b = (0 :: 'a))
```

assume *b dvd a of-int b* ≠ (*0*::′*a*)
then obtain *c* **where** *a* = *b* ∗ *c* **by** (*elim dvdE*)
with ‹*of-int b* ≠ (*0*::′*a*)› **show** *?thesis* **by** *simp*
qed *auto*

lemma *algebraic-altdef*:
fixes *p* :: ′*a* :: *field-char-0 poly*
shows *algebraic x* ⟷ (∃ *p*. (∀ *i*. *coeff p i* ∈ ℚ) ∧ *p* ≠ *0* ∧ *poly p x* = *0*)
proof *safe*
fix *p* **assume** *rat*: ∀ *i*. *coeff p i* ∈ ℚ **and** *root*: *poly p x* = *0* **and** *nz*: *p* ≠ *0*
def *cs* ≡ *coeffs p*
from *rat* **have** ∀ *c*∈*range* (*coeff p*). ∃ *c*′. *c* = *of-rat c*′ **unfolding** *Rats-def* **by** *blast*
then obtain *f* **where** *f*: ⋀*i*. *coeff p i* = *of-rat* (*f* (*coeff p i*))
by (*subst* (*asm*) *bchoice-iff*) *blast*
def *cs*′ ≡ *map* (*quotient-of* ∘ *f*) (*coeffs p*)
def *d* ≡ *Lcm* (*set* (*map snd cs*′))
def *p*′ ≡ *smult* (*of-int d*) *p*

have ∀ *n*. *coeff p*′ *n* ∈ ℤ
proof
fix *n* :: *nat*
show *coeff p*′ *n* ∈ ℤ
proof (*cases n* ≤ *degree p*)
case *True*
def *c* ≡ *coeff p n*
def *a* ≡ *fst* (*quotient-of* (*f* (*coeff p n*))) **and** *b* ≡ *snd* (*quotient-of* (*f* (*coeff p n*)))
have *b-pos*: *b* > *0* **unfolding** *b-def* **using** *quotient-of-denom-pos*′ **by** *simp*
have *coeff p*′ *n* = *of-int d* ∗ *coeff p n* **by** (*simp add*: *p*′*-def*)
also have *coeff p n* = *of-rat* (*of-int a* / *of-int b*) **unfolding** *a-def b-def*
by (*subst quotient-of-div* [*of f* (*coeff p n*), *symmetric*])
(*simp-all add*: *f* [*symmetric*])
also have *of-int d* ∗ ... = *of-rat* (*of-int* (*a*∗*d*) / *of-int b*)
by (*simp add*: *of-rat-mult of-rat-divide*)
also from *nz True* **have** *b* ∈ *snd* ‘ *set cs*′ **unfolding** *cs*′*-def*
by (*force simp*: *o-def b-def coeffs-def simp del*: *upt-Suc*)
hence *b dvd* (*a* ∗ *d*) **unfolding** *d-def* **by** *simp*
hence *of-int* (*a* ∗ *d*) / *of-int b* ∈ (ℤ :: *rat set*)
by (*rule of-int-divide-in-Ints*)
hence *of-rat* (*of-int* (*a* ∗ *d*) / *of-int b*) ∈ ℤ **by** (*elim Ints-cases*) *auto*
finally show *?thesis* **.**
qed (*auto simp*: *p*′*-def not-le coeff-eq-0*)
qed

moreover have *set* (*map snd cs*′) ⊆ {*0*<..}
unfolding *cs*′*-def* **using** *quotient-of-denom-pos*′ **by** (*auto simp*: *coeffs-def simp del*: *upt-Suc*)
hence *d* ≠ *0* **unfolding** *d-def* **by** (*induction cs*′) *simp-all*

with *nz* **have** $p' \neq 0$ **by** (*simp add*: *p'-def*)
moreover from *root* **have** *poly* p' $x = 0$ **by** (*simp add*: *p'-def*)
ultimately show *algebraic x* **unfolding** *algebraic-def* **by** *blast*
next

assume *algebraic x*
then obtain *p* **where** *p*: $\bigwedge i.$ *coeff p i* $\in \mathbb{Z}$ *poly p x* $= 0$ $p \neq 0$
by (*force simp*: *algebraic-def*)
moreover have *coeff p i* $\in \mathbb{Z} \Longrightarrow$ *coeff p i* $\in \mathbb{Q}$ **for** *i* **by** (*elim Ints-cases*) *simp*
ultimately show $(\exists p. (\forall i.$ *coeff p i* $\in \mathbb{Q}) \wedge p \neq 0 \wedge$ *poly p x* $= 0)$ **by** *auto*
qed

Lemmas for Derivatives

lemma *order-unique-lemma*:
fixes *p* :: *'a::idom poly*
assumes [:−*a*, *1*:] ˆ *n dvd p* ¬ [:−*a*, *1*:] ˆ *Suc n dvd p*
shows *n* = *order a p*
unfolding *Polynomial.order-def*
apply (*rule Least-equality* [*symmetric*])
apply (*fact assms*)
apply (*rule classical*)
apply (*erule notE*)
unfolding *not-less-eq-eq*
using *assms*(*1*) **apply** (*rule power-le-dvd*)
apply *assumption*
done

lemma *lemma-order-pderiv1*:
pderiv ([:− *a*, *1*:] ˆ *Suc n* ∗ *q*) = [:− *a*, *1*:] ˆ *Suc n* ∗ *pderiv q* +
smult (*of-nat* (*Suc n*)) (*q* ∗ [:− *a*, *1*:] ˆ *n*)
apply (*simp only*: *pderiv-mult pderiv-power-Suc*)
apply (*simp del*: *power-Suc of-nat-Suc add*: *pderiv-pCons*)
done

lemma *lemma-order-pderiv*:
fixes *p* :: *'a* :: *field-char-0 poly*
assumes *n*: $0 < n$
and *pd*: *pderiv p* $\neq 0$
and *pe*: *p* = [:− *a*, *1*:] ˆ *n* ∗ *q*
and *nd*: ~ [:− *a*, *1*:] *dvd q*
shows *n* = *Suc* (*order a* (*pderiv p*))
using *n*
proof −
have *pderiv* ([:− *a*, *1*:] ˆ *n* ∗ *q*) $\neq 0$
using *assms* **by** *auto*
obtain *n'* **where** *n* = *Suc n'* $0 <$ *Suc n'* *pderiv* ([:− *a*, *1*:] ˆ *Suc n'* ∗ *q*) $\neq 0$
using *assms* **by** (*cases n*) *auto*
have ∗: !!*k l*. *k dvd k* ∗ *pderiv q* + *smult* (*of-nat* (*Suc n'*)) *l* $\Longrightarrow$ *k dvd l*
by (*auto simp del*: *of-nat-Suc simp*: *dvd-add-right-iff dvd-smult-iff*)

have $n' = order\ a\ (pderiv\ ([:-\ a,\ 1:]\ \hat{}\ Suc\ n' * q))$
proof (*rule order-unique-lemma*)
show $[:-\ a,\ 1:]\ \hat{}\ n'\ dvd\ pderiv\ ([:-\ a,\ 1:]\ \hat{}\ Suc\ n' * q)$
apply (*subst lemma-order-pderiv1*)
apply (*rule dvd-add*)
apply (*metis dvdI dvd-mult2 power-Suc2*)
apply (*metis dvd-smult dvd-triv-right*)
done
next
show $\neg\ [:-\ a,\ 1:]\ \hat{}\ Suc\ n'\ dvd\ pderiv\ ([:-\ a,\ 1:]\ \hat{}\ Suc\ n' * q)$
apply (*subst lemma-order-pderiv1*)
by (*metis* ∗ *nd dvd-mult-cancel-right power-not-zero pCons-eq-0-iff power-Suc zero-neq-one*)
qed
then show *?thesis*
by (*metis* ‹$n = Suc\ n'$› *pe*)
qed

lemma *order-decomp*:
assumes $p \neq 0$
shows $\exists\, q.\ p = [:-\ a,\ 1:]\ \hat{}\ order\ a\ p * q \wedge \neg\ [:-\ a,\ 1:]\ dvd\ q$
proof −
from *assms* **have** A: $[:-\ a,\ 1:]\ \hat{}\ order\ a\ p\ dvd\ p$
and B: $\neg\ [:-\ a,\ 1:]\ \hat{}\ Suc\ (order\ a\ p)\ dvd\ p$ **by** (*auto dest*: *order*)
from A **obtain** q **where** C: $p = [:-\ a,\ 1:]\ \hat{}\ order\ a\ p * q$ **..**
with B **have** $\neg\ [:-\ a,\ 1:]\ \hat{}\ Suc\ (order\ a\ p)\ dvd\ [:-\ a,\ 1:]\ \hat{}\ order\ a\ p * q$
by *simp*
then have $\neg\ [:-\ a,\ 1:]\ \hat{}\ order\ a\ p * [:-\ a,\ 1:]\ dvd\ [:-\ a,\ 1:]\ \hat{}\ order\ a\ p * q$
by *simp*
then have D: $\neg\ [:-\ a,\ 1:]\ dvd\ q$
using *idom-class.dvd-mult-cancel-left* [*of* $[:-\ a,\ 1:]\ \hat{}\ order\ a\ p\ [:-\ a,\ 1:]\ q$]
by *auto*
from $C\ D$ **show** *?thesis* **by** *blast*
qed

lemma *order-pderiv*:
$[\![pderiv\ p \neq 0;\ order\ a\ (p :: 'a :: field\text{-}char\text{-}0\ poly) \neq 0]\!] \Longrightarrow$
$(order\ a\ p = Suc\ (order\ a\ (pderiv\ p)))$
apply (*case-tac* $p = 0$, *simp*)
apply (*drule-tac* $a = a$ **and** $p = p$ **in** *order-decomp*)
using *neq0-conv*
apply (*blast intro*: *lemma-order-pderiv*)
done

lemma *order-mult*: $p * q \neq 0 \Longrightarrow order\ a\ (p * q) = order\ a\ p + order\ a\ q$
proof −
def $i \equiv order\ a\ p$
def $j \equiv order\ a\ q$
def $t \equiv [:-a,\ 1:]$

```
  have t-dvd-iff: ⋀u. t dvd u ⟷ poly u a = 0
    unfolding t-def by (simp add: dvd-iff-poly-eq-0)
  assume p * q ≠ 0
  then show order a (p * q) = i + j
    apply clarsimp
    apply (drule order [where a=a and p=p, folded i-def t-def])
    apply (drule order [where a=a and p=q, folded j-def t-def])
    apply clarify
    apply (erule dvdE)+
    apply (rule order-unique-lemma [symmetric], fold t-def)
    apply (simp-all add: power-add t-dvd-iff)
    done
qed

lemma order-smult:
  assumes c ≠ 0
  shows order x (smult c p) = order x p
proof (cases p = 0)
  case False
  have smult c p = [:c:] * p by simp
  also from assms False have order x ... = order x [:c:] + order x p
    by (subst order-mult) simp-all
  also from assms have order x [:c:] = 0 by (intro order-0I) auto
  finally show ?thesis by simp
qed simp


lemma order-1-eq-0 [simp]:order x 1 = 0
  by (metis order-root poly-1 zero-neq-one)

lemma order-power-n-n: order a ([:-a,1:]^n)=n
proof (induct n)
  case 0
  thus ?case by (metis order-root poly-1 power-0 zero-neq-one)
next
  case (Suc n)
  have order a ([:- a, 1:] ^ Suc n)=order a ([:- a, 1:] ^ n) + order a [:-a,1:]
   by (metis (no-types, hide-lams) One-nat-def add-Suc-right monoid-add-class.add.right-neutral

     one-neq-zero order-mult pCons-eq-0-iff power-add power-eq-0-iff power-one-right)
  moreover have order a [:-a,1:]=1 unfolding order-def
    proof (rule Least-equality,rule ccontr)
      assume ¬ ¬ [:- a, 1:] ^ Suc 1 dvd [:- a, 1:]
      hence [:- a, 1:] ^ Suc 1 dvd [:- a, 1:] by simp
      hence degree ([:- a, 1:] ^ Suc 1) ≤ degree ([:- a, 1:] )
        by (rule dvd-imp-degree-le,auto)
      thus False by auto
    next
      fix y assume asm:¬ [:- a, 1:] ^ Suc y dvd [:- a, 1:]
```

```
    show 1 ≤ y
      proof (rule ccontr)
        assume ¬ 1 ≤ y
        hence y=0 by auto
        hence [:− a, 1:] ^ Suc y dvd [:− a, 1:] by auto
        thus False using asm by auto
      qed
  qed
 ultimately show ?case using Suc by auto
qed
```

Now justify the standard squarefree decomposition, i.e. f / gcd(f,f').

```
lemma order-divides: [:−a, 1:] ^ n dvd p ⟷ p = 0 ∨ n ≤ order a p
apply (cases p = 0, auto)
apply (drule order-2 [where a=a and p=p])
apply (metis not-less-eq-eq power-le-dvd)
apply (erule power-le-dvd [OF order-1])
done

lemma poly-squarefree-decomp-order:
 assumes pderiv (p :: 'a :: field-char-0 poly) ≠ 0
 and p: p = q ∗ d
 and p': pderiv p = e ∗ d
 and d: d = r ∗ p + s ∗ pderiv p
 shows order a q = (if order a p = 0 then 0 else 1)
proof (rule classical)
 assume 1: order a q ≠ (if order a p = 0 then 0 else 1)
 from ‹pderiv p ≠ 0› have p ≠ 0 by auto
 with p have order a p = order a q + order a d
   by (simp add: order-mult)
 with 1 have order a p ≠ 0 by (auto split: if-splits)
 have order a (pderiv p) = order a e + order a d
   using ‹pderiv p ≠ 0› ‹pderiv p = e ∗ d› by (simp add: order-mult)
 have order a p = Suc (order a (pderiv p))
   using ‹pderiv p ≠ 0› ‹order a p ≠ 0› by (rule order-pderiv)
 have d ≠ 0 using ‹p ≠ 0› ‹p = q ∗ d› by simp
 have ([:−a, 1:] ^ (order a (pderiv p))) dvd d
   apply (simp add: d)
   apply (rule dvd-add)
   apply (rule dvd-mult)
   apply (simp add: order-divides ‹p ≠ 0›
       ‹order a p = Suc (order a (pderiv p))›)
   apply (rule dvd-mult)
   apply (simp add: order-divides)
   done
 then have order a (pderiv p) ≤ order a d
   using ‹d ≠ 0› by (simp add: order-divides)
 show ?thesis
   using ‹order a p = order a q + order a d›
```

using ⟨*order a* (*pderiv p*) = *order a e* + *order a d*⟩
using ⟨*order a p* = *Suc* (*order a* (*pderiv p*))⟩
using ⟨*order a* (*pderiv p*) ≤ *order a d*⟩
by *auto*
qed

lemma *poly-squarefree-decomp-order2*:
⟦*pderiv p* ≠ (*0* :: ′*a* :: *field-char-0 poly*);
p = *q* ∗ *d*;
pderiv p = *e* ∗ *d*;
d = *r* ∗ *p* + *s* ∗ *pderiv p*
⟧ ⟹ ∀ *a*. *order a q* = (*if order a p* = *0 then 0 else 1*)
by (*blast intro*: *poly-squarefree-decomp-order*)

lemma *order-pderiv2*:
⟦*pderiv p* ≠ *0*; *order a* (*p* :: ′*a* :: *field-char-0 poly*) ≠ *0*⟧
⟹ (*order a* (*pderiv p*) = *n*) = (*order a p* = *Suc n*)
by (*auto dest*: *order-pderiv*)

definition
rsquarefree :: ′*a*::*idom poly* => *bool* **where**
rsquarefree p = (*p* ≠ *0* & (∀ *a*. (*order a p* = *0*) | (*order a p* = *1*)))

lemma *pderiv-iszero*: *pderiv p* = *0* ⟹ ∃ *h*. *p* = [:*h* :: ′*a* :: {*semidom*,*semiring-char-0*}:]
apply (*simp add*: *pderiv-eq-0-iff*)
apply (*case-tac p*, *auto split*: *if-splits*)
done

lemma *rsquarefree-roots*:
fixes *p* :: ′*a* :: *field-char-0 poly*
shows *rsquarefree p* = (∀ *a*. ¬(*poly p a* = *0* ∧ *poly* (*pderiv p*) *a* = *0*))
apply (*simp add*: *rsquarefree-def*)
apply (*case-tac p* = *0*, *simp*, *simp*)
apply (*case-tac pderiv p* = *0*)
apply *simp*
apply (*drule pderiv-iszero*, *clarsimp*)
apply (*metis coeff-0 coeff-pCons-0 degree-pCons-0 le0 le-antisym order-degree*)
apply (*force simp add*: *order-root order-pderiv2*)
done

lemma *poly-squarefree-decomp*:
assumes *pderiv* (*p* :: ′*a* :: *field-char-0 poly*) ≠ *0*
and *p* = *q* ∗ *d*
and *pderiv p* = *e* ∗ *d*
and *d* = *r* ∗ *p* + *s* ∗ *pderiv p*
shows *rsquarefree q* & (∀ *a*. (*poly q a* = *0*) = (*poly p a* = *0*))
proof −
from ⟨*pderiv p* ≠ *0*⟩ **have** *p* ≠ *0* **by** *auto*
with ⟨*p* = *q* ∗ *d*⟩ **have** *q* ≠ *0* **by** *simp*

```
  have ∀ a. order a q = (if order a p = 0 then 0 else 1)
    using assms by (rule poly-squarefree-decomp-order2)
  with ‹p ≠ 0› ‹q ≠ 0› show ?thesis
    by (simp add: rsquarefree-def order-root)
qed


no-notation cCons (infixr ## 65)

end
```

42 Abstract euclidean algorithm

```
theory Euclidean-Algorithm
imports ~~/src/HOL/GCD ~~/src/HOL/Library/Polynomial
begin
```

A Euclidean semiring is a semiring upon which the Euclidean algorithm can be implemented. It must provide:

- division with remainder
- a size function such that *size* (*a mod b*) < *size b* for any $b \neq (0::'a)$

The existence of these functions makes it possible to derive gcd and lcm functions for any Euclidean semiring.

```
class euclidean-semiring = semiring-div + normalization-semidom +
  fixes euclidean-size :: 'a ⇒ nat
  assumes size-0 [simp]: euclidean-size 0 = 0
  assumes mod-size-less:
    b ≠ 0 ⟹ euclidean-size (a mod b) < euclidean-size b
  assumes size-mult-mono:
    b ≠ 0 ⟹ euclidean-size a ≤ euclidean-size (a ∗ b)
begin

lemma euclidean-division:
  fixes a :: 'a and b :: 'a
  assumes b ≠ 0
  obtains s and t where a = s ∗ b + t
    and euclidean-size t < euclidean-size b
proof −
  from div-mod-equality [of a b 0]
     have a = a div b ∗ b + a mod b by simp
  with that and assms show ?thesis by (auto simp add: mod-size-less)
qed

lemma dvd-euclidean-size-eq-imp-dvd:
 assumes a ≠ 0 and b-dvd-a: b dvd a and size-eq: euclidean-size a = euclidean-size
b
```

  shows *a dvd b*
proof (*rule ccontr*)
  assume ¬ *a dvd b*
  then have *b mod a* $\neq$ *0* **by** (*simp add*: *mod-eq-0-iff-dvd*)
  from *b-dvd-a* **have** *b-dvd-mod*: *b dvd b mod a* **by** (*simp add*: *dvd-mod-iff*)
  from *b-dvd-mod* **obtain** *c* **where** *b mod a* = *b* ∗ *c* **unfolding** *dvd-def* **by** *blast*
    with ‹*b mod a* $\neq$ *0*› **have** *c* $\neq$ *0* **by** *auto*
  with ‹*b mod a* = *b* ∗ *c*› **have** *euclidean-size* (*b mod a*) ≥ *euclidean-size b*
      using *size-mult-mono* **by** *force*
  moreover from ‹¬ *a dvd b*› **and** ‹*a* $\neq$ *0*›
  have *euclidean-size* (*b mod a*) < *euclidean-size a*
      using *mod-size-less* **by** *blast*
  ultimately show *False* **using** *size-eq* **by** *simp*
qed

function *gcd-eucl* :: ′*a* ⇒ ′*a* ⇒ ′*a*
where
  gcd-eucl a b = (*if b* = *0 then normalize a else gcd-eucl b* (*a mod b*))
  by *pat-completeness simp*
termination
  by (*relation measure* (*euclidean-size* ∘ *snd*)) (*simp-all add*: *mod-size-less*)

declare *gcd-eucl.simps* [*simp del*]

lemma *gcd-eucl-induct* [*case-names zero mod*]:
  assumes *H1*: ⋀*b*. *P b 0*
  and *H2*: ⋀*a b*. *b* $\neq$ *0* ⟹ *P b* (*a mod b*) ⟹ *P a b*
  shows *P a b*
proof (*induct a b rule*: *gcd-eucl.induct*)
  case (*1 a b*)
  show *?case*
  proof (*cases b* = *0*)
    case *True* **then show** *P a b* **by** *simp* (*rule H1*)
  next
    case *False*
    then have *P b* (*a mod b*)
      by (*rule 1.hyps*)
    with ‹*b* $\neq$ *0*› **show** *P a b*
      by (*blast intro*: *H2*)
  qed
qed

definition *lcm-eucl* :: ′*a* ⇒ ′*a* ⇒ ′*a*
where
  lcm-eucl a b = *normalize* (*a* ∗ *b*) *div gcd-eucl a b*

definition *Lcm-eucl* :: ′*a set* ⇒ ′*a* — Somewhat complicated definition of Lcm that has the advantage of working for infinite sets as well
where

Lcm-eucl A = (*if* ∃ *l*. *l* ≠ *0* ∧ (∀ *a*∈*A*. *a dvd l*) *then*
let l = *SOME l*. *l* ≠ *0* ∧ (∀ *a*∈*A*. *a dvd l*) ∧ *euclidean-size l* =
(*LEAST n*. ∃ *l*. *l* ≠ *0* ∧ (∀ *a*∈*A*. *a dvd l*) ∧ *euclidean-size l* = *n*)
in normalize l
else 0)

definition *Gcd-eucl* :: *'a set* ⇒ *'a*
where
Gcd-eucl A = *Lcm-eucl* {*d*. ∀ *a*∈*A*. *d dvd a*}

declare *Lcm-eucl-def Gcd-eucl-def* [*code del*]

lemma *gcd-eucl-0*:
gcd-eucl a 0 = *normalize a*
by (*simp add*: *gcd-eucl.simps* [*of a 0*])

lemma *gcd-eucl-0-left*:
gcd-eucl 0 a = *normalize a*
by (*simp-all add*: *gcd-eucl-0 gcd-eucl.simps* [*of 0 a*])

lemma *gcd-eucl-non-0*:
b ≠ *0* ⟹ *gcd-eucl a b* = *gcd-eucl b* (*a mod b*)
by (*simp add*: *gcd-eucl.simps* [*of a b*] *gcd-eucl.simps* [*of b 0*])

lemma *gcd-eucl-dvd1* [*iff*]: *gcd-eucl a b dvd a*
and *gcd-eucl-dvd2* [*iff*]: *gcd-eucl a b dvd b*
by (*induct a b rule*: *gcd-eucl-induct*)
(*simp-all add*: *gcd-eucl-0 gcd-eucl-non-0 dvd-mod-iff*)

lemma *normalize-gcd-eucl* [*simp*]:
normalize (*gcd-eucl a b*) = *gcd-eucl a b*
by (*induct a b rule*: *gcd-eucl-induct*) (*simp-all add*: *gcd-eucl-0 gcd-eucl-non-0*)

lemma *gcd-eucl-greatest*:
fixes *k a b* :: *'a*
shows *k dvd a* ⟹ *k dvd b* ⟹ *k dvd gcd-eucl a b*
proof (*induct a b rule*: *gcd-eucl-induct*)
case (*zero a*) **from** *zero*(*1*) **show** *?case* **by** (*rule dvd-trans*) (*simp add*: *gcd-eucl-0*)
next
case (*mod a b*)
then show *?case*
by (*simp add*: *gcd-eucl-non-0 dvd-mod-iff*)
qed

lemma *eq-gcd-euclI*:
fixes *gcd* :: *'a* ⇒ *'a* ⇒ *'a*
assumes ⋀*a b*. *gcd a b dvd a* ⋀*a b*. *gcd a b dvd b* ⋀*a b*. *normalize* (*gcd a b*) =
gcd a b
⋀*a b k*. *k dvd a* ⟹ *k dvd b* ⟹ *k dvd gcd a b*

shows *gcd = gcd-eucl*
by (*intro ext, rule associated-eqI*) (*simp-all add*: *gcd-eucl-greatest assms*)

lemma *gcd-eucl-zero* [*simp*]:
gcd-eucl a b = 0 ⟷ *a = 0* ∧ *b = 0*
by (*metis dvd-0-left dvd-refl gcd-eucl-dvd1 gcd-eucl-dvd2 gcd-eucl-greatest*)+

lemma *dvd-Lcm-eucl* [*simp*]: *a* ∈ *A* ⟹ *a dvd Lcm-eucl A*
and *Lcm-eucl-least*: (⋀*a. a* ∈ *A* ⟹ *a dvd b*) ⟹ *Lcm-eucl A dvd b*
and *unit-factor-Lcm-eucl* [*simp*]:
unit-factor (*Lcm-eucl A*) = (*if Lcm-eucl A = 0 then 0 else 1*)
proof −
have (∀ *a*∈*A. a dvd Lcm-eucl A*) ∧ (∀ *l′*. (∀ *a*∈*A. a dvd l′*) ⟶ *Lcm-eucl A dvd l′*) ∧
unit-factor (*Lcm-eucl A*) = (*if Lcm-eucl A = 0 then 0 else 1*) (**is** *?thesis*)
proof (*cases* ∃ *l. l* ≠ *0* ∧ (∀ *a*∈*A. a dvd l*))
case *False*
hence *Lcm-eucl A = 0* **by** (*auto simp*: *Lcm-eucl-def*)
with *False* **show** *?thesis* **by** *auto*
next
case *True*
then obtain l_0 **where** l_0*-props*: $l_0 \neq 0$ ∧ (∀ *a*∈*A. a dvd* l_0) **by** *blast*
def *n* ≡ *LEAST n.* ∃ *l. l* ≠ *0* ∧ (∀ *a*∈*A. a dvd l*) ∧ *euclidean-size l = n*
def *l* ≡ *SOME l. l* ≠ *0* ∧ (∀ *a*∈*A. a dvd l*) ∧ *euclidean-size l = n*
have ∃ *l. l* ≠ *0* ∧ (∀ *a*∈*A. a dvd l*) ∧ *euclidean-size l = n*
apply (*subst n-def*)
apply (*rule LeastI*[*of - euclidean-size* l_0])
apply (*rule exI*[*of -* l_0])
apply (*simp add*: l_0*-props*)
done
from *someI-ex*[*OF this*] **have** *l* ≠ *0* **and** ∀ *a*∈*A. a dvd l* **and** *euclidean-size l = n*
unfolding *l-def* **by** *simp-all*
{
fix *l′* **assume** ∀ *a*∈*A. a dvd l′*
with ‹∀ *a*∈*A. a dvd l*› **have** ∀ *a*∈*A. a dvd gcd-eucl l l′* **by** (*auto intro*: *gcd-eucl-greatest*)
moreover from ‹*l* ≠ *0*› **have** *gcd-eucl l l′* ≠ *0* **by** *simp*
ultimately have ∃ *b. b* ≠ *0* ∧ (∀ *a*∈*A. a dvd b*) ∧
euclidean-size b = euclidean-size (*gcd-eucl l l′*)
by (*intro exI*[*of - gcd-eucl l l′*], *auto*)
hence *euclidean-size* (*gcd-eucl l l′*) ≥ *n* **by** (*subst n-def*) (*rule Least-le*)
moreover have *euclidean-size* (*gcd-eucl l l′*) ≤ *n*
proof −
have *gcd-eucl l l′ dvd l* **by** *simp*
then obtain *a* **where** *l = gcd-eucl l l′ ∗ a* **unfolding** *dvd-def* **by** *blast*
with ‹*l* ≠ *0*› **have** *a* ≠ *0* **by** *auto*
hence *euclidean-size* (*gcd-eucl l l′*) ≤ *euclidean-size* (*gcd-eucl l l′ ∗ a*)

by (*rule size-mult-mono*)
also have *gcd-eucl l l′ ∗ a = l* **using** ‹*l = gcd-eucl l l′ ∗ a*› **..**
also note ‹*euclidean-size l = n*›
finally show *euclidean-size* (*gcd-eucl l l′*) ≤ *n* **.**
qed
ultimately have ∗: *euclidean-size l = euclidean-size* (*gcd-eucl l l′*)
by (*intro le-antisym*, *simp-all add*: ‹*euclidean-size l = n*›)
from ‹*l ≠ 0*› **have** *l dvd gcd-eucl l l′*
by (*rule dvd-euclidean-size-eq-imp-dvd*) (*auto simp add*: ∗)
hence *l dvd l′* **by** (*rule dvd-trans*[*OF - gcd-eucl-dvd2*])
}

with ‹(∀ *a*∈*A*. *a dvd l*)› **and** *unit-factor-is-unit*[*OF* ‹*l ≠ 0*›] **and** ‹*l ≠ 0*›
have (∀ *a*∈*A*. *a dvd normalize l*) ∧
(∀ *l′*. (∀ *a*∈*A*. *a dvd l′*) ⟶ *normalize l dvd l′*) ∧
unit-factor (*normalize l*) =
(*if normalize l = 0 then 0 else 1*)
by (*auto simp*: *unit-simps*)
also from *True* **have** *normalize l = Lcm-eucl A*
by (*simp add*: *Lcm-eucl-def Let-def n-def l-def*)
finally show *?thesis* **.**
qed
note *A = this*

{fix *a* **assume** *a ∈ A* **then show** *a dvd Lcm-eucl A* **using** *A* **by** *blast***}**
{fix *b* **assume** ⋀*a*. *a ∈ A* ⟹ *a dvd b* **then show** *Lcm-eucl A dvd b* **using** *A* **by** *blast***}**
from *A* **show** *unit-factor* (*Lcm-eucl A*) = (*if Lcm-eucl A = 0 then 0 else 1*) **by** *blast*
qed

lemma *normalize-Lcm-eucl* [*simp*]:
normalize (*Lcm-eucl A*) = *Lcm-eucl A*
proof (*cases Lcm-eucl A = 0*)
case *True* **then show** *?thesis* **by** *simp*
next
case *False*
have *unit-factor* (*Lcm-eucl A*) ∗ *normalize* (*Lcm-eucl A*) = *Lcm-eucl A*
by (*fact unit-factor-mult-normalize*)
with *False* **show** *?thesis* **by** *simp*
qed

lemma *eq-Lcm-euclI*:
fixes *lcm* :: ′*a set* ⇒ ′*a*
assumes ⋀*A a*. *a ∈ A* ⟹ *a dvd lcm A* **and** ⋀*A c*. (⋀*a*. *a ∈ A* ⟹ *a dvd c*) ⟹ *lcm A dvd c*
⋀*A*. *normalize* (*lcm A*) = *lcm A* **shows** *lcm = Lcm-eucl*
by (*intro ext*, *rule associated-eqI*) (*auto simp*: *assms intro*: *Lcm-eucl-least*)

end

class *euclidean-ring* = *euclidean-semiring* + *idom*
begin

subclass *ring-div* **..**

function *euclid-ext-aux* :: *'a* ⇒ - **where**
  euclid-ext-aux r' r s' s t' t = (
    if $r = 0$ *then let* $c = 1$ *div unit-factor* r' *in* ($s' * c$, $t' * c$, *normalize* r')
    else let $q = r'$ *div* r
      in *euclid-ext-aux* r (r' *mod* r) s ($s' - q * s$) t ($t' - q * t$))
by *auto*
termination by (*relation measure* (λ(-,*b*,-,-,-,-). *euclidean-size* *b*)) (*simp-all add*: *mod-size-less*)

declare *euclid-ext-aux.simps* [*simp del*]

lemma *euclid-ext-aux-correct*:
  assumes *gcd-eucl* r' r = *gcd-eucl* x y
  assumes $s' * x + t' * y = r'$
  assumes $s * x + t * y = r$
  shows *case euclid-ext-aux* r' r s' s t' t *of* (*a*,*b*,*c*) ⇒
    $a * x + b * y = c$ ∧ c = *gcd-eucl* x y (**is** *?P* (*euclid-ext-aux* r' r s' s t' t))
using *assms*
proof (*induction* r' r s' s t' t *rule*: *euclid-ext-aux.induct*)
  case (*1* r' r s' s t' t)
  show *?case*
  proof (*cases* $r = 0$)
    case *True*
    hence *euclid-ext-aux* r' r s' s t' t =
      (s' *div unit-factor* r', t' *div unit-factor* r', *normalize* r')
      by (*subst euclid-ext-aux.simps*) (*simp add*: *Let-def*)
    also have *?P* ...
    proof *safe*
      have s' *div unit-factor* $r' * x$ + t' *div unit-factor* $r' * y$ =
        ($s' * x + t' * y$) *div unit-factor* r'
        by (*cases* $r' = 0$) (*simp-all add*: *unit-div-commute*)
      also have $s' * x + t' * y = r'$ **by** *fact*
      also have ... *div unit-factor* r' = *normalize* r' **by** *simp*
      finally show s' *div unit-factor* $r' * x$ + t' *div unit-factor* $r' * y$ = *normalize* r' **.**
    next
      from *1.prems True* **show** *normalize* r' = *gcd-eucl* x y **by** (*simp add*: *gcd-eucl-0*)
    qed
    finally show *?thesis* **.**
  next

```
    case False
    hence euclid-ext-aux r' r s' s t' t =
            euclid-ext-aux r (r' mod r) s (s' − r' div r ∗ s) t (t' − r' div r ∗ t)
      by (subst euclid-ext-aux.simps) (simp add: Let-def)
    also from 1.prems False have ?P . . .
    proof (intro 1.IH)
      have (s' − r' div r ∗ s) ∗ x + (t' − r' div r ∗ t) ∗ y =
           (s' ∗ x + t' ∗ y) − r' div r ∗ (s ∗ x + t ∗ y) by (simp add: algebra-simps)
      also have s' ∗ x + t' ∗ y = r' by fact
      also have s ∗ x + t ∗ y = r by fact
      also have r' − r' div r ∗ r = r' mod r using mod-div-equality[of r' r]
        by (simp add: algebra-simps)
      finally show (s' − r' div r ∗ s) ∗ x + (t' − r' div r ∗ t) ∗ y = r' mod r .
    qed (auto simp: gcd-eucl-non-0 algebra-simps div-mod-equality')
    finally show ?thesis .
  qed
qed

definition euclid-ext where
  euclid-ext a b = euclid-ext-aux a b 1 0 0 1

lemma euclid-ext-0:
  euclid-ext a 0 = (1 div unit-factor a, 0, normalize a)
  by (simp add: euclid-ext-def euclid-ext-aux.simps)

lemma euclid-ext-left-0:
  euclid-ext 0 a = (0, 1 div unit-factor a, normalize a)
  by (simp add: euclid-ext-def euclid-ext-aux.simps)

lemma euclid-ext-correct':
  case euclid-ext x y of (a,b,c) ⇒ a ∗ x + b ∗ y = c ∧ c = gcd-eucl x y
  unfolding euclid-ext-def by (rule euclid-ext-aux-correct) simp-all

lemma euclid-ext-gcd-eucl:
  (case euclid-ext x y of (a,b,c) ⇒ c) = gcd-eucl x y
  using euclid-ext-correct'[of x y] by (simp add: case-prod-unfold)

definition euclid-ext' where
  euclid-ext' x y = (case euclid-ext x y of (a, b, -) ⇒ (a, b))

lemma euclid-ext'-correct':
  case euclid-ext' x y of (a,b) ⇒ a ∗ x + b ∗ y = gcd-eucl x y
  using euclid-ext-correct'[of x y] by (simp add: case-prod-unfold euclid-ext'-def)

lemma euclid-ext'-0: euclid-ext' a 0 = (1 div unit-factor a, 0)
  by (simp add: euclid-ext'-def euclid-ext-0)

lemma euclid-ext'-left-0: euclid-ext' 0 a = (0, 1 div unit-factor a)
  by (simp add: euclid-ext'-def euclid-ext-left-0)
```

end

class *euclidean-semiring-gcd* = *euclidean-semiring* + *gcd* + *Gcd* +
assumes *gcd-gcd-eucl*: *gcd* = *gcd-eucl* **and** *lcm-lcm-eucl*: *lcm* = *lcm-eucl*
assumes *Gcd-Gcd-eucl*: *Gcd* = *Gcd-eucl* **and** *Lcm-Lcm-eucl*: *Lcm* = *Lcm-eucl*
begin

subclass *semiring-gcd*
by *standard* (*simp-all add*: *gcd-gcd-eucl gcd-eucl-greatest lcm-lcm-eucl lcm-eucl-def*)

subclass *semiring-Gcd*
by *standard* (*auto simp*: *Gcd-Gcd-eucl Lcm-Lcm-eucl Gcd-eucl-def intro*: *Lcm-eucl-least*)

lemma *gcd-non-0*:
$b \neq 0 \Longrightarrow gcd\ a\ b = gcd\ b\ (a\ mod\ b)$
unfolding *gcd-gcd-eucl* **by** (*fact gcd-eucl-non-0*)

lemmas *gcd-0* = *gcd-0-right*
lemmas *dvd-gcd-iff* = *gcd-greatest-iff*
lemmas *gcd-greatest-iff* = *dvd-gcd-iff*

lemma *gcd-mod1* [*simp*]:
$gcd\ (a\ mod\ b)\ b = gcd\ a\ b$
by (*rule gcdI*, *metis dvd-mod-iff gcd-dvd1 gcd-dvd2*, *simp-all add*: *gcd-greatest dvd-mod-iff*)

lemma *gcd-mod2* [*simp*]:
$gcd\ a\ (b\ mod\ a) = gcd\ a\ b$
by (*rule gcdI*, *simp*, *metis dvd-mod-iff gcd-dvd1 gcd-dvd2*, *simp-all add*: *gcd-greatest dvd-mod-iff*)

lemma *euclidean-size-gcd-le1* [*simp*]:
assumes $a \neq 0$
shows $euclidean\text{-}size\ (gcd\ a\ b) \leq euclidean\text{-}size\ a$
proof −
have *gcd a b dvd a* **by** (*rule gcd-dvd1*)
then obtain *c* **where** *A*: $a = gcd\ a\ b * c$ **unfolding** *dvd-def* **by** *blast*
with ‹$a \neq 0$› **show** *?thesis* **by** (*subst* (*2*) *A*, *intro size-mult-mono*) *auto*
qed

lemma *euclidean-size-gcd-le2* [*simp*]:
$b \neq 0 \Longrightarrow euclidean\text{-}size\ (gcd\ a\ b) \leq euclidean\text{-}size\ b$
by (*subst gcd.commute*, *rule euclidean-size-gcd-le1*)

lemma *euclidean-size-gcd-less1*:
assumes $a \neq 0$ **and** $\neg a\ dvd\ b$
shows $euclidean\text{-}size\ (gcd\ a\ b) < euclidean\text{-}size\ a$
proof (*rule ccontr*)

assume ¬*euclidean-size* (*gcd a b*) < *euclidean-size a*
with ‹*a* ≠ *0*› **have** *A*: *euclidean-size* (*gcd a b*) = *euclidean-size a*
by (*intro le-antisym*, *simp-all*)
have *a dvd gcd a b*
by (*rule dvd-euclidean-size-eq-imp-dvd*) (*simp-all add*: *assms A*)
hence *a dvd b* **using** *dvd-gcdD2* **by** *blast*
with ‹¬*a dvd b*› **show** *False* **by** *contradiction*
qed

lemma *euclidean-size-gcd-less2*:
assumes *b* ≠ *0* **and** ¬*b dvd a*
shows *euclidean-size* (*gcd a b*) < *euclidean-size b*
using *assms* **by** (*subst gcd.commute*, *rule euclidean-size-gcd-less1*)

lemma *euclidean-size-lcm-le1*:
assumes *a* ≠ *0* **and** *b* ≠ *0*
shows *euclidean-size a* ≤ *euclidean-size* (*lcm a b*)
proof −
have *a dvd lcm a b* **by** (*rule dvd-lcm1*)
then obtain *c* **where** *A*: *lcm a b* = *a* ∗ *c* **..**
with ‹*a* ≠ *0*› **and** ‹*b* ≠ *0*› **have** *c* ≠ *0* **by** (*auto simp*: *lcm-eq-0-iff*)
then show *?thesis* **by** (*subst A*, *intro size-mult-mono*)
qed

lemma *euclidean-size-lcm-le2*:
a ≠ *0* ⟹ *b* ≠ *0* ⟹ *euclidean-size b* ≤ *euclidean-size* (*lcm a b*)
using *euclidean-size-lcm-le1* [*of b a*] **by** (*simp add*: *ac-simps*)

lemma *euclidean-size-lcm-less1*:
assumes *b* ≠ *0* **and** ¬*b dvd a*
shows *euclidean-size a* < *euclidean-size* (*lcm a b*)
proof (*rule ccontr*)
from *assms* **have** *a* ≠ *0* **by** *auto*
assume ¬*euclidean-size a* < *euclidean-size* (*lcm a b*)
with ‹*a* ≠ *0*› **and** ‹*b* ≠ *0*› **have** *euclidean-size* (*lcm a b*) = *euclidean-size a*
by (*intro le-antisym*, *simp*, *intro euclidean-size-lcm-le1*)
with *assms* **have** *lcm a b dvd a*
by (*rule-tac dvd-euclidean-size-eq-imp-dvd*) (*auto simp*: *lcm-eq-0-iff*)
hence *b dvd a* **by** (*rule lcm-dvdD2*)
with ‹¬*b dvd a*› **show** *False* **by** *contradiction*
qed

lemma *euclidean-size-lcm-less2*:
assumes *a* ≠ *0* **and** ¬*a dvd b*
shows *euclidean-size b* < *euclidean-size* (*lcm a b*)
using *assms euclidean-size-lcm-less1* [*of a b*] **by** (*simp add*: *ac-simps*)

lemma *Lcm-eucl-set* [*code*]:
Lcm-eucl (*set xs*) = *foldl lcm-eucl 1 xs*

by (*simp add*: *Lcm-Lcm-eucl* [*symmetric*] *lcm-lcm-eucl Lcm-set*)

lemma *Gcd-eucl-set* [*code*]:
Gcd-eucl (*set xs*) = *foldl gcd-eucl 0 xs*
by (*simp add*: *Gcd-Gcd-eucl* [*symmetric*] *gcd-gcd-eucl Gcd-set*)

end

A Euclidean ring is a Euclidean semiring with additive inverses. It provides a few more lemmas; in particular, Bezout's lemma holds for any Euclidean ring.

class *euclidean-ring-gcd* = *euclidean-semiring-gcd* + *idom*
begin

subclass *euclidean-ring* **..**
subclass *ring-gcd* **..**

lemma *euclid-ext-gcd* [*simp*]:
(*case euclid-ext a b of* (-, - , *t*) ⇒ *t*) = *gcd a b*
using *euclid-ext-correct'*[*of a b*] **by** (*simp add*: *case-prod-unfold Let-def gcd-gcd-eucl*)

lemma *euclid-ext-gcd'* [*simp*]:
euclid-ext a b = (*r*, *s*, *t*) ⟹ *t* = *gcd a b*
by (*insert euclid-ext-gcd*[*of a b*], *drule* (*1*) *subst*, *simp*)

lemma *euclid-ext-correct*:
case euclid-ext x y of (*a*,*b*,*c*) ⇒ *a* ∗ *x* + *b* ∗ *y* = *c* ∧ *c* = *gcd x y*
using *euclid-ext-correct'*[*of x y*]
by (*simp add*: *gcd-gcd-eucl case-prod-unfold*)

lemma *euclid-ext'-correct*:
fst (*euclid-ext' a b*) ∗ *a* + *snd* (*euclid-ext' a b*) ∗ *b* = *gcd a b*
using *euclid-ext-correct'*[*of a b*]
by (*simp add*: *gcd-gcd-eucl case-prod-unfold euclid-ext'-def*)

lemma *bezout*: ∃ *s t*. *s* ∗ *a* + *t* ∗ *b* = *gcd a b*
using *euclid-ext'-correct* **by** *blast*

end

42.1 Typical instances

instantiation *nat* :: *euclidean-semiring*
begin

definition [*simp*]:
euclidean-size-nat = (*id* :: *nat* ⇒ *nat*)

instance proof

qed *simp-all*

end

instantiation *int* :: *euclidean-ring*
begin

definition [*simp*]:
euclidean-size-int = (*nat* ∘ *abs* :: *int* ⇒ *nat*)

instance
by *standard* (*auto simp add*: *abs-mult nat-mult-distrib split*: *abs-split*)

end

instantiation *poly* :: (*field*) *euclidean-ring*
begin

definition *euclidean-size-poly* :: *'a poly* ⇒ *nat*
where *euclidean-size p* = (*if p* = *0 then 0 else 2* ^ *degree p*)

lemma *euclidean-size-poly-0* [*simp*]:
euclidean-size (*0*::*'a poly*) = *0*
by (*simp add*: *euclidean-size-poly-def*)

lemma *euclidean-size-poly-not-0* [*simp*]:
p ≠ *0* ⟹ *euclidean-size p* = *2* ^ *degree p*
by (*simp add*: *euclidean-size-poly-def*)

instance
proof
fix *p q* :: *'a poly*
assume *q* ≠ *0*
then have *p mod q* = *0* ∨ *degree* (*p mod q*) < *degree q*
by (*rule degree-mod-less* [*of q p*])
with ‹*q* ≠ *0*› **show** *euclidean-size* (*p mod q*) < *euclidean-size q*
by (*cases p mod q* = *0*) *simp-all*
next
fix *p q* :: *'a poly*
assume *q* ≠ *0*
from ‹*q* ≠ *0*› **have** *degree p* ≤ *degree* (*p* ∗ *q*)
by (*rule degree-mult-right-le*)
with ‹*q* ≠ *0*› **show** *euclidean-size p* ≤ *euclidean-size* (*p* ∗ *q*)
by (*cases p* = *0*) *simp-all*
qed *simp*

end

instance *nat* :: *euclidean-semiring-gcd*
proof
 show [*simp*]: *gcd* = (*gcd-eucl* :: *nat* ⇒ -) *Lcm* = (*Lcm-eucl* :: *nat set* ⇒ -)
  by (*simp-all add*: *eq-gcd-euclI eq-Lcm-euclI*)
 show *lcm* = (*lcm-eucl* :: *nat* ⇒ -) *Gcd* = (*Gcd-eucl* :: *nat set* ⇒ -)
  by (*intro ext*, *simp add*: *lcm-eucl-def lcm-nat-def Gcd-nat-def Gcd-eucl-def*)+
qed

instance *int* :: *euclidean-ring-gcd*
proof
 show [*simp*]: *gcd* = (*gcd-eucl* :: *int* ⇒ -) *Lcm* = (*Lcm-eucl* :: *int set* ⇒ -)
  by (*simp-all add*: *eq-gcd-euclI eq-Lcm-euclI*)
 show *lcm* = (*lcm-eucl* :: *int* ⇒ -) *Gcd* = (*Gcd-eucl* :: *int set* ⇒ -)
  by (*intro ext*, *simp add*: *lcm-eucl-def lcm-altdef-int*
   semiring-Gcd-class.Gcd-Lcm Gcd-eucl-def abs-mult)+
qed

instantiation *poly* :: (*field*) *euclidean-ring-gcd*
begin

definition *gcd-poly* :: *'a poly* ⇒ *'a poly* ⇒ *'a poly* **where**
 gcd-poly = *gcd-eucl*

definition *lcm-poly* :: *'a poly* ⇒ *'a poly* ⇒ *'a poly* **where**
 lcm-poly = *lcm-eucl*

definition *Gcd-poly* :: *'a poly set* ⇒ *'a poly* **where**
 Gcd-poly = *Gcd-eucl*

definition *Lcm-poly* :: *'a poly set* ⇒ *'a poly* **where**
 Lcm-poly = *Lcm-eucl*

instance by *standard* (*simp-all only*: *gcd-poly-def lcm-poly-def Gcd-poly-def Lcm-poly-def*)
end

lemma *poly-gcd-monic*:
 lead-coeff (*gcd x y*) = (*if x* = *0* ∧ *y* = *0 then 0 else 1*)
 using *unit-factor-gcd*[*of x y*]
 by (*simp add*: *unit-factor-poly-def monom-0 one-poly-def lead-coeff-def split*:
if-split-asm)

lemma *poly-dvd-antisym*:
 fixes *p q* :: *'a::idom poly*
 assumes *coeff*: *coeff p* (*degree p*) = *coeff q* (*degree q*)
 assumes *dvd1*: *p dvd q* **and** *dvd2*: *q dvd p* **shows** *p* = *q*
proof (*cases p* = *0*)

```
  case True with coeff show p = q by simp
next
  case False with coeff have q ≠ 0 by auto
  have degree: degree p = degree q
    using ‹p dvd q› ‹q dvd p› ‹p ≠ 0› ‹q ≠ 0›
    by (intro order-antisym dvd-imp-degree-le)

  from ‹p dvd q› obtain a where a: q = p * a ..
  with ‹q ≠ 0› have a ≠ 0 by auto
  with degree a ‹p ≠ 0› have degree a = 0
    by (simp add: degree-mult-eq)
  with coeff a show p = q
    by (cases a, auto split: if-splits)
qed

lemma poly-gcd-unique:
  fixes d x y :: - poly
  assumes dvd1: d dvd x and dvd2: d dvd y
    and greatest: ⋀k. k dvd x ⟹ k dvd y ⟹ k dvd d
    and monic: coeff d (degree d) = (if x = 0 ∧ y = 0 then 0 else 1)
  shows d = gcd x y
  using assms by (intro gcdI) (auto simp: normalize-poly-def split: if-split-asm)

lemma poly-gcd-code [code]:
  gcd x y = (if y = 0 then normalize x else gcd y (x mod (y :: - poly)))
  by (simp add: gcd-0 gcd-non-0)

end
```

43 A formalization of formal power series

```
theory Formal-Power-Series
imports Complex-Main ~~/src/HOL/Number-Theory/Euclidean-Algorithm
begin
```

43.1 The type of formal power series

```
typedef 'a fps = {f :: nat ⇒ 'a. True}
  morphisms fps-nth Abs-fps
  by simp

notation fps-nth (infixl $ 75)

lemma expand-fps-eq: p = q ⟷ (∀ n. p $ n = q $ n)
  by (simp add: fps-nth-inject [symmetric] fun-eq-iff)

lemma fps-ext: (⋀n. p $ n = q $ n) ⟹ p = q
  by (simp add: expand-fps-eq)
```

lemma *fps-nth-Abs-fps* [*simp*]: *Abs-fps f* \$ *n* = *f n*
by (*simp add*: *Abs-fps-inverse*)

Definition of the basic elements 0 and 1 and the basic operations of addition, negation and multiplication.

instantiation *fps* :: (*zero*) *zero*
begin
definition *fps-zero-def*: *0* = *Abs-fps* (λ*n*. *0*)
instance ..
end

lemma *fps-zero-nth* [*simp*]: *0* \$ *n* = *0*
unfolding *fps-zero-def* **by** *simp*

instantiation *fps* :: ({*one*, *zero*}) *one*
begin
definition *fps-one-def*: *1* = *Abs-fps* (λ*n*. *if n* = *0 then 1 else 0*)
instance ..
end

lemma *fps-one-nth* [*simp*]: *1* \$ *n* = (*if n* = *0 then 1 else 0*)
unfolding *fps-one-def* **by** *simp*

instantiation *fps* :: (*plus*) *plus*
begin
definition *fps-plus-def*: *op* + = (λ*f g*. *Abs-fps* (λ*n*. *f* \$ *n* + *g* \$ *n*))
instance ..
end

lemma *fps-add-nth* [*simp*]: (*f* + *g*) \$ *n* = *f* \$ *n* + *g* \$ *n*
unfolding *fps-plus-def* **by** *simp*

instantiation *fps* :: (*minus*) *minus*
begin
definition *fps-minus-def*: *op* − = (λ*f g*. *Abs-fps* (λ*n*. *f* \$ *n* − *g* \$ *n*))
instance ..
end

lemma *fps-sub-nth* [*simp*]: (*f* − *g*) \$ *n* = *f* \$ *n* − *g* \$ *n*
unfolding *fps-minus-def* **by** *simp*

instantiation *fps* :: (*uminus*) *uminus*
begin
definition *fps-uminus-def*: *uminus* = (λ*f*. *Abs-fps* (λ*n*. − (*f* \$ *n*)))
instance ..
end

lemma *fps-neg-nth* [*simp*]: (− *f*) \$ *n* = − (*f* \$ *n*)
unfolding *fps-uminus-def* **by** *simp*

instantiation *fps* :: ({*comm-monoid-add*, *times*}) *times*
begin
definition *fps-times-def*: *op* $*$ = ($\lambda f\ g.$ *Abs-fps* ($\lambda n.$ $\sum$ *i=0..n. f* \$ *i* $*$ *g* \$ (*n* − *i*)))
instance ..
end

lemma *fps-mult-nth*: (*f* $*$ *g*) \$ *n* = ($\sum$ *i=0..n. f*\$*i* $*$ *g*\$(*n* − *i*))
unfolding *fps-times-def* **by** *simp*

lemma *fps-mult-nth-0* [*simp*]: (*f* $*$ *g*) \$ *0* = *f* \$ *0* $*$ *g* \$ *0*
unfolding *fps-times-def* **by** *simp*

declare *atLeastAtMost-iff* [*presburger*]
declare *Bex-def* [*presburger*]
declare *Ball-def* [*presburger*]

lemma *mult-delta-left*:
fixes *x y* :: *'a::mult-zero*
shows (*if b then x else 0*) $*$ *y* = (*if b then x* $*$ *y else 0*)
by *simp*

lemma *mult-delta-right*:
fixes *x y* :: *'a::mult-zero*
shows *x* $*$ (*if b then y else 0*) = (*if b then x* $*$ *y else 0*)
by *simp*

lemma *cond-value-iff*: *f* (*if b then x else y*) = (*if b then f x else f y*)
by *auto*

lemma *cond-application-beta*: (*if b then f else g*) *x* = (*if b then f x else g x*)
by *auto*

43.2 Formal power series form a commutative ring with unity, if the range of sequences they represent is a commutative ring with unity

instance *fps* :: (*semigroup-add*) *semigroup-add*
proof
fix *a b c* :: *'a fps*
show *a* + *b* + *c* = *a* + (*b* + *c*)
by (*simp add*: *fps-ext add.assoc*)
qed

instance *fps* :: (*ab-semigroup-add*) *ab-semigroup-add*
proof
fix *a b* :: *'a fps*
show *a* + *b* = *b* + *a*

by (*simp add*: *fps-ext add.commute*)
qed

lemma *fps-mult-assoc-lemma*:
fixes k :: *nat*
and f :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *'a::comm-monoid-add*
shows $(\sum j{=}0..k.\ \sum i{=}0..j.\ f\ i\ (j - i)\ (n - j)) =$
$(\sum j{=}0..k.\ \sum i{=}0..k - j.\ f\ j\ i\ (n - j - i))$
by (*induct k*) (*simp-all add*: *Suc-diff-le setsum.distrib add.assoc*)

instance *fps* :: (*semiring-0*) *semigroup-mult*
proof
fix *a b c* :: *'a fps*
show $(a * b) * c = a * (b * c)$
proof (*rule fps-ext*)
fix n :: *nat*
have $(\sum j{=}0..n.\ \sum i{=}0..j.\ a\$i * b\$(j - i) * c\$(n - j)) =$
$(\sum j{=}0..n.\ \sum i{=}0..n - j.\ a\$j * b\$i * c\$(n - j - i))$
by (*rule fps-mult-assoc-lemma*)
then show $((a * b) * c)\ \$\ n = (a * (b * c))\ \$\ n$
by (*simp add*: *fps-mult-nth setsum-right-distrib setsum-left-distrib mult.assoc*)
qed
qed

lemma *fps-mult-commute-lemma*:
fixes n :: *nat*
and f :: *nat* ⇒ *nat* ⇒ *'a::comm-monoid-add*
shows $(\sum i{=}0..n.\ f\ i\ (n - i)) = (\sum i{=}0..n.\ f\ (n - i)\ i)$
by (*rule setsum.reindex-bij-witness*[**where** $i{=}op - n$ **and** $j{=}op - n$]) *auto*

instance *fps* :: (*comm-semiring-0*) *ab-semigroup-mult*
proof
fix *a b* :: *'a fps*
show $a * b = b * a$
proof (*rule fps-ext*)
fix n :: *nat*
have $(\sum i{=}0..n.\ a\$i * b\$(n - i)) = (\sum i{=}0..n.\ a\$(n - i) * b\$i)$
by (*rule fps-mult-commute-lemma*)
then show $(a * b)\ \$\ n = (b * a)\ \$\ n$
by (*simp add*: *fps-mult-nth mult.commute*)
qed
qed

instance *fps* :: (*monoid-add*) *monoid-add*
proof
fix a :: *'a fps*
show $0 + a = a$ **by** (*simp add*: *fps-ext*)
show $a + 0 = a$ **by** (*simp add*: *fps-ext*)
qed

```
instance fps :: (comm-monoid-add) comm-monoid-add
proof
  fix a :: 'a fps
  show 0 + a = a by (simp add: fps-ext)
qed

instance fps :: (semiring-1) monoid-mult
proof
  fix a :: 'a fps
  show 1 * a = a
    by (simp add: fps-ext fps-mult-nth mult-delta-left setsum.delta)
  show a * 1 = a
    by (simp add: fps-ext fps-mult-nth mult-delta-right setsum.delta')
qed

instance fps :: (cancel-semigroup-add) cancel-semigroup-add
proof
  fix a b c :: 'a fps
  show b = c if a + b = a + c
    using that by (simp add: expand-fps-eq)
  show b = c if b + a = c + a
    using that by (simp add: expand-fps-eq)
qed

instance fps :: (cancel-ab-semigroup-add) cancel-ab-semigroup-add
proof
  fix a b c :: 'a fps
  show a + b − a = b
    by (simp add: expand-fps-eq)
  show a − b − c = a − (b + c)
    by (simp add: expand-fps-eq diff-diff-eq)
qed

instance fps :: (cancel-comm-monoid-add) cancel-comm-monoid-add ..

instance fps :: (group-add) group-add
proof
  fix a b :: 'a fps
  show − a + a = 0 by (simp add: fps-ext)
  show a + − b = a − b by (simp add: fps-ext)
qed

instance fps :: (ab-group-add) ab-group-add
proof
  fix a b :: 'a fps
  show − a + a = 0 by (simp add: fps-ext)
  show a − b = a + − b by (simp add: fps-ext)
qed
```

instance *fps* :: (*zero-neq-one*) *zero-neq-one*
  by *standard* (*simp add*: *expand-fps-eq*)

instance *fps* :: (*semiring-0*) *semiring*
proof
  fix *a b c* :: *'a fps*
  show $(a + b) * c = a * c + b * c$
    by (*simp add*: *expand-fps-eq fps-mult-nth distrib-right setsum.distrib*)
  show $a * (b + c) = a * b + a * c$
    by (*simp add*: *expand-fps-eq fps-mult-nth distrib-left setsum.distrib*)
qed

instance *fps* :: (*semiring-0*) *semiring-0*
proof
  fix *a* :: *'a fps*
  show $0 * a = 0$
    by (*simp add*: *fps-ext fps-mult-nth*)
  show $a * 0 = 0$
    by (*simp add*: *fps-ext fps-mult-nth*)
qed

instance *fps* :: (*semiring-0-cancel*) *semiring-0-cancel* **..**

instance *fps* :: (*semiring-1*) *semiring-1* **..**

43.3 Selection of the nth power of the implicit variable in the infinite sum

lemma *fps-nonzero-nth*: $f \neq 0 \longleftrightarrow (\exists\ n.\ f\ \$n \neq 0)$
  by (*simp add*: *expand-fps-eq*)

lemma *fps-nonzero-nth-minimal*: $f \neq 0 \longleftrightarrow (\exists n.\ f\ \$\ n \neq 0 \land (\forall m < n.\ f\ \$\ m = 0))$
  (**is** *?lhs* ⟷ *?rhs*)
proof
  let $?n = LEAST\ n.\ f\ \$\ n \neq 0$
  show *?rhs* **if** *?lhs*
  proof −
    from *that* **have** $\exists n.\ f\ \$\ n \neq 0$
      by (*simp add*: *fps-nonzero-nth*)
    then have $f\ \$\ ?n \neq 0$
      by (*rule LeastI-ex*)
    moreover have $\forall m<?n.\ f\ \$\ m = 0$
      by (*auto dest*: *not-less-Least*)
    ultimately have $f\ \$\ ?n \neq 0 \land (\forall m<?n.\ f\ \$\ m = 0)$ **..**
    then show *?thesis* **..**
  qed
  show *?lhs* **if** *?rhs*

using *that* **by** (*auto simp add*: *expand-fps-eq*)
qed

lemma *fps-eq-iff*: $f = g \longleftrightarrow (\forall n.\ f \ \$ \ n = g \ \$n)$
by (*rule expand-fps-eq*)

lemma *fps-setsum-nth*: *setsum f S* $ *n* = *setsum* (λ*k*. (*f k*) $ *n*) *S*
proof (*cases finite S*)
case *True*
then show *?thesis* **by** (*induct set*: *finite*) *auto*
next
case *False*
then show *?thesis* **by** *simp*
qed

43.4 Injection of the basic ring elements and multiplication by scalars

definition *fps-const c* = *Abs-fps* (λ*n*. *if n* = *0 then c else 0*)

lemma *fps-nth-fps-const* [*simp*]: *fps-const c* $ *n* = (*if n* = *0 then c else 0*)
unfolding *fps-const-def* **by** *simp*

lemma *fps-const-0-eq-0* [*simp*]: *fps-const 0* = *0*
by (*simp add*: *fps-ext*)

lemma *fps-const-1-eq-1* [*simp*]: *fps-const 1* = *1*
by (*simp add*: *fps-ext*)

lemma *fps-const-neg* [*simp*]: − (*fps-const* (*c*::′*a*::*ring*)) = *fps-const* (− *c*)
by (*simp add*: *fps-ext*)

lemma *fps-const-add* [*simp*]: *fps-const* (*c*::′*a*::*monoid-add*) + *fps-const d* = *fps-const* (*c* + *d*)
by (*simp add*: *fps-ext*)

lemma *fps-const-sub* [*simp*]: *fps-const* (*c*::′*a*::*group-add*) − *fps-const d* = *fps-const* (*c* − *d*)
by (*simp add*: *fps-ext*)

lemma *fps-const-mult*[*simp*]: *fps-const* (*c*::′*a*::*ring*) ∗ *fps-const d* = *fps-const* (*c* ∗ *d*)
by (*simp add*: *fps-eq-iff fps-mult-nth setsum.neutral*)

lemma *fps-const-add-left*: *fps-const* (*c*::′*a*::*monoid-add*) + *f* =
Abs-fps (λ*n*. *if n* = *0 then c* + *f*$*0 else f*$*n*)
by (*simp add*: *fps-ext*)

lemma *fps-const-add-right*: *f* + *fps-const* (*c*::′*a*::*monoid-add*) =

Abs-fps ($\lambda n.$ *if* $n = 0$ *then* $f\$0 + c$ *else* $f\$n$)
by (*simp add*: *fps-ext*)

lemma *fps-const-mult-left*: *fps-const* (c::$'a$::*semiring-0*) $*$ f = *Abs-fps* ($\lambda n.$ c $*$ $f\$n$)
unfolding *fps-eq-iff fps-mult-nth*
by (*simp add*: *fps-const-def mult-delta-left setsum.delta*)

lemma *fps-const-mult-right*: f $*$ *fps-const* (c::$'a$::*semiring-0*) = *Abs-fps* ($\lambda n.$ $f\$n$ $*$ c)
unfolding *fps-eq-iff fps-mult-nth*
by (*simp add*: *fps-const-def mult-delta-right setsum.delta'*)

lemma *fps-mult-left-const-nth* [*simp*]: (*fps-const* (c::$'a$::*semiring-1*) $*$ f)$\$n$ = $c*$ $f\$n$
by (*simp add*: *fps-mult-nth mult-delta-left setsum.delta*)

lemma *fps-mult-right-const-nth* [*simp*]: (f $*$ *fps-const* (c::$'a$::*semiring-1*))$\$n$ = $f\$n$ $*$ c
by (*simp add*: *fps-mult-nth mult-delta-right setsum.delta'*)

43.5 Formal power series form an integral domain

instance *fps* :: (*ring*) *ring* **..**

instance *fps* :: (*ring-1*) *ring-1*
by (*intro-classes*, *auto simp add*: *distrib-right*)

instance *fps* :: (*comm-ring-1*) *comm-ring-1*
by (*intro-classes*, *auto simp add*: *distrib-right*)

instance *fps* :: (*ring-no-zero-divisors*) *ring-no-zero-divisors*
proof
fix a b :: $'a$ *fps*
assume $a \neq 0$ **and** $b \neq 0$
then obtain i j **where** i: $a \$ i \neq 0$ $\forall k{<}i.$ $a \$ k = 0$ **and** j: $b \$ j \neq 0$ $\forall k{<}j.$ $b \$ k = 0$
unfolding *fps-nonzero-nth-minimal*
by *blast+*
have $(a * b) \$ (i + j) = (\sum k{=}0..i{+}j.\ a \$ k * b \$ (i + j - k))$
by (*rule fps-mult-nth*)
also have $\ldots = (a \$ i * b \$ (i + j - i)) + (\sum k{\in}\{0..i{+}j\} - \{i\}.\ a \$ k * b \$ (i + j - k))$
by (*rule setsum.remove*) *simp-all*
also have $(\sum k{\in}\{0..i{+}j\}{-}\{i\}.\ a \$ k * b \$ (i + j - k)) = 0$
proof (*rule setsum.neutral* [*rule-format*])
fix k **assume** $k \in \{0..i{+}j\} - \{i\}$
then have $k < i \vee i{+}j{-}k < j$
by *auto*

then show $a \$ k * b \$ (i + j - k) = 0$
using *i j* **by** *auto*
qed
also have $a \$ i * b \$ (i + j - i) + 0 = a \$ i * b \$ j$
by *simp*
also have $a \$ i * b \$ j \neq 0$
using *i j* **by** *simp*
finally have $(a*b) \$ (i+j) \neq 0$ **.**
then show $a * b \neq 0$
unfolding *fps-nonzero-nth* **by** *blast*
qed

instance *fps* :: (*ring-1-no-zero-divisors*) *ring-1-no-zero-divisors* **..**

instance *fps* :: (*idom*) *idom* **..**

lemma *numeral-fps-const*: *numeral k* = *fps-const* (*numeral k*)
by (*induct k*) (*simp-all only*: *numeral.simps fps-const-1-eq-1*
fps-const-add [*symmetric*])

lemma *neg-numeral-fps-const*:
(− *numeral k* :: *'a* :: *ring-1 fps*) = *fps-const* (− *numeral k*)
by (*simp add*: *numeral-fps-const*)

lemma *fps-numeral-nth*: *numeral n* \$ *i* = (*if i* = *0 then numeral n else 0*)
by (*simp add*: *numeral-fps-const*)

lemma *fps-numeral-nth-0* [*simp*]: *numeral n* \$ *0* = *numeral n*
by (*simp add*: *numeral-fps-const*)

43.6 The eXtractor series X

lemma *minus-one-power-iff*: (− (*1*::*'a*::*comm-ring-1*)) ^ *n* = (*if even n then 1 else* − *1*)
by (*induct n*) *auto*

definition *X* = *Abs-fps* ($\lambda n.$ *if n* = *1 then 1 else 0*)

lemma *X-mult-nth* [*simp*]:
(*X* ∗ (*f* :: *'a*::*semiring-1 fps*)) \$*n* = (*if n* = *0 then 0 else f* \$ (*n* − *1*))
proof (*cases n* = *0*)
case *False*
have $(X * f) \$n = (\sum i = 0..n.\ X \$ i * f \$ (n - i))$
by (*simp add*: *fps-mult-nth*)
also have $\ldots = f \$ (n - 1)$
using *False* **by** (*simp add*: *X-def mult-delta-left setsum.delta*)
finally show *?thesis*
using *False* **by** *simp*
next

case *True*
then show *?thesis*
by (*simp add*: *fps-mult-nth X-def*)
qed

lemma *X-mult-right-nth*[*simp*]:
((*f* :: *'a::comm-semiring-1 fps*) ∗ *X*) $*n* = (*if n* = *0 then 0 else f* $ (*n* − *1*))
by (*metis X-mult-nth mult.commute*)

lemma *X-power-iff*: *X*ˆ*k* = *Abs-fps* (λ*n*. *if n* = *k then 1::'a::comm-ring-1 else 0*)
proof (*induct k*)
case *0*
then show *?case* **by** (*simp add*: *X-def fps-eq-iff*)
next
case (*Suc k*)
have (*X*ˆ*Suc k*) $ *m* = (*if m* = *Suc k then 1::'a else 0*) **for** *m*
proof −
have (*X*ˆ*Suc k*) $ *m* = (*if m* = *0 then 0 else* (*X*ˆ*k*) $ (*m* − *1*))
by (*simp del*: *One-nat-def*)
then show *?thesis*
using *Suc.hyps* **by** (*auto cong del*: *if-weak-cong*)
qed
then show *?case*
by (*simp add*: *fps-eq-iff*)
qed

lemma *X-nth*[*simp*]: *X*$*n* = (*if n* = *1 then 1 else 0*)
by (*simp add*: *X-def*)

lemma *X-power-nth*[*simp*]: (*X*ˆ*k*) $*n* = (*if n* = *k then 1 else 0::'a::comm-ring-1*)
by (*simp add*: *X-power-iff*)

lemma *X-power-mult-nth*: (*X*ˆ*k* ∗ (*f* :: *'a::comm-ring-1 fps*)) $*n* = (*if n* < *k then 0 else f* $ (*n* − *k*))
apply (*induct k arbitrary*: *n*)
apply *simp*
unfolding *power-Suc mult.assoc*
apply (*case-tac n*)
apply *auto*
done

lemma *X-power-mult-right-nth*:
((*f* :: *'a::comm-ring-1 fps*) ∗ *X*ˆ*k*) $*n* = (*if n* < *k then 0 else f* $ (*n* − *k*))
by (*metis X-power-mult-nth mult.commute*)

lemma *X-neq-fps-const* [*simp*]: (*X* :: *'a* :: *zero-neq-one fps*) ≠ *fps-const c*
proof
assume (*X::'a fps*) = *fps-const* (*c::'a*)

hence $X\$1 = (fps\text{-}const\ (c::'a))\$1$ **by** (*simp only*:)
thus *False* **by** *auto*
qed

lemma *X-neq-zero* [*simp*]: $(X :: 'a :: zero\text{-}neq\text{-}one\ fps) \neq 0$
by (*simp only*: *fps-const-0-eq-0*[*symmetric*] *X-neq-fps-const*) *simp*

lemma *X-neq-one* [*simp*]: $(X :: 'a :: zero\text{-}neq\text{-}one\ fps) \neq 1$
by (*simp only*: *fps-const-1-eq-1*[*symmetric*] *X-neq-fps-const*) *simp*

lemma *X-neq-numeral* [*simp*]: $(X :: 'a :: \{semiring\text{-}1, zero\text{-}neq\text{-}one\}\ fps) \neq numeral\ c$
by (*simp only*: *numeral-fps-const X-neq-fps-const*) *simp*

lemma *X-pow-eq-X-pow-iff* [*simp*]:
$(X :: ('a :: \{comm\text{-}ring\text{-}1\})\ fps)\ \hat{}\ m = X\ \hat{}\ n \longleftrightarrow m = n$
proof
assume $(X :: 'a\ fps)\ \hat{}\ m = X\ \hat{}\ n$
hence $(X :: 'a\ fps)\ \hat{}\ m\ \$\ m = X\ \hat{}\ n\ \$\ m$ **by** (*simp only*:)
thus $m = n$ **by** (*simp split*: *if-split-asm*)
qed *simp-all*

43.7 Subdegrees

definition *subdegree* :: $('a::zero)\ fps \Rightarrow nat$ **where**
$subdegree\ f = (if\ f = 0\ then\ 0\ else\ LEAST\ n.\ f\$n \neq 0)$

lemma *subdegreeI*:
assumes $f\ \$\ d \neq 0$ **and** $\bigwedge i.\ i < d \Longrightarrow f\ \$\ i = 0$
shows $subdegree\ f = d$
proof−
from *assms*(*1*) **have** $f \neq 0$ **by** *auto*
moreover from *assms*(*1*) **have** $(LEAST\ i.\ f\ \$\ i \neq 0) = d$
proof (*rule Least-equality*)
fix *e* **assume** $f\ \$\ e \neq 0$
with *assms*(*2*) **have** $\neg(e < d)$ **by** *blast*
thus $e \geq d$ **by** *simp*
qed
ultimately show *?thesis* **unfolding** *subdegree-def* **by** *simp*
qed

lemma *nth-subdegree-nonzero* [*simp*,*intro*]: $f \neq 0 \Longrightarrow f\ \$\ subdegree\ f \neq 0$
proof−
assume $f \neq 0$
hence $subdegree\ f = (LEAST\ n.\ f\ \$\ n \neq 0)$ **by** (*simp add*: *subdegree-def*)
also from ‹$f \neq 0$› **have** $\exists n.\ f\$n \neq 0$ **using** *fps-nonzero-nth* **by** *blast*
from *LeastI-ex*[*OF this*] **have** $f\ \$\ (LEAST\ n.\ f\ \$\ n \neq 0) \neq 0$ **.**
finally show *?thesis* **.**
qed

lemma *nth-less-subdegree-zero* [*dest*]: $n < \mathit{subdegree}\ f \Longrightarrow f\ \$\ n = 0$
proof (*cases* $f = 0$)
  assume $f \neq 0$ **and** *less*: $n < \mathit{subdegree}\ f$
  note *less*
  also from ‹$f \neq 0$› **have** $\mathit{subdegree}\ f = (\mathit{LEAST}\ n.\ f\ \$\ n \neq 0)$ **by** (*simp add*: *subdegree-def*)
  finally show $f\ \$\ n = 0$ **using** *not-less-Least* **by** *blast*
qed *simp-all*

lemma *subdegree-geI*:
  assumes $f \neq 0\ \bigwedge i.\ i < n \Longrightarrow f\$i = 0$
  shows $\mathit{subdegree}\ f \geq n$
proof (*rule ccontr*)
  assume $\neg(\mathit{subdegree}\ f \geq n)$
  with *assms*(*2*) **have** $f\ \$\ \mathit{subdegree}\ f = 0$ **by** *simp*
  moreover from *assms*(*1*) **have** $f\ \$\ \mathit{subdegree}\ f \neq 0$ **by** *simp*
  ultimately show *False* **by** *contradiction*
qed

lemma *subdegree-greaterI*:
  assumes $f \neq 0\ \bigwedge i.\ i \leq n \Longrightarrow f\$i = 0$
  shows $\mathit{subdegree}\ f > n$
proof (*rule ccontr*)
  assume $\neg(\mathit{subdegree}\ f > n)$
  with *assms*(*2*) **have** $f\ \$\ \mathit{subdegree}\ f = 0$ **by** *simp*
  moreover from *assms*(*1*) **have** $f\ \$\ \mathit{subdegree}\ f \neq 0$ **by** *simp*
  ultimately show *False* **by** *contradiction*
qed

lemma *subdegree-leI*:
  $f\ \$\ n \neq 0 \Longrightarrow \mathit{subdegree}\ f \leq n$
  by (*rule leI*) *auto*

lemma *subdegree-0* [*simp*]: $\mathit{subdegree}\ 0 = 0$
  by (*simp add*: *subdegree-def*)

lemma *subdegree-1* [*simp*]: $\mathit{subdegree}\ (1 :: ('a :: \mathit{zero\text{-}neq\text{-}one})\ \mathit{fps}) = 0$
  by (*auto intro*!: *subdegreeI*)

lemma *subdegree-X* [*simp*]: $\mathit{subdegree}\ (X :: ('a :: \mathit{zero\text{-}neq\text{-}one})\ \mathit{fps}) = 1$
  by (*auto intro*!: *subdegreeI simp*: *X-def*)

lemma *subdegree-fps-const* [*simp*]: $\mathit{subdegree}\ (\mathit{fps\text{-}const}\ c) = 0$
  by (*cases* $c = 0$) (*auto intro*!: *subdegreeI*)

lemma *subdegree-numeral* [*simp*]: $\mathit{subdegree}\ (\mathit{numeral}\ n) = 0$
  by (*simp add*: *numeral-fps-const*)

lemma *subdegree-eq-0-iff*: *subdegree f* = *0* ⟷ *f* = *0* ∨ *f* \$ *0* ≠ *0*
proof (*cases f* = *0*)
  assume *f* ≠ *0*
  thus *?thesis*
    using *nth-subdegree-nonzero*[*OF* ‹*f* ≠ *0*›] **by** (*fastforce intro*!: *subdegreeI*)
qed *simp-all*

lemma *subdegree-eq-0* [*simp*]: *f* \$ *0* ≠ *0* ⟹ *subdegree f* = *0*
  by (*simp add*: *subdegree-eq-0-iff*)

lemma *nth-subdegree-mult* [*simp*]:
  fixes *f g* :: (*'a* :: {*mult-zero*,*comm-monoid-add*}) *fps*
  shows (*f* ∗ *g*) \$ (*subdegree f* + *subdegree g*) = *f* \$ *subdegree f* ∗ *g* \$ *subdegree g*
proof−
  let *?n* = *subdegree f* + *subdegree g*
  have (*f* ∗ *g*) \$ *?n* = (∑ *i*=*0*..*?n*. *f*\$*i* ∗ *g*\$(*?n*−*i*))
    by (*simp add*: *fps-mult-nth*)
  also have ... = (∑ *i*=*0*..*?n*. *if i* = *subdegree f then f*\$*i* ∗ *g*\$(*?n*−*i*) *else 0*)
  proof (*intro setsum.cong*)
    fix *x* **assume** *x*: *x* ∈ {*0*..*?n*}
    hence *x* = *subdegree f* ∨ *x* < *subdegree f* ∨ *?n* − *x* < *subdegree g* **by** *auto*
    thus *f* \$ *x* ∗ *g* \$ (*?n* − *x*) = (*if x* = *subdegree f then f* \$ *x* ∗ *g* \$ (*?n* − *x*) *else* *0*)
      by (*elim disjE conjE*) *auto*
  qed *auto*
  also have ... = *f* \$ *subdegree f* ∗ *g* \$ *subdegree g* **by** (*simp add*: *setsum.delta*)
  finally show *?thesis* **.**
qed

lemma *subdegree-mult* [*simp*]:
  assumes *f* ≠ *0 g* ≠ *0*
  shows *subdegree* ((*f* :: (*'a* :: {*ring-no-zero-divisors*}) *fps*) ∗ *g*) = *subdegree f* + *subdegree g*
proof (*rule subdegreeI*)
  let *?n* = *subdegree f* + *subdegree g*
  have (*f* ∗ *g*) \$ *?n* = (∑ *i*=*0*..*?n*. *f*\$*i* ∗ *g*\$(*?n*−*i*)) **by** (*simp add*: *fps-mult-nth*)
  also have ... = (∑ *i*=*0*..*?n*. *if i* = *subdegree f then f*\$*i* ∗ *g*\$(*?n*−*i*) *else 0*)
  proof (*intro setsum.cong*)
    fix *x* **assume** *x*: *x* ∈ {*0*..*?n*}
    hence *x* = *subdegree f* ∨ *x* < *subdegree f* ∨ *?n* − *x* < *subdegree g* **by** *auto*
    thus *f* \$ *x* ∗ *g* \$ (*?n* − *x*) = (*if x* = *subdegree f then f* \$ *x* ∗ *g* \$ (*?n* − *x*) *else* *0*)
      by (*elim disjE conjE*) *auto*
  qed *auto*
  also have ... = *f* \$ *subdegree f* ∗ *g* \$ *subdegree g* **by** (*simp add*: *setsum.delta*)
  also from *assms* **have** ... ≠ *0* **by** *auto*
  finally show (*f* ∗ *g*) \$ (*subdegree f* + *subdegree g*) ≠ *0* **.**
next

fix *m* **assume** *m*: *m* < *subdegree f* + *subdegree g*
have (*f* ∗ *g*) $ *m* = (∑ *i*=*0*..*m*. *f*$*i* ∗ *g*$(*m*−*i*)) **by** (*simp add*: *fps-mult-nth*)
also have ... = (∑ *i*=*0*..*m*. *0*)
proof (*rule setsum.cong*)
fix *i* **assume** *i* ∈ {*0*..*m*}
with *m* **have** *i* < *subdegree f* ∨ *m* − *i* < *subdegree g* **by** *auto*
thus *f*$*i* ∗ *g*$(*m*−*i*) = *0* **by** (*elim disjE*) *auto*
qed *auto*
finally show (*f* ∗ *g*) $ *m* = *0* **by** *simp*
qed

lemma *subdegree-power* [*simp*]:
subdegree ((*f* :: ('*a* :: *ring-1-no-zero-divisors*) *fps*) ˆ *n*) = *n* ∗ *subdegree f*
by (*cases f* = *0*; *induction n*) *simp-all*

lemma *subdegree-uminus* [*simp*]:
subdegree (−(*f*::('*a*::*group-add*) *fps*)) = *subdegree f*
by (*simp add*: *subdegree-def*)

lemma *subdegree-minus-commute* [*simp*]:
subdegree (*f*−(*g*::('*a*::*group-add*) *fps*)) = *subdegree* (*g* − *f*)
proof −
have *f* − *g* = −(*g* − *f*) **by** *simp*
also have *subdegree* ... = *subdegree* (*g* − *f*) **by** (*simp only*: *subdegree-uminus*)
finally show *?thesis* **.**
qed

lemma *subdegree-add-ge*:
assumes *f* ≠ −(*g* :: ('*a* :: {*group-add*}) *fps*)
shows *subdegree* (*f* + *g*) ≥ *min* (*subdegree f*) (*subdegree g*)
proof (*rule subdegree-geI*)
from *assms* **show** *f* + *g* ≠ *0* **by** (*subst* (*asm*) *eq-neg-iff-add-eq-0*)
next
fix *i* **assume** *i* < *min* (*subdegree f*) (*subdegree g*)
hence *f* $ *i* = *0* **and** *g* $ *i* = *0* **by** *auto*
thus (*f* + *g*) $ *i* = *0* **by** *force*
qed

lemma *subdegree-add-eq1*:
assumes *f* ≠ *0*
assumes *subdegree f* < *subdegree* (*g* :: ('*a* :: {*group-add*}) *fps*)
shows *subdegree* (*f* + *g*) = *subdegree f*
proof (*rule antisym*[*OF subdegree-leI*])
from *assms* **show** *subdegree* (*f* + *g*) ≥ *subdegree f*
by (*intro order.trans*[*OF min.boundedI subdegree-add-ge*]) *auto*
from *assms* **have** *f* $ *subdegree f* ≠ *0 g* $ *subdegree f* = *0* **by** *auto*
thus (*f* + *g*) $ *subdegree f* ≠ *0* **by** *simp*
qed

lemma *subdegree-add-eq2*:
assumes $g \neq 0$
assumes *subdegree g* < *subdegree* (*f* :: ('*a* :: {*ab-group-add*}) *fps*)
shows *subdegree* (*f* + *g*) = *subdegree g*
using *subdegree-add-eq1*[*OF assms*] **by** (*simp add*: *add.commute*)

lemma *subdegree-diff-eq1*:
assumes $f \neq 0$
assumes *subdegree f* < *subdegree* (*g* :: ('*a* :: {*ab-group-add*}) *fps*)
shows *subdegree* (*f* − *g*) = *subdegree f*
using *subdegree-add-eq1*[*of f* −*g*] *assms* **by** (*simp add*: *add.commute*)

lemma *subdegree-diff-eq2*:
assumes $g \neq 0$
assumes *subdegree g* < *subdegree* (*f* :: ('*a* :: {*ab-group-add*}) *fps*)
shows *subdegree* (*f* − *g*) = *subdegree g*
using *subdegree-add-eq2*[*of* −*g f*] *assms* **by** (*simp add*: *add.commute*)

lemma *subdegree-diff-ge* [*simp*]:
assumes *f* ≠ (*g* :: ('*a* :: {*group-add*}) *fps*)
shows *subdegree* (*f* − *g*) ≥ *min* (*subdegree f*) (*subdegree g*)
using *assms subdegree-add-ge*[*of f* −*g*] **by** *simp*

43.8 Shifting and slicing

definition *fps-shift* :: *nat* ⇒ '*a fps* ⇒ '*a fps* **where**
fps-shift n f = *Abs-fps* (λ*i*. *f* $ (*i* + *n*))

lemma *fps-shift-nth* [*simp*]: *fps-shift n f* $ *i* = *f* $ (*i* + *n*)
by (*simp add*: *fps-shift-def*)

lemma *fps-shift-0* [*simp*]: *fps-shift 0 f* = *f*
by (*intro fps-ext*) (*simp add*: *fps-shift-def*)

lemma *fps-shift-zero* [*simp*]: *fps-shift n 0* = *0*
by (*intro fps-ext*) (*simp add*: *fps-shift-def*)

lemma *fps-shift-one*: *fps-shift n 1* = (*if n* = *0 then 1 else 0*)
by (*intro fps-ext*) (*simp add*: *fps-shift-def*)

lemma *fps-shift-fps-const*: *fps-shift n* (*fps-const c*) = (*if n* = *0 then fps-const c else 0*)
by (*intro fps-ext*) (*simp add*: *fps-shift-def*)

lemma *fps-shift-numeral*: *fps-shift n* (*numeral c*) = (*if n* = *0 then numeral c else 0*)
by (*simp add*: *numeral-fps-const fps-shift-fps-const*)

lemma *fps-shift-X-power* [*simp*]:

$n \leq m \Longrightarrow$ *fps-shift n* $(X \ \hat{}\ m) = (X \ \hat{}\ (m - n)$ ::'*a*::*comm-ring-1 fps*)
by (*intro fps-ext*) (*auto simp*: *fps-shift-def*)

lemma *fps-shift-times-X-power*:
$n \leq$ *subdegree f* $\Longrightarrow$ *fps-shift n f* $*$ $X \ \hat{}\ n$ = (*f* :: '*a* :: *comm-ring-1 fps*)
by (*intro fps-ext*) (*auto simp*: *X-power-mult-right-nth nth-less-subdegree-zero*)

lemma *fps-shift-times-X-power'* [*simp*]:
fps-shift n (*f* $*$ $X\hat{}n$) = (*f* :: '*a* :: *comm-ring-1 fps*)
by (*intro fps-ext*) (*auto simp*: *X-power-mult-right-nth nth-less-subdegree-zero*)

lemma *fps-shift-times-X-power''*:
$m \leq n \Longrightarrow$ *fps-shift n* (*f* $*$ $X\hat{}m$) = *fps-shift* $(n - m)$ (*f* :: '*a* :: *comm-ring-1 fps*)
by (*intro fps-ext*) (*auto simp*: *X-power-mult-right-nth nth-less-subdegree-zero*)

lemma *fps-shift-subdegree* [*simp*]:
$n \leq$ *subdegree f* $\Longrightarrow$ *subdegree* (*fps-shift n f*) = *subdegree* (*f* :: '*a* :: *comm-ring-1 fps*) $- n$
by (*cases f* = *0*) (*force intro*: *nth-less-subdegree-zero subdegreeI*)+

lemma *subdegree-decompose*:
f = *fps-shift* (*subdegree f*) *f* $*$ $X \ \hat{}$ *subdegree* (*f* :: ('*a* :: *comm-ring-1*) *fps*)
by (*rule fps-ext*) (*auto simp*: *X-power-mult-right-nth*)

lemma *subdegree-decompose'*:
$n \leq$ *subdegree* (*f* :: ('*a* :: *comm-ring-1*) *fps*) $\Longrightarrow$ *f* = *fps-shift n f* $*$ $X\hat{}n$
by (*rule fps-ext*) (*auto simp*: *X-power-mult-right-nth intro*!: *nth-less-subdegree-zero*)

lemma *fps-shift-fps-shift*:
fps-shift $(m + n)$ *f* = *fps-shift m* (*fps-shift n f*)
by (*rule fps-ext*) (*simp add*: *add-ac*)

lemma *fps-shift-add*:
fps-shift n $(f + g)$ = *fps-shift n f* + *fps-shift n g*
by (*simp add*: *fps-eq-iff*)

lemma *fps-shift-mult*:
assumes $n \leq$ *subdegree* (*g* :: '*b* :: {*comm-ring-1*} *fps*)
shows *fps-shift n* $(h*g)$ = $h *$ *fps-shift n g*
proof −
from *assms* **have** *g* = *fps-shift n g* $*$ $X\hat{}n$ **by** (*rule subdegree-decompose'*)
also have $h * ...$ = ($h *$ *fps-shift n g*) $*$ $X\hat{}n$ **by** *simp*
also have *fps-shift n ...* = $h *$ *fps-shift n g* **by** *simp*
finally show *?thesis* **.**
qed

lemma *fps-shift-mult-right*:
assumes $n \leq$ *subdegree* (*g* :: '*b* :: {*comm-ring-1*} *fps*)

shows *fps-shift n* $(g{*}h) = h * \textit{fps-shift } n\ g$
by (*subst mult.commute*, *subst fps-shift-mult*) (*simp-all add*: *assms*)

lemma *nth-subdegree-zero-iff* [*simp*]: $f\ \$\ \textit{subdegree } f = 0 \longleftrightarrow f = 0$
by (*cases* $f = 0$) *auto*

lemma *fps-shift-subdegree-zero-iff* [*simp*]:
fps-shift (*subdegree* f) $f = 0 \longleftrightarrow f = 0$
by (*subst* (*1*) *nth-subdegree-zero-iff*[*symmetric*], *cases* $f = 0$)
(*simp-all del*: *nth-subdegree-zero-iff*)

definition *fps-cutoff n f* = *Abs-fps* ($\lambda i.$ *if* $i < n$ *then* $f\$i$ *else* *0*)

lemma *fps-cutoff-nth* [*simp*]: *fps-cutoff n f* $\$\ i$ = (*if* $i < n$ *then* $f\$i$ *else* *0*)
unfolding *fps-cutoff-def* **by** *simp*

lemma *fps-cutoff-zero-iff*: *fps-cutoff n f* $= 0 \longleftrightarrow (f = 0 \lor n \leq \textit{subdegree } f)$
proof
assume *A*: *fps-cutoff n f* = *0*
thus $f = 0 \lor n \leq \textit{subdegree } f$
proof (*cases* $f = 0$)
assume $f \neq 0$
with *A* **have** $n \leq \textit{subdegree } f$
by (*intro subdegree-geI*) (*auto simp*: *fps-eq-iff split*: *if-split-asm*)
thus *?thesis* **..**
qed *simp*
qed (*auto simp*: *fps-eq-iff intro*: *nth-less-subdegree-zero*)

lemma *fps-cutoff-0* [*simp*]: *fps-cutoff 0 f* = *0*
by (*simp add*: *fps-eq-iff*)

lemma *fps-cutoff-zero* [*simp*]: *fps-cutoff n 0* = *0*
by (*simp add*: *fps-eq-iff*)

lemma *fps-cutoff-one*: *fps-cutoff n 1* = (*if* $n = 0$ *then 0 else 1*)
by (*simp add*: *fps-eq-iff*)

lemma *fps-cutoff-fps-const*: *fps-cutoff n* (*fps-const c*) = (*if* $n = 0$ *then 0 else fps-const c*)
by (*simp add*: *fps-eq-iff*)

lemma *fps-cutoff-numeral*: *fps-cutoff n* (*numeral c*) = (*if* $n = 0$ *then 0 else numeral c*)
by (*simp add*: *numeral-fps-const fps-cutoff-fps-const*)

lemma *fps-shift-cutoff*:
fps-shift n (f :: ('*a* :: *comm-ring-1*) *fps*) $* X\hat{\ }n$ + *fps-cutoff n f* = f
by (*simp add*: *fps-eq-iff X-power-mult-right-nth*)

43.9 Formal Power series form a metric space

definition (**in** *dist*) *ball x r* = {*y*. *dist y x* < *r*}

instantiation *fps* :: (*comm-ring-1*) *dist*
begin

definition
dist-fps-def: *dist* (*a* :: ′*a fps*) *b* = (*if a* = *b then 0 else inverse* (*2* ˆ *subdegree* (*a* − *b*)))

lemma *dist-fps-ge0*: *dist* (*a* :: ′*a fps*) *b* ≥ *0*
by (*simp add*: *dist-fps-def*)

lemma *dist-fps-sym*: *dist* (*a* :: ′*a fps*) *b* = *dist b a*
by (*simp add*: *dist-fps-def*)

instance ..

end

instantiation *fps* :: (*comm-ring-1*) *metric-space*
begin

definition *uniformity-fps-def* [*code del*]:
(*uniformity* :: (′*a fps* × ′*a fps*) *filter*) = (*INF e*:{*0* <..}. *principal* {(*x*, *y*). *dist x y* < *e*})

definition *open-fps-def* ′ [*code del*]:
open (*U* :: ′*a fps set*) ⟷ (∀ *x*∈*U*. *eventually* (λ(*x*′, *y*). *x*′ = *x* ⟶ *y* ∈ *U*) *uniformity*)

instance
proof
show *th*: *dist a b* = *0* ⟷ *a* = *b* **for** *a b* :: ′*a fps*
by (*simp add*: *dist-fps-def split*: *if-split-asm*)
then have *th*′[*simp*]: *dist a a* = *0* **for** *a* :: ′*a fps* **by** *simp*

fix *a b c* :: ′*a fps*
consider *a* = *b* | *c* = *a* ∨ *c* = *b* | *a* ≠ *b a* ≠ *c b* ≠ *c* **by** *blast*
then show *dist a b* ≤ *dist a c* + *dist b c*
proof *cases*
case *1*
then show *?thesis* **by** (*simp add*: *dist-fps-def*)
next
case *2*
then show *?thesis*
by (*cases c* = *a*) (*simp-all add*: *th dist-fps-sym*)
next
case *neq*: *3*

have *False* **if** *dist a b* > *dist a c* + *dist b c*
proof −
let *?n* = *subdegree* (*a* − *b*)
from *neq* **have** *dist a b* > *0 dist b c* > *0* **and** *dist a c* > *0* **by** (*simp-all add*: *dist-fps-def*)
with *that* **have** *dist a b* > *dist a c* **and** *dist a b* > *dist b c* **by** *simp-all*
with *neq* **have** *?n* < *subdegree* (*a* − *c*) **and** *?n* < *subdegree* (*b* − *c*)
by (*simp-all add*: *dist-fps-def field-simps*)
hence (*a* − *c*) $ *?n* = *0* **and** (*b* − *c*) $ *?n* = *0*
by (*simp-all only*: *nth-less-subdegree-zero*)
hence (*a* − *b*) $ *?n* = *0* **by** *simp*
moreover from *neq* **have** (*a* − *b*) $ *?n* ≠ *0* **by** (*intro nth-subdegree-nonzero*) *simp-all*
ultimately show *False* **by** *contradiction*
qed
thus *?thesis* **by** (*auto simp add*: *not-le*[*symmetric*])
qed
qed (*rule open-fps-def′ uniformity-fps-def*)+

end

declare *uniformity-Abort*[**where** *′a=′a* :: *comm-ring-1 fps*, *code*]

lemma *open-fps-def*: *open* (*S* :: *′a::comm-ring-1 fps set*) = (∀ *a* ∈ *S*. ∃ *r*. *r* >*0* ∧ *ball a r* ⊆ *S*)
unfolding *open-dist ball-def subset-eq* **by** *simp*

The infinite sums and justification of the notation in textbooks.

lemma *reals-power-lt-ex*:
fixes *x y* :: *real*
assumes *xp*: *x* > *0*
and *y1*: *y* > *1*
shows ∃ *k*>*0*. (*1*/*y*)^*k* < *x*
proof −
have *yp*: *y* > *0*
using *y1* **by** *simp*
from *reals-Archimedean2*[*of max 0* (− *log y x*) + *1*]
obtain *k* :: *nat* **where** *k*: *real k* > *max 0* (− *log y x*) + *1*
by *blast*
from *k* **have** *kp*: *k* > *0*
by *simp*
from *k* **have** *real k* > − *log y x*
by *simp*
then have *ln y* ∗ *real k* > − *ln x*
unfolding *log-def*
using *ln-gt-zero-iff*[*OF yp*] *y1*
by (*simp add*: *minus-divide-left field-simps del*: *minus-divide-left*[*symmetric*])
then have *ln y* ∗ *real k* + *ln x* > *0*
by *simp*

```
  then have exp (real k * ln y + ln x) > exp 0
    by (simp add: ac-simps)
  then have y ^ k * x > 1
    unfolding exp-zero exp-add exp-real-of-nat-mult exp-ln [OF xp] exp-ln [OF yp]
    by simp
  then have x > (1 / y)^k using yp
    by (simp add: field-simps)
  then show ?thesis
    using kp by blast
qed

lemma fps-sum-rep-nth: (setsum (λi. fps-const(a$i)*X^i) {0..m})$n =
    (if n ≤ m then a$n else 0::'a::comm-ring-1)
  apply (auto simp add: fps-setsum-nth cond-value-iff cong del: if-weak-cong)
  apply (simp add: setsum.delta')
  done

lemma fps-notation: (λn. setsum (λi. fps-const(a$i) * X^i) {0..n}) ⟶ a
  (is ?s ⟶ a)
proof -
  have ∃n0. ∀n ≥ n0. dist (?s n) a < r if r > 0 for r
  proof -
    obtain n0 where n0: (1/2)^n0 < r n0 > 0
      using reals-power-lt-ex[OF ‹r > 0›, of 2] by auto
    show ?thesis
    proof -
      have dist (?s n) a < r if nn0: n ≥ n0 for n
      proof -
        from that have thnn0: (1/2)^n ≤ (1/2 :: real)^n0
          by (simp add: divide-simps)
        show ?thesis
        proof (cases ?s n = a)
          case True
          then show ?thesis
            unfolding dist-eq-0-iff[of ?s n a, symmetric]
            using ‹r > 0› by (simp del: dist-eq-0-iff)
        next
          case False
          from False have dth: dist (?s n) a = (1/2)^subdegree (?s n - a)
            by (simp add: dist-fps-def field-simps)
          from False have kn: subdegree (?s n - a) > n
            by (intro subdegree-greaterI) (simp-all add: fps-sum-rep-nth)
          then have dist (?s n) a < (1/2)^n
            by (simp add: field-simps dist-fps-def)
          also have ... ≤ (1/2)^n0
            using nn0 by (simp add: divide-simps)
          also have ... < r
            using n0 by simp
          finally show ?thesis .
```

```
        qed
      qed
      then show ?thesis by blast
    qed
  qed
  then show ?thesis
    unfolding lim-sequentially by blast
qed
```

43.10 Inverses of formal power series

```
declare setsum.cong[fundef-cong]

instantiation fps :: ({comm-monoid-add,inverse,times,uminus}) inverse
begin

fun natfun-inverse:: 'a fps ⇒ nat ⇒ 'a
where
  natfun-inverse f 0 = inverse (f$0)
| natfun-inverse f n = − inverse (f$0) ∗ setsum (λi. f$i ∗ natfun-inverse f (n −
i)) {1..n}

definition fps-inverse-def: inverse f = (if f $ 0 = 0 then 0 else Abs-fps (natfun-inverse
f))

definition fps-divide-def:
  f div g = (if g = 0 then 0 else
     let n = subdegree g; h = fps-shift n g
     in  fps-shift n (f ∗ inverse h))

instance ..

end

lemma fps-inverse-zero [simp]:
  inverse (0 :: 'a::{comm-monoid-add,inverse,times,uminus} fps) = 0
  by (simp add: fps-ext fps-inverse-def)

lemma fps-inverse-one [simp]: inverse (1 :: 'a::{division-ring,zero-neq-one} fps)
= 1
  apply (auto simp add: expand-fps-eq fps-inverse-def)
  apply (case-tac n)
  apply auto
  done

lemma inverse-mult-eq-1 [intro]:
  assumes f0: f$0 ≠ (0::'a::field)
  shows inverse f ∗ f = 1
proof −
```

have *c*: *inverse f* ∗ *f* = *f* ∗ *inverse f*
by (*simp add*: *mult.commute*)
from *f0* **have** *ifn*: $\bigwedge$*n*. *inverse f* $ *n* = *natfun-inverse f n*
by (*simp add*: *fps-inverse-def*)
from *f0* **have** *th0*: (*inverse f* ∗ *f*) $ *0* = *1*
by (*simp add*: *fps-mult-nth fps-inverse-def*)
have (*inverse f* ∗ *f*)$*n* = *0* **if** *np*: *n* > *0* **for** *n*
proof −
from *np* **have** *eq*: {*0*..*n*} = {*0*} ∪ {*1* .. *n*}
by *auto*
have *d*: {*0*} ∩ {*1* .. *n*} = {}
by *auto*
from *f0 np* **have** *th0*: − (*inverse f* $ *n*) =
(*setsum* (λ*i*. *f*$*i* ∗ *natfun-inverse f* (*n* − *i*)) {*1*..*n*}) / (*f*$*0*)
by (*cases n*) (*simp-all add*: *divide-inverse fps-inverse-def*)
from *th0*[*symmetric*, *unfolded nonzero-divide-eq-eq*[*OF f0*]]
have *th1*: *setsum* (λ*i*. *f*$*i* ∗ *natfun-inverse f* (*n* − *i*)) {*1*..*n*} = − (*f*$*0*) ∗ (*inverse f*)$*n*
by (*simp add*: *field-simps*)
have (*f* ∗ *inverse f*) $ *n* = (∑ *i* = *0*..*n*. *f* $*i* ∗ *natfun-inverse f* (*n* − *i*))
unfolding *fps-mult-nth ifn* **..**
also have ... = *f*$*0* ∗ *natfun-inverse f n* + (∑ *i* = *1*..*n*. *f*$*i* ∗ *natfun-inverse f* (*n*−*i*))
by (*simp add*: *eq*)
also have ... = *0*
unfolding *th1 ifn* **by** *simp*
finally show *?thesis* **unfolding** *c* **.**
qed
with *th0* **show** *?thesis*
by (*simp add*: *fps-eq-iff*)
qed

lemma *fps-inverse-0-iff*[*simp*]: (*inverse f*) $ *0* = (*0*::'*a*::*division-ring*) ⟷ *f* $ *0* = *0*
by (*simp add*: *fps-inverse-def nonzero-imp-inverse-nonzero*)

lemma *fps-inverse-nth-0* [*simp*]: *inverse f* $ *0* = *inverse* (*f* $ *0* :: '*a* :: *division-ring*)
by (*simp add*: *fps-inverse-def*)

lemma *fps-inverse-eq-0-iff*[*simp*]: *inverse f* = (*0*:: ('*a*::*division-ring*) *fps*) ⟷ *f* $ *0* = *0*
proof
assume *A*: *inverse f* = *0*
have *0* = *inverse f* $ *0* **by** (*subst A*) *simp*
thus *f* $ *0* = *0* **by** *simp*
qed (*simp add*: *fps-inverse-def*)

lemma *fps-inverse-idempotent*[*intro*, *simp*]:
assumes *f0*: *f*$*0* ≠ (*0*::'*a*::*field*)

shows *inverse* (*inverse f*) = *f*
proof −
from *f0* **have** *if0*: *inverse f* $ *0* ≠ *0* **by** *simp*
from *inverse-mult-eq-1*[*OF f0*] *inverse-mult-eq-1*[*OF if0*]
have *inverse f* ∗ *f* = *inverse f* ∗ *inverse* (*inverse f*)
by (*simp add*: *ac-simps*)
then show *?thesis*
using *f0* **unfolding** *mult-cancel-left* **by** *simp*
qed

lemma *fps-inverse-unique*:
assumes *fg*: (*f* :: *'a* :: *field fps*) ∗ *g* = *1*
shows *inverse f* = *g*
proof −
have *f0*: *f* $ *0* ≠ *0*
proof
assume *f* $ *0* = *0*
hence *0* = (*f* ∗ *g*) $ *0* **by** *simp*
also from *fg* **have** (*f* ∗ *g*) $ *0* = *1* **by** *simp*
finally show *False* **by** *simp*
qed
from *inverse-mult-eq-1*[*OF this*] *fg*
have *th0*: *inverse f* ∗ *f* = *g* ∗ *f*
by (*simp add*: *ac-simps*)
then show *?thesis*
using *f0*
unfolding *mult-cancel-right*
by (*auto simp add*: *expand-fps-eq*)
qed

lemma *setsum-zero-lemma*:
fixes *n*::*nat*
assumes *0* < *n*
shows (∑ *i* = *0*..*n*. *if n* = *i then 1 else if n* − *i* = *1 then* − *1 else 0*) = (*0*::*'a*::*field*)
proof −
let *?f* = λ*i*. *if n* = *i then 1 else if n* − *i* = *1 then* − *1 else 0*
let *?g* = λ*i*. *if i* = *n then 1 else if i* = *n* − *1 then* − *1 else 0*
let *?h* = λ*i*. *if i*=*n* − *1 then* − *1 else 0*
have *th1*: *setsum ?f* {*0*..*n*} = *setsum ?g* {*0*..*n*}
by (*rule setsum.cong*) *auto*
have *th2*: *setsum ?g* {*0*..*n* − *1*} = *setsum ?h* {*0*..*n* − *1*}
apply (*rule setsum.cong*)
using *assms*
apply *auto*
done
have *eq*: {*0* .. *n*} = {*0*.. *n* − *1*} ∪ {*n*}
by *auto*
from *assms* **have** *d*: {*0*.. *n* − *1*} ∩ {*n*} = {}

by *auto*
have *f*: *finite* $\{0.. n - 1\}$ *finite* $\{n\}$
by *auto*
show *?thesis*
unfolding *th1*
apply (*simp add*: *setsum.union-disjoint*[*OF f d*, *unfolded eq*[*symmetric*]] *del*: *One-nat-def*)
unfolding *th2*
apply (*simp add*: *setsum.delta*)
done
qed

lemma *fps-inverse-mult*: *inverse* ($f * g$ :: *'a::field fps*) = *inverse f* $*$ *inverse g*
proof (*cases* $f\$0 = 0 \vee g\$0 = 0$)
assume $\neg(f\$0 = 0 \vee g\$0 = 0)$
hence [*simp*]: $f\$0 \neq 0$ $g\$0 \neq 0$ **by** *simp-all*
show *?thesis*
proof (*rule fps-inverse-unique*)
have $f * g * ($*inverse f* $*$ *inverse g*$) = ($*inverse f* $* f) * ($*inverse g* $* g)$ **by** *simp*
also have ... = *1* **by** (*subst* (*1 2*) *inverse-mult-eq-1*) *simp-all*
finally show $f * g * ($*inverse f* $*$ *inverse g*$) = 1$ **.**
qed
next
assume *A*: $f\$0 = 0 \vee g\$0 = 0$
hence *inverse* $(f * g) = 0$ **by** *simp*
also from *A* **have** ... = *inverse f* $*$ *inverse g* **by** *auto*
finally show *inverse* $(f * g)$ = *inverse f* $*$ *inverse g* **.**
qed

lemma *fps-inverse-gp*: *inverse* (*Abs-fps*($\lambda n.$ (*1::'a::field*))) =
Abs-fps ($\lambda n.$ *if n= 0 then 1 else if n=1 then* $- 1$ *else 0*)
apply (*rule fps-inverse-unique*)
apply (*simp-all add*: *fps-eq-iff fps-mult-nth setsum-zero-lemma*)
done

lemma *subdegree-inverse* [*simp*]: *subdegree* (*inverse* (*f::'a::field fps*)) = *0*
proof (*cases* $f\$0 = 0$)
assume *nz*: $f\$0 \neq 0$
hence *subdegree* (*inverse f*) + *subdegree f* = *subdegree* (*inverse f* $* f$)
by (*subst subdegree-mult*) *auto*
also from *nz* **have** *subdegree f* = *0* **by** (*simp add*: *subdegree-eq-0-iff*)
also from *nz* **have** *inverse f* $* f = 1$ **by** (*rule inverse-mult-eq-1*)
finally show *subdegree* (*inverse f*) = *0* **by** *simp*
qed (*simp-all add*: *fps-inverse-def*)

lemma *fps-is-unit-iff* [*simp*]: (*f* :: *'a* :: *field fps*) *dvd 1* $\longleftrightarrow f \$ 0 \neq 0$
proof

```
  assume f dvd 1
  then obtain g where 1 = f * g by (elim dvdE)
  from this[symmetric] have (f*g) $ 0 = 1 by simp
  thus f $ 0 ≠ 0 by auto
next
  assume A: f $ 0 ≠ 0
  thus f dvd 1 by (simp add: inverse-mult-eq-1[OF A, symmetric])
qed

lemma subdegree-eq-0' [simp]: (f :: 'a :: field fps) dvd 1 ⟹ subdegree f = 0
  by simp

lemma fps-unit-dvd [simp]: (f $ 0 :: 'a :: field) ≠ 0 ⟹ f dvd g
  by (rule dvd-trans, subst fps-is-unit-iff) simp-all



instantiation fps :: (field) ring-div
begin

definition fps-mod-def:
  f mod g = (if g = 0 then f else
     let n = subdegree g; h = fps-shift n g
     in  fps-cutoff n (f * inverse h) * h)

lemma fps-mod-eq-zero:
  assumes g ≠ 0 and subdegree f ≥ subdegree g
  shows   f mod g = 0
  using assms by (cases f = 0) (auto simp: fps-cutoff-zero-iff fps-mod-def Let-def)

lemma fps-times-divide-eq:
  assumes g ≠ 0 and subdegree f ≥ subdegree (g :: 'a fps)
  shows   f div g * g = f
proof (cases f = 0)
  assume nz: f ≠ 0
  def n ≡ subdegree g
  def h ≡ fps-shift n g
  from assms have [simp]: h $ 0 ≠ 0 unfolding h-def by (simp add: n-def)

  from assms nz have f div g * g = fps-shift n (f * inverse h) * g
    by (simp add: fps-divide-def Let-def h-def n-def)
  also have ... = fps-shift n (f * inverse h) * X^n * h unfolding h-def n-def
    by (subst subdegree-decompose[of g]) simp
  also have fps-shift n (f * inverse h) * X^n = f * inverse h
    by (rule fps-shift-times-X-power) (simp-all add: nz assms n-def)
  also have ... * h = f * (inverse h * h) by simp
  also have inverse h * h = 1 by (rule inverse-mult-eq-1) simp
  finally show ?thesis by simp
qed (simp-all add: fps-divide-def Let-def)
```

lemma
  assumes *g$0 ≠ 0*
  shows *fps-divide-unit*: *f div g = f ∗ inverse g* **and** *fps-mod-unit* [*simp*]: *f mod g = 0*
proof −
  from *assms* **have** [*simp*]: *subdegree g = 0* **by** (*simp add*: *subdegree-eq-0-iff*)
  from *assms* **show** *f div g = f ∗ inverse g*
    by (*auto simp*: *fps-divide-def Let-def subdegree-eq-0-iff*)
  from *assms* **show** *f mod g = 0* **by** (*intro fps-mod-eq-zero*) *auto*
qed

context
begin
private lemma *fps-divide-cancel-aux1*:
  assumes *h$0 ≠ (0 :: 'a :: field)*
  shows (*h ∗ f*) *div* (*h ∗ g*) = *f div g*
proof (*cases g = 0*)
  assume *g ≠ 0*
  from *assms* **have** *h ≠ 0* **by** *auto*
  note *nz* [*simp*] = ‹*g ≠ 0*› ‹*h ≠ 0*›
  from *assms* **have** [*simp*]: *subdegree h = 0* **by** (*simp add*: *subdegree-eq-0-iff*)

  have (*h ∗ f*) *div* (*h ∗ g*) =
        fps-shift (*subdegree g*) (*h ∗ f ∗ inverse* (*fps-shift* (*subdegree g*) (*h∗g*)))
    by (*simp add*: *fps-divide-def Let-def*)
  also have *h ∗ f ∗ inverse* (*fps-shift* (*subdegree g*) (*h∗g*)) =
            (*inverse h ∗ h*) *∗ f ∗ inverse* (*fps-shift* (*subdegree g*) *g*)
    by (*subst fps-shift-mult*) (*simp-all add*: *algebra-simps fps-inverse-mult*)
  also from *assms* **have** *inverse h ∗ h = 1* **by** (*rule inverse-mult-eq-1*)
  finally show (*h ∗ f*) *div* (*h ∗ g*) = *f div g* **by** (*simp-all add*: *fps-divide-def Let-def*)
qed (*simp-all add*: *fps-divide-def*)

private lemma *fps-divide-cancel-aux2*:
  (*f ∗ Xˆm*) *div* (*g ∗ Xˆm*) = *f div* (*g :: 'a :: field fps*)
proof (*cases g = 0*)
  assume [*simp*]: *g ≠ 0*
  have (*f ∗ Xˆm*) *div* (*g ∗ Xˆm*) =
            fps-shift (*subdegree g + m*) (*f∗inverse* (*fps-shift* (*subdegree g + m*) (*g∗Xˆm*))*∗Xˆm*)
    by (*simp add*: *fps-divide-def Let-def algebra-simps*)
  also have ... = *f div g*
    by (*simp add*: *fps-shift-times-X-power''* *fps-divide-def Let-def*)
  finally show *?thesis* **.**
qed (*simp-all add*: *fps-divide-def*)

instance proof
  fix *f g :: 'a fps*

def *n* ≡ *subdegree g*
def *h* ≡ *fps-shift n g*

show *f div g* ∗ *g* + *f mod g* = *f*
proof (*cases g* = *0* ∨ *f* = *0*)
assume ¬(*g* = *0* ∨ *f* = *0*)
hence *nz* [*simp*]: *f* ≠ *0 g* ≠ *0* **by** *simp-all*
show *?thesis*
proof (*rule disjE*[*OF le-less-linear*])
assume *subdegree f* ≥ *subdegree g*
with *nz* **show** *?thesis* **by** (*simp add*: *fps-mod-eq-zero fps-times-divide-eq*)
next
assume *subdegree f* < *subdegree g*
have *g-decomp*: *g* = *h* ∗ *X* ˆ*n* **unfolding** *h-def n-def* **by** (*rule subdegree-decompose*)
have *f div g* ∗ *g* + *f mod g* =
fps-shift n (*f* ∗ *inverse h*) ∗ *g* + *fps-cutoff n* (*f* ∗ *inverse h*) ∗ *h*
by (*simp add*: *fps-mod-def fps-divide-def Let-def n-def h-def*)
also have ... = *h* ∗ (*fps-shift n* (*f* ∗ *inverse h*) ∗ *X* ˆ*n* + *fps-cutoff n* (*f* ∗ *inverse h*))
by (*subst g-decomp*) (*simp add*: *algebra-simps*)
also have ... = *f* ∗ (*inverse h* ∗ *h*)
by (*subst fps-shift-cutoff*) *simp*
also have *inverse h* ∗ *h* = *1* **by** (*rule inverse-mult-eq-1*) (*simp add*: *h-def n-def*)
finally show *?thesis* **by** *simp*
qed
qed (*auto simp*: *fps-mod-def fps-divide-def Let-def*)
next

fix *f g h* :: *'a fps*
assume *h* ≠ *0*
show (*h* ∗ *f*) *div* (*h* ∗ *g*) = *f div g*
proof −
def *m* ≡ *subdegree h*
def *h'* ≡ *fps-shift m h*
have *h-decomp*: *h* = *h'* ∗ *X* ˆ *m* **unfolding** *h'-def m-def* **by** (*rule subdegree-decompose*)
from ‹*h* ≠ *0*› **have** [*simp*]: *h'*$*0* ≠ *0* **by** (*simp add*: *h'-def m-def*)
have (*h* ∗ *f*) *div* (*h* ∗ *g*) = (*h'* ∗ *f* ∗ *X* ˆ*m*) *div* (*h'* ∗ *g* ∗ *X* ˆ*m*)
by (*simp add*: *h-decomp algebra-simps*)
also have ... = *f div g* **by** (*simp add*: *fps-divide-cancel-aux1 fps-divide-cancel-aux2*)
finally show *?thesis* **.**
qed

next
fix *f g h* :: *'a fps*
assume [*simp*]: *h* ≠ *0*
def *n* ≡ *subdegree h*
def *h'* ≡ *fps-shift n h*
note *dfs* = *n-def h'-def*

have $(f + g * h)$ *div h* = *fps-shift n* $(f * inverse\ h')$ + *fps-shift n* $(g * (h * inverse\ h'))$
  by (*simp add*: *fps-divide-def Let-def dfs*[*symmetric*] *algebra-simps fps-shift-add*)
also have $h * inverse\ h' = (inverse\ h' * h') * X\hat{}n$
  by (*subst subdegree-decompose*) (*simp-all add*: *dfs*)
also have ... = $X\hat{}n$ **by** (*subst inverse-mult-eq-1*) (*simp-all add*: *dfs*)
also have *fps-shift n* $(g * X\hat{}n) = g$ **by** *simp*
also have *fps-shift n* $(f * inverse\ h')$ = *f div h*
  by (*simp add*: *fps-divide-def Let-def dfs*)
finally show $(f + g * h)$ *div h* = *g* + *f div h* **by** *simp*
qed (*auto simp*: *fps-divide-def fps-mod-def Let-def*)

end
end

lemma *subdegree-mod*:
assumes $f \neq 0$ *subdegree f* < *subdegree g*
shows *subdegree* (*f mod g*) = *subdegree f*
proof (*cases f div g* $* g = 0$)
assume *f div g* $* g \neq 0$
hence [*simp*]: *f div g* $\neq 0$ $g \neq 0$ **by** *auto*
from *mod-div-equality*[*of f g*] **have** *f mod g* = *f* − *f div g* $* g$ **by** (*simp add*: *algebra-simps*)
also from *assms* **have** *subdegree* ... = *subdegree f*
  by (*intro subdegree-diff-eq1*) *simp-all*
finally show *?thesis* **.**
next
assume *zero*: *f div g* $* g = 0$
from *mod-div-equality*[*of f g*] **have** *f mod g* = *f* − *f div g* $* g$ **by** (*simp add*: *algebra-simps*)
also note *zero*
finally show *?thesis* **by** *simp*
qed

lemma *fps-divide-nth-0* [*simp*]: *g* \$ *0* $\neq 0 \Longrightarrow$ (*f div g*) \$ *0* = *f* \$ *0* / (*g* \$ *0* :: -:: *field*)
by (*simp add*: *fps-divide-unit divide-inverse*)

lemma *dvd-imp-subdegree-le*:
(*f* :: *'a* :: *idom fps*) *dvd g* $\Longrightarrow g \neq 0 \Longrightarrow$ *subdegree f* $\leq$ *subdegree g*
by (*auto elim*: *dvdE*)

lemma *fps-dvd-iff*:
assumes (*f* :: *'a* :: *field fps*) $\neq 0$ $g \neq 0$
shows *f dvd g* $\longleftrightarrow$ *subdegree f* $\leq$ *subdegree g*
proof
assume *subdegree f* $\leq$ *subdegree g*
with *assms* **have** *g mod f* = *0*

```
    by (simp add: fps-mod-def Let-def fps-cutoff-zero-iff)
  thus f dvd g by (simp add: dvd-eq-mod-eq-0)
qed (simp add: assms dvd-imp-subdegree-le)

lemma fps-const-inverse: inverse (fps-const (a::'a::field)) = fps-const (inverse a)
  by (cases a ≠ 0, rule fps-inverse-unique) (auto simp: fps-eq-iff)

lemma fps-const-divide: fps-const (x :: - :: field) / fps-const y = fps-const (x / y)
  by (cases y = 0) (simp-all add: fps-divide-unit fps-const-inverse divide-inverse)

lemma inverse-fps-numeral:
  inverse (numeral n :: ('a :: field-char-0) fps) = fps-const (inverse (numeral n))
  by (intro fps-inverse-unique fps-ext) (simp-all add: fps-numeral-nth)


instantiation fps :: (field) normalization-semidom
begin

definition fps-unit-factor-def [simp]:
  unit-factor f = fps-shift (subdegree f) f

definition fps-normalize-def [simp]:
  normalize f = (if f = 0 then 0 else X ^ subdegree f)

instance proof
  fix f :: 'a fps
  show unit-factor f * normalize f = f
    by (simp add: fps-shift-times-X-power)
next
  fix f g :: 'a fps
  show unit-factor (f * g) = unit-factor f * unit-factor g
  proof (cases f = 0 ∨ g = 0)
    assume ¬(f = 0 ∨ g = 0)
    thus unit-factor (f * g) = unit-factor f * unit-factor g
    unfolding fps-unit-factor-def
      by (auto simp: fps-shift-fps-shift fps-shift-mult fps-shift-mult-right)
  qed auto
qed auto

end

instance fps :: (field) algebraic-semidom ..
```

43.11 Formal power series form a Euclidean ring

```
instantiation fps :: (field) euclidean-ring
begin
```

definition *fps-euclidean-size-def*:
  euclidean-size f = (*if f* = *0 then 0 else 2* ˆ *subdegree f*)

instance proof
  fix *f g* :: *'a fps* **assume** [*simp*]: *g* ≠ *0*
  show *euclidean-size f* ≤ *euclidean-size* (*f* ∗ *g*)
    by (*cases f* = *0*) (*auto simp*: *fps-euclidean-size-def*)
  show *euclidean-size* (*f mod g*) < *euclidean-size g*
    apply (*cases f* = *0*, *simp add*: *fps-euclidean-size-def*)
    apply (*rule disjE*[*OF le-less-linear*[*of subdegree g subdegree f*]])
    apply (*simp-all add*: *fps-mod-eq-zero fps-euclidean-size-def subdegree-mod*)
    done
qed (*simp-all add*: *fps-euclidean-size-def*)

end

instantiation *fps* :: (*field*) *euclidean-ring-gcd*
begin
definition *fps-gcd-def*: (*gcd* :: *'a fps* ⇒ *-*) = *gcd-eucl*
definition *fps-lcm-def*: (*lcm* :: *'a fps* ⇒ *-*) = *lcm-eucl*
definition *fps-Gcd-def*: (*Gcd* :: *'a fps set* ⇒ *-*) = *Gcd-eucl*
definition *fps-Lcm-def*: (*Lcm* :: *'a fps set* ⇒ *-*) = *Lcm-eucl*
instance by *standard* (*simp-all add*: *fps-gcd-def fps-lcm-def fps-Gcd-def fps-Lcm-def*)
end

lemma *fps-gcd*:
  assumes [*simp*]: *f* ≠ *0 g* ≠ *0*
  shows *gcd f g* = *X* ˆ *min* (*subdegree f*) (*subdegree g*)
proof −
  let *?m* = *min* (*subdegree f*) (*subdegree g*)
  show *gcd f g* = *X* ˆ *?m*
  proof (*rule sym*, *rule gcdI*)
    fix *d* **assume** *d dvd f d dvd g*
    thus *d dvd X* ˆ *?m* **by** (*cases d* = *0*) (*auto simp*: *fps-dvd-iff*)
  qed (*simp-all add*: *fps-dvd-iff*)
qed

lemma *fps-gcd-altdef*: *gcd* (*f* :: *'a* :: *field fps*) *g* =
  (*if f* = *0* ∧ *g* = *0 then 0 else*
   if f = *0 then X* ˆ *subdegree g else*
   if g = *0 then X* ˆ *subdegree f else*
     X ˆ *min* (*subdegree f*) (*subdegree g*))
  by (*simp add*: *fps-gcd*)

lemma *fps-lcm*:
  assumes [*simp*]: *f* ≠ *0 g* ≠ *0*
  shows *lcm f g* = *X* ˆ *max* (*subdegree f*) (*subdegree g*)
proof −

let *?m = max (subdegree f) (subdegree g)*
show *lcm f g = X ˆ ?m*
proof (*rule sym*, *rule lcmI*)
fix *d* **assume** *f dvd d g dvd d*
thus *X ˆ ?m dvd d* **by** (*cases d = 0*) (*auto simp*: *fps-dvd-iff*)
qed (*simp-all add*: *fps-dvd-iff*)
qed

lemma *fps-lcm-altdef*: *lcm (f :: 'a :: field fps) g =*
(*if f = 0 ∨ g = 0 then 0 else X ˆ max (subdegree f) (subdegree g)*)
by (*simp add*: *fps-lcm*)

lemma *fps-Gcd*:
assumes *A − {0} ≠ {}*
shows *Gcd A = X ˆ (INF f:A−{0}. subdegree f)*
proof (*rule sym*, *rule GcdI*)
fix *f* **assume** *f ∈ A*
thus *X ˆ (INF f:A − {0}. subdegree f) dvd f*
by (*cases f = 0*) (*auto simp*: *fps-dvd-iff intro*!: *cINF-lower*)
next
fix *d* **assume** *d*: *⋀f. f ∈ A ⟹ d dvd f*
from *assms* **obtain** *f* **where** *f ∈ A − {0}* **by** *auto*
with *d[of f]* **have** [*simp*]: *d ≠ 0* **by** *auto*
from *d assms* **have** *subdegree d ≤ (INF f:A−{0}. subdegree f)*
by (*intro cINF-greatest*) (*auto simp*: *fps-dvd-iff*[*symmetric*])
with *d assms* **show** *d dvd X ˆ (INF f:A−{0}. subdegree f)* **by** (*simp add*: *fps-dvd-iff*)
qed *simp-all*

lemma *fps-Gcd-altdef*: *Gcd (A :: 'a :: field fps set) =*
(*if A ⊆ {0} then 0 else X ˆ (INF f:A−{0}. subdegree f)*)
using *fps-Gcd* **by** *auto*

lemma *fps-Lcm*:
assumes *A ≠ {} 0 ∉ A bdd-above (subdegree'A)*
shows *Lcm A = X ˆ (SUP f:A. subdegree f)*
proof (*rule sym*, *rule LcmI*)
fix *f* **assume** *f ∈ A*
moreover from *assms(3)* **have** *bdd-above (subdegree ' A)* **by** *auto*
ultimately show *f dvd X ˆ (SUP f:A. subdegree f)* **using** *assms(2)*
by (*cases f = 0*) (*auto simp*: *fps-dvd-iff intro*!: *cSUP-upper*)
next
fix *d* **assume** *d*: *⋀f. f ∈ A ⟹ f dvd d*
from *assms* **obtain** *f* **where** *f*: *f ∈ A f ≠ 0* **by** *auto*
show *X ˆ (SUP f:A. subdegree f) dvd d*
proof (*cases d = 0*)
assume *d ≠ 0*
moreover from *d* **have** *⋀f. f ∈ A ⟹ f ≠ 0 ⟹ f dvd d* **by** *blast*
ultimately have *subdegree d ≥ (SUP f:A. subdegree f)* **using** *assms*

by (*intro cSUP-least*) (*auto simp*: *fps-dvd-iff*)
with ‹$d \neq 0$› **show** *?thesis* **by** (*simp add*: *fps-dvd-iff*)
qed *simp-all*
qed *simp-all*

lemma *fps-Lcm-altdef*:
Lcm (*A* :: '*a* :: *field fps set*) =
(*if* $0 \in A \vee \neg$*bdd-above* (*subdegree*'*A*) *then 0 else*
if $A = \{\}$ *then 1 else X* ^ (*SUP f*:*A*. *subdegree f*))
proof (*cases bdd-above* (*subdegree*'*A*))
assume *unbounded*: ¬*bdd-above* (*subdegree*'*A*)
have *Lcm A* = *0*
proof (*rule ccontr*)
assume *Lcm A* ≠ *0*
from *unbounded* **obtain** *f* **where** *f*: *f* ∈ *A subdegree* (*Lcm A*) < *subdegree f*
unfolding *bdd-above-def* **by** (*auto simp*: *not-le*)
moreover from *this* **and** ‹*Lcm A* ≠ *0*› **have** *subdegree f* ≤ *subdegree* (*Lcm A*)
by (*intro dvd-imp-subdegree-le dvd-Lcm*) *simp-all*
ultimately show *False* **by** *simp*
qed
with *unbounded* **show** *?thesis* **by** *simp*
qed (*simp-all add*: *fps-Lcm Lcm-eq-0-I*)

43.12 Formal Derivatives, and the MacLaurin theorem around 0

definition *fps-deriv f* = *Abs-fps* (λ*n*. *of-nat* (*n* + *1*) ∗ *f* $ (*n* + *1*))

lemma *fps-deriv-nth*[*simp*]: *fps-deriv f* $ *n* = *of-nat* (*n* +*1*) ∗ *f* $ (*n* + *1*)
by (*simp add*: *fps-deriv-def*)

lemma *fps-deriv-linear*[*simp*]:
fps-deriv (*fps-const* (*a*::'*a*::*comm-semiring-1*) ∗ *f* + *fps-const b* ∗ *g*) =
fps-const a ∗ *fps-deriv f* + *fps-const b* ∗ *fps-deriv g*
unfolding *fps-eq-iff fps-add-nth fps-const-mult-left fps-deriv-nth* **by** (*simp add*: *field-simps*)

lemma *fps-deriv-mult*[*simp*]:
fixes *f* :: '*a*::*comm-ring-1 fps*
shows *fps-deriv* (*f* ∗ *g*) = *f* ∗ *fps-deriv g* + *fps-deriv f* ∗ *g*
proof −
let *?D* = *fps-deriv*
have (*f* ∗ *?D g* + *?D f* ∗ *g*) $ *n* = *?D* (*f*∗*g*) $ *n* **for** *n*
proof −
let *?Zn* = {*0* ..*n*}
let *?Zn1* = {*0* .. *n* + *1*}
let *?g* = λ*i*. *of-nat* (*i*+*1*) ∗ *g* $ (*i*+*1*) ∗ *f* $ (*n* − *i*) +
of-nat (*i*+*1*)∗ *f* $ (*i*+*1*) ∗ *g* $ (*n* − *i*)
let *?h* = λ*i*. *of-nat i* ∗ *g* $ *i* ∗ *f* $ ((*n*+*1*) − *i*) +

of-nat i∗ f $ i ∗ g $ ((n + 1) − i)
have *s0*: *setsum (λi. of-nat i ∗ f $ i ∗ g $ (n + 1 − i)) ?Zn1 =*
setsum (λi. of-nat (n + 1 − i) ∗ f $ (n + 1 − i) ∗ g $ i) ?Zn1
by (*rule setsum.reindex-bij-witness*[**where** *i=op − (n + 1)* **and** *j=op − (n + 1)*]) *auto*
have *s1*: *setsum (λi. f $ i ∗ g $ (n + 1 − i)) ?Zn1 =*
setsum (λi. f $ (n + 1 − i) ∗ g $ i) ?Zn1
by (*rule setsum.reindex-bij-witness*[**where** *i=op − (n + 1)* **and** *j=op − (n + 1)*]) *auto*
have *(f ∗ ?D g + ?D f ∗ g)$n = (?D g ∗ f + ?D f ∗ g)$n*
by (*simp only*: *mult.commute*)
also have ... = (∑ *i = 0..n. ?g i*)
by (*simp add*: *fps-mult-nth setsum.distrib*[*symmetric*])
also have ... = *setsum ?h {0..n+1}*
by (*rule setsum.reindex-bij-witness-not-neutral*
[**where** *S'={}* **and** *T'={0}* **and** *j=Suc* **and** *i=λi. i − 1*]) *auto*
also have ... = (*fps-deriv (f ∗ g)*) $ *n*
apply (*simp only*: *fps-deriv-nth fps-mult-nth setsum.distrib*)
unfolding *s0 s1*
unfolding *setsum.distrib*[*symmetric*] *setsum-right-distrib*
apply (*rule setsum.cong*)
apply (*auto simp add*: *of-nat-diff field-simps*)
done
finally show *?thesis* **.**
qed
then show *?thesis*
unfolding *fps-eq-iff* **by** *auto*
qed

lemma *fps-deriv-X*[*simp*]: *fps-deriv X = 1*
by (*simp add*: *fps-deriv-def X-def fps-eq-iff*)

lemma *fps-deriv-neg*[*simp*]:
fps-deriv (− (f:: 'a::comm-ring-1 fps)) = − (fps-deriv f)
by (*simp add*: *fps-eq-iff fps-deriv-def*)

lemma *fps-deriv-add*[*simp*]:
fps-deriv ((f:: 'a::comm-ring-1 fps) + g) = fps-deriv f + fps-deriv g
using *fps-deriv-linear*[*of 1 f 1 g*] **by** *simp*

lemma *fps-deriv-sub*[*simp*]:
fps-deriv ((f:: 'a::comm-ring-1 fps) − g) = fps-deriv f − fps-deriv g
using *fps-deriv-add* [*of f − g*] **by** *simp*

lemma *fps-deriv-const*[*simp*]: *fps-deriv (fps-const c) = 0*
by (*simp add*: *fps-ext fps-deriv-def fps-const-def*)

lemma *fps-deriv-mult-const-left*[*simp*]:
fps-deriv (fps-const (c::'a::comm-ring-1) ∗ f) = fps-const c ∗ fps-deriv f

```
  by simp

lemma fps-deriv-0[simp]: fps-deriv 0 = 0
  by (simp add: fps-deriv-def fps-eq-iff)

lemma fps-deriv-1[simp]: fps-deriv 1 = 0
  by (simp add: fps-deriv-def fps-eq-iff )

lemma fps-deriv-mult-const-right[simp]:
  fps-deriv (f * fps-const (c::'a::comm-ring-1)) = fps-deriv f * fps-const c
  by simp

lemma fps-deriv-setsum:
  fps-deriv (setsum f S) = setsum (λi. fps-deriv (f i :: 'a::comm-ring-1 fps)) S
proof (cases finite S)
  case False
  then show ?thesis by simp
next
  case True
  show ?thesis by (induct rule: finite-induct [OF True]) simp-all
qed

lemma fps-deriv-eq-0-iff [simp]:
  fps-deriv f = 0 ⟷ f = fps-const (f$0 :: 'a::{idom,semiring-char-0})
  (is ?lhs ⟷ ?rhs)
proof
  show ?lhs if ?rhs
  proof -
    from that have fps-deriv f = fps-deriv (fps-const (f$0))
      by simp
    then show ?thesis
      by simp
  qed
  show ?rhs if ?lhs
  proof -
    from that have ∀ n. (fps-deriv f)$n = 0
      by simp
    then have ∀ n. f$(n+1) = 0
      by (simp del: of-nat-Suc of-nat-add One-nat-def)
    then show ?thesis
      apply (clarsimp simp add: fps-eq-iff fps-const-def)
      apply (erule-tac x=n − 1 in allE)
      apply simp
      done
  qed
qed

lemma fps-deriv-eq-iff:
  fixes f :: 'a::{idom,semiring-char-0} fps
```

shows *fps-deriv f* = *fps-deriv g* ⟷ (*f* = *fps-const*(*f*$*0* − *g*$*0*) + *g*)
proof −
have *fps-deriv f* = *fps-deriv g* ⟷ *fps-deriv* (*f* − *g*) = *0*
by *simp*
also have ... ⟷ *f* − *g* = *fps-const* ((*f* − *g*) $ *0*)
unfolding *fps-deriv-eq-0-iff* **..**
finally show *?thesis*
by (*simp add*: *field-simps*)
qed

lemma *fps-deriv-eq-iff-ex*:
(*fps-deriv f* = *fps-deriv g*) ⟷ (∃ *c*::′*a*::{*idom*,*semiring-char-0*}. *f* = *fps-const c* + *g*)
by (*auto simp*: *fps-deriv-eq-iff*)

fun *fps-nth-deriv* :: *nat* ⇒ ′*a*::*semiring-1 fps* ⇒ ′*a fps*
where
fps-nth-deriv 0 f = *f*
| *fps-nth-deriv* (*Suc n*) *f* = *fps-nth-deriv n* (*fps-deriv f*)

lemma *fps-nth-deriv-commute*: *fps-nth-deriv* (*Suc n*) *f* = *fps-deriv* (*fps-nth-deriv n f*)
by (*induct n arbitrary*: *f*) *auto*

lemma *fps-nth-deriv-linear*[*simp*]:
fps-nth-deriv n (*fps-const* (*a*::′*a*::*comm-semiring-1*) ∗ *f* + *fps-const b* ∗ *g*) =
fps-const a ∗ *fps-nth-deriv n f* + *fps-const b* ∗ *fps-nth-deriv n g*
by (*induct n arbitrary*: *f g*) (*auto simp add*: *fps-nth-deriv-commute*)

lemma *fps-nth-deriv-neg*[*simp*]:
fps-nth-deriv n (− (*f* :: ′*a*::*comm-ring-1 fps*)) = − (*fps-nth-deriv n f*)
by (*induct n arbitrary*: *f*) *simp-all*

lemma *fps-nth-deriv-add*[*simp*]:
fps-nth-deriv n ((*f* :: ′*a*::*comm-ring-1 fps*) + *g*) = *fps-nth-deriv n f* + *fps-nth-deriv n g*
using *fps-nth-deriv-linear*[*of n 1 f 1 g*] **by** *simp*

lemma *fps-nth-deriv-sub*[*simp*]:
fps-nth-deriv n ((*f* :: ′*a*::*comm-ring-1 fps*) − *g*) = *fps-nth-deriv n f* − *fps-nth-deriv n g*
using *fps-nth-deriv-add* [*of n f* − *g*] **by** *simp*

lemma *fps-nth-deriv-0*[*simp*]: *fps-nth-deriv n 0* = *0*
by (*induct n*) *simp-all*

lemma *fps-nth-deriv-1*[*simp*]: *fps-nth-deriv n 1* = (*if n* = *0 then 1 else 0*)
by (*induct n*) *simp-all*

lemma *fps-nth-deriv-const*[*simp*]:
fps-nth-deriv n (*fps-const c*) = (*if n* = *0 then fps-const c else 0*)
by (*cases n*) *simp-all*

lemma *fps-nth-deriv-mult-const-left*[*simp*]:
fps-nth-deriv n (*fps-const* (*c*::*'a*::*comm-ring-1*) ∗ *f*) = *fps-const c* ∗ *fps-nth-deriv n f*
using *fps-nth-deriv-linear*[*of n c f 0 0*] **by** *simp*

lemma *fps-nth-deriv-mult-const-right*[*simp*]:
fps-nth-deriv n (*f* ∗ *fps-const* (*c*::*'a*::*comm-ring-1*)) = *fps-nth-deriv n f* ∗ *fps-const c*
using *fps-nth-deriv-linear*[*of n c f 0 0*] **by** (*simp add*: *mult.commute*)

lemma *fps-nth-deriv-setsum*:
fps-nth-deriv n (*setsum f S*) = *setsum* (*λi. fps-nth-deriv n* (*f i* :: *'a*::*comm-ring-1 fps*)) *S*
proof (*cases finite S*)
case *True*
show *?thesis* **by** (*induct rule*: *finite-induct* [*OF True*]) *simp-all*
next
case *False*
then show *?thesis* **by** *simp*
qed

lemma *fps-deriv-maclauren-0*:
(*fps-nth-deriv k* (*f* :: *'a*::*comm-semiring-1 fps*)) $ *0* = *of-nat* (*fact k*) ∗ *f* $ *k*
by (*induct k arbitrary*: *f*) (*auto simp add*: *field-simps of-nat-mult*)

43.13 Powers

lemma *fps-power-zeroth-eq-one*: *a*$*0* =*1* ⟹ *a*ˆ*n* $ *0* = (*1*::*'a*::*semiring-1*)
by (*induct n*) (*auto simp add*: *expand-fps-eq fps-mult-nth*)

lemma *fps-power-first-eq*: (*a* :: *'a*::*comm-ring-1 fps*) $ *0* =*1* ⟹ *a*ˆ*n* $ *1* = *of-nat n* ∗ *a*$*1*
proof (*induct n*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
show *?case* **unfolding** *power-Suc fps-mult-nth*
using *Suc.hyps*[*OF* ‹*a*$*0* = *1*›] ‹*a*$*0* = *1*› *fps-power-zeroth-eq-one*[*OF* ‹*a*$*0*=*1*›]
by (*simp add*: *field-simps*)
qed

lemma *startsby-one-power*:*a* $ *0* = (*1*::*'a*::*comm-ring-1*) ⟹ *a*ˆ*n* $ *0* = *1*
by (*induct n*) (*auto simp add*: *fps-mult-nth*)

```
lemma startsby-zero-power:a $0 = (0::'a::comm-ring-1) ⟹ n > 0 ⟹ a^n $0
= 0
  by (induct n) (auto simp add: fps-mult-nth)

lemma startsby-power:a $0 = (v::'a::comm-ring-1) ⟹ a^n $0 = v^n
  by (induct n) (auto simp add: fps-mult-nth)

lemma startsby-zero-power-iff[simp]: a^n $0 = (0::'a::idom) ⟷ n ≠ 0 ∧ a$0
= 0
  apply (rule iffI)
  apply (induct n)
  apply (auto simp add: fps-mult-nth)
  apply (rule startsby-zero-power, simp-all)
  done

lemma startsby-zero-power-prefix:
  assumes a0: a $ 0 = (0::'a::idom)
  shows ∀ n < k. a ^ k $ n = 0
  using a0
proof (induct k rule: nat-less-induct)
  fix k
  assume H: ∀ m<k. a $0 =  0 ⟶ (∀ n<m. a ^ m $ n = 0) and a0: a $ 0 = 0
  show ∀ m<k. a ^ k $ m = 0
  proof (cases k)
    case 0
    then show ?thesis by simp
  next
    case (Suc l)
    have a^k $ m = 0 if mk: m < k for m
    proof (cases m = 0)
      case True
      then show ?thesis
        using startsby-zero-power[of a k] Suc a0 by simp
    next
      case False
      have a ^k $ m = (a^l * a) $m
        by (simp add: Suc mult.commute)
      also have ... = (∑ i = 0..m. a ^ l $ i * a $ (m - i))
        by (simp add: fps-mult-nth)
      also have ... = 0
        apply (rule setsum.neutral)
        apply auto
        apply (case-tac x = m)
        using a0 apply simp
        apply (rule H[rule-format])
        using a0 Suc mk apply auto
        done
      finally show ?thesis .
```

```
    qed
    then show ?thesis by blast
  qed
qed

lemma startsby-zero-setsum-depends:
  assumes a0: a $0 = (0::'a::idom)
    and kn: n ≥ k
  shows setsum (λi. (a ^ i)$k) {0 .. n} = setsum (λi. (a ^ i)$k) {0 .. k}
  apply (rule setsum.mono-neutral-right)
  using kn
  apply auto
  apply (rule startsby-zero-power-prefix[rule-format, OF a0])
  apply arith
  done

lemma startsby-zero-power-nth-same:
  assumes a0: a$0 = (0::'a::idom)
  shows a^n $ n = (a$1) ^ n
proof (induct n)
  case 0
  then show ?case by simp
next
  case (Suc n)
  have a ^ Suc n $ (Suc n) = (a^n * a)$(Suc n)
    by (simp add: field-simps)
  also have ... = setsum (λi. a^n$i * a $ (Suc n − i)) {0.. Suc n}
    by (simp add: fps-mult-nth)
  also have ... = setsum (λi. a^n$i * a $ (Suc n − i)) {n .. Suc n}
    apply (rule setsum.mono-neutral-right)
    apply simp
    apply clarsimp
    apply clarsimp
    apply (rule startsby-zero-power-prefix[rule-format, OF a0])
    apply arith
    done
  also have ... = a^n $ n * a$1
    using a0 by simp
  finally show ?case
    using Suc.hyps by simp
qed

lemma fps-inverse-power:
  fixes a :: 'a::field fps
  shows inverse (a^n) = inverse a ^ n
  by (induction n) (simp-all add: fps-inverse-mult)

lemma fps-deriv-power:
  fps-deriv (a ^ n) = fps-const (of-nat n :: 'a::comm-ring-1) * fps-deriv a * a ^ (n
```

− *1*)
apply (*induct n*)
apply (*auto simp add*: *field-simps fps-const-add*[*symmetric*] *simp del*: *fps-const-add*)
apply (*case-tac n*)
apply (*auto simp add*: *field-simps*)
done

lemma *fps-inverse-deriv*:
fixes *a* :: *'a::field fps*
assumes *a0*: *a$0* ≠ *0*
shows *fps-deriv* (*inverse a*) = − *fps-deriv a* ∗ (*inverse a*)^2
proof −
from *inverse-mult-eq-1*[*OF a0*]
have *fps-deriv* (*inverse a* ∗ *a*) = *0* **by** *simp*
then have *inverse a* ∗ *fps-deriv a* + *fps-deriv* (*inverse a*) ∗ *a* = *0*
by *simp*
then have *inverse a* ∗ (*inverse a* ∗ *fps-deriv a* + *fps-deriv* (*inverse a*) ∗ *a*) = *0*
by *simp*
with *inverse-mult-eq-1*[*OF a0*]
have (*inverse a*)^2 ∗ *fps-deriv a* + *fps-deriv* (*inverse a*) = *0*
unfolding *power2-eq-square*
apply (*simp add*: *field-simps*)
apply (*simp add*: *mult.assoc*[*symmetric*])
done
then have (*inverse a*)^2 ∗ *fps-deriv a* + *fps-deriv* (*inverse a*) − *fps-deriv a* ∗ (*inverse a*)^2 =
0 − *fps-deriv a* ∗ (*inverse a*)^2
by *simp*
then show *fps-deriv* (*inverse a*) = − *fps-deriv a* ∗ (*inverse a*)^2
by (*simp add*: *field-simps*)
qed

lemma *fps-inverse-deriv'*:
fixes *a* :: *'a::field fps*
assumes *a0*: *a* $ *0* ≠ *0*
shows *fps-deriv* (*inverse a*) = − *fps-deriv a* / *a*^{2}
using *fps-inverse-deriv*[*OF a0*] *a0*
by (*simp add*: *fps-divide-unit power2-eq-square fps-inverse-mult*)

lemma *inverse-mult-eq-1'*:
assumes *f0*: *f$0* ≠ (*0::'a::field*)
shows *f* ∗ *inverse f* = *1*
by (*metis mult.commute inverse-mult-eq-1 f0*)

lemma *fps-divide-deriv*:
assumes *b dvd* (*a* :: *'a* :: *field fps*)
shows *fps-deriv* (*a* / *b*) = (*fps-deriv a* ∗ *b* − *a* ∗ *fps-deriv b*) / *b*ˆ*2*
proof −

have *eq-divide-imp*: $c \neq 0 \Longrightarrow a * c = b \Longrightarrow a = b$ *div c* **for** *a b c* :: *'a* :: *field fps*
  by (*drule sym*) (*simp add*: *mult.assoc*)
from *assms* **have** $a = a / b * b$ **by** *simp*
also have *fps-deriv* $(a / b * b) =$ *fps-deriv* $(a / b) * b + a / b *$ *fps-deriv* b **by** *simp*
finally have *fps-deriv* $(a / b) * b\hat{}2 =$ *fps-deriv* $a * b - a *$ *fps-deriv* b **using** *assms*
  by (*simp add*: *power2-eq-square algebra-simps*)
thus *?thesis* **by** (*cases* $b = 0$) (*auto simp*: *eq-divide-imp*)
qed

lemma *fps-inverse-gp'*: *inverse* (*Abs-fps* (λ*n*. *1*::*'a*::*field*)) = *1* − *X*
  by (*simp add*: *fps-inverse-gp fps-eq-iff X-def*)

lemma *fps-nth-deriv-X*[*simp*]: *fps-nth-deriv n X* = (*if n* = *0 then X else if n*=*1 then 1 else 0*)
  by (*cases n*) *simp-all*

lemma *fps-inverse-X-plus1*: *inverse* (*1* + *X*) = *Abs-fps* (λ*n*. (− (*1*::*'a*::*field*)) ˆ *n*)
  (**is** - = *?r*)
proof −
  have *eq*: (*1* + *X*) ∗ *?r* = *1*
    unfolding *minus-one-power-iff*
    by (*auto simp add*: *field-simps fps-eq-iff*)
  show *?thesis*
    by (*auto simp add*: *eq intro*: *fps-inverse-unique*)
qed

43.14 Integration

definition *fps-integral* :: *'a*::*field-char-0 fps* ⇒ *'a* ⇒ *'a fps*
  where *fps-integral a a0* = *Abs-fps* (λ*n*. *if n* = *0 then a0 else* (*a*\$(*n* − *1*) / *of-nat n*))

lemma *fps-deriv-fps-integral*: *fps-deriv* (*fps-integral a a0*) = *a*
  unfolding *fps-integral-def fps-deriv-def*
  by (*simp add*: *fps-eq-iff del*: *of-nat-Suc*)

lemma *fps-integral-linear*:
  fps-integral (*fps-const a* ∗ *f* + *fps-const b* ∗ *g*) (*a*∗*a0* + *b*∗*b0*) =
    fps-const a ∗ *fps-integral f a0* + *fps-const b* ∗ *fps-integral g b0*
  (**is** *?l* = *?r*)
proof −
  have *fps-deriv ?l* = *fps-deriv ?r*
    by (*simp add*: *fps-deriv-fps-integral*)
  moreover have *?l*\$*0* = *?r*\$*0*
    by (*simp add*: *fps-integral-def*)

ultimately show *?thesis*
unfolding *fps-deriv-eq-iff* **by** *auto*
qed

43.15 Composition of FPSs

definition *fps-compose* :: *'a::semiring-1 fps* ⇒ *'a fps* ⇒ *'a fps* (**infixl** *oo 55*)
where $a\ oo\ b = Abs\text{-}fps\ (\lambda n.\ setsum\ (\lambda i.\ a\$i * (b\hat{\ }i\$n))\ \{0..n\})$

lemma *fps-compose-nth*: $(a\ oo\ b)\$n = setsum\ (\lambda i.\ a\$i * (b\hat{\ }i\$n))\ \{0..n\}$
by (*simp add*: *fps-compose-def*)

lemma *fps-compose-nth-0* [*simp*]: $(f\ oo\ g)\ \$\ 0 = f\ \$\ 0$
by (*simp add*: *fps-compose-nth*)

lemma *fps-compose-X*[*simp*]: *a oo X* = (*a* :: *'a::comm-ring-1 fps*)
by (*simp add*: *fps-ext fps-compose-def mult-delta-right setsum.delta'*)

lemma *fps-const-compose*[*simp*]: *fps-const* (*a::'a::comm-ring-1*) *oo b* = *fps-const a*
by (*simp add*: *fps-eq-iff fps-compose-nth mult-delta-left setsum.delta*)

lemma *numeral-compose*[*simp*]: (*numeral k* :: *'a::comm-ring-1 fps*) *oo b* = *numeral k*
unfolding *numeral-fps-const* **by** *simp*

lemma *neg-numeral-compose*[*simp*]: (− *numeral k* :: *'a::comm-ring-1 fps*) *oo b* = − *numeral k*
unfolding *neg-numeral-fps-const* **by** *simp*

lemma *X-fps-compose-startby0*[*simp*]: $a\$0 = 0 \Longrightarrow X\ oo\ a = (a :: 'a::comm\text{-}ring\text{-}1\ fps)$
by (*simp add*: *fps-eq-iff fps-compose-def mult-delta-left setsum.delta not-le*)

43.16 Rules from Herbert Wilf's Generatingfunctionology

43.16.1 Rule 1

lemma *fps-power-mult-eq-shift*:
$X\hat{\ }Suc\ k * Abs\text{-}fps\ (\lambda n.\ a\ (n + Suc\ k)) =$
$Abs\text{-}fps\ a - setsum\ (\lambda i.\ fps\text{-}const\ (a\ i :: 'a::comm\text{-}ring\text{-}1) * X\hat{\ }i)\ \{0\ ..\ k\}$
(**is** *?lhs* = *?rhs*)
proof −
have *?lhs* \$ *n* = *?rhs* \$ *n* **for** *n* :: *nat*
proof −
have *?lhs* \$ *n* = (*if n* < *Suc k then 0 else a n*)
unfolding *X-power-mult-nth* **by** *auto*
also have . . . = *?rhs* \$ *n*
proof (*induct k*)
case *0*

```
      then show ?case
        by (simp add: fps-setsum-nth)
    next
      case (Suc k)
      have (Abs-fps a − setsum (λi. fps-const (a i :: 'a) ∗ Xˆi) {0 .. Suc k})$n =
        (Abs-fps a − setsum (λi. fps-const (a i :: 'a) ∗ Xˆi) {0 .. k} −
          fps-const (a (Suc k)) ∗ Xˆ Suc k) $ n
        by (simp add: field-simps)
      also have ... = (if n < Suc k then 0 else a n) − (fps-const (a (Suc k)) ∗ Xˆ
Suc k)$n
        using Suc.hyps[symmetric] unfolding fps-sub-nth by simp
      also have ... = (if n < Suc (Suc k) then 0 else a n)
        unfolding X-power-mult-right-nth
        apply (auto simp add: not-less fps-const-def)
        apply (rule cong[of a a, OF refl])
        apply arith
        done
      finally show ?case
        by simp
    qed
    finally show ?thesis .
  qed
  then show ?thesis
    by (simp add: fps-eq-iff)
qed
```

43.16.2 Rule 2

```
definition XD = op ∗ X ∘ fps-deriv

lemma XD-add[simp]:XD (a + b) = XD a + XD (b :: 'a::comm-ring-1 fps)
  by (simp add: XD-def field-simps)

lemma XD-mult-const[simp]:XD (fps-const (c::'a::comm-ring-1) ∗ a) = fps-const
c ∗ XD a
  by (simp add: XD-def field-simps)

lemma XD-linear[simp]: XD (fps-const c ∗ a + fps-const d ∗ b) =
    fps-const c ∗ XD a + fps-const d ∗ XD (b :: 'a::comm-ring-1 fps)
  by simp

lemma XDN-linear:
  (XD ˆˆ n) (fps-const c ∗ a + fps-const d ∗ b) =
    fps-const c ∗ (XD ˆˆ n) a + fps-const d ∗ (XD ˆˆ n) (b :: 'a::comm-ring-1 fps)
  by (induct n) simp-all

lemma fps-mult-X-deriv-shift: X∗ fps-deriv a = Abs-fps (λn. of-nat n∗ a$n)
  by (simp add: fps-eq-iff)
```

lemma *fps-mult-XD-shift*:
  $(XD \ \hat{}\hat{}\ k)$ $(a :: 'a::comm\text{-}ring\text{-}1\ fps) = Abs\text{-}fps\ (\lambda n.\ (of\text{-}nat\ n\ \hat{}\ k) * a\$n)$
 by (*induct k arbitrary*: *a*) (*simp-all add*: *XD-def fps-eq-iff field-simps del*: *One-nat-def*)

43.16.3 Rule 3

Rule 3 is trivial and is given by *fps-times-def*.

43.16.4 Rule 5 — summation and "division" by (1 - X)

lemma *fps-divide-X-minus1-setsum-lemma*:
  $a = ((1::'a::comm\text{-}ring\text{-}1\ fps) - X) * Abs\text{-}fps\ (\lambda n.\ setsum\ (\lambda i.\ a\ \$\ i)\ \{0..n\})$
proof −
  let $?sa = Abs\text{-}fps\ (\lambda n.\ setsum\ (\lambda i.\ a\ \$\ i)\ \{0..n\})$
  have *th0*: $\bigwedge i.\ (1 - (X::'a\ fps))\ \$\ i = (if\ i = 0\ then\ 1\ else\ if\ i = 1\ then\ -\ 1\ else\ 0)$
    by *simp*
  have $a\$n = ((1 - X) * ?sa)\ \$\ n$ **for** *n*
  proof (*cases* $n = 0$)
    case *True*
    then show *?thesis*
      by (*simp add*: *fps-mult-nth*)
  next
    case *False*
    then have *u*: $\{0\} \cup (\{1\} \cup \{2..n\}) = \{0..n\}$ $\{1\} \cup \{2..n\} = \{1..n\}$
      $\{0..n - 1\} \cup \{n\} = \{0..n\}$
      by (*auto simp*: *set-eq-iff*)
    have *d*: $\{0\} \cap (\{1\} \cup \{2..n\}) = \{\}$ $\{1\} \cap \{2..n\} = \{\}$ $\{0..n - 1\} \cap \{n\} = \{\}$
      using *False* **by** *simp-all*
    have *f*: *finite* $\{0\}$ *finite* $\{1\}$ *finite* $\{2\ ..\ n\}$
      finite $\{0\ ..\ n - 1\}$ *finite* $\{n\}$ **by** *simp-all*
    have $((1 - X) * ?sa)\ \$\ n = setsum\ (\lambda i.\ (1 - X)\$\ i * ?sa\ \$\ (n - i))\ \{0\ ..\ n\}$
      by (*simp add*: *fps-mult-nth*)
    also have $\ldots = a\$n$
      unfolding *th0*
       unfolding *setsum.union-disjoint*[*OF f*(*1*) *finite-UnI*[*OF f*(*2*,*3*)] *d*(*1*), *unfolded u*(*1*)]
      unfolding *setsum.union-disjoint*[*OF f*(*2*) *f*(*3*) *d*(*2*)]
      apply (*simp*)
      unfolding *setsum.union-disjoint*[*OF f*(*4*,*5*) *d*(*3*), *unfolded u*(*3*)]
      apply *simp*
      done
    finally show *?thesis*
      by *simp*
  qed
  then show *?thesis*
    unfolding *fps-eq-iff* **by** *blast*
qed

lemma *fps-divide-X-minus1-setsum*:
  a /((*1*::'*a*::*field fps*) − *X*) = *Abs-fps* (λ*n*. *setsum* (λ*i*. *a* $ *i*) {*0*..*n*})
proof −
  let *?X* = *1* − (*X*::'*a fps*)
  have *th0*: *?X* $ *0* ≠ *0*
    by *simp*
  have *a* /*?X* = *?X* ∗ *Abs-fps* (λ*n*::*nat*. *setsum* (*op* $ *a*) {*0*..*n*}) ∗ *inverse ?X*
    using *fps-divide-X-minus1-setsum-lemma*[*of a*, *symmetric*] *th0*
    by (*simp add*: *fps-divide-def mult.assoc*)
  also have ... = (*inverse ?X* ∗ *?X*) ∗ *Abs-fps* (λ*n*::*nat*. *setsum* (*op* $ *a*) {*0*..*n*})

    by (*simp add*: *ac-simps*)
  finally show *?thesis*
    by (*simp add*: *inverse-mult-eq-1*[*OF th0*])
qed

43.16.5 Rule 4 in its more general form: generalizes Rule 3 for an arbitrary finite product of FPS, also the relvant instance of powers of a FPS

definition *natpermute n k* = {*l* :: *nat list*. *length l* = *k* ∧ *listsum l* = *n*}

lemma *natlist-trivial-1*: *natpermute n 1* = {[*n*]}
  apply (*auto simp add*: *natpermute-def*)
  apply (*case-tac x*)
  apply *auto*
  done

lemma *append-natpermute-less-eq*:
  assumes *xs* @ *ys* ∈ *natpermute n k*
  shows *listsum xs* ≤ *n*
    and *listsum ys* ≤ *n*
proof −
  from *assms* **have** *listsum* (*xs* @ *ys*) = *n*
    by (*simp add*: *natpermute-def*)
  then have *listsum xs* + *listsum ys* = *n*
    by *simp*
  then show *listsum xs* ≤ *n* **and** *listsum ys* ≤ *n*
    by *simp-all*
qed

lemma *natpermute-split*:
  assumes *h* ≤ *k*
  shows *natpermute n k* =
    (⋃*m* ∈{*0*..*n*}. {*l1* @ *l2* |*l1 l2*. *l1* ∈ *natpermute m h* ∧ *l2* ∈ *natpermute* (*n* − *m*) (*k* − *h*)})
  (**is** *?L* = *?R* **is** - = (⋃*m* ∈{*0*..*n*}. *?S m*))
proof

```
  show ?R ⊆ ?L
  proof
    fix l
    assume l: l ∈ ?R
    from l obtain m xs ys where h: m ∈ {0..n}
      and xs: xs ∈ natpermute m h
      and ys: ys ∈ natpermute (n − m) (k − h)
      and leq: l = xs@ys by blast
    from xs have xs': listsum xs = m
      by (simp add: natpermute-def)
    from ys have ys': listsum ys = n − m
      by (simp add: natpermute-def)
    show l ∈ ?L using leq xs ys h
      apply (clarsimp simp add: natpermute-def)
      unfolding xs' ys'
      using assms xs ys
      unfolding natpermute-def
      apply simp
      done
  qed
  show ?L ⊆ ?R
  proof
    fix l
    assume l: l ∈ natpermute n k
    let ?xs = take h l
    let ?ys = drop h l
    let ?m = listsum ?xs
    from l have ls: listsum (?xs @ ?ys) = n
      by (simp add: natpermute-def)
    have xs: ?xs ∈ natpermute ?m h using l assms
      by (simp add: natpermute-def)
    have l-take-drop: listsum l = listsum (take h l @ drop h l)
      by simp
    then have ys: ?ys ∈ natpermute (n − ?m) (k − h)
     using l assms ls by (auto simp add: natpermute-def simp del: append-take-drop-id)
    from ls have m: ?m ∈ {0..n}
      by (simp add: l-take-drop del: append-take-drop-id)
    from xs ys ls show l ∈ ?R
      apply auto
      apply (rule bexI [where x = ?m])
      apply (rule exI [where x = ?xs])
      apply (rule exI [where x = ?ys])
      using ls l
     apply (auto simp add: natpermute-def l-take-drop simp del: append-take-drop-id)
      apply simp
      done
  qed
qed
```

lemma *natpermute-0*: *natpermute n 0* = (*if n* = *0 then* {[]} *else* {})
  by (*auto simp add*: *natpermute-def*)

lemma *natpermute-0'*[*simp*]: *natpermute 0 k* = (*if k* = *0 then* {[]} *else* {*replicate k 0*})
  apply (*auto simp add*: *set-replicate-conv-if natpermute-def*)
  apply (*rule nth-equalityI*)
  apply *simp-all*
  done

lemma *natpermute-finite*: *finite* (*natpermute n k*)
proof (*induct k arbitrary*: *n*)
  case *0*
  then show *?case*
    apply (*subst natpermute-split*[*of 0 0*, *simplified*])
    apply (*simp add*: *natpermute-0*)
    done
next
  case (*Suc k*)
  then show *?case* **unfolding** *natpermute-split* [*of k Suc k*, *simplified*]
    apply −
    apply (*rule finite-UN-I*)
    apply *simp*
    unfolding *One-nat-def*[*symmetric*] *natlist-trivial-1*
    apply *simp*
    done
qed

lemma *natpermute-contain-maximal*:
  {*xs* ∈ *natpermute n* (*k* + *1*). *n* ∈ *set xs*} = (⋃ *i*∈{*0* .. *k*}. {(*replicate* (*k* + *1*) *0*) [*i*:=*n*]})
  (**is** *?A* = *?B*)
proof
  show *?A* ⊆ *?B*
  proof
    fix *xs*
    assume *xs* ∈ *?A*
    then have *H*: *xs* ∈ *natpermute n* (*k* + *1*) **and** *n*: *n* ∈ *set xs*
      by *blast*+
    then obtain *i* **where** *i*: *i* ∈ {*0*.. *k*} *xs*!*i* = *n*
      unfolding *in-set-conv-nth* **by** (*auto simp add*: *less-Suc-eq-le natpermute-def*)
    have *eqs*: ({*0*..*k*} − {*i*}) ∪ {*i*} = {*0*..*k*}
      using *i* **by** *auto*
    have *f*: *finite*({*0*..*k*} − {*i*}) *finite* {*i*}
      by *auto*
    have *d*: ({*0*..*k*} − {*i*}) ∩ {*i*} = {}
      using *i* **by** *auto*
    from *H* **have** *n* = *setsum* (*nth xs*) {*0*..*k*}
      apply (*simp add*: *natpermute-def*)

```
    apply (auto simp add: atLeastLessThanSuc-atLeastAtMost listsum-setsum-nth)
    done
   also have ... = n + setsum (nth xs) ({0..k} − {i})
    unfolding setsum.union-disjoint[OF f d, unfolded eqs] using i by simp
   finally have zxs: ∀ j∈ {0..k} − {i}. xs!j = 0
    by auto
   from H have xsl: length xs = k+1
    by (simp add: natpermute-def)
   from i have i': i < length (replicate (k+1) 0)   i < k+1
    unfolding length-replicate by presburger+
   have xs = replicate (k+1) 0 [i := n]
    apply (rule nth-equalityI)
    unfolding xsl length-list-update length-replicate
    apply simp
    apply clarify
    unfolding nth-list-update[OF i'(1)]
    using i zxs
    apply (case-tac ia = i)
    apply (auto simp del: replicate.simps)
    done
   then show xs ∈ ?B using i by blast
  qed
  show ?B ⊆ ?A
  proof
   fix xs
   assume xs ∈ ?B
   then obtain i where i: i ∈ {0..k} and xs: xs = replicate (k + 1) 0 [i:=n]
    by auto
   have nxs: n ∈ set xs
    unfolding xs
    apply (rule set-update-memI)
    using i apply simp
    done
   have xsl: length xs = k + 1
    by (simp only: xs length-replicate length-list-update)
   have listsum xs = setsum (nth xs) {0..<k+1}
    unfolding listsum-setsum-nth xsl ..
   also have ... = setsum (λj. if j = i then n else 0) {0..< k+1}
    by (rule setsum.cong) (simp-all add: xs del: replicate.simps)
   also have ... = n using i by (simp add: setsum.delta)
   finally have xs ∈ natpermute n (k + 1)
    using xsl unfolding natpermute-def mem-Collect-eq by blast
   then show xs ∈ ?A
    using nxs by blast
  qed
qed
```

The general form.

```
lemma fps-setprod-nth:
```

```
  fixes m :: nat
    and a :: nat ⇒ 'a::comm-ring-1 fps
  shows (setprod a {0 .. m}) $ n =
    setsum (λv. setprod (λj. (a j) $ (v!j)) {0..m}) (natpermute n (m+1))
  (is ?P m n)
proof (induct m arbitrary: n rule: nat-less-induct)
  fix m n assume H: ∀ m' < m. ∀ n. ?P m' n
  show ?P m n
  proof (cases m)
    case 0
    then show ?thesis
      apply simp
      unfolding natlist-trivial-1[where n = n, unfolded One-nat-def]
      apply simp
      done
  next
    case (Suc k)
    then have km: k < m by arith
    have u0: {0 .. k} ∪ {m} = {0..m}
      using Suc by (simp add: set-eq-iff) presburger
    have f0: finite {0 .. k} finite {m} by auto
    have d0: {0 .. k} ∩ {m} = {} using Suc by auto
    have (setprod a {0 .. m}) $ n = (setprod a {0 .. k} * a m) $ n
      unfolding setprod.union-disjoint[OF f0 d0, unfolded u0] by simp
    also have ... = (∑ i = 0..n. (∑ v∈natpermute i (k + 1). ∏ j∈{0..k}. a j $
v ! j) * a m $ (n − i))
      unfolding fps-mult-nth H[rule-format, OF km] ..
    also have ... = (∑ v∈natpermute n (m + 1). ∏ j∈{0..m}. a j $ v ! j)
      apply (simp add: Suc)
      unfolding natpermute-split[of m m + 1, simplified, of n,
        unfolded natlist-trivial-1[unfolded One-nat-def] Suc]
      apply (subst setsum.UNION-disjoint)
      apply simp
      apply simp
      unfolding image-Collect[symmetric]
      apply clarsimp
      apply (rule finite-imageI)
      apply (rule natpermute-finite)
      apply (clarsimp simp add: set-eq-iff)
      apply auto
      apply (rule setsum.cong)
      apply (rule refl)
      unfolding setsum-left-distrib
      apply (rule sym)
      apply (rule-tac l = λxs. xs @ [n − x] in setsum.reindex-cong)
      apply (simp add: inj-on-def)
      apply auto
      unfolding setprod.union-disjoint[OF f0 d0, unfolded u0, unfolded Suc]
      apply (clarsimp simp add: natpermute-def nth-append)
```

done
finally show *?thesis* **.**
qed
qed

The special form for powers.

lemma *fps-power-nth-Suc*:
fixes *m* :: *nat*
and *a* :: *'a::comm-ring-1 fps*
shows $(a\ \hat{}\ Suc\ m)\$n = setsum\ (\lambda v.\ setprod\ (\lambda j.\ a\ \$\ (v!j))\ \{0..m\})\ (natpermute\ n\ (m+1))$
proof −
have *th0*: $a\hat{}Suc\ m = setprod\ (\lambda i.\ a)\ \{0..m\}$
by (*simp add*: *setprod-constant*)
show *?thesis* **unfolding** *th0 fps-setprod-nth* **..**
qed

lemma *fps-power-nth*:
fixes *m* :: *nat*
and *a* :: *'a::comm-ring-1 fps*
shows $(a\ \hat{}m)\$n =$
$(if\ m=0\ then\ 1\$n\ else\ setsum\ (\lambda v.\ setprod\ (\lambda j.\ a\ \$\ (v!j))\ \{0..m - 1\})\ (natpermute\ n\ m))$
by (*cases m*) (*simp-all add*: *fps-power-nth-Suc del*: *power-Suc*)

lemma *fps-nth-power-0*:
fixes *m* :: *nat*
and *a* :: *'a::comm-ring-1 fps*
shows $(a\ \hat{}m)\$0 = (a\$0)\ \hat{}\ m$
proof (*cases m*)
case *0*
then show *?thesis* **by** *simp*
next
case (*Suc n*)
then have *c*: $m = card\ \{0..n\}$ **by** *simp*
have $(a\ \hat{}m)\$0 = setprod\ (\lambda i.\ a\$0)\ \{0..n\}$
by (*simp add*: *Suc fps-power-nth del*: *replicate.simps power-Suc*)
also have ... $= (a\$0)\ \hat{}\ m$
unfolding *c* **by** (*rule setprod-constant*)
finally show *?thesis* **.**
qed

lemma *fps-compose-inj-right*:
assumes *a0*: $a\$0 = (0::'a::idom)$
and *a1*: $a\$1 \neq 0$
shows $(b\ oo\ a = c\ oo\ a) \longleftrightarrow b = c$
(**is** $?lhs \longleftrightarrow ?rhs$)
proof
show *?lhs* **if** *?rhs* **using** *that* **by** *simp*

show *?rhs* **if** *?lhs*
proof −
have $b\$n = c\$n$ **for** n
proof (*induct n rule*: *nat-less-induct*)
fix n
assume H: $\forall\, m<n.\ b\$m = c\$m$
show $b\$n = c\$n$
proof (*cases n*)
case *0*
from ‹*?lhs*› **have** $(b\ oo\ a)\$n = (c\ oo\ a)\$n$
by *simp*
then show *?thesis*
using *0* **by** (*simp add*: *fps-compose-nth*)
next
case (*Suc n1*)
have *f*: *finite* $\{0\ ..\ n1\}$ *finite* $\{n\}$ **by** *simp-all*
have *eq*: $\{0\ ..\ n1\} \cup \{n\} = \{0\ ..\ n\}$ **using** *Suc* **by** *auto*
have *d*: $\{0\ ..\ n1\} \cap \{n\} = \{\}$ **using** *Suc* **by** *auto*
have *seq*: $(\sum i = 0..n1.\ b\ \$\ i * a\ \hat{}\ i\ \$\ n) = (\sum i = 0..n1.\ c\ \$\ i * a\ \hat{}\ i\ \$$
$n)$
apply (*rule setsum.cong*)
using *H Suc*
apply *auto*
done
have *th0*: $(b\ oo\ a)\ \$n = (\sum i = 0..n1.\ c\ \$\ i * a\ \hat{}\ i\ \$\ n) + b\$n * (a\$1)\hat{}n$
unfolding *fps-compose-nth setsum.union-disjoint*[*OF f d*, *unfolded eq*] *seq*
using *startsby-zero-power-nth-same*[*OF a0*]
by *simp*
have *th1*: $(c\ oo\ a)\ \$n = (\sum i = 0..n1.\ c\ \$\ i * a\ \hat{}\ i\ \$\ n) + c\$n * (a\$1)\hat{}n$
unfolding *fps-compose-nth setsum.union-disjoint*[*OF f d*, *unfolded eq*]
using *startsby-zero-power-nth-same*[*OF a0*]
by *simp*
from ‹*?lhs*›[*unfolded fps-eq-iff*, *rule-format*, *of n*] *th0 th1 a1*
show *?thesis* **by** *auto*
qed
qed
then show *?rhs* **by** (*simp add*: *fps-eq-iff*)
qed
qed

43.17 Radicals

declare *setprod.cong* [*fundef-cong*]

function *radical* :: (*nat* ⇒ *'a* ⇒ *'a*) ⇒ *nat* ⇒ *'a*::*field fps* ⇒ *nat* ⇒ *'a*
where
radical r 0 a 0 = *1*
| *radical r 0 a* (*Suc n*) = *0*
| *radical r* (*Suc k*) *a 0* = *r* (*Suc k*) (*a$0*)

| *radical r* (*Suc k*) *a* (*Suc n*) =
  (*a$ Suc n* − *setsum* (λ*xs. setprod* (λ*j. radical r* (*Suc k*) *a* (*xs* ! *j*)) {*0..k*})
   {*xs. xs* ∈ *natpermute* (*Suc n*) (*Suc k*) ∧ *Suc n* ∉ *set xs*}) /
  (*of-nat* (*Suc k*) ∗ (*radical r* (*Suc k*) *a 0*)ˆ*k*)
 by *pat-completeness auto*

termination *radical*
proof
 let *?R* = *measure* (λ(*r*, *k*, *a*, *n*). *n*)
 {
  show *wf ?R* **by** *auto*
 next
  fix *r k a n xs i*
  assume *xs*: *xs* ∈ {*xs* ∈ *natpermute* (*Suc n*) (*Suc k*). *Suc n* ∉ *set xs*} **and** *i*: *i* ∈ {*0..k*}
  have *False* **if** *c*: *Suc n* ≤ *xs* ! *i*
  proof −
   from *xs i* **have** *xs* !*i* ≠ *Suc n*
    by (*auto simp add*: *in-set-conv-nth natpermute-def*)
   with *c* **have** *c′*: *Suc n* < *xs*!*i* **by** *arith*
   have *fths*: *finite* {*0* ..< *i*} *finite* {*i*} *finite* {*i+1..<Suc k*}
    by *simp-all*
   have *d*: {*0* ..< *i*} ∩ ({*i*} ∪ {*i+1* ..< *Suc k*}) = {} {*i*} ∩ {*i+1..< Suc k*} = {}
    by *auto*
   have *eqs*: {*0..<Suc k*} = {*0* ..< *i*} ∪ ({*i*} ∪ {*i+1* ..< *Suc k*})
    using *i* **by** *auto*
   from *xs* **have** *Suc n* = *listsum xs*
    by (*simp add*: *natpermute-def*)
   also have . . . = *setsum* (*nth xs*) {*0..<Suc k*} **using** *xs*
    by (*simp add*: *natpermute-def listsum-setsum-nth*)
   also have . . . = *xs*!*i* + *setsum* (*nth xs*) {*0..<i*} + *setsum* (*nth xs*) {*i+1..<Suc k*}
    unfolding *eqs setsum.union-disjoint*[*OF fths*(*1*) *finite-UnI*[*OF fths*(*2*,*3*)] *d*(*1*)]
    unfolding *setsum.union-disjoint*[*OF fths*(*2*) *fths*(*3*) *d*(*2*)]
    by *simp*
   finally show *?thesis* **using** *c′* **by** *simp*
  qed
  then show ((*r*, *Suc k*, *a*, *xs*!*i*), *r*, *Suc k*, *a*, *Suc n*) ∈ *?R*
   apply *auto*
   apply (*metis not-less*)
   done
 next
  fix *r k a n*
  show ((*r*, *Suc k*, *a*, *0*), *r*, *Suc k*, *a*, *Suc n*) ∈ *?R* **by** *simp*
 }
qed

```
definition fps-radical r n a = Abs-fps (radical r n a)

lemma fps-radical0[simp]: fps-radical r 0 a = 1
  apply (auto simp add: fps-eq-iff fps-radical-def)
  apply (case-tac n)
  apply auto
  done

lemma fps-radical-nth-0[simp]: fps-radical r n a $ 0 = (if n = 0 then 1 else r n
(a$0))
  by (cases n) (simp-all add: fps-radical-def)

lemma fps-radical-power-nth[simp]:
  assumes r: (r k (a$0)) ^ k = a$0
  shows fps-radical r k a ^ k $ 0 = (if k = 0 then 1 else a$0)
proof (cases k)
  case 0
  then show ?thesis by simp
next
  case (Suc h)
  have eq1: fps-radical r k a ^ k $ 0 = (∏ j∈{0..h}. fps-radical r k a $ (replicate
k 0) ! j)
    unfolding fps-power-nth Suc by simp
  also have ... = (∏ j∈{0..h}. r k (a$0))
    apply (rule setprod.cong)
    apply simp
    using Suc
    apply (subgoal-tac replicate k 0 ! x = 0)
    apply (auto intro: nth-replicate simp del: replicate.simps)
    done
  also have ... = a$0
    using r Suc by (simp add: setprod-constant)
  finally show ?thesis
    using Suc by simp
qed

lemma natpermute-max-card:
  assumes n0: n ≠ 0
  shows card {xs ∈ natpermute n (k + 1). n ∈ set xs} = k + 1
  unfolding natpermute-contain-maximal
proof −
  let ?A = λi. {replicate (k + 1) 0[i := n]}
  let ?K = {0 ..k}
  have fK: finite ?K
    by simp
  have fAK: ∀ i∈?K. finite (?A i)
    by auto
  have d: ∀ i∈ ?K. ∀ j∈ ?K. i ≠ j ⟶
    {replicate (k + 1) 0[i := n]} ∩ {replicate (k + 1) 0[j := n]} = {}
```

```
  proof clarify
    fix i j
    assume i: i ∈ ?K and j: j ∈ ?K and ij: i ≠ j
    have False if eq: replicate (k+1) 0 [i:=n] = replicate (k+1) 0 [j:= n]
    proof -
      have (replicate (k+1) 0 [i:=n] ! i) = n
        using i by (simp del: replicate.simps)
      moreover
      have (replicate (k+1) 0 [j:=n] ! i) = 0
        using i ij by (simp del: replicate.simps)
      ultimately show ?thesis
        using eq n0 by (simp del: replicate.simps)
    qed
    then show {replicate (k + 1) 0[i := n]} ∩ {replicate (k + 1) 0[j := n]} = {}
      by auto
  qed
  from card-UN-disjoint[OF fK fAK d]
  show card (⋃i∈{0..k}. {replicate (k + 1) 0[i := n]}) = k + 1
    by simp
qed

lemma power-radical:
  fixes a:: 'a::field-char-0 fps
  assumes a0: a$0 ≠ 0
  shows (r (Suc k) (a$0)) ^ Suc k = a$0 ⟷ (fps-radical r (Suc k) a) ^ (Suc k)
= a
    (is ?lhs ⟷ ?rhs)
proof
  let ?r = fps-radical r (Suc k) a
  show ?rhs if r0: ?lhs
  proof -
    from a0 r0 have r00: r (Suc k) (a$0) ≠ 0 by auto
    have ?r ^ Suc k $ z = a$z for z
    proof (induct z rule: nat-less-induct)
      fix n
      assume H: ∀m<n. ?r ^ Suc k $ m = a$m
      show ?r ^ Suc k $ n = a $n
      proof (cases n)
        case 0
        then show ?thesis
          using fps-radical-power-nth[of r Suc k a, OF r0] by simp
      next
        case (Suc n1)
        then have n ≠ 0 by simp
        let ?Pnk = natpermute n (k + 1)
        let ?Pnkn = {xs ∈ ?Pnk. n ∈ set xs}
        let ?Pnknn = {xs ∈ ?Pnk. n ∉ set xs}
        have eq: ?Pnkn ∪ ?Pnknn = ?Pnk by blast
        have d: ?Pnkn ∩ ?Pnknn = {} by blast
```

have *f*: *finite ?Pnkn finite ?Pnknn*
using *finite-Un*[*of ?Pnkn ?Pnknn*, *unfolded eq*]
by (*metis natpermute-finite*)+
let *?f* = $\lambda v. \prod j\in\{0..k\}.$ *?r* \$ *v* ! *j*
have *setsum ?f ?Pnkn* = *setsum* ($\lambda v.$ *?r* \$ *n* ∗ *r* (*Suc k*) (*a* \$ *0*) ˆ *k*) *?Pnkn*
proof (*rule setsum.cong*)
fix *v* **assume** *v*: $v \in \{xs \in$ *natpermute n* (*k* + *1*). $n \in$ *set xs*}
let *?ths* = ($\prod j\in\{0..k\}.$ *fps-radical r* (*Suc k*) *a* \$ *v* ! *j*) =
fps-radical r (*Suc k*) *a* \$ *n* ∗ *r* (*Suc k*) (*a* \$ *0*) ˆ *k*
from *v* **obtain** *i* **where** *i*: $i \in \{0..k\}$ *v* = *replicate* (*k*+*1*) *0* [*i*:= *n*]
unfolding *natpermute-contain-maximal* **by** *auto*
have ($\prod j\in\{0..k\}.$ *fps-radical r* (*Suc k*) *a* \$ *v* ! *j*) =
($\prod j\in\{0..k\}.$ *if j* = *i then fps-radical r* (*Suc k*) *a* \$ *n else r* (*Suc k*)
(*a*\$*0*))
apply (*rule setprod.cong*, *simp*)
using *i r0*
apply (*simp del*: *replicate.simps*)
done
also have . . . = (*fps-radical r* (*Suc k*) *a* \$ *n*) ∗ *r* (*Suc k*) (*a*\$*0*) ˆ *k*
using *i r0* **by** (*simp add*: *setprod-gen-delta*)
finally show *?ths* **.**
qed *rule*
then have *setsum ?f ?Pnkn* = *of-nat* (*k*+*1*) ∗ *?r* \$ *n* ∗ *r* (*Suc k*) (*a* \$ *0*)
ˆ *k*
by (*simp add*: *natpermute-max-card*[*OF* ‹*n* ≠ *0*›, *simplified*])
also have . . . = *a*\$*n* − *setsum ?f ?Pnknn*
unfolding *Suc* **using** *r00 a0* **by** (*simp add*: *field-simps fps-radical-def del*:
of-nat-Suc)
finally have *fn*: *setsum ?f ?Pnkn* = *a*\$*n* − *setsum ?f ?Pnknn* **.**
have (*?r* ˆ *Suc k*)\$*n* = *setsum ?f ?Pnkn* + *setsum ?f ?Pnknn*
unfolding *fps-power-nth-Suc setsum.union-disjoint*[*OF f d*, *unfolded eq*] **..**
also have . . . = *a*\$*n* **unfolding** *fn* **by** *simp*
finally show *?thesis* **.**
qed
qed
then show *?thesis* **using** *r0* **by** (*simp add*: *fps-eq-iff*)
qed
show *?lhs* **if** *?rhs*
proof −
from *that* **have** ((*fps-radical r* (*Suc k*) *a*) ˆ (*Suc k*))\$*0* = *a*\$*0*
by *simp*
then show *?thesis*
unfolding *fps-power-nth-Suc*
by (*simp add*: *setprod-constant del*: *replicate.simps*)
qed
qed

lemma *eq-divide-imp′*:

fixes *c* :: *'a::field*
shows $c \neq 0 \Longrightarrow a * c = b \Longrightarrow a = b\ /\ c$
by (*simp add*: *field-simps*)

lemma *radical-unique*:
assumes *r0*: (*r* (*Suc k*) (*b\$0*)) ^ *Suc k* = *b\$0*
and *a0*: *r* (*Suc k*) (*b\$0* ::*'a::field-char-0*) = *a\$0*
and *b0*: *b\$0* ≠ *0*
shows *a*^(*Suc k*) = *b* ⟷ *a* = *fps-radical r* (*Suc k*) *b*
(**is** *?lhs* ⟷ *?rhs* **is** - ⟷ *a* = *?r*)
proof
show *?lhs* **if** *?rhs*
using *that* **using** *power-radical*[*OF b0*, *of r k*, *unfolded r0*] **by** *simp*
show *?rhs* **if** *?lhs*
proof −
have *r00*: *r* (*Suc k*) (*b\$0*) ≠ *0* **using** *b0 r0* **by** *auto*
have *ceq*: *card* {*0..k*} = *Suc k* **by** *simp*
from *a0* **have** *a0r0*: *a\$0* = *?r\$0* **by** *simp*
have *a* \$ *n* = *?r* \$ *n* **for** *n*
proof (*induct n rule*: *nat-less-induct*)
fix *n*
assume *h*: ∀ *m*<*n*. *a\$m* = *?r* \$*m*
show *a\$n* = *?r* \$ *n*
proof (*cases n*)
case *0*
then show *?thesis* **using** *a0* **by** *simp*
next
case (*Suc n1*)
have *fK*: *finite* {*0..k*} **by** *simp*
have *nz*: *n* ≠ *0* **using** *Suc* **by** *simp*
let *?Pnk* = *natpermute n* (*Suc k*)
let *?Pnkn* = {*xs* ∈ *?Pnk*. *n* ∈ *set xs*}
let *?Pnknn* = {*xs* ∈ *?Pnk*. *n* ∉ *set xs*}
have *eq*: *?Pnkn* ∪ *?Pnknn* = *?Pnk* **by** *blast*
have *d*: *?Pnkn* ∩ *?Pnknn* = {} **by** *blast*
have *f*: *finite ?Pnkn finite ?Pnknn*
using *finite-Un*[*of ?Pnkn ?Pnknn*, *unfolded eq*]
by (*metis natpermute-finite*)+
let *?f* = λ*v*. ∏*j*∈{*0..k*}. *?r* \$ *v* ! *j*
let *?g* = λ*v*. ∏*j*∈{*0..k*}. *a* \$ *v* ! *j*
have *setsum ?g ?Pnkn* = *setsum* (λ*v*. *a* \$ *n* ∗ (*?r\$0*)^*k*) *?Pnkn*
proof (*rule setsum.cong*)
fix *v*
assume *v*: *v* ∈ {*xs* ∈ *natpermute n* (*Suc k*). *n* ∈ *set xs*}
let *?ths* = (∏*j*∈{*0..k*}. *a* \$ *v* ! *j*) = *a* \$ *n* ∗ (*?r\$0*)^*k*
from *v* **obtain** *i* **where** *i*: *i* ∈ {*0..k*} *v* = *replicate* (*k+1*) *0* [*i*:= *n*]
unfolding *Suc-eq-plus1 natpermute-contain-maximal*
by (*auto simp del*: *replicate.simps*)
have (∏*j*∈{*0..k*}. *a* \$ *v* ! *j*) = (∏*j*∈{*0..k*}. *if j* = *i then a* \$ *n else r*

(*Suc k*) (*b*$*0*))
apply (*rule setprod.cong*, *simp*)
using *i a0*
apply (*simp del*: *replicate.simps*)
done
also have ... = *a* $ *n* ∗ (*?r* $ *0*) ^*k*
using *i* **by** (*simp add*: *setprod-gen-delta*)
finally show *?ths* **.**
qed *rule*
then have *th0*: *setsum ?g ?Pnkn* = *of-nat* (*k*+*1*) ∗ *a* $ *n* ∗ (*?r* $ *0*) ^*k*
by (*simp add*: *natpermute-max-card*[*OF nz*, *simplified*])
have *th1*: *setsum ?g ?Pnknn* = *setsum ?f ?Pnknn*
proof (*rule setsum.cong*, *rule refl*, *rule setprod.cong*, *simp*)
fix *xs i*
assume *xs*: *xs* ∈ *?Pnknn* **and** *i*: *i* ∈ {*0..k*}
have *False* **if** *c*: *n* ≤ *xs* ! *i*
proof −
from *xs i* **have** *xs* ! *i* ≠ *n*
by (*auto simp add*: *in-set-conv-nth natpermute-def*)
with *c* **have** *c'*: *n* < *xs*!*i* **by** *arith*
have *fths*: *finite* {*0* ..< *i*} *finite* {*i*} *finite* {*i*+*1*..<*Suc k*}
by *simp-all*
have *d*: {*0* ..< *i*} ∩ ({*i*} ∪ {*i*+*1* ..< *Suc k*}) = {} {*i*} ∩ {*i*+*1*..< *Suc k*} = {}
by *auto*
have *eqs*: {*0*..<*Suc k*} = {*0* ..< *i*} ∪ ({*i*} ∪ {*i*+*1* ..< *Suc k*})
using *i* **by** *auto*
from *xs* **have** *n* = *listsum xs*
by (*simp add*: *natpermute-def*)
also have ... = *setsum* (*nth xs*) {*0*..<*Suc k*}
using *xs* **by** (*simp add*: *natpermute-def listsum-setsum-nth*)
also have ... = *xs*!*i* + *setsum* (*nth xs*) {*0*..<*i*} + *setsum* (*nth xs*) {*i*+*1*..<*Suc k*}
unfolding *eqs setsum.union-disjoint*[*OF fths*(*1*) *finite-UnI*[*OF fths*(*2*,*3*)] *d*(*1*)]
unfolding *setsum.union-disjoint*[*OF fths*(*2*) *fths*(*3*) *d*(*2*)]
by *simp*
finally show *?thesis* **using** *c'* **by** *simp*
qed
then have *thn*: *xs*!*i* < *n* **by** *presburger*
from *h*[*rule-format*, *OF thn*] **show** *a*$(*xs* !*i*) = *?r*$(*xs*!*i*) **.**
qed
have *th00*: ⋀*x*::'*a*. *of-nat* (*Suc k*) ∗ (*x* ∗ *inverse* (*of-nat* (*Suc k*))) = *x*
by (*simp add*: *field-simps del*: *of-nat-Suc*)
from ‹*?lhs*› **have** *b*$*n* = *a*^*Suc k* $ *n*
by (*simp add*: *fps-eq-iff*)
also have *a* ^ *Suc k*$*n* = *setsum ?g ?Pnkn* + *setsum ?g ?Pnknn*
unfolding *fps-power-nth-Suc*
using *setsum.union-disjoint*[*OF f d*, *unfolded Suc-eq-plus1*[*symmetric*],

```
          unfolded eq, of ?g] by simp
      also have ... = of-nat (k+1) * a $ n * (?r $ 0)^k + setsum ?f ?Pnknn
        unfolding th0 th1 ..
      finally have of-nat (k+1) * a $ n * (?r $ 0)^k = b$n - setsum ?f ?Pnknn
        by simp
      then have a$n = (b$n - setsum ?f ?Pnknn) / (of-nat (k+1) * (?r $ 0)^k)
        apply -
        apply (rule eq-divide-imp')
        using r00
        apply (simp del: of-nat-Suc)
        apply (simp add: ac-simps)
        done
      then show ?thesis
        apply (simp del: of-nat-Suc)
        unfolding fps-radical-def Suc
        apply (simp add: field-simps Suc th00 del: of-nat-Suc)
        done
    qed
  qed
  then show ?rhs by (simp add: fps-eq-iff)
 qed
qed


lemma radical-power:
  assumes r0: r (Suc k) ((a$0) ^ Suc k) = a$0
    and a0: (a$0 :: 'a::field-char-0) ≠ 0
  shows (fps-radical r (Suc k) (a ^ Suc k)) = a
proof -
  let ?ak = a^ Suc k
  have ak0: ?ak $ 0 = (a$0) ^ Suc k
    by (simp add: fps-nth-power-0 del: power-Suc)
  from r0 have th0: r (Suc k) (a ^ Suc k $ 0) ^ Suc k = a ^ Suc k $ 0
    using ak0 by auto
  from r0 ak0 have th1: r (Suc k) (a ^ Suc k $ 0) = a $ 0
    by auto
  from ak0 a0 have ak00: ?ak $ 0 ≠0
    by auto
  from radical-unique[of r k ?ak a, OF th0 th1 ak00] show ?thesis
    by metis
qed

lemma fps-deriv-radical:
  fixes a :: 'a::field-char-0 fps
  assumes r0: (r (Suc k) (a$0)) ^ Suc k = a$0
    and a0: a$0 ≠ 0
  shows fps-deriv (fps-radical r (Suc k) a) =
    fps-deriv a / (fps-const (of-nat (Suc k)) * (fps-radical r (Suc k) a) ^ k)
proof -
```

let *?r = fps-radical r (Suc k) a*
let *?w = (fps-const (of-nat (Suc k)) * ?r ˆ k)*
from *a0 r0* **have** *r0′: r (Suc k) (a$0) ≠ 0*
by *auto*
from *r0′* **have** *w0: ?w $ 0 ≠ 0*
by *(simp del: of-nat-Suc)*
note *th0 = inverse-mult-eq-1[OF w0]*
let *?iw = inverse ?w*
from *iffD1[OF power-radical[of a r], OF a0 r0]*
have *fps-deriv (?r ˆ Suc k) = fps-deriv a*
by *simp*
then have *fps-deriv ?r * ?w = fps-deriv a*
by *(simp add: fps-deriv-power ac-simps del: power-Suc)*
then have *?iw * fps-deriv ?r * ?w = ?iw * fps-deriv a*
by *simp*
with *a0 r0* **have** *fps-deriv ?r * (?iw * ?w) = fps-deriv a / ?w*
by *(subst fps-divide-unit) (auto simp del: of-nat-Suc)*
then show *?thesis* **unfolding** *th0* **by** *simp*
qed

lemma *radical-mult-distrib*:
fixes *a :: ′a::field-char-0 fps*
assumes *k: k > 0*
and *ra0: r k (a $ 0) ˆ k = a $ 0*
and *rb0: r k (b $ 0) ˆ k = b $ 0*
and *a0: a $ 0 ≠ 0*
and *b0: b $ 0 ≠ 0*
shows *r k ((a * b) $ 0) = r k (a $ 0) * r k (b $ 0) ⟷*
*fps-radical r k (a * b) = fps-radical r k a * fps-radical r k b*
(**is** *?lhs ⟷ ?rhs*)
proof
show *?rhs* **if** *r0′: ?lhs*
proof −
from *r0′* **have** *r0: (r k ((a * b) $ 0)) ˆ k = (a * b) $ 0*
by *(simp add: fps-mult-nth ra0 rb0 power-mult-distrib)*
show *?thesis*
proof *(cases k)*
case *0*
then show *?thesis* **using** *r0′* **by** *simp*
next
case *(Suc h)*
let *?ra = fps-radical r (Suc h) a*
let *?rb = fps-radical r (Suc h) b*
have *th0: r (Suc h) ((a * b) $ 0) = (fps-radical r (Suc h) a * fps-radical r (Suc h) b) $ 0*
using *r0′ Suc* **by** *(simp add: fps-mult-nth)*
have *ab0: (a*b) $ 0 ≠ 0*
using *a0 b0* **by** *(simp add: fps-mult-nth)*
from *radical-unique[of r h a*b fps-radical r (Suc h) a * fps-radical r (Suc h)*

b, *OF r0*[*unfolded Suc*] *th0 ab0*, *symmetric*]
      iffD1[*OF power-radical*[*of - r*], *OF a0 ra0*[*unfolded Suc*]] *iffD1*[*OF power-radical*[*of - r*], *OF b0 rb0*[*unfolded Suc*]] *Suc r0'*
      show *?thesis*
        by (*auto simp add*: *power-mult-distrib simp del*: *power-Suc*)
    qed
  qed
  show *?lhs* **if** *?rhs*
  proof −
    from *that* **have** (*fps-radical r k* (*a* ∗ *b*)) $ *0* = (*fps-radical r k a* ∗ *fps-radical r k b*) $ *0*
      by *simp*
    then show *?thesis*
      using *k* **by** (*simp add*: *fps-mult-nth*)
  qed
qed

lemma *fps-divide-1* [*simp*]: (*a* :: *'a*::*field fps*) / *1* = *a*
  by (*fact divide-1*)

lemma *radical-divide*:
  fixes *a* :: *'a*::*field-char-0 fps*
  assumes *kp*: *k* > *0*
    and *ra0*: (*r k* (*a* $ *0*)) ˆ *k* = *a* $ *0*
    and *rb0*: (*r k* (*b* $ *0*)) ˆ *k* = *b* $ *0*
    and *a0*: *a*$*0* ≠ *0*
    and *b0*: *b*$*0* ≠ *0*
  shows *r k* ((*a* $ *0*) / (*b*$*0*)) = *r k* (*a*$*0*) / *r k* (*b* $ *0*) ⟷
    fps-radical r k (*a*/*b*) = *fps-radical r k a* / *fps-radical r k b*
  (**is** *?lhs* = *?rhs*)
proof
  let *?r* = *fps-radical r k*
  from *kp* **obtain** *h* **where** *k*: *k* = *Suc h*
    by (*cases k*) *auto*
  have *ra0'*: *r k* (*a*$*0*) ≠ *0* **using** *a0 ra0 k* **by** *auto*
  have *rb0'*: *r k* (*b*$*0*) ≠ *0* **using** *b0 rb0 k* **by** *auto*

  show *?lhs* **if** *?rhs*
  proof −
    from *that* **have** *?r* (*a*/*b*) $ *0* = (*?r a* / *?r b*)$*0*
      by *simp*
    then show *?thesis*
      using *k a0 b0 rb0'* **by** (*simp add*: *fps-divide-unit fps-mult-nth fps-inverse-def divide-inverse*)
  qed
  show *?rhs* **if** *?lhs*
  proof −

from *a0 b0* **have** *ab0[simp]*: *(a/b)$0 = a$0 / b$0*
by (*simp add*: *fps-divide-def fps-mult-nth divide-inverse fps-inverse-def*)
have *th0*: *r k ((a/b)$0) ^ k = (a/b)$0*
by (*simp add*: ‹*?lhs*› *power-divide ra0 rb0*)
from *a0 b0 ra0' rb0' kp* ‹*?lhs*›
have *th1*: *r k ((a / b) $ 0) = (fps-radical r k a / fps-radical r k b) $ 0*
by (*simp add*: *fps-divide-unit fps-mult-nth fps-inverse-def divide-inverse*)
from *a0 b0 ra0' rb0' kp* **have** *ab0'*: *(a / b) $ 0* ≠ *0*
by (*simp add*: *fps-divide-unit fps-mult-nth fps-inverse-def nonzero-imp-inverse-nonzero*)
note *tha[simp] = iffD1[OF power-radical[***where** *r=r* **and** *k=h*], *OF a0 ra0[unfolded k], unfolded k[symmetric]]*
note *thb[simp] = iffD1[OF power-radical[***where** *r=r* **and** *k=h*], *OF b0 rb0[unfolded k], unfolded k[symmetric]]*
from *b0 rb0'* **have** *th2*: *(?r a / ?r b)^k = a/b*
by (*simp add*: *fps-divide-unit power-mult-distrib fps-inverse-power[symmetric]*)

from *iffD1[OF radical-unique[***where** *r=r* **and** *a=?r a / ?r b* **and** *b=a/b* **and** *k=h*], *symmetric*, *unfolded k[symmetric]*, *OF th0 th1 ab0' th2*]
show *?thesis* **.**
qed
qed

lemma *radical-inverse*:
fixes *a* :: *'a::field-char-0 fps*
assumes *k*: *k > 0*
and *ra0*: *r k (a $ 0) ^ k = a $ 0*
and *r1*: *(r k 1)^k = 1*
and *a0*: *a$0* ≠ *0*
shows *r k (inverse (a $ 0)) = r k 1 / (r k (a $ 0))* ⟷
fps-radical r k (inverse a) = fps-radical r k 1 / fps-radical r k a
using *radical-divide[***where** *k=k* **and** *r=r* **and** *a=1* **and** *b=a*, *OF k*] *ra0 r1 a0*
by (*simp add*: *divide-inverse fps-divide-def*)

43.18 Derivative of composition

lemma *fps-compose-deriv*:
fixes *a* :: *'a::idom fps*
assumes *b0*: *b$0 = 0*
shows *fps-deriv (a oo b) = ((fps-deriv a) oo b) * fps-deriv b*
proof −
have *(fps-deriv (a oo b))$n = (((fps-deriv a) oo b) * (fps-deriv b)) $n* **for** *n*
proof −
have *(fps-deriv (a oo b))$n = setsum (λi. a $ i * (fps-deriv (b^i))$n) {0.. Suc n}*
by (*simp add*: *fps-compose-def field-simps setsum-right-distrib del*: *of-nat-Suc*)
also have ... *= setsum (λi. a$i * ((fps-const (of-nat i)) * (fps-deriv b * (b^(i − 1))))$n) {0.. Suc n}*
by (*simp add*: *field-simps fps-deriv-power del*: *fps-mult-left-const-nth of-nat-Suc*)
also have ... *= setsum (λi. of-nat i * a$i * (((b^(i − 1)) * fps-deriv b))$n)*

{0.. Suc n}
unfolding *fps-mult-left-const-nth* **by** (*simp add*: *field-simps*)
also have ... = *setsum* (λ*i*. *of-nat i* ∗ *a*\$*i* ∗ (*setsum* (λ*j*. (*b*ˆ (*i* − *1*))\$*j* ∗ (*fps-deriv b*)\$(*n* − *j*)) {*0*..*n*})) {*0*.. *Suc n*}
unfolding *fps-mult-nth* **..**
also have ... = *setsum* (λ*i*. *of-nat i* ∗ *a*\$*i* ∗ (*setsum* (λ*j*. (*b*ˆ (*i* − *1*))\$*j* ∗ (*fps-deriv b*)\$(*n* − *j*)) {*0*..*n*})) {*1*.. *Suc n*}
apply (*rule setsum.mono-neutral-right*)
apply (*auto simp add*: *mult-delta-left setsum.delta not-le*)
done
also have ... = *setsum* (λ*i*. *of-nat* (*i* + *1*) ∗ *a*\$(*i*+*1*) ∗ (*setsum* (λ*j*. (*b*ˆ *i*)\$*j* ∗ *of-nat* (*n* − *j* + *1*) ∗ *b*\$(*n* − *j* + *1*)) {*0*..*n*})) {*0*.. *n*}
unfolding *fps-deriv-nth*
by (*rule setsum.reindex-cong* [*of Suc*]) (*auto simp add*: *mult.assoc*)
finally have *th0*: (*fps-deriv* (*a oo b*))\$*n* =
setsum (λ*i*. *of-nat* (*i* + *1*) ∗ *a*\$(*i*+*1*) ∗ (*setsum* (λ*j*. (*b*ˆ *i*)\$*j* ∗ *of-nat* (*n* − *j* + *1*) ∗ *b*\$(*n* − *j* + *1*)) {*0*..*n*})) {*0*.. *n*} **.**

have (((*fps-deriv a*) *oo b*) ∗ (*fps-deriv b*))\$*n* = *setsum* (λ*i*. (*fps-deriv b*)\$ (*n* − *i*) ∗ ((*fps-deriv a*) *oo b*)\$*i*) {*0*..*n*}
unfolding *fps-mult-nth* **by** (*simp add*: *ac-simps*)
also have ... = *setsum* (λ*i*. *setsum* (λ*j*. *of-nat* (*n* − *i* +*1*) ∗ *b*\$(*n* − *i* + *1*) ∗ *of-nat* (*j* + *1*) ∗ *a*\$(*j*+*1*) ∗ (*b*ˆ*j*)\$*i*) {*0*..*n*}) {*0*..*n*}
unfolding *fps-deriv-nth fps-compose-nth setsum-right-distrib mult.assoc*
apply (*rule setsum.cong*)
apply (*rule refl*)
apply (*rule setsum.mono-neutral-left*)
apply (*simp-all add*: *subset-eq*)
apply *clarify*
apply (*subgoal-tac b*ˆ*i*\$*x* = *0*)
apply *simp*
apply (*rule startsby-zero-power-prefix*[*OF b0*, *rule-format*])
apply *simp*
done
also have ... = *setsum* (λ*i*. *of-nat* (*i* + *1*) ∗ *a*\$(*i*+*1*) ∗ (*setsum* (λ*j*. (*b*ˆ *i*)\$*j* ∗ *of-nat* (*n* − *j* + *1*) ∗ *b*\$(*n* − *j* + *1*)) {*0*..*n*})) {*0*.. *n*}
unfolding *setsum-right-distrib*
apply (*subst setsum.commute*)
apply (*rule setsum.cong*, *rule refl*)+
apply *simp*
done
finally show *?thesis*
unfolding *th0* **by** *simp*
qed
then show *?thesis* **by** (*simp add*: *fps-eq-iff*)
qed

lemma *fps-mult-X-plus-1-nth*:
((*1*+*X*)∗*a*) \$*n* = (*if n* = *0 then* (*a*\$*n* :: '*a*::*comm-ring-1*) *else a*\$*n* + *a*\$(*n* − *1*))

proof (*cases n*)
  case *0*
  then show *?thesis*
    by (*simp add*: *fps-mult-nth*)
next
  case (*Suc m*)
  have $((1 + X)*a)\ \$\ n = setsum\ (\lambda i.\ (1 + X)\ \$\ i * a\ \$\ (n - i))\ \{0..n\}$
    by (*simp add*: *fps-mult-nth*)
  also have $\ldots = setsum\ (\lambda i.\ (1{+}X)\$i * a\$(n{-}i))\ \{0..\ 1\}$
    unfolding *Suc* **by** (*rule setsum.mono-neutral-right*) *auto*
  also have $\ldots = (if\ n = 0\ then\ (a\$n :: 'a::comm\text{-}ring\text{-}1)\ else\ a\$n + a\$(n - 1))$
    by (*simp add*: *Suc*)
  finally show *?thesis* **.**
qed

43.19 Finite FPS (i.e. polynomials) and X

lemma *fps-poly-sum-X*:
  assumes $\forall\, i > n.\ a\$i = (0::'a::comm\text{-}ring\text{-}1)$
  shows $a = setsum\ (\lambda i.\ fps\text{-}const\ (a\$i) * X\hat{\ }i)\ \{0..n\}$ (**is** $a = ?r$)
proof −
  have $a\$i = ?r\$i$ **for** *i*
    unfolding *fps-setsum-nth fps-mult-left-const-nth X-power-nth*
    by (*simp add*: *mult-delta-right setsum.delta′ assms*)
  then show *?thesis*
    unfolding *fps-eq-iff* **by** *blast*
qed

43.20 Compositional inverses

fun *compinv* :: $'a\ fps \Rightarrow nat \Rightarrow 'a::field$
where
  $compinv\ a\ 0 = X\$0$
| $compinv\ a\ (Suc\ n) =$
    $(X\$\ Suc\ n - setsum\ (\lambda i.\ (compinv\ a\ i) * (a\hat{\ }i)\$Suc\ n)\ \{0\ ..\ n\})\ /\ (a\$1)\ \hat{\ }\ Suc$
n

definition $fps\text{-}inv\ a = Abs\text{-}fps\ (compinv\ a)$

lemma *fps-inv*:
  assumes $a0$: $a\$0 = 0$
    and $a1$: $a\$1 \neq 0$
  shows $fps\text{-}inv\ a\ oo\ a = X$
proof −
  let $?i = fps\text{-}inv\ a\ oo\ a$
  have $?i\ \$n = X\$n$ **for** *n*
  proof (*induct n rule*: *nat-less-induct*)
    fix *n*
    assume h: $\forall\, m{<}n.\ ?i\$m = X\$m$
    show $?i\ \$\ n = X\$n$

```
  proof (cases n)
    case 0
    then show ?thesis using a0
      by (simp add: fps-compose-nth fps-inv-def)
  next
    case (Suc n1)
    have ?i $ n = setsum (λi. (fps-inv a $ i) * (a^i)$n) {0 .. n1} + fps-inv a $
Suc n1 * (a $ 1)^ Suc n1
      by (simp only: fps-compose-nth) (simp add: Suc startsby-zero-power-nth-same
[OF a0] del: power-Suc)
    also have ... = setsum (λi. (fps-inv a $ i) * (a^i)$n) {0 .. n1} +
      (X$ Suc n1 - setsum (λi. (fps-inv a $ i) * (a^i)$n) {0 .. n1})
      using a0 a1 Suc by (simp add: fps-inv-def)
    also have ... = X$n using Suc by simp
    finally show ?thesis .
  qed
 qed
 then show ?thesis
  by (simp add: fps-eq-iff)
qed


fun gcompinv :: 'a fps ⇒ 'a fps ⇒ nat ⇒ 'a::field
where
  gcompinv b a 0 = b$0
| gcompinv b a (Suc n) =
    (b$ Suc n - setsum (λi. (gcompinv b a i) * (a^i)$Suc n) {0 .. n}) / (a$1) ^
Suc n

definition fps-ginv b a = Abs-fps (gcompinv b a)

lemma fps-ginv:
 assumes a0: a$0 = 0
  and a1: a$1 ≠ 0
 shows fps-ginv b a oo a = b
proof -
 let ?i = fps-ginv b a oo a
 have ?i $n = b$n for n
 proof (induct n rule: nat-less-induct)
  fix n
  assume h: ∀ m<n. ?i$m = b$m
  show ?i $ n = b$n
  proof (cases n)
    case 0
    then show ?thesis using a0
      by (simp add: fps-compose-nth fps-ginv-def)
  next
    case (Suc n1)
    have ?i $ n = setsum (λi. (fps-ginv b a $ i) * (a^i)$n) {0 .. n1} + fps-ginv
```

b a \$ *Suc n1* ∗ (*a* \$ *1*) ^ *Suc n1*
by (*simp only*: *fps-compose-nth*) (*simp add*: *Suc startsby-zero-power-nth-same* [*OF a0*] *del*: *power-Suc*)
also have . . . = *setsum* (λ*i*. (*fps-ginv b a* \$ *i*) ∗ (*a*^*i*)\$*n*) {*0* .. *n1*} +
(*b*\$ *Suc n1* − *setsum* (λ*i*. (*fps-ginv b a* \$ *i*) ∗ (*a*^*i*)\$*n*) {*0* .. *n1*})
using *a0 a1 Suc* **by** (*simp add*: *fps-ginv-def*)
also have . . . = *b*\$*n* **using** *Suc* **by** *simp*
finally show *?thesis* **.**
qed
qed
then show *?thesis*
by (*simp add*: *fps-eq-iff*)
qed

lemma *fps-inv-ginv*: *fps-inv* = *fps-ginv X*
apply (*auto simp add*: *fun-eq-iff fps-eq-iff fps-inv-def fps-ginv-def*)
apply (*induct-tac n rule*: *nat-less-induct*)
apply *auto*
apply (*case-tac na*)
apply *simp*
apply *simp*
done

lemma *fps-compose-1*[*simp*]: *1 oo a* = *1*
by (*simp add*: *fps-eq-iff fps-compose-nth mult-delta-left setsum.delta*)

lemma *fps-compose-0*[*simp*]: *0 oo a* = *0*
by (*simp add*: *fps-eq-iff fps-compose-nth*)

lemma *fps-compose-0-right*[*simp*]: *a oo 0* = *fps-const* (*a* \$ *0*)
by (*auto simp add*: *fps-eq-iff fps-compose-nth power-0-left setsum.neutral*)

lemma *fps-compose-add-distrib*: (*a* + *b*) *oo c* = (*a oo c*) + (*b oo c*)
by (*simp add*: *fps-eq-iff fps-compose-nth field-simps setsum.distrib*)

lemma *fps-compose-setsum-distrib*: (*setsum f S*) *oo a* = *setsum* (λ*i*. *f i oo a*) *S*
proof (*cases finite S*)
case *True*
show *?thesis*
proof (*rule finite-induct*[*OF True*])
show *setsum f* {} *oo a* = (∑ *i*∈{}. *f i oo a*)
by *simp*
next
fix *x F*
assume *fF*: *finite F*
and *xF*: *x* ∉ *F*
and *h*: *setsum f F oo a* = *setsum* (λ*i*. *f i oo a*) *F*
show *setsum f* (*insert x F*) *oo a* = *setsum* (λ*i*. *f i oo a*) (*insert x F*)
using *fF xF h* **by** (*simp add*: *fps-compose-add-distrib*)

```
  qed
next
  case False
  then show ?thesis by simp
qed

lemma convolution-eq:
  setsum (λi. a (i :: nat) * b (n − i)) {0 .. n} =
    setsum (λ(i,j). a i * b j) {(i,j). i ≤ n ∧ j ≤ n ∧ i + j = n}
  by (rule setsum.reindex-bij-witness[where i=fst and j=λi. (i, n − i)]) auto

lemma product-composition-lemma:
  assumes c0: c$0 = (0::'a::idom)
    and d0: d$0 = 0
  shows ((a oo c) * (b oo d))$n =
    setsum (λ(k,m). a$k * b$m * (c^k * d^m) $ n) {(k,m). k + m ≤ n}  (is ?l =
?r)
proof −
  let ?S = {(k::nat, m::nat). k + m ≤ n}
  have s: ?S ⊆ {0..n} × {0..n} by (auto simp add: subset-eq)
  have f: finite {(k::nat, m::nat). k + m ≤ n}
    apply (rule finite-subset[OF s])
    apply auto
    done
  have ?r =  setsum (λi. setsum (λ(k,m). a$k * (c^k)$i * b$m * (d^m) $ (n −
i)) {(k,m). k + m ≤ n}) {0..n}
    apply (simp add: fps-mult-nth setsum-right-distrib)
    apply (subst setsum.commute)
    apply (rule setsum.cong)
    apply (auto simp add: field-simps)
    done
  also have ... = ?l
    apply (simp add: fps-mult-nth fps-compose-nth setsum-product)
    apply (rule setsum.cong)
    apply (rule refl)
    apply (simp add: setsum.cartesian-product mult.assoc)
    apply (rule setsum.mono-neutral-right[OF f])
    apply (simp add: subset-eq)
    apply presburger
    apply clarsimp
    apply (rule ccontr)
    apply (clarsimp simp add: not-le)
    apply (case-tac x < aa)
    apply simp
    apply (frule-tac startsby-zero-power-prefix[rule-format, OF c0])
    apply blast
    apply simp
    apply (frule-tac startsby-zero-power-prefix[rule-format, OF d0])
    apply blast
```

```
    done
  finally show ?thesis by simp
qed

lemma product-composition-lemma':
  assumes c0: c$0 = (0::'a::idom)
    and d0: d$0 = 0
  shows ((a oo c) * (b oo d))$n =
    setsum (λk. setsum (λm. a$k * b$m * (c^k * d^m) $ n) {0..n}) {0..n}  (is ?l
= ?r)
  unfolding product-composition-lemma[OF c0 d0]
  unfolding setsum.cartesian-product
  apply (rule setsum.mono-neutral-left)
  apply simp
  apply (clarsimp simp add: subset-eq)
  apply clarsimp
  apply (rule ccontr)
  apply (subgoal-tac (c^aa * d^ba) $ n = 0)
  apply simp
  unfolding fps-mult-nth
  apply (rule setsum.neutral)
  apply (clarsimp simp add: not-le)
  apply (case-tac x < aa)
  apply (rule startsby-zero-power-prefix[OF c0, rule-format])
  apply simp
  apply (subgoal-tac n - x < ba)
  apply (frule-tac k = ba in startsby-zero-power-prefix[OF d0, rule-format])
  apply simp
  apply arith
  done


lemma setsum-pair-less-iff:
  setsum (λ((k::nat),m). a k * b m * c (k + m)) {(k,m). k + m ≤ n} =
    setsum (λs. setsum (λi. a i * b (s - i) * c s) {0..s}) {0..n}
  (is ?l = ?r)
proof -
  let ?KM = {(k,m). k + m ≤ n}
  let ?f = λs. UNION {(0::nat)..s} (λi. {(i,s - i)})
  have th0: ?KM = UNION {0..n} ?f
    by auto
  show ?l = ?r
    unfolding th0
    apply (subst setsum.UNION-disjoint)
    apply auto
    apply (subst setsum.UNION-disjoint)
    apply auto
    done
qed
```

lemma *fps-compose-mult-distrib-lemma*:
assumes *c0*: *c$0* = (*0*::'*a*::*idom*)
shows ((*a oo c*) ∗ (*b oo c*))$*n* = *setsum* (λ*s*. *setsum* (λ*i*. *a*$*i* ∗ *b*$(*s* − *i*) ∗ (*c*ˆ*s*) $ *n*) {*0*..*s*}) {*0*..*n*}
unfolding *product-composition-lemma*[*OF c0 c0*] *power-add*[*symmetric*]
unfolding *setsum-pair-less-iff* [**where** *a* = λ*k*. *a*$*k* **and** *b*=λ*m*. *b*$*m* **and** *c*=λ*s*. (*c* ˆ *s*)$*n* **and** *n* = *n*] **..**

lemma *fps-compose-mult-distrib*:
assumes *c0*: *c* $ *0* = (*0*::'*a*::*idom*)
shows (*a* ∗ *b*) *oo c* = (*a oo c*) ∗ (*b oo c*)
apply (*simp add*: *fps-eq-iff fps-compose-mult-distrib-lemma* [*OF c0*])
apply (*simp add*: *fps-compose-nth fps-mult-nth setsum-left-distrib*)
done

lemma *fps-compose-setprod-distrib*:
assumes *c0*: *c$0* = (*0*::'*a*::*idom*)
shows *setprod a S oo c* = *setprod* (λ*k*. *a k oo c*) *S*
apply (*cases finite S*)
apply *simp-all*
apply (*induct S rule*: *finite-induct*)
apply *simp*
apply (*simp add*: *fps-compose-mult-distrib*[*OF c0*])
done

lemma *fps-compose-power*:
assumes *c0*: *c$0* = (*0*::'*a*::*idom*)
shows (*a oo c*)ˆ*n* = *a*ˆ*n oo c*
proof (*cases n*)
case *0*
then show *?thesis* **by** *simp*
next
case (*Suc m*)
have *th0*: *a*ˆ*n* = *setprod* (λ*k*. *a*) {*0*..*m*} (*a oo c*) ˆ *n* = *setprod* (λ*k*. *a oo c*) {*0*..*m*}
by (*simp-all add*: *setprod-constant Suc*)
then show *?thesis*
by (*simp add*: *fps-compose-setprod-distrib*[*OF c0*])
qed

lemma *fps-compose-uminus*: − (*a*::'*a*::*ring-1 fps*) *oo c* = − (*a oo c*)
by (*simp add*: *fps-eq-iff fps-compose-nth field-simps setsum-negf*[*symmetric*])

lemma *fps-compose-sub-distrib*: (*a* − *b*) *oo* (*c*::'*a*::*ring-1 fps*) = (*a oo c*) − (*b oo c*)
using *fps-compose-add-distrib* [*of a* − *b c*] **by** (*simp add*: *fps-compose-uminus*)

lemma *X-fps-compose*: *X oo a* = *Abs-fps* (λ*n*. *if n* = *0 then* (*0*::'*a*::*comm-ring-1*)

else a$n)
  by (*simp add*: *fps-eq-iff fps-compose-nth mult-delta-left setsum.delta*)

lemma *fps-inverse-compose*:
  assumes *b0*: (*b$0* :: *'a*::*field*) = *0*
    and *a0*: *a$0* ≠ *0*
  shows *inverse a oo b* = *inverse* (*a oo b*)
proof −
  let *?ia* = *inverse a*
  let *?ab* = *a oo b*
  let *?iab* = *inverse ?ab*

  from *a0* **have** *ia0*: *?ia* $ *0* ≠ *0* **by** *simp*
  from *a0* **have** *ab0*: *?ab* $ *0* ≠ *0* **by** (*simp add*: *fps-compose-def*)
  have (*?ia oo b*) ∗ (*a oo b*) = *1*
    unfolding *fps-compose-mult-distrib*[*OF b0*, *symmetric*]
    unfolding *inverse-mult-eq-1*[*OF a0*]
    fps-compose-1 **..**

  then have (*?ia oo b*) ∗ (*a oo b*) ∗ *?iab* = *1* ∗ *?iab* **by** *simp*
  then have (*?ia oo b*) ∗ (*?iab* ∗ (*a oo b*)) = *?iab* **by** *simp*
  then show *?thesis* **unfolding** *inverse-mult-eq-1*[*OF ab0*] **by** *simp*
qed

lemma *fps-divide-compose*:
  assumes *c0*: (*c$0* :: *'a*::*field*) = *0*
    and *b0*: *b$0* ≠ *0*
  shows (*a*/*b*) *oo c* = (*a oo c*) / (*b oo c*)
   using *b0 c0* **by** (*simp add*: *fps-divide-unit fps-inverse-compose fps-compose-mult-distrib*)

lemma *gp*:
  assumes *a0*: *a$0* = (*0*::*'a*::*field*)
  shows (*Abs-fps* (λ*n*. *1*)) *oo a* = *1*/(*1* − *a*)
    (**is** *?one oo a* = -)
proof −
  have *o0*: *?one* $ *0* ≠ *0* **by** *simp*
  have *th0*: (*1* − *X*) $ *0* ≠ (*0*::*'a*) **by** *simp*
  from *fps-inverse-gp*[**where** *?'a* = *'a*]
  have *inverse ?one* = *1* − *X* **by** (*simp add*: *fps-eq-iff*)
  then have *inverse* (*inverse ?one*) = *inverse* (*1* − *X*) **by** *simp*
  then have *th*: *?one* = *1*/(*1* − *X*) **unfolding** *fps-inverse-idempotent*[*OF o0*]
    by (*simp add*: *fps-divide-def*)
  show *?thesis*
    unfolding *th*
    unfolding *fps-divide-compose*[*OF a0 th0*]
    fps-compose-1 fps-compose-sub-distrib X-fps-compose-startby0[*OF a0*] **..**
qed

lemma *fps-const-power* [*simp*]: *fps-const* (*c*::*'a*::*ring-1*) ^ *n* = *fps-const* (*c*^*n*)

by (*induct n*) *auto*

lemma *fps-compose-radical*:
assumes *b0*: *b$0 = (0::'a::field-char-0)*
and *ra0*: *r (Suc k) (a$0) ^ Suc k = a$0*
and *a0*: *a$0 ≠ 0*
shows *fps-radical r (Suc k) a oo b = fps-radical r (Suc k) (a oo b)*
proof –
let *?r = fps-radical r (Suc k)*
let *?ab = a oo b*
have *ab0*: *?ab $ 0 = a$0*
by (*simp add*: *fps-compose-def*)
from *ab0 a0 ra0* **have** *rab0*: *?ab $ 0 ≠ 0 r (Suc k) (?ab $ 0) ^ Suc k = ?ab $ 0*
by *simp-all*
have *th00*: *r (Suc k) ((a oo b) $ 0) = (fps-radical r (Suc k) a oo b) $ 0*
by (*simp add*: *ab0 fps-compose-def*)
have *th0*: *(?r a oo b) ^ (Suc k) = a oo b*
unfolding *fps-compose-power[OF b0]*
unfolding *iffD1[OF power-radical[of a r k], OF a0 ra0]* **..**
from *iffD1[OF radical-unique[***where** *r=r* **and** *k=k* **and** *b= ?ab* **and** *a = ?r a oo b, OF rab0(2) th00 rab0(1)], OF th0]*
show *?thesis* **.**
qed

lemma *fps-const-mult-apply-left*: *fps-const c * (a oo b) = (fps-const c * a) oo b*
by (*simp add*: *fps-eq-iff fps-compose-nth setsum-right-distrib mult.assoc*)

lemma *fps-const-mult-apply-right*:
*(a oo b) * fps-const (c::'a::comm-semiring-1) = (fps-const c * a) oo b*
by (*auto simp add*: *fps-const-mult-apply-left mult.commute*)

lemma *fps-compose-assoc*:
assumes *c0*: *c$0 = (0::'a::idom)*
and *b0*: *b$0 = 0*
shows *a oo (b oo c) = a oo b oo c* (**is** *?l = ?r*)
proof –
have *?l$n = ?r$n* **for** *n*
proof –
have *?l$n = (setsum (λi. (fps-const (a$i) * b^i) oo c) {0..n})$n*
by (*simp add*: *fps-compose-nth fps-compose-power[OF c0] fps-const-mult-apply-left setsum-right-distrib mult.assoc fps-setsum-nth*)
also have ... *= ((setsum (λi. fps-const (a$i) * b^i) {0..n}) oo c)$n*
by (*simp add*: *fps-compose-setsum-distrib*)
also have ... *= ?r$n*
apply (*simp add*: *fps-compose-nth fps-setsum-nth setsum-left-distrib mult.assoc*)
apply (*rule setsum.cong*)
apply (*rule refl*)
apply (*rule setsum.mono-neutral-right*)
apply (*auto simp add*: *not-le*)

```
    apply (erule startsby-zero-power-prefix[OF b0, rule-format])
    done
   finally show ?thesis .
  qed
  then show ?thesis
   by (simp add: fps-eq-iff)
qed


lemma fps-X-power-compose:
  assumes a0: a$0=0
  shows X^k oo a = (a::'a::idom fps)^k
  (is ?l = ?r)
proof (cases k)
  case 0
  then show ?thesis by simp
next
  case (Suc h)
  have ?l $ n = ?r $n for n
  proof −
   consider k > n | k ≤ n by arith
   then show ?thesis
   proof cases
    case 1
    then show ?thesis
     using a0 startsby-zero-power-prefix[OF a0] Suc
     by (simp add: fps-compose-nth del: power-Suc)
   next
    case 2
    then show ?thesis
     by (simp add: fps-compose-nth mult-delta-left setsum.delta)
   qed
  qed
  then show ?thesis
   unfolding fps-eq-iff by blast
qed

lemma fps-inv-right:
  assumes a0: a$0 = 0
   and a1: a$1 ≠ 0
  shows a oo fps-inv a = X
proof −
  let ?ia = fps-inv a
  let ?iaa = a oo fps-inv a
  have th0: ?ia $ 0 = 0
   by (simp add: fps-inv-def)
  have th1: ?iaa $ 0 = 0
   using a0 a1 by (simp add: fps-inv-def fps-compose-nth)
  have th2: X$0 = 0
```

```
    by simp
  from fps-inv[OF a0 a1] have a oo (fps-inv a oo a) = a oo X
    by simp
  then have (a oo fps-inv a) oo a = X oo a
    by (simp add: fps-compose-assoc[OF a0 th0] X-fps-compose-startby0[OF a0])
  with fps-compose-inj-right[OF a0 a1] show ?thesis
    by simp
qed

lemma fps-inv-deriv:
  assumes a0: a$0 = (0::'a::field)
    and a1: a$1 ≠ 0
  shows fps-deriv (fps-inv a) = inverse (fps-deriv a oo fps-inv a)
proof –
  let ?ia = fps-inv a
  let ?d = fps-deriv a oo ?ia
  let ?dia = fps-deriv ?ia
  have ia0: ?ia$0 = 0
    by (simp add: fps-inv-def)
  have th0: ?d$0 ≠ 0
    using a1 by (simp add: fps-compose-nth)
  from fps-inv-right[OF a0 a1] have ?d * ?dia = 1
    by (simp add: fps-compose-deriv[OF ia0, of a, symmetric] )
  then have inverse ?d * ?d * ?dia = inverse ?d * 1
    by simp
  with inverse-mult-eq-1 [OF th0] show ?dia = inverse ?d
    by simp
qed

lemma fps-inv-idempotent:
  assumes a0: a$0 = 0
    and a1: a$1 ≠ 0
  shows fps-inv (fps-inv a) = a
proof –
  let ?r = fps-inv
  have ra0: ?r a $ 0 = 0
    by (simp add: fps-inv-def)
  from a1 have ra1: ?r a $ 1 ≠ 0
    by (simp add: fps-inv-def field-simps)
  have X0: X$0 = 0
    by simp
  from fps-inv[OF ra0 ra1] have ?r (?r a) oo ?r a = X .
  then have ?r (?r a) oo ?r a oo a = X oo a
    by simp
  then have ?r (?r a) oo (?r a oo a) = a
    unfolding X-fps-compose-startby0[OF a0]
    unfolding fps-compose-assoc[OF a0 ra0, symmetric] .
  then show ?thesis
    unfolding fps-inv[OF a0 a1] by simp
```

qed

lemma *fps-ginv-ginv*:
  assumes *a0*: *a$0 = 0*
    and *a1*: *a$1 ≠ 0*
    and *c0*: *c$0 = 0*
    and *c1*: *c$1 ≠ 0*
  shows *fps-ginv b (fps-ginv c a) = b oo a oo fps-inv c*
proof −
  let *?r = fps-ginv*
  from *c0* **have** *rca0*: *?r c a $0 = 0*
    by (*simp add*: *fps-ginv-def*)
  from *a1 c1* **have** *rca1*: *?r c a $ 1 ≠ 0*
    by (*simp add*: *fps-ginv-def field-simps*)
  from *fps-ginv*[*OF rca0 rca1*]
  have *?r b (?r c a) oo ?r c a = b* **.**
  then have *?r b (?r c a) oo ?r c a oo a = b oo a*
    by *simp*
  then have *?r b (?r c a) oo (?r c a oo a) = b oo a*
    apply (*subst fps-compose-assoc*)
    using *a0 c0*
    apply (*auto simp add*: *fps-ginv-def*)
    done
  then have *?r b (?r c a) oo c = b oo a*
    unfolding *fps-ginv*[*OF a0 a1*] **.**
  then have *?r b (?r c a) oo c oo fps-inv c= b oo a oo fps-inv c*
    by *simp*
  then have *?r b (?r c a) oo (c oo fps-inv c) = b oo a oo fps-inv c*
    apply (*subst fps-compose-assoc*)
    using *a0 c0*
    apply (*auto simp add*: *fps-inv-def*)
    done
  then show *?thesis*
    unfolding *fps-inv-right*[*OF c0 c1*] **by** *simp*
qed

lemma *fps-ginv-deriv*:
  assumes *a0*:*a$0 = (0::'a::field)*
    and *a1*: *a$1 ≠ 0*
  shows *fps-deriv (fps-ginv b a) = (fps-deriv b / fps-deriv a) oo fps-ginv X a*
proof −
  let *?ia = fps-ginv b a*
  let *?iXa = fps-ginv X a*
  let *?d = fps-deriv*
  let *?dia = ?d ?ia*
  have *iXa0*: *?iXa $ 0 = 0*
    by (*simp add*: *fps-ginv-def*)
  have *da0*: *?d a $ 0 ≠ 0*
    using *a1* **by** *simp*

from *fps-ginv*[*OF a0 a1*, *of b*] **have** *?d* (*?ia oo a*) = *fps-deriv b*
by *simp*
then have (*?d ?ia oo a*) ∗ *?d a* = *?d b*
unfolding *fps-compose-deriv*[*OF a0*] **.**
then have (*?d ?ia oo a*) ∗ *?d a* ∗ *inverse* (*?d a*) = *?d b* ∗ *inverse* (*?d a*)
by *simp*
with *a1* **have** (*?d ?ia oo a*) ∗ (*inverse* (*?d a*) ∗ *?d a*) = *?d b* / *?d a*
by (*simp add*: *fps-divide-unit*)
then have (*?d ?ia oo a*) *oo ?iXa* = (*?d b* / *?d a*) *oo ?iXa*
unfolding *inverse-mult-eq-1*[*OF da0*] **by** *simp*
then have *?d ?ia oo* (*a oo ?iXa*) = (*?d b* / *?d a*) *oo ?iXa*
unfolding *fps-compose-assoc*[*OF iXa0 a0*] **.**
then show *?thesis* **unfolding** *fps-inv-ginv*[*symmetric*]
unfolding *fps-inv-right*[*OF a0 a1*] **by** *simp*
qed

43.21 Elementary series

43.21.1 Exponential series

definition *E x* = *Abs-fps* (λ*n*. *x*ˆ*n* / *of-nat* (*fact n*))

lemma *E-deriv*[*simp*]: *fps-deriv* (*E a*) = *fps-const* (*a*::′*a*::*field-char-0*) ∗ *E a* (**is** *?l* = *?r*)
proof −
have *?l*$*n* = *?r* $ *n* **for** *n*
apply (*auto simp add*: *E-def field-simps power-Suc*[*symmetric*]
simp del: *fact.simps of-nat-Suc power-Suc*)
apply (*simp add*: *of-nat-mult field-simps*)
done
then show *?thesis*
by (*simp add*: *fps-eq-iff*)
qed

lemma *E-unique-ODE*:
fps-deriv a = *fps-const c* ∗ *a* ⟷ *a* = *fps-const* (*a*$*0*) ∗ *E* (*c*::′*a*::*field-char-0*)
(**is** *?lhs* ⟷ *?rhs*)
proof
show *?rhs* **if** *?lhs*
proof −
from *that* **have** *th*: ⋀*n*. *a* $ *Suc n* = *c* ∗ *a*$*n* / *of-nat* (*Suc n*)
by (*simp add*: *fps-deriv-def fps-eq-iff field-simps del*: *of-nat-Suc*)
have *th*′: *a*$*n* = *a*$*0* ∗ *c* ˆ *n*/ (*fact n*) **for** *n*
proof (*induct n*)
case *0*
then show *?case* **by** *simp*
next
case *Suc*
then show *?case*
unfolding *th*

using *fact-gt-zero*
apply (*simp add*: *field-simps del*: *of-nat-Suc fact-Suc*)
apply *simp*
done
qed
show *?thesis*
by (*auto simp add*: *fps-eq-iff fps-const-mult-left E-def intro*: *th'*)
qed
show *?lhs* **if** *?rhs*
using *that* **by** (*metis E-deriv fps-deriv-mult-const-left mult.left-commute*)
qed

lemma *E-add-mult*: $E\ (a + b) = E\ (a::'a::field\text{-}char\text{-}0) * E\ b$ (**is** *?l* = *?r*)
proof −
have *fps-deriv ?r* = *fps-const* $(a + b) * ?r$
by (*simp add*: *fps-const-add*[*symmetric*] *field-simps del*: *fps-const-add*)
then have *?r* = *?l*
by (*simp only*: *E-unique-ODE*) (*simp add*: *fps-mult-nth E-def*)
then show *?thesis* **..**
qed

lemma *E-nth*[*simp*]: $E\ a\ \$\ n = a\hat{}n\ /\ of\text{-}nat\ (fact\ n)$
by (*simp add*: *E-def*)

lemma *E0*[*simp*]: $E\ (0::'a::field) = 1$
by (*simp add*: *fps-eq-iff power-0-left*)

lemma *E-neg*: $E\ (-\ a) = inverse\ (E\ (a::'a::field\text{-}char\text{-}0))$
proof −
from *E-add-mult*[*of a* − *a*] **have** *th0*: $E\ a * E\ (-\ a) = 1$ **by** *simp*
from *fps-inverse-unique*[*OF th0*] **show** *?thesis* **by** *simp*
qed

lemma *E-nth-deriv*[*simp*]: *fps-nth-deriv n* $(E\ (a::'a::field\text{-}char\text{-}0)) = (fps\text{-}const\ a)\hat{}n * (E\ a)$
by (*induct n*) *auto*

lemma *X-compose-E*[*simp*]: $X\ oo\ E\ (a::'a::field) = E\ a - 1$
by (*simp add*: *fps-eq-iff X-fps-compose*)

lemma *LE-compose*:
assumes *a*: $a \neq 0$
shows *fps-inv* $(E\ a - 1)\ oo\ (E\ a - 1) = X$
and $(E\ a - 1)\ oo$ *fps-inv* $(E\ a - 1) = X$
proof −
let $?b = E\ a - 1$
have *b0*: $?b\ \$\ 0 = 0$
by *simp*
have *b1*: $?b\ \$\ 1 \neq 0$

by (*simp add*: *a*)
from *fps-inv*[*OF b0 b1*] **show** *fps-inv* (*E a* − *1*) *oo* (*E a* − *1*) = *X* **.**
from *fps-inv-right*[*OF b0 b1*] **show** (*E a* − *1*) *oo fps-inv* (*E a* − *1*) = *X* **.**
qed

lemma *E-power-mult*: (*E* (*c*::′*a*::*field-char-0*))ˆ*n* = *E* (*of-nat n* ∗ *c*)
by (*induct n*) (*auto simp add*: *field-simps E-add-mult*)

lemma *radical-E*:
assumes *r*: *r* (*Suc k*) *1* = *1*
shows *fps-radical r* (*Suc k*) (*E* (*c*::′*a*::*field-char-0*)) = *E* (*c* / *of-nat* (*Suc k*))
proof −
let *?ck* = (*c* / *of-nat* (*Suc k*))
let *?r* = *fps-radical r* (*Suc k*)
have *eq0*[*simp*]: *?ck* ∗ *of-nat* (*Suc k*) = *c of-nat* (*Suc k*) ∗ *?ck* = *c*
by (*simp-all del*: *of-nat-Suc*)
have *th0*: *E ?ck* ˆ (*Suc k*) = *E c* **unfolding** *E-power-mult eq0* **..**
have *th*: *r* (*Suc k*) (*E c* $*0*) ˆ *Suc k* = *E c* $ *0*
r (*Suc k*) (*E c* $ *0*) = *E ?ck* $ *0 E c* $ *0* ≠ *0* **using** *r* **by** *simp-all*
from *th0 radical-unique*[**where** *r*=*r* **and** *k*=*k*, *OF th*] **show** *?thesis*
by *auto*
qed

lemma *Ec-E1-eq*: *E* (*1*::′*a*::*field-char-0*) *oo* (*fps-const c* ∗ *X*) = *E c*
apply (*auto simp add*: *fps-eq-iff E-def fps-compose-def power-mult-distrib*)
apply (*simp add*: *cond-value-iff cond-application-beta setsum.delta′ cong del*: *if-weak-cong*)
done

43.21.2 Logarithmic series

lemma *Abs-fps-if-0*:
Abs-fps (λ*n*. *if n* = *0 then* (*v*::′*a*::*ring-1*) *else f n*) =
fps-const v + *X* ∗ *Abs-fps* (λ*n*. *f* (*Suc n*))
by (*auto simp add*: *fps-eq-iff*)

definition *L* :: ′*a*::*field-char-0* ⇒ ′*a fps*
where *L c* = *fps-const* (*1*/*c*) ∗ *Abs-fps* (λ*n*. *if n* = *0 then 0 else* (− *1*) ˆ (*n* − *1*) / *of-nat n*)

lemma *fps-deriv-L*: *fps-deriv* (*L c*) = *fps-const* (*1*/*c*) ∗ *inverse* (*1* + *X*)
unfolding *fps-inverse-X-plus1*
by (*simp add*: *L-def fps-eq-iff del*: *of-nat-Suc*)

lemma *L-nth*: *L c* $ *n* = (*if n* = *0 then 0 else 1*/*c* ∗ ((− *1*) ˆ (*n* − *1*) / *of-nat n*))
by (*simp add*: *L-def field-simps*)

lemma *L-0*[*simp*]: *L c* $ *0* = *0* **by** (*simp add*: *L-def*)

lemma *L-E-inv*:
  fixes *a* :: *'a::field-char-0*
  assumes *a*: $a \neq 0$
  shows *L a* = *fps-inv* (*E a* − *1*) (**is** *?l* = *?r*)
proof −
  let *?b* = *E a* − *1*
  have *b0*: *?b* $ *0* = *0* **by** *simp*
  have *b1*: *?b* $ *1* ≠ *0* **by** (*simp add*: *a*)
  have *fps-deriv* (*E a* − *1*) *oo fps-inv* (*E a* − *1*) =
    (*fps-const a* ∗ (*E a* − *1*) + *fps-const a*) *oo fps-inv* (*E a* − *1*)
    by (*simp add*: *field-simps*)
  also have ... = *fps-const a* ∗ (*X* + *1*)
    apply (*simp add*: *fps-compose-add-distrib fps-const-mult-apply-left*[*symmetric*] *fps-inv-right*[*OF b0 b1*])
    apply (*simp add*: *field-simps*)
    done
  finally have *eq*: *fps-deriv* (*E a* − *1*) *oo fps-inv* (*E a* − *1*) = *fps-const a* ∗ (*X* + *1*) **.**
  from *fps-inv-deriv*[*OF b0 b1*, *unfolded eq*]
  have *fps-deriv* (*fps-inv ?b*) = *fps-const* (*inverse a*) / (*X* + *1*)
    using *a*
    by (*simp add*: *fps-const-inverse eq fps-divide-def fps-inverse-mult*)
  then have *fps-deriv ?l* = *fps-deriv ?r*
    by (*simp add*: *fps-deriv-L add.commute fps-divide-def divide-inverse*)
  then show *?thesis* **unfolding** *fps-deriv-eq-iff*
    by (*simp add*: *L-nth fps-inv-def*)
qed

lemma *L-mult-add*:
  assumes *c0*: *c*≠*0*
    and *d0*: *d*≠*0*
  shows *L c* + *L d* = *fps-const* (*c*+*d*) ∗ *L* (*c*∗*d*)
  (**is** *?r* = *?l*)
proof−
  from *c0 d0* **have** *eq*: *1*/*c* + *1*/*d* = (*c*+*d*)/(*c*∗*d*) **by** (*simp add*: *field-simps*)
  have *fps-deriv ?r* = *fps-const* (*1*/*c* + *1*/*d*) ∗ *inverse* (*1* + *X*)
   by (*simp add*: *fps-deriv-L fps-const-add*[*symmetric*] *algebra-simps del*: *fps-const-add*)
  also have ... = *fps-deriv ?l*
    apply (*simp add*: *fps-deriv-L*)
    apply (*simp add*: *fps-eq-iff eq*)
    done
  finally show *?thesis*
    unfolding *fps-deriv-eq-iff* **by** *simp*
qed

43.21.3 Binomial series

definition *fps-binomial a* = *Abs-fps* (λ*n. a gchoose n*)

lemma *fps-binomial-nth*[*simp*]: *fps-binomial a* $ *n* = *a gchoose n*
by (*simp add*: *fps-binomial-def*)

lemma *fps-binomial-ODE-unique*:
fixes *c* :: *'a::field-char-0*
shows *fps-deriv a* = (*fps-const c* ∗ *a*) / (*1* + *X*) ⟷ *a* = *fps-const* (*a*$*0*) ∗ *fps-binomial c*
(**is** *?lhs* ⟷ *?rhs*)
proof
let *?da* = *fps-deriv a*
let *?x1* = (*1* + *X*):: *'a fps*
let *?l* = *?x1* ∗ *?da*
let *?r* = *fps-const c* ∗ *a*

have *eq*: *?l* = *?r* ⟷ *?lhs*
proof −
have *x10*: *?x1* $ *0* ≠ *0* **by** *simp*
have *?l* = *?r* ⟷ *inverse ?x1* ∗ *?l* = *inverse ?x1* ∗ *?r* **by** *simp*
also have ... ⟷ *?da* = (*fps-const c* ∗ *a*) / *?x1*
apply (*simp only*: *fps-divide-def mult.assoc*[*symmetric*] *inverse-mult-eq-1*[*OF x10*])
apply (*simp add*: *field-simps*)
done
finally show *?thesis* **.**
qed

show *?rhs* **if** *?lhs*
proof −
from *eq that* **have** *h*: *?l* = *?r* **..**
have *th0*: *a*$ *Suc n* = ((*c* − *of-nat n*) / *of-nat* (*Suc n*)) ∗ *a* $*n* **for** *n*
proof −
from *h* **have** *?l* $ *n* = *?r* $ *n* **by** *simp*
then show *?thesis*
apply (*simp add*: *field-simps del*: *of-nat-Suc*)
apply (*cases n*)
apply (*simp-all add*: *field-simps del*: *of-nat-Suc*)
done
qed
have *th1*: *a* $ *n* = (*c gchoose n*) ∗ *a* $ *0* **for** *n*
proof (*induct n*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc m*)
then show *?case*
unfolding *th0*
apply (*simp add*: *field-simps del*: *of-nat-Suc*)
unfolding *mult.assoc*[*symmetric*] *gbinomial-mult-1*

```
        apply (simp add: field-simps)
        done
    qed
    show ?thesis
      apply (simp add: fps-eq-iff)
      apply (subst th1)
      apply (simp add: field-simps)
      done
  qed

  show ?lhs if ?rhs
  proof -
    have th00: x * (a $ 0 * y) = a $ 0 * (x * y) for x y
      by (simp add: mult.commute)
    have ?l = ?r
      apply (subst ‹?rhs›)
      apply (subst (2) ‹?rhs›)
      apply (clarsimp simp add: fps-eq-iff field-simps)
      unfolding mult.assoc[symmetric] th00 gbinomial-mult-1
      apply (simp add: field-simps gbinomial-mult-1)
      done
    with eq show ?thesis ..
  qed
qed

lemma fps-binomial-deriv: fps-deriv (fps-binomial c) = fps-const c * fps-binomial
c / (1 + X)
proof -
  let ?a = fps-binomial c
  have th0: ?a = fps-const (?a$0) * ?a by (simp)
  from iffD2[OF fps-binomial-ODE-unique, OF th0] show ?thesis .
qed

lemma fps-binomial-add-mult: fps-binomial (c+d) = fps-binomial c * fps-binomial
d (is ?l = ?r)
proof -
  let ?P = ?r - ?l
  let ?b = fps-binomial
  let ?db = λx. fps-deriv (?b x)
  have fps-deriv ?P = ?db c * ?b d + ?b c * ?db d - ?db (c + d)  by simp
  also have ... = inverse (1 + X) *
      (fps-const c * ?b c * ?b d + fps-const d * ?b c * ?b d - fps-const (c+d) * ?b
(c + d))
    unfolding fps-binomial-deriv
    by (simp add: fps-divide-def field-simps)
  also have ... = (fps-const (c + d)/ (1 + X)) * ?P
   by (simp add: field-simps fps-divide-unit fps-const-add[symmetric] del: fps-const-add)
  finally have th0: fps-deriv ?P = fps-const (c+d) * ?P / (1 + X)
    by (simp add: fps-divide-def)
```

have *?P = fps-const (?P$0) ∗ ?b (c + d)*
unfolding *fps-binomial-ODE-unique[symmetric]*
using *th0* **by** *simp*
then have *?P = 0* **by** *(simp add: fps-mult-nth)*
then show *?thesis* **by** *simp*
qed

lemma *fps-binomial-minus-one*: *fps-binomial (− 1) = inverse (1 + X)*
(**is** *?l = inverse ?r*)
proof−
have *th*: *?r$0 ≠ 0* **by** *simp*
have *th'*: *fps-deriv (inverse ?r) = fps-const (− 1) ∗ inverse ?r / (1 + X)*
by *(simp add: fps-inverse-deriv[OF th] fps-divide-def*
power2-eq-square mult.commute fps-const-neg[symmetric] del: fps-const-neg)
have *eq*: *inverse ?r $ 0 = 1*
by *(simp add: fps-inverse-def)*
from *iffD1[OF fps-binomial-ODE-unique[of inverse (1 + X) − 1] th'] eq*
show *?thesis* **by** *(simp add: fps-inverse-def)*
qed

Vandermonde's Identity as a consequence.

lemma *gbinomial-Vandermonde*:
setsum (λk. (a gchoose k) ∗ (b gchoose (n − k))) {0..n} = (a + b) gchoose n
proof −
let *?ba = fps-binomial a*
let *?bb = fps-binomial b*
let *?bab = fps-binomial (a + b)*
from *fps-binomial-add-mult[of a b]* **have** *?bab $ n = (?ba ∗ ?bb)$n* **by** *simp*
then show *?thesis* **by** *(simp add: fps-mult-nth)*
qed

lemma *binomial-Vandermonde*:
setsum (λk. (a choose k) ∗ (b choose (n − k))) {0..n} = (a + b) choose n
using *gbinomial-Vandermonde[of (of-nat a) of-nat b n]*
by *(simp only: binomial-gbinomial[symmetric] of-nat-mult[symmetric]*
of-nat-setsum[symmetric] of-nat-add[symmetric] of-nat-eq-iff)

lemma *binomial-Vandermonde-same*: *setsum (λk. (n choose k)^2) {0..n} = (2 ∗ n) choose n*
using *binomial-Vandermonde[of n n n, symmetric]*
unfolding *mult-2*
apply *(simp add: power2-eq-square)*
apply *(rule setsum.cong)*
apply *(auto intro: binomial-symmetric)*
done

lemma *Vandermonde-pochhammer-lemma*:
fixes *a :: 'a::field-char-0*
assumes *b*: *∀ j∈{0 ..<n}. b ≠ of-nat j*

```
  shows setsum (λk. (pochhammer (− a) k ∗ pochhammer (− (of-nat n)) k) /
      (of-nat (fact k) ∗ pochhammer (b − of-nat n + 1) k)) {0..n} =
    pochhammer (− (a + b)) n / pochhammer (− b) n
  (is ?l = ?r)
proof −
  let ?m1 = λm. (− 1 :: 'a) ^ m
  let ?f = λm. of-nat (fact m)
  let ?p = λ(x::'a). pochhammer (− x)
  from b have bn0: ?p b n ≠ 0
    unfolding pochhammer-eq-0-iff by simp
  have th00:
    b gchoose (n − k) =
        (?m1 n ∗ ?p b n ∗ ?m1 k ∗ ?p (of-nat n) k) / (?f n ∗ pochhammer (b −
of-nat n + 1) k)
      (is ?gchoose)
    pochhammer (1 + b − of-nat n) k ≠ 0
      (is ?pochhammer)
    if kn: k ∈ {0..n} for k
  proof −
    have nz: pochhammer (1 + b − of-nat n) n ≠ 0
    proof
      assume pochhammer (1 + b − of-nat n) n = 0
      then have c: pochhammer (b − of-nat n + 1) n = 0
        by (simp add: algebra-simps)
      then obtain j where j: j < n b − of-nat n + 1 = − of-nat j
        unfolding pochhammer-eq-0-iff by blast
      from j have b = of-nat n − of-nat j − of-nat 1
        by (simp add: algebra-simps)
      then have b = of-nat (n − j − 1)
        using j kn by (simp add: of-nat-diff)
      with b show False using j by auto
    qed

    from nz kn [simplified] have nz': pochhammer (1 + b − of-nat n) k ≠ 0
      by (rule pochhammer-neq-0-mono)

    consider k = 0 ∨ n = 0 | k ≠ 0 n ≠ 0
      by blast
    then have b gchoose (n − k) =
       (?m1 n ∗ ?p b n ∗ ?m1 k ∗ ?p (of-nat n) k) / (?f n ∗ pochhammer (b −
of-nat n + 1) k)
    proof cases
      case 1
      then show ?thesis
        using kn by (cases k = 0) (simp-all add: gbinomial-pochhammer)
    next
      case neq: 2
      then obtain m where m: n = Suc m
        by (cases n) auto
```

from *neq*(*1*) **obtain** *h* **where** *h*: *k* = *Suc h*
  by (*cases k*) *auto*
show *?thesis*
proof (*cases k* = *n*)
  case *True*
  then show *?thesis*
    using *pochhammer-minus'*[**where** *k*=*k* **and** *b*=*b*]
    apply (*simp add*: *pochhammer-same*)
    using *bn0*
    apply (*simp add*: *field-simps power-add*[*symmetric*])
    done
next
  case *False*
  with *kn* **have** *kn'*: *k* < *n*
    by *simp*
  have *m1nk*: *?m1 n* = *setprod* (λ*i*. − *1*) {*0*..*m*} *?m1 k* = *setprod* (λ*i*. − *1*) {*0*..*h*}
    by (*simp-all add*: *setprod-constant m h*)
  have *bnz0*: *pochhammer* (*b* − *of-nat n* + *1*) *k* ≠ *0*
    using *bn0 kn*
    unfolding *pochhammer-eq-0-iff*
    apply *auto*
    apply (*erule-tac x*= *n* − *ka* − *1* **in** *allE*)
    apply (*auto simp add*: *algebra-simps of-nat-diff*)
    done
  have *eq1*: *setprod* (λ*k*. (*1*::'*a*) + *of-nat m* − *of-nat k*) {*0* .. *h*} =
    setprod of-nat {*Suc* (*m* − *h*) .. *Suc m*}
    using *kn' h m*
    by (*intro setprod.reindex-bij-witness*[**where** *i*=λ*k*. *Suc m* − *k* **and** *j*=λ*k*. *Suc m* − *k*])
      (*auto simp*: *of-nat-diff*)

  have *th1*: (*?m1 k* ∗ *?p* (*of-nat n*) *k*) / *?f n* = *1* / *of-nat*(*fact* (*n* − *k*))
    unfolding *m1nk*
    unfolding *m h pochhammer-Suc-setprod*
    apply (*simp add*: *field-simps del*: *fact-Suc*)
    unfolding *fact-altdef id-def*
    unfolding *of-nat-setprod*
    unfolding *setprod.distrib*[*symmetric*]
    apply *auto*
    unfolding *eq1*
    apply (*subst setprod.union-disjoint*[*symmetric*])
    apply (*auto*)
    apply (*rule setprod.cong*)
    apply *auto*
    done
  have *th20*: *?m1 n* ∗ *?p b n* = *setprod* (λ*i*. *b* − *of-nat i*) {*0*..*m*}
    unfolding *m1nk*
    unfolding *m h pochhammer-Suc-setprod*

unfolding *setprod.distrib*[*symmetric*]
apply (*rule setprod.cong*)
apply *auto*
done
have *th21*:*pochhammer* (*b* − *of-nat n* + *1*) *k* = *setprod* (λ*i*. *b* − *of-nat i*) {*n* − *k* .. *n* − *1*}
unfolding *h m*
unfolding *pochhammer-Suc-setprod*
using *kn m h*
by (*intro setprod.reindex-bij-witness*[**where** *i*=λ*k*. *n* − *1* − *k* **and** *j*=λ*i*. *m*−*i*])
(*auto simp*: *of-nat-diff*)

have *?m1 n* ∗ *?p b n* =
pochhammer (*b* − *of-nat n* + *1*) *k* ∗ *setprod* (λ*i*. *b* − *of-nat i*) {*0*.. *n* − *k* − *1*}
unfolding *th20 th21*
unfolding *h m*
apply (*subst setprod.union-disjoint*[*symmetric*])
using *kn′ h m*
apply *auto*
apply (*rule setprod.cong*)
apply *auto*
done
then have *th2*: (*?m1 n* ∗ *?p b n*)/*pochhammer* (*b* − *of-nat n* + *1*) *k* =
setprod (λ*i*. *b* − *of-nat i*) {*0*.. *n* − *k* − *1*}
using *nz′* **by** (*simp add*: *field-simps*)
have (*?m1 n* ∗ *?p b n* ∗ *?m1 k* ∗ *?p* (*of-nat n*) *k*) / (*?f n* ∗ *pochhammer* (*b* − *of-nat n* + *1*) *k*) =
((*?m1 k* ∗ *?p* (*of-nat n*) *k*) / *?f n*) ∗ ((*?m1 n* ∗ *?p b n*)/*pochhammer* (*b* − *of-nat n* + *1*) *k*)
using *bnz0*
by (*simp add*: *field-simps*)
also have . . . = *b gchoose* (*n* − *k*)
unfolding *th1 th2*
using *kn′* **by** (*simp add*: *gbinomial-def*)
finally show *?thesis* **by** *simp*
qed
qed
then show *?gchoose* **and** *?pochhammer*
apply (*cases n* = *0*)
using *nz′*
apply *auto*
done
qed
have *?r* = ((*a* + *b*) *gchoose n*) ∗ (*of-nat* (*fact n*) / (*?m1 n* ∗ *pochhammer* (−*b*) *n*))
unfolding *gbinomial-pochhammer*
using *bn0* **by** (*auto simp add*: *field-simps*)

also have . . . = *?l*
unfolding *gbinomial-Vandermonde*[*symmetric*]
apply (*simp add*: *th00*)
unfolding *gbinomial-pochhammer*
using *bn0*
apply (*simp add*: *setsum-left-distrib setsum-right-distrib field-simps*)
apply (*rule setsum.cong*)
apply (*rule refl*)
apply (*drule th00*(*2*))
apply (*simp add*: *field-simps power-add*[*symmetric*])
done
finally show *?thesis* **by** *simp*
qed

lemma *Vandermonde-pochhammer*:
fixes *a* :: *'a::field-char-0*
assumes *c*: ∀ *i* ∈ {*0*..< *n*}. *c* ≠ − *of-nat i*
shows *setsum* (λ*k*. (*pochhammer a k* ∗ *pochhammer* (− (*of-nat n*)) *k*) /
(*of-nat* (*fact k*) ∗ *pochhammer c k*)) {*0*..*n*} = *pochhammer* (*c* − *a*) *n* / *pochhammer c n*
proof −
let *?a* = − *a*
let *?b* = *c* + *of-nat n* − *1*
have *h*: ∀ *j* ∈{*0*..< *n*}. *?b* ≠ *of-nat j*
using *c*
apply (*auto simp add*: *algebra-simps of-nat-diff*)
apply (*erule-tac x* = *n* − *j* − *1* **in** *ballE*)
apply (*auto simp add*: *of-nat-diff algebra-simps*)
done
have *th0*: *pochhammer* (− (*?a* + *?b*)) *n* = (− *1*) ˆ*n* ∗ *pochhammer* (*c* − *a*) *n*
unfolding *pochhammer-minus*
by (*simp add*: *algebra-simps*)
have *th1*: *pochhammer* (− *?b*) *n* = (− *1*) ˆ*n* ∗ *pochhammer c n*
unfolding *pochhammer-minus*
by *simp*
have *nz*: *pochhammer c n* ≠ *0* **using** *c*
by (*simp add*: *pochhammer-eq-0-iff*)
from *Vandermonde-pochhammer-lemma*[**where** *a* = *?a* **and** *b*=*?b* **and** *n*=*n*, *OF h*, *unfolded th0 th1*]
show *?thesis*
using *nz* **by** (*simp add*: *field-simps setsum-right-distrib*)
qed

43.21.4 Formal trigonometric functions

definition *fps-sin* (*c*::*'a*::*field-char-0*) =
Abs-fps (λ*n*. *if even n then 0 else* (− *1*) ˆ((*n* − *1*) *div 2*) ∗ *c*ˆ*n* /(*of-nat* (*fact n*)))

definition *fps-cos* (*c*::′*a*::*field-char-0*) =
  Abs-fps (*λn. if even n then* (− *1*) ˆ (*n div 2*) ∗ *c*ˆ*n* / (*of-nat* (*fact n*)) *else 0*)

lemma *fps-sin-deriv*:
  fps-deriv (*fps-sin c*) = *fps-const c* ∗ *fps-cos c*
  (**is** *?lhs* = *?rhs*)
proof (*rule fps-ext*)
  fix *n* :: *nat*
  show *?lhs* $ *n* = *?rhs* $ *n*
  proof (*cases even n*)
    case *True*
    have *?lhs*$*n* = *of-nat* (*n*+*1*) ∗ (*fps-sin c* $ (*n*+*1*)) **by** *simp*
    also have . . . = *of-nat* (*n*+*1*) ∗ ((− *1*)ˆ(*n div 2*) ∗ *c*ˆ*Suc n* / *of-nat* (*fact* (*Suc n*)))
      using *True* **by** (*simp add*: *fps-sin-def*)
    also have . . . = (− *1*)ˆ(*n div 2*) ∗ *c*ˆ*Suc n* ∗ (*of-nat* (*n*+*1*) / (*of-nat* (*Suc n*) ∗ *of-nat* (*fact n*)))
      unfolding *fact-Suc of-nat-mult*
      by (*simp add*: *field-simps del*: *of-nat-add of-nat-Suc*)
    also have . . . = (− *1*)ˆ(*n div 2*) ∗*c*ˆ*Suc n* / *of-nat* (*fact n*)
      by (*simp add*: *field-simps del*: *of-nat-add of-nat-Suc*)
    finally show *?thesis*
      using *True* **by** (*simp add*: *fps-cos-def field-simps*)
  next
    case *False*
    then show *?thesis*
      by (*simp-all add*: *fps-deriv-def fps-sin-def fps-cos-def*)
  qed
qed

lemma *fps-cos-deriv*: *fps-deriv* (*fps-cos c*) = *fps-const* (− *c*)∗ (*fps-sin c*)
  (**is** *?lhs* = *?rhs*)
proof (*rule fps-ext*)
  have *th0*: − ((− *1*::′*a*) ˆ *n*) = (− *1*)ˆ*Suc n* **for** *n*
    by *simp*
  show *?lhs* $ *n* = *?rhs* $ *n* **for** *n*
  proof (*cases even n*)
    case *False*
    then have *n0*: *n* ≠ *0* **by** *presburger*
    from *False* **have** *th1*: *Suc* ((*n* − *1*) *div 2*) = *Suc n div 2*
      by (*cases n*) *simp-all*
    have *?lhs*$*n* = *of-nat* (*n*+*1*) ∗ (*fps-cos c* $ (*n*+*1*)) **by** *simp*
    also have . . . = *of-nat* (*n*+*1*) ∗ ((− *1*)ˆ((*n* + *1*) *div 2*) ∗ *c*ˆ*Suc n* / *of-nat* (*fact* (*Suc n*)))
      using *False* **by** (*simp add*: *fps-cos-def*)
    also have . . . = (− *1*)ˆ((*n* + *1*) *div 2*)∗*c*ˆ*Suc n* ∗ (*of-nat* (*n*+*1*) / (*of-nat* (*Suc n*) ∗ *of-nat* (*fact n*)))
      unfolding *fact-Suc of-nat-mult*
      by (*simp add*: *field-simps del*: *of-nat-add of-nat-Suc*)

also have ... = (− *1*) ^((*n* + *1*) *div* *2*) ∗ *c* ^*Suc n* / *of-nat* (*fact n*)
  by (*simp add*: *field-simps del*: *of-nat-add of-nat-Suc*)
also have ... = (− ((− *1*) ^((*n* − *1*) *div* *2*))) ∗ *c* ^*Suc n* / *of-nat* (*fact n*)
  unfolding *th0* **unfolding** *th1* **by** *simp*
finally show *?thesis*
  using *False* **by** (*simp add*: *fps-sin-def field-simps*)
next
  case *True*
  then show *?thesis*
    by (*simp-all add*: *fps-deriv-def fps-sin-def fps-cos-def*)
qed
qed

lemma *fps-sin-cos-sum-of-squares*: (*fps-cos c*)^2 + (*fps-sin c*)^2 = *1*
(**is** *?lhs* = -)
proof −
have *fps-deriv ?lhs* = *0*
  apply (*simp add*: *fps-deriv-power fps-sin-deriv fps-cos-deriv*)
  apply (*simp add*: *field-simps fps-const-neg*[*symmetric*] *del*: *fps-const-neg*)
  done
then have *?lhs* = *fps-const* (*?lhs* $ *0*)
  unfolding *fps-deriv-eq-0-iff* **.**
also have ... = *1*
  by (*auto simp add*: *fps-eq-iff numeral-2-eq-2 fps-mult-nth fps-cos-def fps-sin-def*)
finally show *?thesis* **.**
qed

lemma *fps-sin-nth-0* [*simp*]: *fps-sin c* $ *0* = *0*
unfolding *fps-sin-def* **by** *simp*

lemma *fps-sin-nth-1* [*simp*]: *fps-sin c* $ *1* = *c*
unfolding *fps-sin-def* **by** *simp*

lemma *fps-sin-nth-add-2*:
fps-sin c $ (*n* + *2*) = − (*c* ∗ *c* ∗ *fps-sin c* $ *n* / (*of-nat* (*n* + *1*) ∗ *of-nat* (*n*
+ *2*)))
unfolding *fps-sin-def*
apply (*cases n*)
apply *simp*
apply (*simp add*: *nonzero-divide-eq-eq nonzero-eq-divide-eq del*: *of-nat-Suc fact-Suc*)
apply (*simp add*: *of-nat-mult del*: *of-nat-Suc mult-Suc*)
done

lemma *fps-cos-nth-0* [*simp*]: *fps-cos c* $ *0* = *1*
unfolding *fps-cos-def* **by** *simp*

lemma *fps-cos-nth-1* [*simp*]: *fps-cos c* $ *1* = *0*
unfolding *fps-cos-def* **by** *simp*

lemma *fps-cos-nth-add-2*:
fps-cos c $ (*n* + *2*) = − (*c* ∗ *c* ∗ *fps-cos c* $ *n* / (*of-nat* (*n* + *1*) ∗ *of-nat* (*n* + *2*)))
unfolding *fps-cos-def*
apply (*simp add*: *nonzero-divide-eq-eq nonzero-eq-divide-eq del*: *of-nat-Suc fact-Suc*)
apply (*simp add*: *of-nat-mult del*: *of-nat-Suc mult-Suc*)
done

lemma *nat-induct2*: *P 0* ⟹ *P 1* ⟹ (⋀*n*. *P n* ⟹ *P* (*n* + *2*)) ⟹ *P* (*n*::*nat*)
unfolding *One-nat-def numeral-2-eq-2*
apply (*induct n rule*: *nat-less-induct*)
apply (*case-tac n*)
apply *simp*
apply (*rename-tac m*)
apply (*case-tac m*)
apply *simp*
apply (*rename-tac k*)
apply (*case-tac k*)
apply *simp-all*
done

lemma *nat-add-1-add-1*: (*n*::*nat*) + *1* + *1* = *n* + *2*
by *simp*

lemma *eq-fps-sin*:
assumes *0*: *a* $ *0* = *0*
and *1*: *a* $ *1* = *c*
and *2*: *fps-deriv* (*fps-deriv a*) = − (*fps-const c* ∗ *fps-const c* ∗ *a*)
shows *a* = *fps-sin c*
apply (*rule fps-ext*)
apply (*induct-tac n rule*: *nat-induct2*)
apply (*simp add*: *0*)
apply (*simp add*: *1 del*: *One-nat-def*)
apply (*rename-tac m*, *cut-tac f*=λ*a*. *a* $ *m* **in** *arg-cong* [*OF 2*])
apply (*simp add*: *nat-add-1-add-1 fps-sin-nth-add-2*
del: *One-nat-def of-nat-Suc of-nat-add add-2-eq-Suc′*)
apply (*subst minus-divide-left*)
apply (*subst nonzero-eq-divide-eq*)
apply (*simp del*: *of-nat-add of-nat-Suc*)
apply (*simp only*: *ac-simps*)
done

lemma *eq-fps-cos*:
assumes *0*: *a* $ *0* = *1*
and *1*: *a* $ *1* = *0*
and *2*: *fps-deriv* (*fps-deriv a*) = − (*fps-const c* ∗ *fps-const c* ∗ *a*)
shows *a* = *fps-cos c*
apply (*rule fps-ext*)
apply (*induct-tac n rule*: *nat-induct2*)

apply (*simp add*: *0*)
apply (*simp add*: *1 del*: *One-nat-def*)
apply (*rename-tac m*, *cut-tac f*=λ*a*. *a* $ *m* **in** *arg-cong* [*OF 2*])
apply (*simp add*: *nat-add-1-add-1 fps-cos-nth-add-2*
del: *One-nat-def of-nat-Suc of-nat-add add-2-eq-Suc'*)
apply (*subst minus-divide-left*)
apply (*subst nonzero-eq-divide-eq*)
apply (*simp del*: *of-nat-add of-nat-Suc*)
apply (*simp only*: *ac-simps*)
done

lemma *mult-nth-0* [*simp*]: (*a* ∗ *b*) $ *0* = *a* $ *0* ∗ *b* $ *0*
by (*simp add*: *fps-mult-nth*)

lemma *mult-nth-1* [*simp*]: (*a* ∗ *b*) $ *1* = *a* $ *0* ∗ *b* $ *1* + *a* $ *1* ∗ *b* $ *0*
by (*simp add*: *fps-mult-nth*)

lemma *fps-sin-add*: *fps-sin* (*a* + *b*) = *fps-sin a* ∗ *fps-cos b* + *fps-cos a* ∗ *fps-sin b*
apply (*rule eq-fps-sin* [*symmetric*], *simp*, *simp del*: *One-nat-def*)
apply (*simp del*: *fps-const-neg fps-const-add fps-const-mult*
add: *fps-const-add* [*symmetric*] *fps-const-neg* [*symmetric*]
fps-sin-deriv fps-cos-deriv algebra-simps)
done

lemma *fps-cos-add*: *fps-cos* (*a* + *b*) = *fps-cos a* ∗ *fps-cos b* − *fps-sin a* ∗ *fps-sin b*
apply (*rule eq-fps-cos* [*symmetric*], *simp*, *simp del*: *One-nat-def*)
apply (*simp del*: *fps-const-neg fps-const-add fps-const-mult*
add: *fps-const-add* [*symmetric*] *fps-const-neg* [*symmetric*]
fps-sin-deriv fps-cos-deriv algebra-simps)
done

lemma *fps-sin-even*: *fps-sin* (− *c*) = − *fps-sin c*
by (*auto simp add*: *fps-eq-iff fps-sin-def*)

lemma *fps-cos-odd*: *fps-cos* (− *c*) = *fps-cos c*
by (*auto simp add*: *fps-eq-iff fps-cos-def*)

definition *fps-tan c* = *fps-sin c* / *fps-cos c*

lemma *fps-tan-deriv*: *fps-deriv* (*fps-tan c*) = *fps-const c* / (*fps-cos c*)^2
proof −
have *th0*: *fps-cos c* $ *0* ≠ *0* **by** (*simp add*: *fps-cos-def*)
from *this* **have** *fps-cos c* ≠ *0* **by** (*intro notI*) *simp*
hence *fps-deriv* (*fps-tan c*) =
fps-const c ∗ (*fps-cos c*ˆ*2* + *fps-sin c*ˆ*2*) / (*fps-cos c*ˆ*2*)
by (*simp add*: *fps-tan-def fps-divide-deriv power2-eq-square algebra-simps*
fps-sin-deriv fps-cos-deriv fps-const-neg[*symmetric*] *div-mult-swap*
del: *fps-const-neg*)
also note *fps-sin-cos-sum-of-squares*

finally show *?thesis* **by** *simp*
qed

Connection to E c over the complex numbers — Euler and de Moivre.

lemma *Eii-sin-cos*: *E* (*ii* ∗ *c*) = *fps-cos c* + *fps-const ii* ∗ *fps-sin c*
(**is** *?l* = *?r*)
proof −
have *?l* $ *n* = *?r* $ *n* **for** *n*
proof (*cases even n*)
case *True*
then obtain *m* **where** *m*: *n* = *2* ∗ *m* **..**
show *?thesis*
by (*simp add*: *m fps-sin-def fps-cos-def power-mult-distrib power-mult power-minus* [*of c* ^ *2*])
next
case *False*
then obtain *m* **where** *m*: *n* = *2* ∗ *m* + *1* **..**
show *?thesis*
by (*simp add*: *m fps-sin-def fps-cos-def power-mult-distrib*
power-mult power-minus [*of c* ^ *2*])
qed
then show *?thesis*
by (*simp add*: *fps-eq-iff*)
qed

lemma *E-minus-ii-sin-cos*: *E* (− (*ii* ∗ *c*)) = *fps-cos c* − *fps-const ii* ∗ *fps-sin c*
unfolding *minus-mult-right Eii-sin-cos* **by** (*simp add*: *fps-sin-even fps-cos-odd*)

lemma *fps-const-minus*: *fps-const* (*c*::*'a*::*group-add*) − *fps-const d* = *fps-const* (*c* − *d*)
by (*fact fps-const-sub*)

lemma *fps-numeral-fps-const*: *numeral i* = *fps-const* (*numeral i* :: *'a*::*comm-ring-1*)
by (*fact numeral-fps-const*)

lemma *fps-cos-Eii*: *fps-cos c* = (*E* (*ii* ∗ *c*) + *E* (− *ii* ∗ *c*)) / *fps-const 2*
proof −
have *th*: *fps-cos c* + *fps-cos c* = *fps-cos c* ∗ *fps-const 2*
by (*simp add*: *numeral-fps-const*)
show *?thesis*
unfolding *Eii-sin-cos minus-mult-commute*
by (*simp add*: *fps-sin-even fps-cos-odd numeral-fps-const fps-divide-unit fps-const-inverse th*)
qed

lemma *fps-sin-Eii*: *fps-sin c* = (*E* (*ii* ∗ *c*) − *E* (− *ii* ∗ *c*)) / *fps-const* (*2*∗*ii*)
proof −
have *th*: *fps-const* i ∗ *fps-sin c* + *fps-const* i ∗ *fps-sin c* = *fps-sin c* ∗ *fps-const* (*2* ∗ *ii*)

by (*simp add*: *fps-eq-iff numeral-fps-const*)
show *?thesis*
unfolding *Eii-sin-cos minus-mult-commute*
by (*simp add*: *fps-sin-even fps-cos-odd fps-divide-unit fps-const-inverse th*)
qed

lemma *fps-tan-Eii*:
fps-tan c = (*E* (*ii* ∗ *c*) − *E* (− *ii* ∗ *c*)) / (*fps-const ii* ∗ (*E* (*ii* ∗ *c*) + *E* (− *ii* ∗ *c*)))
unfolding *fps-tan-def fps-sin-Eii fps-cos-Eii mult-minus-left E-neg*
apply (*simp add*: *fps-divide-unit fps-inverse-mult fps-const-mult*[*symmetric*] *fps-const-inverse del*: *fps-const-mult*)
apply *simp*
done

lemma *fps-demoivre*:
(*fps-cos a* + *fps-const ii* ∗ *fps-sin a*) ˆ*n* =
fps-cos (*of-nat n* ∗ *a*) + *fps-const ii* ∗ *fps-sin* (*of-nat n* ∗ *a*)
unfolding *Eii-sin-cos*[*symmetric*] *E-power-mult*
by (*simp add*: *ac-simps*)

43.22 Hypergeometric series

definition *F as bs* (*c*::′*a*::{*field-char-0*,*field*}) =
Abs-fps (λ*n*. (*foldl* (λ*r a*. *r*∗ *pochhammer a n*) *1 as* ∗ *c*ˆ*n*) /
(*foldl* (λ*r b*. *r* ∗ *pochhammer b n*) *1 bs* ∗ *of-nat* (*fact n*)))

lemma *F-nth*[*simp*]: *F as bs c* $ *n* =
(*foldl* (λ*r a*. *r*∗ *pochhammer a n*) *1 as* ∗ *c*ˆ*n*) /
(*foldl* (λ*r b*. *r* ∗ *pochhammer b n*) *1 bs* ∗ *of-nat* (*fact n*))
by (*simp add*: *F-def*)

lemma *foldl-mult-start*:
fixes *v* :: ′*a*::*comm-ring-1*
shows *foldl* (λ*r x*. *r* ∗ *f x*) *v as* ∗ *x* = *foldl* (λ*r x*. *r* ∗ *f x*) (*v* ∗ *x*) *as*
by (*induct as arbitrary*: *x v*) (*auto simp add*: *algebra-simps*)

lemma *foldr-mult-foldl*:
fixes *v* :: ′*a*::*comm-ring-1*
shows *foldr* (λ*x r*. *r* ∗ *f x*) *as v* = *foldl* (λ*r x*. *r* ∗ *f x*) *v as*
by (*induct as arbitrary*: *v*) (*auto simp add*: *foldl-mult-start*)

lemma *F-nth-alt*:
F as bs c $ *n* = *foldr* (λ*a r*. *r* ∗ *pochhammer a n*) *as* (*c* ˆ *n*) /
foldr (λ*b r*. *r* ∗ *pochhammer b n*) *bs* (*of-nat* (*fact n*))
by (*simp add*: *foldl-mult-start foldr-mult-foldl*)

lemma *F-E*[*simp*]: *F* [] [] *c* = *E c*
by (*simp add*: *fps-eq-iff*)

lemma *F-1-0*[*simp*]: *F* [*1*] [] *c* = *1*/(*1* − *fps-const c* ∗ *X*)
proof −
  let *?a* = (*Abs-fps* (λ*n*. *1*)) *oo* (*fps-const c* ∗ *X*)
  have *th0*: (*fps-const c* ∗ *X*) $ *0* = *0* **by** *simp*
  show *?thesis* **unfolding** *gp*[*OF th0*, *symmetric*]
    by (*auto simp add*: *fps-eq-iff pochhammer-fact*[*symmetric*]
        fps-compose-nth power-mult-distrib cond-value-iff setsum.delta′ cong del: *if-weak-cong*)
qed

lemma *F-B*[*simp*]: *F* [−*a*] [] (− *1*) = *fps-binomial a*
  by (*simp add*: *fps-eq-iff gbinomial-pochhammer algebra-simps*)

lemma *F-0*[*simp*]: *F as bs c* $ *0* = *1*
  apply *simp*
  apply (*subgoal-tac* ∀ *as*. *foldl* (λ(*r*::′*a*) (*a*::′*a*). *r*) *1 as* = *1*)
  apply *auto*
  apply (*induct-tac as*)
  apply *auto*
  done

lemma *foldl-prod-prod*:
  foldl (λ(*r*::′*b*::*comm-ring-1*) (*x*::′*a*::*comm-ring-1*). *r* ∗ *f x*) *v as* ∗ *foldl* (λ*r x*. *r* ∗ *g x*) *w as* =
    foldl (λ*r x*. *r* ∗ *f x* ∗ *g x*) (*v* ∗ *w*) *as*
  by (*induct as arbitrary*: *v w*) (*auto simp add*: *algebra-simps*)

lemma *F-rec*:
  F as bs c $ *Suc n* = ((*foldl* (λ*r a*. *r*∗ (*a* + *of-nat n*)) *c as*) /
    (*foldl* (λ*r b*. *r* ∗ (*b* + *of-nat n*)) (*of-nat* (*Suc n*)) *bs*)) ∗ *F as bs c* $ *n*
  apply (*simp del*: *of-nat-Suc of-nat-add fact-Suc*)
  apply (*simp add*: *foldl-mult-start del*: *fact-Suc of-nat-Suc*)
  unfolding *foldl-prod-prod*[*unfolded foldl-mult-start*] *pochhammer-Suc*
  apply (*simp add*: *algebra-simps of-nat-mult*)
  done

lemma *XD-nth*[*simp*]: *XD a* $ *n* = (*if n* = *0 then 0 else of-nat n* ∗ *a*$*n*)
  by (*simp add*: *XD-def*)

lemma *XD-0th*[*simp*]: *XD a* $ *0* = *0*
  by *simp*
lemma *XD-Suc*[*simp*]: *XD a* $ *Suc n* = *of-nat* (*Suc n*) ∗ *a* $ *Suc n*
  by *simp*

definition *XDp c a* = *XD a* + *fps-const c* ∗ *a*

lemma *XDp-nth*[*simp*]: *XDp c a* $ *n* = (*c* + *of-nat n*) ∗ *a*$*n*

by (*simp add*: *XDp-def algebra-simps*)

lemma *XDp-commute*: *XDp b* ∘ *XDp* (*c*::′*a*::*comm-ring-1*) = *XDp c* ∘ *XDp b*
by (*auto simp add*: *XDp-def fun-eq-iff fps-eq-iff algebra-simps*)

lemma *XDp0* [*simp*]: *XDp 0* = *XD*
by (*simp add*: *fun-eq-iff fps-eq-iff*)

lemma *XDp-fps-integral* [*simp*]: *XDp 0* (*fps-integral a c*) = *X* ∗ *a*
by (*simp add*: *fps-eq-iff fps-integral-def*)

lemma *F-minus-nat*:
F [− *of-nat n*] [− *of-nat* (*n* + *m*)] (*c*::′*a*::{*field-char-0*,*field*}) $ *k* =
(*if k* ≤ *n then*
pochhammer (− *of-nat n*) *k* ∗ *c* ˆ *k* / (*pochhammer* (− *of-nat* (*n* + *m*)) *k* ∗ *of-nat* (*fact k*))
else 0)
F [− *of-nat m*] [− *of-nat* (*m* + *n*)] (*c*::′*a*::{*field-char-0*,*field*}) $ *k* =
(*if k* ≤ *m then*
pochhammer (− *of-nat m*) *k* ∗ *c* ˆ *k* / (*pochhammer* (− *of-nat* (*m* + *n*)) *k* ∗ *of-nat* (*fact k*))
else 0)
by (*auto simp add*: *pochhammer-eq-0-iff*)

lemma *setsum-eq-if*: *setsum f* {(*n*::*nat*) .. *m*} = (*if m* < *n then 0 else f n* + *setsum f* {*n*+*1* .. *m*})
apply *simp*
apply (*subst setsum.insert*[*symmetric*])
apply (*auto simp add*: *not-less setsum-head-Suc*)
done

lemma *pochhammer-rec-if*: *pochhammer a n* = (*if n* = *0 then 1 else a* ∗ *pochhammer* (*a* + *1*) (*n* − *1*))
by (*cases n*) (*simp-all add*: *pochhammer-rec*)

lemma *XDp-foldr-nth* [*simp*]: *foldr* (λ*c r*. *XDp c* ∘ *r*) *cs* (λ*c*. *XDp c a*) *c0* $ *n* =
foldr (λ*c r*. (*c* + *of-nat n*) ∗ *r*) *cs* (*c0* + *of-nat n*) ∗ *a*$*n*
by (*induct cs arbitrary*: *c0*) (*auto simp add*: *algebra-simps*)

lemma *genric-XDp-foldr-nth*:
assumes *f*: ∀ *n c a*. *f c a* $ *n* = (*of-nat n* + *k c*) ∗ *a*$*n*
shows *foldr* (λ*c r*. *f c* ∘ *r*) *cs* (λ*c*. *g c a*) *c0* $ *n* =
foldr (λ*c r*. (*k c* + *of-nat n*) ∗ *r*) *cs* (*g c0 a* $ *n*)
by (*induct cs arbitrary*: *c0*) (*auto simp add*: *algebra-simps f*)

lemma *dist-less-imp-nth-equal*:
assumes *dist f g* < *inverse* (*2* ˆ *i*)
and*j* ≤ *i*
shows *f* $ *j* = *g* $ *j*

proof (*rule ccontr*)
  assume $f \$ j \neq g \$ j$
  hence $f \neq g$ **by** *auto*
  with *assms* **have** $i < subdegree\ (f - g)$
    by (*simp add*: *if-split-asm dist-fps-def*)
  also have ... $\leq j$
    using ‹$f \$ j \neq g \$ j$› **by** (*intro subdegree-leI*) *simp-all*
  finally show *False* **using** ‹$j \leq i$› **by** *simp*
qed

lemma *nth-equal-imp-dist-less*:
  assumes $\bigwedge j.\ j \leq i \Longrightarrow f \$ j = g \$ j$
  shows $dist\ f\ g < inverse\ (2\ \hat{}\ i)$
proof (*cases* $f = g$)
  case *True*
  then show *?thesis* **by** *simp*
next
  case *False*
  with *assms* **have** $dist\ f\ g = inverse\ (2\ \hat{}\ subdegree\ (f - g))$
    by (*simp add*: *if-split-asm dist-fps-def*)
  moreover
  from *assms* **and** *False* **have** $i < subdegree\ (f - g)$
    by (*intro subdegree-greaterI*) *simp-all*
  ultimately show *?thesis* **by** *simp*
qed

lemma *dist-less-eq-nth-equal*: $dist\ f\ g < inverse\ (2\ \hat{}\ i) \longleftrightarrow (\forall j \leq i.\ f \$ j = g \$ j)$
  using *dist-less-imp-nth-equal nth-equal-imp-dist-less* **by** *blast*

instance *fps* :: (*comm-ring-1*) *complete-space*
proof
  fix $X :: nat \Rightarrow 'a\ fps$
  assume *Cauchy X*
  obtain *M* **where** $M: \forall i.\ \forall m \geq M\ i.\ \forall j \leq i.\ X\ (M\ i) \$ j = X\ m \$ j$
  proof −
    have $\exists M.\ \forall m \geq M.\ \forall j{\leq}i.\ X\ M \$ j = X\ m \$ j$ **for** *i*
    proof −
      have $0 < inverse\ ((2{::}real)\hat{}i)$ **by** *simp*
      from *metric-CauchyD*[*OF* ‹*Cauchy X*› *this*] *dist-less-imp-nth-equal*
      show *?thesis* **by** *blast*
    qed
    then show *?thesis* **using** *that* **by** *metis*
  qed

  show *convergent X*
  proof (*rule convergentI*)
    show $X \longrightarrow Abs\text{-}fps\ (\lambda i.\ X\ (M\ i) \$ i)$
      unfolding *tendsto-iff*

```
  proof safe
    fix e::real assume e: 0 < e
   have (λn. inverse (2 ^ n) :: real) ⟶ 0 by (rule LIMSEQ-inverse-realpow-zero)
simp-all
    from this and e have eventually (λi. inverse (2 ^ i) < e) sequentially
      by (rule order-tendstoD)
    then obtain i where inverse (2 ^ i) < e
      by (auto simp: eventually-sequentially)
    have eventually (λx. M i ≤ x) sequentially
      by (auto simp: eventually-sequentially)
     then show eventually (λx. dist (X x) (Abs-fps (λi. X (M i) $ i)) < e)
sequentially
    proof eventually-elim
      fix x
      assume x: M i ≤ x
      have X (M i) $ j = X (M j) $ j if j ≤ i for j
        using M that by (metis nat-le-linear)
      with x have dist (X x) (Abs-fps (λj. X (M j) $ j)) < inverse (2 ^ i)
        using M by (force simp: dist-less-eq-nth-equal)
      also note ‹inverse (2 ^ i) < e›
      finally show dist (X x) (Abs-fps (λj. X (M j) $ j)) < e .
    qed
  qed
 qed
qed

end
```

44 A formalization of the fraction field of any integral domain; generalization of theory Rat from int to any integral domain

theory *Fraction-Field*
imports *Main*
begin

44.1 General fractions construction

44.1.1 Construction of the type of fractions

context *idom* **begin**

definition *fractrel* :: $'a \times 'a \Rightarrow 'a * 'a \Rightarrow bool$ **where**
 $fractrel = (\lambda x\ y.\ snd\ x \neq 0 \wedge snd\ y \neq 0 \wedge fst\ x * snd\ y = fst\ y * snd\ x)$

lemma *fractrel-iff* [*simp*]:
 $fractrel\ x\ y \longleftrightarrow snd\ x \neq 0 \wedge snd\ y \neq 0 \wedge fst\ x * snd\ y = fst\ y * snd\ x$
 by (*simp add*: *fractrel-def*)

lemma *symp-fractrel*: *symp fractrel*
  by (*simp add*: *symp-def*)

lemma *transp-fractrel*: *transp fractrel*
proof (*rule transpI*, *unfold split-paired-all*)
  fix $a\ b\ a'\ b'\ a''\ b'' :: 'a$
  assume *A*: *fractrel* $(a, b)\ (a', b')$
  assume *B*: *fractrel* $(a', b')\ (a'', b'')$
  have $b' * (a * b'') = b'' * (a * b')$ **by** (*simp add*: *ac-simps*)
  also from *A* **have** $a * b' = a' * b$ **by** *auto*
  also have $b'' * (a' * b) = b * (a' * b'')$ **by** (*simp add*: *ac-simps*)
  also from *B* **have** $a' * b'' = a'' * b'$ **by** *auto*
  also have $b * (a'' * b') = b' * (a'' * b)$ **by** (*simp add*: *ac-simps*)
  finally have $b' * (a * b'') = b' * (a'' * b)$ **.**
  moreover from *B* **have** $b' \neq 0$ **by** *auto*
  ultimately have $a * b'' = a'' * b$ **by** *simp*
  with *A B* **show** *fractrel* $(a, b)\ (a'', b'')$ **by** *auto*
qed

lemma *part-equivp-fractrel*: *part-equivp fractrel*
using - *symp-fractrel transp-fractrel*
by(*rule part-equivpI*)(*rule exI*[**where** $x=(0, 1)$]; *simp*)

end

quotient-type (**overloaded**) $'a$ *fract* = $'a ::$ *idom* $\times$ $'a$ / *partial*: *fractrel*
by(*rule part-equivp-fractrel*)

44.1.2 Representation and basic operations

lift-definition *Fract* :: $'a ::$ *idom* $\Rightarrow 'a \Rightarrow 'a$ *fract*
  is $\lambda a\ b.$ *if* $b = 0$ *then* $(0, 1)$ *else* (a, b)
  by *simp*

lemma *Fract-cases* [*cases type*: *fract*]:
  obtains (*Fract*) *a b* **where** $q =$ *Fract a b* $b \neq 0$
by *transfer simp*

lemma *Fract-induct* [*case-names Fract*, *induct type*: *fract*]:
  $(\bigwedge a\ b.\ b \neq 0 \Longrightarrow P\ (Fract\ a\ b)) \Longrightarrow P\ q$
  by (*cases q*) *simp*

lemma *eq-fract*:
  shows $\bigwedge a\ b\ c\ d.\ b \neq 0 \Longrightarrow d \neq 0 \Longrightarrow Fract\ a\ b = Fract\ c\ d \longleftrightarrow a * d = c * b$
    and $\bigwedge a.\ Fract\ a\ 0 = Fract\ 0\ 1$
    and $\bigwedge a\ c.\ Fract\ 0\ a = Fract\ 0\ c$
by(*transfer*; *simp*)+

instantiation *fract* :: (*idom*) {*comm-ring-1*,*power*}
begin

lift-definition *zero-fract* :: *'a fract* **is** (*0*, *1*) **by** *simp*

lemma *Zero-fract-def*: *0* = *Fract 0 1*
by *transfer simp*

lift-definition *one-fract* :: *'a fract* **is** (*1*, *1*) **by** *simp*

lemma *One-fract-def*: *1* = *Fract 1 1*
by *transfer simp*

lift-definition *plus-fract* :: *'a fract* ⇒ *'a fract* ⇒ *'a fract*
  is λ*q r*. (*fst q* ∗ *snd r* + *fst r* ∗ *snd q*, *snd q* ∗ *snd r*)
by(*auto simp add*: *algebra-simps*)

lemma *add-fract* [*simp*]:
  ⟦ *b* ≠ *0*; *d* ≠ *0* ⟧ ⟹ *Fract a b* + *Fract c d* = *Fract* (*a* ∗ *d* + *c* ∗ *b*) (*b* ∗ *d*)
by *transfer simp*

lift-definition *uminus-fract* :: *'a fract* ⇒ *'a fract*
  is λ*x*. (− *fst x*, *snd x*)
by *simp*

lemma *minus-fract* [*simp*]:
  fixes *a b* :: *'a*::*idom*
  shows − *Fract a b* = *Fract* (− *a*) *b*
by *transfer simp*

lemma *minus-fract-cancel* [*simp*]: *Fract* (− *a*) (− *b*) = *Fract a b*
  by (*cases b* = *0*) (*simp-all add*: *eq-fract*)

definition *diff-fract-def*: *q* − *r* = *q* + − (*r*::*'a fract*)

lemma *diff-fract* [*simp*]:
  ⟦ *b* ≠ *0*; *d* ≠ *0* ⟧ ⟹ *Fract a b* − *Fract c d* = *Fract* (*a* ∗ *d* − *c* ∗ *b*) (*b* ∗ *d*)
  by (*simp add*: *diff-fract-def*)

lift-definition *times-fract* :: *'a fract* ⇒ *'a fract* ⇒ *'a fract*
  is λ*q r*. (*fst q* ∗ *fst r*, *snd q* ∗ *snd r*)
by(*simp add*: *algebra-simps*)

lemma *mult-fract* [*simp*]: *Fract* (*a*::*'a*::*idom*) *b* ∗ *Fract c d* = *Fract* (*a* ∗ *c*) (*b* ∗ *d*)
by *transfer simp*

lemma *mult-fract-cancel*:
  c ≠ *0* ⟹ *Fract* (*c* ∗ *a*) (*c* ∗ *b*) = *Fract a b*

by *transfer simp*

instance
proof
  fix *q r s* :: *'a fract*
  show (*q* ∗ *r*) ∗ *s* = *q* ∗ (*r* ∗ *s*)
    by (*cases q*, *cases r*, *cases s*) (*simp add*: *eq-fract algebra-simps*)
  show *q* ∗ *r* = *r* ∗ *q*
    by (*cases q*, *cases r*) (*simp add*: *eq-fract algebra-simps*)
  show *1* ∗ *q* = *q*
    by (*cases q*) (*simp add*: *One-fract-def eq-fract*)
  show (*q* + *r*) + *s* = *q* + (*r* + *s*)
    by (*cases q*, *cases r*, *cases s*) (*simp add*: *eq-fract algebra-simps*)
  show *q* + *r* = *r* + *q*
    by (*cases q*, *cases r*) (*simp add*: *eq-fract algebra-simps*)
  show *0* + *q* = *q*
    by (*cases q*) (*simp add*: *Zero-fract-def eq-fract*)
  show − *q* + *q* = *0*
    by (*cases q*) (*simp add*: *Zero-fract-def eq-fract*)
  show *q* − *r* = *q* + − *r*
    by (*cases q*, *cases r*) (*simp add*: *eq-fract*)
  show (*q* + *r*) ∗ *s* = *q* ∗ *s* + *r* ∗ *s*
    by (*cases q*, *cases r*, *cases s*) (*simp add*: *eq-fract algebra-simps*)
  show (*0*::*'a fract*) ≠ *1*
    by (*simp add*: *Zero-fract-def One-fract-def eq-fract*)
qed

end

lemma *of-nat-fract*: *of-nat k* = *Fract* (*of-nat k*) *1*
  by (*induct k*) (*simp-all add*: *Zero-fract-def One-fract-def*)

lemma *Fract-of-nat-eq*: *Fract* (*of-nat k*) *1* = *of-nat k*
  by (*rule of-nat-fract* [*symmetric*])

lemma *fract-collapse*:
  Fract 0 k = *0*
  Fract 1 1 = *1*
  Fract k 0 = *0*
by(*transfer*; *simp*)+

lemma *fract-expand*:
  0 = *Fract 0 1*
  1 = *Fract 1 1*
  by (*simp-all add*: *fract-collapse*)

lemma *Fract-cases-nonzero*:
  obtains (*Fract*) *a b* **where** *q* = *Fract a b* **and** *b* ≠ *0* **and** *a* ≠ *0*
    | (*0*) *q* = *0*

proof (*cases q* = *0*)
  case *True*
  then show *thesis* **using** *0* **by** *auto*
next
  case *False*
  then obtain *a b* **where** *q* = *Fract a b* **and** *b* ≠ *0* **by** (*cases q*) *auto*
  with *False* **have** *0* ≠ *Fract a b* **by** *simp*
  with ‹*b* ≠ *0*› **have** *a* ≠ *0* **by** (*simp add*: *Zero-fract-def eq-fract*)
  with *Fract* ‹*q* = *Fract a b*› ‹*b* ≠ *0*› **show** *thesis* **by** *auto*
qed

44.1.3 The field of rational numbers

context *idom*
begin

subclass *ring-no-zero-divisors* **..**

end

instantiation *fract* :: (*idom*) *field*
begin

lift-definition *inverse-fract* :: ′*a fract* ⇒ ′*a fract*
  is λ*x*. *if fst x* = *0 then* (*0*, *1*) *else* (*snd x*, *fst x*)
by(*auto simp add*: *algebra-simps*)

lemma *inverse-fract* [*simp*]: *inverse* (*Fract a b*) = *Fract* (*b*::′*a*::*idom*) *a*
by *transfer simp*

definition *divide-fract-def*: *q div r* = *q* ∗ *inverse* (*r*:: ′*a fract*)

lemma *divide-fract* [*simp*]: *Fract a b div Fract c d* = *Fract* (*a* ∗ *d*) (*b* ∗ *c*)
  by (*simp add*: *divide-fract-def*)

instance
proof
  fix *q* :: ′*a fract*
  assume *q* ≠ *0*
  then show *inverse q* ∗ *q* = *1*
    by (*cases q rule*: *Fract-cases-nonzero*)
      (*simp-all add*: *fract-expand eq-fract mult.commute*)
next
  fix *q r* :: ′*a fract*
  show *q div r* = *q* ∗ *inverse r* **by** (*simp add*: *divide-fract-def*)
next
  show *inverse 0* = (*0*:: ′*a fract*)
    by (*simp add*: *fract-expand*) (*simp add*: *fract-collapse*)
qed

end

44.1.4 The ordered field of fractions over an ordered idom

instantiation *fract* :: (*linordered-idom*) *linorder*
begin

lemma *less-eq-fract-respect*:
  fixes $a\ b\ a'\ b'\ c\ d\ c'\ d' :: 'a$
  assumes *neq*: $b \neq 0 \quad b' \neq 0 \quad d \neq 0 \quad d' \neq 0$
  assumes *eq1*: $a * b' = a' * b$
  assumes *eq2*: $c * d' = c' * d$
  shows $((a * d) * (b * d) \leq (c * b) * (b * d)) \longleftrightarrow ((a' * d') * (b' * d') \leq (c' * b') * (b' * d'))$
proof –
  let $?le = \lambda a\ b\ c\ d.\ ((a * d) * (b * d) \leq (c * b) * (b * d))$
  {
    fix $a\ b\ c\ d\ x :: 'a$
    assume x: $x \neq 0$
    have $?le\ a\ b\ c\ d = ?le\ (a * x)\ (b * x)\ c\ d$
    proof –
      from x **have** $0 < x * x$
        by (*auto simp add*: *zero-less-mult-iff*)
      then have $?le\ a\ b\ c\ d =$
          $((a * d) * (b * d) * (x * x) \leq (c * b) * (b * d) * (x * x))$
        by (*simp add*: *mult-le-cancel-right*)
      also have ... $= ?le\ (a * x)\ (b * x)\ c\ d$
        by (*simp add*: *ac-simps*)
      finally show *?thesis* **.**
    qed
  } **note** *le-factor* = *this*

  let $?D = b * d$ **and** $?D' = b' * d'$
  from *neq* **have** D: $?D \neq 0$ **by** *simp*
  from *neq* **have** $?D' \neq 0$ **by** *simp*
  then have $?le\ a\ b\ c\ d = ?le\ (a * ?D')\ (b * ?D')\ c\ d$
    by (*rule le-factor*)
  also have ... $= ((a * b') * ?D * ?D' * d * d' \leq (c * d') * ?D * ?D' * b * b')$
    by (*simp add*: *ac-simps*)
  also have ... $= ((a' * b) * ?D * ?D' * d * d' \leq (c' * d) * ?D * ?D' * b * b')$
    by (*simp only*: *eq1 eq2*)
  also have ... $= ?le\ (a' * ?D)\ (b' * ?D)\ c'\ d'$
    by (*simp add*: *ac-simps*)
  also from D **have** ... $= ?le\ a'\ b'\ c'\ d'$
    by (*rule le-factor* [*symmetric*])
  finally show $?le\ a\ b\ c\ d = ?le\ a'\ b'\ c'\ d'$ **.**
qed

lift-definition *less-eq-fract* :: *'a fract* ⇒ *'a fract* ⇒ *bool*
  is $\lambda q\ r.\ (fst\ q * snd\ r) * (snd\ q * snd\ r) \leq (fst\ r * snd\ q) * (snd\ q * snd\ r)$
by (*clarsimp simp add*: *less-eq-fract-respect*)

definition *less-fract-def*: $z < (w::'a\ fract) \longleftrightarrow z \leq w \land \neg\ w \leq z$

lemma *le-fract* [*simp*]:
  $\llbracket\ b \neq 0;\ d \neq 0\ \rrbracket \Longrightarrow Fract\ a\ b \leq Fract\ c\ d \longleftrightarrow (a * d) * (b * d) \leq (c * b) * (b * d)$
  by *transfer simp*

lemma *less-fract* [*simp*]:
  $\llbracket\ b \neq 0;\ d \neq 0\ \rrbracket \Longrightarrow Fract\ a\ b < Fract\ c\ d \longleftrightarrow (a * d) * (b * d) < (c * b) * (b * d)$
  by (*simp add*: *less-fract-def less-le-not-le ac-simps assms*)

instance
proof
  fix *q r s* :: *'a fract*
  assume $q \leq r$ **and** $r \leq s$
  then show $q \leq s$
  proof (*induct q*, *induct r*, *induct s*)
    fix *a b c d e f* :: *'a*
    assume *neq*: $b \neq 0\ d \neq 0\ f \neq 0$
    assume *1*: $Fract\ a\ b \leq Fract\ c\ d$
    assume *2*: $Fract\ c\ d \leq Fract\ e\ f$
    show $Fract\ a\ b \leq Fract\ e\ f$
    proof −
      from *neq* **obtain** *bb*: $0 < b * b$ **and** *dd*: $0 < d * d$ **and** *ff*: $0 < f * f$
        by (*auto simp add*: *zero-less-mult-iff linorder-neq-iff*)
      have $(a * d) * (b * d) * (f * f) \leq (c * b) * (b * d) * (f * f)$
      proof −
        from *neq 1* **have** $(a * d) * (b * d) \leq (c * b) * (b * d)$
          by *simp*
        with *ff* **show** *?thesis* **by** (*simp add*: *mult-le-cancel-right*)
      qed
      also have ... $= (c * f) * (d * f) * (b * b)$
        by (*simp only*: *ac-simps*)
      also have ... $\leq (e * d) * (d * f) * (b * b)$
      proof −
        from *neq 2* **have** $(c * f) * (d * f) \leq (e * d) * (d * f)$
          by *simp*
        with *bb* **show** *?thesis* **by** (*simp add*: *mult-le-cancel-right*)
      qed
      finally have $(a * f) * (b * f) * (d * d) \leq e * b * (b * f) * (d * d)$
        by (*simp only*: *ac-simps*)
      with *dd* **have** $(a * f) * (b * f) \leq (e * b) * (b * f)$
        by (*simp add*: *mult-le-cancel-right*)
      with *neq* **show** *?thesis* **by** *simp*

```
    qed
  qed
next
  fix q r :: 'a fract
  assume q ≤ r and r ≤ q
  then show q = r
  proof (induct q, induct r)
    fix a b c d :: 'a
    assume neq: b ≠ 0 d ≠ 0
    assume 1: Fract a b ≤ Fract c d
    assume 2: Fract c d ≤ Fract a b
    show Fract a b = Fract c d
    proof −
      from neq 1 have (a ∗ d) ∗ (b ∗ d) ≤ (c ∗ b) ∗ (b ∗ d)
        by simp
      also have ... ≤ (a ∗ d) ∗ (b ∗ d)
      proof −
        from neq 2 have (c ∗ b) ∗ (d ∗ b) ≤ (a ∗ d) ∗ (d ∗ b)
          by simp
        then show ?thesis by (simp only: ac-simps)
      qed
      finally have (a ∗ d) ∗ (b ∗ d) = (c ∗ b) ∗ (b ∗ d) .
      moreover from neq have b ∗ d ≠ 0 by simp
      ultimately have a ∗ d = c ∗ b by simp
      with neq show ?thesis by (simp add: eq-fract)
    qed
  qed
next
  fix q r :: 'a fract
  show q ≤ q
    by (induct q) simp
  show (q < r) = (q ≤ r ∧ ¬ r ≤ q)
    by (simp only: less-fract-def)
  show q ≤ r ∨ r ≤ q
    by (induct q, induct r)
       (simp add: mult.commute, rule linorder-linear)
qed

end

instantiation fract :: (linordered-idom) {distrib-lattice,abs-if,sgn-if}
begin

definition abs-fract-def2: |q| = (if q < 0 then −q else (q::'a fract))

definition sgn-fract-def:
  sgn (q::'a fract) = (if q = 0 then 0 else if 0 < q then 1 else − 1)

theorem abs-fract [simp]: |Fract a b| = Fract |a| |b|
```

unfolding *abs-fract-def2 not-le*[*symmetric*]
by *transfer*(*auto simp add*: *zero-less-mult-iff le-less*)

definition *inf-fract-def*:
  (*inf* :: *'a fract* ⇒ *'a fract* ⇒ *'a fract*) = *min*

definition *sup-fract-def*:
  (*sup* :: *'a fract* ⇒ *'a fract* ⇒ *'a fract*) = *max*

instance
by *intro-classes* (*simp-all add*: *abs-fract-def2 sgn-fract-def inf-fract-def sup-fract-def max-min-distrib2*)

end

instance *fract* :: (*linordered-idom*) *linordered-field*
proof
  fix *q r s* :: *'a fract*
  assume $q \le r$
  then show $s + q \le s + r$
  proof (*induct q*, *induct r*, *induct s*)
    fix *a b c d e f* :: *'a*
    assume *neq*: $b \neq 0$ $d \neq 0$ $f \neq 0$
    assume *le*: *Fract a b* ≤ *Fract c d*
    show *Fract e f* + *Fract a b* ≤ *Fract e f* + *Fract c d*
    proof −
      let *?F* = $f * f$ **from** *neq* **have** *F*: *0* < *?F*
        by (*auto simp add*: *zero-less-mult-iff*)
      from *neq le* **have** $(a * d) * (b * d) \le (c * b) * (b * d)$
        by *simp*
      with *F* **have** $(a * d) * (b * d) *$ *?F* ∗ *?F* ≤ $(c * b) * (b * d) *$ *?F* ∗ *?F*
        by (*simp add*: *mult-le-cancel-right*)
      with *neq* **show** *?thesis* **by** (*simp add*: *field-simps*)
    qed
  qed
next
  fix *q r s* :: *'a fract*
  assume $q < r$ **and** $0 < s$
  then show $s * q < s * r$
  proof (*induct q*, *induct r*, *induct s*)
    fix *a b c d e f* :: *'a*
    assume *neq*: $b \neq 0$ $d \neq 0$ $f \neq 0$
    assume *le*: *Fract a b* < *Fract c d*
    assume *gt*: *0* < *Fract e f*
    show *Fract e f* ∗ *Fract a b* < *Fract e f* ∗ *Fract c d*
    proof −
      let *?E* = $e * f$ **and** *?F* = $f * f$
      from *neq gt* **have** *0* < *?E*
        by (*auto simp add*: *Zero-fract-def order-less-le eq-fract*)

moreover from *neq* **have** *0 < ?F*
by (*auto simp add*: *zero-less-mult-iff*)
moreover from *neq le* **have** $(a * d) * (b * d) < (c * b) * (b * d)$
by *simp*
ultimately have $(a * d) * (b * d) * ?E * ?F < (c * b) * (b * d) * ?E * ?F$
by (*simp add*: *mult-less-cancel-right*)
with *neq* **show** *?thesis*
by (*simp add*: *ac-simps*)
qed
qed
qed

lemma *fract-induct-pos* [*case-names Fract*]:
fixes *P* :: *'a::linordered-idom fract* ⇒ *bool*
assumes *step*: $\bigwedge a\ b.\ 0 < b \Longrightarrow P\ (Fract\ a\ b)$
shows *P q*
proof (*cases q*)
case (*Fract a b*)
{
fix *a b* :: *'a*
assume *b*: *b < 0*
have *P* (*Fract a b*)
proof −
from *b* **have** *0 < − b* **by** *simp*
then have *P* (*Fract* (− *a*) (− *b*))
by (*rule step*)
then show *P* (*Fract a b*)
by (*simp add*: *order-less-imp-not-eq* [*OF b*])
qed
}
with *Fract* **show** *P q*
by (*auto simp add*: *linorder-neq-iff step*)
qed

lemma *zero-less-Fract-iff*: $0 < b \Longrightarrow 0 < Fract\ a\ b \longleftrightarrow 0 < a$
by (*auto simp add*: *Zero-fract-def zero-less-mult-iff*)

lemma *Fract-less-zero-iff*: $0 < b \Longrightarrow Fract\ a\ b < 0 \longleftrightarrow a < 0$
by (*auto simp add*: *Zero-fract-def mult-less-0-iff*)

lemma *zero-le-Fract-iff*: $0 < b \Longrightarrow 0 \le Fract\ a\ b \longleftrightarrow 0 \le a$
by (*auto simp add*: *Zero-fract-def zero-le-mult-iff*)

lemma *Fract-le-zero-iff*: $0 < b \Longrightarrow Fract\ a\ b \le 0 \longleftrightarrow a \le 0$
by (*auto simp add*: *Zero-fract-def mult-le-0-iff*)

lemma *one-less-Fract-iff*: $0 < b \Longrightarrow 1 < Fract\ a\ b \longleftrightarrow b < a$
by (*auto simp add*: *One-fract-def mult-less-cancel-right-disj*)

lemma *Fract-less-one-iff*: $0 < b \Longrightarrow Fract\ a\ b < 1 \longleftrightarrow a < b$
by (*auto simp add*: *One-fract-def mult-less-cancel-right-disj*)

lemma *one-le-Fract-iff*: $0 < b \Longrightarrow 1 \leq Fract\ a\ b \longleftrightarrow b \leq a$
by (*auto simp add*: *One-fract-def mult-le-cancel-right*)

lemma *Fract-le-one-iff*: $0 < b \Longrightarrow Fract\ a\ b \leq 1 \longleftrightarrow a \leq b$
by (*auto simp add*: *One-fract-def mult-le-cancel-right*)

end

45 Type of finite sets defined as a subtype of sets

theory *FSet*
imports *Conditionally-Complete-Lattices*
begin

45.1 Definition of the type

typedef $'a\ fset = \{A :: 'a\ set.\ finite\ A\}$ **morphisms** *fset Abs-fset*
by *auto*

setup-lifting *type-definition-fset*

45.2 Basic operations and type class instantiations

instantiation *fset* :: (*finite*) *finite*
begin
instance by (*standard*; *transfer*; *simp*)
end

instantiation *fset* :: (*type*) {*bounded-lattice-bot*, *distrib-lattice*, *minus*}
begin

interpretation *lifting-syntax* **.**

lift-definition *bot-fset* :: *'a fset* **is** {} **parametric** *empty-transfer* **by** *simp*

lift-definition *less-eq-fset* :: $'a\ fset \Rightarrow 'a\ fset \Rightarrow bool$ **is** *subset-eq* **parametric** *subset-transfer*
.

definition *less-fset* :: $'a\ fset \Rightarrow 'a\ fset \Rightarrow bool$ **where** $xs < ys \equiv xs \leq ys \wedge xs \neq (ys{::}'a\ fset)$

lemma *less-fset-transfer*[*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows ((*pcr-fset A*) ===> (*pcr-fset A*) ===> *op* =) *op* ⊂ *op* <
unfolding *less-fset-def*[*abs-def*] *psubset-eq*[*abs-def*] **by** *transfer-prover*

lift-definition *sup-fset* :: 'a fset ⇒ 'a fset ⇒ 'a fset **is** *union* **parametric** *union-transfer*
by *simp*

lift-definition *inf-fset* :: 'a fset ⇒ 'a fset ⇒ 'a fset **is** *inter* **parametric** *inter-transfer*
by *simp*

lift-definition *minus-fset* :: 'a fset ⇒ 'a fset ⇒ 'a fset **is** *minus* **parametric** *Diff-transfer*
by *simp*

instance
by (*standard*; *transfer*; *auto*)+

end

abbreviation *fempty* :: 'a fset ({||}) **where** {||} ≡ *bot*
abbreviation *fsubset-eq* :: 'a fset ⇒ 'a fset ⇒ bool (**infix** |⊆| *50*) **where** *xs* |⊆| *ys* ≡ *xs* ≤ *ys*
abbreviation *fsubset* :: 'a fset ⇒ 'a fset ⇒ bool (**infix** |⊂| *50*) **where** *xs* |⊂| *ys* ≡ *xs* < *ys*
abbreviation *funion* :: 'a fset ⇒ 'a fset ⇒ 'a fset (**infixl** |∪| *65*) **where** *xs* |∪| *ys* ≡ *sup xs ys*
abbreviation *finter* :: 'a fset ⇒ 'a fset ⇒ 'a fset (**infixl** |∩| *65*) **where** *xs* |∩| *ys* ≡ *inf xs ys*
abbreviation *fminus* :: 'a fset ⇒ 'a fset ⇒ 'a fset (**infixl** |−| *65*) **where** *xs* |−| *ys* ≡ *minus xs ys*

instantiation *fset* :: (*equal*) *equal*
begin
definition *HOL.equal A B* ⟷ *A* |⊆| *B* ∧ *B* |⊆| *A*
instance by *intro-classes* (*auto simp add*: *equal-fset-def*)
end

instantiation *fset* :: (*type*) *conditionally-complete-lattice*
begin

interpretation *lifting-syntax* **.**

lemma *right-total-Inf-fset-transfer*:
assumes [*transfer-rule*]: *bi-unique A* **and** [*transfer-rule*]: *right-total A*
shows (*rel-set* (*rel-set A*) ===> *rel-set A*)
(λ*S*. *if finite* (⋂*S* ∩ *Collect* (*Domainp A*)) *then* ⋂*S* ∩ *Collect* (*Domainp A*) *else* {})
(λ*S*. *if finite* (*Inf S*) *then Inf S else* {})
by *transfer-prover*

lemma *Inf-fset-transfer*:

assumes [*transfer-rule*]: *bi-unique A* **and** [*transfer-rule*]: *bi-total A*
shows (*rel-set* (*rel-set A*) ===> *rel-set A*) (λ*A*. *if finite* (*Inf A*) *then Inf A else* {})
(λ*A*. *if finite* (*Inf A*) *then Inf A else* {})
by *transfer-prover*

lift-definition *Inf-fset* :: *'a fset set* ⇒ *'a fset* **is** λ*A*. *if finite* (*Inf A*) *then Inf A else* {}
parametric *right-total-Inf-fset-transfer Inf-fset-transfer* **by** *simp*

lemma *Sup-fset-transfer*:
assumes [*transfer-rule*]: *bi-unique A*
shows (*rel-set* (*rel-set A*) ===> *rel-set A*) (λ*A*. *if finite* (*Sup A*) *then Sup A else* {})
(λ*A*. *if finite* (*Sup A*) *then Sup A else* {}) **by** *transfer-prover*

lift-definition *Sup-fset* :: *'a fset set* ⇒ *'a fset* **is** λ*A*. *if finite* (*Sup A*) *then Sup A else* {}
parametric *Sup-fset-transfer* **by** *simp*

lemma *finite-Sup*: ∃ *z*. *finite z* ∧ (∀ *a*. *a* ∈ *X* ⟶ *a* ≤ *z*) ⟹ *finite* (*Sup X*)
by (*auto intro*: *finite-subset*)

lemma *transfer-bdd-below*[*transfer-rule*]: (*rel-set* (*pcr-fset op* =) ===> *op* =) *bdd-below bdd-below*
by *auto*

instance
proof
fix *x z* :: *'a fset*
fix *X* :: *'a fset set*
{
assume *x* ∈ *X bdd-below X*
then show *Inf X* |⊆| *x* **by** *transfer auto*
next
assume *X* ≠ {} (⋀*x*. *x* ∈ *X* ⟹ *z* |⊆| *x*)
then show *z* |⊆| *Inf X* **by** *transfer* (*clarsimp*, *blast*)
next
assume *x* ∈ *X bdd-above X*
then obtain *z* **where** *x* ∈ *X* (⋀*x*. *x* ∈ *X* ⟹ *x* |⊆| *z*)
by (*auto simp*: *bdd-above-def*)
then show *x* |⊆| *Sup X*
by *transfer* (*auto intro*!: *finite-Sup*)
next
assume *X* ≠ {} (⋀*x*. *x* ∈ *X* ⟹ *x* |⊆| *z*)
then show *Sup X* |⊆| *z* **by** *transfer* (*clarsimp*, *blast*)
}
qed
end

instantiation *fset* :: (*finite*) *complete-lattice*
begin

lift-definition *top-fset* :: *'a fset* **is** *UNIV* **parametric** *right-total-UNIV-transfer UNIV-transfer*
by *simp*

instance
by (*standard*; *transfer*; *auto*)

end

instantiation *fset* :: (*finite*) *complete-boolean-algebra*
begin

lift-definition *uminus-fset* :: *'a fset* ⇒ *'a fset* **is** *uminus*
parametric *right-total-Compl-transfer Compl-transfer* **by** *simp*

instance
by (*standard*; *transfer*) (*simp-all add*: *Diff-eq*)

end

abbreviation *fUNIV* :: *'a::finite fset* **where** *fUNIV* ≡ *top*
abbreviation *fuminus* :: *'a::finite fset* ⇒ *'a fset* (|−| - [*81*] *80*) **where** |−| *x* ≡ *uminus x*

declare *top-fset.rep-eq*[*simp*]

45.3 Other operations

lift-definition *finsert* :: *'a* ⇒ *'a fset* ⇒ *'a fset* **is** *insert* **parametric** *Lifting-Set.insert-transfer*
by *simp*

syntax
-insert-fset :: *args* => *'a fset* ({|(-)|})

translations
{|*x*, *xs*|} == *CONST finsert x* {|*xs*|}
{|*x*|} == *CONST finsert x* {||}

lift-definition *fmember* :: *'a* ⇒ *'a fset* ⇒ *bool* (**infix** |∈| *50*) **is** *Set.member*
parametric *member-transfer* **.**

abbreviation *notin-fset* :: *'a* ⇒ *'a fset* ⇒ *bool* (**infix** |∉| *50*) **where** *x* |∉| *S* ≡ ¬ (*x* |∈| *S*)

context

begin

interpretation *lifting-syntax* **.**

lift-definition *ffilter* :: (*'a* ⇒ *bool*) ⇒ *'a fset* ⇒ *'a fset* **is** *Set.filter*
parametric *Lifting-Set.filter-transfer* **unfolding** *Set.filter-def* **by** *simp*

lift-definition *fPow* :: *'a fset* ⇒ *'a fset fset* **is** *Pow* **parametric** *Pow-transfer*
by (*simp add*: *finite-subset*)

lift-definition *fcard* :: *'a fset* ⇒ *nat* **is** *card* **parametric** *card-transfer* **.**

lift-definition *fimage* :: (*'a* ⇒ *'b*) ⇒ *'a fset* ⇒ *'b fset* (**infixr** |'| *90*) **is** *image*
parametric *image-transfer* **by** *simp*

lift-definition *fthe-elem* :: *'a fset* ⇒ *'a* **is** *the-elem* **.**

lift-definition *fbind* :: *'a fset* ⇒ (*'a* ⇒ *'b fset*) ⇒ *'b fset* **is** *Set.bind* **parametric** *bind-transfer*
by (*simp add*: *Set.bind-def*)

lift-definition *ffUnion* :: *'a fset fset* ⇒ *'a fset* **is** *Union* **parametric** *Union-transfer*
by *simp*

lift-definition *fBall* :: *'a fset* ⇒ (*'a* ⇒ *bool*) ⇒ *bool* **is** *Ball* **parametric** *Ball-transfer*
.
lift-definition *fBex* :: *'a fset* ⇒ (*'a* ⇒ *bool*) ⇒ *bool* **is** *Bex* **parametric** *Bex-transfer*
.

lift-definition *ffold* :: (*'a* ⇒ *'b* ⇒ *'b*) ⇒ *'b* ⇒ *'a fset* ⇒ *'b* **is** *Finite-Set.fold* **.**

45.4 Transferred lemmas from Set.thy

lemmas *fset-eqI* = *set-eqI*[*Transfer.transferred*]
lemmas *fset-eq-iff*[*no-atp*] = *set-eq-iff*[*Transfer.transferred*]
lemmas *fBallI*[*intro*!] = *ballI*[*Transfer.transferred*]
lemmas *fbspec*[*dest?*] = *bspec*[*Transfer.transferred*]
lemmas *fBallE*[*elim*] = *ballE*[*Transfer.transferred*]
lemmas *fBexI*[*intro*] = *bexI*[*Transfer.transferred*]
lemmas *rev-fBexI*[*intro?*] = *rev-bexI*[*Transfer.transferred*]
lemmas *fBexCI* = *bexCI*[*Transfer.transferred*]
lemmas *fBexE*[*elim*!] = *bexE*[*Transfer.transferred*]
lemmas *fBall-triv*[*simp*] = *ball-triv*[*Transfer.transferred*]
lemmas *fBex-triv*[*simp*] = *bex-triv*[*Transfer.transferred*]
lemmas *fBex-triv-one-point1*[*simp*] = *bex-triv-one-point1*[*Transfer.transferred*]
lemmas *fBex-triv-one-point2*[*simp*] = *bex-triv-one-point2*[*Transfer.transferred*]
lemmas *fBex-one-point1*[*simp*] = *bex-one-point1*[*Transfer.transferred*]
lemmas *fBex-one-point2*[*simp*] = *bex-one-point2*[*Transfer.transferred*]
lemmas *fBall-one-point1*[*simp*] = *ball-one-point1*[*Transfer.transferred*]

lemmas *fBall-one-point2*[*simp*] = *ball-one-point2*[*Transfer.transferred*]
lemmas *fBall-conj-distrib* = *ball-conj-distrib*[*Transfer.transferred*]
lemmas *fBex-disj-distrib* = *bex-disj-distrib*[*Transfer.transferred*]
lemmas *fBall-cong* = *ball-cong*[*Transfer.transferred*]
lemmas *fBex-cong* = *bex-cong*[*Transfer.transferred*]
lemmas *fsubsetI*[*intro*!] = *subsetI*[*Transfer.transferred*]
lemmas *fsubsetD*[*elim*, *intro?*] = *subsetD*[*Transfer.transferred*]
lemmas *rev-fsubsetD*[*no-atp*,*intro?*] = *rev-subsetD*[*Transfer.transferred*]
lemmas *fsubsetCE*[*no-atp*,*elim*] = *subsetCE*[*Transfer.transferred*]
lemmas *fsubset-eq*[*no-atp*] = *subset-eq*[*Transfer.transferred*]
lemmas *contra-fsubsetD*[*no-atp*] = *contra-subsetD*[*Transfer.transferred*]
lemmas *fsubset-refl* = *subset-refl*[*Transfer.transferred*]
lemmas *fsubset-trans* = *subset-trans*[*Transfer.transferred*]
lemmas *fset-rev-mp* = *set-rev-mp*[*Transfer.transferred*]
lemmas *fset-mp* = *set-mp*[*Transfer.transferred*]
lemmas *fsubset-not-fsubset-eq*[*code*] = *subset-not-subset-eq*[*Transfer.transferred*]
lemmas *eq-fmem-trans* = *eq-mem-trans*[*Transfer.transferred*]
lemmas *fsubset-antisym*[*intro*!] = *subset-antisym*[*Transfer.transferred*]
lemmas *fequalityD1* = *equalityD1*[*Transfer.transferred*]
lemmas *fequalityD2* = *equalityD2*[*Transfer.transferred*]
lemmas *fequalityE* = *equalityE*[*Transfer.transferred*]
lemmas *fequalityCE*[*elim*] = *equalityCE*[*Transfer.transferred*]
lemmas *eqfset-imp-iff* = *eqset-imp-iff*[*Transfer.transferred*]
lemmas *eqfelem-imp-iff* = *eqelem-imp-iff*[*Transfer.transferred*]
lemmas *fempty-iff*[*simp*] = *empty-iff*[*Transfer.transferred*]
lemmas *fempty-fsubsetI*[*iff*] = *empty-subsetI*[*Transfer.transferred*]
lemmas *equalsffemptyI* = *equals0I*[*Transfer.transferred*]
lemmas *equalsffemptyD* = *equals0D*[*Transfer.transferred*]
lemmas *fBall-fempty*[*simp*] = *ball-empty*[*Transfer.transferred*]
lemmas *fBex-fempty*[*simp*] = *bex-empty*[*Transfer.transferred*]
lemmas *fPow-iff*[*iff*] = *Pow-iff*[*Transfer.transferred*]
lemmas *fPowI* = *PowI*[*Transfer.transferred*]
lemmas *fPowD* = *PowD*[*Transfer.transferred*]
lemmas *fPow-bottom* = *Pow-bottom*[*Transfer.transferred*]
lemmas *fPow-top* = *Pow-top*[*Transfer.transferred*]
lemmas *fPow-not-fempty* = *Pow-not-empty*[*Transfer.transferred*]
lemmas *finter-iff*[*simp*] = *Int-iff*[*Transfer.transferred*]
lemmas *finterI*[*intro*!] = *IntI*[*Transfer.transferred*]
lemmas *finterD1* = *IntD1*[*Transfer.transferred*]
lemmas *finterD2* = *IntD2*[*Transfer.transferred*]
lemmas *finterE*[*elim*!] = *IntE*[*Transfer.transferred*]
lemmas *funion-iff*[*simp*] = *Un-iff*[*Transfer.transferred*]
lemmas *funionI1*[*elim?*] = *UnI1*[*Transfer.transferred*]
lemmas *funionI2*[*elim?*] = *UnI2*[*Transfer.transferred*]
lemmas *funionCI*[*intro*!] = *UnCI*[*Transfer.transferred*]
lemmas *funionE*[*elim*!] = *UnE*[*Transfer.transferred*]
lemmas *fminus-iff*[*simp*] = *Diff-iff*[*Transfer.transferred*]
lemmas *fminusI*[*intro*!] = *DiffI*[*Transfer.transferred*]
lemmas *fminusD1* = *DiffD1*[*Transfer.transferred*]

lemmas *fminusD2* = *DiffD2*[*Transfer.transferred*]
lemmas *fminusE*[*elim*!] = *DiffE*[*Transfer.transferred*]
lemmas *finsert-iff*[*simp*] = *insert-iff*[*Transfer.transferred*]
lemmas *finsertI1* = *insertI1*[*Transfer.transferred*]
lemmas *finsertI2* = *insertI2*[*Transfer.transferred*]
lemmas *finsertE*[*elim*!] = *insertE*[*Transfer.transferred*]
lemmas *finsertCI*[*intro*!] = *insertCI*[*Transfer.transferred*]
lemmas *fsubset-finsert-iff* = *subset-insert-iff*[*Transfer.transferred*]
lemmas *finsert-ident* = *insert-ident*[*Transfer.transferred*]
lemmas *fsingletonI*[*intro*!,*no-atp*] = *singletonI*[*Transfer.transferred*]
lemmas *fsingletonD*[*dest*!,*no-atp*] = *singletonD*[*Transfer.transferred*]
lemmas *fsingleton-iff* = *singleton-iff*[*Transfer.transferred*]
lemmas *fsingleton-inject*[*dest*!] = *singleton-inject*[*Transfer.transferred*]
lemmas *fsingleton-finsert-inj-eq*[*iff*,*no-atp*] = *singleton-insert-inj-eq*[*Transfer.transferred*]
lemmas *fsingleton-finsert-inj-eq'*[*iff*,*no-atp*] = *singleton-insert-inj-eq'*[*Transfer.transferred*]
lemmas *fsubset-fsingletonD* = *subset-singletonD*[*Transfer.transferred*]
lemmas *fminus-single-finsert* = *Diff-single-insert*[*Transfer.transferred*]
lemmas *fdoubleton-eq-iff* = *doubleton-eq-iff*[*Transfer.transferred*]
lemmas *funion-fsingleton-iff* = *Un-singleton-iff*[*Transfer.transferred*]
lemmas *fsingleton-funion-iff* = *singleton-Un-iff*[*Transfer.transferred*]
lemmas *fimage-eqI*[*simp*, *intro*] = *image-eqI*[*Transfer.transferred*]
lemmas *fimageI* = *imageI*[*Transfer.transferred*]
lemmas *rev-fimage-eqI* = *rev-image-eqI*[*Transfer.transferred*]
lemmas *fimageE*[*elim*!] = *imageE*[*Transfer.transferred*]
lemmas *Compr-fimage-eq* = *Compr-image-eq*[*Transfer.transferred*]
lemmas *fimage-funion* = *image-Un*[*Transfer.transferred*]
lemmas *fimage-iff* = *image-iff*[*Transfer.transferred*]
lemmas *fimage-fsubset-iff*[*no-atp*] = *image-subset-iff*[*Transfer.transferred*]
lemmas *fimage-fsubsetI* = *image-subsetI*[*Transfer.transferred*]
lemmas *fimage-ident*[*simp*] = *image-ident*[*Transfer.transferred*]
lemmas *if-split-fmem1* = *if-split-mem1*[*Transfer.transferred*]
lemmas *if-split-fmem2* = *if-split-mem2*[*Transfer.transferred*]
lemmas *pfsubsetI*[*intro*!,*no-atp*] = *psubsetI*[*Transfer.transferred*]
lemmas *pfsubsetE*[*elim*!,*no-atp*] = *psubsetE*[*Transfer.transferred*]
lemmas *pfsubset-finsert-iff* = *psubset-insert-iff*[*Transfer.transferred*]
lemmas *pfsubset-eq* = *psubset-eq*[*Transfer.transferred*]
lemmas *pfsubset-imp-fsubset* = *psubset-imp-subset*[*Transfer.transferred*]
lemmas *pfsubset-trans* = *psubset-trans*[*Transfer.transferred*]
lemmas *pfsubsetD* = *psubsetD*[*Transfer.transferred*]
lemmas *pfsubset-fsubset-trans* = *psubset-subset-trans*[*Transfer.transferred*]
lemmas *fsubset-pfsubset-trans* = *subset-psubset-trans*[*Transfer.transferred*]
lemmas *pfsubset-imp-ex-fmem* = *psubset-imp-ex-mem*[*Transfer.transferred*]
lemmas *fimage-fPow-mono* = *image-Pow-mono*[*Transfer.transferred*]
lemmas *fimage-fPow-surj* = *image-Pow-surj*[*Transfer.transferred*]
lemmas *fsubset-finsertI* = *subset-insertI*[*Transfer.transferred*]
lemmas *fsubset-finsertI2* = *subset-insertI2*[*Transfer.transferred*]
lemmas *fsubset-finsert* = *subset-insert*[*Transfer.transferred*]
lemmas *funion-upper1* = *Un-upper1*[*Transfer.transferred*]
lemmas *funion-upper2* = *Un-upper2*[*Transfer.transferred*]

lemmas *funion-least* = *Un-least*[*Transfer.transferred*]
lemmas *finter-lower1* = *Int-lower1*[*Transfer.transferred*]
lemmas *finter-lower2* = *Int-lower2*[*Transfer.transferred*]
lemmas *finter-greatest* = *Int-greatest*[*Transfer.transferred*]
lemmas *fminus-fsubset* = *Diff-subset*[*Transfer.transferred*]
lemmas *fminus-fsubset-conv* = *Diff-subset-conv*[*Transfer.transferred*]
lemmas *fsubset-fempty*[*simp*] = *subset-empty*[*Transfer.transferred*]
lemmas *not-pfsubset-fempty*[*iff*] = *not-psubset-empty*[*Transfer.transferred*]
lemmas *finsert-is-funion* = *insert-is-Un*[*Transfer.transferred*]
lemmas *finsert-not-fempty*[*simp*] = *insert-not-empty*[*Transfer.transferred*]
lemmas *fempty-not-finsert* = *empty-not-insert*[*Transfer.transferred*]
lemmas *finsert-absorb* = *insert-absorb*[*Transfer.transferred*]
lemmas *finsert-absorb2*[*simp*] = *insert-absorb2*[*Transfer.transferred*]
lemmas *finsert-commute* = *insert-commute*[*Transfer.transferred*]
lemmas *finsert-fsubset*[*simp*] = *insert-subset*[*Transfer.transferred*]
lemmas *finsert-inter-finsert*[*simp*] = *insert-inter-insert*[*Transfer.transferred*]
lemmas *finsert-disjoint*[*simp,no-atp*] = *insert-disjoint*[*Transfer.transferred*]
lemmas *disjoint-finsert*[*simp,no-atp*] = *disjoint-insert*[*Transfer.transferred*]
lemmas *fimage-fempty*[*simp*] = *image-empty*[*Transfer.transferred*]
lemmas *fimage-finsert*[*simp*] = *image-insert*[*Transfer.transferred*]
lemmas *fimage-constant* = *image-constant*[*Transfer.transferred*]
lemmas *fimage-constant-conv* = *image-constant-conv*[*Transfer.transferred*]
lemmas *fimage-fimage* = *image-image*[*Transfer.transferred*]
lemmas *finsert-fimage*[*simp*] = *insert-image*[*Transfer.transferred*]
lemmas *fimage-is-fempty*[*iff*] = *image-is-empty*[*Transfer.transferred*]
lemmas *fempty-is-fimage*[*iff*] = *empty-is-image*[*Transfer.transferred*]
lemmas *fimage-cong* = *image-cong*[*Transfer.transferred*]
lemmas *fimage-finter-fsubset* = *image-Int-subset*[*Transfer.transferred*]
lemmas *fimage-fminus-fsubset* = *image-diff-subset*[*Transfer.transferred*]
lemmas *finter-absorb* = *Int-absorb*[*Transfer.transferred*]
lemmas *finter-left-absorb* = *Int-left-absorb*[*Transfer.transferred*]
lemmas *finter-commute* = *Int-commute*[*Transfer.transferred*]
lemmas *finter-left-commute* = *Int-left-commute*[*Transfer.transferred*]
lemmas *finter-assoc* = *Int-assoc*[*Transfer.transferred*]
lemmas *finter-ac* = *Int-ac*[*Transfer.transferred*]
lemmas *finter-absorb1* = *Int-absorb1*[*Transfer.transferred*]
lemmas *finter-absorb2* = *Int-absorb2*[*Transfer.transferred*]
lemmas *finter-fempty-left* = *Int-empty-left*[*Transfer.transferred*]
lemmas *finter-fempty-right* = *Int-empty-right*[*Transfer.transferred*]
lemmas *disjoint-iff-fnot-equal* = *disjoint-iff-not-equal*[*Transfer.transferred*]
lemmas *finter-funion-distrib* = *Int-Un-distrib*[*Transfer.transferred*]
lemmas *finter-funion-distrib2* = *Int-Un-distrib2*[*Transfer.transferred*]
lemmas *finter-fsubset-iff*[*no-atp, simp*] = *Int-subset-iff*[*Transfer.transferred*]
lemmas *funion-absorb* = *Un-absorb*[*Transfer.transferred*]
lemmas *funion-left-absorb* = *Un-left-absorb*[*Transfer.transferred*]
lemmas *funion-commute* = *Un-commute*[*Transfer.transferred*]
lemmas *funion-left-commute* = *Un-left-commute*[*Transfer.transferred*]
lemmas *funion-assoc* = *Un-assoc*[*Transfer.transferred*]
lemmas *funion-ac* = *Un-ac*[*Transfer.transferred*]

lemmas *funion-absorb1* = *Un-absorb1*[*Transfer.transferred*]
lemmas *funion-absorb2* = *Un-absorb2*[*Transfer.transferred*]
lemmas *funion-fempty-left* = *Un-empty-left*[*Transfer.transferred*]
lemmas *funion-fempty-right* = *Un-empty-right*[*Transfer.transferred*]
lemmas *funion-finsert-left*[*simp*] = *Un-insert-left*[*Transfer.transferred*]
lemmas *funion-finsert-right*[*simp*] = *Un-insert-right*[*Transfer.transferred*]
lemmas *finter-finsert-left* = *Int-insert-left*[*Transfer.transferred*]
lemmas *finter-finsert-left-ifffempty*[*simp*] = *Int-insert-left-if0*[*Transfer.transferred*]
lemmas *finter-finsert-left-if1*[*simp*] = *Int-insert-left-if1*[*Transfer.transferred*]
lemmas *finter-finsert-right* = *Int-insert-right*[*Transfer.transferred*]
lemmas *finter-finsert-right-ifffempty*[*simp*] = *Int-insert-right-if0*[*Transfer.transferred*]
lemmas *finter-finsert-right-if1*[*simp*] = *Int-insert-right-if1*[*Transfer.transferred*]
lemmas *funion-finter-distrib* = *Un-Int-distrib*[*Transfer.transferred*]
lemmas *funion-finter-distrib2* = *Un-Int-distrib2*[*Transfer.transferred*]
lemmas *funion-finter-crazy* = *Un-Int-crazy*[*Transfer.transferred*]
lemmas *fsubset-funion-eq* = *subset-Un-eq*[*Transfer.transferred*]
lemmas *funion-fempty*[*iff*] = *Un-empty*[*Transfer.transferred*]
lemmas *funion-fsubset-iff*[*no-atp*, *simp*] = *Un-subset-iff*[*Transfer.transferred*]
lemmas *funion-fminus-finter* = *Un-Diff-Int*[*Transfer.transferred*]
lemmas *fminus-finter2* = *Diff-Int2*[*Transfer.transferred*]
lemmas *funion-finter-assoc-eq* = *Un-Int-assoc-eq*[*Transfer.transferred*]
lemmas *fBall-funion* = *ball-Un*[*Transfer.transferred*]
lemmas *fBex-funion* = *bex-Un*[*Transfer.transferred*]
lemmas *fminus-eq-fempty-iff*[*simp*,*no-atp*] = *Diff-eq-empty-iff*[*Transfer.transferred*]
lemmas *fminus-cancel*[*simp*] = *Diff-cancel*[*Transfer.transferred*]
lemmas *fminus-idemp*[*simp*] = *Diff-idemp*[*Transfer.transferred*]
lemmas *fminus-triv* = *Diff-triv*[*Transfer.transferred*]
lemmas *fempty-fminus*[*simp*] = *empty-Diff*[*Transfer.transferred*]
lemmas *fminus-fempty*[*simp*] = *Diff-empty*[*Transfer.transferred*]
lemmas *fminus-finsertffempty*[*simp*,*no-atp*] = *Diff-insert0*[*Transfer.transferred*]
lemmas *fminus-finsert* = *Diff-insert*[*Transfer.transferred*]
lemmas *fminus-finsert2* = *Diff-insert2*[*Transfer.transferred*]
lemmas *finsert-fminus-if* = *insert-Diff-if*[*Transfer.transferred*]
lemmas *finsert-fminus1*[*simp*] = *insert-Diff1*[*Transfer.transferred*]
lemmas *finsert-fminus-single*[*simp*] = *insert-Diff-single*[*Transfer.transferred*]
lemmas *finsert-fminus* = *insert-Diff*[*Transfer.transferred*]
lemmas *fminus-finsert-absorb* = *Diff-insert-absorb*[*Transfer.transferred*]
lemmas *fminus-disjoint*[*simp*] = *Diff-disjoint*[*Transfer.transferred*]
lemmas *fminus-partition* = *Diff-partition*[*Transfer.transferred*]
lemmas *double-fminus* = *double-diff*[*Transfer.transferred*]
lemmas *funion-fminus-cancel*[*simp*] = *Un-Diff-cancel*[*Transfer.transferred*]
lemmas *funion-fminus-cancel2*[*simp*] = *Un-Diff-cancel2*[*Transfer.transferred*]
lemmas *fminus-funion* = *Diff-Un*[*Transfer.transferred*]
lemmas *fminus-finter* = *Diff-Int*[*Transfer.transferred*]
lemmas *funion-fminus* = *Un-Diff*[*Transfer.transferred*]
lemmas *finter-fminus* = *Int-Diff*[*Transfer.transferred*]
lemmas *fminus-finter-distrib* = *Diff-Int-distrib*[*Transfer.transferred*]
lemmas *fminus-finter-distrib2* = *Diff-Int-distrib2*[*Transfer.transferred*]
lemmas *fUNIV-bool*[*no-atp*] = *UNIV-bool*[*Transfer.transferred*]

lemmas *fPow-fempty*[*simp*] = *Pow-empty*[*Transfer.transferred*]
lemmas *fPow-finsert* = *Pow-insert*[*Transfer.transferred*]
lemmas *funion-fPow-fsubset* = *Un-Pow-subset*[*Transfer.transferred*]
lemmas *fPow-finter-eq*[*simp*] = *Pow-Int-eq*[*Transfer.transferred*]
lemmas *fset-eq-fsubset* = *set-eq-subset*[*Transfer.transferred*]
lemmas *fsubset-iff* [*no-atp*] = *subset-iff* [*Transfer.transferred*]
lemmas *fsubset-iff-pfsubset-eq* = *subset-iff-psubset-eq*[*Transfer.transferred*]
lemmas *all-not-fin-conv*[*simp*] = *all-not-in-conv*[*Transfer.transferred*]
lemmas *ex-fin-conv* = *ex-in-conv*[*Transfer.transferred*]
lemmas *fimage-mono* = *image-mono*[*Transfer.transferred*]
lemmas *fPow-mono* = *Pow-mono*[*Transfer.transferred*]
lemmas *finsert-mono* = *insert-mono*[*Transfer.transferred*]
lemmas *funion-mono* = *Un-mono*[*Transfer.transferred*]
lemmas *finter-mono* = *Int-mono*[*Transfer.transferred*]
lemmas *fminus-mono* = *Diff-mono*[*Transfer.transferred*]
lemmas *fin-mono* = *in-mono*[*Transfer.transferred*]
lemmas *fthe-felem-eq*[*simp*] = *the-elem-eq*[*Transfer.transferred*]
lemmas *fLeast-mono* = *Least-mono*[*Transfer.transferred*]
lemmas *fbind-fbind* = *bind-bind*[*Transfer.transferred*]
lemmas *fempty-fbind*[*simp*] = *empty-bind*[*Transfer.transferred*]
lemmas *nonfempty-fbind-const* = *nonempty-bind-const*[*Transfer.transferred*]
lemmas *fbind-const* = *bind-const*[*Transfer.transferred*]
lemmas *ffmember-filter*[*simp*] = *member-filter*[*Transfer.transferred*]
lemmas *fequalityI* = *equalityI*[*Transfer.transferred*]

45.5 Additional lemmas

45.5.1 *fsingleton*

lemmas *fsingletonE* = *fsingletonD* [*elim-format*]

45.5.2 *femepty*

lemma *fempty-ffilter*[*simp*]: *ffilter* (λ-. *False*) *A* = {||}
by *transfer auto*

lemma *femptyE* [*elim*!]: *a* |∈| {||} ⟹ *P*
by *simp*

45.5.3 *fset*

lemmas *fset-simps*[*simp*] = *bot-fset.rep-eq finsert.rep-eq*

lemma *finite-fset* [*simp*]:
shows *finite* (*fset S*)
by *transfer simp*

lemmas *fset-cong* = *fset-inject*

lemma *filter-fset* [*simp*]:
shows *fset* (*ffilter P xs*) = *Collect P* ∩ *fset xs*
by *transfer auto*

lemma *notin-fset*: x |∉| S ⟷ x ∉ *fset S* **by** (*simp add*: *fmember.rep-eq*)

lemmas *inter-fset*[*simp*] = *inf-fset.rep-eq*

lemmas *union-fset*[*simp*] = *sup-fset.rep-eq*

lemmas *minus-fset*[*simp*] = *minus-fset.rep-eq*

45.5.4 *filter-fset*

lemma *subset-ffilter*:
ffilter P A |⊆| *ffilter Q A* = (∀ *x*. *x* |∈| *A* ⟶ *P x* ⟶ *Q x*)
by *transfer auto*

lemma *eq-ffilter*:
(*ffilter P A* = *ffilter Q A*) = (∀ *x*. *x* |∈| *A* ⟶ *P x* = *Q x*)
by *transfer auto*

lemma *pfsubset-ffilter*:
(⋀*x*. *x* |∈| *A* ⟹ *P x* ⟹ *Q x*) ⟹ (*x* |∈| *A* & ¬ *P x* & *Q x*) ⟹
ffilter P A |⊂| *ffilter Q A*
unfolding *less-fset-def* **by** (*auto simp add*: *subset-ffilter eq-ffilter*)

45.5.5 *finsert*

lemma *set-finsert*:
assumes *x* |∈| *A*
obtains *B* **where** *A* = *finsert x B* **and** *x* |∉| *B*
using *assms* **by** *transfer* (*metis Set.set-insert finite-insert*)

lemma *mk-disjoint-finsert*: *a* |∈| *A* ⟹ ∃ *B*. *A* = *finsert a B* ∧ *a* |∉| *B*
by (*rule-tac x* = *A* |−| {|*a*|} **in** *exI*, *blast*)

45.5.6 *fimage*

lemma *subset-fimage-iff*: (*B* |⊆| *f*|‘|*A*) = (∃ *AA*. *AA* |⊆| *A* ∧ *B* = *f*|‘|*AA*)
by *transfer* (*metis mem-Collect-eq rev-finite-subset subset-image-iff*)

45.5.7 bounded quantification

lemma *bex-simps* [*simp*, *no-atp*]:
⋀*A P Q*. *fBex A* (λ*x*. *P x* ∧ *Q*) = (*fBex A P* ∧ *Q*)
⋀*A P Q*. *fBex A* (λ*x*. *P* ∧ *Q x*) = (*P* ∧ *fBex A Q*)
⋀*P*. *fBex* {||} *P* = *False*
⋀*a B P*. *fBex* (*finsert a B*) *P* = (*P a* ∨ *fBex B P*)
⋀*A P f*. *fBex* (*f* |‘| *A*) *P* = *fBex A* (λ*x*. *P* (*f x*))

$\bigwedge A\ P.\ (\neg\ fBex\ A\ P) = fBall\ A\ (\lambda x.\ \neg\ P\ x)$
by *auto*

lemma *ball-simps* [*simp*, *no-atp*]:
$\bigwedge A\ P\ Q.\ fBall\ A\ (\lambda x.\ P\ x \vee Q) = (fBall\ A\ P \vee Q)$
$\bigwedge A\ P\ Q.\ fBall\ A\ (\lambda x.\ P \vee Q\ x) = (P \vee fBall\ A\ Q)$
$\bigwedge A\ P\ Q.\ fBall\ A\ (\lambda x.\ P \longrightarrow Q\ x) = (P \longrightarrow fBall\ A\ Q)$
$\bigwedge A\ P\ Q.\ fBall\ A\ (\lambda x.\ P\ x \longrightarrow Q) = (fBex\ A\ P \longrightarrow Q)$
$\bigwedge P.\ fBall\ \{||\}\ P = True$
$\bigwedge a\ B\ P.\ fBall\ (finsert\ a\ B)\ P = (P\ a \wedge fBall\ B\ P)$
$\bigwedge A\ P\ f.\ fBall\ (f\ |`|\ A)\ P = fBall\ A\ (\lambda x.\ P\ (f\ x))$
$\bigwedge A\ P.\ (\neg\ fBall\ A\ P) = fBex\ A\ (\lambda x.\ \neg\ P\ x)$
by *auto*

lemma *atomize-fBall*:
$(\bigwedge x.\ x\ |\in|\ A ==> P\ x) == Trueprop\ (fBall\ A\ (\lambda x.\ P\ x))$
apply (*simp only*: *atomize-all atomize-imp*)
apply (*rule equal-intr-rule*)
by (*transfer*, *simp*)+

end

45.5.8 *fcard*

lemma *fcard-fempty*:
$fcard\ \{||\} = 0$
by *transfer* (*rule card-empty*)

lemma *fcard-finsert-disjoint*:
$x\ |\notin|\ A \Longrightarrow fcard\ (finsert\ x\ A) = Suc\ (fcard\ A)$
by *transfer* (*rule card-insert-disjoint*)

lemma *fcard-finsert-if*:
$fcard\ (finsert\ x\ A) = (if\ x\ |\in|\ A\ then\ fcard\ A\ else\ Suc\ (fcard\ A))$
by *transfer* (*rule card-insert-if*)

lemma *card-0-eq* [*simp*, *no-atp*]:
$fcard\ A = 0 \longleftrightarrow A = \{||\}$
by *transfer* (*rule card-0-eq*)

lemma *fcard-Suc-fminus1*:
$x\ |\in|\ A \Longrightarrow Suc\ (fcard\ (A\ |-|\ \{|x|\})) = fcard\ A$
by *transfer* (*rule card-Suc-Diff1*)

lemma *fcard-fminus-fsingleton*:
$x\ |\in|\ A \Longrightarrow fcard\ (A\ |-|\ \{|x|\}) = fcard\ A - 1$
by *transfer* (*rule card-Diff-singleton*)

lemma *fcard-fminus-fsingleton-if*:

fcard $(A \mid - \mid \{|x|\}) = ($*if* $x \mid\in\mid A$ *then fcard* $A - 1$ *else fcard* $A)$
by *transfer* (*rule card-Diff-singleton-if*)

lemma *fcard-fminus-finsert*[*simp*]:
assumes $a \mid\in\mid A$ **and** $a \mid\notin\mid B$
shows *fcard* $(A \mid - \mid$ *finsert* $a\ B) =$ *fcard* $(A \mid - \mid B) - 1$
using *assms* **by** *transfer* (*rule card-Diff-insert*)

lemma *fcard-finsert*: *fcard* (*finsert* $x\ A$) = *Suc* (*fcard* $(A \mid - \mid \{|x|\})$)
by *transfer* (*rule card-insert*)

lemma *fcard-finsert-le*: *fcard* $A \leq$ *fcard* (*finsert* $x\ A$)
by *transfer* (*rule card-insert-le*)

lemma *fcard-mono*:
$A \mid\subseteq\mid B \Longrightarrow$ *fcard* $A \leq$ *fcard* B
by *transfer* (*rule card-mono*)

lemma *fcard-seteq*: $A \mid\subseteq\mid B \Longrightarrow$ *fcard* $B \leq$ *fcard* $A \Longrightarrow A = B$
by *transfer* (*rule card-seteq*)

lemma *pfsubset-fcard-mono*: $A \mid\subset\mid B \Longrightarrow$ *fcard* $A <$ *fcard* B
by *transfer* (*rule psubset-card-mono*)

lemma *fcard-funion-finter*:
fcard A + *fcard* B = *fcard* $(A \mid\cup\mid B)$ + *fcard* $(A \mid\cap\mid B)$
by *transfer* (*rule card-Un-Int*)

lemma *fcard-funion-disjoint*:
$A \mid\cap\mid B = \{||\} \Longrightarrow$ *fcard* $(A \mid\cup\mid B)$ = *fcard* A + *fcard* B
by *transfer* (*rule card-Un-disjoint*)

lemma *fcard-funion-fsubset*:
$B \mid\subseteq\mid A \Longrightarrow$ *fcard* $(A \mid - \mid B)$ = *fcard* A − *fcard* B
by *transfer* (*rule card-Diff-subset*)

lemma *diff-fcard-le-fcard-fminus*:
fcard A − *fcard* $B \leq$ *fcard*$(A \mid - \mid B)$
by *transfer* (*rule diff-card-le-card-Diff*)

lemma *fcard-fminus1-less*: $x \mid\in\mid A \Longrightarrow$ *fcard* $(A \mid - \mid \{|x|\}) <$ *fcard* A
by *transfer* (*rule card-Diff1-less*)

lemma *fcard-fminus2-less*:
$x \mid\in\mid A \Longrightarrow y \mid\in\mid A \Longrightarrow$ *fcard* $(A \mid - \mid \{|x|\} \mid - \mid \{|y|\}) <$ *fcard* A
by *transfer* (*rule card-Diff2-less*)

lemma *fcard-fminus1-le*: *fcard* $(A \mid - \mid \{|x|\}) \leq$ *fcard* A
by *transfer* (*rule card-Diff1-le*)

lemma *fcard-pfsubset*: A |⊆| $B \Longrightarrow$ *fcard* $A <$ *fcard* $B \Longrightarrow A < B$
by *transfer* (*rule card-psubset*)

45.5.9 *ffold*

context *comp-fun-commute*
begin
lemmas *ffold-empty*[*simp*] = *fold-empty*[*Transfer.transferred*]

lemma *ffold-finsert* [*simp*]:
assumes x |∉| A
shows *ffold f z* (*finsert x A*) = *f x* (*ffold f z A*)
using *assms* **by** (*transfer fixing*: *f*) (*rule fold-insert*)

lemma *ffold-fun-left-comm*:
f x (*ffold f z A*) = *ffold f* (*f x z*) *A*
by (*transfer fixing*: *f*) (*rule fold-fun-left-comm*)

lemma *ffold-finsert2*:
x |∉| $A \Longrightarrow$ *ffold f z* (*finsert x A*) = *ffold f* (*f x z*) *A*
by (*transfer fixing*: *f*) (*rule fold-insert2*)

lemma *ffold-rec*:
assumes x |∈| A
shows *ffold f z A* = *f x* (*ffold f z* (*A* |−| {|*x*|}))
using *assms* **by** (*transfer fixing*: *f*) (*rule fold-rec*)

lemma *ffold-finsert-fremove*:
ffold f z (*finsert x A*) = *f x* (*ffold f z* (*A* |−| {|*x*|}))
by (*transfer fixing*: *f*) (*rule fold-insert-remove*)
end

lemma *ffold-fimage*:
assumes *inj-on g* (*fset A*)
shows *ffold f z* (*g* |‘| *A*) = *ffold* ($f \circ g$) *z A*
using *assms* **by** *transfer′* (*rule fold-image*)

lemma *ffold-cong*:
assumes *comp-fun-commute f comp-fun-commute g*
$\bigwedge x.\ x$ |∈| $A \Longrightarrow f\ x = g\ x$
and $s = t$ **and** $A = B$
shows *ffold f s A* = *ffold g t B*
using *assms* **by** *transfer* (*metis Finite-Set.fold-cong*)

context *comp-fun-idem*
begin

lemma *ffold-finsert-idem*:

ffold f z (*finsert x A*) = *f x* (*ffold f z A*)
by (*transfer fixing*: *f*) (*rule fold-insert-idem*)

declare *ffold-finsert* [*simp del*] *ffold-finsert-idem* [*simp*]

lemma *ffold-finsert-idem2*:
ffold f z (*finsert x A*) = *ffold f* (*f x z*) *A*
by (*transfer fixing*: *f*) (*rule fold-insert-idem2*)

end

45.6 Choice in fsets

lemma *fset-choice*:
assumes $\forall x.\ x\ |{\in}|\ A \longrightarrow (\exists y.\ P\ x\ y)$
shows $\exists f.\ \forall x.\ x\ |{\in}|\ A \longrightarrow P\ x\ (f\ x)$
using *assms* **by** *transfer metis*

45.7 Induction and Cases rules for fsets

lemma *fset-exhaust* [*case-names empty insert*, *cases type*: *fset*]:
assumes *fempty-case*: $S = \{||\} \Longrightarrow P$
and *finsert-case*: $\bigwedge x\ S'.\ S = \mathit{finsert}\ x\ S' \Longrightarrow P$
shows *P*
using *assms* **by** *transfer blast*

lemma *fset-induct* [*case-names empty insert*]:
assumes *fempty-case*: $P\ \{||\}$
and *finsert-case*: $\bigwedge x\ S.\ P\ S \Longrightarrow P\ (\mathit{finsert}\ x\ S)$
shows *P S*
proof −

note *Domainp-forall-transfer*[*transfer-rule*]
show *?thesis*
using *assms* **by** *transfer* (*auto intro*: *finite-induct*)
qed

lemma *fset-induct-stronger* [*case-names empty insert*, *induct type*: *fset*]:
assumes *empty-fset-case*: $P\ \{||\}$
and *insert-fset-case*: $\bigwedge x\ S.\ [\![x\ |{\notin}|\ S;\ P\ S]\!] \Longrightarrow P\ (\mathit{finsert}\ x\ S)$
shows *P S*
proof −

note *Domainp-forall-transfer*[*transfer-rule*]
show *?thesis*
using *assms* **by** *transfer* (*auto intro*: *finite-induct*)
qed

lemma *fset-card-induct*:
assumes *empty-fset-case*: $P\ \{||\}$

and *card-fset-Suc-case*: $\bigwedge S\ T.\ Suc\ (fcard\ S) = (fcard\ T) \Longrightarrow P\ S \Longrightarrow P\ T$
shows $P\ S$
proof (*induct S*)
case *empty*
show P {||} **by** (*rule empty-fset-case*)
next
case (*insert x S*)
have *h*: $P\ S$ **by** *fact*
have x |∉| S **by** *fact*
then have *Suc* (*fcard S*) = *fcard* (*finsert x S*)
by *transfer auto*
then show *P* (*finsert x S*)
using *h card-fset-Suc-case* **by** *simp*
qed

lemma *fset-strong-cases*:
obtains *xs* = {||}
| *ys x* **where** *x* |∉| *ys* **and** *xs* = *finsert x ys*
by *transfer blast*

lemma *fset-induct2*:
P {||} {||} $\Longrightarrow$
($\bigwedge$*x xs. x* |∉| *xs* $\Longrightarrow$ *P* (*finsert x xs*) {||}) $\Longrightarrow$
($\bigwedge$*y ys. y* |∉| *ys* $\Longrightarrow$ *P* {||} (*finsert y ys*)) $\Longrightarrow$
($\bigwedge$*x xs y ys.* ⟦*P xs ys*; *x* |∉| *xs*; *y* |∉| *ys*⟧ $\Longrightarrow$ *P* (*finsert x xs*) (*finsert y ys*)) $\Longrightarrow$
P xsa ysa
apply (*induct xsa arbitrary*: *ysa*)
apply (*induct-tac x rule*: *fset-induct-stronger*)
apply *simp-all*
apply (*induct-tac xa rule*: *fset-induct-stronger*)
apply *simp-all*
done

45.8 Setup for Lifting/Transfer

45.8.1 Relator and predicator properties

lift-definition *rel-fset* :: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ '*a fset* ⇒ '*b fset* ⇒ *bool* **is** *rel-set*
parametric *rel-set-transfer* **.**

lemma *rel-fset-alt-def*: *rel-fset R* = (λ*A B*. (∀ *x*.∃ *y*. *x*|∈|*A* ⟶ *y*|∈|*B* ∧ *R x y*)
∧ (∀ *y*. ∃ *x*. *y*|∈|*B* ⟶ *x*|∈|*A* ∧ *R x y*))
apply (*rule ext*)+
apply *transfer'*
apply (*subst rel-set-def*[*unfolded fun-eq-iff*])
by *blast*

lemma *finite-rel-set*:
assumes *fin*: *finite X finite Z*
assumes *R-S*: *rel-set* (*R OO S*) *X Z*

shows ∃ *Y*. *finite Y* ∧ *rel-set R X Y* ∧ *rel-set S Y Z*
proof –
obtain *f* **where** *f*: ∀ *x*∈*X*. *R x* (*f x*) ∧ (∃ *z*∈*Z*. *S* (*f x*) *z*)
apply *atomize-elim*
apply (*subst bchoice-iff* [*symmetric*])
using *R-S*[*unfolded rel-set-def OO-def*] **by** *blast*

obtain *g* **where** *g*: ∀ *z*∈*Z*. *S* (*g z*) *z* ∧ (∃ *x*∈*X*. *R x* (*g z*))
apply *atomize-elim*
apply (*subst bchoice-iff* [*symmetric*])
using *R-S*[*unfolded rel-set-def OO-def*] **by** *blast*

let *?Y* = *f* ' *X* ∪ *g* ' *Z*
have *finite ?Y* **by** (*simp add*: *fin*)
moreover have *rel-set R X ?Y*
unfolding *rel-set-def*
using *f g* **by** *clarsimp blast*
moreover have *rel-set S ?Y Z*
unfolding *rel-set-def*
using *f g* **by** *clarsimp blast*
ultimately show *?thesis* **by** *metis*
qed

45.8.2 Transfer rules for the Transfer package

Unconditional transfer rules

context
begin

interpretation *lifting-syntax* .

lemmas *fempty-transfer* [*transfer-rule*] = *empty-transfer*[*Transfer.transferred*]

lemma *finsert-transfer* [*transfer-rule*]:
(*A* ===> *rel-fset A* ===> *rel-fset A*) *finsert finsert*
unfolding *rel-fun-def rel-fset-alt-def* **by** *blast*

lemma *funion-transfer* [*transfer-rule*]:
(*rel-fset A* ===> *rel-fset A* ===> *rel-fset A*) *funion funion*
unfolding *rel-fun-def rel-fset-alt-def* **by** *blast*

lemma *ffUnion-transfer* [*transfer-rule*]:
(*rel-fset* (*rel-fset A*) ===> *rel-fset A*) *ffUnion ffUnion*
unfolding *rel-fun-def rel-fset-alt-def* **by** *transfer* (*simp*, *fast*)

lemma *fimage-transfer* [*transfer-rule*]:
((*A* ===> *B*) ===> *rel-fset A* ===> *rel-fset B*) *fimage fimage*
unfolding *rel-fun-def rel-fset-alt-def* **by** *simp blast*

lemma *fBall-transfer* [*transfer-rule*]:
(*rel-fset A* ===> (*A* ===> *op* =) ===> *op* =) *fBall fBall*
unfolding *rel-fset-alt-def rel-fun-def* **by** *blast*

lemma *fBex-transfer* [*transfer-rule*]:
(*rel-fset A* ===> (*A* ===> *op* =) ===> *op* =) *fBex fBex*
unfolding *rel-fset-alt-def rel-fun-def* **by** *blast*

lemma *fPow-transfer* [*transfer-rule*]:
(*rel-fset A* ===> *rel-fset* (*rel-fset A*)) *fPow fPow*
unfolding *rel-fun-def*
using *Pow-transfer*[*unfolded rel-fun-def*, *rule-format*, *Transfer.transferred*]
by *blast*

lemma *rel-fset-transfer* [*transfer-rule*]:
((*A* ===> *B* ===> *op* =) ===> *rel-fset A* ===> *rel-fset B* ===> *op* =)
rel-fset rel-fset
unfolding *rel-fun-def*
using *rel-set-transfer*[*unfolded rel-fun-def*,*rule-format*, *Transfer.transferred*, **where** *A* = *A* **and** *B* = *B*]
by *simp*

lemma *bind-transfer* [*transfer-rule*]:
(*rel-fset A* ===> (*A* ===> *rel-fset B*) ===> *rel-fset B*) *fbind fbind*
using *assms* **unfolding** *rel-fun-def*
using *bind-transfer*[*unfolded rel-fun-def*, *rule-format*, *Transfer.transferred*] **by** *blast*

Rules requiring bi-unique, bi-total or right-total relations

lemma *fmember-transfer* [*transfer-rule*]:
assumes *bi-unique A*
shows (*A* ===> *rel-fset A* ===> *op* =) (*op* |∈|) (*op* |∈|)
using *assms* **unfolding** *rel-fun-def rel-fset-alt-def bi-unique-def* **by** *metis*

lemma *finter-transfer* [*transfer-rule*]:
assumes *bi-unique A*
shows (*rel-fset A* ===> *rel-fset A* ===> *rel-fset A*) *finter finter*
using *assms* **unfolding** *rel-fun-def*
using *inter-transfer*[*unfolded rel-fun-def*, *rule-format*, *Transfer.transferred*] **by** *blast*

lemma *fminus-transfer* [*transfer-rule*]:
assumes *bi-unique A*
shows (*rel-fset A* ===> *rel-fset A* ===> *rel-fset A*) (*op* |−|) (*op* |−|)
using *assms* **unfolding** *rel-fun-def*
using *Diff-transfer*[*unfolded rel-fun-def*, *rule-format*, *Transfer.transferred*] **by** *blast*

lemma *fsubset-transfer* [*transfer-rule*]:
 assumes *bi-unique A*
 shows (*rel-fset A* ===> *rel-fset A* ===> *op* =) (*op* |⊆|) (*op* |⊆|)
 using *assms* **unfolding** *rel-fun-def*
 using *subset-transfer*[*unfolded rel-fun-def*, *rule-format*, *Transfer.transferred*] **by** *blast*

lemma *fSup-transfer* [*transfer-rule*]:
 bi-unique A ⟹ (*rel-set* (*rel-fset A*) ===> *rel-fset A*) *Sup Sup*
 using *assms* **unfolding** *rel-fun-def*
 apply *clarify*
 apply *transfer'*
 using *Sup-fset-transfer*[*unfolded rel-fun-def*] **by** *blast*

lemma *fInf-transfer* [*transfer-rule*]:
 assumes *bi-unique A* **and** *bi-total A*
 shows (*rel-set* (*rel-fset A*) ===> *rel-fset A*) *Inf Inf*
 using *assms* **unfolding** *rel-fun-def*
 apply *clarify*
 apply *transfer'*
 using *Inf-fset-transfer*[*unfolded rel-fun-def*] **by** *blast*

lemma *ffilter-transfer* [*transfer-rule*]:
 assumes *bi-unique A*
 shows ((*A* ===> *op*=) ===> *rel-fset A* ===> *rel-fset A*) *ffilter ffilter*
 using *assms* **unfolding** *rel-fun-def*
 using *Lifting-Set.filter-transfer*[*unfolded rel-fun-def*, *rule-format*, *Transfer.transferred*] **by** *blast*

lemma *card-transfer* [*transfer-rule*]:
 bi-unique A ⟹ (*rel-fset A* ===> *op* =) *fcard fcard*
 using *assms* **unfolding** *rel-fun-def*
 using *card-transfer*[*unfolded rel-fun-def*, *rule-format*, *Transfer.transferred*] **by** *blast*

end

lifting-update *fset.lifting*
lifting-forget *fset.lifting*

45.9 BNF setup

context
includes *fset.lifting*
begin

lemma *rel-fset-alt*:

rel-fset R a b ⟷ (∀ *t* ∈ *fset a*. ∃ *u* ∈ *fset b*. *R t u*) ∧ (∀ *t* ∈ *fset b*. ∃ *u* ∈ *fset a*. *R u t*)
by *transfer* (*simp add*: *rel-set-def*)

lemma *fset-to-fset*: *finite A* ⟹ *fset* (*the-inv fset A*) = *A*
apply (*rule f-the-inv-into-f*[*unfolded inj-on-def*])
apply (*simp add*: *fset-inject*)
apply (*rule range-eqI Abs-fset-inverse*[*symmetric*] *CollectI*)+
.

lemma *rel-fset-aux*:
(∀ *t* ∈ *fset a*. ∃ *u* ∈ *fset b*. *R t u*) ∧ (∀ *u* ∈ *fset b*. ∃ *t* ∈ *fset a*. *R t u*) ⟷
((*BNF-Def.Grp* {*a*. *fset a* ⊆ {(*a*, *b*). *R a b*}} (*fimage fst*))$^{-1-1}$ *OO*
BNF-Def.Grp {*a*. *fset a* ⊆ {(*a*, *b*). *R a b*}} (*fimage snd*)) *a b* (**is** *?L* = *?R*)
proof
assume *?L*
def *R′* ≡ *the-inv fset* (*Collect* (*case-prod R*) ∩ (*fset a* × *fset b*)) (**is** *the-inv fset ?L′*)
have *finite ?L′* **by** (*intro finite-Int*[*OF disjI2*] *finite-cartesian-product*) (*transfer*, *simp*)+
hence ∗: *fset R′* = *?L′* **unfolding** *R′-def* **by** (*intro fset-to-fset*)
show *?R* **unfolding** *Grp-def relcompp.simps conversep.simps*
proof (*intro CollectI case-prodI exI*[*of - a*] *exI*[*of - b*] *exI*[*of - R′*] *conjI refl*)
from ∗ **show** *a* = *fimage fst R′* **using** *conjunct1*[*OF* ‹*?L*›]
by (*transfer*, *auto simp add*: *image-def Int-def split*: *prod.splits*)
from ∗ **show** *b* = *fimage snd R′* **using** *conjunct2*[*OF* ‹*?L*›]
by (*transfer*, *auto simp add*: *image-def Int-def split*: *prod.splits*)
qed (*auto simp add*: ∗)
next
assume *?R* **thus** *?L* **unfolding** *Grp-def relcompp.simps conversep.simps*
apply (*simp add*: *subset-eq Ball-def*)
apply (*rule conjI*)
apply (*transfer*, *clarsimp*, *metis snd-conv*)
by (*transfer*, *clarsimp*, *metis fst-conv*)
qed

bnf *′a fset*
map: *fimage*
sets: *fset*
bd: *natLeq*
wits: {||}
rel: *rel-fset*
apply −
apply *transfer′* **apply** *simp*
apply *transfer′* **apply** *force*
apply *transfer* **apply** *force*
apply *transfer′* **apply** *force*
apply (*rule natLeq-card-order*)
apply (*rule natLeq-cinfinite*)

apply *transfer* **apply** (*metis ordLess-imp-ordLeq finite-iff-ordLess-natLeq*)
apply (*fastforce simp*: *rel-fset-alt*)
apply (*simp add*: *Grp-def relcompp.simps conversep.simps fun-eq-iff rel-fset-alt rel-fset-aux*[*unfolded OO-Grp-alt*])
apply *transfer* **apply** *simp*
done

lemma *rel-fset-fset*: *rel-set* χ (*fset A1*) (*fset A2*) = *rel-fset* χ *A1 A2*
by *transfer* (*rule refl*)

end

lemmas [*simp*] = *fset.map-comp fset.map-id fset.set-map*

45.10 Size setup

context includes *fset.lifting* **begin**
lift-definition *size-fset* :: (*'a* ⇒ *nat*) ⇒ *'a fset* ⇒ *nat* **is** λf. *setsum* (*Suc* ∘ *f*) **.**
end

instantiation *fset* :: (*type*) *size* **begin**
definition *size-fset* **where**
size-fset-overloaded-def: *size-fset* = *FSet.size-fset* (λ-. *0*)
instance ..
end

lemmas *size-fset-simps*[*simp*] =
size-fset-def[*THEN meta-eq-to-obj-eq*, *THEN fun-cong*, *THEN fun-cong*, *unfolded map-fun-def comp-def id-apply*]

lemmas *size-fset-overloaded-simps*[*simp*] =
size-fset-simps[*of* λ-. *0*, *unfolded add-0-left add-0-right*, *folded size-fset-overloaded-def*]

lemma *fset-size-o-map*: *inj f* ⟹ *size-fset g* ∘ *fimage f* = *size-fset* (*g* ∘ *f*)
apply (*subst fun-eq-iff*)
including *fset.lifting* **by** *transfer* (*auto intro*: *setsum.reindex-cong subset-inj-on*)

setup ‹
BNF-LFP-Size.register-size-global @{*type-name fset*} @{*const-name size-fset*}
@{*thm size-fset-overloaded-def*} @{*thms size-fset-simps size-fset-overloaded-simps*}
@{*thms fset-size-o-map*}
›

lifting-update *fset.lifting*
lifting-forget *fset.lifting*

45.11 Advanced relator customization

lemma *rel-set-rel-sum*[*simp*]:

rel-set (*rel-sum* χ φ) *A1 A2* $\longleftrightarrow$
rel-set χ (*Inl* $-$' *A1*) (*Inl* $-$' *A2*) $\wedge$ *rel-set* φ (*Inr* $-$' *A1*) (*Inr* $-$' *A2*)
(**is** *?L* $\longleftrightarrow$ *?Rl* $\wedge$ *?Rr*)
proof *safe*
  assume *L*: *?L*
  show *?Rl* **unfolding** *rel-set-def Bex-def vimage-eq* **proof** *safe*
    fix *l1* **assume** *Inl l1* $\in$ *A1*
    then obtain *a2* **where** *a2*: *a2* $\in$ *A2* **and** *rel-sum* χ φ (*Inl l1*) *a2*
    using *L* **unfolding** *rel-set-def* **by** *auto*
    then obtain *l2* **where** *a2* = *Inl l2* $\wedge$ χ *l1 l2* **by** (*cases a2*, *auto*)
    thus $\exists$ *l2*. *Inl l2* $\in$ *A2* $\wedge$ χ *l1 l2* **using** *a2* **by** *auto*
  next
    fix *l2* **assume** *Inl l2* $\in$ *A2*
    then obtain *a1* **where** *a1*: *a1* $\in$ *A1* **and** *rel-sum* χ φ *a1* (*Inl l2*)
    using *L* **unfolding** *rel-set-def* **by** *auto*
    then obtain *l1* **where** *a1* = *Inl l1* $\wedge$ χ *l1 l2* **by** (*cases a1*, *auto*)
    thus $\exists$ *l1*. *Inl l1* $\in$ *A1* $\wedge$ χ *l1 l2* **using** *a1* **by** *auto*
  qed
  show *?Rr* **unfolding** *rel-set-def Bex-def vimage-eq* **proof** *safe*
    fix *r1* **assume** *Inr r1* $\in$ *A1*
    then obtain *a2* **where** *a2*: *a2* $\in$ *A2* **and** *rel-sum* χ φ (*Inr r1*) *a2*
    using *L* **unfolding** *rel-set-def* **by** *auto*
    then obtain *r2* **where** *a2* = *Inr r2* $\wedge$ φ *r1 r2* **by** (*cases a2*, *auto*)
    thus $\exists$ *r2*. *Inr r2* $\in$ *A2* $\wedge$ φ *r1 r2* **using** *a2* **by** *auto*
  next
    fix *r2* **assume** *Inr r2* $\in$ *A2*
    then obtain *a1* **where** *a1*: *a1* $\in$ *A1* **and** *rel-sum* χ φ *a1* (*Inr r2*)
    using *L* **unfolding** *rel-set-def* **by** *auto*
    then obtain *r1* **where** *a1* = *Inr r1* $\wedge$ φ *r1 r2* **by** (*cases a1*, *auto*)
    thus $\exists$ *r1*. *Inr r1* $\in$ *A1* $\wedge$ φ *r1 r2* **using** *a1* **by** *auto*
  qed
next
  assume *Rl*: *?Rl* **and** *Rr*: *?Rr*
  show *?L* **unfolding** *rel-set-def Bex-def vimage-eq* **proof** *safe*
    fix *a1* **assume** *a1*: *a1* $\in$ *A1*
    show $\exists$ *a2*. *a2* $\in$ *A2* $\wedge$ *rel-sum* χ φ *a1 a2*
    proof(*cases a1*)
      case (*Inl l1*) **then obtain** *l2* **where** *Inl l2* $\in$ *A2* $\wedge$ χ *l1 l2*
      using *Rl a1* **unfolding** *rel-set-def* **by** *blast*
      thus *?thesis* **unfolding** *Inl* **by** *auto*
    next
      case (*Inr r1*) **then obtain** *r2* **where** *Inr r2* $\in$ *A2* $\wedge$ φ *r1 r2*
      using *Rr a1* **unfolding** *rel-set-def* **by** *blast*
      thus *?thesis* **unfolding** *Inr* **by** *auto*
    qed
  next
    fix *a2* **assume** *a2*: *a2* $\in$ *A2*
    show $\exists$ *a1*. *a1* $\in$ *A1* $\wedge$ *rel-sum* χ φ *a1 a2*
    proof(*cases a2*)

case (*Inl l2*) **then obtain** *l1* **where** $Inl\ l1 \in A1 \wedge \chi\ l1\ l2$
using *Rl a2* **unfolding** *rel-set-def* **by** *blast*
thus *?thesis* **unfolding** *Inl* **by** *auto*
next
case (*Inr r2*) **then obtain** *r1* **where** $Inr\ r1 \in A1 \wedge \varphi\ r1\ r2$
using *Rr a2* **unfolding** *rel-set-def* **by** *blast*
thus *?thesis* **unfolding** *Inr* **by** *auto*
qed
qed
qed

45.12 Quickcheck setup

Setup adapted from sets.

notation *Quickcheck-Exhaustive.orelse* (**infixr** *orelse 55*)

definition (**in** *term-syntax*) [*code-unfold*]:
valterm-femptyset = *Code-Evaluation.valtermify* ({||} :: ('*a* :: *typerep*) *fset*)

definition (**in** *term-syntax*) [*code-unfold*]:
valtermify-finsert x s = *Code-Evaluation.valtermify finsert* {·} (*x* :: ('*a* :: *typerep* ∗ -)) {·} *s*

instantiation *fset* :: (*exhaustive*) *exhaustive*
begin

fun *exhaustive-fset* **where**
exhaustive-fset f i = (*if i* = *0 then None else* (*f* {||} *orelse exhaustive-fset* (λ*A*. *f A orelse Quickcheck-Exhaustive.exhaustive* (λ*x*. *if x* |∈| *A then None else f* (*finsert x A*)) (*i* − *1*)) (*i* − *1*)))

instance ..

end

instantiation *fset* :: (*full-exhaustive*) *full-exhaustive*
begin

fun *full-exhaustive-fset* **where**
full-exhaustive-fset f i = (*if i* = *0 then None else* (*f valterm-femptyset orelse full-exhaustive-fset* (λ*A*. *f A orelse Quickcheck-Exhaustive.full-exhaustive* (λ*x*. *if fst x* |∈| *fst A then None else f* (*valtermify-finsert x A*)) (*i* − *1*)) (*i* − *1*)))

instance ..

end

no-notation *Quickcheck-Exhaustive.orelse* (**infixr** *orelse 55*)

notation *scomp* (**infixl** ∘→ *60*)

instantiation *fset* :: (*random*) *random*
begin

fun *random-aux-fset* :: *natural* ⇒ *natural* ⇒ *natural* × *natural* ⇒ (*'a fset* × (*unit* ⇒ *term*)) × *natural* × *natural* **where**
random-aux-fset 0 j = *Quickcheck-Random.collapse* (*Random.select-weight* [(*1*, *Pair valterm-femptyset*)]) |
random-aux-fset (*Code-Numeral.Suc i*) *j* =
  Quickcheck-Random.collapse (*Random.select-weight*
    [(*1*, *Pair valterm-femptyset*),
     (*Code-Numeral.Suc i*,
        Quickcheck-Random.random j ∘→ (λ*x*. *random-aux-fset i j* ∘→ (λ*s*. *Pair* (*valtermify-finsert x s*))))])

lemma [*code*]:
  random-aux-fset i j =
   Quickcheck-Random.collapse (*Random.select-weight* [(*1*, *Pair valterm-femptyset*),
     (*i*, *Quickcheck-Random.random j* ∘→ (λ*x*. *random-aux-fset* (*i* − *1*) *j* ∘→ (λ*s*. *Pair* (*valtermify-finsert x s*))))])
proof (*induct i rule*: *natural.induct*)
  case *zero*
 show *?case* **by** (*subst select-weight-drop-zero*[*symmetric*]) (*simp add*: *less-natural-def*)
next
  case (*Suc i*)
  show *?case* **by** (*simp only*: *random-aux-fset.simps Suc-natural-minus-one*)
qed

definition *random-fset i* = *random-aux-fset i i*

instance ..

end

no-notation *scomp* (**infixl** ∘→ *60*)

end

46 Pi and Function Sets

theory *FuncSet*
imports *Hilbert-Choice Main*
begin

definition *Pi* :: *'a set* ⇒ (*'a* ⇒ *'b set*) ⇒ (*'a* ⇒ *'b*) *set*
  where *Pi A B* = {*f*. ∀ *x*. *x* ∈ *A* ⟶ *f x* ∈ *B x*}

definition *extensional* :: *'a set* ⇒ (*'a* ⇒ *'b*) *set*

where *extensional* $A = \{f.\ \forall x.\ x \notin A \longrightarrow f\ x = undefined\}$

definition *restrict* :: $('a \Rightarrow 'b) \Rightarrow 'a\ set \Rightarrow 'a \Rightarrow 'b$
where *restrict* $f\ A = (\lambda x.\ if\ x \in A\ then\ f\ x\ else\ undefined)$

abbreviation *funcset* :: $'a\ set \Rightarrow 'b\ set \Rightarrow ('a \Rightarrow 'b)\ set$ (**infixr** $\rightarrow$ *60*)
where $A \rightarrow B \equiv Pi\ A\ (\lambda\text{-}.\ B)$

syntax (*ASCII*)
-Pi :: $pttrn \Rightarrow 'a\ set \Rightarrow 'b\ set \Rightarrow ('a \Rightarrow 'b)\ set$ ((*3PI* -:-./ -) *10*)
-lam :: $pttrn \Rightarrow 'a\ set \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a \Rightarrow 'b)$ ((*3%*-:-./ -) [*0*,*0*,*3*] *3*)
syntax
-Pi :: $pttrn \Rightarrow 'a\ set \Rightarrow 'b\ set \Rightarrow ('a \Rightarrow 'b)\ set$ ((3Π -$\in$-./ -) *10*)
-lam :: $pttrn \Rightarrow 'a\ set \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b)$ ((3λ-$\in$-./ -) [*0*,*0*,*3*] *3*)
translations
$\Pi\ x{\in}A.\ B \rightleftharpoons CONST\ Pi\ A\ (\lambda x.\ B)$
$\lambda x{\in}A.\ f \rightleftharpoons CONST\ restrict\ (\lambda x.\ f)\ A$

definition *compose* :: $'a\ set \Rightarrow ('b \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'c)$
where *compose* $A\ g\ f = (\lambda x{\in}A.\ g\ (f\ x))$

46.1 Basic Properties of *Pi*

lemma *Pi-I*[*intro*!]: $(\bigwedge x.\ x \in A \Longrightarrow f\ x \in B\ x) \Longrightarrow f \in Pi\ A\ B$
by (*simp add*: *Pi-def*)

lemma *Pi-I'*[*simp*]: $(\bigwedge x.\ x \in A \longrightarrow f\ x \in B\ x) \Longrightarrow f \in Pi\ A\ B$
by (*simp add*:*Pi-def*)

lemma *funcsetI*: $(\bigwedge x.\ x \in A \Longrightarrow f\ x \in B) \Longrightarrow f \in A \rightarrow B$
by (*simp add*: *Pi-def*)

lemma *Pi-mem*: $f \in Pi\ A\ B \Longrightarrow x \in A \Longrightarrow f\ x \in B\ x$
by (*simp add*: *Pi-def*)

lemma *Pi-iff*: $f \in Pi\ I\ X \longleftrightarrow (\forall i{\in}I.\ f\ i \in X\ i)$
unfolding *Pi-def* **by** *auto*

lemma *PiE* [*elim*]: $f \in Pi\ A\ B \Longrightarrow (f\ x \in B\ x \Longrightarrow Q) \Longrightarrow (x \notin A \Longrightarrow Q) \Longrightarrow Q$
by (*auto simp*: *Pi-def*)

lemma *Pi-cong*: $(\bigwedge w.\ w \in A \Longrightarrow f\ w = g\ w) \Longrightarrow f \in Pi\ A\ B \longleftrightarrow g \in Pi\ A\ B$
by (*auto simp*: *Pi-def*)

lemma *funcset-id* [*simp*]: $(\lambda x.\ x) \in A \rightarrow A$
by *auto*

lemma *funcset-mem*: $f \in A \rightarrow B \Longrightarrow x \in A \Longrightarrow f\ x \in B$
by (*simp add*: *Pi-def*)

lemma *funcset-image*: $f \in A \rightarrow B \Longrightarrow f \text{ ` } A \subseteq B$
  by *auto*

lemma *image-subset-iff-funcset*: $F \text{ ` } A \subseteq B \longleftrightarrow F \in A \rightarrow B$
  by *auto*

lemma *Pi-eq-empty*[*simp*]: $(\Pi\ x \in A.\ B\ x) = \{\} \longleftrightarrow (\exists\, x{\in}A.\ B\ x = \{\})$
  apply (*simp add*: *Pi-def*)
  apply *auto*

Converse direction requires Axiom of Choice to exhibit a function picking an element from each non-empty $B\ x$

  apply (*drule-tac* $x = \lambda u.\ SOME\ y.\ y \in B\ u$ **in** *spec*)
  apply *auto*
  apply (*cut-tac* $P = \lambda y.\ y \in B\ x$ **in** *some-eq-ex*)
  apply *auto*
  done

lemma *Pi-empty* [*simp*]: $Pi\ \{\}\ B = UNIV$
  by (*simp add*: *Pi-def*)

lemma *Pi-Int*: $Pi\ I\ E \cap Pi\ I\ F = (\Pi\ i{\in}I.\ E\ i \cap F\ i)$
  by *auto*

lemma *Pi-UN*:
  fixes $A :: nat \Rightarrow 'i \Rightarrow 'a\ set$
  assumes *finite I*
    and *mono*: $\bigwedge i\ n\ m.\ i \in I \Longrightarrow n \le m \Longrightarrow A\ n\ i \subseteq A\ m\ i$
  shows $(\bigcup n.\ Pi\ I\ (A\ n)) = (\Pi\ i{\in}I.\ \bigcup n.\ A\ n\ i)$
proof (*intro set-eqI iffI*)
  fix f
  assume $f \in (\Pi\ i{\in}I.\ \bigcup n.\ A\ n\ i)$
  then have $\forall\, i{\in}I.\ \exists\, n.\ f\ i \in A\ n\ i$
    by *auto*
  from *bchoice*[*OF this*] **obtain** n **where** n: $\bigwedge i.\ i \in I \Longrightarrow f\ i \in (A\ (n\ i)\ i)$
    by *auto*
  obtain k **where** k: $\bigwedge i.\ i \in I \Longrightarrow n\ i \le k$
    using ‹*finite I*› *finite-nat-set-iff-bounded-le*[*of n‘I*] **by** *auto*
  have $f \in Pi\ I\ (A\ k)$
  proof (*intro Pi-I*)
    fix i
    assume $i \in I$
    from *mono*[*OF this*, *of n i k*] *k*[*OF this*] *n*[*OF this*]
    show $f\ i \in A\ k\ i$ **by** *auto*
  qed
  then show $f \in (\bigcup n.\ Pi\ I\ (A\ n))$
    by *auto*
qed *auto*

lemma *Pi-UNIV* [*simp*]: $A \to UNIV = UNIV$
by (*simp add*: *Pi-def*)

Covariance of Pi-sets in their second argument

lemma *Pi-mono*: $(\bigwedge x.\ x \in A \Longrightarrow B\ x \subseteq C\ x) \Longrightarrow Pi\ A\ B \subseteq Pi\ A\ C$
by *auto*

Contravariance of Pi-sets in their first argument

lemma *Pi-anti-mono*: $A' \subseteq A \Longrightarrow Pi\ A\ B \subseteq Pi\ A'\ B$
by *auto*

lemma *prod-final*:
assumes *1*: $fst \circ f \in Pi\ A\ B$
and *2*: $snd \circ f \in Pi\ A\ C$
shows $f \in (\Pi\ z \in A.\ B\ z \times C\ z)$
proof (*rule Pi-I*)
fix *z*
assume *z*: $z \in A$
have $f\ z = (fst\ (f\ z),\ snd\ (f\ z))$
by *simp*
also have $\ldots \in B\ z \times C\ z$
by (*metis SigmaI PiE o-apply 1 2 z*)
finally show $f\ z \in B\ z \times C\ z$ **.**
qed

lemma *Pi-split-domain*[*simp*]: $x \in Pi\ (I \cup J)\ X \longleftrightarrow x \in Pi\ I\ X \wedge x \in Pi\ J\ X$
by (*auto simp*: *Pi-def*)

lemma *Pi-split-insert-domain*[*simp*]: $x \in Pi\ (insert\ i\ I)\ X \longleftrightarrow x \in Pi\ I\ X \wedge x\ i \in X\ i$
by (*auto simp*: *Pi-def*)

lemma *Pi-cancel-fupd-range*[*simp*]: $i \notin I \Longrightarrow x \in Pi\ I\ (B(i := b)) \longleftrightarrow x \in Pi\ I\ B$
by (*auto simp*: *Pi-def*)

lemma *Pi-cancel-fupd*[*simp*]: $i \notin I \Longrightarrow x(i := a) \in Pi\ I\ B \longleftrightarrow x \in Pi\ I\ B$
by (*auto simp*: *Pi-def*)

lemma *Pi-fupd-iff*: $i \in I \Longrightarrow f \in Pi\ I\ (B(i := A)) \longleftrightarrow f \in Pi\ (I - \{i\})\ B \wedge f\ i \in A$
apply *auto*
apply (*drule-tac x=x* **in** *Pi-mem*)
apply (*simp-all split*: *if-split-asm*)
apply (*drule-tac x=i* **in** *Pi-mem*)
apply (*auto dest*!: *Pi-mem*)
done

46.2 Composition With a Restricted Domain: *compose*

lemma *funcset-compose*: $f \in A \to B \Longrightarrow g \in B \to C \Longrightarrow compose\ A\ g\ f \in A \to C$
by (*simp add*: *Pi-def compose-def restrict-def*)

lemma *compose-assoc*:
assumes $f \in A \to B$
and $g \in B \to C$
and $h \in C \to D$
shows $compose\ A\ h\ (compose\ A\ g\ f) = compose\ A\ (compose\ B\ h\ g)\ f$
using *assms* **by** (*simp add*: *fun-eq-iff Pi-def compose-def restrict-def*)

lemma *compose-eq*: $x \in A \Longrightarrow compose\ A\ g\ f\ x = g\ (f\ x)$
by (*simp add*: *compose-def restrict-def*)

lemma *surj-compose*: $f\ `\ A = B \Longrightarrow g\ `\ B = C \Longrightarrow compose\ A\ g\ f\ `\ A = C$
by (*auto simp add*: *image-def compose-eq*)

46.3 Bounded Abstraction: *restrict*

lemma *restrict-cong*: $I = J \Longrightarrow (\bigwedge i.\ i \in J$ =simp=> $f\ i = g\ i) \Longrightarrow restrict\ f\ I = restrict\ g\ J$
by (*auto simp*: *restrict-def fun-eq-iff simp-implies-def*)

lemma *restrict-in-funcset*: $(\bigwedge x.\ x \in A \Longrightarrow f\ x \in B) \Longrightarrow (\lambda x{\in}A.\ f\ x) \in A \to B$
by (*simp add*: *Pi-def restrict-def*)

lemma *restrictI*[*intro*!]: $(\bigwedge x.\ x \in A \Longrightarrow f\ x \in B\ x) \Longrightarrow (\lambda x{\in}A.\ f\ x) \in Pi\ A\ B$
by (*simp add*: *Pi-def restrict-def*)

lemma *restrict-apply*[*simp*]: $(\lambda y{\in}A.\ f\ y)\ x = (if\ x \in A\ then\ f\ x\ else\ undefined)$
by (*simp add*: *restrict-def*)

lemma *restrict-apply'*: $x \in A \Longrightarrow (\lambda y{\in}A.\ f\ y)\ x = f\ x$
by *simp*

lemma *restrict-ext*: $(\bigwedge x.\ x \in A \Longrightarrow f\ x = g\ x) \Longrightarrow (\lambda x{\in}A.\ f\ x) = (\lambda x{\in}A.\ g\ x)$
by (*simp add*: *fun-eq-iff Pi-def restrict-def*)

lemma *restrict-UNIV*: $restrict\ f\ UNIV = f$
by (*simp add*: *restrict-def*)

lemma *inj-on-restrict-eq* [*simp*]: $inj\text{-}on\ (restrict\ f\ A)\ A = inj\text{-}on\ f\ A$
by (*simp add*: *inj-on-def restrict-def*)

lemma *Id-compose*: $f \in A \to B \Longrightarrow f \in extensional\ A \Longrightarrow compose\ A\ (\lambda y{\in}B.\ y)\ f = f$
by (*auto simp add*: *fun-eq-iff compose-def extensional-def Pi-def*)

lemma *compose-Id*: $g \in A \to B \Longrightarrow g \in extensional\ A \Longrightarrow compose\ A\ g\ (\lambda x{\in}A.\ x) = g$
  by (*auto simp add*: *fun-eq-iff compose-def extensional-def Pi-def*)

lemma *image-restrict-eq* [*simp*]: $(restrict\ f\ A)$ ` $A = f$ ` A
  by (*auto simp add*: *restrict-def*)

lemma *restrict-restrict*[*simp*]: $restrict\ (restrict\ f\ A)\ B = restrict\ f\ (A \cap B)$
  unfolding *restrict-def* **by** (*simp add*: *fun-eq-iff*)

lemma *restrict-fupd*[*simp*]: $i \notin I \Longrightarrow restrict\ (f\ (i := x))\ I = restrict\ f\ I$
  by (*auto simp*: *restrict-def*)

lemma *restrict-upd*[*simp*]: $i \notin I \Longrightarrow (restrict\ f\ I)(i := y) = restrict\ (f(i := y))\ (insert\ i\ I)$
  by (*auto simp*: *fun-eq-iff*)

lemma *restrict-Pi-cancel*: $restrict\ x\ I \in Pi\ I\ A \longleftrightarrow x \in Pi\ I\ A$
  by (*auto simp*: *restrict-def Pi-def*)

46.4 Bijections Between Sets

The definition of *bij-betw* is in *Fun.thy*, but most of the theorems belong here, or need at least *Hilbert-Choice*.

lemma *bij-betwI*:
  assumes $f \in A \to B$
    and $g \in B \to A$
    and *g-f*: $\bigwedge x.\ x{\in}A \Longrightarrow g\ (f\ x) = x$
    and *f-g*: $\bigwedge y.\ y{\in}B \Longrightarrow f\ (g\ y) = y$
  shows *bij-betw f A B*
  unfolding *bij-betw-def*
proof
  show *inj-on f A*
    by (*metis g-f inj-on-def*)
  have f ` $A \subseteq B$
    using ‹$f \in A \to B$› **by** *auto*
  moreover
  have $B \subseteq f$ ` A
    by *auto* (*metis Pi-mem* ‹$g \in B \to A$› *f-g image-iff*)
  ultimately show f ` $A = B$
    by *blast*
qed

lemma *bij-betw-imp-funcset*: $bij\text{-}betw\ f\ A\ B \Longrightarrow f \in A \to B$
  by (*auto simp add*: *bij-betw-def*)

lemma *inj-on-compose*: $bij\text{-}betw\ f\ A\ B \Longrightarrow inj\text{-}on\ g\ B \Longrightarrow inj\text{-}on\ (compose\ A\ g\ f)\ A$
  by (*auto simp add*: *bij-betw-def inj-on-def compose-eq*)

lemma *bij-betw-compose*: *bij-betw f A B* $\Longrightarrow$ *bij-betw g B C* $\Longrightarrow$ *bij-betw* (*compose A g f*) *A C*
apply (*simp add*: *bij-betw-def compose-eq inj-on-compose*)
apply (*auto simp add*: *compose-def image-def*)
done

lemma *bij-betw-restrict-eq* [*simp*]: *bij-betw* (*restrict f A*) *A B* = *bij-betw f A B*
by (*simp add*: *bij-betw-def*)

46.5 Extensionality

lemma *extensional-empty*[*simp*]: *extensional* {} = {λx. *undefined*}
unfolding *extensional-def* **by** *auto*

lemma *extensional-arb*: *f* $\in$ *extensional A* $\Longrightarrow$ *x* $\notin$ *A* $\Longrightarrow$ *f x* = *undefined*
by (*simp add*: *extensional-def*)

lemma *restrict-extensional* [*simp*]: *restrict f A* $\in$ *extensional A*
by (*simp add*: *restrict-def extensional-def*)

lemma *compose-extensional* [*simp*]: *compose A f g* $\in$ *extensional A*
by (*simp add*: *compose-def*)

lemma *extensionalityI*:
assumes *f* $\in$ *extensional A*
and *g* $\in$ *extensional A*
and $\bigwedge x$. *x* $\in$ *A* $\Longrightarrow$ *f x* = *g x*
shows *f* = *g*
using *assms* **by** (*force simp add*: *fun-eq-iff extensional-def*)

lemma *extensional-restrict*: *f* $\in$ *extensional A* $\Longrightarrow$ *restrict f A* = *f*
by (*rule extensionalityI*[*OF restrict-extensional*]) *auto*

lemma *extensional-subset*: *f* $\in$ *extensional A* $\Longrightarrow$ *A* $\subseteq$ *B* $\Longrightarrow$ *f* $\in$ *extensional B*
unfolding *extensional-def* **by** *auto*

lemma *inv-into-funcset*: *f* ' *A* = *B* $\Longrightarrow$ ($\lambda x \in B$. *inv-into A f x*) $\in$ *B* $\to$ *A*
by (*unfold inv-into-def*) (*fast intro*: *someI2*)

lemma *compose-inv-into-id*: *bij-betw f A B* $\Longrightarrow$ *compose A* ($\lambda y \in B$. *inv-into A f y*) *f* = ($\lambda x \in A$. *x*)
apply (*simp add*: *bij-betw-def compose-def*)
apply (*rule restrict-ext*, *auto*)
done

lemma *compose-id-inv-into*: *f* ' *A* = *B* $\Longrightarrow$ *compose B f* ($\lambda y \in B$. *inv-into A f y*) = ($\lambda x \in B$. *x*)
apply (*simp add*: *compose-def*)

apply (*rule restrict-ext*)
apply (*simp add*: *f-inv-into-f*)
done

lemma *extensional-insert*[*intro*, *simp*]:
assumes $a \in extensional\ (insert\ i\ I)$
shows $a(i := b) \in extensional\ (insert\ i\ I)$
using *assms* **unfolding** *extensional-def* **by** *auto*

lemma *extensional-Int*[*simp*]: $extensional\ I \cap extensional\ I' = extensional\ (I \cap I')$
unfolding *extensional-def* **by** *auto*

lemma *extensional-UNIV*[*simp*]: $extensional\ UNIV = UNIV$
by (*auto simp*: *extensional-def*)

lemma *restrict-extensional-sub*[*intro*]: $A \subseteq B \Longrightarrow restrict\ f\ A \in extensional\ B$
unfolding *restrict-def extensional-def* **by** *auto*

lemma *extensional-insert-undefined*[*intro*, *simp*]:
$a \in extensional\ (insert\ i\ I) \Longrightarrow a(i := undefined) \in extensional\ I$
unfolding *extensional-def* **by** *auto*

lemma *extensional-insert-cancel*[*intro*, *simp*]:
$a \in extensional\ I \Longrightarrow a \in extensional\ (insert\ i\ I)$
unfolding *extensional-def* **by** *auto*

46.6 Cardinality

lemma *card-inj*: $f \in A \to B \Longrightarrow inj\text{-}on\ f\ A \Longrightarrow finite\ B \Longrightarrow card\ A \le card\ B$
by (*rule card-inj-on-le*) *auto*

lemma *card-bij*:
assumes $f \in A \to B$ *inj-on f A*
and $g \in B \to A$ *inj-on g B*
and *finite A finite B*
shows $card\ A = card\ B$
using *assms* **by** (*blast intro*: *card-inj order-antisym*)

46.7 Extensional Function Spaces

definition $PiE :: 'a\ set \Rightarrow ('a \Rightarrow 'b\ set) \Rightarrow ('a \Rightarrow 'b)\ set$
where $PiE\ S\ T = Pi\ S\ T \cap extensional\ S$

abbreviation $Pi_E\ A\ B \equiv PiE\ A\ B$

syntax (*ASCII*)
$\text{-}PiE :: pttrn \Rightarrow 'a\ set \Rightarrow 'b\ set \Rightarrow ('a \Rightarrow 'b)\ set$ ((*3PIE -:-./ -*) *10*)
syntax
$\text{-}PiE :: pttrn \Rightarrow 'a\ set \Rightarrow 'b\ set \Rightarrow ('a \Rightarrow 'b)\ set$ ((*3*Π_E -∈-./ -) *10*)

translations
  Π_E *x*∈*A*. *B* ⇌ *CONST Pi*$_E$ *A* (λ*x*. *B*)

abbreviation *extensional-funcset* :: *'a set* ⇒ *'b set* ⇒ (*'a* ⇒ *'b*) *set* (**infixr** $\rightarrow_E$ *60*)
  where *A* $\rightarrow_E$ *B* ≡ (Π_E *i*∈*A*. *B*)

lemma *extensional-funcset-def*: *extensional-funcset S T* = (*S* → *T*) ∩ *extensional S*
  by (*simp add*: *PiE-def*)

lemma *PiE-empty-domain*[*simp*]: *PiE* {} *T* = {λ*x*. *undefined*}
  unfolding *PiE-def* **by** *simp*

lemma *PiE-UNIV-domain*: *PiE UNIV T* = *Pi UNIV T*
  unfolding *PiE-def* **by** *simp*

lemma *PiE-empty-range*[*simp*]: *i* ∈ *I* ⟹ *F i* = {} ⟹ (Π_E *i*∈*I*. *F i*) = {}
  unfolding *PiE-def* **by** *auto*

lemma *PiE-eq-empty-iff*: *Pi*$_E$ *I F* = {} ⟷ (∃ *i*∈*I*. *F i* = {})
proof
  assume *Pi*$_E$ *I F* = {}
  show ∃ *i*∈*I*. *F i* = {}
  proof (*rule ccontr*)
    assume ¬ *?thesis*
    then have ∀ *i*. ∃ *y*. (*i* ∈ *I* ⟶ *y* ∈ *F i*) ∧ (*i* ∉ *I* ⟶ *y* = *undefined*)
      by *auto*
    from *choice*[*OF this*]
    obtain *f* **where** ∀ *x*. (*x* ∈ *I* ⟶ *f x* ∈ *F x*) ∧ (*x* ∉ *I* ⟶ *f x* = *undefined*) **..**
    then have *f* ∈ *Pi*$_E$ *I F*
      by (*auto simp*: *extensional-def PiE-def*)
    with ‹*Pi*$_E$ *I F* = {}› **show** *False*
      by *auto*
  qed
qed (*auto simp*: *PiE-def*)

lemma *PiE-arb*: *f* ∈ *PiE S T* ⟹ *x* ∉ *S* ⟹ *f x* = *undefined*
  unfolding *PiE-def* **by** *auto* (*auto dest*!: *extensional-arb*)

lemma *PiE-mem*: *f* ∈ *PiE S T* ⟹ *x* ∈ *S* ⟹ *f x* ∈ *T x*
  unfolding *PiE-def* **by** *auto*

lemma *PiE-fun-upd*: *y* ∈ *T x* ⟹ *f* ∈ *PiE S T* ⟹ *f*(*x* := *y*) ∈ *PiE* (*insert x S*) *T*
  unfolding *PiE-def extensional-def* **by** *auto*

lemma *fun-upd-in-PiE*: *x* ∉ *S* ⟹ *f* ∈ *PiE* (*insert x S*) *T* ⟹ *f*(*x* := *undefined*) ∈ *PiE S T*

unfolding *PiE-def extensional-def* **by** *auto*

lemma *PiE-insert-eq*: *PiE* (*insert x S*) *T* = (λ(*y*, *g*). *g*(*x* := *y*)) ‘ (*T x* × *PiE S T*)
proof −
{
fix *f* **assume** *f* ∈ *PiE* (*insert x S*) *T x* ∉ *S*
with *assms* **have** *f* ∈ (λ(*y*, *g*). *g*(*x* := *y*)) ‘ (*T x* × *PiE S T*)
by (*auto intro*!: *image-eqI*[**where** *x*=(*f x*, *f*(*x* := *undefined*))] *intro*: *fun-upd-in-PiE PiE-mem*)
}
moreover
{
fix *f* **assume** *f* ∈ *PiE* (*insert x S*) *T x* ∈ *S*
with *assms* **have** *f* ∈ (λ(*y*, *g*). *g*(*x* := *y*)) ‘ (*T x* × *PiE S T*)
by (*auto intro*!: *image-eqI*[**where** *x*=(*f x*, *f*)] *intro*: *fun-upd-in-PiE PiE-mem simp*: *insert-absorb*)
}
ultimately show *?thesis*
using *assms* **by** (*auto intro*: *PiE-fun-upd*)
qed

lemma *PiE-Int*: Pi_E *I A* ∩ Pi_E *I B* = Pi_E *I* (λ*x*. *A x* ∩ *B x*)
by (*auto simp*: *PiE-def*)

lemma *PiE-cong*: (⋀*i*. *i*∈*I* ⟹ *A i* = *B i*) ⟹ Pi_E *I A* = Pi_E *I B*
unfolding *PiE-def* **by** (*auto simp*: *Pi-cong*)

lemma *PiE-E* [*elim*]:
assumes *f* ∈ *PiE A B*
obtains *x* ∈ *A* **and** *f x* ∈ *B x*
| *x* ∉ *A* **and** *f x* = *undefined*
using *assms* **by** (*auto simp*: *Pi-def PiE-def extensional-def*)

lemma *PiE-I*[*intro*!]:
(⋀*x*. *x* ∈ *A* ⟹ *f x* ∈ *B x*) ⟹ (⋀*x*. *x* ∉ *A* ⟹ *f x* = *undefined*) ⟹ *f* ∈ *PiE A B*
by (*simp add*: *PiE-def extensional-def*)

lemma *PiE-mono*: (⋀*x*. *x* ∈ *A* ⟹ *B x* ⊆ *C x*) ⟹ *PiE A B* ⊆ *PiE A C*
by *auto*

lemma *PiE-iff*: *f* ∈ *PiE I X* ⟷ (∀ *i*∈*I*. *f i* ∈ *X i*) ∧ *f* ∈ *extensional I*
by (*simp add*: *PiE-def Pi-iff*)

lemma *PiE-restrict*[*simp*]: *f* ∈ *PiE A B* ⟹ *restrict f A* = *f*
by (*simp add*: *extensional-restrict PiE-def*)

lemma *restrict-PiE*[*simp*]: *restrict f I* ∈ *PiE I S* ⟷ *f* ∈ *Pi I S*

by (*auto simp*: *PiE-iff*)

lemma *PiE-eq-subset*:
assumes *ne*: $\bigwedge i.\ i \in I \Longrightarrow F\ i \neq \{\}$ $\bigwedge i.\ i \in I \Longrightarrow F'\ i \neq \{\}$
and *eq*: $Pi_E\ I\ F = Pi_E\ I\ F'$
and $i \in I$
shows $F\ i \subseteq F'\ i$
proof
fix x
assume $x \in F\ i$
with *ne* **have** $\forall j.\ \exists y.\ (j \in I \longrightarrow y \in F\ j \land (i = j \longrightarrow x = y)) \land (j \notin I \longrightarrow y = undefined)$
by *auto*
from *choice*[*OF this*] **obtain** f
where f: $\forall j.\ (j \in I \longrightarrow f\ j \in F\ j \land (i = j \longrightarrow x = f\ j)) \land (j \notin I \longrightarrow f\ j = undefined)$ **..**
then have $f \in Pi_E\ I\ F$
by (*auto simp*: *extensional-def PiE-def*)
then have $f \in Pi_E\ I\ F'$
using *assms* **by** *simp*
then show $x \in F'\ i$
using f ‹$i \in I$› **by** (*auto simp*: *PiE-def*)
qed

lemma *PiE-eq-iff-not-empty*:
assumes *ne*: $\bigwedge i.\ i \in I \Longrightarrow F\ i \neq \{\}$ $\bigwedge i.\ i \in I \Longrightarrow F'\ i \neq \{\}$
shows $Pi_E\ I\ F = Pi_E\ I\ F' \longleftrightarrow (\forall i{\in}I.\ F\ i = F'\ i)$
proof (*intro iffI ballI*)
fix i
assume *eq*: $Pi_E\ I\ F = Pi_E\ I\ F'$
assume i: $i \in I$
show $F\ i = F'\ i$
using *PiE-eq-subset*[*of I F F'*, *OF ne eq i*]
using *PiE-eq-subset*[*of I F' F*, *OF ne*(*2*,*1*) *eq*[*symmetric*] *i*]
by *auto*
qed (*auto simp*: *PiE-def*)

lemma *PiE-eq-iff*:
$Pi_E\ I\ F = Pi_E\ I\ F' \longleftrightarrow (\forall i{\in}I.\ F\ i = F'\ i) \lor ((\exists i{\in}I.\ F\ i = \{\}) \land (\exists i{\in}I.\ F'\ i = \{\}))$
proof (*intro iffI disjCI*)
assume *eq*[*simp*]: $Pi_E\ I\ F = Pi_E\ I\ F'$
assume $\neg\ ((\exists i{\in}I.\ F\ i = \{\}) \land (\exists i{\in}I.\ F'\ i = \{\}))$
then have $(\forall i{\in}I.\ F\ i \neq \{\}) \land (\forall i{\in}I.\ F'\ i \neq \{\})$
using *PiE-eq-empty-iff*[*of I F*] *PiE-eq-empty-iff*[*of I F'*] **by** *auto*
with *PiE-eq-iff-not-empty*[*of I F F'*] **show** $\forall i{\in}I.\ F\ i = F'\ i$
by *auto*
next
assume $(\forall i{\in}I.\ F\ i = F'\ i) \lor (\exists i{\in}I.\ F\ i = \{\}) \land (\exists i{\in}I.\ F'\ i = \{\})$

then show $Pi_E\ I\ F = Pi_E\ I\ F'$
using *PiE-eq-empty-iff* [*of I F*] *PiE-eq-empty-iff* [*of I F'*] **by** (*auto simp*: *PiE-def*)
qed

lemma *extensional-funcset-fun-upd-restricts-rangeI*:
$\forall y \in S.\ f\ x \neq f\ y \Longrightarrow f \in (insert\ x\ S) \rightarrow_E T \Longrightarrow f(x := undefined) \in S \rightarrow_E (T - \{f\ x\})$
unfolding *extensional-funcset-def extensional-def*
apply *auto*
apply (*case-tac* $x = xa$)
apply *auto*
done

lemma *extensional-funcset-fun-upd-extends-rangeI*:
assumes $a \in T\ f \in S \rightarrow_E (T - \{a\})$
shows $f(x := a) \in insert\ x\ S \rightarrow_E\ T$
using *assms* **unfolding** *extensional-funcset-def extensional-def* **by** *auto*

46.7.1 Injective Extensional Function Spaces

lemma *extensional-funcset-fun-upd-inj-onI*:
assumes $f \in S \rightarrow_E (T - \{a\})$
and *inj-on f S*
shows *inj-on* $(f(x := a))\ S$
using *assms*
unfolding *extensional-funcset-def* **by** (*auto intro*!: *inj-on-fun-updI*)

lemma *extensional-funcset-extend-domain-inj-on-eq*:
assumes $x \notin S$
shows $\{f.\ f \in (insert\ x\ S) \rightarrow_E T \wedge inj\text{-}on\ f\ (insert\ x\ S)\} =$
$(\lambda(y, g).\ g(x{:=}y))$ ‘ $\{(y, g).\ y \in T \wedge g \in S \rightarrow_E (T - \{y\}) \wedge inj\text{-}on\ g\ S\}$
using *assms*
apply (*auto del*: *PiE-I PiE-E*)
apply (*auto intro*: *extensional-funcset-fun-upd-inj-onI*
extensional-funcset-fun-upd-extends-rangeI del: *PiE-I PiE-E*)
apply (*auto simp add*: *image-iff inj-on-def*)
apply (*rule-tac x=xa x* **in** *exI*)
apply (*auto intro*: *PiE-mem del*: *PiE-I PiE-E*)
apply (*rule-tac* $x{=}xa(x := undefined)$ **in** *exI*)
apply (*auto intro*!: *extensional-funcset-fun-upd-restricts-rangeI*)
apply (*auto dest*!: *PiE-mem split*: *if-split-asm*)
done

lemma *extensional-funcset-extend-domain-inj-onI*:
assumes $x \notin S$
shows *inj-on* $(\lambda(y, g).\ g(x := y))\ \{(y, g).\ y \in T \wedge g \in S \rightarrow_E (T - \{y\}) \wedge inj\text{-}on\ g\ S\}$
using *assms*
apply (*auto intro*!: *inj-onI*)

apply (*metis fun-upd-same*)
apply (*metis assms PiE-arb fun-upd-triv fun-upd-upd*)
done

46.7.2 Cardinality

lemma *finite-PiE*: *finite S* $\Longrightarrow$ ($\bigwedge i$. $i \in S \Longrightarrow$ *finite* ($T\ i$)) $\Longrightarrow$ *finite* ($\Pi_E\ i \in S.\ T\ i$)
by (*induct S arbitrary*: *T rule*: *finite-induct*) (*simp-all add*: *PiE-insert-eq*)

lemma *inj-combinator*: $x \notin S \Longrightarrow$ *inj-on* ($\lambda(y, g).\ g(x := y)$) ($T\ x \times Pi_E\ S\ T$)
proof (*safe intro*!: *inj-onI ext*)
fix *f y g z*
assume $x \notin S$
assume *fg*: $f \in Pi_E\ S\ T\ g \in Pi_E\ S\ T$
assume $f(x := y) = g(x := z)$
then have ∗: $\bigwedge i.\ (f(x := y))\ i = (g(x := z))\ i$
unfolding *fun-eq-iff* **by** *auto*
from *this*[*of x*] **show** $y = z$ **by** *simp*
fix *i* **from** ∗[*of i*] ‹$x \notin S$› *fg* **show** $f\ i = g\ i$
by (*auto split*: *if-split-asm simp*: *PiE-def extensional-def*)
qed

lemma *card-PiE*: *finite S* $\Longrightarrow$ *card* ($\Pi_E\ i \in S.\ T\ i$) = ($\prod$ $i{\in}S$. *card* ($T\ i$))
proof (*induct rule*: *finite-induct*)
case *empty*
then show *?case* **by** *auto*
next
case (*insert x S*)
then show *?case*
by (*simp add*: *PiE-insert-eq inj-combinator card-image card-cartesian-product*)
qed

end

47 Pointwise instantiation of functions to division

theory *Function-Division*
imports *Function-Algebras*
begin

47.1 Syntactic with division

instantiation *fun* :: (*type*, *inverse*) *inverse*
begin

definition *inverse f* = *inverse* ∘ *f*

definition *f div g* = ($\lambda x.\ f\ x\ /\ g\ x$)

instance ..

end

lemma *inverse-fun-apply* [*simp*]:
inverse f x = *inverse* (*f x*)
by (*simp add*: *inverse-fun-def*)

lemma *divide-fun-apply* [*simp*]:
(*f* / *g*) *x* = *f x* / *g x*
by (*simp add*: *divide-fun-def*)

Unfortunately, we cannot lift this operations to algebraic type classes for division: being different from the constant zero function $f \neq (0{::}'a)$ is too weak as precondition. So we must introduce our own set of lemmas.

abbreviation *zero-free* :: ('*b* ⇒ '*a*::*field*) ⇒ *bool* **where**
zero-free f ≡ ¬ (∃ *x*. *f x* = *0*)

lemma *fun-left-inverse*:
fixes *f* :: '*b* ⇒ '*a*::*field*
shows *zero-free f* ⟹ *inverse f* ∗ *f* = *1*
by (*simp add*: *fun-eq-iff*)

lemma *fun-right-inverse*:
fixes *f* :: '*b* ⇒ '*a*::*field*
shows *zero-free f* ⟹ *f* ∗ *inverse f* = *1*
by (*simp add*: *fun-eq-iff*)

lemma *fun-divide-inverse*:
fixes *f g* :: '*b* ⇒ '*a*::*field*
shows *f* / *g* = *f* ∗ *inverse g*
by (*simp add*: *fun-eq-iff divide-inverse*)

Feel free to extend this.

Another possibility would be a reformulation of the division type classes to user a *zero-free* predicate rather than a direct $a \neq (0{::}'a)$ condition.

end

48 Preorders with explicit equivalence relation

theory *Preorder*
imports *Orderings*
begin

class *preorder-equiv* = *preorder*
begin

definition *equiv* :: $'a \Rightarrow 'a \Rightarrow bool$ **where**
equiv $x\ y \longleftrightarrow x \le y \land y \le x$

notation
equiv (*op* ≈) **and**
equiv ((-/ ≈ -) [*51*, *51*] *50*)

lemma *refl* [*iff*]:
$x \approx x$
unfolding *equiv-def* **by** *simp*

lemma *trans*:
$x \approx y \Longrightarrow y \approx z \Longrightarrow x \approx z$
unfolding *equiv-def* **by** (*auto intro*: *order-trans*)

lemma *antisym*:
$x \le y \Longrightarrow y \le x \Longrightarrow x \approx y$
unfolding *equiv-def* **..**

lemma *less-le*: $x < y \longleftrightarrow x \le y \land \neg\ x \approx y$
by (*auto simp add*: *equiv-def less-le-not-le*)

lemma *le-less*: $x \le y \longleftrightarrow x < y \lor x \approx y$
by (*auto simp add*: *equiv-def less-le*)

lemma *le-imp-less-or-eq*: $x \le y \Longrightarrow x < y \lor x \approx y$
by (*simp add*: *less-le*)

lemma *less-imp-not-eq*: $x < y \Longrightarrow x \approx y \longleftrightarrow False$
by (*simp add*: *less-le*)

lemma *less-imp-not-eq2*: $x < y \Longrightarrow y \approx x \longleftrightarrow False$
by (*simp add*: *equiv-def less-le*)

lemma *neq-le-trans*: $\neg\ a \approx b \Longrightarrow a \le b \Longrightarrow a < b$
by (*simp add*: *less-le*)

lemma *le-neq-trans*: $a \le b \Longrightarrow \neg\ a \approx b \Longrightarrow a < b$
by (*simp add*: *less-le*)

lemma *antisym-conv*: $y \le x \Longrightarrow x \le y \longleftrightarrow x \approx y$
by (*simp add*: *equiv-def*)

end

end

49 Common discrete functions

theory *Discrete*
imports *Main*
begin

49.1 Discrete logarithm

context
begin

qualified fun *log* :: *nat* ⇒ *nat*
  where [*simp del*]: *log n* = (*if n* < *2 then 0 else Suc* (*log* (*n div 2*)))

lemma *log-induct* [*consumes 1*, *case-names one double*]:
  fixes *n* :: *nat*
  assumes *n* > *0*
  assumes *one*: *P 1*
  assumes *double*: ⋀*n*. *n* ≥ *2* ⟹ *P* (*n div 2*) ⟹ *P n*
  shows *P n*
using ‹*n* > *0*› **proof** (*induct n rule*: *log.induct*)
  fix *n*
  assume ¬ *n* < *2* ⟹
        0 < *n div 2* ⟹ *P* (*n div 2*)
  then have ∗: *n* ≥ *2* ⟹ *P* (*n div 2*) **by** *simp*
  assume *n* > *0*
  show *P n*
  proof (*cases n* = *1*)
    case *True* **with** *one* **show** *?thesis* **by** *simp*
  next
    case *False* **with** ‹*n* > *0*› **have** *n* ≥ *2* **by** *auto*
    moreover with ∗ **have** *P* (*n div 2*) **.**
    ultimately show *?thesis* **by** (*rule double*)
  qed
qed

lemma *log-zero* [*simp*]: *log 0* = *0*
  by (*simp add*: *log.simps*)

lemma *log-one* [*simp*]: *log 1* = *0*
  by (*simp add*: *log.simps*)

lemma *log-Suc-zero* [*simp*]: *log* (*Suc 0*) = *0*
  using *log-one* **by** *simp*

lemma *log-rec*: *n* ≥ *2* ⟹ *log n* = *Suc* (*log* (*n div 2*))
  by (*simp add*: *log.simps*)

lemma *log-twice* [*simp*]: *n* ≠ *0* ⟹ *log* (*2* ∗ *n*) = *Suc* (*log n*)
  by (*simp add*: *log-rec*)

lemma *log-half* [*simp*]: *log* (*n div 2*) = *log n* − *1*
proof (*cases n* < *2*)
  case *True*
  then have *n* = *0* ∨ *n* = *1* **by** *arith*
  then show *?thesis* **by** (*auto simp del*: *One-nat-def*)
next
  case *False*
  then show *?thesis* **by** (*simp add*: *log-rec*)
qed

lemma *log-exp* [*simp*]: *log* (*2* ˆ *n*) = *n*
  by (*induct n*) *simp-all*

lemma *log-mono*: *mono log*
proof
  fix *m n* :: *nat*
  assume *m* ≤ *n*
  then show *log m* ≤ *log n*
  proof (*induct m arbitrary*: *n rule*: *log.induct*)
    case (*1 m*)
    then have *mn2*: *m div 2* ≤ *n div 2* **by** *arith*
    show *log m* ≤ *log n*
    proof (*cases m* ≥ *2*)
      case *False*
      then have *m* = *0* ∨ *m* = *1* **by** *arith*
      then show *?thesis* **by** (*auto simp del*: *One-nat-def*)
    next
      case *True* **then have** ¬ *m* < *2* **by** *simp*
      with *mn2* **have** *n* ≥ *2* **by** *arith*
      from *True* **have** *m2-0*: *m div 2* ≠ *0* **by** *arith*
      with *mn2* **have** *n2-0*: *n div 2* ≠ *0* **by** *arith*
      from ‹¬ *m* < *2*› *1.hyps mn2* **have** *log* (*m div 2*) ≤ *log* (*n div 2*) **by** *blast*
      with *m2-0 n2-0* **have** *log* (*2* ∗ (*m div 2*)) ≤ *log* (*2* ∗ (*n div 2*)) **by** *simp*
      with *m2-0 n2-0* ‹*m* ≥ *2*› ‹*n* ≥ *2*› **show** *?thesis* **by** (*simp only*: *log-rec* [*of m*] *log-rec* [*of n*]) *simp*
    qed
  qed
qed

lemma *log-exp2-le*:
  assumes *n* > *0*
  shows *2* ˆ *log n* ≤ *n*
using *assms* **proof** (*induct n rule*: *log-induct*)
  show *2* ˆ *log 1* ≤ (*1* :: *nat*) **by** *simp*
next
  fix *n* :: *nat*
  assume *n* ≥ *2*
  with *log-mono* **have** *log n* ≥ *Suc 0*

by (*simp add*: *log.simps*)
assume $2 \hat{} \ log\ (n\ div\ 2) \leq n\ div\ 2$
with ‹$n \geq 2$› **have** $2 \hat{} \ (log\ n - Suc\ 0) \leq n\ div\ 2$ **by** *simp*
then have $2 \hat{} \ (log\ n - Suc\ 0) * 2 \hat{} \ 1 \leq n\ div\ 2 * 2$ **by** *simp*
with ‹$log\ n \geq Suc\ 0$› **have** $2 \hat{} \ log\ n \leq n\ div\ 2 * 2$
unfolding *power-add* [*symmetric*] **by** *simp*
also have $n\ div\ 2 * 2 \leq n$ **by** (*cases even n*) *simp-all*
finally show $2 \hat{} \ log\ n \leq n$ **.**
qed

49.2 Discrete square root

qualified definition *sqrt* :: *nat* ⇒ *nat*
where $sqrt\ n = Max\ \{m.\ m^2 \leq n\}$

lemma *sqrt-aux*:
fixes *n* :: *nat*
shows *finite* $\{m.\ m^2 \leq n\}$ **and** $\{m.\ m^2 \leq n\} \neq \{\}$
proof –
{ fix *m*
assume $m^2 \leq n$
then have $m \leq n$
by (*cases m*) (*simp-all add*: *power2-eq-square*)
} note ∗∗ = *this*
then have $\{m.\ m^2 \leq n\} \subseteq \{m.\ m \leq n\}$ **by** *auto*
then show *finite* $\{m.\ m^2 \leq n\}$ **by** (*rule finite-subset*) *rule*
have $0^2 \leq n$ **by** *simp*
then show ∗: $\{m.\ m^2 \leq n\} \neq \{\}$ **by** *blast*
qed

lemma [*code*]: $sqrt\ n = Max\ (Set.filter\ (\lambda m.\ m^2 \leq n)\ \{0..n\})$
proof –
from *power2-nat-le-imp-le* [*of - n*] **have** $\{m.\ m \leq n \wedge m^2 \leq n\} = \{m.\ m^2 \leq n\}$ **by** *auto*
then show *?thesis* **by** (*simp add*: *sqrt-def Set.filter-def*)
qed

lemma *sqrt-inverse-power2* [*simp*]: $sqrt\ (n^2) = n$
proof –
have $\{m.\ m \leq n\} \neq \{\}$ **by** *auto*
then have $Max\ \{m.\ m \leq n\} \leq n$ **by** *auto*
then show *?thesis*
by (*auto simp add*: *sqrt-def power2-nat-le-eq-le intro*: *antisym*)
qed

lemma *sqrt-zero* [*simp*]: *sqrt 0 = 0*
using *sqrt-inverse-power2* [*of 0*] **by** *simp*

lemma *sqrt-one* [*simp*]: *sqrt 1 = 1*

using *sqrt-inverse-power2* [*of 1*] **by** *simp*

lemma *mono-sqrt*: *mono sqrt*
proof
fix *m n* :: *nat*
have ∗: $0 * 0 \le m$ **by** *simp*
assume $m \le n$
then show *sqrt m* ≤ *sqrt n*
by (*auto intro*!: *Max-mono* ‹$0 * 0 \le m$› *finite-less-ub simp add*: *power2-eq-square sqrt-def*)
qed

lemma *sqrt-greater-zero-iff* [*simp*]: *sqrt n* > *0* ⟷ *n* > *0*
proof −
have ∗: $0 < Max\ \{m.\ m^2 \le n\} \longleftrightarrow (\exists\, a \in \{m.\ m^2 \le n\}.\ 0 < a)$
by (*rule Max-gr-iff*) (*fact sqrt-aux*)+
show *?thesis*
proof
assume *0* < *sqrt n*
then have $0 < Max\ \{m.\ m^2 \le n\}$ **by** (*simp add*: *sqrt-def*)
with ∗ **show** *0* < *n* **by** (*auto dest*: *power2-nat-le-imp-le*)
next
assume *0* < *n*
then have $1^2 \le n \wedge 0 < (1::nat)$ **by** *simp*
then have $\exists\, q.\ q^2 \le n \wedge 0 < q$ **..**
with ∗ **have** $0 < Max\ \{m.\ m^2 \le n\}$ **by** *blast*
then show *0* < *sqrt n* **by** (*simp add*: *sqrt-def*)
qed
qed

lemma *sqrt-power2-le* [*simp*]: $(sqrt\ n)^2 \le n$
proof (*cases n* > *0*)
case *False* **then show** *?thesis* **by** *simp*
next
case *True* **then have** *sqrt n* > *0* **by** *simp*
then have *mono* (*times* (*Max* $\{m.\ m^2 \le n\}$)) **by** (*auto intro*: *mono-times-nat simp add*: *sqrt-def*)
then have ∗: $Max\ \{m.\ m^2 \le n\} * Max\ \{m.\ m^2 \le n\} = Max\ (times\ (Max\ \{m.\ m^2 \le n\})\ `\ \{m.\ m^2 \le n\})$
using *sqrt-aux* [*of n*] **by** (*rule mono-Max-commute*)
have $Max\ (op * (Max\ \{m.\ m * m \le n\})\ `\ \{m.\ m * m \le n\}) \le n$
apply (*subst Max-le-iff*)
apply (*metis* (*mono-tags*) *finite-imageI finite-less-ub le-square*)
apply *simp*
apply (*metis le0 mult-0-right*)
apply *auto*
proof −
fix *q*
assume $q * q \le n$

```
    show Max {m. m * m ≤ n} * q ≤ n
    proof (cases q > 0)
      case False then show ?thesis by simp
    next
      case True then have mono (times q) by (rule mono-times-nat)
      then have q * Max {m. m * m ≤ n} = Max (times q ‘ {m. m * m ≤ n})
      using sqrt-aux [of n] by (auto simp add: power2-eq-square intro: mono-Max-commute)
      then have Max {m. m * m ≤ n} * q = Max (times q ‘ {m. m * m ≤ n})
by (simp add: ac-simps)
      then show ?thesis
        apply simp
        apply (subst Max-le-iff)
        apply auto
        apply (metis (mono-tags) finite-imageI finite-less-ub le-square)
        apply (metis ‹q * q ≤ n›)
      apply (metis ‹q * q ≤ n› le-cases mult-le-mono1 mult-le-mono2 order-trans)
        done
    qed
  qed
  with * show ?thesis by (simp add: sqrt-def power2-eq-square)
qed

lemma sqrt-le: sqrt n ≤ n
  using sqrt-aux [of n] by (auto simp add: sqrt-def intro: power2-nat-le-imp-le)

end

end
```

50 Comparing growth of functions on natural numbers by a preorder relation

```
theory Function-Growth
imports Main Preorder Discrete
begin

context linorder
begin

lemma mono-invE:
  fixes f :: 'a ⇒ 'b::order
  assumes mono f
  assumes f x < f y
  obtains x < y
proof
  show x < y
```

```
  proof (rule ccontr)
    assume ¬ x < y
    then have y ≤ x by simp
    with ‹mono f› obtain f y ≤ f x by (rule monoE)
    with ‹f x < f y› show False by simp
  qed
qed

end

lemma (in semidom-divide) power-diff:
  fixes a :: 'a
  assumes a ≠ 0
  assumes m ≥ n
  shows a ^ (m − n) = (a ^ m) div (a ^ n)
proof −
  def q == m − n
  moreover with assms have m = q + n by (simp add: q-def)
  ultimately show ?thesis using ‹a ≠ 0› by (simp add: power-add)
qed
```

50.1 Motivation

When comparing growth of functions in computer science, it is common to adhere on Landau Symbols ("O-Notation"). However these come at the cost of notational oddities, particularly writing $f = O(g)$ for $f \in O(g)$ etc.

Here we suggest a different way, following Hardy (G. H. Hardy and J. E. Littlewood, Some problems of Diophantine approximation, Acta Mathematica 37 (1914), p. 225). We establish a quasi order relation $\lesssim$ on functions such that $f \lesssim g \longleftrightarrow f \in O(g)$. From a didactic point of view, this does not only avoid the notational oddities mentioned above but also emphasizes the key insight of a growth hierarchy of functions: $(\lambda n.\ 0) \lesssim (\lambda n.\ k) \lesssim \mathit{Discrete.log} \lesssim \mathit{Discrete.sqrt} \lesssim \mathit{id} \lesssim \ldots$.

50.2 Model

Our growth functions are of type $\mathbb{N} \Rightarrow \mathbb{N}$. This is different to the usual conventions for Landau symbols for which $\mathbb{R} \Rightarrow \mathbb{R}$ would be appropriate, but we argue that $\mathbb{R} \Rightarrow \mathbb{R}$ is more appropriate for analysis, whereas our setting is discrete.

Note that we also restrict the additional coefficients to $\mathbb{N}$, something we discuss at the particular definitions.

50.3 The $\lesssim$ relation

```
definition less-eq-fun :: (nat ⇒ nat) ⇒ (nat ⇒ nat) ⇒ bool (infix ≲ 50)
where
```

$f \lesssim g \longleftrightarrow (\exists\, c{>}0.\ \exists\, n.\ \forall\, m{>}n.\ f\ m \le c * g\ m)$

This yields $f \lesssim g \longleftrightarrow f \in O(g)$. Note that c is restricted to $\mathbb{N}$. This does not pose any problems since if $f \in O(g)$ holds for a $c \in \mathbb{R}$, it also holds for $\lceil c \rceil \in \mathbb{N}$ by transitivity.

lemma *less-eq-funI* [*intro?*]:
assumes $\exists\, c{>}0.\ \exists\, n.\ \forall\, m{>}n.\ f\ m \le c * g\ m$
shows $f \lesssim g$
unfolding *less-eq-fun-def* **by** (*rule assms*)

lemma *not-less-eq-funI*:
assumes $\bigwedge c\ n.\ c > 0 \Longrightarrow \exists\, m{>}n.\ c * g\ m < f\ m$
shows $\neg\ f \lesssim g$
using *assms* **unfolding** *less-eq-fun-def linorder-not-le* [*symmetric*] **by** *blast*

lemma *less-eq-funE* [*elim?*]:
assumes $f \lesssim g$
obtains $n\ c$ **where** $c > 0$ **and** $\bigwedge m.\ m > n \Longrightarrow f\ m \le c * g\ m$
using *assms* **unfolding** *less-eq-fun-def* **by** *blast*

lemma *not-less-eq-funE*:
assumes $\neg\ f \lesssim g$ **and** $c > 0$
obtains m **where** $m > n$ **and** $c * g\ m < f\ m$
using *assms* **unfolding** *less-eq-fun-def linorder-not-le* [*symmetric*] **by** *blast*

50.4 The ≈ relation, the equivalence relation induced by $\lesssim$

definition *equiv-fun* :: $(nat \Rightarrow nat) \Rightarrow (nat \Rightarrow nat) \Rightarrow bool$ (**infix** $\cong$ *50*)
where
$f \cong g \longleftrightarrow$
$(\exists\, c_1{>}0.\ \exists\, c_2{>}0.\ \exists\, n.\ \forall\, m{>}n.\ f\ m \le c_1 * g\ m \wedge g\ m \le c_2 * f\ m)$

This yields $f \cong g \longleftrightarrow f \in \Theta(g)$. Concerning c_1 and c_2 restricted to *nat*, see note above on $(\lesssim)$.

lemma *equiv-funI*:
assumes $\exists\, c_1{>}0.\ \exists\, c_2{>}0.\ \exists\, n.\ \forall\, m{>}n.\ f\ m \le c_1 * g\ m \wedge g\ m \le c_2 * f\ m$
shows $f \cong g$
unfolding *equiv-fun-def* **by** (*rule assms*)

lemma *not-equiv-funI*:
assumes $\bigwedge c_1\ c_2\ n.\ c_1 > 0 \Longrightarrow c_2 > 0 \Longrightarrow$
$\exists\, m{>}n.\ c_1 * f\ m < g\ m \vee c_2 * g\ m < f\ m$
shows $\neg\ f \cong g$
using *assms* **unfolding** *equiv-fun-def linorder-not-le* [*symmetric*] **by** *blast*

lemma *equiv-funE*:
assumes $f \cong g$
obtains $n\ c_1\ c_2$ **where** $c_1 > 0$ **and** $c_2 > 0$
and $\bigwedge m.\ m > n \Longrightarrow f\ m \le c_1 * g\ m \wedge g\ m \le c_2 * f\ m$

using *assms* **unfolding** *equiv-fun-def* **by** *blast*

lemma *not-equiv-funE*:
fixes $n\ c_1\ c_2$
assumes $\neg f \cong g$ **and** $c_1 > 0$ **and** $c_2 > 0$
obtains m **where** $m > n$
and $c_1 * f\ m < g\ m \lor c_2 * g\ m < f\ m$
using *assms* **unfolding** *equiv-fun-def linorder-not-le* [*symmetric*] **by** *blast*

50.5 The $\prec$ relation, the strict part of $\lesssim$

definition *less-fun* :: $(nat \Rightarrow nat) \Rightarrow (nat \Rightarrow nat) \Rightarrow bool$ (**infix** $\prec$ *50*)
where
$f \prec g \longleftrightarrow f \lesssim g \land \neg g \lesssim f$

lemma *less-funI*:
assumes $\exists c{>}0.\ \exists n.\ \forall m{>}n.\ f\ m \le c * g\ m$
and $\bigwedge c\ n.\ c > 0 \Longrightarrow \exists m{>}n.\ c * f\ m < g\ m$
shows $f \prec g$
using *assms* **unfolding** *less-fun-def less-eq-fun-def linorder-not-less* [*symmetric*]
by *blast*

lemma *not-less-funI*:
assumes $\bigwedge c\ n.\ c > 0 \Longrightarrow \exists m{>}n.\ c * g\ m < f\ m$
and $\exists c{>}0.\ \exists n.\ \forall m{>}n.\ g\ m \le c * f\ m$
shows $\neg f \prec g$
using *assms* **unfolding** *less-fun-def less-eq-fun-def linorder-not-less* [*symmetric*]
by *blast*

lemma *less-funE* [*elim?*]:
assumes $f \prec g$
obtains $n\ c$ **where** $c > 0$ **and** $\bigwedge m.\ m > n \Longrightarrow f\ m \le c * g\ m$
and $\bigwedge c\ n.\ c > 0 \Longrightarrow \exists m{>}n.\ c * f\ m < g\ m$
proof –
from *assms* **have** $f \lesssim g$ **and** $\neg g \lesssim f$ **by** (*simp-all add*: *less-fun-def*)
from ‹$f \lesssim g$› **obtain** $n\ c$ **where** $*{:}c > 0\ \bigwedge m.\ m > n \Longrightarrow f\ m \le c * g\ m$
by (*rule less-eq-funE*) *blast*
{ fix $c\ n$ :: *nat*
assume $c > 0$
with ‹$\neg g \lesssim f$› **obtain** m **where** $m > n\ c * f\ m < g\ m$
by (*rule not-less-eq-funE*) *blast*
then have $**{:}\ \exists m{>}n.\ c * f\ m < g\ m$ **by** *blast*
} note $** =$ *this*
from $*\ **$ **show** *thesis* **by** (*rule that*)
qed

lemma *not-less-funE*:
assumes $\neg f \prec g$ **and** $c > 0$
obtains m **where** $m > n$ **and** $c * g\ m < f\ m$

| d q **where** $\bigwedge m.\ d > 0 \Longrightarrow m > q \Longrightarrow g\ q \leq d * f\ q$
using *assms* **unfolding** *less-fun-def linorder-not-less* [*symmetric*] **by** *blast*

I did not find a proof for $f \prec g \longleftrightarrow f \in o(g)$. Maybe this only holds if f and/or g are of a certain class of functions. However $f \in o(g) \longrightarrow f \prec g$ is provable, and this yields a handy introduction rule.

Note that D. Knuth ignores o altogether. So what ...

Something still has to be said about the coefficient c in the definition of $(\prec)$. In the typical definition of o, it occurs on the *right* hand side of the $(>)$. The reason is that the situation is dual to the definition of O: the definition works since c may become arbitrary small. Since this is not possible within $\mathbb{N}$, we push the coefficient to the left hand side instead such that it may become arbitrary big instead.

lemma *less-fun-strongI*:
assumes $\bigwedge c.\ c > 0 \Longrightarrow \exists n.\ \forall m>n.\ c * f\ m < g\ m$
shows $f \prec g$
proof (*rule less-funI*)
have $1 > (0::nat)$ **by** *simp*
from *assms* ‹$1 > 0$› **have** $\exists n.\ \forall m>n.\ 1 * f\ m < g\ m$ **.**
then obtain n **where** $*$: $\bigwedge m.\ m > n \Longrightarrow 1 * f\ m < g\ m$ **by** *blast*
have $\forall m>n.\ f\ m \leq 1 * g\ m$
proof (*rule allI*, *rule impI*)
fix m
assume $m > n$
with $*$ **have** $1 * f\ m < g\ m$ **by** *simp*
then show $f\ m \leq 1 * g\ m$ **by** *simp*
qed
with ‹$1 > 0$› **show** $\exists c>0.\ \exists n.\ \forall m>n.\ f\ m \leq c * g\ m$ **by** *blast*
fix c n :: *nat*
assume $c > 0$
with *assms* **obtain** q **where** $\bigwedge m.\ m > q \Longrightarrow c * f\ m < g\ m$ **by** *blast*
then have $c * f\ (Suc\ (q + n)) < g\ (Suc\ (q + n))$ **by** *simp*
moreover have $Suc\ (q + n) > n$ **by** *simp*
ultimately show $\exists m>n.\ c * f\ m < g\ m$ **by** *blast*
qed

50.6 $\lesssim$ is a preorder

This yields all lemmas relating $\lesssim$, $\prec$ and $\cong$.

interpretation *fun-order*: *preorder-equiv less-eq-fun less-fun*
rewrites *fun-order.equiv* = *equiv-fun*
proof −
interpret *preorder*: *preorder-equiv less-eq-fun less-fun*
proof
fix f g h
show $f \lesssim f$
proof
have $\exists n.\ \forall m>n.\ f\ m \leq 1 * f\ m$ **by** *auto*

then show $\exists c>0.\ \exists n.\ \forall m>n.\ f\ m \le c * f\ m$ **by** *blast*
qed
show $f \prec g \longleftrightarrow f \lesssim g \wedge \neg\ g \lesssim f$
by (*fact less-fun-def*)
assume $f \lesssim g$ **and** $g \lesssim h$
show $f \lesssim h$
proof
from $\langle f \lesssim g \rangle$ **obtain** $n_1\ c_1$
where $c_1 > 0$ **and** P_1: $\bigwedge m.\ m > n_1 \Longrightarrow f\ m \le c_1 * g\ m$
by *rule blast*
from $\langle g \lesssim h \rangle$ **obtain** $n_2\ c_2$
where $c_2 > 0$ **and** P_2: $\bigwedge m.\ m > n_2 \Longrightarrow g\ m \le c_2 * h\ m$
by *rule blast*
have $\forall m>max\ n_1\ n_2.\ f\ m \le (c_1 * c_2) * h\ m$
proof (*rule allI*, *rule impI*)
fix m
assume Q: $m > max\ n_1\ n_2$
from $P_1\ Q$ **have** $*$: $f\ m \le c_1 * g\ m$ **by** *simp*
from $P_2\ Q$ **have** $g\ m \le c_2 * h\ m$ **by** *simp*
with $\langle c_1 > 0 \rangle$ **have** $c_1 * g\ m \le (c_1 * c_2) * h\ m$ **by** *simp*
with $*$ **show** $f\ m \le (c_1 * c_2) * h\ m$ **by** (*rule order-trans*)
qed
then have $\exists n.\ \forall m>n.\ f\ m \le (c_1 * c_2) * h\ m$ **by** *rule*
moreover from $\langle c_1 > 0 \rangle\ \langle c_2 > 0 \rangle$ **have** $c_1 * c_2 > 0$ **by** *simp*
ultimately show $\exists c>0.\ \exists n.\ \forall m>n.\ f\ m \le c * h\ m$ **by** *blast*
qed
qed
from *preorder.preorder-equiv-axioms* **show** *class.preorder-equiv less-eq-fun less-fun*
.
show *preorder-equiv.equiv less-eq-fun* = *equiv-fun*
proof (*rule ext*, *rule ext*, *unfold preorder.equiv-def*)
fix $f\ g$
show $f \lesssim g \wedge g \lesssim f \longleftrightarrow f \cong g$
proof
assume $f \cong g$
then obtain $n\ c_1\ c_2$ **where** $c_1 > 0$ **and** $c_2 > 0$
and $*$: $\bigwedge m.\ m > n \Longrightarrow f\ m \le c_1 * g\ m \wedge g\ m \le c_2 * f\ m$
by (*rule equiv-funE*) *blast*
have $\forall m>n.\ f\ m \le c_1 * g\ m$
proof (*rule allI*, *rule impI*)
fix m
assume $m > n$
with $*$ **show** $f\ m \le c_1 * g\ m$ **by** *simp*
qed
with $\langle c_1 > 0 \rangle$ **have** $\exists c>0.\ \exists n.\ \forall m>n.\ f\ m \le c * g\ m$ **by** *blast*
then have $f \lesssim g$ **..**
have $\forall m>n.\ g\ m \le c_2 * f\ m$
proof (*rule allI*, *rule impI*)
fix m

assume $m > n$
with $*$ **show** $g\ m \leq c_2 * f\ m$ **by** *simp*
qed
with ‹$c_2 > 0$› **have** $\exists\, c>0.\ \exists\, n.\ \forall\, m>n.\ g\ m \leq c * f\ m$ **by** *blast*
then have $g \lesssim f$ **..**
from ‹$f \lesssim g$› **and** ‹$g \lesssim f$› **show** $f \lesssim g \wedge g \lesssim f$ **..**
next
assume $f \lesssim g \wedge g \lesssim f$
then have $f \lesssim g$ **and** $g \lesssim f$ **by** *auto*
from ‹$f \lesssim g$› **obtain** $n_1\ c_1$ **where** $c_1 > 0$
and P_1: $\bigwedge m.\ m > n_1 \Longrightarrow f\ m \leq c_1 * g\ m$ **by** *rule blast*
from ‹$g \lesssim f$› **obtain** $n_2\ c_2$ **where** $c_2 > 0$
and P_2: $\bigwedge m.\ m > n_2 \Longrightarrow g\ m \leq c_2 * f\ m$ **by** *rule blast*
have $\forall\, m>max\ n_1\ n_2.\ f\ m \leq c_1 * g\ m \wedge g\ m \leq c_2 * f\ m$
proof (*rule allI*, *rule impI*)
fix m
assume Q: $m > max\ n_1\ n_2$
from $P_1\ Q$ **have** $f\ m \leq c_1 * g\ m$ **by** *simp*
moreover from $P_2\ Q$ **have** $g\ m \leq c_2 * f\ m$ **by** *simp*
ultimately show $f\ m \leq c_1 * g\ m \wedge g\ m \leq c_2 * f\ m$ **..**
qed
with ‹$c_1 > 0$› ‹$c_2 > 0$› **have** $\exists\, c_1>0.\ \exists\, c_2>0.\ \exists\, n.$
$\forall\, m>n.\ f\ m \leq c_1 * g\ m \wedge g\ m \leq c_2 * f\ m$ **by** *blast*
then show $f \cong g$ **by** (*rule equiv-funI*)
qed
qed
qed

declare *fun-order.antisym* [*intro?*]

50.7 Simple examples

Most of these are left as constructive exercises for the reader. Note that additional preconditions to the functions may be necessary. The list here is by no means to be intended as complete construction set for typical functions, here surely something has to be added yet.

$(\lambda n.\ f\ n + k) \cong f$

lemma *equiv-fun-mono-const*:
assumes *mono f* **and** $\exists\, n.\ f\ n > 0$
shows $(\lambda n.\ f\ n + k) \cong f$
proof (*cases* $k = 0$)
case *True* **then show** *?thesis* **by** *simp*
next
case *False*
show *?thesis*
proof
show $(\lambda n.\ f\ n + k) \lesssim f$
proof

```
    from ‹∃ n. f n > 0› obtain n where f n > 0 ..
    have ∀ m>n. f m + k ≤ Suc k * f m
    proof (rule allI, rule impI)
      fix m
      assume n < m
      with ‹mono f› have f n ≤ f m
        using less-imp-le-nat monoE by blast
      with ‹0 < f n› have 0 < f m by auto
      then obtain l where f m = Suc l by (cases f m) simp-all
      then show f m + k ≤ Suc k * f m by simp
    qed
    then show ∃ c>0. ∃ n. ∀ m>n. f m + k ≤ c * f m by blast
  qed
  show f ≲ (λn. f n + k)
  proof
    have f m ≤ 1 * (f m + k) for m by simp
    then show ∃ c>0. ∃ n. ∀ m>n. f m ≤ c * (f m + k) by blast
  qed
 qed
qed

lemma
 assumes strict-mono f
 shows (λn. f n + k) ≅ f
proof (rule equiv-fun-mono-const)
 from assms show mono f by (rule strict-mono-mono)
 show ∃ n. 0 < f n
 proof (rule ccontr)
  assume ¬ (∃ n. 0 < f n)
  then have ⋀n. f n = 0 by simp
  then have f 0 = f 1 by simp
  moreover from ‹strict-mono f› have f 0 < f 1
    by (simp add: strict-mono-def)
  ultimately show False by simp
 qed
qed

lemma
 (λn. Suc k * f n) ≅ f
proof
 show (λn. Suc k * f n) ≲ f
 proof
  have Suc k * f m ≤ Suc k * f m for m by simp
  then show ∃ c>0. ∃ n. ∀ m>n. Suc k * f m ≤ c * f m by blast
 qed
 show f ≲ (λn. Suc k * f n)
 proof
  have f m ≤ 1 * (Suc k * f m) for m by simp
  then show ∃ c>0. ∃ n. ∀ m>n. f m ≤ c * (Suc k * f m) by blast
```

  qed
qed

lemma
  $f \lesssim (\lambda n.\ f\ n + g\ n)$
  by *rule auto*

lemma
  $(\lambda\text{-}.\ 0) \prec (\lambda n.\ Suc\ k)$
  by (*rule less-fun-strongI*) *auto*

lemma
  $(\lambda\text{-}.\ k) \prec Discrete.log$
proof (*rule less-fun-strongI*)
  fix $c :: nat$
  have $\forall m>2\ \hat{}\ (Suc\ (c * k)).\ c * k < Discrete.log\ m$
  proof (*rule allI*, *rule impI*)
    fix $m :: nat$
    assume $2\ \hat{}\ Suc\ (c * k) < m$
    then have $2\ \hat{}\ Suc\ (c * k) \leq m$ **by** *simp*
    with *log-mono* **have** $Discrete.log\ (2\ \hat{}\ (Suc\ (c * k))) \leq Discrete.log\ m$
      by (*blast dest*: *monoD*)
    moreover have $c * k < Discrete.log\ (2\ \hat{}\ (Suc\ (c * k)))$ **by** *simp*
    ultimately show $c * k < Discrete.log\ m$ **by** *auto*
  qed
  then show $\exists n.\ \forall m>n.\ c * k < Discrete.log\ m$ **..**
qed

$Discrete.log \prec Discrete.sqrt$

lemma
  $Discrete.sqrt \prec id$
proof (*rule less-fun-strongI*)
  fix $c :: nat$
  assume $0 < c$
  have $\forall m>(Suc\ c)^2.\ c * Discrete.sqrt\ m < id\ m$
  proof (*rule allI*, *rule impI*)
    fix m
    assume $(Suc\ c)^2 < m$
    then have $(Suc\ c)^2 \leq m$ **by** *simp*
    with *mono-sqrt* **have** $Discrete.sqrt\ ((Suc\ c)^2) \leq Discrete.sqrt\ m$ **by** (*rule monoE*)
    then have $Suc\ c \leq Discrete.sqrt\ m$ **by** *simp*
    then have $c < Discrete.sqrt\ m$ **by** *simp*
    moreover from ‹$(Suc\ c)^2 < m$› **have** $Discrete.sqrt\ m > 0$ **by** *simp*
    ultimately have $c * Discrete.sqrt\ m < Discrete.sqrt\ m * Discrete.sqrt\ m$ **by** *simp*
    also have $\ldots \leq m$ **by** (*simp add*: *power2-eq-square* [*symmetric*])
    finally show $c * Discrete.sqrt\ m < id\ m$ **by** *simp*
  qed

then show $\exists\, n.\ \forall\, m{>}n.\ c * Discrete.sqrt\ m < id\ m$ **..**
qed

lemma
$id \prec (\lambda n.\ n^2)$
by (*rule less-fun-strongI*) (*auto simp add*: *power2-eq-square*)

lemma
$(\lambda n.\ n\ \hat{}\ k) \prec (\lambda n.\ n\ \hat{}\ Suc\ k)$
by (*rule less-fun-strongI*) *auto*

$(\lambda n.\ n^k) \prec op\ \hat{}\ 2$

end

51 Fundamental Theorem of Algebra

theory *Fundamental-Theorem-Algebra*
imports *Polynomial Complex-Main*
begin

51.1 More lemmas about module of complex numbers

The triangle inequality for cmod

lemma *complex-mod-triangle-sub*: $cmod\ w \leq cmod\ (w + z) + norm\ z$
using *complex-mod-triangle-ineq2*[*of* $w + z\ -z$] **by** *auto*

51.2 Basic lemmas about polynomials

lemma *poly-bound-exists*:
fixes p :: $'a$::{*comm-semiring-0*,*real-normed-div-algebra*} *poly*
shows $\exists\, m.\ m > 0 \wedge (\forall\, z.\ norm\ z \leq r \longrightarrow norm\ (poly\ p\ z) \leq m)$
proof (*induct p*)
case *0*
then show *?case* **by** (*rule exI*[**where** $x{=}1$]) *simp*
next
case (*pCons c cs*)
from *pCons.hyps* **obtain** m **where** m: $\forall\, z.\ norm\ z \leq r \longrightarrow norm\ (poly\ cs\ z) \leq m$
by *blast*
let $?k = 1 + norm\ c + |r * m|$
have *kp*: $?k > 0$
using *abs-ge-zero*[*of* $r{*}m$] *norm-ge-zero*[*of* c] **by** *arith*
have $norm\ (poly\ (pCons\ c\ cs)\ z) \leq ?k$ **if** H: $norm\ z \leq r$ **for** z
proof −
from $m\ H$ **have** *th*: $norm\ (poly\ cs\ z) \leq m$
by *blast*
from H **have** *rp*: $r \geq 0$
using *norm-ge-zero*[*of* z] **by** *arith*

```
    have norm (poly (pCons c cs) z) ≤ norm c + norm (z * poly cs z)
      using norm-triangle-ineq[of c z* poly cs z] by simp
    also have ... ≤ norm c + r * m
      using mult-mono[OF H th rp norm-ge-zero[of poly cs z]]
      by (simp add: norm-mult)
    also have ... ≤ ?k
      by simp
    finally show ?thesis .
  qed
  with kp show ?case by blast
qed
```

Offsetting the variable in a polynomial gives another of same degree

```
definition offset-poly :: 'a::comm-semiring-0 poly ⇒ 'a ⇒ 'a poly
  where offset-poly p h = fold-coeffs (λa q. smult h q + pCons a q) p 0

lemma offset-poly-0: offset-poly 0 h = 0
  by (simp add: offset-poly-def)

lemma offset-poly-pCons:
  offset-poly (pCons a p) h =
    smult h (offset-poly p h) + pCons a (offset-poly p h)
  by (cases p = 0 ∧ a = 0) (auto simp add: offset-poly-def)

lemma offset-poly-single: offset-poly [:a:] h = [:a:]
  by (simp add: offset-poly-pCons offset-poly-0)

lemma poly-offset-poly: poly (offset-poly p h) x = poly p (h + x)
  apply (induct p)
  apply (simp add: offset-poly-0)
  apply (simp add: offset-poly-pCons algebra-simps)
  done

lemma offset-poly-eq-0-lemma: smult c p + pCons a p = 0 ⟹ p = 0
  by (induct p arbitrary: a) (simp, force)

lemma offset-poly-eq-0-iff: offset-poly p h = 0 ⟷ p = 0
  apply (safe intro!: offset-poly-0)
  apply (induct p)
  apply simp
  apply (simp add: offset-poly-pCons)
  apply (frule offset-poly-eq-0-lemma, simp)
  done

lemma degree-offset-poly: degree (offset-poly p h) = degree p
  apply (induct p)
  apply (simp add: offset-poly-0)
  apply (case-tac p = 0)
  apply (simp add: offset-poly-0 offset-poly-pCons)
```

apply (*simp add*: *offset-poly-pCons*)
apply (*subst degree-add-eq-right*)
apply (*rule le-less-trans* [*OF degree-smult-le*])
apply (*simp add*: *offset-poly-eq-0-iff*)
apply (*simp add*: *offset-poly-eq-0-iff*)
done

definition *psize* $p = (if\ p = 0\ then\ 0\ else\ Suc\ (degree\ p))$

lemma *psize-eq-0-iff* [*simp*]: $psize\ p = 0 \longleftrightarrow p = 0$
unfolding *psize-def* **by** *simp*

lemma *poly-offset*:
fixes $p :: 'a::comm\text{-}ring\text{-}1\ poly$
shows $\exists q.\ psize\ q = psize\ p \wedge (\forall x.\ poly\ q\ x = poly\ p\ (a + x))$
proof (*intro exI conjI*)
show $psize\ (offset\text{-}poly\ p\ a) = psize\ p$
unfolding *psize-def*
by (*simp add*: *offset-poly-eq-0-iff degree-offset-poly*)
show $\forall x.\ poly\ (offset\text{-}poly\ p\ a)\ x = poly\ p\ (a + x)$
by (*simp add*: *poly-offset-poly*)
qed

An alternative useful formulation of completeness of the reals

lemma *real-sup-exists*:
assumes *ex*: $\exists x.\ P\ x$
and *bz*: $\exists z.\ \forall x.\ P\ x \longrightarrow x < z$
shows $\exists s::real.\ \forall y.\ (\exists x.\ P\ x \wedge y < x) \longleftrightarrow y < s$
proof
from *bz* **have** *bdd-above* (*Collect P*)
by (*force intro*: *less-imp-le*)
then show $\forall y.\ (\exists x.\ P\ x \wedge y < x) \longleftrightarrow y < Sup\ (Collect\ P)$
using *ex bz* **by** (*subst less-cSup-iff*) *auto*
qed

51.3 Fundamental theorem of algebra

lemma *unimodular-reduce-norm*:
assumes *md*: $cmod\ z = 1$
shows $cmod\ (z + 1) < 1 \vee cmod\ (z - 1) < 1 \vee cmod\ (z + ii) < 1 \vee cmod\ (z - ii) < 1$
proof −
obtain *x y* **where** *z*: $z = Complex\ x\ y$
by (*cases z*) *auto*
from *md z* **have** *xy*: $x^2 + y^2 = 1$
by (*simp add*: *cmod-def*)
have *False* **if** $cmod\ (z + 1) \geq 1\ cmod\ (z - 1) \geq 1\ cmod\ (z + ii) \geq 1\ cmod\ (z - ii) \geq 1$
proof −
from *that z xy* **have** $2 * x \leq 1\ 2 * x \geq -1\ 2 * y \leq 1\ 2 * y \geq -1$

```
    by (simp-all add: cmod-def power2-eq-square algebra-simps)
  then have |2 ∗ x| ≤ 1 |2 ∗ y| ≤ 1
    by simp-all
  then have |2 ∗ x|^2 ≤ 1^2 |2 ∗ y|^2 ≤ 1^2
    by − (rule power-mono, simp, simp)+
  then have th0: 4 ∗ x^2 ≤ 1 4 ∗ y^2 ≤ 1
    by (simp-all add: power-mult-distrib)
  from add-mono[OF th0] xy show ?thesis
    by simp
 qed
 then show ?thesis
  unfolding linorder-not-le[symmetric] by blast
qed
```

Hence we can always reduce modulus of *1 + b z^n* if nonzero

```
lemma reduce-poly-simple:
 assumes b: b ≠ 0
  and n: n ≠ 0
 shows ∃ z. cmod (1 + b ∗ z^n) < 1
 using n
proof (induct n rule: nat-less-induct)
 fix n
 assume IH: ∀ m<n. m ≠ 0 ⟶ (∃ z. cmod (1 + b ∗ z ^ m) < 1)
 assume n: n ≠ 0
 let ?P = λz n. cmod (1 + b ∗ z ^ n) < 1
 show ∃ z. ?P z n
 proof cases
  assume even n
  then have ∃ m. n = 2 ∗ m
    by presburger
  then obtain m where m: n = 2 ∗ m
    by blast
  from n m have m ≠ 0 m < n
    by presburger+
  with IH[rule-format, of m] obtain z where z: ?P z m
    by blast
  from z have ?P (csqrt z) n
    by (simp add: m power-mult)
  then show ?thesis ..
 next
  assume odd n
  then have ∃ m. n = Suc (2 ∗ m)
    by presburger+
  then obtain m where m: n = Suc (2 ∗ m)
    by blast
  have th0: cmod (complex-of-real (cmod b) / b) = 1
    using b by (simp add: norm-divide)
  from unimodular-reduce-norm[OF th0] ‹odd n›
  have ∃ v. cmod (complex-of-real (cmod b) / b + v^n) < 1
```

```
    apply (cases cmod (complex-of-real (cmod b) / b + 1) < 1)
    apply (rule-tac x=1 in exI)
    apply simp
    apply (cases cmod (complex-of-real (cmod b) / b − 1) < 1)
    apply (rule-tac x=−1 in exI)
    apply simp
    apply (cases cmod (complex-of-real (cmod b) / b + ii) < 1)
    apply (cases even m)
    apply (rule-tac x=ii in exI)
    apply (simp add: m power-mult)
    apply (rule-tac x=− ii in exI)
    apply (simp add: m power-mult)
    apply (cases even m)
    apply (rule-tac x=− ii in exI)
    apply (simp add: m power-mult)
    apply (auto simp add: m power-mult)
    apply (rule-tac x=ii in exI)
    apply (auto simp add: m power-mult)
    done
  then obtain v where v: cmod (complex-of-real (cmod b) / b + v^n) < 1
    by blast
  let ?w = v / complex-of-real (root n (cmod b))
  from odd-real-root-pow[OF ‹odd n›, of cmod b]
  have th1: ?w ^ n = v^n / complex-of-real (cmod b)
    by (simp add: power-divide of-real-power[symmetric])
  have th2:cmod (complex-of-real (cmod b) / b) = 1
    using b by (simp add: norm-divide)
  then have th3: cmod (complex-of-real (cmod b) / b) ≥ 0
    by simp
  have th4: cmod (complex-of-real (cmod b) / b) ∗
      cmod (1 + b ∗ (v ^ n / complex-of-real (cmod b))) <
      cmod (complex-of-real (cmod b) / b) ∗ 1
    apply (simp only: norm-mult[symmetric] distrib-left)
    using b v
    apply (simp add: th2)
    done
  from mult-left-less-imp-less[OF th4 th3]
  have ?P ?w n unfolding th1 .
  then show ?thesis ..
 qed
qed
```

Bolzano-Weierstrass type property for closed disc in complex plane.

```
lemma metric-bound-lemma: cmod (x − y) ≤ |Re x − Re y| + |Im x − Im y|
  using real-sqrt-sum-squares-triangle-ineq[of Re x − Re y 0 0 Im x − Im y]
  unfolding cmod-def by simp

lemma bolzano-weierstrass-complex-disc:
  assumes r: ∀ n. cmod (s n) ≤ r
```

shows $\exists f\ z.\ subseq\ f \wedge (\forall\ e > 0.\ \exists\ N.\ \forall\ n \geq N.\ cmod\ (s\ (f\ n) - z) < e)$
proof −
from *seq-monosub*[*of Re* ∘ *s*]
obtain *f* **where** *f*: *subseq f monoseq* (λ*n. Re* (*s* (*f n*)))
unfolding *o-def* **by** *blast*
from *seq-monosub*[*of Im* ∘ *s* ∘ *f*]
obtain *g* **where** *g*: *subseq g monoseq* (λ*n. Im* (*s* (*f* (*g n*))))
unfolding *o-def* **by** *blast*
let *?h* = *f* ∘ *g*
from *r*[*rule-format*, *of 0*] **have** *rp*: $r \geq 0$
using *norm-ge-zero*[*of s 0*] **by** *arith*
have *th*: ∀ *n*. *r* + *1* ≥ |*Re* (*s n*)|
proof
fix *n*
from *abs-Re-le-cmod*[*of s n*] *r*[*rule-format*, *of n*]
show |*Re* (*s n*)| ≤ *r* + *1* **by** *arith*
qed
have *conv1*: *convergent* (λ*n. Re* (*s* (*f n*)))
apply (*rule Bseq-monoseq-convergent*)
apply (*simp add*: *Bseq-def*)
apply (*metis gt-ex le-less-linear less-trans order.trans th*)
apply (*rule f*(*2*))
done
have *th*: ∀ *n*. *r* + *1* ≥ |*Im* (*s n*)|
proof
fix *n*
from *abs-Im-le-cmod*[*of s n*] *r*[*rule-format*, *of n*]
show |*Im* (*s n*)| ≤ *r* + *1*
by *arith*
qed

have *conv2*: *convergent* (λ*n. Im* (*s* (*f* (*g n*))))
apply (*rule Bseq-monoseq-convergent*)
apply (*simp add*: *Bseq-def*)
apply (*metis gt-ex le-less-linear less-trans order.trans th*)
apply (*rule g*(*2*))
done

from *conv1*[*unfolded convergent-def*] **obtain** *x* **where** *LIMSEQ* (λ*n. Re* (*s* (*f n*))) *x*
by *blast*
then have *x*: ∀ *r*>*0*. ∃ *n0*. ∀ *n*≥*n0*. |*Re* (*s* (*f n*)) − *x*| < *r*
unfolding *LIMSEQ-iff real-norm-def* **.**

from *conv2*[*unfolded convergent-def*] **obtain** *y* **where** *LIMSEQ* (λ*n. Im* (*s* (*f* (*g n*)))) *y*
by *blast*
then have *y*: ∀ *r*>*0*. ∃ *n0*. ∀ *n*≥*n0*. |*Im* (*s* (*f* (*g n*))) − *y*| < *r*
unfolding *LIMSEQ-iff real-norm-def* **.**

```
  let ?w = Complex x y
  from f(1) g(1) have hs: subseq ?h
    unfolding subseq-def by auto
  have ∃N. ∀n≥N. cmod (s (?h n) − ?w) < e if e > 0 for e
  proof −
    from that have e2: e/2 > 0
      by simp
    from x[rule-format, OF e2] y[rule-format, OF e2]
    obtain N1 N2 where N1: ∀n≥N1. |Re (s (f n)) − x| < e / 2
      and N2: ∀n≥N2. |Im (s (f (g n))) − y| < e / 2
      by blast
    have cmod (s (?h n) − ?w) < e if n ≥ N1 + N2 for n
    proof −
      from that have nN1: g n ≥ N1 and nN2: n ≥ N2
        using seq-suble[OF g(1), of n] by arith+
        from add-strict-mono[OF N1[rule-format, OF nN1] N2[rule-format, OF
nN2]]
      show ?thesis
        using metric-bound-lemma[of s (f (g n)) ?w] by simp
    qed
    then show ?thesis by blast
  qed
  with hs show ?thesis by blast
qed
```

Polynomial is continuous.

```
lemma poly-cont:
  fixes p :: 'a::{comm-semiring-0,real-normed-div-algebra} poly
  assumes ep: e > 0
  shows ∃d >0. ∀w. 0 < norm (w − z) ∧ norm (w − z) < d ⟶ norm (poly p
w − poly p z) < e
proof −
  obtain q where q: degree q = degree p ⋀x. poly q x = poly p (z + x)
  proof
    show degree (offset-poly p z) = degree p
      by (rule degree-offset-poly)
    show ⋀x. poly (offset-poly p z) x = poly p (z + x)
      by (rule poly-offset-poly)
  qed
  have th: ⋀w. poly q (w − z) = poly p w
    using q(2)[of w − z for w] by simp
  show ?thesis unfolding th[symmetric]
  proof (induct q)
    case 0
    then show ?case
      using ep by auto
  next
    case (pCons c cs)
    from poly-bound-exists[of 1 cs]
```

```
    obtain m where m: m > 0 ⋀z. norm z ≤ 1 ⟹ norm (poly cs z) ≤ m
      by blast
    from ep m(1) have em0: e/m > 0
      by (simp add: field-simps)
    have one0: 1 > (0::real)
      by arith
    from real-lbound-gt-zero[OF one0 em0]
    obtain d where d: d > 0 d < 1 d < e / m
      by blast
    from d(1,3) m(1) have dm: d ∗ m > 0 d ∗ m < e
      by (simp-all add: field-simps)
    show ?case
     proof (rule ex-forward[OF real-lbound-gt-zero[OF one0 em0]], clarsimp simp
add: norm-mult)
      fix d w
      assume H: d > 0 d < 1 d < e/m w ≠ z norm (w − z) < d
      then have d1: norm (w−z) ≤ 1 d ≥ 0
        by simp-all
      from H(3) m(1) have dme: d∗m < e
        by (simp add: field-simps)
      from H have th: norm (w − z) ≤ d
        by simp
      from mult-mono[OF th m(2)[OF d1(1)] d1(2) norm-ge-zero] dme
      show norm (w − z) ∗ norm (poly cs (w − z)) < e
        by simp
    qed
  qed
qed
```

Hence a polynomial attains minimum on a closed disc in the complex plane.

```
lemma poly-minimum-modulus-disc: ∃z. ∀w. cmod w ≤ r ⟶ cmod (poly p z) ≤
cmod (poly p w)
proof −
  show ?thesis
  proof (cases r ≥ 0)
    case False
    then show ?thesis
      by (metis norm-ge-zero order.trans)
  next
    case True
    then have cmod 0 ≤ r ∧ cmod (poly p 0) = − (− cmod (poly p 0))
      by simp
    then have mth1: ∃x z. cmod z ≤ r ∧ cmod (poly p z) = − x
      by blast
    have False if cmod z ≤ r cmod (poly p z) = − x ¬ x < 1 for x z
    proof −
      from that have − x < 0
        by arith
```

with *that*(*2*) *norm-ge-zero*[*of poly p z*] **show** *?thesis*
by *simp*
qed
then have *mth2*: $\exists z. \forall x. (\exists z.\ cmod\ z \le r \wedge cmod\ (poly\ p\ z) = -\ x) \longrightarrow x < z$
by *blast*
from *real-sup-exists*[*OF mth1 mth2*] **obtain** *s* **where**
s: $\forall y. (\exists x. (\exists z.\ cmod\ z \le r \wedge cmod\ (poly\ p\ z) = -\ x) \wedge y < x) \longleftrightarrow y < s$
by *blast*
let *?m* = − *s*
have *s1*[*unfolded minus-minus*]:
$(\exists z\ x.\ cmod\ z \le r \wedge -\ (-\ cmod\ (poly\ p\ z)) < y) \longleftrightarrow ?m < y$ **for** *y*
using *s*[*rule-format*, *of* −*y*]
unfolding *minus-less-iff*[*of y*] *equation-minus-iff* **by** *blast*
from *s1*[*of ?m*] **have** *s1m*: $\bigwedge z\ x.\ cmod\ z \le r \Longrightarrow cmod\ (poly\ p\ z) \ge ?m$
by *auto*
have $\exists z.\ cmod\ z \le r \wedge cmod\ (poly\ p\ z) < -\ s + 1\ /\ real\ (Suc\ n)$ **for** *n*
using *s1*[*rule-format*, *of ?m* + *1*/*real* (*Suc n*)] **by** *simp*
then have *th*: $\forall n. \exists z.\ cmod\ z \le r \wedge cmod\ (poly\ p\ z) < -\ s + 1\ /\ real\ (Suc\ n)$ **..**
from *choice*[*OF th*] **obtain** *g* **where**
g: $\forall n.\ cmod\ (g\ n) \le r\ \forall n.\ cmod\ (poly\ p\ (g\ n)) < ?m + 1\ /real(Suc\ n)$
by *blast*
from *bolzano-weierstrass-complex-disc*[*OF g*(*1*)]
obtain *f z* **where** *fz*: $subseq\ f\ \forall e>0. \exists N. \forall n\ge N.\ cmod\ (g\ (f\ n) - z) < e$
by *blast*
{
fix *w*
assume *wr*: $cmod\ w \le r$
let $?e = |cmod\ (poly\ p\ z) - ?m|$
{
assume *e*: $?e > 0$
then have *e2*: $?e/2 > 0$
by *simp*
from *poly-cont*[*OF e2*, *of z p*] **obtain** *d* **where**
d: $d > 0\ \forall w.\ 0<cmod\ (w - z)\wedge cmod(w - z) < d \longrightarrow cmod(poly\ p\ w - poly\ p\ z) < ?e/2$
by *blast*
have *th1*: $cmod(poly\ p\ w - poly\ p\ z) < ?e\ /\ 2$ **if** *w*: $cmod\ (w - z) < d$ **for** *w*
using *d*(*2*)[*rule-format*, *of w*] *w e* **by** (*cases w* = *z*) *simp-all*
from *fz*(*2*) *d*(*1*) **obtain** *N1* **where** *N1*: $\forall n\ge N1.\ cmod\ (g\ (f\ n) - z) < d$
by *blast*
from *reals-Archimedean2*[*of 2*/*?e*] **obtain** *N2* :: *nat* **where** *N2*: $2/?e < real\ N2$
by *blast*
have *th2*: $cmod\ (poly\ p\ (g\ (f\ (N1 + N2))) - poly\ p\ z) < ?e/2$
using *N1*[*rule-format*, *of N1* + *N2*] *th1* **by** *simp*
have *th0*: $a < e2 \Longrightarrow |b - m| < e2 \Longrightarrow 2 * e2 \le |b - m| + a \Longrightarrow False$

for *a b e2 m* :: *real*
by *arith*
have *ath*: $m \leq x \Longrightarrow x < m + e \Longrightarrow |x - m| < e$ **for** *m x e* :: *real*
by *arith*
from *s1m*[*OF g*(*1*)[*rule-format*]] **have** *th31*: *?m* ≤ *cmod*(*poly p* (*g* (*f* (*N1* + *N2*)))) **.**
from *seq-suble*[*OF fz*(*1*), *of N1* + *N2*]
have *th00*: *real* (*Suc* (*N1* + *N2*)) ≤ *real* (*Suc* (*f* (*N1* + *N2*)))
by *simp*
have *th000*: *0* ≤ (*1*::*real*) (*1*::*real*) ≤ *1 real* (*Suc* (*N1* + *N2*)) > *0*
using *N2* **by** *auto*
from *frac-le*[*OF th000 th00*]
have *th00*: *?m* + *1* / *real* (*Suc* (*f* (*N1* + *N2*))) ≤ *?m* + *1* / *real* (*Suc* (*N1* + *N2*))
by *simp*
from *g*(*2*)[*rule-format*, *of f* (*N1* + *N2*)]
have *th01*:*cmod* (*poly p* (*g* (*f* (*N1* + *N2*)))) < − *s* + *1* / *real* (*Suc* (*f* (*N1* + *N2*))) **.**
from *order-less-le-trans*[*OF th01 th00*]
have *th32*: *cmod* (*poly p* (*g* (*f* (*N1* + *N2*)))) < *?m* + (*1*/ *real*(*Suc* (*N1* + *N2*))) **.**
from *N2* **have** *2*/*?e* < *real* (*Suc* (*N1* + *N2*))
by *arith*
with *e2 less-imp-inverse-less*[*of 2*/*?e real* (*Suc* (*N1* + *N2*))]
have *?e*/*2* > *1*/ *real* (*Suc* (*N1* + *N2*))
by (*simp add*: *inverse-eq-divide*)
with *ath*[*OF th31 th32*] **have** *thc1*: |*cmod* (*poly p* (*g* (*f* (*N1* + *N2*)))) − *?m*| < *?e*/*2*
by *arith*
have *ath2*: $|a - b| \leq c \Longrightarrow |b - m| \leq |a - m| + c$ **for** *a b c m* :: *real*
by *arith*
have *th22*: |*cmod* (*poly p* (*g* (*f* (*N1* + *N2*)))) − *cmod* (*poly p z*)| ≤
cmod (*poly p* (*g* (*f* (*N1* + *N2*))) − *poly p z*)
by (*simp add*: *norm-triangle-ineq3*)
from *ath2*[*OF th22*, *of ?m*]
have *thc2*: *2* ∗ (*?e*/*2*) ≤
|*cmod*(*poly p* (*g* (*f* (*N1* + *N2*)))) − *?m*| + *cmod* (*poly p* (*g* (*f* (*N1* + *N2*))) − *poly p z*)
by *simp*
from *th0*[*OF th2 thc1 thc2*] **have** *False* **.**
}
then have *?e* = *0*
by *auto*
then have *cmod* (*poly p z*) = *?m*
by *simp*
with *s1m*[*OF wr*] **have** *cmod* (*poly p z*) ≤ *cmod* (*poly p w*)
by *simp*
}
then show *?thesis* **by** *blast*

  qed
qed

Nonzero polynomial in z goes to infinity as z does.

lemma *poly-infinity*:
  fixes *p*:: *'a*::{*comm-semiring-0*,*real-normed-div-algebra*} *poly*
  assumes *ex*: $p \neq 0$
  shows $\exists\, r.\ \forall\, z.\ r \leq norm\ z \longrightarrow d \leq norm\ (poly\ (pCons\ a\ p)\ z)$
  using *ex*
proof (*induct p arbitrary*: *a d*)
  case *0*
  then show *?case* **by** *simp*
next
  case (*pCons c cs a d*)
  show *?case*
  proof (*cases cs = 0*)
    case *False*
    with *pCons.hyps* **obtain** *r* **where** *r*: $\forall\, z.\ r \leq norm\ z \longrightarrow d + norm\ a \leq norm$ $(poly\ (pCons\ c\ cs)\ z)$
      by *blast*
    let $?r = 1 + |r|$
    have $d \leq norm\ (poly\ (pCons\ a\ (pCons\ c\ cs))\ z)$ **if** $1 + |r| \leq norm\ z$ **for** *z*
    proof −
      have *r0*: $r \leq norm\ z$
        using *that* **by** *arith*
      from *r*[*rule-format*, *OF r0*] **have** *th0*: $d + norm\ a \leq 1 * norm(poly\ (pCons$ $c\ cs)\ z)$
        by *arith*
      from *that* **have** *z1*: $norm\ z \geq 1$
        by *arith*
      from *order-trans*[*OF th0 mult-right-mono*[*OF z1 norm-ge-zero*[*of poly* (*pCons* *c cs*) *z*]]]
      have *th1*: $d \leq norm(z * poly\ (pCons\ c\ cs)\ z) - norm\ a$
        unfolding *norm-mult* **by** (*simp add*: *algebra-simps*)
      from *norm-diff-ineq*[*of z* ∗ *poly* (*pCons c cs*) *z a*]
      have *th2*: $norm\ (z * poly\ (pCons\ c\ cs)\ z) - norm\ a \leq norm\ (poly\ (pCons\ a$ $(pCons\ c\ cs))\ z)$
        by (*simp add*: *algebra-simps*)
      from *th1 th2* **show** *?thesis*
        by *arith*
    qed
    then show *?thesis* **by** *blast*
  next
    case *True*
    with *pCons.prems* **have** *c0*: $c \neq 0$
      by *simp*
    have $d \leq norm\ (poly\ (pCons\ a\ (pCons\ c\ cs))\ z)$
      if *h*: $(|d| + norm\ a)\ /\ norm\ c \leq norm\ z$ **for** *z* :: *'a*
    proof −

from *c0* **have** *norm c > 0*
by *simp*
from *h c0* **have** *th0*: $|d|$ + *norm a* ≤ *norm* (*z* ∗ *c*)
by (*simp add*: *field-simps norm-mult*)
have *ath*: ⋀*mzh mazh ma. mzh* ≤ *mazh* + *ma* ⟹ $|d|$ + *ma* ≤ *mzh* ⟹ *d* ≤ *mazh*
by *arith*
from *norm-diff-ineq*[*of z* ∗ *c a*] **have** *th1*: *norm* (*z* ∗ *c*) ≤ *norm* (*a* + *z* ∗ *c*) + *norm a*
by (*simp add*: *algebra-simps*)
from *ath*[*OF th1 th0*] **show** *?thesis*
using *True* **by** *simp*
qed
then show *?thesis* **by** *blast*
qed
qed

Hence polynomial's modulus attains its minimum somewhere.

lemma *poly-minimum-modulus*: ∃ *z*.∀ *w*. *cmod* (*poly p z*) ≤ *cmod* (*poly p w*)
proof (*induct p*)
case *0*
then show *?case* **by** *simp*
next
case (*pCons c cs*)
show *?case*
proof (*cases cs* = *0*)
case *False*
from *poly-infinity*[*OF False*, *of cmod* (*poly* (*pCons c cs*) *0*) *c*]
obtain *r* **where** *r*: ⋀*z*. *r* ≤ *cmod z* ⟹ *cmod* (*poly* (*pCons c cs*) *0*) ≤ *cmod* (*poly* (*pCons c cs*) *z*)
by *blast*
have *ath*: ⋀*z r*. *r* ≤ *cmod z* ∨ *cmod z* ≤ $|r|$
by *arith*
from *poly-minimum-modulus-disc*[*of* $|r|$ *pCons c cs*]
obtain *v* **where** *v*: ⋀*w*. *cmod w* ≤ $|r|$ ⟹ *cmod* (*poly* (*pCons c cs*) *v*) ≤ *cmod* (*poly* (*pCons c cs*) *w*)
by *blast*
have *cmod* (*poly* (*pCons c cs*) *v*) ≤ *cmod* (*poly* (*pCons c cs*) *z*) **if** *z*: *r* ≤ *cmod z* **for** *z*
using *v*[*of 0*] *r*[*OF z*] **by** *simp*
with *v ath*[*of r*] **show** *?thesis*
by *blast*
next
case *True*
with *pCons.hyps* **show** *?thesis*
by *simp*
qed
qed

Constant function (non-syntactic characterization).

definition *constant f* $\longleftrightarrow$ $(\forall\, x\ y.\ f\ x = f\ y)$

lemma *nonconstant-length*: $\neg$ *constant* (*poly p*) $\Longrightarrow$ *psize p* $\geq$ *2*
  by (*induct p*) (*auto simp*: *constant-def psize-def*)

lemma *poly-replicate-append*: *poly* (*monom 1 n* $*$ *p*) (*x*::'*a*::*comm-ring-1*) = *x*ˆ*n* $*$ *poly p x*
  by (*simp add*: *poly-monom*)

Decomposition of polynomial, skipping zero coefficients after the first.

lemma *poly-decompose-lemma*:
  assumes *nz*: $\neg$ ($\forall$ *z*. *z* $\neq$ *0* $\longrightarrow$ *poly p z* = (*0*::'*a*::*idom*))
  shows $\exists$ *k a q*. *a* $\neq$ *0* $\wedge$ *Suc* (*psize q* + *k*) = *psize p* $\wedge$ ($\forall$ *z*. *poly p z* = *z*ˆ*k* $*$ *poly* (*pCons a q*) *z*)
  unfolding *psize-def*
  using *nz*
proof (*induct p*)
  case *0*
  then show *?case* **by** *simp*
next
  case (*pCons c cs*)
  show *?case*
  proof (*cases c* = *0*)
    case *True*
    from *pCons.hyps pCons.prems True* **show** *?thesis*
      apply *auto*
      apply (*rule-tac x*=*k*+*1* **in** *exI*)
      apply (*rule-tac x*=*a* **in** *exI*)
      apply *clarsimp*
      apply (*rule-tac x*=*q* **in** *exI*)
      apply *auto*
      done
  next
    case *False*
    show *?thesis*
      apply (*rule exI*[**where** *x*=*0*])
      apply (*rule exI*[**where** *x*=*c*])
      apply (*auto simp*: *False*)
      done
  qed
qed

lemma *poly-decompose*:
  assumes *nc*: $\neg$ *constant* (*poly p*)
  shows $\exists$ *k a q*. *a* $\neq$ (*0*::'*a*::*idom*) $\wedge$ *k* $\neq$ *0* $\wedge$
        psize q + *k* + *1* = *psize p* $\wedge$
        ($\forall$ *z*. *poly p z* = *poly p 0* + *z*ˆ*k* $*$ *poly* (*pCons a q*) *z*)
  using *nc*
proof (*induct p*)

```
  case 0
  then show ?case
    by (simp add: constant-def)
next
  case (pCons c cs)
  have ¬ (∀ z. z ≠ 0 ⟶ poly cs z = 0)
  proof
    assume ∀ z. z ≠ 0 ⟶ poly cs z = 0
    then have poly (pCons c cs) x = poly (pCons c cs) y for x y
      by (cases x = 0) auto
    with pCons.prems show False
      by (auto simp add: constant-def)
  qed
  from poly-decompose-lemma[OF this]
  show ?case
    apply clarsimp
    apply (rule-tac x=k+1 in exI)
    apply (rule-tac x=a in exI)
    apply simp
    apply (rule-tac x=q in exI)
    apply (auto simp add: psize-def split: if-splits)
    done
qed
```

Fundamental theorem of algebra

```
lemma fundamental-theorem-of-algebra:
  assumes nc: ¬ constant (poly p)
  shows ∃ z::complex. poly p z = 0
  using nc
proof (induct psize p arbitrary: p rule: less-induct)
  case less
  let ?p = poly p
  let ?ths = ∃ z. ?p z = 0

  from nonconstant-length[OF less(2)] have n2: psize p ≥ 2 .
  from poly-minimum-modulus obtain c where c: ∀ w. cmod (?p c) ≤ cmod (?p
w)
    by blast

  show ?ths
  proof (cases ?p c = 0)
    case True
    then show ?thesis by blast
  next
    case False
    from poly-offset[of p c] obtain q where q: psize q = psize p ∀ x. poly q x =
?p (c + x)
      by blast
    have False if h: constant (poly q)
```

proof −
  from *q(2)* **have** *th*: ∀ *x*. *poly q* (*x* − *c*) = *?p x*
    by *auto*
  have *?p x* = *?p y* **for** *x y*
  proof −
    from *th* **have** *?p x* = *poly q* (*x* − *c*)
      by *auto*
    also have . . . = *poly q* (*y* − *c*)
      using *h* **unfolding** *constant-def* **by** *blast*
    also have . . . = *?p y*
      using *th* **by** *auto*
    finally show *?thesis* **.**
  qed
  with *less(2)* **show** *?thesis*
    unfolding *constant-def* **by** *blast*
qed
then have *qnc*: ¬ *constant* (*poly q*)
  by *blast*
from *q(2)* **have** *pqc0*: *?p c* = *poly q 0*
  by *simp*
from *c pqc0* **have** *cq0*: ∀ *w*. *cmod* (*poly q 0*) ≤ *cmod* (*?p w*)
  by *simp*
let *?a0* = *poly q 0*
from *False pqc0* **have** *a00*: *?a0* ≠ *0*
  by *simp*
from *a00* **have** *qr*: ∀ *z*. *poly q z* = *poly* (*smult* (*inverse ?a0*) *q*) *z* ∗ *?a0*
  by *simp*
let *?r* = *smult* (*inverse ?a0*) *q*
have *lgqr*: *psize q* = *psize ?r*
  using *a00*
  unfolding *psize-def degree-def*
  by (*simp add*: *poly-eq-iff*)
have *False* **if** *h*: ⋀*x y*. *poly ?r x* = *poly ?r y*
proof −
  have *poly q x* = *poly q y* **for** *x y*
  proof −
    from *qr*[*rule-format*, *of x*] **have** *poly q x* = *poly ?r x* ∗ *?a0*
      by *auto*
    also have . . . = *poly ?r y* ∗ *?a0*
      using *h* **by** *simp*
    also have . . . = *poly q y*
      using *qr*[*rule-format*, *of y*] **by** *simp*
    finally show *?thesis* **.**
  qed
  with *qnc* **show** *?thesis*
    unfolding *constant-def* **by** *blast*
qed
then have *rnc*: ¬ *constant* (*poly ?r*)
  unfolding *constant-def* **by** *blast*

from *qr*[*rule-format*, *of 0*] *a00* **have** *r01*: *poly ?r 0* = *1*
  by *auto*
have *mrmq-eq*: *cmod* (*poly ?r w*) < *1* ⟷ *cmod* (*poly q w*) < *cmod ?a0* **for** *w*
proof −
  have *cmod* (*poly ?r w*) < *1* ⟷ *cmod* (*poly q w* / *?a0*) < *1*
    using *qr*[*rule-format*, *of w*] *a00* **by** (*simp add*: *divide-inverse ac-simps*)
  also have ... ⟷ *cmod* (*poly q w*) < *cmod ?a0*
    using *a00* **unfolding** *norm-divide* **by** (*simp add*: *field-simps*)
  finally show *?thesis* **.**
qed
from *poly-decompose*[*OF rnc*] **obtain** *k a s* **where**
  kas: *a* ≠ *0 k* ≠ *0 psize s* + *k* + *1* = *psize ?r*
    ∀ *z*. *poly ?r z* = *poly ?r 0* + *z*ˆ*k*∗ *poly* (*pCons a s*) *z* **by** *blast*
have ∃ *w*. *cmod* (*poly ?r w*) < *1*
proof (*cases psize p* = *k* + *1*)
  case *True*
  with *kas*(*3*) *lgqr*[*symmetric*] *q*(*1*) **have** *s0*: *s* = *0*
    by *auto*
  have *hth*[*symmetric*]: *cmod* (*poly ?r w*) = *cmod* (*1* + *a* ∗ *w* ˆ *k*) **for** *w*
    using *kas*(*4*)[*rule-format*, *of w*] *s0 r01* **by** (*simp add*: *algebra-simps*)
  from *reduce-poly-simple*[*OF kas*(*1*,*2*)] **show** *?thesis*
    unfolding *hth* **by** *blast*
next
  case *False* **note** *kn* = *this*
  from *kn kas*(*3*) *q*(*1*) *lgqr* **have** *k1n*: *k* + *1* < *psize p*
    by *simp*
  have *th01*: ¬ *constant* (*poly* (*pCons 1* (*monom a* (*k* − *1*))))
    unfolding *constant-def poly-pCons poly-monom*
    using *kas*(*1*)
    apply *simp*
    apply (*rule exI*[**where** *x*=*0*])
    apply (*rule exI*[**where** *x*=*1*])
    apply *simp*
    done
  from *kas*(*1*) *kas*(*2*) **have** *th02*: *k* + *1* = *psize* (*pCons 1* (*monom a* (*k* − *1*)))
    by (*simp add*: *psize-def degree-monom-eq*)
  from *less*(*1*) [*OF k1n* [*simplified th02*] *th01*]
  obtain *w* **where** *w*: *1* + *w*ˆ*k* ∗ *a* = *0*
    unfolding *poly-pCons poly-monom*
    using *kas*(*2*) **by** (*cases k*) (*auto simp add*: *algebra-simps*)
  from *poly-bound-exists*[*of cmod w s*] **obtain** *m* **where**
    m: *m* > *0* ∀ *z*. *cmod z* ≤ *cmod w* ⟶ *cmod* (*poly s z*) ≤ *m* **by** *blast*
  have *w0*: *w* ≠ *0*
    using *kas*(*2*) *w* **by** (*auto simp add*: *power-0-left*)
  from *w* **have** (*1* + *w* ˆ *k* ∗ *a*) − *1* = *0* − *1*
    by *simp*
  then have *wm1*: *w*ˆ*k* ∗ *a* = − *1*
    by *simp*
  have *inv0*: *0* < *inverse* (*cmod w* ˆ (*k* + *1*) ∗ *m*)

using *norm-ge-zero*[*of w*] *w0 m*(*1*)
by (*simp add*: *inverse-eq-divide zero-less-mult-iff*)
with *real-lbound-gt-zero*[*OF zero-less-one*] **obtain** *t* **where**
t: *t* > *0 t* < *1 t* < *inverse* (*cmod w* ^ (*k* + *1*) ∗ *m*) **by** *blast*
let *?ct* = *complex-of-real t*
let *?w* = *?ct* ∗ *w*
have *1* + *?w*^*k* ∗ (*a* + *?w* ∗ *poly s ?w*) = *1* + *?ct*^*k* ∗ (*w*^*k* ∗ *a*) + *?w*^*k* ∗ *?w* ∗ *poly s ?w*
using *kas*(*1*) **by** (*simp add*: *algebra-simps power-mult-distrib*)
also have ... = *complex-of-real* (*1* − *t*^*k*) + *?w*^*k* ∗ *?w* ∗ *poly s ?w*
unfolding *wm1* **by** *simp*
finally have *cmod* (*1* + *?w*^*k* ∗ (*a* + *?w* ∗ *poly s ?w*)) =
cmod (*complex-of-real* (*1* − *t*^*k*) + *?w*^*k* ∗ *?w* ∗ *poly s ?w*)
by *metis*
with *norm-triangle-ineq*[*of complex-of-real* (*1* − *t*^*k*) *?w*^*k* ∗ *?w* ∗ *poly s ?w*]
have *th11*: *cmod* (*1* + *?w*^*k* ∗ (*a* + *?w* ∗ *poly s ?w*)) ≤ |*1* − *t*^*k*| + *cmod* (*?w*^*k* ∗ *?w* ∗ *poly s ?w*)
unfolding *norm-of-real* **by** *simp*
have *ath*: ⋀*x t*::*real*. *0* ≤ *x* ⟹ *x* < *t* ⟹ *t* ≤ *1* ⟹ |*1* − *t*| + *x* < *1*
by *arith*
have *t* ∗ *cmod w* ≤ *1* ∗ *cmod w*
apply (*rule mult-mono*)
using *t*(*1*,*2*)
apply *auto*
done
then have *tw*: *cmod ?w* ≤ *cmod w*
using *t*(*1*) **by** (*simp add*: *norm-mult*)
from *t inv0* **have** *t* ∗ (*cmod w* ^ (*k* + *1*) ∗ *m*) < *1*
by (*simp add*: *field-simps*)
with *zero-less-power*[*OF t*(*1*), *of k*] **have** *th30*: *t*^*k* ∗ (*t*∗ (*cmod w* ^ (*k* + *1*) ∗ *m*)) < *t*^*k* ∗ *1*
by *simp*
have *cmod* (*?w*^*k* ∗ *?w* ∗ *poly s ?w*) = *t*^*k* ∗ (*t*∗ (*cmod w* ^ (*k* + *1*) ∗ *cmod* (*poly s ?w*)))
using *w0 t*(*1*)
by (*simp add*: *algebra-simps power-mult-distrib norm-power norm-mult*)
then have *cmod* (*?w*^*k* ∗ *?w* ∗ *poly s ?w*) ≤ *t*^*k* ∗ (*t*∗ (*cmod w* ^ (*k* + *1*) ∗ *m*))
using *t*(*1*,*2*) *m*(*2*)[*rule-format*, *OF tw*] *w0*
by *auto*
with *th30* **have** *th120*: *cmod* (*?w*^*k* ∗ *?w* ∗ *poly s ?w*) < *t*^*k*
by *simp*
from *power-strict-mono*[*OF t*(*2*), *of k*] *t*(*1*) *kas*(*2*) **have** *th121*: *t*^*k* ≤ *1*
by *auto*
from *ath*[*OF norm-ge-zero*[*of ?w*^*k* ∗ *?w* ∗ *poly s ?w*] *th120 th121*]
have *th12*: |*1* − *t*^*k*| + *cmod* (*?w*^*k* ∗ *?w* ∗ *poly s ?w*) < *1* **.**
from *th11 th12* **have** *cmod* (*1* + *?w*^*k* ∗ (*a* + *?w* ∗ *poly s ?w*)) < *1*
by *arith*
then have *cmod* (*poly ?r ?w*) < *1*

```
      unfolding kas(4)[rule-format, of ?w] r01 by simp
    then show ?thesis
      by blast
  qed
  with cq0 q(2) show ?thesis
    unfolding mrmq-eq not-less[symmetric] by auto
 qed
qed
```

Alternative version with a syntactic notion of constant polynomial.

```
lemma fundamental-theorem-of-algebra-alt:
 assumes nc: ¬ (∃ a l. a ≠ 0 ∧ l = 0 ∧ p = pCons a l)
 shows ∃ z. poly p z = (0::complex)
 using nc
proof (induct p)
 case 0
 then show ?case by simp
next
 case (pCons c cs)
 show ?case
 proof (cases c = 0)
  case True
  then show ?thesis by auto
 next
  case False
  have ¬ constant (poly (pCons c cs))
  proof
   assume nc: constant (poly (pCons c cs))
   from nc[unfolded constant-def, rule-format, of 0]
   have ∀ w. w ≠ 0 ⟶ poly cs w = 0 by auto
   then have cs = 0
   proof (induct cs)
    case 0
    then show ?case by simp
   next
    case (pCons d ds)
    show ?case
    proof (cases d = 0)
     case True
     then show ?thesis
      using pCons.prems pCons.hyps by simp
    next
     case False
     from poly-bound-exists[of 1 ds] obtain m where
      m: m > 0 ∀ z. ∀ z. cmod z ≤ 1 ⟶ cmod (poly ds z) ≤ m by blast
     have dm: cmod d / m > 0
      using False m(1) by (simp add: field-simps)
     from real-lbound-gt-zero[OF dm zero-less-one]
     obtain x where x: x > 0 x < cmod d / m x < 1
```

```
          by blast
        let ?x = complex-of-real x
        from x have cx: ?x ≠ 0 cmod ?x ≤ 1
          by simp-all
        from pCons.prems[rule-format, OF cx(1)]
        have cth: cmod (?x*poly ds ?x) = cmod d
          by (simp add: eq-diff-eq[symmetric])
        from m(2)[rule-format, OF cx(2)] x(1)
        have th0: cmod (?x*poly ds ?x) ≤ x*m
          by (simp add: norm-mult)
        from x(2) m(1) have x * m < cmod d
          by (simp add: field-simps)
        with th0 have cmod (?x*poly ds ?x) ≠ cmod d
          by auto
        with cth show ?thesis
          by blast
      qed
    qed
    then show False
      using pCons.prems False by blast
  qed
  then show ?thesis
    by (rule fundamental-theorem-of-algebra)
 qed
qed
```

51.4 Nullstellensatz, degrees and divisibility of polynomials

```
lemma nullstellensatz-lemma:
  fixes p :: complex poly
  assumes ∀ x. poly p x = 0 ⟶ poly q x = 0
    and degree p = n
    and n ≠ 0
  shows p dvd (q ^ n)
  using assms
proof (induct n arbitrary: p q rule: nat-less-induct)
  fix n :: nat
  fix p q :: complex poly
  assume IH: ∀ m<n. ∀ p q.
               (∀ x. poly p x = (0::complex) ⟶ poly q x = 0) ⟶
               degree p = m ⟶ m ≠ 0 ⟶ p dvd (q ^ m)
    and pq0: ∀ x. poly p x = 0 ⟶ poly q x = 0
    and dpn: degree p = n
    and n0: n ≠ 0
  from dpn n0 have pne: p ≠ 0 by auto
  show p dvd (q ^ n)
  proof (cases ∃ a. poly p a = 0)
    case True
    then obtain a where a: poly p a = 0 ..
```

```
have ?thesis if oa: order a p ≠ 0
proof -
  let ?op = order a p
  from pne have ap: ([:- a, 1:] ^ ?op) dvd p ¬ [:- a, 1:] ^ (Suc ?op) dvd p
    using order by blast+
  note oop = order-degree[OF pne, unfolded dpn]
  show ?thesis
  proof (cases q = 0)
    case True
    with n0 show ?thesis by (simp add: power-0-left)
  next
    case False
    from pq0[rule-format, OF a, unfolded poly-eq-0-iff-dvd]
    obtain r where r: q = [:- a, 1:] * r by (rule dvdE)
    from ap(1) obtain s where s: p = [:- a, 1:] ^ ?op * s
      by (rule dvdE)
    have sne: s ≠ 0
      using s pne by auto
    show ?thesis
    proof (cases degree s = 0)
      case True
      then obtain k where kpn: s = [:k:]
        by (cases s) (auto split: if-splits)
      from sne kpn have k: k ≠ 0 by simp
      let ?w = ([:1/k:] * ([:-a,1:] ^ (n - ?op))) * (r ^ n)
      have q ^ n = p * ?w
        apply (subst r)
        apply (subst s)
        apply (subst kpn)
        using k oop [of a]
        apply (subst power-mult-distrib)
        apply simp
        apply (subst power-add [symmetric])
        apply simp
        done
      then show ?thesis
        unfolding dvd-def by blast
    next
      case False
      with sne dpn s oa have dsn: degree s < n
        apply auto
        apply (erule ssubst)
        apply (simp add: degree-mult-eq degree-linear-power)
        done
      have poly r x = 0 if h: poly s x = 0 for x
      proof -
        have xa: x ≠ a
        proof
          assume x = a
```

```
            from h[unfolded this poly-eq-0-iff-dvd] obtain u where u: s = [:- a,
1:] * u
              by (rule dvdE)
            have p = [:- a, 1:] ^ (Suc ?op) * u
              apply (subst s)
              apply (subst u)
              apply (simp only: power-Suc ac-simps)
              done
            with ap(2)[unfolded dvd-def] show False
              by blast
          qed
          from h have poly p x = 0
            by (subst s) simp
          with pq0 have poly q x = 0
            by blast
          with r xa show ?thesis
            by auto
        qed
        with IH[rule-format, OF dsn, of s r] False have s dvd (r ^ (degree s))
          by blast
        then obtain u where u: r ^ (degree s) = s * u ..
        then have u': ⋀x. poly s x * poly u x = poly r x ^ degree s
          by (simp only: poly-mult[symmetric] poly-power[symmetric])
        let ?w = (u * ([:-a,1:] ^ (n - ?op))) * (r ^ (n - degree s))
        from oop[of a] dsn have q ^ n = p * ?w
          apply -
          apply (subst s)
          apply (subst r)
          apply (simp only: power-mult-distrib)
          apply (subst mult.assoc [where b=s])
          apply (subst mult.assoc [where a=u])
          apply (subst mult.assoc [where b=u, symmetric])
          apply (subst u [symmetric])
          apply (simp add: ac-simps power-add [symmetric])
          done
        then show ?thesis
          unfolding dvd-def by blast
      qed
    qed
  qed
  then show ?thesis
    using a order-root pne by blast
next
  case False
  with fundamental-theorem-of-algebra-alt[of p]
  obtain c where ccs: c ≠ 0 p = pCons c 0
    by blast
  then have pp: poly p x = c for x
    by simp
```

```
    let ?w = [:1/c:] * (q ^ n)
    from ccs have (q ^ n) = (p * ?w)
      by simp
    then show ?thesis
      unfolding dvd-def by blast
  qed
qed

lemma nullstellensatz-univariate:
  (∀x. poly p x = (0::complex) ⟶ poly q x = 0) ⟷
    p dvd (q ^ (degree p)) ∨ (p = 0 ∧ q = 0)
proof −
  consider p = 0 | p ≠ 0 degree p = 0 | n where p ≠ 0 degree p = Suc n
    by (cases degree p) auto
  then show ?thesis
  proof cases
    case p: 1
    then have eq: (∀x. poly p x = (0::complex) ⟶ poly q x = 0) ⟷ q = 0
      by (auto simp add: poly-all-0-iff-0)
    {
      assume p dvd (q ^ (degree p))
      then obtain r where r: q ^ (degree p) = p * r ..
      from r p have False by simp
    }
    with eq p show ?thesis by blast
  next
    case dp: 2
    then obtain k where k: p = [:k:] k ≠ 0
      by (cases p) (simp split: if-splits)
    then have th1: ∀x. poly p x ≠ 0
      by simp
    from k dp(2) have q ^ (degree p) = p * [:1/k:]
      by (simp add: one-poly-def)
    then have th2: p dvd (q ^ (degree p)) ..
    from dp(1) th1 th2 show ?thesis
      by blast
  next
    case dp: 3
    have False if dvd: p dvd (q ^ (Suc n)) and h: poly p x = 0 poly q x ≠ 0 for x
    proof −
      from dvd obtain u where u: q ^ (Suc n) = p * u ..
      from h have poly (q ^ (Suc n)) x ≠ 0
        by simp
      with u h(1) show ?thesis
        by (simp only: poly-mult) simp
    qed
    with dp nullstellensatz-lemma[of p q degree p] show ?thesis
      by auto
  qed
```

qed

Useful lemma

lemma *constant-degree*:
fixes *p* :: *'a*::{*idom*,*ring-char-0*} *poly*
shows *constant* (*poly p*) ⟷ *degree p* = *0* (**is** *?lhs* = *?rhs*)
proof
show *?rhs* **if** *?lhs*
proof −
from *that*[*unfolded constant-def*, *rule-format*, *of* - *0*]
have *th*: *poly p* = *poly* [:*poly p 0*:]
by *auto*
then have *p* = [:*poly p 0*:]
by (*simp add*: *poly-eq-poly-eq-iff*)
then have *degree p* = *degree* [:*poly p 0*:]
by *simp*
then show *?thesis*
by *simp*
qed
show *?lhs* **if** *?rhs*
proof −
from *that* **obtain** *k* **where** *p* = [:*k*:]
by (*cases p*) (*simp split*: *if-splits*)
then show *?thesis*
unfolding *constant-def* **by** *auto*
qed
qed

Arithmetic operations on multivariate polynomials.

lemma *mpoly-base-conv*:
fixes *x* :: *'a*::*comm-ring-1*
shows *0* = *poly 0 x c* = *poly* [:*c*:] *x x* = *poly* [:*0*,*1*:] *x*
by *simp-all*

lemma *mpoly-norm-conv*:
fixes *x* :: *'a*::*comm-ring-1*
shows *poly* [:*0*:] *x* = *poly 0 x poly* [:*poly 0 y*:] *x* = *poly 0 x*
by *simp-all*

lemma *mpoly-sub-conv*:
fixes *x* :: *'a*::*comm-ring-1*
shows *poly p x* − *poly q x* = *poly p x* + −*1* ∗ *poly q x*
by *simp*

lemma *poly-pad-rule*: *poly p x* = *0* ⟹ *poly* (*pCons 0 p*) *x* = *0*
by *simp*

lemma *poly-cancel-eq-conv*:
fixes *x* :: *'a*::*field*

shows $x = 0 \Longrightarrow a \neq 0 \Longrightarrow y = 0 \longleftrightarrow a * y - b * x = 0$
by *auto*

lemma *poly-divides-pad-rule*:
fixes p:: ('*a*::*comm-ring-1*) *poly*
assumes *pq*: *p dvd q*
shows *p dvd* (*pCons 0 q*)
proof −
have *pCons 0 q* = $q *$ [:*0*,*1*:] **by** *simp*
then have *q dvd* (*pCons 0 q*) **..**
with *pq* **show** *?thesis* **by** (*rule dvd-trans*)
qed

lemma *poly-divides-conv0*:
fixes p:: '*a*::*field poly*
assumes *lgpq*: *degree q* < *degree p*
and *lq*: $p \neq 0$
shows *p dvd q* ⟷ $q = 0$ (**is** *?lhs* ⟷ *?rhs*)
proof
assume *?rhs*
then have $q = p * 0$ **by** *simp*
then show *?lhs* **..**
next
assume *l*: *?lhs*
show *?rhs*
proof (*cases* $q = 0$)
case *True*
then show *?thesis* **by** *simp*
next
assume *q0*: $q \neq 0$
from *l q0* **have** *degree p* ≤ *degree q*
by (*rule dvd-imp-degree-le*)
with *lgpq* **show** *?thesis* **by** *simp*
qed
qed

lemma *poly-divides-conv1*:
fixes p :: '*a*::*field poly*
assumes *a0*: $a \neq 0$
and *pp'*: *p dvd p'*
and *qrp'*: *smult a q* − $p' = r$
shows *p dvd q* ⟷ *p dvd r* (**is** *?lhs* ⟷ *?rhs*)
proof
from *pp'* **obtain** *t* **where** *t*: $p' = p * t$ **..**
show *?rhs* **if** *?lhs*
proof −
from *that* **obtain** *u* **where** *u*: $q = p * u$ **..**
have $r = p *$ (*smult a u* − *t*)
using *u qrp'* [*symmetric*] *t* **by** (*simp add*: *algebra-simps*)

then show *?thesis* **..**
qed
show *?lhs* **if** *?rhs*
proof –
from *that* **obtain** *u* **where** *u*: $r = p * u$ **..**
from *u* [*symmetric*] *t qrp'* [*symmetric*] *a0*
have $q = p * smult\ (1/a)\ (u + t)$
by (*simp add*: *algebra-simps*)
then show *?thesis* **..**
qed
qed

lemma *basic-cqe-conv1*:
$(\exists x.\ poly\ p\ x = 0 \wedge poly\ 0\ x \neq 0) \longleftrightarrow False$
$(\exists x.\ poly\ 0\ x \neq 0) \longleftrightarrow False$
$(\exists x.\ poly\ [:c:]\ x \neq 0) \longleftrightarrow c \neq 0$
$(\exists x.\ poly\ 0\ x = 0) \longleftrightarrow True$
$(\exists x.\ poly\ [:c:]\ x = 0) \longleftrightarrow c = 0$
by *simp-all*

lemma *basic-cqe-conv2*:
assumes *l*: $p \neq 0$
shows $\exists x.\ poly\ (pCons\ a\ (pCons\ b\ p))\ x = (0::complex)$
proof –
have *False* **if** $h \neq 0$ $t = 0$ **and** $pCons\ a\ (pCons\ b\ p) = pCons\ h\ t$ **for** *h t*
using *l that* **by** *simp*
then have *th*: $\neg (\exists\ h\ t.\ h \neq 0 \wedge t = 0 \wedge pCons\ a\ (pCons\ b\ p) = pCons\ h\ t)$
by *blast*
from *fundamental-theorem-of-algebra-alt*[*OF th*] **show** *?thesis*
by *auto*
qed

lemma *basic-cqe-conv-2b*: $(\exists x.\ poly\ p\ x \neq (0::complex)) \longleftrightarrow p \neq 0$
by (*metis poly-all-0-iff-0*)

lemma *basic-cqe-conv3*:
fixes *p q* :: *complex poly*
assumes *l*: $p \neq 0$
shows $(\exists x.\ poly\ (pCons\ a\ p)\ x = 0 \wedge poly\ q\ x \neq 0) \longleftrightarrow \neg (pCons\ a\ p)\ dvd\ (q\ \hat{}\ psize\ p)$
proof –
from *l* **have** *dp*: $degree\ (pCons\ a\ p) = psize\ p$
by (*simp add*: *psize-def*)
from *nullstellensatz-univariate*[*of pCons a p q*] *l*
show *?thesis*
by (*metis dp pCons-eq-0-iff*)
qed

lemma *basic-cqe-conv4*:

fixes *p q* :: *complex poly*
assumes *h*: $\bigwedge x$. *poly* (*q* ^ *n*) *x* = *poly r x*
shows *p dvd* (*q* ^ *n*) ⟷ *p dvd r*
proof −
from *h* **have** *poly* (*q* ^ *n*) = *poly r*
by *auto*
then have (*q* ^ *n*) = *r*
by (*simp add*: *poly-eq-poly-eq-iff*)
then show *p dvd* (*q* ^ *n*) ⟷ *p dvd r*
by *simp*
qed

lemma *poly-const-conv*:
fixes *x* :: *'a*::*comm-ring-1*
shows *poly* [:*c*:] *x* = *y* ⟷ *c* = *y*
by *simp*

end

52 Lexical order on functions

theory *Fun-Lexorder*
imports *Main*
begin

definition *less-fun* :: (*'a*::*linorder* ⇒ *'b*::*linorder*) ⇒ (*'a* ⇒ *'b*) ⇒ *bool*
where
less-fun f g ⟷ (∃ *k*. *f k* < *g k* ∧ (∀ *k'* < *k*. *f k'* = *g k'*))

lemma *less-funI*:
assumes ∃ *k*. *f k* < *g k* ∧ (∀ *k'* < *k*. *f k'* = *g k'*)
shows *less-fun f g*
using *assms* **by** (*simp add*: *less-fun-def*)

lemma *less-funE*:
assumes *less-fun f g*
obtains *k* **where** *f k* < *g k* **and** $\bigwedge k'$. *k'* < *k* ⟹ *f k'* = *g k'*
using *assms* **unfolding** *less-fun-def* **by** *blast*

lemma *less-fun-asym*:
assumes *less-fun f g*
shows ¬ *less-fun g f*
proof
from *assms* **obtain** *k1* **where** *k1*: *f k1* < *g k1* $\bigwedge k'$. *k'* < *k1* ⟹ *f k'* = *g k'*
by (*blast elim*!: *less-funE*)
assume *less-fun g f* **then obtain** *k2* **where** *k2*: *g k2* < *f k2* $\bigwedge k'$. *k'* < *k2* ⟹
g k' = *f k'*
by (*blast elim*!: *less-funE*)
show *False* **proof** (*cases k1 k2 rule*: *linorder-cases*)

case *equal* **with** *k1 k2* **show** *False* **by** *simp*
next
case *less* **with** *k2* **have** $g\ k1 = f\ k1$ **by** *simp*
with *k1* **show** *False* **by** *simp*
next
case *greater* **with** *k1* **have** $f\ k2 = g\ k2$ **by** *simp*
with *k2* **show** *False* **by** *simp*
qed
qed

lemma *less-fun-irrefl*:
$\neg$ *less-fun f f*
proof
assume *less-fun f f*
then obtain *k* **where** *k*: $f\ k < f\ k$
by (*blast elim*!: *less-funE*)
then show *False* **by** *simp*
qed

lemma *less-fun-trans*:
assumes *less-fun f g* **and** *less-fun g h*
shows *less-fun f h*
proof (*rule less-funI*)
from ⟨*less-fun f g*⟩ **obtain** *k1* **where** *k1*: $f\ k1 < g\ k1\ \bigwedge k'.\ k' < k1 \Longrightarrow f\ k' = g\ k'$
by (*blast elim*!: *less-funE*)
from ⟨*less-fun g h*⟩ **obtain** *k2* **where** *k2*: $g\ k2 < h\ k2\ \bigwedge k'.\ k' < k2 \Longrightarrow g\ k' = h\ k'$
by (*blast elim*!: *less-funE*)
show $\exists k.\ f\ k < h\ k \wedge (\forall k'<k.\ f\ k' = h\ k')$
proof (*cases k1 k2 rule*: *linorder-cases*)
case *equal* **with** *k1 k2* **show** *?thesis* **by** (*auto simp add*: *exI* [*of* - *k2*])
next
case *less* **with** *k2* **have** $g\ k1 = h\ k1\ \bigwedge k'.\ k' < k1 \Longrightarrow g\ k' = h\ k'$ **by** *simp-all*
with *k1* **show** *?thesis* **by** (*auto intro*: *exI* [*of* - *k1*])
next
case *greater* **with** *k1* **have** $f\ k2 = g\ k2\ \bigwedge k'.\ k' < k2 \Longrightarrow f\ k' = g\ k'$ **by** *simp-all*
with *k2* **show** *?thesis* **by** (*auto intro*: *exI* [*of* - *k2*])
qed
qed

lemma *order-less-fun*:
class.order $(\lambda f\ g.$ *less-fun f g* $\vee f = g)$ *less-fun*
by (*rule order-strictI*) (*auto intro*: *less-fun-trans intro*!: *less-fun-irrefl less-fun-asym*)

lemma *less-fun-trichotomy*:
assumes *finite* $\{k.\ f\ k \neq g\ k\}$
shows *less-fun f g* $\vee f = g \vee$ *less-fun g f*

proof –
  { def $K \equiv \{k.\ f\ k \neq g\ k\}$
    assume $f \neq g$
    then obtain k' **where** $f\ k' \neq g\ k'$ **by** *auto*
    then have [*simp*]: $K \neq \{\}$ **by** (*auto simp add*: *K-def*)
    with *assms* **have** [*simp*]: *finite* K **by** (*simp add*: *K-def*)
    def $q \equiv$ *Min* K
    then have $q \in K$ **and** $\bigwedge k.\ k \in K \Longrightarrow k \geq q$ **by** *auto*
    then have $\bigwedge k.\ \neg\ k \geq q \Longrightarrow k \notin K$ **by** *blast*
    then have $*$: $\bigwedge k.\ k < q \Longrightarrow f\ k = g\ k$ **by** (*simp add*: *K-def*)
    from ‹$q \in K$› **have** $f\ q \neq g\ q$ **by** (*simp add*: *K-def*)
    then have $f\ q < g\ q \vee f\ q > g\ q$ **by** *auto*
    with $*$ **have** *less-fun f g* $\vee$ *less-fun g f*
      by (*auto intro*!: *less-funI*)
  } then show *?thesis* **by** *blast*
qed

end

53 Big sum and product over function bodies

theory *Groups-Big-Fun*
imports
  Main
begin

53.1 Abstract product

no-notation *times* (**infixl** $*$ *70*)
no-notation *Groups.one* (*1*)

locale *comm-monoid-fun* = *comm-monoid*
begin

definition G :: $('b \Rightarrow 'a) \Rightarrow 'a$
where
  expand-set: $G\ g$ = *comm-monoid-set.F f 1 g* $\{a.\ g\ a \neq 1\}$

interpretation F: *comm-monoid-set f 1*
  ..

lemma *expand-superset*:
  assumes *finite* A **and** $\{a.\ g\ a \neq 1\} \subseteq A$
  shows $G\ g = F.F\ g\ A$
  apply (*simp add*: *expand-set*)
  apply (*rule F.same-carrierI* [*of A*])
  apply (*simp-all add*: *assms*)
  done

lemma *conditionalize*:
  assumes *finite A*
  shows $F.F\ g\ A = G\ (\lambda a.\ if\ a \in A\ then\ g\ a\ else\ 1)$
  using *assms*
  apply (*simp add*: *expand-set*)
  apply (*rule F.same-carrierI* [*of A*])
  apply *auto*
  done

lemma *neutral* [*simp*]:
  $G\ (\lambda a.\ 1) = 1$
  by (*simp add*: *expand-set*)

lemma *update* [*simp*]:
  assumes *finite* $\{a.\ g\ a \neq 1\}$
  assumes $g\ a = 1$
  shows $G\ (g(a := b)) = b * G\ g$
proof (*cases* $b = 1$)
  case *True* **with** ‹$g\ a = 1$› **show** *?thesis*
    by (*simp add*: *expand-set*) (*rule F.cong*, *auto*)
next
  case *False*
  moreover have $\{a'.\ a' \neq a \longrightarrow g\ a' \neq 1\} = insert\ a\ \{a.\ g\ a \neq 1\}$
    by *auto*
  moreover from ‹$g\ a = 1$› **have** $a \notin \{a.\ g\ a \neq 1\}$
    by *simp*
  moreover have $F.F\ (\lambda a'.\ if\ a' = a\ then\ b\ else\ g\ a')\ \{a.\ g\ a \neq 1\} = F.F\ g\ \{a.\ g\ a \neq 1\}$
    by (*rule F.cong*) (*auto simp add*: ‹$g\ a = 1$›)
  ultimately show *?thesis* **using** ‹*finite* $\{a.\ g\ a \neq 1\}$› **by** (*simp add*: *expand-set*)
qed

lemma *infinite* [*simp*]:
  $\neg\ finite\ \{a.\ g\ a \neq 1\} \Longrightarrow G\ g = 1$
  by (*simp add*: *expand-set*)

lemma *cong*:
  assumes $\bigwedge a.\ g\ a = h\ a$
  shows $G\ g = G\ h$
  using *assms* **by** (*simp add*: *expand-set*)

lemma *strong-cong* [*cong*]:
  assumes $\bigwedge a.\ g\ a = h\ a$
  shows $G\ (\lambda a.\ g\ a) = G\ (\lambda a.\ h\ a)$
  using *assms* **by** (*fact cong*)

lemma *not-neutral-obtains-not-neutral*:
  assumes $G\ g \neq 1$
  obtains *a* **where** $g\ a \neq 1$

using *assms* **by** (*auto elim*: *F.not-neutral-contains-not-neutral simp add*: *expand-set*)

lemma *reindex-cong*:
assumes *bij l*
assumes $g \circ l = h$
shows $G\ g = G\ h$
proof –
from *assms* **have** *unfold*: $h = g \circ l$ **by** *simp*
from ‹*bij l*› **have** *inj l* **by** (*rule bij-is-inj*)
then have *inj-on l* $\{a.\ h\ a \neq 1\}$ **by** (*rule subset-inj-on*) *simp*
moreover from ‹*bij l*› **have** $\{a.\ g\ a \neq 1\} = l$ ‘ $\{a.\ h\ a \neq 1\}$
by (*auto simp add*: *image-Collect unfold elim*: *bij-pointE*)
moreover have $\bigwedge x.\ x \in \{a.\ h\ a \neq 1\} \Longrightarrow g\ (l\ x) = h\ x$
by (*simp add*: *unfold*)
ultimately have $F.F\ g\ \{a.\ g\ a \neq 1\} = F.F\ h\ \{a.\ h\ a \neq 1\}$
by (*rule F.reindex-cong*)
then show *?thesis* **by** (*simp add*: *expand-set*)
qed

lemma *distrib*:
assumes *finite* $\{a.\ g\ a \neq 1\}$ **and** *finite* $\{a.\ h\ a \neq 1\}$
shows $G\ (\lambda a.\ g\ a * h\ a) = G\ g * G\ h$
proof –
from *assms* **have** *finite* $(\{a.\ g\ a \neq 1\} \cup \{a.\ h\ a \neq 1\})$ **by** *simp*
moreover have $\{a.\ g\ a * h\ a \neq 1\} \subseteq \{a.\ g\ a \neq 1\} \cup \{a.\ h\ a \neq 1\}$
by *auto* (*drule sym*, *simp*)
ultimately show *?thesis*
using *assms*
by (*simp add*: *expand-superset* [*of* $\{a.\ g\ a \neq 1\} \cup \{a.\ h\ a \neq 1\}$] *F.distrib*)
qed

lemma *commute*:
assumes *finite C*
assumes *subset*: $\{a.\ \exists b.\ g\ a\ b \neq 1\} \times \{b.\ \exists a.\ g\ a\ b \neq 1\} \subseteq C$ (**is** $?A \times ?B \subseteq C$)
shows $G\ (\lambda a.\ G\ (g\ a)) = G\ (\lambda b.\ G\ (\lambda a.\ g\ a\ b))$
proof –
from ‹*finite C*› *subset*
have *finite* $(\{a.\ \exists b.\ g\ a\ b \neq 1\} \times \{b.\ \exists a.\ g\ a\ b \neq 1\})$
by (*rule rev-finite-subset*)
then have *fins*:
finite $\{b.\ \exists a.\ g\ a\ b \neq 1\}$ *finite* $\{a.\ \exists b.\ g\ a\ b \neq 1\}$
by (*auto simp add*: *finite-cartesian-product-iff*)
have *subsets*: $\bigwedge a.\ \{b.\ g\ a\ b \neq 1\} \subseteq \{b.\ \exists a.\ g\ a\ b \neq 1\}$
$\bigwedge b.\ \{a.\ g\ a\ b \neq 1\} \subseteq \{a.\ \exists b.\ g\ a\ b \neq 1\}$
$\{a.\ F.F\ (g\ a)\ \{b.\ \exists a.\ g\ a\ b \neq 1\} \neq 1\} \subseteq \{a.\ \exists b.\ g\ a\ b \neq 1\}$
$\{a.\ F.F\ (\lambda aa.\ g\ aa\ a)\ \{a.\ \exists b.\ g\ a\ b \neq 1\} \neq 1\} \subseteq \{b.\ \exists a.\ g\ a\ b \neq 1\}$
by (*auto elim*: *F.not-neutral-contains-not-neutral*)
from *F.commute* **have**

```
    F.F (λa. F.F (g a) {b. ∃a. g a b ≠ 1}) {a. ∃b. g a b ≠ 1} =
      F.F (λb. F.F (λa. g a b) {a. ∃b. g a b ≠ 1}) {b. ∃a. g a b ≠ 1} .
  with subsets fins have G (λa. F.F (g a) {b. ∃a. g a b ≠ 1}) =
    G (λb. F.F (λa. g a b) {a. ∃b. g a b ≠ 1})
    by (auto simp add: expand-superset [of {b. ∃a. g a b ≠ 1}]
      expand-superset [of {a. ∃b. g a b ≠ 1}])
  with subsets fins show ?thesis
    by (auto simp add: expand-superset [of {b. ∃a. g a b ≠ 1}]
      expand-superset [of {a. ∃b. g a b ≠ 1}])
qed

lemma cartesian-product:
  assumes finite C
  assumes subset: {a. ∃b. g a b ≠ 1} × {b. ∃a. g a b ≠ 1} ⊆ C (is ?A × ?B ⊆
C)
  shows G (λa. G (g a)) = G (λ(a, b). g a b)
proof −
  from subset ‹finite C› have fin-prod: finite (?A × ?B)
    by (rule finite-subset)
  from fin-prod have finite ?A and finite ?B
    by (auto simp add: finite-cartesian-product-iff)
  have ∗: G (λa. G (g a)) =
    (F.F (λa. F.F (g a) {b. ∃a. g a b ≠ 1}) {a. ∃b. g a b ≠ 1})
    apply (subst expand-superset [of ?B])
    apply (rule ‹finite ?B›)
    apply auto
    apply (subst expand-superset [of ?A])
    apply (rule ‹finite ?A›)
    apply auto
    apply (erule F.not-neutral-contains-not-neutral)
    apply auto
    done
  have {p. (case p of (a, b) ⇒ g a b) ≠ 1} ⊆ ?A × ?B
    by auto
  with subset have ∗∗: {p. (case p of (a, b) ⇒ g a b) ≠ 1} ⊆ C
    by blast
  show ?thesis
    apply (simp add: ∗)
    apply (simp add: F.cartesian-product)
    apply (subst expand-superset [of C])
    apply (rule ‹finite C›)
    apply (simp-all add: ∗∗)
    apply (rule F.same-carrierI [of C])
    apply (rule ‹finite C›)
    apply (simp-all add: subset)
    apply auto
    done
qed
```

lemma *cartesian-product2*:
assumes *fin*: *finite D*
assumes *subset*: $\{(a, b). \exists c. g\ a\ b\ c \neq 1\} \times \{c. \exists a\ b. g\ a\ b\ c \neq 1\} \subseteq D$ (**is** $?AB \times ?C \subseteq D$)
shows $G\ (\lambda(a, b). G\ (g\ a\ b)) = G\ (\lambda(a, b, c). g\ a\ b\ c)$
proof −
have *bij*: $bij\ (\lambda(a, b, c). ((a, b), c))$
by (*auto intro*!: *bijI injI simp add*: *image-def*)
have $\{p. \exists c. g\ (fst\ p)\ (snd\ p)\ c \neq 1\} \times \{c. \exists p. g\ (fst\ p)\ (snd\ p)\ c \neq 1\} \subseteq D$
by *auto* (*insert subset*, *blast*)
with *fin* **have** $G\ (\lambda p. G\ (g\ (fst\ p)\ (snd\ p))) = G\ (\lambda(p, c). g\ (fst\ p)\ (snd\ p)\ c)$
by (*rule cartesian-product*)
then have $G\ (\lambda(a, b). G\ (g\ a\ b)) = G\ (\lambda((a, b), c). g\ a\ b\ c)$
by (*auto simp add*: *split-def*)
also have $G\ (\lambda((a, b), c). g\ a\ b\ c) = G\ (\lambda(a, b, c). g\ a\ b\ c)$
using *bij* **by** (*rule reindex-cong* [*of* $\lambda(a, b, c). ((a, b), c)$]) (*simp add*: *fun-eq-iff*)
finally show *?thesis* **.**
qed

lemma *delta* [*simp*]:
$G\ (\lambda b. \text{if } b = a \text{ then } g\ b \text{ else } 1) = g\ a$
proof −
have $\{b. (\text{if } b = a \text{ then } g\ b \text{ else } 1) \neq 1\} \subseteq \{a\}$ **by** *auto*
then show *?thesis* **by** (*simp add*: *expand-superset* [*of* $\{a\}$])
qed

lemma *delta'* [*simp*]:
$G\ (\lambda b. \text{if } a = b \text{ then } g\ b \text{ else } 1) = g\ a$
proof −
have $(\lambda b. \text{if } a = b \text{ then } g\ b \text{ else } 1) = (\lambda b. \text{if } b = a \text{ then } g\ b \text{ else } 1)$
by (*simp add*: *fun-eq-iff*)
then have $G\ (\lambda b. \text{if } a = b \text{ then } g\ b \text{ else } 1) = G\ (\lambda b. \text{if } b = a \text{ then } g\ b \text{ else } 1)$
by (*simp cong del*: *strong-cong*)
then show *?thesis* **by** *simp*
qed

end

notation *times* (**infixl** ∗ *70*)
notation *Groups.one* (*1*)

53.2 Concrete sum

context *comm-monoid-add*
begin

sublocale *Sum-any*: *comm-monoid-fun plus 0*
defines
Sum-any = *Sum-any.G*

rewrites
  comm-monoid-set.F plus 0 = setsum
proof –
  show *comm-monoid-fun plus 0* **..**
  then interpret *Sum-any*: *comm-monoid-fun plus 0* **.**
  from *setsum-def* **show** *comm-monoid-set.F plus 0 = setsum* **by** (*auto intro*: *sym*)
qed

end

syntax (*ASCII*)
  -Sum-any :: *pttrn* ⇒ *'a* ⇒ *'a*::*comm-monoid-add*   ((*3SUM* -. -) [*0*, *10*] *10*)
syntax
  -Sum-any :: *pttrn* ⇒ *'a* ⇒ *'a*::*comm-monoid-add*   ((*3*$\sum$ -. -) [*0*, *10*] *10*)
translations
  $\sum$ *a*. *b* ⇌ *CONST Sum-any* (λ*a*. *b*)

lemma *Sum-any-left-distrib*:
  fixes *r* :: *'a* :: *semiring-0*
  assumes *finite* {*a*. *g a* ≠ *0*}
  shows *Sum-any g* ∗ *r* = ($\sum$ *n*. *g n* ∗ *r*)
proof –
  note *assms*
  moreover have {*a*. *g a* ∗ *r* ≠ *0*} ⊆ {*a*. *g a* ≠ *0*} **by** *auto*
  ultimately show *?thesis*
    by (*simp add*: *setsum-left-distrib Sum-any.expand-superset* [*of* {*a*. *g a* ≠ *0*}])
qed

lemma *Sum-any-right-distrib*:
  fixes *r* :: *'a* :: *semiring-0*
  assumes *finite* {*a*. *g a* ≠ *0*}
  shows *r* ∗ *Sum-any g* = ($\sum$ *n*. *r* ∗ *g n*)
proof –
  note *assms*
  moreover have {*a*. *r* ∗ *g a* ≠ *0*} ⊆ {*a*. *g a* ≠ *0*} **by** *auto*
  ultimately show *?thesis*
    by (*simp add*: *setsum-right-distrib Sum-any.expand-superset* [*of* {*a*. *g a* ≠ *0*}])
qed

lemma *Sum-any-product*:
  fixes *f g* :: *'b* ⇒ *'a*::*semiring-0*
  assumes *finite* {*a*. *f a* ≠ *0*} **and** *finite* {*b*. *g b* ≠ *0*}
  shows *Sum-any f* ∗ *Sum-any g* = ($\sum$ *a*. $\sum$ *b*. *f a* ∗ *g b*)
proof –
  have *subset-f*: {*a*. ($\sum$ *b*. *f a* ∗ *g b*) ≠ *0*} ⊆ {*a*. *f a* ≠ *0*}
    by *rule* (*simp*, *rule*, *auto*)
  moreover have *subset-g*: ⋀*a*. {*b*. *f a* ∗ *g b* ≠ *0*} ⊆ {*b*. *g b* ≠ *0*}
    by *rule* (*simp*, *rule*, *auto*)

ultimately show *?thesis* **using** *assms*
by (*auto simp add*: *Sum-any.expand-set* [*of f*] *Sum-any.expand-set* [*of g*]
Sum-any.expand-superset [*of* {*a*. *f a* ≠ *0*}] *Sum-any.expand-superset* [*of* {*b*. *g b* ≠ *0*}]
setsum-product)
qed

lemma *Sum-any-eq-zero-iff* [*simp*]:
fixes *f* :: *'a* ⇒ *nat*
assumes *finite* {*a*. *f a* ≠ *0*}
shows *Sum-any f* = *0* ⟷ *f* = (λ-. *0*)
using *assms* **by** (*simp add*: *Sum-any.expand-set fun-eq-iff*)

53.3 Concrete product

context *comm-monoid-mult*
begin

sublocale *Prod-any*: *comm-monoid-fun times 1*
defines
Prod-any = *Prod-any.G*
rewrites
comm-monoid-set.F times 1 = *setprod*
proof −
show *comm-monoid-fun times 1* **..**
then interpret *Prod-any*: *comm-monoid-fun times 1* **.**
from *setprod-def* **show** *comm-monoid-set.F times 1* = *setprod* **by** (*auto intro*: *sym*)
qed

end

syntax (*ASCII*)
-Prod-any :: *pttrn* ⇒ *'a* ⇒ *'a*::*comm-monoid-mult* ((*3PROD* -. -) [*0*, *10*] *10*)
syntax
-Prod-any :: *pttrn* ⇒ *'a* ⇒ *'a*::*comm-monoid-mult* ((*3*∏ -. -) [*0*, *10*] *10*)
translations
∏ *a*. *b* == *CONST Prod-any* (λ*a*. *b*)

lemma *Prod-any-zero*:
fixes *f* :: *'b* ⇒ *'a* :: *comm-semiring-1*
assumes *finite* {*a*. *f a* ≠ *1*}
assumes *f a* = *0*
shows (∏ *a*. *f a*) = *0*
proof −
from ‹*f a* = *0*› **have** *f a* ≠ *1* **by** *simp*
with ‹*f a* = *0*› **have** ∃ *a*. *f a* ≠ *1* ∧ *f a* = *0* **by** *blast*
with ‹*finite* {*a*. *f a* ≠ *1*}› **show** *?thesis*
by (*simp add*: *Prod-any.expand-set setprod-zero*)

qed

lemma *Prod-any-not-zero*:
fixes f :: $'b \Rightarrow 'a$ :: *comm-semiring-1*
assumes *finite* $\{a.\ f\ a \neq 1\}$
assumes $(\prod a.\ f\ a) \neq 0$
shows $f\ a \neq 0$
using *assms Prod-any-zero* [*of f*] **by** *blast*

lemma *power-Sum-any*:
assumes *finite* $\{a.\ f\ a \neq 0\}$
shows $c\ \hat{}\ (\sum a.\ f\ a) = (\prod a.\ c\ \hat{}\ f\ a)$
proof −
have $\{a.\ c\ \hat{}\ f\ a \neq 1\} \subseteq \{a.\ f\ a \neq 0\}$
by (*auto intro*: *ccontr*)
with *assms* **show** *?thesis*
by (*simp add*: *Sum-any.expand-set Prod-any.expand-superset power-setsum*)
qed

end

54 Immutable Arrays with Code Generation

theory *IArray*
imports *Main*
begin

Immutable arrays are lists wrapped up in an additional constructor. There are no update operations. Hence code generation can safely implement this type by efficient target language arrays. Currently only SML is provided. Should be extended to other target languages and more operations.

Note that arrays cannot be printed directly but only by turning them into lists first. Arrays could be converted back into lists for printing if they were wrapped up in an additional constructor.

context
begin

datatype *'a iarray* = *IArray 'a list*

qualified primrec *list-of* :: *'a iarray* ⇒ *'a list* **where**
list-of (*IArray xs*) = *xs*

qualified definition *of-fun* :: (*nat* ⇒ *'a*) ⇒ *nat* ⇒ *'a iarray* **where**
[*simp*]: *of-fun f n* = *IArray* (*map f* [*0*..<*n*])

qualified definition *sub* :: *'a iarray* ⇒ *nat* ⇒ *'a* (**infixl** !! *100*) **where**
[*simp*]: *as* !! *n* = *IArray.list-of as* ! *n*

qualified definition *length* :: *'a iarray* ⇒ *nat* **where**
[*simp*]: *length as* = *List.length* (*IArray.list-of as*)

qualified fun *all* :: (*'a* ⇒ *bool*) ⇒ *'a iarray* ⇒ *bool* **where**
all p (*IArray as*) = (*ALL a* : *set as*. *p a*)

qualified fun *exists* :: (*'a* ⇒ *bool*) ⇒ *'a iarray* ⇒ *bool* **where**
exists p (*IArray as*) = (*EX a* : *set as*. *p a*)

lemma *list-of-code* [*code*]:
IArray.list-of as = *map* (λ*n*. *as* !! *n*) [*0* ..< *IArray.length as*]
by (*cases as*) (*simp add*: *map-nth*)

end

54.1 Code Generation

code-reserved *SML Vector*

code-printing
type-constructor *iarray* ⇀ (*SML*) - *Vector.vector*
| **constant** *IArray* ⇀ (*SML*) *Vector.fromList*
| **constant** *IArray.all* ⇀ (*SML*) *Vector.all*
| **constant** *IArray.exists* ⇀ (*SML*) *Vector.exists*

lemma [*code*]:
size (*as* :: *'a iarray*) = *Suc* (*length* (*IArray.list-of as*))
by (*cases as*) *simp*

lemma [*code*]:
size-iarray f as = *Suc* (*size-list f* (*IArray.list-of as*))
by (*cases as*) *simp*

lemma [*code*]:
rec-iarray f as = *f* (*IArray.list-of as*)
by (*cases as*) *simp*

lemma [*code*]:
case-iarray f as = *f* (*IArray.list-of as*)
by (*cases as*) *simp*

lemma [*code*]:
set-iarray as = *set* (*IArray.list-of as*)
by (*case-tac as*) *auto*

lemma [*code*]:
map-iarray f as = *IArray* (*map f* (*IArray.list-of as*))
by (*case-tac as*) *auto*

lemma [*code*]:
rel-iarray r as bs = *list-all2 r* (*IArray.list-of as*) (*IArray.list-of bs*)
by (*case-tac as*) (*case-tac bs*, *auto*)

lemma [*code*]:
HOL.equal as bs ⟷ *HOL.equal* (*IArray.list-of as*) (*IArray.list-of bs*)
by (*cases as*, *cases bs*) (*simp add*: *equal*)

context
begin

qualified primrec *tabulate* :: *integer* × (*integer* ⇒ *'a*) ⇒ *'a iarray* **where**
tabulate (*n*, *f*) = *IArray* (*map* (*f* ∘ *integer-of-nat*) [*0*..<*nat-of-integer n*])

end

lemma [*code*]:
IArray.of-fun f n = *IArray.tabulate* (*integer-of-nat n*, *f* ∘ *nat-of-integer*)
by *simp*

code-printing
constant *IArray.tabulate* ⇀ (*SML*) *Vector.tabulate*

context
begin

qualified primrec *sub'* :: *'a iarray* × *integer* ⇒ *'a* **where**
[*code del*]: *sub'* (*as*, *n*) = *IArray.list-of as* ! *nat-of-integer n*

end

lemma [*code*]:
IArray.sub' (*IArray as*, *n*) = *as* ! *nat-of-integer n*
by *simp*

lemma [*code*]:
as !! *n* = *IArray.sub'* (*as*, *integer-of-nat n*)
by *simp*

code-printing
constant *IArray.sub'* ⇀ (*SML*) *Vector.sub*

context
begin

qualified definition *length'* :: *'a iarray* ⇒ *integer* **where**
[*code del*, *simp*]: *length' as* = *integer-of-nat* (*List.length* (*IArray.list-of as*))

end

lemma [*code*]:
IArray.length′ (*IArray as*) = *integer-of-nat* (*List.length as*)
by *simp*

lemma [*code*]:
IArray.length as = *nat-of-integer* (*IArray.length′ as*)
by *simp*

context *term-syntax*
begin

lemma [*code*]:
Code-Evaluation.term-of (*as* :: *′a::typerep iarray*) =
Code-Evaluation.Const (*STR ″IArray.iarray.IArray″*) (*TYPEREP*(*′a list* ⇒ *′a iarray*)) <·> (*Code-Evaluation.term-of* (*IArray.list-of as*))
by (*subst term-of-anything*) *rule*

end

code-printing
constant *IArray.length′* ⇀ (*SML*) *Vector.length*

end

theory *Lattice-Constructions*
imports *Main*
begin

54.2 Values extended by a bottom element

datatype *′a bot* = *Value ′a* | *Bot*

instantiation *bot* :: (*preorder*) *preorder*
begin

definition *less-eq-bot* **where**
$x \leq y \longleftrightarrow$ (*case x of Bot* ⇒ *True* | *Value x* ⇒ (*case y of Bot* ⇒ *False* | *Value y* ⇒ $x \leq y$))

definition *less-bot* **where**
$x < y \longleftrightarrow$ (*case y of Bot* ⇒ *False* | *Value y* ⇒ (*case x of Bot* ⇒ *True* | *Value x* ⇒ $x < y$))

lemma *less-eq-bot-Bot* [*simp*]: *Bot* $\leq x$
by (*simp add*: *less-eq-bot-def*)

lemma *less-eq-bot-Bot-code* [*code*]: $Bot \leq x \longleftrightarrow True$
by *simp*

lemma *less-eq-bot-Bot-is-Bot*: $x \leq Bot \Longrightarrow x = Bot$
by (*cases x*) (*simp-all add*: *less-eq-bot-def*)

lemma *less-eq-bot-Value-Bot* [*simp*, *code*]: $Value\ x \leq Bot \longleftrightarrow False$
by (*simp add*: *less-eq-bot-def*)

lemma *less-eq-bot-Value* [*simp*, *code*]: $Value\ x \leq Value\ y \longleftrightarrow x \leq y$
by (*simp add*: *less-eq-bot-def*)

lemma *less-bot-Bot* [*simp*, *code*]: $x < Bot \longleftrightarrow False$
by (*simp add*: *less-bot-def*)

lemma *less-bot-Bot-is-Value*: $Bot < x \Longrightarrow \exists z.\ x = Value\ z$
by (*cases x*) (*simp-all add*: *less-bot-def*)

lemma *less-bot-Bot-Value* [*simp*]: $Bot < Value\ x$
by (*simp add*: *less-bot-def*)

lemma *less-bot-Bot-Value-code* [*code*]: $Bot < Value\ x \longleftrightarrow True$
by *simp*

lemma *less-bot-Value* [*simp*, *code*]: $Value\ x < Value\ y \longleftrightarrow x < y$
by (*simp add*: *less-bot-def*)

instance
by *standard*
(*auto simp add*: *less-eq-bot-def less-bot-def less-le-not-le elim*: *order-trans split*: *bot.splits*)

end

instance *bot* :: (*order*) *order*
by *standard* (*auto simp add*: *less-eq-bot-def less-bot-def split*: *bot.splits*)

instance *bot* :: (*linorder*) *linorder*
by *standard* (*auto simp add*: *less-eq-bot-def less-bot-def split*: *bot.splits*)

instantiation *bot* :: (*order*) *bot*
begin
definition $bot = Bot$
instance ..
end

instantiation *bot* :: (*top*) *top*
begin
definition $top = Value\ top$

```
  instance ..
end

instantiation bot :: (semilattice-inf) semilattice-inf
begin

definition inf-bot
where
  inf x y =
    (case x of
      Bot ⇒ Bot
    | Value v ⇒
        (case y of
          Bot ⇒ Bot
        | Value v' ⇒ Value (inf v v')))

instance
  by standard (auto simp add: inf-bot-def less-eq-bot-def split: bot.splits)

end

instantiation bot :: (semilattice-sup) semilattice-sup
begin

definition sup-bot
where
  sup x y =
    (case x of
      Bot ⇒ y
    | Value v ⇒
        (case y of
          Bot ⇒ x
        | Value v' ⇒ Value (sup v v')))

instance
  by standard (auto simp add: sup-bot-def less-eq-bot-def split: bot.splits)

end

instance bot :: (lattice) bounded-lattice-bot
  by intro-classes (simp add: bot-bot-def)
```

54.3 Values extended by a top element

```
datatype 'a top = Value 'a | Top

instantiation top :: (preorder) preorder
begin
```

definition *less-eq-top* **where**
$x \leq y \longleftrightarrow (case\ y\ of\ Top \Rightarrow True \mid Value\ y \Rightarrow (case\ x\ of\ Top \Rightarrow False \mid Value\ x \Rightarrow x \leq y))$

definition *less-top* **where**
$x < y \longleftrightarrow (case\ x\ of\ Top \Rightarrow False \mid Value\ x \Rightarrow (case\ y\ of\ Top \Rightarrow True \mid Value\ y \Rightarrow x < y))$

lemma *less-eq-top-Top* [*simp*]: $x \leq Top$
by (*simp add*: *less-eq-top-def*)

lemma *less-eq-top-Top-code* [*code*]: $x \leq Top \longleftrightarrow True$
by *simp*

lemma *less-eq-top-is-Top*: $Top \leq x \Longrightarrow x = Top$
by (*cases x*) (*simp-all add*: *less-eq-top-def*)

lemma *less-eq-top-Top-Value* [*simp*, *code*]: $Top \leq Value\ x \longleftrightarrow False$
by (*simp add*: *less-eq-top-def*)

lemma *less-eq-top-Value-Value* [*simp*, *code*]: $Value\ x \leq Value\ y \longleftrightarrow x \leq y$
by (*simp add*: *less-eq-top-def*)

lemma *less-top-Top* [*simp*, *code*]: $Top < x \longleftrightarrow False$
by (*simp add*: *less-top-def*)

lemma *less-top-Top-is-Value*: $x < Top \Longrightarrow \exists z.\ x = Value\ z$
by (*cases x*) (*simp-all add*: *less-top-def*)

lemma *less-top-Value-Top* [*simp*]: $Value\ x < Top$
by (*simp add*: *less-top-def*)

lemma *less-top-Value-Top-code* [*code*]: $Value\ x < Top \longleftrightarrow True$
by *simp*

lemma *less-top-Value* [*simp*, *code*]: $Value\ x < Value\ y \longleftrightarrow x < y$
by (*simp add*: *less-top-def*)

instance
by *standard*
(*auto simp add*: *less-eq-top-def less-top-def less-le-not-le elim*: *order-trans split*: *top.splits*)

end

instance *top* :: (*order*) *order*
by *standard* (*auto simp add*: *less-eq-top-def less-top-def split*: *top.splits*)

instance *top* :: (*linorder*) *linorder*

by *standard* (*auto simp add*: *less-eq-top-def less-top-def split*: *top.splits*)

instantiation *top* :: (*order*) *top*
begin
  definition *top* = *Top*
  instance ..
end

instantiation *top* :: (*bot*) *bot*
begin
  definition *bot* = *Value bot*
  instance ..
end

instantiation *top* :: (*semilattice-inf*) *semilattice-inf*
begin

definition *inf-top*
where
  inf x y =
    (*case x of*
      Top ⇒ *y*
    | *Value v* ⇒
        (*case y of*
          Top ⇒ *x*
        | *Value v′* ⇒ *Value* (*inf v v′*)))

instance
  by *standard* (*auto simp add*: *inf-top-def less-eq-top-def split*: *top.splits*)

end

instantiation *top* :: (*semilattice-sup*) *semilattice-sup*
begin

definition *sup-top*
where
  sup x y =
    (*case x of*
      Top ⇒ *Top*
    | *Value v* ⇒
        (*case y of*
          Top ⇒ *Top*
        | *Value v′* ⇒ *Value* (*sup v v′*)))

instance
  by *standard* (*auto simp add*: *sup-top-def less-eq-top-def split*: *top.splits*)

end

instance *top* :: (*lattice*) *bounded-lattice-top*
 by *standard* (*simp add*: *top-top-def*)

54.4 Values extended by a top and a bottom element

datatype *'a flat-complete-lattice* = *Value 'a* | *Bot* | *Top*

instantiation *flat-complete-lattice* :: (*type*) *order*
begin

definition *less-eq-flat-complete-lattice*
where

```
x ≤ y ≡
  (case x of
    Bot ⇒ True
  | Value v1 ⇒
      (case y of
        Bot ⇒ False
      | Value v2 ⇒ v1 = v2
      | Top ⇒ True)
  | Top ⇒ y = Top)
```

definition *less-flat-complete-lattice*
where

```
x < y =
  (case x of
    Bot ⇒ y ≠ Bot
  | Value v1 ⇒ y = Top
  | Top ⇒ False)
```

lemma [*simp*]: *Bot* ≤ *y*
 unfolding *less-eq-flat-complete-lattice-def* **by** *auto*

lemma [*simp*]: *y* ≤ *Top*
 unfolding *less-eq-flat-complete-lattice-def* **by** (*auto split*: *flat-complete-lattice.splits*)

lemma *greater-than-two-values*:
 assumes *a* ≠ *b* *Value a* ≤ *z* *Value b* ≤ *z*
 shows *z* = *Top*
 using *assms*
 by (*cases z*) (*auto simp add*: *less-eq-flat-complete-lattice-def*)

lemma *lesser-than-two-values*:
 assumes *a* ≠ *b* *z* ≤ *Value a* *z* ≤ *Value b*
 shows *z* = *Bot*
 using *assms*
 by (*cases z*) (*auto simp add*: *less-eq-flat-complete-lattice-def*)

```
instance
  by standard
    (auto simp add: less-eq-flat-complete-lattice-def less-flat-complete-lattice-def
      split: flat-complete-lattice.splits)

end

instantiation flat-complete-lattice :: (type) bot
begin
  definition bot = Bot
  instance ..
end

instantiation flat-complete-lattice :: (type) top
begin
  definition top = Top
  instance ..
end

instantiation flat-complete-lattice :: (type) lattice
begin

definition inf-flat-complete-lattice
where
  inf x y =
    (case x of
      Bot ⇒ Bot
    | Value v1 ⇒
        (case y of
          Bot ⇒ Bot
        | Value v2 ⇒ if v1 = v2 then x else Bot
        | Top ⇒ x)
    | Top ⇒ y)

definition sup-flat-complete-lattice
where
  sup x y =
    (case x of
      Bot ⇒ y
    | Value v1 ⇒
        (case y of
          Bot ⇒ x
        | Value v2 ⇒ if v1 = v2 then x else Top
        | Top ⇒ Top)
    | Top ⇒ Top)

instance
  by standard
    (auto simp add: inf-flat-complete-lattice-def sup-flat-complete-lattice-def
```

less-eq-flat-complete-lattice-def split: *flat-complete-lattice.splits*)

end

instantiation *flat-complete-lattice* :: (*type*) *complete-lattice*
begin

definition *Sup-flat-complete-lattice*
where
Sup A =
(*if A* = {} ∨ *A* = {*Bot*} *then Bot*
else if ∃ *v*. *A* − {*Bot*} = {*Value v*} *then Value* (*THE v*. *A* − {*Bot*} = {*Value v*})
else Top)

definition *Inf-flat-complete-lattice*
where
Inf A =
(*if A* = {} ∨ *A* = {*Top*} *then Top*
else if ∃ *v*. *A* − {*Top*} = {*Value v*} *then Value* (*THE v*. *A* − {*Top*} = {*Value v*})
else Bot)

instance
proof
fix *x* :: '*a flat-complete-lattice*
fix *A*
assume *x* ∈ *A*
{
fix *v*
assume *A* − {*Top*} = {*Value v*}
then have (*THE v*. *A* − {*Top*} = {*Value v*}) = *v*
by (*auto intro*!: *the1-equality*)
moreover
from ‹*x* ∈ *A*› ‹*A* − {*Top*} = {*Value v*}› **have** *x* = *Top* ∨ *x* = *Value v*
by *auto*
ultimately have *Value* (*THE v*. *A* − {*Top*} = {*Value v*}) ≤ *x*
by *auto*
}
with ‹*x* ∈ *A*› **show** *Inf A* ≤ *x*
unfolding *Inf-flat-complete-lattice-def*
by *fastforce*
next
fix *z* :: '*a flat-complete-lattice*
fix *A*
show *z* ≤ *Inf A* **if** *z*: ⋀*x*. *x* ∈ *A* ⟹ *z* ≤ *x*
proof −
consider *A* = {} ∨ *A* = {*Top*}
| *A* ≠ {} *A* ≠ {*Top*} ∃ *v*. *A* − {*Top*} = {*Value v*}

```
    | A ≠ {} A ≠ {Top} ¬ (∃ v. A − {Top} = {Value v})
    by blast
  then show ?thesis
  proof cases
    case 1
    then have Inf A = Top
      unfolding Inf-flat-complete-lattice-def by auto
    then show ?thesis by simp
  next
    case 2
    then obtain v where v1: A − {Top} = {Value v}
      by auto
    then have v2: (THE v. A − {Top} = {Value v}) = v
      by (auto intro!: the1-equality)
    from 2 v2 have Inf: Inf A = Value v
      unfolding Inf-flat-complete-lattice-def by simp
    from v1 have Value v ∈ A by blast
    then have z ≤ Value v by (rule z)
    with Inf show ?thesis by simp
  next
    case 3
    then have Inf: Inf A = Bot
      unfolding Inf-flat-complete-lattice-def by auto
    have z ≤ Bot
    proof (cases A − {Top} = {Bot})
      case True
      then have Bot ∈ A by blast
      then show ?thesis by (rule z)
    next
      case False
      from 3 obtain a1 where a1: a1 ∈ A − {Top}
        by auto
      from 3 False a1 obtain a2 where a2 ∈ A − {Top} ∧ a1 ≠ a2
        by (cases a1) auto
      with a1 z[of a1] z[of a2] show ?thesis
        apply (cases a1)
        apply auto
        apply (cases a2)
        apply auto
        apply (auto dest!: lesser-than-two-values)
        done
    qed
    with Inf show ?thesis by simp
  qed
 qed
next
 fix x :: 'a flat-complete-lattice
 fix A
 assume x ∈ A
```

```
{
  fix v
  assume A − {Bot} = {Value v}
  then have (THE v. A − {Bot} = {Value v}) = v
    by (auto intro!: the1-equality)
  moreover
  from ‹x ∈ A› ‹A − {Bot} = {Value v}› have x = Bot ∨ x = Value v
    by auto
  ultimately have x ≤ Value (THE v. A − {Bot} = {Value v})
    by auto
}
with ‹x ∈ A› show x ≤ Sup A
  unfolding Sup-flat-complete-lattice-def
  by fastforce
next
  fix z :: 'a flat-complete-lattice
  fix A
  show Sup A ≤ z if z: ⋀x. x ∈ A ⟹ x ≤ z
  proof −
    consider A = {} ∨ A = {Bot}
      | A ≠ {} A ≠ {Bot} ∃v. A − {Bot} = {Value v}
      | A ≠ {} A ≠ {Bot} ¬ (∃v. A − {Bot} = {Value v})
      by blast
    then show ?thesis
    proof cases
      case 1
      then have Sup A = Bot
        unfolding Sup-flat-complete-lattice-def by auto
      then show ?thesis by simp
    next
      case 2
      then obtain v where v1: A − {Bot} = {Value v}
        by auto
      then have v2: (THE v. A − {Bot} = {Value v}) = v
        by (auto intro!: the1-equality)
      from 2 v2 have Sup: Sup A = Value v
        unfolding Sup-flat-complete-lattice-def by simp
      from v1 have Value v ∈ A by blast
      then have Value v ≤ z by (rule z)
      with Sup show ?thesis by simp
    next
      case 3
      then have Sup: Sup A = Top
        unfolding Sup-flat-complete-lattice-def by auto
      have Top ≤ z
      proof (cases A − {Bot} = {Top})
        case True
        then have Top ∈ A by blast
        then show ?thesis by (rule z)
```

```
      next
        case False
        from 3 obtain a1 where a1: a1 ∈ A − {Bot}
          by auto
        from 3 False a1 obtain a2 where a2 ∈ A − {Bot} ∧ a1 ≠ a2
          by (cases a1) auto
        with a1 z[of a1] z[of a2] show ?thesis
          apply (cases a1)
          apply auto
          apply (cases a2)
          apply (auto dest!: greater-than-two-values)
          done
      qed
      with Sup show ?thesis by simp
    qed
  qed
next
  show Inf {} = (top :: 'a flat-complete-lattice)
    by (simp add: Inf-flat-complete-lattice-def top-flat-complete-lattice-def)
  show Sup {} = (bot :: 'a flat-complete-lattice)
    by (simp add: Sup-flat-complete-lattice-def bot-flat-complete-lattice-def)
qed

end

end
```

55 Infinite Streams

```
theory Stream
imports ~~/src/HOL/Library/Nat-Bijection
begin

codatatype (sset: 'a) stream =
  SCons (shd: 'a) (stl: 'a stream) (infixr ## 65)
for
  map: smap
  rel: stream-all2

context
begin


qualified definition smember :: 'a ⇒ 'a stream ⇒ bool where
  [code-abbrev]: smember x s ⟷ x ∈ sset s

lemma smember-code[code, simp]: smember x (y ## s) = (if x = y then True
else smember x s)
  unfolding smember-def by auto
```

end

lemmas *smap-simps*[*simp*] = *stream.map-sel*
lemmas *shd-sset* = *stream.set-sel*(*1*)
lemmas *stl-sset* = *stream.set-sel*(*2*)

theorem *sset-induct*[*consumes 1*, *case-names shd stl*, *induct set*: *sset*]:
  assumes $y \in$ *sset s* **and** $\bigwedge s.$ *P* (*shd s*) *s* **and** $\bigwedge s\ y.$ ⟦$y \in$ *sset* (*stl s*); *P y* (*stl s*)⟧ $\Longrightarrow$ *P y s*
  shows *P y s*
using *assms* **by** *induct* (*metis stream.sel*(*1*), *auto*)

lemma *smap-ctr*: *smap f s* = *x* ## *s'* $\longleftrightarrow$ *f* (*shd s*) = *x* $\wedge$ *smap f* (*stl s*) = *s'*
  by (*cases s*) *simp*

55.1 prepend list to stream

primrec *shift* :: *'a list* ⇒ *'a stream* ⇒ *'a stream* (**infixr** @− *65*) **where**
  shift [] *s* = *s*
| *shift* (*x* # *xs*) *s* = *x* ## *shift xs s*

lemma *smap-shift*[*simp*]: *smap f* (*xs* @− *s*) = *map f xs* @− *smap f s*
  by (*induct xs*) *auto*

lemma *shift-append*[*simp*]: (*xs* @ *ys*) @− *s* = *xs* @− *ys* @− *s*
  by (*induct xs*) *auto*

lemma *shift-simps*[*simp*]:
   shd (*xs* @− *s*) = (*if xs* = [] *then shd s else hd xs*)
   stl (*xs* @− *s*) = (*if xs* = [] *then stl s else tl xs* @− *s*)
  by (*induct xs*) *auto*

lemma *sset-shift*[*simp*]: *sset* (*xs* @− *s*) = *set xs* ∪ *sset s*
  by (*induct xs*) *auto*

lemma *shift-left-inj*[*simp*]: *xs* @− *s1* = *xs* @− *s2* $\longleftrightarrow$ *s1* = *s2*
  by (*induct xs*) *auto*

55.2 set of streams with elements in some fixed set

context
  notes [[*inductive-internals*]]
begin

coinductive-set
  streams :: *'a set* ⇒ *'a stream set*
  for *A* :: *'a set*
where
  Stream[*intro*!, *simp*, *no-atp*]: ⟦$a \in A$; $s \in$ *streams A*⟧ $\Longrightarrow$ *a* ## $s \in$ *streams A*

end

lemma *in-streams*: *stl s* ∈ *streams S* ⟹ *shd s* ∈ *S* ⟹ *s* ∈ *streams S*
  by (*cases s*) *auto*

lemma *streamsE*: *s* ∈ *streams A* ⟹ (*shd s* ∈ *A* ⟹ *stl s* ∈ *streams A* ⟹ *P*) ⟹ *P*
  by (*erule streams.cases*) *simp-all*

lemma *Stream-image*: *x* ## *y* ∈ (*op* ## *x'*) ' *Y* ⟷ *x* = *x'* ∧ *y* ∈ *Y*
  by *auto*

lemma *shift-streams*: ⟦*w* ∈ *lists A*; *s* ∈ *streams A*⟧ ⟹ *w* @− *s* ∈ *streams A*
  by (*induct w*) *auto*

lemma *streams-Stream*: *x* ## *s* ∈ *streams A* ⟷ *x* ∈ *A* ∧ *s* ∈ *streams A*
  by (*auto elim*: *streams.cases*)

lemma *streams-stl*: *s* ∈ *streams A* ⟹ *stl s* ∈ *streams A*
  by (*cases s*) (*auto simp*: *streams-Stream*)

lemma *streams-shd*: *s* ∈ *streams A* ⟹ *shd s* ∈ *A*
  by (*cases s*) (*auto simp*: *streams-Stream*)

lemma *sset-streams*:
  assumes *sset s* ⊆ *A*
  shows *s* ∈ *streams A*
using *assms* **proof** (*coinduction arbitrary*: *s*)
  case *streams* **then show** *?case* **by** (*cases s*) *simp*
qed

lemma *streams-sset*:
  assumes *s* ∈ *streams A*
  shows *sset s* ⊆ *A*
proof
  fix *x* **assume** *x* ∈ *sset s* **from** *this* ‹*s* ∈ *streams A*› **show** *x* ∈ *A*
    by (*induct s*) (*auto intro*: *streams-shd streams-stl*)
qed

lemma *streams-iff-sset*: *s* ∈ *streams A* ⟷ *sset s* ⊆ *A*
  by (*metis sset-streams streams-sset*)

lemma *streams-mono*: *s* ∈ *streams A* ⟹ *A* ⊆ *B* ⟹ *s* ∈ *streams B*
  unfolding *streams-iff-sset* **by** *auto*

lemma *streams-mono2*: *S* ⊆ *T* ⟹ *streams S* ⊆ *streams T*
  by (*auto intro*: *streams-mono*)

lemma *smap-streams*: $s \in streams\ A \Longrightarrow (\bigwedge x.\ x \in A \Longrightarrow f\ x \in B) \Longrightarrow smap\ f\ s \in streams\ B$
unfolding *streams-iff-sset stream.set-map* **by** *auto*

lemma *streams-empty*: $streams\ \{\} = \{\}$
by (*auto elim*: *streams.cases*)

lemma *streams-UNIV*[*simp*]: $streams\ UNIV = UNIV$
by (*auto simp*: *streams-iff-sset*)

55.3 nth, take, drop for streams

primrec *snth* :: $'a\ stream \Rightarrow nat \Rightarrow 'a$ (**infixl** !! *100*) **where**
$s\ !!\ 0 = shd\ s$
$|\ s\ !!\ Suc\ n = stl\ s\ !!\ n$

lemma *snth-Stream*: $(x\ \#\#\ s)\ !!\ Suc\ i = s\ !!\ i$
by *simp*

lemma *snth-smap*[*simp*]: $smap\ f\ s\ !!\ n = f\ (s\ !!\ n)$
by (*induct n arbitrary*: *s*) *auto*

lemma *shift-snth-less*[*simp*]: $p < length\ xs \Longrightarrow (xs\ @-\ s)\ !!\ p = xs\ !\ p$
by (*induct p arbitrary*: *xs*) (*auto simp*: *hd-conv-nth nth-tl*)

lemma *shift-snth-ge*[*simp*]: $p \geq length\ xs \Longrightarrow (xs\ @-\ s)\ !!\ p = s\ !!\ (p - length\ xs)$
by (*induct p arbitrary*: *xs*) (*auto simp*: *Suc-diff-eq-diff-pred*)

lemma *shift-snth*: $(xs\ @-\ s)\ !!\ n = (if\ n < length\ xs\ then\ xs\ !\ n\ else\ s\ !!\ (n - length\ xs))$
by *auto*

lemma *snth-sset*[*simp*]: $s\ !!\ n \in sset\ s$
by (*induct n arbitrary*: *s*) (*auto intro*: *shd-sset stl-sset*)

lemma *sset-range*: $sset\ s = range\ (snth\ s)$
proof (*intro equalityI subsetI*)
fix *x* **assume** $x \in sset\ s$
thus $x \in range\ (snth\ s)$
proof (*induct s*)
case (*stl s x*)
then obtain *n* **where** $x = stl\ s\ !!\ n$ **by** *auto*
thus *?case* **by** (*auto intro*: *range-eqI*[*of - - Suc n*])
qed (*auto intro*: *range-eqI*[*of - - 0*])
qed *auto*

lemma *streams-iff-snth*: $s \in streams\ X \longleftrightarrow (\forall n.\ s\ !!\ n \in X)$
by (*force simp*: *streams-iff-sset sset-range*)

lemma *snth-in*: $s \in streams\ X \Longrightarrow s\ !!\ n \in X$
  by (*simp add*: *streams-iff-snth*)

primrec *stake* :: $nat \Rightarrow 'a\ stream \Rightarrow 'a\ list$ **where**
  $stake\ 0\ s = []$
| $stake\ (Suc\ n)\ s = shd\ s\ \#\ stake\ n\ (stl\ s)$

lemma *length-stake*[*simp*]: $length\ (stake\ n\ s) = n$
  by (*induct n arbitrary*: *s*) *auto*

lemma *stake-smap*[*simp*]: $stake\ n\ (smap\ f\ s) = map\ f\ (stake\ n\ s)$
  by (*induct n arbitrary*: *s*) *auto*

lemma *take-stake*: $take\ n\ (stake\ m\ s) = stake\ (min\ n\ m)\ s$
proof (*induct m arbitrary*: *s n*)
  case (*Suc m*) **thus** *?case* **by** (*cases n*) *auto*
qed *simp*

primrec *sdrop* :: $nat \Rightarrow 'a\ stream \Rightarrow 'a\ stream$ **where**
  $sdrop\ 0\ s = s$
| $sdrop\ (Suc\ n)\ s = sdrop\ n\ (stl\ s)$

lemma *sdrop-simps*[*simp*]:
  $shd\ (sdrop\ n\ s) = s\ !!\ n\ \ stl\ (sdrop\ n\ s) = sdrop\ (Suc\ n)\ s$
  by (*induct n arbitrary*: *s*) *auto*

lemma *sdrop-smap*[*simp*]: $sdrop\ n\ (smap\ f\ s) = smap\ f\ (sdrop\ n\ s)$
  by (*induct n arbitrary*: *s*) *auto*

lemma *sdrop-stl*: $sdrop\ n\ (stl\ s) = stl\ (sdrop\ n\ s)$
  by (*induct n*) *auto*

lemma *drop-stake*: $drop\ n\ (stake\ m\ s) = stake\ (m - n)\ (sdrop\ n\ s)$
proof (*induct m arbitrary*: *s n*)
  case (*Suc m*) **thus** *?case* **by** (*cases n*) *auto*
qed *simp*

lemma *stake-sdrop*: $stake\ n\ s\ @-\ sdrop\ n\ s = s$
  by (*induct n arbitrary*: *s*) *auto*

lemma *id-stake-snth-sdrop*:
  $s = stake\ i\ s\ @-\ s\ !!\ i\ \#\#\ sdrop\ (Suc\ i)\ s$
  by (*subst stake-sdrop*[*symmetric*, *of - i*]) (*metis sdrop-simps stream.collapse*)

lemma *smap-alt*: $smap\ f\ s = s' \longleftrightarrow (\forall n.\ f\ (s\ !!\ n) = s'\ !!\ n)$ (**is** *?L = ?R*)
proof
  assume *?R*
  then have $\bigwedge n.\ smap\ f\ (sdrop\ n\ s) = sdrop\ n\ s'$

by *coinduction* (*auto intro*: *exI*[*of* - *0*] *simp del*: *sdrop.simps*(*2*))
then show *?L* **using** *sdrop.simps*(*1*) **by** *metis*
qed *auto*

lemma *stake-invert-Nil*[*iff*]: *stake n s* = [] $\longleftrightarrow$ *n* = *0*
by (*induct n*) *auto*

lemma *sdrop-shift*: *sdrop i* (*w* @− *s*) = *drop i w* @− *sdrop* (*i* − *length w*) *s*
by (*induct i arbitrary*: *w s*) (*auto simp*: *drop-tl drop-Suc neq-Nil-conv*)

lemma *stake-shift*: *stake i* (*w* @− *s*) = *take i w* @ *stake* (*i* − *length w*) *s*
by (*induct i arbitrary*: *w s*) (*auto simp*: *neq-Nil-conv*)

lemma *stake-add*[*simp*]: *stake m s* @ *stake n* (*sdrop m s*) = *stake* (*m* + *n*) *s*
by (*induct m arbitrary*: *s*) *auto*

lemma *sdrop-add*[*simp*]: *sdrop n* (*sdrop m s*) = *sdrop* (*m* + *n*) *s*
by (*induct m arbitrary*: *s*) *auto*

lemma *sdrop-snth*: *sdrop n s* !! *m* = *s* !! (*n* + *m*)
by (*induct n arbitrary*: *m s*) *auto*

partial-function (*tailrec*) *sdrop-while* :: (*'a* ⇒ *bool*) ⇒ *'a stream* ⇒ *'a stream*
where
sdrop-while P s = (*if P* (*shd s*) *then sdrop-while P* (*stl s*) *else s*)

lemma *sdrop-while-SCons*[*code*]:
sdrop-while P (*a* ## *s*) = (*if P a then sdrop-while P s else a* ## *s*)
by (*subst sdrop-while.simps*) *simp*

lemma *sdrop-while-sdrop-LEAST*:
assumes ∃ *n*. *P* (*s* !! *n*)
shows *sdrop-while* (*Not o P*) *s* = *sdrop* (*LEAST n*. *P* (*s* !! *n*)) *s*
proof −
from *assms* **obtain** *m* **where** *P* (*s* !! *m*) $\bigwedge$*n*. *P* (*s* !! *n*) ⟹ *m* ≤ *n*
and ∗: (*LEAST n*. *P* (*s* !! *n*)) = *m* **by** *atomize-elim* (*auto intro*: *LeastI Least-le*)
thus *?thesis* **unfolding** ∗
proof (*induct m arbitrary*: *s*)
case (*Suc m*)
hence *sdrop-while* (*Not* ∘ *P*) (*stl s*) = *sdrop m* (*stl s*)
by (*metis* (*full-types*) *not-less-eq-eq snth.simps*(*2*))
moreover from *Suc*(*3*) **have** ¬ (*P* (*s* !! *0*)) **by** *blast*
ultimately show *?case* **by** (*subst sdrop-while.simps*) *simp*
qed (*metis comp-apply sdrop.simps*(*1*) *sdrop-while.simps snth.simps*(*1*))
qed

primcorec *sfilter* **where**
shd (*sfilter P s*) = *shd* (*sdrop-while* (*Not o P*) *s*)
| *stl* (*sfilter P s*) = *sfilter P* (*stl* (*sdrop-while* (*Not o P*) *s*))

lemma *sfilter-Stream*: *sfilter P* (*x* ## *s*) = (*if P x then x* ## *sfilter P s else sfilter P s*)
proof (*cases P x*)
 case *True* **thus** *?thesis* **by** (*subst sfilter.ctr*) (*simp add*: *sdrop-while-SCons*)
next
 case *False* **thus** *?thesis* **by** (*subst* (*1 2*) *sfilter.ctr*) (*simp add*: *sdrop-while-SCons*)
qed

55.4 unary predicates lifted to streams

definition *stream-all P s* = (∀ *p*. *P* (*s* !! *p*))

lemma *stream-all-iff* [*iff*]: *stream-all P s* ⟷ *Ball* (*sset s*) *P*
 unfolding *stream-all-def sset-range* **by** *auto*

lemma *stream-all-shift*[*simp*]: *stream-all P* (*xs* @− *s*) = (*list-all P xs* ∧ *stream-all P s*)
 unfolding *stream-all-iff list-all-iff* **by** *auto*

lemma *stream-all-Stream*: *stream-all P* (*x* ## *X*) ⟷ *P x* ∧ *stream-all P X*
 by *simp*

55.5 recurring stream out of a list

primcorec *cycle* :: *'a list* ⇒ *'a stream* **where**
 shd (*cycle xs*) = *hd xs*
| *stl* (*cycle xs*) = *cycle* (*tl xs* @ [*hd xs*])

lemma *cycle-decomp*: *u* ≠ [] ⟹ *cycle u* = *u* @− *cycle u*
proof (*coinduction arbitrary*: *u*)
 case *Eq-stream* **then show** *?case* **using** *stream.collapse*[*of cycle u*]
  by (*auto intro*!: *exI*[*of - tl u* @ [*hd u*]])
qed

lemma *cycle-Cons*[*code*]: *cycle* (*x* # *xs*) = *x* ## *cycle* (*xs* @ [*x*])
 by (*subst cycle.ctr*) *simp*

lemma *cycle-rotated*: ⟦*v* ≠ []; *cycle u* = *v* @− *s*⟧ ⟹ *cycle* (*tl u* @ [*hd u*]) = *tl v* @− *s*
 by (*auto dest*: *arg-cong*[*of - - stl*])

lemma *stake-append*: *stake n* (*u* @− *s*) = *take* (*min* (*length u*) *n*) *u* @ *stake* (*n* − *length u*) *s*
proof (*induct n arbitrary*: *u*)
 case (*Suc n*) **thus** *?case* **by** (*cases u*) *auto*
qed *auto*

lemma *stake-cycle-le*[*simp*]:
 assumes *u* ≠ [] *n* < *length u*

shows *stake n* (*cycle u*) = *take n u*
using *min-absorb2*[*OF less-imp-le-nat*[*OF assms*(*2*)]]
by (*subst cycle-decomp*[*OF assms*(*1*)], *subst stake-append*) *auto*

lemma *stake-cycle-eq*[*simp*]: $u \neq [] \Longrightarrow$ *stake* (*length u*) (*cycle u*) = *u*
by (*subst cycle-decomp*) (*auto simp*: *stake-shift*)

lemma *sdrop-cycle-eq*[*simp*]: $u \neq [] \Longrightarrow$ *sdrop* (*length u*) (*cycle u*) = *cycle u*
by (*subst cycle-decomp*) (*auto simp*: *sdrop-shift*)

lemma *stake-cycle-eq-mod-0*[*simp*]: $[\![u \neq []; n\ mod\ length\ u = 0]\!] \Longrightarrow$
stake n (*cycle u*) = *concat* (*replicate* (*n div length u*) *u*)
by (*induct n div length u arbitrary*: *n u*) (*auto simp*: *stake-add*[*symmetric*])

lemma *sdrop-cycle-eq-mod-0*[*simp*]: $[\![u \neq []; n\ mod\ length\ u = 0]\!] \Longrightarrow$
sdrop n (*cycle u*) = *cycle u*
by (*induct n div length u arbitrary*: *n u*) (*auto simp*: *sdrop-add*[*symmetric*])

lemma *stake-cycle*: $u \neq [] \Longrightarrow$
stake n (*cycle u*) = *concat* (*replicate* (*n div length u*) *u*) @ *take* (*n mod length u*) *u*
by (*subst mod-div-equality*[*of n length u*, *symmetric*], *unfold stake-add*[*symmetric*]) *auto*

lemma *sdrop-cycle*: $u \neq [] \Longrightarrow$ *sdrop n* (*cycle u*) = *cycle* (*rotate* (*n mod length u*) *u*)
by (*induct n arbitrary*: *u*) (*auto simp*: *rotate1-rotate-swap rotate1-hd-tl rotate-conv-mod*[*symmetric*])

55.6 iterated application of a function

primcorec *siterate* **where**
shd (*siterate f x*) = *x*
| *stl* (*siterate f x*) = *siterate f* (*f x*)

lemma *stake-Suc*: *stake* (*Suc n*) *s* = *stake n s* @ [*s* !! *n*]
by (*induct n arbitrary*: *s*) *auto*

lemma *snth-siterate*[*simp*]: *siterate f x* !! *n* = (*f*^^*n*) *x*
by (*induct n arbitrary*: *x*) (*auto simp*: *funpow-swap1*)

lemma *sdrop-siterate*[*simp*]: *sdrop n* (*siterate f x*) = *siterate f* ((*f*^^*n*) *x*)
by (*induct n arbitrary*: *x*) (*auto simp*: *funpow-swap1*)

lemma *stake-siterate*[*simp*]: *stake n* (*siterate f x*) = *map* (λn. (*f*^^*n*) *x*) [*0* ..< *n*]
by (*induct n arbitrary*: *x*) (*auto simp del*: *stake.simps*(*2*) *simp*: *stake-Suc*)

lemma *sset-siterate*: *sset* (*siterate f x*) = {(*f*^^*n*) *x* | *n*. *True*}
by (*auto simp*: *sset-range*)

lemma *smap-siterate*: *smap f* (*siterate f x*) = *siterate f* (*f x*)
by (*coinduction arbitrary*: *x*) *auto*

55.7 stream repeating a single element

abbreviation *sconst* ≡ *siterate id*

lemma *shift-replicate-sconst*[*simp*]: *replicate n x* @− *sconst x* = *sconst x*
by (*subst* (*3*) *stake-sdrop*[*symmetric*]) (*simp add*: *map-replicate-trivial*)

lemma *sset-sconst*[*simp*]: *sset* (*sconst x*) = {*x*}
by (*simp add*: *sset-siterate*)

lemma *sconst-alt*: *s* = *sconst x* ⟷ *sset s* = {*x*}
proof
assume *sset s* = {*x*}
then show *s* = *sconst x*
proof (*coinduction arbitrary*: *s*)
case *Eq-stream*
then have *shd s* = *x sset* (*stl s*) ⊆ {*x*} **by** (*case-tac* [!] *s*) *auto*
then have *sset* (*stl s*) = {*x*} **by** (*cases stl s*) *auto*
with ‹*shd s* = *x*› **show** *?case* **by** *auto*
qed
qed *simp*

lemma *sconst-cycle*: *sconst x* = *cycle* [*x*]
by *coinduction auto*

lemma *smap-sconst*: *smap f* (*sconst x*) = *sconst* (*f x*)
by *coinduction auto*

lemma *sconst-streams*: *x* ∈ *A* ⟹ *sconst x* ∈ *streams A*
by (*simp add*: *streams-iff-sset*)

55.8 stream of natural numbers

abbreviation *fromN* ≡ *siterate Suc*

abbreviation *nats* ≡ *fromN 0*

lemma *sset-fromN*[*simp*]: *sset* (*fromN n*) = {*n* ..}
by (*auto simp add*: *sset-siterate le-iff-add*)

lemma *stream-smap-fromN*: *s* = *smap* (λ*j*. *let i* = *j* − *n in s* !! *i*) (*fromN n*)
by (*coinduction arbitrary*: *s n*)
(*force simp*: *neq-Nil-conv Let-def snth.simps*(*2*)[*symmetric*] *Suc-diff-Suc*
intro: *stream.map-cong split*: *if-splits simp del*: *snth.simps*(*2*))

lemma *stream-smap-nats*: *s* = *smap* (*snth s*) *nats*
using *stream-smap-fromN*[**where** *n* = *0*] **by** *simp*

55.9 flatten a stream of lists

primcorec *flat* **where**
  shd (*flat ws*) = *hd* (*shd ws*)
| *stl* (*flat ws*) = *flat* (*if tl* (*shd ws*) = [] *then stl ws else tl* (*shd ws*) ## *stl ws*)

lemma *flat-Cons*[*simp*, *code*]: *flat* ((*x* # *xs*) ## *ws*) = *x* ## *flat* (*if xs* = [] *then ws else xs* ## *ws*)
  by (*subst flat.ctr*) *simp*

lemma *flat-Stream*[*simp*]: *xs* ≠ [] ⟹ *flat* (*xs* ## *ws*) = *xs* @− *flat ws*
  by (*induct xs*) *auto*

lemma *flat-unfold*: *shd ws* ≠ [] ⟹ *flat ws* = *shd ws* @− *flat* (*stl ws*)
  by (*cases ws*) *auto*

lemma *flat-snth*: ∀ *xs* ∈ *sset s*. *xs* ≠ [] ⟹ *flat s* !! *n* = (*if n* < *length* (*shd s*) *then*
  shd s ! *n else flat* (*stl s*) !! (*n* − *length* (*shd s*)))
  by (*metis flat-unfold not-less shd-sset shift-snth-ge shift-snth-less*)

lemma *sset-flat*[*simp*]: ∀ *xs* ∈ *sset s*. *xs* ≠ [] ⟹
  sset (*flat s*) = (⋃ *xs* ∈ *sset s*. *set xs*) (**is** *?P* ⟹ *?L* = *?R*)
proof *safe*
  fix *x* **assume** *?P x* : *?L*
  then obtain *m* **where** *x* = *flat s* !! *m* **by** (*metis image-iff sset-range*)
  with ‹*?P*› **obtain** *n m′* **where** *x* = *s* !! *n* ! *m′ m′* < *length* (*s* !! *n*)
  proof (*atomize-elim*, *induct m arbitrary*: *s rule*: *less-induct*)
    case (*less y*)
    thus *?case*
    proof (*cases y* < *length* (*shd s*))
      case *True* **thus** *?thesis* **by** (*metis flat-snth less*(*2*,*3*) *snth.simps*(*1*))
    next
      case *False*
      hence *x* = *flat* (*stl s*) !! (*y* − *length* (*shd s*)) **by** (*metis less*(*2*,*3*) *flat-snth*)
      moreover
      { from *less*(*2*) **have** ∗: *length* (*shd s*) > *0* **by** (*cases s*) *simp-all*
        with *False* **have** *y* > *0* **by** (*cases y*) *simp-all*
        with ∗ **have** *y* − *length* (*shd s*) < *y* **by** *simp*
      }
      moreover have ∀ *xs* ∈ *sset* (*stl s*). *xs* ≠ [] **using** *less*(*2*) **by** (*cases s*) *auto*
      ultimately have ∃ *n m′*. *x* = *stl s* !! *n* ! *m′* ∧ *m′* < *length* (*stl s* !! *n*) **by** (*intro less*(*1*)) *auto*
      thus *?thesis* **by** (*metis snth.simps*(*2*))
    qed
  qed
  thus *x* ∈ *?R* **by** (*auto simp*: *sset-range dest*!: *nth-mem*)
next
  fix *x xs* **assume** *xs* ∈ *sset s ?P x* ∈ *set xs* **thus** *x* ∈ *?L*
    by (*induct rule*: *sset-induct*)

(*metis UnI1 flat-unfold shift.simps(1) sset-shift*,
metis UnI2 flat-unfold shd-sset stl-sset sset-shift)
qed

55.10 merge a stream of streams

definition *smerge* :: *'a stream stream* ⇒ *'a stream* **where**
smerge ss = *flat* (*smap* (λ*n*. *map* (λ*s*. *s* !! *n*) (*stake* (*Suc n*) *ss*) @ *stake n* (*ss* !! *n*)) *nats*)

lemma *stake-nth*[*simp*]: *m* < *n* ⟹ *stake n s* ! *m* = *s* !! *m*
by (*induct n arbitrary*: *s m*) (*auto simp*: *nth-Cons'*, *metis Suc-pred snth.simps*(*2*))

lemma *snth-sset-smerge*: *ss* !! *n* !! *m* ∈ *sset* (*smerge ss*)
proof (*cases n* ≤ *m*)
case *False* **thus** *?thesis* **unfolding** *smerge-def*
by (*subst sset-flat*)
(*auto simp*: *stream.set-map in-set-conv-nth simp del*: *stake.simps*
intro!: *exI*[*of* - *n*, *OF disjI2*] *exI*[*of* - *m*, *OF mp*])
next
case *True* **thus** *?thesis* **unfolding** *smerge-def*
by (*subst sset-flat*)
(*auto simp*: *stream.set-map in-set-conv-nth image-iff simp del*: *stake.simps snth.simps*
intro!: *exI*[*of* - *m*, *OF disjI1*] *bexI*[*of* - *ss* !! *n*] *exI*[*of* - *n*, *OF mp*])
qed

lemma *sset-smerge*: *sset* (*smerge ss*) = *UNION* (*sset ss*) *sset*
proof *safe*
fix *x* **assume** *x* ∈ *sset* (*smerge ss*)
thus *x* ∈ *UNION* (*sset ss*) *sset*
unfolding *smerge-def* **by** (*subst* (*asm*) *sset-flat*)
(*auto simp*: *stream.set-map in-set-conv-nth sset-range simp del*: *stake.simps*, *fast*+)
next
fix *s x* **assume** *s* ∈ *sset ss x* ∈ *sset s*
thus *x* ∈ *sset* (*smerge ss*) **using** *snth-sset-smerge* **by** (*auto simp*: *sset-range*)
qed

55.11 product of two streams

definition *sproduct* :: *'a stream* ⇒ *'b stream* ⇒ (*'a* × *'b*) *stream* **where**
sproduct s1 s2 = *smerge* (*smap* (λ*x*. *smap* (*Pair x*) *s2*) *s1*)

lemma *sset-sproduct*: *sset* (*sproduct s1 s2*) = *sset s1* × *sset s2*
unfolding *sproduct-def sset-smerge* **by** (*auto simp*: *stream.set-map*)

55.12 interleave two streams

primcorec *sinterleave* **where**

shd (*sinterleave s1 s2*) = *shd s1*
| *stl* (*sinterleave s1 s2*) = *sinterleave s2* (*stl s1*)

lemma *sinterleave-code*[*code*]:
sinterleave (*x* ## *s1*) *s2* = *x* ## *sinterleave s2 s1*
by (*subst sinterleave.ctr*) *simp*

lemma *sinterleave-snth*[*simp*]:
even n ⟹ *sinterleave s1 s2* !! *n* = *s1* !! (*n div 2*)
odd n ⟹ *sinterleave s1 s2* !! *n* = *s2* !! (*n div 2*)
by (*induct n arbitrary*: *s1 s2*) *simp-all*

lemma *sset-sinterleave*: *sset* (*sinterleave s1 s2*) = *sset s1* ∪ *sset s2*
proof (*intro equalityI subsetI*)
fix *x* **assume** *x* ∈ *sset* (*sinterleave s1 s2*)
then obtain *n* **where** *x* = *sinterleave s1 s2* !! *n* **unfolding** *sset-range* **by** *blast*
thus *x* ∈ *sset s1* ∪ *sset s2* **by** (*cases even n*) *auto*
next
fix *x* **assume** *x* ∈ *sset s1* ∪ *sset s2*
thus *x* ∈ *sset* (*sinterleave s1 s2*)
proof
assume *x* ∈ *sset s1*
then obtain *n* **where** *x* = *s1* !! *n* **unfolding** *sset-range* **by** *blast*
hence *sinterleave s1 s2* !! (*2* ∗ *n*) = *x* **by** *simp*
thus *?thesis* **unfolding** *sset-range* **by** *blast*
next
assume *x* ∈ *sset s2*
then obtain *n* **where** *x* = *s2* !! *n* **unfolding** *sset-range* **by** *blast*
hence *sinterleave s1 s2* !! (*2* ∗ *n* + *1*) = *x* **by** *simp*
thus *?thesis* **unfolding** *sset-range* **by** *blast*
qed
qed

55.13 zip

primcorec *szip* **where**
shd (*szip s1 s2*) = (*shd s1*, *shd s2*)
| *stl* (*szip s1 s2*) = *szip* (*stl s1*) (*stl s2*)

lemma *szip-unfold*[*code*]: *szip* (*a* ## *s1*) (*b* ## *s2*) = (*a*, *b*) ## (*szip s1 s2*)
by (*subst szip.ctr*) *simp*

lemma *snth-szip*[*simp*]: *szip s1 s2* !! *n* = (*s1* !! *n*, *s2* !! *n*)
by (*induct n arbitrary*: *s1 s2*) *auto*

lemma *stake-szip*[*simp*]:
stake n (*szip s1 s2*) = *zip* (*stake n s1*) (*stake n s2*)
by (*induct n arbitrary*: *s1 s2*) *auto*

lemma *sdrop-szip*[*simp*]: *sdrop n* (*szip s1 s2*) = *szip* (*sdrop n s1*) (*sdrop n s2*)
by (*induct n arbitrary*: *s1 s2*) *auto*

lemma *smap-szip-fst*:
smap (λ*x*. *f* (*fst x*)) (*szip s1 s2*) = *smap f s1*
by (*coinduction arbitrary*: *s1 s2*) *auto*

lemma *smap-szip-snd*:
smap (λ*x*. *g* (*snd x*)) (*szip s1 s2*) = *smap g s2*
by (*coinduction arbitrary*: *s1 s2*) *auto*

55.14 zip via function

primcorec *smap2* **where**
shd (*smap2 f s1 s2*) = *f* (*shd s1*) (*shd s2*)
| *stl* (*smap2 f s1 s2*) = *smap2 f* (*stl s1*) (*stl s2*)

lemma *smap2-unfold*[*code*]:
smap2 f (*a* ## *s1*) (*b* ## *s2*) = *f a b* ## (*smap2 f s1 s2*)
by (*subst smap2.ctr*) *simp*

lemma *smap2-szip*:
smap2 f s1 s2 = *smap* (*case-prod f*) (*szip s1 s2*)
by (*coinduction arbitrary*: *s1 s2*) *auto*

lemma *smap-smap2*[*simp*]:
smap f (*smap2 g s1 s2*) = *smap2* (λ*x y*. *f* (*g x y*)) *s1 s2*
unfolding *smap2-szip stream.map-comp o-def split-def* **..**

lemma *smap2-alt*:
(*smap2 f s1 s2* = *s*) = (∀ *n*. *f* (*s1* !! *n*) (*s2* !! *n*) = *s* !! *n*)
unfolding *smap2-szip smap-alt* **by** *auto*

lemma *snth-smap2*[*simp*]:
smap2 f s1 s2 !! *n* = *f* (*s1* !! *n*) (*s2* !! *n*)
by (*induct n arbitrary*: *s1 s2*) *auto*

lemma *stake-smap2*[*simp*]:
stake n (*smap2 f s1 s2*) = *map* (*case-prod f*) (*zip* (*stake n s1*) (*stake n s2*))
by (*induct n arbitrary*: *s1 s2*) *auto*

lemma *sdrop-smap2*[*simp*]:
sdrop n (*smap2 f s1 s2*) = *smap2 f* (*sdrop n s1*) (*sdrop n s2*)
by (*induct n arbitrary*: *s1 s2*) *auto*

end

56 List prefixes, suffixes, and homeomorphic embedding

theory *Sublist*
imports *Main*
begin

56.1 Prefix order on lists

definition *prefixeq* :: *'a list* ⇒ *'a list* ⇒ *bool*
  where *prefixeq xs ys* ⟷ (∃ *zs*. *ys* = *xs* @ *zs*)

definition *prefix* :: *'a list* ⇒ *'a list* ⇒ *bool*
  where *prefix xs ys* ⟷ *prefixeq xs ys* ∧ *xs* ≠ *ys*

interpretation *prefix-order*: *order prefixeq prefix*
  by *standard* (*auto simp*: *prefixeq-def prefix-def*)

interpretation *prefix-bot*: *order-bot Nil prefixeq prefix*
  by *standard* (*simp add*: *prefixeq-def*)

lemma *prefixeqI* [*intro?*]: *ys* = *xs* @ *zs* ⟹ *prefixeq xs ys*
  unfolding *prefixeq-def* **by** *blast*

lemma *prefixeqE* [*elim?*]:
  assumes *prefixeq xs ys*
  obtains *zs* **where** *ys* = *xs* @ *zs*
  using *assms* **unfolding** *prefixeq-def* **by** *blast*

lemma *prefixI'* [*intro?*]: *ys* = *xs* @ *z* # *zs* ⟹ *prefix xs ys*
  unfolding *prefix-def prefixeq-def* **by** *blast*

lemma *prefixE'* [*elim?*]:
  assumes *prefix xs ys*
  obtains *z zs* **where** *ys* = *xs* @ *z* # *zs*
proof −
  from ⟨*prefix xs ys*⟩ **obtain** *us* **where** *ys* = *xs* @ *us* **and** *xs* ≠ *ys*
    unfolding *prefix-def prefixeq-def* **by** *blast*
  with *that* **show** *?thesis* **by** (*auto simp add*: *neq-Nil-conv*)
qed

lemma *prefixI* [*intro?*]: *prefixeq xs ys* ⟹ *xs* ≠ *ys* ⟹ *prefix xs ys*
  unfolding *prefix-def* **by** *blast*

lemma *prefixE* [*elim?*]:
  fixes *xs ys* :: *'a list*
  assumes *prefix xs ys*
  obtains *prefixeq xs ys* **and** *xs* ≠ *ys*
  using *assms* **unfolding** *prefix-def* **by** *blast*

56.2 Basic properties of prefixes

theorem *Nil-prefixeq* [*iff*]: *prefixeq* [] *xs*
 by (*simp add*: *prefixeq-def*)

theorem *prefixeq-Nil* [*simp*]: (*prefixeq xs* []) = (*xs* = [])
 by (*induct xs*) (*simp-all add*: *prefixeq-def*)

lemma *prefixeq-snoc* [*simp*]: *prefixeq xs* (*ys* @ [*y*]) ⟷ *xs* = *ys* @ [*y*] ∨ *prefixeq xs ys*
proof
 assume *prefixeq xs* (*ys* @ [*y*])
 then obtain *zs* **where** *zs*: *ys* @ [*y*] = *xs* @ *zs* **..**
 show *xs* = *ys* @ [*y*] ∨ *prefixeq xs ys*
  by (*metis append-Nil2 butlast-append butlast-snoc prefixeqI zs*)
next
 assume *xs* = *ys* @ [*y*] ∨ *prefixeq xs ys*
 then show *prefixeq xs* (*ys* @ [*y*])
  by (*metis prefix-order.eq-iff prefix-order.order-trans prefixeqI*)
qed

lemma *Cons-prefixeq-Cons* [*simp*]: *prefixeq* (*x* # *xs*) (*y* # *ys*) = (*x* = *y* ∧ *prefixeq xs ys*)
 by (*auto simp add*: *prefixeq-def*)

lemma *prefixeq-code* [*code*]:
 prefixeq [] *xs* ⟷ *True*
 prefixeq (*x* # *xs*) [] ⟷ *False*
 prefixeq (*x* # *xs*) (*y* # *ys*) ⟷ *x* = *y* ∧ *prefixeq xs ys*
 by *simp-all*

lemma *same-prefixeq-prefixeq* [*simp*]: *prefixeq* (*xs* @ *ys*) (*xs* @ *zs*) = *prefixeq ys zs*
 by (*induct xs*) *simp-all*

lemma *same-prefixeq-nil* [*iff*]: *prefixeq* (*xs* @ *ys*) *xs* = (*ys* = [])
 by (*metis append-Nil2 append-self-conv prefix-order.eq-iff prefixeqI*)

lemma *prefixeq-prefixeq* [*simp*]: *prefixeq xs ys* ⟹ *prefixeq xs* (*ys* @ *zs*)
 by (*metis prefix-order.le-less-trans prefixeqI prefixE prefixI*)

lemma *append-prefixeqD*: *prefixeq* (*xs* @ *ys*) *zs* ⟹ *prefixeq xs zs*
 by (*auto simp add*: *prefixeq-def*)

theorem *prefixeq-Cons*: *prefixeq xs* (*y* # *ys*) = (*xs* = [] ∨ (∃ *zs*. *xs* = *y* # *zs* ∧ *prefixeq zs ys*))
 by (*cases xs*) (*auto simp add*: *prefixeq-def*)

theorem *prefixeq-append*:
 prefixeq xs (*ys* @ *zs*) = (*prefixeq xs ys* ∨ (∃ *us*. *xs* = *ys* @ *us* ∧ *prefixeq us zs*))
 apply (*induct zs rule*: *rev-induct*)

apply *force*
apply (*simp del*: *append-assoc add*: *append-assoc* [*symmetric*])
apply (*metis append-eq-appendI*)
done

lemma *append-one-prefixeq*:
prefixeq xs ys $\Longrightarrow$ *length xs* $<$ *length ys* $\Longrightarrow$ *prefixeq* (*xs* @ [*ys* ! *length xs*]) *ys*
proof (*unfold prefixeq-def*)
assume *a1*: $\exists$ *zs*. *ys* = *xs* @ *zs*
then obtain *sk* :: *'a list* **where** *sk*: *ys* = *xs* @ *sk* **by** *fastforce*
assume *a2*: *length xs* $<$ *length ys*
have *f1*: $\bigwedge$*v*. ([]::*'a list*) @ *v* = *v* **using** *append-Nil2* **by** *simp*
have [] $\neq$ *sk* **using** *a1 a2 sk less-not-refl* **by** *force*
hence $\exists$ *v*. *xs* @ *hd sk* # *v* = *ys* **using** *sk* **by** (*metis hd-Cons-tl*)
thus $\exists$ *zs*. *ys* = (*xs* @ [*ys* ! *length xs*]) @ *zs* **using** *f1* **by** *fastforce*
qed

theorem *prefixeq-length-le*: *prefixeq xs ys* $\Longrightarrow$ *length xs* $\leq$ *length ys*
by (*auto simp add*: *prefixeq-def*)

lemma *prefixeq-same-cases*:
prefixeq (xs_1::*'a list*) *ys* $\Longrightarrow$ *prefixeq* xs_2 *ys* $\Longrightarrow$ *prefixeq* xs_1 xs_2 $\vee$ *prefixeq* xs_2 xs_1
unfolding *prefixeq-def* **by** (*force simp*: *append-eq-append-conv2*)

lemma *set-mono-prefixeq*: *prefixeq xs ys* $\Longrightarrow$ *set xs* $\subseteq$ *set ys*
by (*auto simp add*: *prefixeq-def*)

lemma *take-is-prefixeq*: *prefixeq* (*take n xs*) *xs*
unfolding *prefixeq-def* **by** (*metis append-take-drop-id*)

lemma *map-prefixeqI*: *prefixeq xs ys* $\Longrightarrow$ *prefixeq* (*map f xs*) (*map f ys*)
by (*auto simp*: *prefixeq-def*)

lemma *prefixeq-length-less*: *prefix xs ys* $\Longrightarrow$ *length xs* $<$ *length ys*
by (*auto simp*: *prefix-def prefixeq-def*)

lemma *prefix-simps* [*simp*, *code*]:
prefix xs [] $\longleftrightarrow$ *False*
prefix [] (*x* # *xs*) $\longleftrightarrow$ *True*
prefix (*x* # *xs*) (*y* # *ys*) $\longleftrightarrow$ *x* = *y* $\wedge$ *prefix xs ys*
by (*simp-all add*: *prefix-def cong*: *conj-cong*)

lemma *take-prefix*: *prefix xs ys* $\Longrightarrow$ *prefix* (*take n xs*) *ys*
apply (*induct n arbitrary*: *xs ys*)
apply (*case-tac ys*; *simp*)
apply (*metis prefix-order.less-trans prefixI take-is-prefixeq*)
done

```
lemma not-prefixeq-cases:
  assumes pfx: ¬ prefixeq ps ls
  obtains
    (c1) ps ≠ [] and ls = []
  | (c2) a as x xs where ps = a#as and ls = x#xs and x = a and ¬ prefixeq
as xs
  | (c3) a as x xs where ps = a#as and ls = x#xs and x ≠ a
proof (cases ps)
  case Nil
  then show ?thesis using pfx by simp
next
  case (Cons a as)
  note c = ‹ps = a#as›
  show ?thesis
  proof (cases ls)
    case Nil then show ?thesis by (metis append-Nil2 pfx c1 same-prefixeq-nil)
  next
    case (Cons x xs)
    show ?thesis
    proof (cases x = a)
      case True
      have ¬ prefixeq as xs using pfx c Cons True by simp
      with c Cons True show ?thesis by (rule c2)
    next
      case False
      with c Cons show ?thesis by (rule c3)
    qed
  qed
qed

lemma not-prefixeq-induct [consumes 1, case-names Nil Neq Eq]:
  assumes np: ¬ prefixeq ps ls
    and base: ⋀x xs. P (x#xs) []
    and r1: ⋀x xs y ys. x ≠ y ⟹ P (x#xs) (y#ys)
    and r2: ⋀x xs y ys. ⟦ x = y; ¬ prefixeq xs ys; P xs ys ⟧ ⟹ P (x#xs) (y#ys)
  shows P ps ls using np
proof (induct ls arbitrary: ps)
  case Nil then show ?case
    by (auto simp: neq-Nil-conv elim!: not-prefixeq-cases intro!: base)
next
  case (Cons y ys)
  then have npfx: ¬ prefixeq ps (y # ys) by simp
  then obtain x xs where pv: ps = x # xs
    by (rule not-prefixeq-cases) auto
  show ?case by (metis Cons.hyps Cons-prefixeq-Cons npfx pv r1 r2)
qed
```

56.3 Parallel lists

definition *parallel* :: *'a list* ⇒ *'a list* ⇒ *bool* (**infixl** ‖ *50*)
  where (*xs* ‖ *ys*) = (¬ *prefixeq xs ys* ∧ ¬ *prefixeq ys xs*)

lemma *parallelI* [*intro*]: ¬ *prefixeq xs ys* ⟹ ¬ *prefixeq ys xs* ⟹ *xs* ‖ *ys*
  unfolding *parallel-def* **by** *blast*

lemma *parallelE* [*elim*]:
  assumes *xs* ‖ *ys*
  obtains ¬ *prefixeq xs ys* ∧ ¬ *prefixeq ys xs*
  using *assms* **unfolding** *parallel-def* **by** *blast*

theorem *prefixeq-cases*:
  obtains *prefixeq xs ys* | *prefix ys xs* | *xs* ‖ *ys*
  unfolding *parallel-def prefix-def* **by** *blast*

theorem *parallel-decomp*:
  xs ‖ *ys* ⟹ ∃ *as b bs c cs*. *b* ≠ *c* ∧ *xs* = *as* @ *b* # *bs* ∧ *ys* = *as* @ *c* # *cs*
proof (*induct xs rule*: *rev-induct*)
  case *Nil*
  then have *False* **by** *auto*
  then show *?case* **..**
next
  case (*snoc x xs*)
  show *?case*
  proof (*rule prefixeq-cases*)
    assume *le*: *prefixeq xs ys*
    then obtain *ys'* **where** *ys*: *ys* = *xs* @ *ys'* **..**
    show *?thesis*
    proof (*cases ys'*)
      assume *ys'* = []
      then show *?thesis* **by** (*metis append-Nil2 parallelE prefixeqI snoc.prems ys*)
    next
      fix *c cs* **assume** *ys'*: *ys'* = *c* # *cs*
      have *x* ≠ *c* **using** *snoc.prems ys ys'* **by** *fastforce*
      thus ∃ *as b bs c cs*. *b* ≠ *c* ∧ *xs* @ [*x*] = *as* @ *b* # *bs* ∧ *ys* = *as* @ *c* # *cs*
        using *ys ys'* **by** *blast*
    qed
  next
    assume *prefix ys xs*
    then have *prefixeq ys* (*xs* @ [*x*]) **by** (*simp add*: *prefix-def*)
    with *snoc* **have** *False* **by** *blast*
    then show *?thesis* **..**
  next
    assume *xs* ‖ *ys*
    with *snoc* **obtain** *as b bs c cs* **where** *neq*: (*b*::*'a*) ≠ *c*
      and *xs*: *xs* = *as* @ *b* # *bs* **and** *ys*: *ys* = *as* @ *c* # *cs*
      by *blast*
    from *xs* **have** *xs* @ [*x*] = *as* @ *b* # (*bs* @ [*x*]) **by** *simp*

with *neq ys* **show** *?thesis* **by** *blast*
qed
qed

lemma *parallel-append*: $a \parallel b \Longrightarrow a\ @\ c \parallel b\ @\ d$
apply (*rule parallelI*)
apply (*erule parallelE*, *erule conjE*,
induct rule: *not-prefixeq-induct*, *simp*+)+
done

lemma *parallel-appendI*: $xs \parallel ys \Longrightarrow x = xs\ @\ xs' \Longrightarrow y = ys\ @\ ys' \Longrightarrow x \parallel y$
by (*simp add*: *parallel-append*)

lemma *parallel-commute*: $a \parallel b \longleftrightarrow b \parallel a$
unfolding *parallel-def* **by** *auto*

56.4 Suffix order on lists

definition *suffixeq* :: $'a\ list \Rightarrow 'a\ list \Rightarrow bool$
where *suffixeq xs ys* = ($\exists$ *zs*. *ys* = *zs* @ *xs*)

definition *suffix* :: $'a\ list \Rightarrow 'a\ list \Rightarrow bool$
where *suffix xs ys* $\longleftrightarrow$ ($\exists$ *us*. *ys* = *us* @ *xs* $\wedge$ *us* $\neq$ [])

lemma *suffix-imp-suffixeq*:
suffix xs ys $\Longrightarrow$ *suffixeq xs ys*
by (*auto simp*: *suffixeq-def suffix-def*)

lemma *suffixeqI* [*intro?*]: *ys* = *zs* @ *xs* $\Longrightarrow$ *suffixeq xs ys*
unfolding *suffixeq-def* **by** *blast*

lemma *suffixeqE* [*elim?*]:
assumes *suffixeq xs ys*
obtains *zs* **where** *ys* = *zs* @ *xs*
using *assms* **unfolding** *suffixeq-def* **by** *blast*

lemma *suffixeq-refl* [*iff*]: *suffixeq xs xs*
by (*auto simp add*: *suffixeq-def*)
lemma *suffix-trans*:
suffix xs ys $\Longrightarrow$ *suffix ys zs* $\Longrightarrow$ *suffix xs zs*
by (*auto simp*: *suffix-def*)
lemma *suffixeq-trans*: ⟦*suffixeq xs ys*; *suffixeq ys zs*⟧ $\Longrightarrow$ *suffixeq xs zs*
by (*auto simp add*: *suffixeq-def*)
lemma *suffixeq-antisym*: ⟦*suffixeq xs ys*; *suffixeq ys xs*⟧ $\Longrightarrow$ *xs* = *ys*
by (*auto simp add*: *suffixeq-def*)

lemma *suffixeq-tl* [*simp*]: *suffixeq* (*tl xs*) *xs*
by (*induct xs*) (*auto simp*: *suffixeq-def*)

lemma *suffix-tl* [*simp*]: *xs* ≠ [] ⟹ *suffix* (*tl xs*) *xs*
by (*induct xs*) (*auto simp*: *suffix-def*)

lemma *Nil-suffixeq* [*iff*]: *suffixeq* [] *xs*
by (*simp add*: *suffixeq-def*)
lemma *suffixeq-Nil* [*simp*]: (*suffixeq xs* []) = (*xs* = [])
by (*auto simp add*: *suffixeq-def*)

lemma *suffixeq-ConsI*: *suffixeq xs ys* ⟹ *suffixeq xs* (*y* # *ys*)
by (*auto simp add*: *suffixeq-def*)
lemma *suffixeq-ConsD*: *suffixeq* (*x* # *xs*) *ys* ⟹ *suffixeq xs ys*
by (*auto simp add*: *suffixeq-def*)

lemma *suffixeq-appendI*: *suffixeq xs ys* ⟹ *suffixeq xs* (*zs* @ *ys*)
by (*auto simp add*: *suffixeq-def*)
lemma *suffixeq-appendD*: *suffixeq* (*zs* @ *xs*) *ys* ⟹ *suffixeq xs ys*
by (*auto simp add*: *suffixeq-def*)

lemma *suffix-set-subset*:
suffix xs ys ⟹ *set xs* ⊆ *set ys* **by** (*auto simp*: *suffix-def*)

lemma *suffixeq-set-subset*:
suffixeq xs ys ⟹ *set xs* ⊆ *set ys* **by** (*auto simp*: *suffixeq-def*)

lemma *suffixeq-ConsD2*: *suffixeq* (*x* # *xs*) (*y* # *ys*) ⟹ *suffixeq xs ys*
proof −
  assume *suffixeq* (*x* # *xs*) (*y* # *ys*)
  then obtain *zs* **where** *y* # *ys* = *zs* @ *x* # *xs* **..**
  then show *?thesis*
    by (*induct zs*) (*auto intro*!: *suffixeq-appendI suffixeq-ConsI*)
qed

lemma *suffixeq-to-prefixeq* [*code*]: *suffixeq xs ys* ⟷ *prefixeq* (*rev xs*) (*rev ys*)
proof
  assume *suffixeq xs ys*
  then obtain *zs* **where** *ys* = *zs* @ *xs* **..**
  then have *rev ys* = *rev xs* @ *rev zs* **by** *simp*
  then show *prefixeq* (*rev xs*) (*rev ys*) **..**
next
  assume *prefixeq* (*rev xs*) (*rev ys*)
  then obtain *zs* **where** *rev ys* = *rev xs* @ *zs* **..**
  then have *rev* (*rev ys*) = *rev zs* @ *rev* (*rev xs*) **by** *simp*
  then have *ys* = *rev zs* @ *xs* **by** *simp*
  then show *suffixeq xs ys* **..**
qed

lemma *distinct-suffixeq*: *distinct ys* ⟹ *suffixeq xs ys* ⟹ *distinct xs*
by (*clarsimp elim*!: *suffixeqE*)

lemma *suffixeq-map*: *suffixeq xs ys* ⟹ *suffixeq* (*map f xs*) (*map f ys*)
  by (*auto elim*!: *suffixeqE intro*: *suffixeqI*)

lemma *suffixeq-drop*: *suffixeq* (*drop n as*) *as*
  unfolding *suffixeq-def*
  apply (*rule exI* [**where** *x* = *take n as*])
  apply *simp*
  done

lemma *suffixeq-take*: *suffixeq xs ys* ⟹ *ys* = *take* (*length ys* − *length xs*) *ys* @ *xs*
  by (*auto elim*!: *suffixeqE*)

lemma *suffixeq-suffix-reflclp-conv*: *suffixeq* = $suffix^{==}$
proof (*intro ext iffI*)
  fix *xs ys* :: *'a list*
  assume *suffixeq xs ys*
  show $suffix^{==}$ *xs ys*
  proof
    assume *xs* ≠ *ys*
    with ‹*suffixeq xs ys*› **show** *suffix xs ys*
      by (*auto simp*: *suffixeq-def suffix-def*)
  qed
next
  fix *xs ys* :: *'a list*
  assume $suffix^{==}$ *xs ys*
  then show *suffixeq xs ys*
  proof
    assume *suffix xs ys* **then show** *suffixeq xs ys*
      by (*rule suffix-imp-suffixeq*)
  next
    assume *xs* = *ys* **then show** *suffixeq xs ys*
      by (*auto simp*: *suffixeq-def*)
  qed
qed

lemma *parallelD1*: *x* ∥ *y* ⟹ ¬ *prefixeq x y*
  by *blast*

lemma *parallelD2*: *x* ∥ *y* ⟹ ¬ *prefixeq y x*
  by *blast*

lemma *parallel-Nil1* [*simp*]: ¬ *x* ∥ []
  unfolding *parallel-def* **by** *simp*

lemma *parallel-Nil2* [*simp*]: ¬ [] ∥ *x*
  unfolding *parallel-def* **by** *simp*

lemma *Cons-parallelI1*: *a* ≠ *b* ⟹ *a* # *as* ∥ *b* # *bs*
  by *auto*

lemma *Cons-parallelI2*: $[\![a = b; as \parallel bs]\!] \Longrightarrow a \# as \parallel b \# bs$
by (*metis Cons-prefixeq-Cons parallelE parallelI*)

lemma *not-equal-is-parallel*:
assumes *neq*: $xs \neq ys$
and *len*: *length xs* = *length ys*
shows $xs \parallel ys$
using *len neq*
proof (*induct rule*: *list-induct2*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a as b bs*)
have *ih*: $as \neq bs \Longrightarrow as \parallel bs$ **by** *fact*
show *?case*
proof (*cases a* = *b*)
case *True*
then have $as \neq bs$ **using** *Cons* **by** *simp*
then show *?thesis* **by** (*rule Cons-parallelI2* [*OF True ih*])
next
case *False*
then show *?thesis* **by** (*rule Cons-parallelI1*)
qed
qed

lemma *suffix-reflclp-conv*: $suffix^{==} = suffixeq$
by (*intro ext*) (*auto simp*: *suffixeq-def suffix-def*)

lemma *suffix-lists*: *suffix xs ys* $\Longrightarrow ys \in$ *lists A* $\Longrightarrow xs \in$ *lists A*
unfolding *suffix-def* **by** *auto*

56.5 Homeomorphic embedding on lists

inductive *list-emb* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a\ list \Rightarrow bool$
for P :: $('a \Rightarrow 'a \Rightarrow bool)$
where
list-emb-Nil [*intro*, *simp*]: *list-emb P* [] *ys*
| *list-emb-Cons* [*intro*] : *list-emb P xs ys* $\Longrightarrow$ *list-emb P xs* (*y*#*ys*)
| *list-emb-Cons2* [*intro*]: *P x y* $\Longrightarrow$ *list-emb P xs ys* $\Longrightarrow$ *list-emb P* (*x*#*xs*) (*y*#*ys*)

lemma *list-emb-mono*:
assumes $\bigwedge x\ y.\ P\ x\ y \longrightarrow Q\ x\ y$
shows *list-emb P xs ys* $\longrightarrow$ *list-emb Q xs ys*
proof
assume *list-emb P xs ys*
then show *list-emb Q xs ys* **by** (*induct*) (*auto simp*: *assms*)
qed

lemma *list-emb-Nil2* [*simp*]:
  assumes *list-emb P xs* [] **shows** *xs* = []
  using *assms* **by** (*cases rule*: *list-emb.cases*) *auto*

lemma *list-emb-refl*:
  assumes $\bigwedge x.\ x \in set\ xs \Longrightarrow P\ x\ x$
  shows *list-emb P xs xs*
  using *assms* **by** (*induct xs*) *auto*

lemma *list-emb-Cons-Nil* [*simp*]: *list-emb P* (*x#xs*) [] = *False*
proof −
  { assume *list-emb P* (*x#xs*) []
    from *list-emb-Nil2* [*OF this*] **have** *False* **by** *simp*
  } moreover {
    assume *False*
    then have *list-emb P* (*x#xs*) [] **by** *simp*
  } ultimately show *?thesis* **by** *blast*
qed

lemma *list-emb-append2* [*intro*]: *list-emb P xs ys* $\Longrightarrow$ *list-emb P xs* (*zs* @ *ys*)
  by (*induct zs*) *auto*

lemma *list-emb-prefix* [*intro*]:
  assumes *list-emb P xs ys* **shows** *list-emb P xs* (*ys* @ *zs*)
  using *assms*
  by (*induct arbitrary*: *zs*) *auto*

lemma *list-emb-ConsD*:
  assumes *list-emb P* (*x#xs*) *ys*
  shows ∃ *us v vs*. *ys* = *us* @ *v* # *vs* ∧ *P x v* ∧ *list-emb P xs vs*
using *assms*
proof (*induct x* ≡ *x* # *xs ys arbitrary*: *x xs*)
  case *list-emb-Cons*
  then show *?case* **by** (*metis append-Cons*)
next
  case (*list-emb-Cons2 x y xs ys*)
  then show *?case* **by** *blast*
qed

lemma *list-emb-appendD*:
  assumes *list-emb P* (*xs* @ *ys*) *zs*
  shows ∃ *us vs*. *zs* = *us* @ *vs* ∧ *list-emb P xs us* ∧ *list-emb P ys vs*
using *assms*
proof (*induction xs arbitrary*: *ys zs*)
  case *Nil* **then show** *?case* **by** *auto*
next
  case (*Cons x xs*)
  then obtain *us v vs* **where**
    zs: *zs* = *us* @ *v* # *vs* **and** *p*: *P x v* **and** *lh*: *list-emb P* (*xs* @ *ys*) *vs*

by (*auto dest*: *list-emb-ConsD*)
obtain sk_0 :: $'a$ *list* $\Rightarrow$ $'a$ *list* $\Rightarrow$ $'a$ *list* **and** sk_1 :: $'a$ *list* $\Rightarrow$ $'a$ *list* $\Rightarrow$ $'a$ *list* **where**
sk: $\forall x_0\ x_1.\ \neg$ *list-emb P* (xs @ x_0) $x_1 \lor sk_0\ x_0\ x_1$ @ $sk_1\ x_0\ x_1 = x_1 \land$ *list-emb P xs* ($sk_0\ x_0\ x_1$) $\land$ *list-emb P* x_0 ($sk_1\ x_0\ x_1$)
using *Cons*(*1*) **by** (*metis* (*no-types*))
hence $\forall x_2.$ *list-emb P* (x # xs) (x_2 @ v # sk_0 *ys vs*) **using** *p lh* **by** *auto*
thus *?case* **using** *lh zs sk* **by** (*metis* (*no-types*) *append-Cons append-assoc*)
qed

lemma *list-emb-suffix*:
assumes *list-emb P xs ys* **and** *suffix ys zs*
shows *list-emb P xs zs*
using *assms*(*2*) **and** *list-emb-append2* [*OF assms*(*1*)] **by** (*auto simp*: *suffix-def*)

lemma *list-emb-suffixeq*:
assumes *list-emb P xs ys* **and** *suffixeq ys zs*
shows *list-emb P xs zs*
using *assms* **and** *list-emb-suffix* **unfolding** *suffixeq-suffix-reflclp-conv* **by** *auto*

lemma *list-emb-length*: *list-emb P xs ys* $\Longrightarrow$ *length xs* $\leq$ *length ys*
by (*induct rule*: *list-emb.induct*) *auto*

lemma *list-emb-trans*:
assumes $\bigwedge x\ y\ z.$ $[\![x \in$ *set xs*; $y \in$ *set ys*; $z \in$ *set zs*; $P\ x\ y$; $P\ y\ z]\!] \Longrightarrow P\ x\ z$
shows $[\![$*list-emb P xs ys*; *list-emb P ys zs*$]\!] \Longrightarrow$ *list-emb P xs zs*
proof –
assume *list-emb P xs ys* **and** *list-emb P ys zs*
then show *list-emb P xs zs* **using** *assms*
proof (*induction arbitrary*: *zs*)
case *list-emb-Nil* **show** *?case* **by** *blast*
next
case (*list-emb-Cons xs ys y*)
from *list-emb-ConsD* [*OF* ‹*list-emb P* (*y#ys*) *zs*›] **obtain** *us v vs*
where *zs*: *zs* = *us* @ *v* # *vs* **and** $P^{==}$ *y v* **and** *list-emb P ys vs* **by** *blast*
then have *list-emb P ys* (*v#vs*) **by** *blast*
then have *list-emb P ys zs* **unfolding** *zs* **by** (*rule list-emb-append2*)
from *list-emb-Cons.IH* [*OF this*] **and** *list-emb-Cons.prems* **show** *?case* **by** *auto*
next
case (*list-emb-Cons2 x y xs ys*)
from *list-emb-ConsD* [*OF* ‹*list-emb P* (*y#ys*) *zs*›] **obtain** *us v vs*
where *zs*: *zs* = *us* @ *v* # *vs* **and** *P y v* **and** *list-emb P ys vs* **by** *blast*
with *list-emb-Cons2* **have** *list-emb P xs vs* **by** *auto*
moreover have *P x v*
proof –
from *zs* **have** *v* $\in$ *set zs* **by** *auto*
moreover have *x* $\in$ *set* (*x#xs*) **and** *y* $\in$ *set* (*y#ys*) **by** *simp-all*
ultimately show *?thesis*

using ‹*P x y*› **and** ‹*P y v*› **and** *list-emb-Cons2*
by *blast*
qed
ultimately have *list-emb P* (*x#xs*) (*v#vs*) **by** *blast*
then show *?case* **unfolding** *zs* **by** (*rule list-emb-append2*)
qed
qed

lemma *list-emb-set*:
assumes *list-emb P xs ys* **and** $x \in$ *set xs*
obtains *y* **where** $y \in$ *set ys* **and** *P x y*
using *assms* **by** (*induct*) *auto*

56.6 Sublists (special case of homeomorphic embedding)

abbreviation *sublisteq* :: *'a list* ⇒ *'a list* ⇒ *bool*
where *sublisteq xs ys* ≡ *list-emb* (*op* =) *xs ys*

lemma *sublisteq-Cons2*: *sublisteq xs ys* ⟹ *sublisteq* (*x#xs*) (*x#ys*) **by** *auto*

lemma *sublisteq-same-length*:
assumes *sublisteq xs ys* **and** *length xs* = *length ys* **shows** *xs* = *ys*
using *assms* **by** (*induct*) (*auto dest*: *list-emb-length*)

lemma *not-sublisteq-length* [*simp*]: *length ys* < *length xs* ⟹ ¬ *sublisteq xs ys*
by (*metis list-emb-length linorder-not-less*)

lemma [*code*]:
list-emb P [] *ys* ⟷ *True*
list-emb P (*x#xs*) [] ⟷ *False*
by (*simp-all*)

lemma *sublisteq-Cons'*: *sublisteq* (*x#xs*) *ys* ⟹ *sublisteq xs ys*
by (*induct xs*, *simp*, *blast dest*: *list-emb-ConsD*)

lemma *sublisteq-Cons2'*:
assumes *sublisteq* (*x#xs*) (*x#ys*) **shows** *sublisteq xs ys*
using *assms* **by** (*cases*) (*rule sublisteq-Cons'*)

lemma *sublisteq-Cons2-neq*:
assumes *sublisteq* (*x#xs*) (*y#ys*)
shows *x* ≠ *y* ⟹ *sublisteq* (*x#xs*) *ys*
using *assms* **by** (*cases*) *auto*

lemma *sublisteq-Cons2-iff* [*simp*, *code*]:
sublisteq (*x#xs*) (*y#ys*) = (*if x* = *y then sublisteq xs ys else sublisteq* (*x#xs*) *ys*)
by (*metis list-emb-Cons sublisteq-Cons2 sublisteq-Cons2' sublisteq-Cons2-neq*)

lemma *sublisteq-append'*: *sublisteq* (*zs* @ *xs*) (*zs* @ *ys*) ⟷ *sublisteq xs ys*

by (*induct zs*) *simp-all*

lemma *sublisteq-refl* [*simp*, *intro*!]: *sublisteq xs xs* **by** (*induct xs*) *simp-all*

lemma *sublisteq-antisym*:
  assumes *sublisteq xs ys* **and** *sublisteq ys xs*
  shows *xs* = *ys*
using *assms*
proof (*induct*)
  case *list-emb-Nil*
  from *list-emb-Nil2* [*OF this*] **show** *?case* **by** *simp*
next
  case *list-emb-Cons2*
  thus *?case* **by** *simp*
next
  case *list-emb-Cons*
  hence *False* **using** *sublisteq-Cons′* **by** *fastforce*
  thus *?case* **..**
qed

lemma *sublisteq-trans*: *sublisteq xs ys* $\Longrightarrow$ *sublisteq ys zs* $\Longrightarrow$ *sublisteq xs zs*
  by (*rule list-emb-trans* [*of - - - op* =]) *auto*

lemma *sublisteq-append-le-same-iff*: *sublisteq* (*xs* @ *ys*) *ys* $\longleftrightarrow$ *xs* = []
  by (*auto dest*: *list-emb-length*)

lemma *list-emb-append-mono*:
  $[\![$ *list-emb P xs xs′*; *list-emb P ys ys′* $]\!]$ $\Longrightarrow$ *list-emb P* (*xs*@*ys*) (*xs′*@*ys′*)
  apply (*induct rule*: *list-emb.induct*)
    apply (*metis eq-Nil-appendI list-emb-append2*)
   apply (*metis append-Cons list-emb-Cons*)
  apply (*metis append-Cons list-emb-Cons2*)
  done

56.7 Appending elements

lemma *sublisteq-append* [*simp*]:
  sublisteq (*xs* @ *zs*) (*ys* @ *zs*) $\longleftrightarrow$ *sublisteq xs ys* (**is** *?l* = *?r*)
proof
  { **fix** *xs′ ys′ xs ys zs* :: *′a list* **assume** *sublisteq xs′ ys′*
    then have *xs′* = *xs* @ *zs* & *ys′* = *ys* @ *zs* $\longrightarrow$ *sublisteq xs ys*
    proof (*induct arbitrary*: *xs ys zs*)
      case *list-emb-Nil* **show** *?case* **by** *simp*
    next
      case (*list-emb-Cons xs′ ys′ x*)
      { **assume** *ys*=[] **then have** *?case* **using** *list-emb-Cons*(*1*) **by** *auto* }
      moreover
      { **fix** *us* **assume** *ys* = *x*#*us*
     then have *?case* **using** *list-emb-Cons*(*2*) **by**(*simp add*: *list-emb.list-emb-Cons*)

}
ultimately **show** *?case* **by** (*auto simp*:*Cons-eq-append-conv*)
next
case (*list-emb-Cons2 x y xs′ ys′*)
{ **assume** *xs*=[] **then have** *?case* **using** *list-emb-Cons2*(*1*) **by** *auto* }
moreover
{ **fix** *us vs* **assume** *xs*=*x*#*us ys*=*x*#*vs* **then have** *?case* **using** *list-emb-Cons2* **by** *auto*}
moreover
{ **fix** *us* **assume** *xs*=*x*#*us ys*=[] **then have** *?case* **using** *list-emb-Cons2*(*2*) **by** *bestsimp* }
ultimately show *?case* **using** ‹*op* = *x y*› **by** (*auto simp*: *Cons-eq-append-conv*)
qed }
moreover assume *?l*
ultimately show *?r* **by** *blast*
next
assume *?r* **then show** *?l* **by** (*metis list-emb-append-mono sublisteq-refl*)
qed

lemma *sublisteq-drop-many*: *sublisteq xs ys* ⟹ *sublisteq xs* (*zs* @ *ys*)
by (*induct zs*) *auto*

lemma *sublisteq-rev-drop-many*: *sublisteq xs ys* ⟹ *sublisteq xs* (*ys* @ *zs*)
by (*metis append-Nil2 list-emb-Nil list-emb-append-mono*)

56.8 Relation to standard list operations

lemma *sublisteq-map*:
assumes *sublisteq xs ys* **shows** *sublisteq* (*map f xs*) (*map f ys*)
using *assms* **by** (*induct*) *auto*

lemma *sublisteq-filter-left* [*simp*]: *sublisteq* (*filter P xs*) *xs*
by (*induct xs*) *auto*

lemma *sublisteq-filter* [*simp*]:
assumes *sublisteq xs ys* **shows** *sublisteq* (*filter P xs*) (*filter P ys*)
using *assms* **by** *induct auto*

lemma *sublisteq xs ys* ⟷ (∃ *N*. *xs* = *sublist ys N*) (**is** *?L* = *?R*)
proof
assume *?L*
then show *?R*
proof (*induct*)
case *list-emb-Nil* **show** *?case* **by** (*metis sublist-empty*)
next
case (*list-emb-Cons xs ys x*)
then obtain *N* **where** *xs* = *sublist ys N* **by** *blast*
then have *xs* = *sublist* (*x*#*ys*) (*Suc* ‘ *N*)
by (*clarsimp simp add*:*sublist-Cons inj-image-mem-iff*)

```
    then show ?case by blast
  next
    case (list-emb-Cons2 x y xs ys)
    then obtain N where xs = sublist ys N by blast
    then have x#xs = sublist (x#ys) (insert 0 (Suc ‘ N))
      by (clarsimp simp add:sublist-Cons inj-image-mem-iff)
    moreover from list-emb-Cons2 have x = y by simp
    ultimately show ?case by blast
  qed
next
  assume ?R
  then obtain N where xs = sublist ys N ..
  moreover have sublisteq (sublist ys N) ys
  proof (induct ys arbitrary: N)
    case Nil show ?case by simp
  next
    case Cons then show ?case by (auto simp: sublist-Cons)
  qed
  ultimately show ?L by simp
qed

end
```

57 Linear Temporal Logic on Streams

```
theory Linear-Temporal-Logic-on-Streams
  imports Stream Sublist Extended-Nat Infinite-Set
begin
```

58 Preliminaries

```
lemma shift-prefix:
assumes xl @- xs = yl @- ys and length xl ≤ length yl
shows prefixeq xl yl
using assms proof(induct xl arbitrary: yl xs ys)
  case (Cons x xl yl xs ys)
  thus ?case by (cases yl) auto
qed auto

lemma shift-prefix-cases:
assumes xl @- xs = yl @- ys
shows prefixeq xl yl ∨ prefixeq yl xl
using shift-prefix[OF assms]
by (cases length xl ≤ length yl) (metis, metis assms nat-le-linear shift-prefix)
```

59 Linear temporal logic

```
abbreviation (input) IMPL (infix impl 60)
```

where φ *impl* $\psi \equiv \lambda$ *xs*. φ *xs* $\longrightarrow \psi$ *xs*

abbreviation (*input*) *OR* (**infix** *or 60*)
where φ *or* $\psi \equiv \lambda$ *xs*. φ *xs* $\vee$ ψ *xs*

abbreviation (*input*) *AND* (**infix** *aand 60*)
where φ *aand* $\psi \equiv \lambda$ *xs*. φ *xs* $\wedge$ ψ *xs*

abbreviation (*input*) *not* $\varphi \equiv \lambda$ *xs*. $\neg$ φ *xs*

abbreviation (*input*) *true* $\equiv \lambda$ *xs*. *True*

abbreviation (*input*) *false* $\equiv \lambda$ *xs*. *False*

lemma *impl-not-or*: φ *impl* ψ = (*not* φ) *or* ψ
by *blast*

lemma *not-or*: *not* (φ *or* ψ) = (*not* φ) *aand* (*not* ψ)
by *blast*

lemma *not-aand*: *not* (φ *aand* ψ) = (*not* φ) *or* (*not* ψ)
by *blast*

lemma *non-not*[*simp*]: *not* (*not* φ) = φ **by** *simp*

fun *holds* **where** *holds P xs* $\longleftrightarrow$ *P* (*shd xs*)
fun *nxt* **where** *nxt* φ *xs* = φ (*stl xs*)

definition *HLD s* = *holds* (λx. $x \in s$)

abbreviation *HLD-nxt* (**infixr** $\cdot$ *65*) **where**
s $\cdot$ *P* $\equiv$ *HLD s aand nxt P*

context
notes [[*inductive-internals*]]
begin

inductive *ev* **for** φ **where**
base: φ *xs* $\Longrightarrow$ *ev* φ *xs*
|
step: *ev* φ (*stl xs*) $\Longrightarrow$ *ev* φ *xs*

coinductive *alw* **for** φ **where**
alw: $[\![\varphi$ *xs*; *alw* φ (*stl xs*)$]\!]$ $\Longrightarrow$ *alw* φ *xs*

coinductive *UNTIL* (**infix** *until 60*) **for** φ ψ **where**
base: ψ *xs* $\Longrightarrow$ (φ *until* ψ) *xs*

|
step: $[\![\varphi\ xs;\ (\varphi\ until\ \psi)\ (stl\ xs)]\!] \Longrightarrow (\varphi\ until\ \psi)\ xs$

end

lemma *holds-mono*:
assumes *holds*: *holds P xs* **and** *0*: $\bigwedge x.\ P\ x \Longrightarrow Q\ x$
shows *holds Q xs*
using *assms* **by** *auto*

lemma *holds-aand*:
$(holds\ P\ aand\ holds\ Q)\ steps \longleftrightarrow holds\ (\lambda\ step.\ P\ step \wedge Q\ step)\ steps$ **by** *auto*

lemma *HLD-iff*: $HLD\ s\ \omega \longleftrightarrow shd\ \omega \in s$
by (*simp add*: *HLD-def*)

lemma *HLD-Stream*[*simp*]: $HLD\ X\ (x\ \#\#\ \omega) \longleftrightarrow x \in X$
by (*simp add*: *HLD-iff*)

lemma *nxt-mono*:
assumes *nxt*: $nxt\ \varphi\ xs$ **and** *0*: $\bigwedge xs.\ \varphi\ xs \Longrightarrow \psi\ xs$
shows $nxt\ \psi\ xs$
using *assms* **by** *auto*

declare *ev.intros*[*intro*]
declare *alw.cases*[*elim*]

lemma *ev-induct-strong*[*consumes 1*, *case-names base step*]:
$ev\ \varphi\ x \Longrightarrow (\bigwedge xs.\ \varphi\ xs \Longrightarrow P\ xs) \Longrightarrow (\bigwedge xs.\ ev\ \varphi\ (stl\ xs) \Longrightarrow \neg\ \varphi\ xs \Longrightarrow P\ (stl\ xs) \Longrightarrow P\ xs) \Longrightarrow P\ x$
by (*induct rule*: *ev.induct*) *auto*

lemma *alw-coinduct*[*consumes 1*, *case-names alw stl*]:
$X\ x \Longrightarrow (\bigwedge x.\ X\ x \Longrightarrow \varphi\ x) \Longrightarrow (\bigwedge x.\ X\ x \Longrightarrow \neg\ alw\ \varphi\ (stl\ x) \Longrightarrow X\ (stl\ x)) \Longrightarrow alw\ \varphi\ x$
using *alw.coinduct*[*of X x* φ] **by** *auto*

lemma *ev-mono*:
assumes *ev*: $ev\ \varphi\ xs$ **and** *0*: $\bigwedge xs.\ \varphi\ xs \Longrightarrow \psi\ xs$
shows $ev\ \psi\ xs$
using *ev* **by** *induct* (*auto simp*: *0*)

lemma *alw-mono*:
assumes *alw*: $alw\ \varphi\ xs$ **and** *0*: $\bigwedge xs.\ \varphi\ xs \Longrightarrow \psi\ xs$
shows $alw\ \psi\ xs$
using *alw* **by** *coinduct* (*auto simp*: *0*)

lemma *until-monoL*:
assumes *until*: $(\varphi 1\ until\ \psi)\ xs$ **and** *0*: $\bigwedge xs.\ \varphi 1\ xs \Longrightarrow \varphi 2\ xs$

shows ($\varphi 2$ *until* ψ) *xs*
using *until* **by** *coinduct* (*auto elim*: *UNTIL.cases simp*: *0*)

lemma *until-monoR*:
assumes *until*: (φ *until* $\psi 1$) *xs* **and** *0*: $\bigwedge$ *xs*. $\psi 1$ *xs* $\Longrightarrow$ $\psi 2$ *xs*
shows (φ *until* $\psi 2$) *xs*
using *until* **by** *coinduct* (*auto elim*: *UNTIL.cases simp*: *0*)

lemma *until-mono*:
assumes *until*: ($\varphi 1$ *until* $\psi 1$) *xs* **and**
0: $\bigwedge$ *xs*. $\varphi 1$ *xs* $\Longrightarrow$ $\varphi 2$ *xs* $\bigwedge$ *xs*. $\psi 1$ *xs* $\Longrightarrow$ $\psi 2$ *xs*
shows ($\varphi 2$ *until* $\psi 2$) *xs*
using *until* **by** *coinduct* (*auto elim*: *UNTIL.cases simp*: *0*)

lemma *until-false*: φ *until false* = *alw* φ
proof−
  {**fix** *xs* **assume** (φ *until false*) *xs* **hence** *alw* φ *xs*
   by *coinduct* (*auto elim*: *UNTIL.cases*)
  }
  moreover
  {**fix** *xs* **assume** *alw* φ *xs* **hence** (φ *until false*) *xs*
   by *coinduct auto*
  }
  ultimately show *?thesis* **by** *blast*
qed

lemma *ev-nxt*: *ev* φ = (φ *or nxt* (*ev* φ))
by (*rule ext*) (*metis ev.simps nxt.simps*)

lemma *alw-nxt*: *alw* φ = (φ *aand nxt* (*alw* φ))
by (*rule ext*) (*metis alw.simps nxt.simps*)

lemma *ev-ev*[*simp*]: *ev* (*ev* φ) = *ev* φ
proof−
  {**fix** *xs*
   assume *ev* (*ev* φ) *xs* **hence** *ev* φ *xs*
   by *induct auto*
  }
  thus *?thesis* **by** *auto*
qed

lemma *alw-alw*[*simp*]: *alw* (*alw* φ) = *alw* φ
proof−
  {**fix** *xs*
   assume *alw* φ *xs* **hence** *alw* (*alw* φ) *xs*
   by *coinduct auto*
  }
  thus *?thesis* **by** *auto*
qed

lemma *ev-shift*:
assumes *ev* φ *xs*
shows *ev* φ (*xl* @− *xs*)
using *assms* **by** (*induct xl*) *auto*

lemma *ev-imp-shift*:
assumes *ev* φ *xs* **shows** ∃ *xl xs2*. *xs* = *xl* @− *xs2* ∧ φ *xs2*
using *assms* **by** *induct* (*metis shift.simps*(*1*), *metis shift.simps*(*2*) *stream.collapse*)+

lemma *alw-ev-shift*: *alw* φ *xs1* ⟹ *ev* (*alw* φ) (*xl* @− *xs1*)
by (*auto intro*: *ev-shift*)

lemma *alw-shift*:
assumes *alw* φ (*xl* @− *xs*)
shows *alw* φ *xs*
using *assms* **by** (*induct xl*) *auto*

lemma *ev-ex-nxt*:
assumes *ev* φ *xs*
shows ∃ *n*. (*nxt* ^^ *n*) φ *xs*
using *assms* **proof** *induct*
  case (*base xs*) **thus** *?case* **by** (*intro exI*[*of* - *0*]) *auto*
next
  case (*step xs*)
  then obtain *n* **where** (*nxt* ^^ *n*) φ (*stl xs*) **by** *blast*
  thus *?case* **by** (*intro exI*[*of* - *Suc n*]) (*metis funpow.simps*(*2*) *nxt.simps o-def*)
qed

lemma *alw-sdrop*:
assumes *alw* φ *xs* **shows** *alw* φ (*sdrop n xs*)
by (*metis alw-shift assms stake-sdrop*)

lemma *nxt-sdrop*: (*nxt* ^^ *n*) φ *xs* ⟷ φ (*sdrop n xs*)
by (*induct n arbitrary*: *xs*) *auto*

definition *wait* φ *xs* ≡ *LEAST n*. (*nxt* ^^ *n*) φ *xs*

lemma *nxt-wait*:
assumes *ev* φ *xs* **shows** (*nxt* ^^ (*wait* φ *xs*)) φ *xs*
unfolding *wait-def* **using** *ev-ex-nxt*[*OF assms*] **by**(*rule LeastI-ex*)

lemma *nxt-wait-least*:
assumes *ev*: *ev* φ *xs* **and** *nxt*: (*nxt* ^^ *n*) φ *xs* **shows** *wait* φ *xs* ≤ *n*
unfolding *wait-def* **using** *ev-ex-nxt*[*OF ev*] **by** (*metis Least-le nxt*)

lemma *sdrop-wait*:
assumes *ev* φ *xs* **shows** φ (*sdrop* (*wait* φ *xs*) *xs*)
using *nxt-wait*[*OF assms*] **unfolding** *nxt-sdrop* **.**

lemma *sdrop-wait-least*:
assumes *ev*: *ev* φ *xs* **and** *nxt*: φ (*sdrop n xs*) **shows** *wait* φ *xs* $\leq n$
using *assms nxt-wait-least* **unfolding** *nxt-sdrop* **by** *auto*

lemma *nxt-ev*: (*nxt* ^^ *n*) φ *xs* $\Longrightarrow$ *ev* φ *xs*
by (*induct n arbitrary*: *xs*) *auto*

lemma *not-ev*: *not* (*ev* φ) = *alw* (*not* φ)
proof(*rule ext*, *safe*)
  fix *xs* **assume** *not* (*ev* φ) *xs* **thus** *alw* (*not* φ) *xs*
  by (*coinduct*) *auto*
next
  fix *xs* **assume** *ev* φ *xs* **and** *alw* (*not* φ) *xs* **thus** *False*
  by (*induct*) *auto*
qed

lemma *not-alw*: *not* (*alw* φ) = *ev* (*not* φ)
proof−
  have *not* (*alw* φ) = *not* (*alw* (*not* (*not* φ))) **by** *simp*
  also have ... = *ev* (*not* φ) **unfolding** *not-ev*[*symmetric*] **by** *simp*
  finally show *?thesis* **.**
qed

lemma *not-ev-not*[*simp*]: *not* (*ev* (*not* φ)) = *alw* φ
unfolding *not-ev* **by** *simp*

lemma *not-alw-not*[*simp*]: *not* (*alw* (*not* φ)) = *ev* φ
unfolding *not-alw* **by** *simp*

lemma *alw-ev-sdrop*:
assumes *alw* (*ev* φ) (*sdrop m xs*)
shows *alw* (*ev* φ) *xs*
using *assms*
by *coinduct* (*metis alw-nxt ev-shift funpow-swap1 nxt.simps nxt-sdrop stake-sdrop*)

lemma *ev-alw-imp-alw-ev*:
assumes *ev* (*alw* φ) *xs* **shows** *alw* (*ev* φ) *xs*
using *assms* **by** *induct* (*metis* (*full-types*) *alw-mono ev.base*, *metis alw alw-nxt ev.step*)

lemma *alw-aand*: *alw* (φ *aand* ψ) = *alw* φ *aand alw* ψ
proof−
  {fix *xs* **assume** *alw* (φ *aand* ψ) *xs* **hence** (*alw* φ *aand alw* ψ) *xs*
   by (*auto elim*: *alw-mono*)
  }
  moreover
  {fix *xs* **assume** (*alw* φ *aand alw* ψ) *xs* **hence** *alw* (φ *aand* ψ) *xs*
   by *coinduct auto*

```
  }
  ultimately show ?thesis by blast
qed

lemma ev-or: ev (φ or ψ) = ev φ or ev ψ
proof−
  {fix xs assume (ev φ or ev ψ) xs hence ev (φ or ψ) xs
   by (auto elim: ev-mono)
  }
  moreover
  {fix xs assume ev (φ or ψ) xs hence (ev φ or ev ψ) xs
   by induct auto
  }
  ultimately show ?thesis by blast
qed

lemma ev-alw-aand:
assumes φ: ev (alw φ) xs and ψ: ev (alw ψ) xs
shows ev (alw (φ aand ψ)) xs
proof−
  obtain xl xs1 where xs1: xs = xl @− xs1 and φφ: alw φ xs1
  using φ by (metis ev-imp-shift)
  moreover obtain yl ys1 where xs2: xs = yl @− ys1 and ψψ: alw ψ ys1
  using ψ by (metis ev-imp-shift)
  ultimately have 0: xl @− xs1 = yl @− ys1 by auto
  hence prefixeq xl yl ∨ prefixeq yl xl using shift-prefix-cases by auto
  thus ?thesis proof
    assume prefixeq xl yl
    then obtain yl1 where yl: yl = xl @ yl1 by (elim prefixeqE)
    have xs1': xs1 = yl1 @− ys1 using 0 unfolding yl by simp
    have alw φ ys1 using φφ unfolding xs1' by (metis alw-shift)
    hence alw (φ aand ψ) ys1 using ψψ unfolding alw-aand by auto
    thus ?thesis unfolding xs2 by (auto intro: alw-ev-shift)
  next
    assume prefixeq yl xl
    then obtain xl1 where xl: xl = yl @ xl1 by (elim prefixeqE)
    have ys1': ys1 = xl1 @− xs1 using 0 unfolding xl by simp
    have alw ψ xs1 using ψψ unfolding ys1' by (metis alw-shift)
    hence alw (φ aand ψ) xs1 using φφ unfolding alw-aand by auto
    thus ?thesis unfolding xs1 by (auto intro: alw-ev-shift)
  qed
qed

lemma ev-alw-alw-impl:
assumes ev (alw φ) xs and alw (alw φ impl ev ψ) xs
shows ev ψ xs
using assms by induct auto

lemma ev-alw-stl[simp]: ev (alw φ) (stl x) ⟷ ev (alw φ) x
```

by (*metis* (*full-types*) *alw-nxt ev-nxt nxt.simps*)

lemma *alw-alw-impl-ev*:
alw (*alw* φ *impl ev* ψ) = (*ev* (*alw* φ) *impl alw* (*ev* ψ)) (**is** *?A* = *?B*)
proof−
  {**fix** *xs* **assume** *?A xs* ∧ *ev* (*alw* φ) *xs* **hence** *alw* (*ev* ψ) *xs*
    by *coinduct* (*auto elim*: *ev-alw-alw-impl*)
  }
  moreover
  {**fix** *xs* **assume** *?B xs* **hence** *?A xs*
   by *coinduct auto*
  }
  ultimately show *?thesis* **by** *blast*
qed

lemma *ev-alw-impl*:
assumes *ev* φ *xs* **and** *alw* (φ *impl* ψ) *xs* **shows** *ev* ψ *xs*
using *assms* **by** *induct auto*

lemma *ev-alw-impl-ev*:
assumes *ev* φ *xs* **and** *alw* (φ *impl ev* ψ) *xs* **shows** *ev* ψ *xs*
using *ev-alw-impl*[*OF assms*] **by** *simp*

lemma *alw-mp*:
assumes *alw* φ *xs* **and** *alw* (φ *impl* ψ) *xs*
shows *alw* ψ *xs*
proof−
  {**assume** *alw* φ *xs* ∧ *alw* (φ *impl* ψ) *xs* **hence** *?thesis*
   by *coinduct auto*
  }
  thus *?thesis* **using** *assms* **by** *auto*
qed

lemma *all-imp-alw*:
assumes $\bigwedge$ *xs*. φ *xs* **shows** *alw* φ *xs*
proof−
  {**assume** ∀ *xs*. φ *xs*
   hence *?thesis* **by** *coinduct auto*
  }
  thus *?thesis* **using** *assms* **by** *auto*
qed

lemma *alw-impl-ev-alw*:
assumes *alw* (φ *impl ev* ψ) *xs*
shows *alw* (*ev* φ *impl ev* ψ) *xs*
using *assms* **by** *coinduct* (*auto dest*: *ev-alw-impl*)

lemma *ev-holds-sset*:
ev (*holds P*) *xs* ⟷ (∃ *x* ∈ *sset xs*. *P x*) (**is** *?L* ⟷ *?R*)

proof *safe*
 assume *?L* **thus** *?R* **by** *induct* (*metis holds.simps stream.set-sel*(*1*), *metis stl-sset*)
next
 fix *x* **assume** $x \in$ *sset xs P x*
 thus *?L* **by** (*induct rule*: *sset-induct*) (*simp-all add*: *ev.base ev.step*)
qed

lemma *alw-invar*:
assumes φ *xs* **and** *alw* (φ *impl nxt* φ) *xs*
shows *alw* φ *xs*
proof–
 {**assume** φ *xs* $\wedge$ *alw* (φ *impl nxt* φ) *xs* **hence** *?thesis*
 by *coinduct auto*
 }
 thus *?thesis* **using** *assms* **by** *auto*
qed

lemma *variance*:
assumes *1*: φ *xs* **and** *2*: *alw* (φ *impl* (ψ *or nxt* φ)) *xs*
shows (*alw* φ *or ev* ψ) *xs*
proof–
 {**assume** $\neg$ *ev* ψ *xs* **hence** *alw* (*not* ψ) *xs* **unfolding** *not-ev*[*symmetric*] **.**
 moreover have *alw* (*not* ψ *impl* (φ *impl nxt* φ)) *xs*
 using *2* **by** *coinduct auto*
 ultimately have *alw* (φ *impl nxt* φ) *xs* **by**(*auto dest*: *alw-mp*)
 with *1* **have** *alw* φ *xs* **by**(*rule alw-invar*)
 }
 thus *?thesis* **by** *blast*
qed

lemma *ev-alw-imp-nxt*:
assumes *e*: *ev* φ *xs* **and** *a*: *alw* (φ *impl* (*nxt* φ)) *xs*
shows *ev* (*alw* φ) *xs*
proof–
 obtain *xl xs1* **where** *xs*: *xs* = *xl* @– *xs1* **and** φ: φ *xs1*
 using *e* **by** (*metis ev-imp-shift*)
 have φ *xs1* $\wedge$ *alw* (φ *impl* (*nxt* φ)) *xs1* **using** *a* φ **unfolding** *xs* **by** (*metis alw-shift*)
 hence *alw* φ *xs1* **by**(*coinduct xs1 rule*: *alw.coinduct*) *auto*
 thus *?thesis* **unfolding** *xs* **by** (*auto intro*: *alw-ev-shift*)
qed

inductive *ev-at* :: (*'a stream* $\Rightarrow$ *bool*) $\Rightarrow$ *nat* $\Rightarrow$ *'a stream* $\Rightarrow$ *bool* **for** *P* :: *'a stream* $\Rightarrow$ *bool* **where**
 base: *P* $\omega \Longrightarrow$ *ev-at P 0* ω
| *step*:$\neg$ *P* $\omega \Longrightarrow$ *ev-at P n* (*stl* ω) $\Longrightarrow$ *ev-at P* (*Suc n*) ω

inductive-simps *ev-at-0*[*simp*]: *ev-at P 0 ω*
inductive-simps *ev-at-Suc*[*simp*]: *ev-at P* (*Suc n*) *ω*

lemma *ev-at-imp-snth*: *ev-at P n ω* ⟹ *P* (*sdrop n ω*)
  by (*induction n arbitrary*: *ω*) *auto*

lemma *ev-at-HLD-imp-snth*: *ev-at* (*HLD X*) *n ω* ⟹ *ω* !! *n* ∈ *X*
  by (*auto dest*!: *ev-at-imp-snth simp*: *HLD-iff*)

lemma *ev-at-HLD-single-imp-snth*: *ev-at* (*HLD* {*x*}) *n ω* ⟹ *ω* !! *n* = *x*
  by (*drule ev-at-HLD-imp-snth*) *simp*

lemma *ev-at-unique*: *ev-at P n ω* ⟹ *ev-at P m ω* ⟹ *n* = *m*
proof (*induction arbitrary*: *m rule*: *ev-at.induct*)
  case (*base ω*) **then show** *?case*
    by (*simp add*: *ev-at.simps*[*of - - ω*])
next
  case (*step ω n*) **from** *step.prems step.hyps step.IH*[*of m − 1*] **show** *?case*
    by (*auto simp add*: *ev-at.simps*[*of - - ω*])
qed

lemma *ev-iff-ev-at*: *ev P ω* ⟷ (∃ *n*. *ev-at P n ω*)
proof
  assume *ev P ω* **then show** ∃ *n*. *ev-at P n ω*
    by (*induction rule*: *ev-induct-strong*) (*auto intro*: *ev-at.intros*)
next
  assume ∃ *n*. *ev-at P n ω*
  then obtain *n* **where** *ev-at P n ω*
    by *auto*
  then show *ev P ω*
    by *induction auto*
qed

lemma *ev-at-shift*: *ev-at* (*HLD X*) *i* (*stake* (*Suc i*) *ω* @− *ω′* :: *′s stream*) ⟷ *ev-at* (*HLD X*) *i ω*
  by (*induction i arbitrary*: *ω*) (*auto simp*: *HLD-iff*)

lemma *ev-iff-ev-at-unqiue*: *ev P ω* ⟷ (∃!*n*. *ev-at P n ω*)
  by (*auto intro*: *ev-at-unique simp*: *ev-iff-ev-at*)

lemma *alw-HLD-iff-streams*: *alw* (*HLD X*) *ω* ⟷ *ω* ∈ *streams X*
proof
  assume *alw* (*HLD X*) *ω* **then show** *ω* ∈ *streams X*
  proof (*coinduction arbitrary*: *ω*)
    case (*streams ω*) **then show** *?case* **by** (*cases ω*) *auto*
  qed
next
  assume *ω* ∈ *streams X* **then show** *alw* (*HLD X*) *ω*
  proof (*coinduction arbitrary*: *ω*)

case (*alw* ω) **then show** *?case* **by** (*cases* ω) *auto*
qed
qed

lemma *not-HLD*: *not* (*HLD X*) = *HLD* (− *X*)
by (*auto simp*: *HLD-iff*)

lemma *not-alw-iff*: ¬ (*alw P* ω) ⟷ *ev* (*not P*) ω
using *not-alw*[*of P*] **by** (*simp add*: *fun-eq-iff*)

lemma *not-ev-iff*: ¬ (*ev P* ω) ⟷ *alw* (*not P*) ω
using *not-alw-iff*[*of not P* ω, *symmetric*] **by** *simp*

lemma *ev-Stream*: *ev P* (*x* ## *s*) ⟷ *P* (*x* ## *s*) ∨ *ev P s*
by (*auto elim*: *ev.cases*)

lemma *alw-ev-imp-ev-alw*:
assumes *alw* (*ev P*) ω **shows** *ev* (*P aand alw* (*ev P*)) ω
proof −
have *ev P* ω **using** *assms* **by** *auto*
from *this assms* **show** *?thesis*
by *induct auto*
qed

lemma *ev-False*: *ev* (λ*x*. *False*) ω ⟷ *False*
proof
assume *ev* (λ*x*. *False*) ω **then show** *False*
by *induct auto*
qed *auto*

lemma *alw-False*: *alw* (λ*x*. *False*) ω ⟷ *False*
by *auto*

lemma *ev-iff-sdrop*: *ev P* ω ⟷ (∃ *m*. *P* (*sdrop m* ω))
proof *safe*
assume *ev P* ω **then show** ∃ *m*. *P* (*sdrop m* ω)
by (*induct rule*: *ev-induct-strong*) (*auto intro*: *exI*[*of - 0*] *exI*[*of - Suc n* **for** *n*])
next
fix *m* **assume** *P* (*sdrop m* ω) **then show** *ev P* ω
by (*induct m arbitrary*: ω) *auto*
qed

lemma *alw-iff-sdrop*: *alw P* ω ⟷ (∀ *m*. *P* (*sdrop m* ω))
proof *safe*
fix *m* **assume** *alw P* ω **then show** *P* (*sdrop m* ω)
by (*induct m arbitrary*: ω) *auto*
next
assume ∀ *m*. *P* (*sdrop m* ω) **then show** *alw P* ω
by (*coinduction arbitrary*: ω) (*auto elim*: *allE*[*of - 0*] *allE*[*of - Suc n* **for** *n*])

qed

lemma *infinite-iff-alw-ev*: *infinite* $\{m.\ P\ (sdrop\ m\ \omega)\} \longleftrightarrow alw\ (ev\ P)\ \omega$
  unfolding *infinite-nat-iff-unbounded-le alw-iff-sdrop ev-iff-sdrop*
  by *simp* (*metis le-Suc-ex le-add1*)

lemma *alw-inv*:
  assumes *stl*: $\bigwedge s.\ f\ (stl\ s) = stl\ (f\ s)$
  shows $alw\ P\ (f\ s) \longleftrightarrow alw\ (\lambda x.\ P\ (f\ x))\ s$
proof
  assume $alw\ P\ (f\ s)$ **then show** $alw\ (\lambda x.\ P\ (f\ x))\ s$
    by (*coinduction arbitrary*: *s rule*: *alw-coinduct*)
      (*auto simp*: *stl*)
next
  assume $alw\ (\lambda x.\ P\ (f\ x))\ s$ **then show** $alw\ P\ (f\ s)$
    by (*coinduction arbitrary*: *s rule*: *alw-coinduct*) (*auto simp*: *stl*[*symmetric*])
qed

lemma *ev-inv*:
  assumes *stl*: $\bigwedge s.\ f\ (stl\ s) = stl\ (f\ s)$
  shows $ev\ P\ (f\ s) \longleftrightarrow ev\ (\lambda x.\ P\ (f\ x))\ s$
proof
  assume $ev\ P\ (f\ s)$ **then show** $ev\ (\lambda x.\ P\ (f\ x))\ s$
    by (*induction f s arbitrary*: *s*) (*auto simp*: *stl*)
next
  assume $ev\ (\lambda x.\ P\ (f\ x))\ s$ **then show** $ev\ P\ (f\ s)$
    by *induction* (*auto simp*: *stl*[*symmetric*])
qed

lemma *alw-smap*: $alw\ P\ (smap\ f\ s) \longleftrightarrow alw\ (\lambda x.\ P\ (smap\ f\ x))\ s$
  by (*rule alw-inv*) *simp*

lemma *ev-smap*: $ev\ P\ (smap\ f\ s) \longleftrightarrow ev\ (\lambda x.\ P\ (smap\ f\ x))\ s$
  by (*rule ev-inv*) *simp*

lemma *alw-cong*:
  assumes *P*: $alw\ P\ \omega$ **and** *eq*: $\bigwedge\omega.\ P\ \omega \Longrightarrow Q1\ \omega \longleftrightarrow Q2\ \omega$
  shows $alw\ Q1\ \omega \longleftrightarrow alw\ Q2\ \omega$
proof –
  from *eq* **have** $(alw\ P\ aand\ Q1) = (alw\ P\ aand\ Q2)$ **by** *auto*
  then have $alw\ (alw\ P\ aand\ Q1)\ \omega = alw\ (alw\ P\ aand\ Q2)\ \omega$ **by** *auto*
  with *P* **show** $alw\ Q1\ \omega \longleftrightarrow alw\ Q2\ \omega$
    by (*simp add*: *alw-aand*)
qed

lemma *ev-cong*:
  assumes *P*: $alw\ P\ \omega$ **and** *eq*: $\bigwedge\omega.\ P\ \omega \Longrightarrow Q1\ \omega \longleftrightarrow Q2\ \omega$
  shows $ev\ Q1\ \omega \longleftrightarrow ev\ Q2\ \omega$
proof –

from P **have** *alw* ($\lambda xs.\ Q1\ xs \longrightarrow Q2\ xs$) ω **by** (*rule alw-mono*) (*simp add*: *eq*)
moreover from P **have** *alw* ($\lambda xs.\ Q2\ xs \longrightarrow Q1\ xs$) ω **by** (*rule alw-mono*) (*simp add*: *eq*)
moreover note *ev-alw-impl*[*of Q1 ω Q2*] *ev-alw-impl*[*of Q2 ω Q1*]
ultimately show *ev Q1* $\omega \longleftrightarrow$ *ev Q2* ω
by *auto*
qed

lemma *alwD*: *alw P x* $\Longrightarrow$ *P x*
by *auto*

lemma *alw-alwD*: *alw P* $\omega \Longrightarrow$ *alw* (*alw P*) ω
by *simp*

lemma *alw-ev-stl*: *alw* (*ev P*) (*stl* ω) $\longleftrightarrow$ *alw* (*ev P*) ω
by (*auto intro*: *alw.intros*)

lemma *holds-Stream*: *holds P* (*x ## s*) $\longleftrightarrow$ *P x*
by *simp*

lemma *holds-eq1*[*simp*]: *holds* (*op* = *x*) = *HLD* {*x*}
by *rule* (*auto simp*: *HLD-iff*)

lemma *holds-eq2*[*simp*]: *holds* ($\lambda y.\ y = x$) = *HLD* {*x*}
by *rule* (*auto simp*: *HLD-iff*)

lemma *not-holds-eq*[*simp*]: *holds* (− *op* = *x*) = *not* (*HLD* {*x*})
by *rule* (*auto simp*: *HLD-iff*)

Strong until

context
notes [[*inductive-internals*]]
begin

inductive *suntil* (**infix** *suntil 60*) **for** $\varphi\ \psi$ **where**
base: $\psi\ \omega \Longrightarrow (\varphi$ *suntil* $\psi)\ \omega$
| *step*: $\varphi\ \omega \Longrightarrow (\varphi$ *suntil* $\psi)$ (*stl* ω) $\Longrightarrow (\varphi$ *suntil* $\psi)\ \omega$

inductive-simps *suntil-Stream*: (φ *suntil* ψ) (*x ## s*)

end

lemma *suntil-induct-strong*[*consumes 1*, *case-names base step*]:
(φ *suntil* ψ) $x \Longrightarrow$
($\bigwedge\omega.\ \psi\ \omega \Longrightarrow P\ \omega$) $\Longrightarrow$
($\bigwedge\omega.\ \varphi\ \omega \Longrightarrow \neg\ \psi\ \omega \Longrightarrow$ (φ *suntil* ψ) (*stl* ω) $\Longrightarrow$ *P* (*stl* ω) $\Longrightarrow P\ \omega$) $\Longrightarrow P\ x$
using *suntil.induct*[*of* $\varphi\ \psi$ *x P*] **by** *blast*

lemma *ev-suntil*: (φ *suntil* ψ) $\omega \Longrightarrow$ *ev* $\psi\ \omega$

by (*induct rule*: *suntil.induct*) *auto*

lemma *suntil-inv*:
assumes *stl*: $\bigwedge s.\ f\ (stl\ s) = stl\ (f\ s)$
shows $(P\ suntil\ Q)\ (f\ s) \longleftrightarrow ((\lambda x.\ P\ (f\ x))\ suntil\ (\lambda x.\ Q\ (f\ x)))\ s$
proof
assume $(P\ suntil\ Q)\ (f\ s)$ **then show** $((\lambda x.\ P\ (f\ x))\ suntil\ (\lambda x.\ Q\ (f\ x)))\ s$
by (*induction f s arbitrary*: *s*) (*auto simp*: *stl intro*: *suntil.intros*)
next
assume $((\lambda x.\ P\ (f\ x))\ suntil\ (\lambda x.\ Q\ (f\ x)))\ s$ **then show** $(P\ suntil\ Q)\ (f\ s)$
by *induction* (*auto simp*: *stl*[*symmetric*] *intro*: *suntil.intros*)
qed

lemma *suntil-smap*: $(P\ suntil\ Q)\ (smap\ f\ s) \longleftrightarrow ((\lambda x.\ P\ (smap\ f\ x))\ suntil\ (\lambda x.\ Q\ (smap\ f\ x)))\ s$
by (*rule suntil-inv*) *simp*

lemma *hld-smap*: $HLD\ x\ (smap\ f\ s) = holds\ (\lambda y.\ f\ y \in x)\ s$
by (*simp add*: *HLD-def*)

lemma *suntil-mono*:
assumes *eq*: $\bigwedge\omega.\ P\ \omega \Longrightarrow Q1\ \omega \Longrightarrow Q2\ \omega\ \bigwedge\omega.\ P\ \omega \Longrightarrow R1\ \omega \Longrightarrow R2\ \omega$
assumes $*$: $(Q1\ suntil\ R1)\ \omega\ alw\ P\ \omega$ **shows** $(Q2\ suntil\ R2)\ \omega$
using $*$ **by** *induct* (*auto intro*: *eq suntil.intros*)

lemma *suntil-cong*:
$alw\ P\ \omega \Longrightarrow (\bigwedge\omega.\ P\ \omega \Longrightarrow Q1\ \omega \longleftrightarrow Q2\ \omega) \Longrightarrow (\bigwedge\omega.\ P\ \omega \Longrightarrow R1\ \omega \longleftrightarrow R2\ \omega) \Longrightarrow$
$(Q1\ suntil\ R1)\ \omega \longleftrightarrow (Q2\ suntil\ R2)\ \omega$
using *suntil-mono*[*of* $P\ Q1\ Q2\ R1\ R2\ \omega$] *suntil-mono*[*of* $P\ Q2\ Q1\ R2\ R1\ \omega$] **by** *auto*

lemma *ev-suntil-iff*: $ev\ (P\ suntil\ Q)\ \omega \longleftrightarrow ev\ Q\ \omega$
proof
assume $ev\ (P\ suntil\ Q)\ \omega$ **then show** $ev\ Q\ \omega$
by *induct* (*auto dest*: *ev-suntil*)
next
assume $ev\ Q\ \omega$ **then show** $ev\ (P\ suntil\ Q)\ \omega$
by *induct* (*auto intro*: *suntil.intros*)
qed

lemma *true-suntil*: $((\lambda\text{-}.\ True)\ suntil\ P) = ev\ P$
by (*simp add*: *suntil-def ev-def*)

lemma *suntil-lfp*: $(\varphi\ suntil\ \psi) = lfp\ (\lambda P\ s.\ \psi\ s \lor (\varphi\ s \land P\ (stl\ s)))$
by (*simp add*: *suntil-def*)

lemma *sfilter-P*[*simp*]: $P\ (shd\ s) \Longrightarrow sfilter\ P\ s = shd\ s\ \#\#\ sfilter\ P\ (stl\ s)$
using *sfilter-Stream*[*of P shd s stl s*] **by** *simp*

lemma *sfilter-not-P*[*simp*]: ¬ *P* (*shd s*) ⟹ *sfilter P s* = *sfilter P* (*stl s*)
  using *sfilter-Stream*[*of P shd s stl s*] **by** *simp*

lemma *sfilter-eq*:
  assumes *ev* (*holds P*) *s*
  shows *sfilter P s* = *x* ## *s'* ⟷
    P x ∧ (*not* (*holds P*) *suntil* (*HLD* {*x*} *aand nxt* (λ*s*. *sfilter P s* = *s'*))) *s*
  using *assms*
  by (*induct rule*: *ev-induct-strong*)
    (*auto simp add*: *HLD-iff intro*: *suntil.intros elim*: *suntil.cases*)

lemma *sfilter-streams*:
  alw (*ev* (*holds P*)) ω ⟹ ω ∈ *streams A* ⟹ *sfilter P* ω ∈ *streams* {*x*∈*A*. *P x*}
proof (*coinduction arbitrary*: ω)
  case (*streams* ω)
  then have *ev* (*holds P*) ω **by** *blast*
  from *this streams* **show** *?case*
    by (*induct rule*: *ev-induct-strong*) (*auto elim*: *streamsE*)
qed

lemma *alw-sfilter*:
  assumes ∗: *alw* (*ev* (*holds P*)) *s*
  shows *alw Q* (*sfilter P s*) ⟷ *alw* (λ*x*. *Q* (*sfilter P x*)) *s*
proof
  assume *alw Q* (*sfilter P s*) **with** ∗ **show** *alw* (λ*x*. *Q* (*sfilter P x*)) *s*
  proof (*coinduction arbitrary*: *s rule*: *alw-coinduct*)
    case (*stl s*)
    then have *ev* (*holds P*) *s*
      by *blast*
    from *this stl* **show** *?case*
      by (*induct rule*: *ev-induct-strong*) *auto*
  qed *auto*
next
  assume *alw* (λ*x*. *Q* (*sfilter P x*)) *s* **with** ∗ **show** *alw Q* (*sfilter P s*)
  proof (*coinduction arbitrary*: *s rule*: *alw-coinduct*)
    case (*stl s*)
    then have *ev* (*holds P*) *s*
      by *blast*
    from *this stl* **show** *?case*
      by (*induct rule*: *ev-induct-strong*) *auto*
  qed *auto*
qed

lemma *ev-sfilter*:
  assumes ∗: *alw* (*ev* (*holds P*)) *s*
  shows *ev Q* (*sfilter P s*) ⟷ *ev* (λ*x*. *Q* (*sfilter P x*)) *s*
proof
  assume *ev Q* (*sfilter P s*) **from** *this* ∗ **show** *ev* (λ*x*. *Q* (*sfilter P x*)) *s*

proof (*induction sfilter P s arbitrary*: *s rule*: *ev-induct-strong*)
case (*step s*)
then have *ev* (*holds P*) *s*
by *blast*
from *this step* **show** *?case*
by (*induct rule*: *ev-induct-strong*) *auto*
qed *auto*
next
assume *ev* (λx. *Q* (*sfilter P x*)) *s* **then show** *ev Q* (*sfilter P s*)
proof (*induction rule*: *ev-induct-strong*)
case (*step s*) **then show** *?case*
by (*cases P* (*shd s*)) *auto*
qed *auto*
qed

lemma *holds-sfilter*:
assumes *ev* (*holds Q*) *s* **shows** *holds P* (*sfilter Q s*) ⟷ (*not* (*holds Q*) *suntil* (*holds* (*Q aand P*))) *s*
proof
assume *holds P* (*sfilter Q s*) **with** *assms* **show** (*not* (*holds Q*) *suntil* (*holds* (*Q aand P*))) *s*
by (*induct rule*: *ev-induct-strong*) (*auto intro*: *suntil.intros*)
next
assume (*not* (*holds Q*) *suntil* (*holds* (*Q aand P*))) *s* **then show** *holds P* (*sfilter Q s*)
by *induct auto*
qed

lemma *suntil-aand-nxt*:
(φ *suntil* (φ *aand nxt* ψ)) ω ⟷ (φ *aand nxt* (φ *suntil* ψ)) ω
proof
assume (φ *suntil* (φ *aand nxt* ψ)) ω **then show** (φ *aand nxt* (φ *suntil* ψ)) ω
by *induction* (*auto intro*: *suntil.intros*)
next
assume (φ *aand nxt* (φ *suntil* ψ)) ω
then have (φ *suntil* ψ) (*stl* ω) φ ω
by *auto*
then show (φ *suntil* (φ *aand nxt* ψ)) ω
by (*induction stl* ω *arbitrary*: ω)
(*auto elim*: *suntil.cases intro*: *suntil.intros*)
qed

lemma *alw-sconst*: *alw P* (*sconst x*) ⟷ *P* (*sconst x*)
proof
assume *P* (*sconst x*) **then show** *alw P* (*sconst x*)
by *coinduction auto*
qed *auto*

lemma *ev-sconst*: *ev P* (*sconst x*) ⟷ *P* (*sconst x*)

proof
  assume *ev P* (*sconst x*) **then show** *P* (*sconst x*)
    by (*induction sconst x*) *auto*
qed *auto*

lemma *suntil-sconst*: (φ *suntil* ψ) (*sconst x*) $\longleftrightarrow$ ψ (*sconst x*)
proof
  assume (φ *suntil* ψ) (*sconst x*) **then show** ψ (*sconst x*)
    by (*induction sconst x*) *auto*
qed (*auto intro*: *suntil.intros*)

lemma *hld-smap'*: *HLD x* (*smap f s*) = *HLD* (*f* −' *x*) *s*
  by (*simp add*: *HLD-def*)

end

60 Lists as vectors

theory *ListVector*
imports *List Main*
begin

A vector-space like structure of lists and arithmetic operations on them. Is only a vector space if restricted to lists of the same length.

Multiplication with a scalar:

abbreviation *scale* :: ('*a*::*times*) ⇒ '*a list* ⇒ '*a list* (**infix** $*_s$ *70*)
where *x* $*_s$ *xs* ≡ *map* (*op* ∗ *x*) *xs*

lemma *scale1*[*simp*]: (*1*::'*a*::*monoid-mult*) $*_s$ *xs* = *xs*
by (*induct xs*) *simp-all*

60.1 + and −

fun *zipwith0* :: ('*a*::*zero* ⇒ '*b*::*zero* ⇒ '*c*) ⇒ '*a list* ⇒ '*b list* ⇒ '*c list*
where
zipwith0 f [] [] = [] |
zipwith0 f (*x*#*xs*) (*y*#*ys*) = *f x y* # *zipwith0 f xs ys* |
zipwith0 f (*x*#*xs*) [] = *f x 0* # *zipwith0 f xs* [] |
zipwith0 f [] (*y*#*ys*) = *f 0 y* # *zipwith0 f* [] *ys*

instantiation *list* :: ({*zero*, *plus*}) *plus*
begin

definition
  list-add-def: *op* + = *zipwith0* (*op* +)

instance ..

end

instantiation *list* :: ({*zero*, *uminus*}) *uminus*
begin

definition
list-uminus-def: *uminus* = *map uminus*

instance ..

end

instantiation *list* :: ({*zero*,*minus*}) *minus*
begin

definition
list-diff-def: *op* − = *zipwith0* (*op* −)

instance ..

end

lemma *zipwith0-Nil*[*simp*]: *zipwith0 f* [] *ys* = *map* (*f 0*) *ys*
by(*induct ys*) *simp-all*

lemma *list-add-Nil*[*simp*]: [] + *xs* = (*xs*::′*a*::*monoid-add list*)
by (*induct xs*) (*auto simp*:*list-add-def*)

lemma *list-add-Nil2*[*simp*]: *xs* + [] = (*xs*::′*a*::*monoid-add list*)
by (*induct xs*) (*auto simp*:*list-add-def*)

lemma *list-add-Cons*[*simp*]: (*x*#*xs*) + (*y*#*ys*) = (*x*+*y*)#(*xs*+*ys*)
by(*auto simp*:*list-add-def*)

lemma *list-diff-Nil*[*simp*]: [] − *xs* = −(*xs*::′*a*::*group-add list*)
by (*induct xs*) (*auto simp*:*list-diff-def list-uminus-def*)

lemma *list-diff-Nil2*[*simp*]: *xs* − [] = (*xs*::′*a*::*group-add list*)
by (*induct xs*) (*auto simp*:*list-diff-def*)

lemma *list-diff-Cons-Cons*[*simp*]: (*x*#*xs*) − (*y*#*ys*) = (*x*−*y*)#(*xs*−*ys*)
by (*induct xs*) (*auto simp*:*list-diff-def*)

lemma *list-uminus-Cons*[*simp*]: −(*x*#*xs*) = (−*x*)#(−*xs*)
by (*induct xs*) (*auto simp*:*list-uminus-def*)

lemma *self-list-diff*:
xs − *xs* = *replicate* (*length*(*xs*::′*a*::*group-add list*)) *0*
by(*induct xs*) *simp-all*

lemma *list-add-assoc*: **fixes** *xs* :: *'a::monoid-add list*
shows $(xs+ys)+zs = xs+(ys+zs)$
apply(*induct xs arbitrary*: *ys zs*)
 apply *simp*
apply(*case-tac ys*)
 apply(*simp*)
apply(*simp*)
apply(*case-tac zs*)
 apply(*simp*)
apply(*simp add*: *add.assoc*)
done

60.2 Inner product

definition *iprod* :: *'a::ring list* ⇒ *'a list* ⇒ *'a* (⟨-,-⟩) **where**
$\langle xs,ys\rangle = (\sum (x,y) \leftarrow zip\ xs\ ys.\ x*y)$

lemma *iprod-Nil*[*simp*]: $\langle [],ys\rangle = 0$
by(*simp add*: *iprod-def*)

lemma *iprod-Nil2*[*simp*]: $\langle xs,[]\rangle = 0$
by(*simp add*: *iprod-def*)

lemma *iprod-Cons*[*simp*]: $\langle x\#xs,y\#ys\rangle = x*y + \langle xs,ys\rangle$
by(*simp add*: *iprod-def*)

lemma *iprod0-if-coeffs0*: $\forall c\in set\ cs.\ c = 0 \Longrightarrow \langle cs,xs\rangle = 0$
apply(*induct cs arbitrary*:*xs*)
 apply *simp*
apply(*case-tac xs*) **apply** *simp*
apply *auto*
done

lemma *iprod-uminus*[*simp*]: $\langle -xs,ys\rangle = -\langle xs,ys\rangle$
by(*simp add*: *iprod-def uminus-listsum-map o-def split-def map-zip-map list-uminus-def*)

lemma *iprod-left-add-distrib*: $\langle xs + ys,zs\rangle = \langle xs,zs\rangle + \langle ys,zs\rangle$
apply(*induct xs arbitrary*: *ys zs*)
apply (*simp add*: *o-def split-def*)
apply(*case-tac ys*)
apply *simp*
apply(*case-tac zs*)
apply (*simp*)
apply(*simp add*: *distrib-right*)
done

lemma *iprod-left-diff-distrib*: $\langle xs - ys,\ zs\rangle = \langle xs,zs\rangle - \langle ys,zs\rangle$
apply(*induct xs arbitrary*: *ys zs*)

apply (*simp add*: *o-def split-def*)
apply(*case-tac ys*)
apply *simp*
apply(*case-tac zs*)
apply (*simp*)
apply(*simp add*: *left-diff-distrib*)
done

lemma *iprod-assoc*: $\langle x *_s xs, ys\rangle = x * \langle xs,ys\rangle$
apply(*induct xs arbitrary*: *ys*)
apply *simp*
apply(*case-tac ys*)
apply (*simp*)
apply (*simp add*: *distrib-left mult.assoc*)
done

end

61 Definitions of Least Upper Bounds and Greatest Lower Bounds

theory *Lub-Glb*
imports *Complex-Main*
begin

Thanks to suggestions by James Margetson

definition *setle* :: $'a$ *set* $\Rightarrow$ $'a::ord \Rightarrow bool$ (**infixl** $*\!<\!=$ *70*)
  where $S *\!<\!= x = (ALL\ y: S.\ y \le x)$

definition *setge* :: $'a::ord \Rightarrow 'a$ *set* $\Rightarrow bool$ (**infixl** $<\!=\!*$ *70*)
  where $x <\!=\!* S = (ALL\ y: S.\ x \le y)$

61.1 Rules for the Relations $*\!<\!=$ and $<\!=\!*$

lemma *setleI*: $ALL\ y: S.\ y \le x \Longrightarrow S *\!<\!= x$
  by (*simp add*: *setle-def*)

lemma *setleD*: $S *\!<\!= x \Longrightarrow y: S \Longrightarrow y \le x$
  by (*simp add*: *setle-def*)

lemma *setgeI*: $ALL\ y: S.\ x \le y \Longrightarrow x <\!=\!* S$
  by (*simp add*: *setge-def*)

lemma *setgeD*: $x <\!=\!* S \Longrightarrow y: S \Longrightarrow x \le y$
  by (*simp add*: *setge-def*)

definition *leastP* :: $('a \Rightarrow bool) \Rightarrow 'a::ord \Rightarrow bool$

where *leastP P x* = (*P x* ∧ *x* <=∗ *Collect P*)

definition *isUb* :: ′*a set* ⇒ ′*a set* ⇒ ′*a*::*ord* ⇒ *bool*
where *isUb R S x* = (*S* ∗<= *x* ∧ *x*: *R*)

definition *isLub* :: ′*a set* ⇒ ′*a set* ⇒ ′*a*::*ord* ⇒ *bool*
where *isLub R S x* = *leastP* (*isUb R S*) *x*

definition *ubs* :: ′*a set* ⇒ ′*a*::*ord set* ⇒ ′*a set*
where *ubs R S* = *Collect* (*isUb R S*)

61.2 Rules about the Operators *leastP*, *ub* and *lub*

lemma *leastPD1*: *leastP P x* ⟹ *P x*
by (*simp add*: *leastP-def*)

lemma *leastPD2*: *leastP P x* ⟹ *x* <=∗ *Collect P*
by (*simp add*: *leastP-def*)

lemma *leastPD3*: *leastP P x* ⟹ *y*: *Collect P* ⟹ *x* ≤ *y*
by (*blast dest*!: *leastPD2 setgeD*)

lemma *isLubD1*: *isLub R S x* ⟹ *S* ∗<= *x*
by (*simp add*: *isLub-def isUb-def leastP-def*)

lemma *isLubD1a*: *isLub R S x* ⟹ *x*: *R*
by (*simp add*: *isLub-def isUb-def leastP-def*)

lemma *isLub-isUb*: *isLub R S x* ⟹ *isUb R S x*
unfolding *isUb-def* **by** (*blast dest*: *isLubD1 isLubD1a*)

lemma *isLubD2*: *isLub R S x* ⟹ *y* : *S* ⟹ *y* ≤ *x*
by (*blast dest*!: *isLubD1 setleD*)

lemma *isLubD3*: *isLub R S x* ⟹ *leastP* (*isUb R S*) *x*
by (*simp add*: *isLub-def*)

lemma *isLubI1*: *leastP*(*isUb R S*) *x* ⟹ *isLub R S x*
by (*simp add*: *isLub-def*)

lemma *isLubI2*: *isUb R S x* ⟹ *x* <=∗ *Collect* (*isUb R S*) ⟹ *isLub R S x*
by (*simp add*: *isLub-def leastP-def*)

lemma *isUbD*: *isUb R S x* ⟹ *y* : *S* ⟹ *y* ≤ *x*
by (*simp add*: *isUb-def setle-def*)

lemma *isUbD2*: *isUb R S x* ⟹ *S* ∗<= *x*
by (*simp add*: *isUb-def*)

lemma *isUbD2a*: *isUb R S x* $\Longrightarrow$ *x*: *R*
by (*simp add*: *isUb-def*)

lemma *isUbI*: *S* $*$<= *x* $\Longrightarrow$ *x*: *R* $\Longrightarrow$ *isUb R S x*
by (*simp add*: *isUb-def*)

lemma *isLub-le-isUb*: *isLub R S x* $\Longrightarrow$ *isUb R S y* $\Longrightarrow$ $x \leq y$
unfolding *isLub-def* **by** (*blast intro*!: *leastPD3*)

lemma *isLub-ubs*: *isLub R S x* $\Longrightarrow$ *x* <=$*$ *ubs R S*
unfolding *ubs-def isLub-def* **by** (*rule leastPD2*)

lemma *isLub-unique*: [| *isLub R S x*; *isLub R S y* |] ==> *x* = (*y*::'*a*::*linorder*)
apply (*frule isLub-isUb*)
apply (*frule-tac x* = *y* **in** *isLub-isUb*)
apply (*blast intro*!: *order-antisym dest*!: *isLub-le-isUb*)
done

lemma *isUb-UNIV-I*: ($\bigwedge y.\ y \in S \Longrightarrow y \leq u$) $\Longrightarrow$ *isUb UNIV S u*
by (*simp add*: *isUbI setleI*)

definition *greatestP* :: ('*a* $\Rightarrow$ *bool*) $\Rightarrow$ '*a*::*ord* $\Rightarrow$ *bool*
where *greatestP P x* = (*P x* $\wedge$ *Collect P* $*$<= *x*)

definition *isLb* :: '*a set* $\Rightarrow$ '*a set* $\Rightarrow$ '*a*::*ord* $\Rightarrow$ *bool*
where *isLb R S x* = (*x* <=$*$ *S* $\wedge$ *x*: *R*)

definition *isGlb* :: '*a set* $\Rightarrow$ '*a set* $\Rightarrow$ '*a*::*ord* $\Rightarrow$ *bool*
where *isGlb R S x* = *greatestP* (*isLb R S*) *x*

definition *lbs* :: '*a set* $\Rightarrow$ '*a*::*ord set* $\Rightarrow$ '*a set*
where *lbs R S* = *Collect* (*isLb R S*)

61.3 Rules about the Operators *greatestP*, *isLb* and *isGlb*

lemma *greatestPD1*: *greatestP P x* $\Longrightarrow$ *P x*
by (*simp add*: *greatestP-def*)

lemma *greatestPD2*: *greatestP P x* $\Longrightarrow$ *Collect P* $*$<= *x*
by (*simp add*: *greatestP-def*)

lemma *greatestPD3*: *greatestP P x* $\Longrightarrow$ *y*: *Collect P* $\Longrightarrow$ $x \geq y$
by (*blast dest*!: *greatestPD2 setleD*)

lemma *isGlbD1*: *isGlb R S x* $\Longrightarrow$ *x* <=$*$ *S*
by (*simp add*: *isGlb-def isLb-def greatestP-def*)

lemma *isGlbD1a*: *isGlb R S x* $\Longrightarrow$ *x*: *R*

by (*simp add*: *isGlb-def isLb-def greatestP-def*)

lemma *isGlb-isLb*: *isGlb R S x* $\Longrightarrow$ *isLb R S x*
unfolding *isLb-def* **by** (*blast dest*: *isGlbD1 isGlbD1a*)

lemma *isGlbD2*: *isGlb R S x* $\Longrightarrow$ *y* : *S* $\Longrightarrow$ *y* $\geq$ *x*
by (*blast dest*!: *isGlbD1 setgeD*)

lemma *isGlbD3*: *isGlb R S x* $\Longrightarrow$ *greatestP* (*isLb R S*) *x*
by (*simp add*: *isGlb-def*)

lemma *isGlbI1*: *greatestP* (*isLb R S*) *x* $\Longrightarrow$ *isGlb R S x*
by (*simp add*: *isGlb-def*)

lemma *isGlbI2*: *isLb R S x* $\Longrightarrow$ *Collect* (*isLb R S*) *<= *x* $\Longrightarrow$ *isGlb R S x*
by (*simp add*: *isGlb-def greatestP-def*)

lemma *isLbD*: *isLb R S x* $\Longrightarrow$ *y* : *S* $\Longrightarrow$ *y* $\geq$ *x*
by (*simp add*: *isLb-def setge-def*)

lemma *isLbD2*: *isLb R S x* $\Longrightarrow$ *x* <=* *S*
by (*simp add*: *isLb-def*)

lemma *isLbD2a*: *isLb R S x* $\Longrightarrow$ *x*: *R*
by (*simp add*: *isLb-def*)

lemma *isLbI*: *x* <=* *S* $\Longrightarrow$ *x*: *R* $\Longrightarrow$ *isLb R S x*
by (*simp add*: *isLb-def*)

lemma *isGlb-le-isLb*: *isGlb R S x* $\Longrightarrow$ *isLb R S y* $\Longrightarrow$ *x* $\geq$ *y*
unfolding *isGlb-def* **by** (*blast intro*!: *greatestPD3*)

lemma *isGlb-ubs*: *isGlb R S x* $\Longrightarrow$ *lbs R S* *<= *x*
unfolding *lbs-def isGlb-def* **by** (*rule greatestPD2*)

lemma *isGlb-unique*: [| *isGlb R S x*; *isGlb R S y* |] ==> *x* = (*y*::'*a*::*linorder*)
apply (*frule isGlb-isLb*)
apply (*frule-tac x* = *y* **in** *isGlb-isLb*)
apply (*blast intro*!: *order-antisym dest*!: *isGlb-le-isLb*)
done

lemma *bdd-above-setle*: *bdd-above A* $\longleftrightarrow$ ($\exists$ *a*. *A* *<= *a*)
by (*auto simp*: *bdd-above-def setle-def*)

lemma *bdd-below-setge*: *bdd-below A* $\longleftrightarrow$ ($\exists$ *a*. *a* <=* *A*)
by (*auto simp*: *bdd-below-def setge-def*)

lemma *isLub-cSup*:
(*S*::'*a* :: *conditionally-complete-lattice set*) $\neq$ {} $\Longrightarrow$ ($\exists$ *b*. *S* *<= *b*) $\Longrightarrow$ *isLub*

UNIV S (Sup S)
by (*auto simp add*: *isLub-def setle-def leastP-def isUb-def intro*!: *setgeI cSup-upper cSup-least*)

lemma *isGlb-cInf*:
$(S::'a :: conditionally\text{-}complete\text{-}lattice\ set) \neq \{\} \Longrightarrow (\exists\, b.\ b <=*\ S) \Longrightarrow isGlb$ *UNIV S (Inf S)*
by (*auto simp add*: *isGlb-def setge-def greatestP-def isLb-def intro*!: *setleI cInf-lower cInf-greatest*)

lemma *cSup-le*: $(S::'a::conditionally\text{-}complete\text{-}lattice\ set) \neq \{\} \Longrightarrow S *<= b \Longrightarrow Sup\ S \leq b$
by (*metis cSup-least setle-def*)

lemma *cInf-ge*: $(S::'a :: conditionally\text{-}complete\text{-}lattice\ set) \neq \{\} \Longrightarrow b <=* S \Longrightarrow Inf\ S \geq b$
by (*metis cInf-greatest setge-def*)

lemma *cSup-bounds*:
fixes $S :: 'a :: conditionally\text{-}complete\text{-}lattice\ set$
shows $S \neq \{\} \Longrightarrow a <=* S \Longrightarrow S *<= b \Longrightarrow a \leq Sup\ S \wedge Sup\ S \leq b$
using *cSup-least*[*of S b*] *cSup-upper2*[*of - S a*]
by (*auto simp*: *bdd-above-setle setge-def setle-def*)

lemma *cSup-unique*: $(S::'a :: \{conditionally\text{-}complete\text{-}linorder, no\text{-}bot\}\ set) *<= b \Longrightarrow (\forall\, b'<b.\ \exists\, x\in S.\ b' < x) \Longrightarrow Sup\ S = b$
by (*rule cSup-eq*) (*auto simp*: *not-le*[*symmetric*] *setle-def*)

lemma *cInf-unique*: $b <=* (S::'a :: \{conditionally\text{-}complete\text{-}linorder, no\text{-}top\}\ set) \Longrightarrow (\forall\, b'>b.\ \exists\, x\in S.\ b' > x) \Longrightarrow Inf\ S = b$
by (*rule cInf-eq*) (*auto simp*: *not-le*[*symmetric*] *setge-def*)

Use completeness of reals (supremum property) to show that any bounded sequence has a least upper bound

lemma *reals-complete*: $\exists\, X.\ X \in S \Longrightarrow \exists\, Y.\ isUb\ (UNIV::real\ set)\ S\ Y \Longrightarrow \exists\, t.\ isLub\ (UNIV :: real\ set)\ S\ t$
by (*intro exI*[*of - Sup S*] *isLub-cSup*) (*auto simp*: *setle-def isUb-def intro*!: *cSup-upper*)

lemma *Bseq-isUb*: $\bigwedge X :: nat \Rightarrow real.\ Bseq\ X \Longrightarrow \exists\, U.\ isUb\ (UNIV::real\ set)\ \{x.\ \exists\, n.\ X\ n = x\}\ U$
by (*auto intro*: *isUbI setleI simp add*: *Bseq-def abs-le-iff*)

lemma *Bseq-isLub*: $\bigwedge X :: nat \Rightarrow real.\ Bseq\ X \Longrightarrow \exists\, U.\ isLub\ (UNIV::real\ set)\ \{x.\ \exists\, n.\ X\ n = x\}\ U$
by (*blast intro*: *reals-complete Bseq-isUb*)

lemma *isLub-mono-imp-LIMSEQ*:
fixes $X :: nat \Rightarrow real$

assumes *u*: *isLub UNIV* $\{x. \exists n. X\ n = x\}$ *u*
assumes *X*: $\forall m\ n.\ m \leq n \longrightarrow X\ m \leq X\ n$
shows $X \longrightarrow u$
proof −
have $X \longrightarrow (SUP\ i.\ X\ i)$
using *u*[*THEN isLubD1*] *X*
by (*intro LIMSEQ-incseq-SUP*) (*auto simp*: *incseq-def image-def eq-commute bdd-above-setle*)
also have $(SUP\ i.\ X\ i) = u$
using *isLub-cSup*[*of range X*] *u*[*THEN isLubD1*]
by (*intro isLub-unique*[*OF - u*]) (*auto simp add*: *image-def eq-commute*)
finally show *?thesis* **.**
qed

lemmas *real-isGlb-unique* = *isGlb-unique*[**where** *'a=real*]

lemma *real-le-inf-subset*: $t \neq \{\} \Longrightarrow t \subseteq s \Longrightarrow \exists b.\ b <=* s \Longrightarrow Inf\ s \leq Inf$ (*t*::*real set*)
by (*rule cInf-superset-mono*) (*auto simp*: *bdd-below-setge*)

lemma *real-ge-sup-subset*: $t \neq \{\} \Longrightarrow t \subseteq s \Longrightarrow \exists b.\ s *<= b \Longrightarrow Sup\ s \geq Sup$ (*t*::*real set*)
by (*rule cSup-subset-mono*) (*auto simp*: *bdd-above-setle*)

end

62 An abstract view on maps for code generation.

theory *Mapping*
imports *Main*
begin

62.1 Parametricity transfer rules

lemma *map-of-foldr*: — FIXME move
map-of xs = *foldr* ($\lambda(k, v)\ m.\ m(k \mapsto v)$) *xs Map.empty*
using *map-add-map-of-foldr* [*of Map.empty*] **by** *auto*

context
begin

interpretation *lifting-syntax* **.**

lemma *empty-parametric*:
(*A* ===> *rel-option B*) *Map.empty Map.empty*
by *transfer-prover*

lemma *lookup-parametric*: ((*A* ===> *B*) ===> *A* ===> *B*) ($\lambda m\ k.\ m\ k$) ($\lambda m\ k.\ m\ k$)

by *transfer-prover*

lemma *update-parametric*:
assumes [*transfer-rule*]: *bi-unique A*
shows (*A* ===> *B* ===> (*A* ===> *rel-option B*) ===> *A* ===> *rel-option B*)
(λ*k v m*. *m*(*k* ↦ *v*)) (λ*k v m*. *m*(*k* ↦ *v*))
by *transfer-prover*

lemma *delete-parametric*:
assumes [*transfer-rule*]: *bi-unique A*
shows (*A* ===> (*A* ===> *rel-option B*) ===> *A* ===> *rel-option B*)
(λ*k m*. *m*(*k* := *None*)) (λ*k m*. *m*(*k* := *None*))
by *transfer-prover*

lemma *is-none-parametric* [*transfer-rule*]:
(*rel-option A* ===> *HOL.eq*) *Option.is-none Option.is-none*
by (*auto simp add*: *Option.is-none-def rel-fun-def rel-option-iff split*: *option.split*)

lemma *dom-parametric*:
assumes [*transfer-rule*]: *bi-total A*
shows ((*A* ===> *rel-option B*) ===> *rel-set A*) *dom dom*
unfolding *dom-def* [*abs-def*] *Option.is-none-def* [*symmetric*] **by** *transfer-prover*

lemma *map-of-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique R1*
shows (*list-all2* (*rel-prod R1 R2*) ===> *R1* ===> *rel-option R2*) *map-of map-of*
unfolding *map-of-def* **by** *transfer-prover*

lemma *map-entry-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*A* ===> (*B* ===> *B*) ===> (*A* ===> *rel-option B*) ===> *A* ===> *rel-option B*)
(λ*k f m*. (*case m k of None* ⇒ *m*
| *Some v* ⇒ *m* (*k* ↦ (*f v*)))) (λ*k f m*. (*case m k of None* ⇒ *m*
| *Some v* ⇒ *m* (*k* ↦ (*f v*))))
by *transfer-prover*

lemma *tabulate-parametric*:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> (*A* ===> *B*) ===> *A* ===> *rel-option B*)
(λ*ks f*. (*map-of* (*map* (λ*k*. (*k*, *f k*)) *ks*))) (λ*ks f*. (*map-of* (*map* (λ*k*. (*k*, *f k*)) *ks*)))
by *transfer-prover*

lemma *bulkload-parametric*:
(*list-all2 A* ===> *HOL.eq* ===> *rel-option A*)
(λ*xs k*. *if k* < *length xs then Some* (*xs* ! *k*) *else None*) (λ*xs k*. *if k* < *length xs*

then Some (*xs* ! *k*) *else None*)
proof
  fix *xs ys*
  assume *list-all2 A xs ys*
  then show (*HOL.eq* ===> *rel-option A*)
    (λ*k*. *if k* < *length xs then Some* (*xs* ! *k*) *else None*)
    (λ*k*. *if k* < *length ys then Some* (*ys* ! *k*) *else None*)
    apply *induct*
    apply *auto*
    unfolding *rel-fun-def*
    apply *clarsimp*
    apply (*case-tac xa*)
    apply (*auto dest*: *list-all2-lengthD list-all2-nthD*)
    done
qed

lemma *map-parametric*:
  ((*A* ===> *B*) ===> (*C* ===> *D*) ===> (*B* ===> *rel-option C*) ===> *A* ===> *rel-option D*)
    (λ*f g m*. (*map-option g* ∘ *m* ∘ *f*)) (λ*f g m*. (*map-option g* ∘ *m* ∘ *f*))
  by *transfer-prover*

end

62.2 Type definition and primitive operations

typedef (′*a*, ′*b*) *mapping* = *UNIV* :: (′*a* ⇀ ′*b*) *set*
  morphisms *rep Mapping*
  ..

setup-lifting *type-definition-mapping*

lift-definition *empty* :: (′*a*, ′*b*) *mapping*
  is *Map.empty* **parametric** *empty-parametric* **.**

lift-definition *lookup* :: (′*a*, ′*b*) *mapping* ⇒ ′*a* ⇒ ′*b option*
  is λ*m k*. *m k* **parametric** *lookup-parametric* **.**

declare [[*code drop*: *Mapping.lookup*]]
setup ‹*Code.add-default-eqn* @{*thm Mapping.lookup.abs-eq*}› — FIXME lifting

lift-definition *update* :: ′*a* ⇒ ′*b* ⇒ (′*a*, ′*b*) *mapping* ⇒ (′*a*, ′*b*) *mapping*
  is λ*k v m*. *m*(*k* ↦ *v*) **parametric** *update-parametric* **.**

lift-definition *delete* :: ′*a* ⇒ (′*a*, ′*b*) *mapping* ⇒ (′*a*, ′*b*) *mapping*
  is λ*k m*. *m*(*k* := *None*) **parametric** *delete-parametric* **.**

lift-definition *keys* :: (′*a*, ′*b*) *mapping* ⇒ ′*a set*
  is *dom* **parametric** *dom-parametric* **.**

lift-definition *tabulate* :: *'a list* ⇒ (*'a* ⇒ *'b*) ⇒ (*'a*, *'b*) *mapping*
is λ*ks f*. (*map-of* (*List.map* (λ*k*. (*k*, *f k*)) *ks*)) **parametric** *tabulate-parametric*
.

lift-definition *bulkload* :: *'a list* ⇒ (*nat*, *'a*) *mapping*
is λ*xs k*. *if k* < *length xs then Some* (*xs* ! *k*) *else None* **parametric** *bulkload-parametric*
.

lift-definition *map* :: (*'c* ⇒ *'a*) ⇒ (*'b* ⇒ *'d*) ⇒ (*'a*, *'b*) *mapping* ⇒ (*'c*, *'d*) *mapping*
is λ*f g m*. (*map-option g* ∘ *m* ∘ *f*) **parametric** *map-parametric* **.**

declare [[*code drop*: *map*]]

62.3 Functorial structure

functor *map*: *map*
by (*transfer*, *auto simp add*: *fun-eq-iff option.map-comp option.map-id*)+

62.4 Derived operations

definition *ordered-keys* :: (*'a*::*linorder*, *'b*) *mapping* ⇒ *'a list*
where
ordered-keys m = (*if finite* (*keys m*) *then sorted-list-of-set* (*keys m*) *else* [])

definition *is-empty* :: (*'a*, *'b*) *mapping* ⇒ *bool*
where
is-empty m ⟷ *keys m* = {}

definition *size* :: (*'a*, *'b*) *mapping* ⇒ *nat*
where
size m = (*if finite* (*keys m*) *then card* (*keys m*) *else 0*)

definition *replace* :: *'a* ⇒ *'b* ⇒ (*'a*, *'b*) *mapping* ⇒ (*'a*, *'b*) *mapping*
where
replace k v m = (*if k* ∈ *keys m then update k v m else m*)

definition *default* :: *'a* ⇒ *'b* ⇒ (*'a*, *'b*) *mapping* ⇒ (*'a*, *'b*) *mapping*
where
default k v m = (*if k* ∈ *keys m then m else update k v m*)

Manual derivation of transfer rule is non-trivial

lift-definition *map-entry* :: *'a* ⇒ (*'b* ⇒ *'b*) ⇒ (*'a*, *'b*) *mapping* ⇒ (*'a*, *'b*) *mapping*
is
λ*k f m*. (*case m k of None* ⇒ *m*
| *Some v* ⇒ *m* (*k* ↦ (*f v*))) **parametric** *map-entry-parametric* **.**

lemma *map-entry-code* [*code*]:
map-entry k f m = (*case lookup m k of None* ⇒ *m*

| *Some v* ⇒ *update k* (*f v*) *m*)
by *transfer rule*

definition *map-default* :: *'a* ⇒ *'b* ⇒ (*'b* ⇒ *'b*) ⇒ (*'a*, *'b*) *mapping* ⇒ (*'a*, *'b*) *mapping*
where
map-default k v f m = *map-entry k f* (*default k v m*)

definition *of-alist* :: (*'k* × *'v*) *list* ⇒ (*'k*, *'v*) *mapping*
where
of-alist xs = *foldr* (λ(*k*, *v*) *m*. *update k v m*) *xs empty*

instantiation *mapping* :: (*type*, *type*) *equal*
begin

definition
HOL.equal m1 m2 ⟷ (∀ *k*. *lookup m1 k* = *lookup m2 k*)

instance
by *standard* (*unfold equal-mapping-def*, *transfer*, *auto*)

end

context
begin

interpretation *lifting-syntax* **.**

lemma [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-total A*
assumes [*transfer-rule*]: *bi-unique B*
shows (*pcr-mapping A B* ===> *pcr-mapping A B* ===> *op*=) *HOL.eq HOL.equal*
by (*unfold equal*) *transfer-prover*

lemma *of-alist-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique R1*
shows (*list-all2* (*rel-prod R1 R2*) ===> *pcr-mapping R1 R2*) *map-of of-alist*
unfolding *of-alist-def* [*abs-def*] *map-of-foldr* [*abs-def*] **by** *transfer-prover*

end

62.5 Properties

lemma *lookup-update*:
lookup (*update k v m*) *k* = *Some v*
by *transfer simp*

lemma *lookup-update-neq*:
k ≠ *k'* ⟹ *lookup* (*update k v m*) *k'* = *lookup m k'*

by *transfer simp*

lemma *lookup-empty*:
lookup empty k = None
by *transfer simp*

lemma *keys-is-none-rep* [*code-unfold*]:
$k \in keys\ m \longleftrightarrow \neg\ (Option.is\text{-}none\ (lookup\ m\ k))$
by *transfer* (*auto simp add*: *Option.is-none-def*)

lemma *update-update*:
update k v (*update k w m*) = *update k v m*
$k \neq l \Longrightarrow$ *update k v* (*update l w m*) = *update l w* (*update k v m*)
by (*transfer*, *simp add*: *fun-upd-twist*)+

lemma *update-delete* [*simp*]:
update k v (*delete k m*) = *update k v m*
by *transfer simp*

lemma *delete-update*:
delete k (*update k v m*) = *delete k m*
$k \neq l \Longrightarrow$ *delete k* (*update l v m*) = *update l v* (*delete k m*)
by (*transfer*, *simp add*: *fun-upd-twist*)+

lemma *delete-empty* [*simp*]:
delete k empty = *empty*
by *transfer simp*

lemma *replace-update*:
$k \notin keys\ m \Longrightarrow replace\ k\ v\ m = m$
$k \in keys\ m \Longrightarrow replace\ k\ v\ m = update\ k\ v\ m$
by (*transfer*, *auto simp add*: *replace-def fun-upd-twist*)+

lemma *size-empty* [*simp*]:
size empty = *0*
unfolding *size-def* **by** *transfer simp*

lemma *size-update*:
finite (*keys m*) $\Longrightarrow$ *size* (*update k v m*) =
(*if* $k \in$ *keys m then size m else Suc* (*size m*))
unfolding *size-def* **by** *transfer* (*auto simp add*: *insert-dom*)

lemma *size-delete*:
size (*delete k m*) = (*if* $k \in$ *keys m then size m* − *1 else size m*)
unfolding *size-def* **by** *transfer simp*

lemma *size-tabulate* [*simp*]:
size (*tabulate ks f*) = *length* (*remdups ks*)
unfolding *size-def* **by** *transfer* (*auto simp add*: *map-of-map-restrict card-set*

comp-def)

lemma *bulkload-tabulate*:
bulkload xs = *tabulate* [*0*..<*length xs*] (*nth xs*)
by *transfer* (*auto simp add*: *map-of-map-restrict*)

lemma *is-empty-empty* [*simp*]:
is-empty empty
unfolding *is-empty-def* **by** *transfer simp*

lemma *is-empty-update* [*simp*]:
¬ *is-empty* (*update k v m*)
unfolding *is-empty-def* **by** *transfer simp*

lemma *is-empty-delete*:
is-empty (*delete k m*) ⟷ *is-empty m* ∨ *keys m* = {*k*}
unfolding *is-empty-def* **by** *transfer* (*auto simp del*: *dom-eq-empty-conv*)

lemma *is-empty-replace* [*simp*]:
is-empty (*replace k v m*) ⟷ *is-empty m*
unfolding *is-empty-def replace-def* **by** *transfer auto*

lemma *is-empty-default* [*simp*]:
¬ *is-empty* (*default k v m*)
unfolding *is-empty-def default-def* **by** *transfer auto*

lemma *is-empty-map-entry* [*simp*]:
is-empty (*map-entry k f m*) ⟷ *is-empty m*
unfolding *is-empty-def* **by** *transfer* (*auto split*: *option.split*)

lemma *is-empty-map-default* [*simp*]:
¬ *is-empty* (*map-default k v f m*)
by (*simp add*: *map-default-def*)

lemma *keys-dom-lookup*:
keys m = *dom* (*Mapping.lookup m*)
by *transfer rule*

lemma *keys-empty* [*simp*]:
keys empty = {}
by *transfer simp*

lemma *keys-update* [*simp*]:
keys (*update k v m*) = *insert k* (*keys m*)
by *transfer simp*

lemma *keys-delete* [*simp*]:
keys (*delete k m*) = *keys m* − {*k*}
by *transfer simp*

lemma *keys-replace* [*simp*]:
keys (*replace k v m*) = *keys m*
unfolding *replace-def* **by** *transfer* (*simp add*: *insert-absorb*)

lemma *keys-default* [*simp*]:
keys (*default k v m*) = *insert k* (*keys m*)
unfolding *default-def* **by** *transfer* (*simp add*: *insert-absorb*)

lemma *keys-map-entry* [*simp*]:
keys (*map-entry k f m*) = *keys m*
by *transfer* (*auto split*: *option.split*)

lemma *keys-map-default* [*simp*]:
keys (*map-default k v f m*) = *insert k* (*keys m*)
by (*simp add*: *map-default-def*)

lemma *keys-tabulate* [*simp*]:
keys (*tabulate ks f*) = *set ks*
by *transfer* (*simp add*: *map-of-map-restrict o-def*)

lemma *keys-bulkload* [*simp*]:
keys (*bulkload xs*) = {*0..<length xs*}
by (*simp add*: *bulkload-tabulate*)

lemma *distinct-ordered-keys* [*simp*]:
distinct (*ordered-keys m*)
by (*simp add*: *ordered-keys-def*)

lemma *ordered-keys-infinite* [*simp*]:
¬ *finite* (*keys m*) ⟹ *ordered-keys m* = []
by (*simp add*: *ordered-keys-def*)

lemma *ordered-keys-empty* [*simp*]:
ordered-keys empty = []
by (*simp add*: *ordered-keys-def*)

lemma *ordered-keys-update* [*simp*]:
k ∈ *keys m* ⟹ *ordered-keys* (*update k v m*) = *ordered-keys m*
finite (*keys m*) ⟹ *k* ∉ *keys m* ⟹ *ordered-keys* (*update k v m*) = *insort k* (*ordered-keys m*)
by (*simp-all add*: *ordered-keys-def*) (*auto simp only*: *sorted-list-of-set-insert* [*symmetric*] *insert-absorb*)

lemma *ordered-keys-delete* [*simp*]:
ordered-keys (*delete k m*) = *remove1 k* (*ordered-keys m*)
proof (*cases finite* (*keys m*))
case *False* **then show** *?thesis* **by** *simp*
next

case *True* **note** *fin* = *True*
show *?thesis*
proof (*cases* $k \in$ *keys m*)
case *False* **with** *fin* **have** $k \notin$ *set* (*sorted-list-of-set* (*keys m*)) **by** *simp*
with *False* **show** *?thesis* **by** (*simp add*: *ordered-keys-def remove1-idem*)
next
case *True* **with** *fin* **show** *?thesis* **by** (*simp add*: *ordered-keys-def sorted-list-of-set-remove*)
qed
qed

lemma *ordered-keys-replace* [*simp*]:
ordered-keys (*replace k v m*) = *ordered-keys m*
by (*simp add*: *replace-def*)

lemma *ordered-keys-default* [*simp*]:
$k \in$ *keys m* $\Longrightarrow$ *ordered-keys* (*default k v m*) = *ordered-keys m*
finite (*keys m*) $\Longrightarrow$ $k \notin$ *keys m* $\Longrightarrow$ *ordered-keys* (*default k v m*) = *insort k* (*ordered-keys m*)
by (*simp-all add*: *default-def*)

lemma *ordered-keys-map-entry* [*simp*]:
ordered-keys (*map-entry k f m*) = *ordered-keys m*
by (*simp add*: *ordered-keys-def*)

lemma *ordered-keys-map-default* [*simp*]:
$k \in$ *keys m* $\Longrightarrow$ *ordered-keys* (*map-default k v f m*) = *ordered-keys m*
finite (*keys m*) $\Longrightarrow$ $k \notin$ *keys m* $\Longrightarrow$ *ordered-keys* (*map-default k v f m*) = *insort k* (*ordered-keys m*)
by (*simp-all add*: *map-default-def*)

lemma *ordered-keys-tabulate* [*simp*]:
ordered-keys (*tabulate ks f*) = *sort* (*remdups ks*)
by (*simp add*: *ordered-keys-def sorted-list-of-set-sort-remdups*)

lemma *ordered-keys-bulkload* [*simp*]:
ordered-keys (*bulkload ks*) = [*0..<length ks*]
by (*simp add*: *ordered-keys-def*)

lemma *tabulate-fold*:
tabulate xs f = *fold* (λk *m*. *update k* (*f k*) *m*) *xs empty*
proof *transfer*
fix $f :: 'a \Rightarrow 'b$ **and** *xs*
have *map-of* (*List.map* (λk. (*k*, *f k*)) *xs*) = *foldr* (λk *m*. *m*($k \mapsto f\ k$)) *xs* *Map.empty*
by (*simp add*: *foldr-map comp-def map-of-foldr*)
also have *foldr* (λk *m*. *m*($k \mapsto f\ k$)) *xs* = *fold* (λk *m*. *m*($k \mapsto f\ k$)) *xs*
by (*rule foldr-fold*) (*simp add*: *fun-eq-iff*)
ultimately show *map-of* (*List.map* (λk. (*k*, *f k*)) *xs*) = *fold* (λk *m*. *m*($k \mapsto f$ *k*)) *xs Map.empty*

by *simp*
qed

62.6 Code generator setup

hide-const (**open**) *empty is-empty rep lookup update delete ordered-keys keys size replace default map-entry map-default tabulate bulkload map of-alist*

end

63 Adhoc overloading of constants based on their types

theory *Adhoc-Overloading*
imports *Pure*
keywords *adhoc-overloading* :: *thy-decl* **and** *no-adhoc-overloading* :: *thy-decl*
begin

ML-file *adhoc-overloading.ML*

end

64 Monad notation for arbitrary types

theory *Monad-Syntax*
imports *Main* ~~/*src*/*Tools*/*Adhoc-Overloading*
begin

We provide a convenient do-notation for monadic expressions well-known from Haskell. *Let* is printed specially in do-expressions.

consts
bind :: $['a, 'b \Rightarrow 'c] \Rightarrow 'd$ (**infixr** ≫= *54*)

notation (*ASCII*)
bind (**infixr** >>= *54*)

abbreviation (*do-notation*)
bind-do :: $['a, 'b \Rightarrow 'c] \Rightarrow 'd$
where *bind-do* ≡ *bind*

notation (**output**)
bind-do (**infixr** ≫= *54*)

notation (*ASCII* **output**)
bind-do (**infixr** >>= *54*)

nonterminal *do-binds* **and** *do-bind*
syntax
  -do-block :: *do-binds* ⇒ *'a* (*do* {//(*2* -)//} [*12*] *62*)
  -do-bind :: [*pttrn*, *'a*] ⇒ *do-bind* ((*2*- ←/ -) *13*)
  -do-let :: [*pttrn*, *'a*] ⇒ *do-bind* ((*2let* - =/ -) [*1000*, *13*] *13*)
  -do-then :: *'a* ⇒ *do-bind* (- [*14*] *13*)
  -do-final :: *'a* ⇒ *do-binds* (-)
  -do-cons :: [*do-bind*, *do-binds*] ⇒ *do-binds* (-;//- [*13*, *12*] *12*)
  -thenM :: [*'a*, *'b*] ⇒ *'c* (**infixr** ≫ *54*)

syntax (*ASCII*)
  -do-bind :: [*pttrn*, *'a*] ⇒ *do-bind* ((*2*- <−/ -) *13*)
  -thenM :: [*'a*, *'b*] ⇒ *'c* (**infixr** >> *54*)

translations
  -do-block (*-do-cons* (*-do-then* *t*) (*-do-final* *e*))
    ⇌ *CONST bind-do t* (λ-. *e*)
  -do-block (*-do-cons* (*-do-bind* *p t*) (*-do-final* *e*))
    ⇌ *CONST bind-do t* (λ*p*. *e*)
  -do-block (*-do-cons* (*-do-let* *p t*) *bs*)
    ⇌ *let p* = *t in -do-block bs*
  -do-block (*-do-cons* *b* (*-do-cons* *c cs*))
    ⇌ *-do-block* (*-do-cons* *b* (*-do-final* (*-do-block* (*-do-cons* *c cs*))))
  -do-cons (*-do-let* *p t*) (*-do-final* *s*)
    ⇌ *-do-final* (*let p* = *t in s*)
  -do-block (*-do-final* *e*) ⇀ *e*
  (*m* ≫ *n*) ⇀ (*m* ≫= (λ-. *n*))

adhoc-overloading
  bind Set.bind Predicate.bind Option.bind List.bind

end

65 (Finite) multisets

theory *Multiset*
imports *Main*
begin

65.1 The type of multisets

definition *multiset* = {*f* :: *'a* ⇒ *nat*. *finite* {*x*. *f x* > *0*}}

typedef *'a multiset* = *multiset* :: (*'a* ⇒ *nat*) *set*
  morphisms *count Abs-multiset*
  unfolding *multiset-def*
proof
  show (λ*x*. *0*::*nat*) ∈ {*f*. *finite* {*x*. *f x* > *0*}} **by** *simp*
qed

setup-lifting *type-definition-multiset*

lemma *multiset-eq-iff*: $M = N \longleftrightarrow (\forall a.\ count\ M\ a = count\ N\ a)$
by (*simp only*: *count-inject* [*symmetric*] *fun-eq-iff*)

lemma *multiset-eqI*: $(\bigwedge x.\ count\ A\ x = count\ B\ x) \Longrightarrow A = B$
using *multiset-eq-iff* **by** *auto*

Preservation of the representing set *multiset*.

lemma *const0-in-multiset*: $(\lambda a.\ 0) \in multiset$
by (*simp add*: *multiset-def*)

lemma *only1-in-multiset*: $(\lambda b.\ if\ b = a\ then\ n\ else\ 0) \in multiset$
by (*simp add*: *multiset-def*)

lemma *union-preserves-multiset*: $M \in multiset \Longrightarrow N \in multiset \Longrightarrow (\lambda a.\ M\ a + N\ a) \in multiset$
by (*simp add*: *multiset-def*)

lemma *diff-preserves-multiset*:
assumes $M \in multiset$
shows $(\lambda a.\ M\ a - N\ a) \in multiset$
proof −
have $\{x.\ N\ x < M\ x\} \subseteq \{x.\ 0 < M\ x\}$
by *auto*
with *assms* **show** *?thesis*
by (*auto simp add*: *multiset-def intro*: *finite-subset*)
qed

lemma *filter-preserves-multiset*:
assumes $M \in multiset$
shows $(\lambda x.\ if\ P\ x\ then\ M\ x\ else\ 0) \in multiset$
proof −
have $\{x.\ (P\ x \longrightarrow 0 < M\ x) \wedge P\ x\} \subseteq \{x.\ 0 < M\ x\}$
by *auto*
with *assms* **show** *?thesis*
by (*auto simp add*: *multiset-def intro*: *finite-subset*)
qed

lemmas *in-multiset* = *const0-in-multiset only1-in-multiset*
union-preserves-multiset diff-preserves-multiset filter-preserves-multiset

65.2 Representing multisets

Multiset enumeration

instantiation *multiset* :: (*type*) *cancel-comm-monoid-add*
begin

lift-definition *zero-multiset* :: *'a multiset* **is** $\lambda a.\ 0$
by (*rule const0-in-multiset*)

abbreviation *Mempty* :: *'a multiset* ({#}) **where**
Mempty ≡ *0*

lift-definition *plus-multiset* :: *'a multiset* ⇒ *'a multiset* ⇒ *'a multiset* **is** $\lambda M\ N.$ $(\lambda a.\ M\ a + N\ a)$
by (*rule union-preserves-multiset*)

lift-definition *minus-multiset* :: *'a multiset* ⇒ *'a multiset* ⇒ *'a multiset* **is** $\lambda\ M$ $N.\ \lambda a.\ M\ a - N\ a$
by (*rule diff-preserves-multiset*)

instance
by (*standard*; *transfer*; *simp add*: *fun-eq-iff*)

end

lift-definition *single* :: *'a* ⇒ *'a multiset* **is** $\lambda a\ b.$ *if b = a then 1 else 0*
by (*rule only1-in-multiset*)

syntax
-multiset :: *args* ⇒ *'a multiset* ({#(-)#})
translations
{#*x*, *xs*#} == {#*x*#} + {#*xs*#}
{#*x*#} == *CONST single x*

lemma *count-empty* [*simp*]: *count* {#} *a* = *0*
by (*simp add*: *zero-multiset.rep-eq*)

lemma *count-single* [*simp*]: *count* {#*b*#} *a* = (*if b = a then 1 else 0*)
by (*simp add*: *single.rep-eq*)

65.3 Basic operations

65.3.1 Conversion to set and membership

definition *set-mset* :: *'a multiset* ⇒ *'a set*
where *set-mset M* = {*x*. *count M x* > *0*}

abbreviation *Melem* :: *'a* ⇒ *'a multiset* ⇒ *bool*
where *Melem a M* ≡ *a* ∈ *set-mset M*

notation
Melem (*op* ∈#) **and**
Melem ((-/ ∈# -) [*51*, *51*] *50*)

notation (*ASCII*)
Melem (*op* :#) **and**

Melem ((-/ :# -) [*51*, *51*] *50*)

abbreviation *not-Melem* :: ′*a* ⇒ ′*a multiset* ⇒ *bool*
where *not-Melem a M* ≡ *a* ∉ *set-mset M*

notation
not-Melem (*op* ∉#) **and**
not-Melem ((-/ ∉# -) [*51*, *51*] *50*)

notation (*ASCII*)
not-Melem (*op* ~:#) **and**
not-Melem ((-/ ~:# -) [*51*, *51*] *50*)

context
begin

qualified abbreviation *Ball* :: ′*a multiset* ⇒ (′*a* ⇒ *bool*) ⇒ *bool*
where *Ball M* ≡ *Set.Ball* (*set-mset M*)

qualified abbreviation *Bex* :: ′*a multiset* ⇒ (′*a* ⇒ *bool*) ⇒ *bool*
where *Bex M* ≡ *Set.Bex* (*set-mset M*)

end

syntax
-MBall :: *pttrn* ⇒ ′*a set* ⇒ *bool* ⇒ *bool* ((*3*∀ -∈#-./ -) [*0*, *0*, *10*] *10*)
-MBex :: *pttrn* ⇒ ′*a set* ⇒ *bool* ⇒ *bool* ((*3*∃ -∈#-./ -) [*0*, *0*, *10*] *10*)

syntax (*ASCII*)
-MBall :: *pttrn* ⇒ ′*a set* ⇒ *bool* ⇒ *bool* ((*3*∀ -:#-./ -) [*0*, *0*, *10*] *10*)
-MBex :: *pttrn* ⇒ ′*a set* ⇒ *bool* ⇒ *bool* ((*3*∃ -:#-./ -) [*0*, *0*, *10*] *10*)

translations
∀ *x*∈#*A*. *P* ⇌ *CONST Multiset.Ball A* (λ*x*. *P*)
∃ *x*∈#*A*. *P* ⇌ *CONST Multiset.Bex A* (λ*x*. *P*)

lemma *count-eq-zero-iff*:
count M x = *0* ⟷ *x* ∉# *M*
by (*auto simp add*: *set-mset-def*)

lemma *not-in-iff*:
x ∉# *M* ⟷ *count M x* = *0*
by (*auto simp add*: *count-eq-zero-iff*)

lemma *count-greater-zero-iff* [*simp*]:
count M x > *0* ⟷ *x* ∈# *M*
by (*auto simp add*: *set-mset-def*)

lemma *count-inI*:

assumes *count M x* = *0* ⟹ *False*
shows *x* ∈# *M*
proof (*rule ccontr*)
assume *x* ∉# *M*
with *assms* **show** *False* **by** (*simp add*: *not-in-iff*)
qed

lemma *in-countE*:
assumes *x* ∈# *M*
obtains *n* **where** *count M x* = *Suc n*
proof −
from *assms* **have** *count M x* > *0* **by** *simp*
then obtain *n* **where** *count M x* = *Suc n*
using *gr0-conv-Suc* **by** *blast*
with *that* **show** *thesis* **.**
qed

lemma *count-greater-eq-Suc-zero-iff* [*simp*]:
count M x ≥ *Suc 0* ⟷ *x* ∈# *M*
by (*simp add*: *Suc-le-eq*)

lemma *count-greater-eq-one-iff* [*simp*]:
count M x ≥ *1* ⟷ *x* ∈# *M*
by *simp*

lemma *set-mset-empty* [*simp*]:
set-mset {#} = {}
by (*simp add*: *set-mset-def*)

lemma *set-mset-single* [*simp*]:
set-mset {#*b*#} = {*b*}
by (*simp add*: *set-mset-def*)

lemma *set-mset-eq-empty-iff* [*simp*]:
set-mset M = {} ⟷ *M* = {#}
by (*auto simp add*: *multiset-eq-iff count-eq-zero-iff*)

lemma *finite-set-mset* [*iff*]:
finite (*set-mset M*)
using *count* [*of M*] **by** (*simp add*: *multiset-def*)

65.3.2 Union

lemma *count-union* [*simp*]:
count (*M* + *N*) *a* = *count M a* + *count N a*
by (*simp add*: *plus-multiset.rep-eq*)

lemma *set-mset-union* [*simp*]:
set-mset (*M* + *N*) = *set-mset M* ∪ *set-mset N*

by (*simp only*: *set-eq-iff count-greater-zero-iff* [*symmetric*] *count-union*) *simp*

65.3.3 Difference

instance *multiset* :: (*type*) *comm-monoid-diff*
by *standard* (*transfer*; *simp add*: *fun-eq-iff*)

lemma *count-diff* [*simp*]:
count ($M - N$) *a* = *count M a* − *count N a*
by (*simp add*: *minus-multiset.rep-eq*)

lemma *in-diff-count*:
$a \in\# M - N \longleftrightarrow$ *count N a* < *count M a*
by (*simp add*: *set-mset-def*)

lemma *count-in-diffI*:
assumes $\bigwedge n$. *count N x* = *n* + *count M x* $\Longrightarrow$ *False*
shows $x \in\# M - N$
proof (*rule ccontr*)
assume $x \notin\# M - N$
then have *count N x* = (*count N x* − *count M x*) + *count M x*
by (*simp add*: *in-diff-count not-less*)
with *assms* **show** *False* **by** *auto*
qed

lemma *in-diff-countE*:
assumes $x \in\# M - N$
obtains *n* **where** *count M x* = *Suc n* + *count N x*
proof −
from *assms* **have** *count M x* − *count N x* > *0* **by** (*simp add*: *in-diff-count*)
then have *count M x* > *count N x* **by** *simp*
then obtain *n* **where** *count M x* = *Suc n* + *count N x*
using *less-iff-Suc-add* **by** *auto*
with *that* **show** *thesis* **.**
qed

lemma *in-diffD*:
assumes $a \in\# M - N$
shows $a \in\# M$
proof −
have *0* ≤ *count N a* **by** *simp*
also from *assms* **have** *count N a* < *count M a*
by (*simp add*: *in-diff-count*)
finally show *?thesis* **by** *simp*
qed

lemma *set-mset-diff*:
set-mset ($M - N$) = {*a*. *count N a* < *count M a*}
by (*simp add*: *set-mset-def*)

lemma *diff-empty* [*simp*]: $M - \{\#\} = M \wedge \{\#\} - M = \{\#\}$
  by *rule* (*fact Groups.diff-zero*, *fact Groups.zero-diff*)

lemma *diff-cancel* [*simp*]: $A - A = \{\#\}$
  by (*fact Groups.diff-cancel*)

lemma *diff-union-cancelR* [*simp*]: $M + N - N = (M::'a\ multiset)$
  by (*fact add-diff-cancel-right'*)

lemma *diff-union-cancelL* [*simp*]: $N + M - N = (M::'a\ multiset)$
  by (*fact add-diff-cancel-left'*)

lemma *diff-right-commute*:
  fixes $M\ N\ Q :: 'a\ multiset$
  shows $M - N - Q = M - Q - N$
  by (*fact diff-right-commute*)

lemma *diff-add*:
  fixes $M\ N\ Q :: 'a\ multiset$
  shows $M - (N + Q) = M - N - Q$
  by (*rule sym*) (*fact diff-diff-add*)

lemma *insert-DiffM*: $x \in\# M \Longrightarrow \{\#x\#\} + (M - \{\#x\#\}) = M$
  by (*clarsimp simp*: *multiset-eq-iff*)

lemma *insert-DiffM2* [*simp*]: $x \in\# M \Longrightarrow M - \{\#x\#\} + \{\#x\#\} = M$
  by (*clarsimp simp*: *multiset-eq-iff*)

lemma *diff-union-swap*: $a \neq b \Longrightarrow M - \{\#a\#\} + \{\#b\#\} = M + \{\#b\#\} - \{\#a\#\}$
  by (*auto simp add*: *multiset-eq-iff*)

lemma *diff-union-single-conv*:
  $a \in\# J \Longrightarrow I + J - \{\#a\#\} = I + (J - \{\#a\#\})$
  by (*simp add*: *multiset-eq-iff Suc-le-eq*)

lemma *mset-add* [*elim?*]:
  assumes $a \in\# A$
  obtains B **where** $A = B + \{\#a\#\}$
proof −
  from *assms* **have** $A = (A - \{\#a\#\}) + \{\#a\#\}$
    by *simp*
  with *that* **show** *thesis* **.**
qed

lemma *union-iff*:
  $a \in\# A + B \longleftrightarrow a \in\# A \vee a \in\# B$
  by *auto*

65.3.4 Equality of multisets

lemma *single-not-empty* [*simp*]: $\{\#a\#\} \neq \{\#\} \wedge \{\#\} \neq \{\#a\#\}$
by (*simp add*: *multiset-eq-iff*)

lemma *single-eq-single* [*simp*]: $\{\#a\#\} = \{\#b\#\} \longleftrightarrow a = b$
by (*auto simp add*: *multiset-eq-iff*)

lemma *union-eq-empty* [*iff*]: $M + N = \{\#\} \longleftrightarrow M = \{\#\} \wedge N = \{\#\}$
by (*auto simp add*: *multiset-eq-iff*)

lemma *empty-eq-union* [*iff*]: $\{\#\} = M + N \longleftrightarrow M = \{\#\} \wedge N = \{\#\}$
by (*auto simp add*: *multiset-eq-iff*)

lemma *multi-self-add-other-not-self* [*simp*]: $M = M + \{\#x\#\} \longleftrightarrow False$
by (*auto simp add*: *multiset-eq-iff*)

lemma *diff-single-trivial*: $\neg\ x \in\# M \Longrightarrow M - \{\#x\#\} = M$
by (*auto simp add*: *multiset-eq-iff not-in-iff*)

lemma *diff-single-eq-union*: $x \in\# M \Longrightarrow M - \{\#x\#\} = N \longleftrightarrow M = N + \{\#x\#\}$
by *auto*

lemma *union-single-eq-diff*: $M + \{\#x\#\} = N \Longrightarrow M = N - \{\#x\#\}$
by (*auto dest*: *sym*)

lemma *union-single-eq-member*: $M + \{\#x\#\} = N \Longrightarrow x \in\# N$
by *auto*

lemma *union-is-single*:
$M + N = \{\#a\#\} \longleftrightarrow M = \{\#a\#\} \wedge N = \{\#\} \vee M = \{\#\} \wedge N = \{\#a\#\}$
(**is** *?lhs* = *?rhs*)
proof
show *?lhs* **if** *?rhs* **using** *that* **by** *auto*
show *?rhs* **if** *?lhs*
by (*metis Multiset.diff-cancel add.commute add-diff-cancel-left′ diff-add-zero diff-single-trivial insert-DiffM that*)
qed

lemma *single-is-union*: $\{\#a\#\} = M + N \longleftrightarrow \{\#a\#\} = M \wedge N = \{\#\} \vee M = \{\#\} \wedge \{\#a\#\} = N$
by (*auto simp add*: *eq-commute* [*of* $\{\#a\#\}$ $M + N$] *union-is-single*)

lemma *add-eq-conv-diff*:
$M + \{\#a\#\} = N + \{\#b\#\} \longleftrightarrow M = N \wedge a = b \vee M = N - \{\#a\#\} + \{\#b\#\} \wedge N = M - \{\#b\#\} + \{\#a\#\}$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)

proof

show *?lhs* **if** *?rhs*
using *that*
by (*auto simp add*: *add.assoc add.commute* [*of* {#*b*#}])
(*drule sym*, *simp add*: *add.assoc* [*symmetric*])
show *?rhs* **if** *?lhs*
proof (*cases* $a = b$)
case *True* **with** ‹*?lhs*› **show** *?thesis* **by** *simp*
next
case *False*
from ‹*?lhs*› **have** $a \in\# N + \{\#b\#\}$ **by** (*rule union-single-eq-member*)
with *False* **have** $a \in\# N$ **by** *auto*
moreover from ‹*?lhs*› **have** $M = N + \{\#b\#\} - \{\#a\#\}$ **by** (*rule union-single-eq-diff*)
moreover note *False*
ultimately show *?thesis* **by** (*auto simp add*: *diff-right-commute* [*of* - {#*a*#}]
diff-union-swap)
qed
qed

lemma *insert-noteq-member*:
assumes *BC*: $B + \{\#b\#\} = C + \{\#c\#\}$
and *bnotc*: $b \neq c$
shows $c \in\# B$
proof −
have $c \in\# C + \{\#c\#\}$ **by** *simp*
have *nc*: $\neg\ c \in\# \{\#b\#\}$ **using** *bnotc* **by** *simp*
then have $c \in\# B + \{\#b\#\}$ **using** *BC* **by** *simp*
then show $c \in\# B$ **using** *nc* **by** *simp*
qed

lemma *add-eq-conv-ex*:
$(M + \{\#a\#\} = N + \{\#b\#\}) =$
$(M = N \wedge a = b \vee (\exists K.\ M = K + \{\#b\#\} \wedge N = K + \{\#a\#\}))$
by (*auto simp add*: *add-eq-conv-diff*)

lemma *multi-member-split*: $x \in\# M \Longrightarrow \exists A.\ M = A + \{\#x\#\}$
by (*rule exI* [**where** $x = M - \{\#x\#\}$]) *simp*

lemma *multiset-add-sub-el-shuffle*:
assumes $c \in\# B$
and $b \neq c$
shows $B - \{\#c\#\} + \{\#b\#\} = B + \{\#b\#\} - \{\#c\#\}$
proof −
from ‹$c \in\# B$› **obtain** *A* **where** *B*: $B = A + \{\#c\#\}$
by (*blast dest*: *multi-member-split*)
have $A + \{\#b\#\} = A + \{\#b\#\} + \{\#c\#\} - \{\#c\#\}$ **by** *simp*
then have $A + \{\#b\#\} = A + \{\#c\#\} + \{\#b\#\} - \{\#c\#\}$
by (*simp add*: *ac-simps*)
then show *?thesis* **using** *B* **by** *simp*
qed

65.3.5 Pointwise ordering induced by count

definition *subseteq-mset* :: $'a$ *multiset* $\Rightarrow$ $'a$ *multiset* $\Rightarrow$ *bool* (**infix** $\subseteq\#$ *50*)
where $A \subseteq\# B = (\forall a.\ count\ A\ a \leq count\ B\ a)$

definition *subset-mset* :: $'a$ *multiset* $\Rightarrow$ $'a$ *multiset* $\Rightarrow$ *bool* (**infix** $\subset\#$ *50*)
where $A \subset\# B = (A \subseteq\# B \wedge A \neq B)$

abbreviation (*input*) *supseteq-mset* :: $'a$ *multiset* $\Rightarrow$ $'a$ *multiset* $\Rightarrow$ *bool* (**infix** $\supseteq\#$ *50*)
where *supseteq-mset* $A\ B \equiv B \subseteq\# A$

abbreviation (*input*) *supset-mset* :: $'a$ *multiset* $\Rightarrow$ $'a$ *multiset* $\Rightarrow$ *bool* (**infix** $\supset\#$ *50*)
where *supset-mset* $A\ B \equiv B \subset\# A$

notation (*input*)
subseteq-mset (**infix** $\leq\#$ *50*) **and**
supseteq-mset (**infix** $\geq\#$ *50*)

notation (*ASCII*)
subseteq-mset (**infix** $<=\#$ *50*) **and**
subset-mset (**infix** $<\#$ *50*) **and**
supseteq-mset (**infix** $>=\#$ *50*) **and**
supset-mset (**infix** $>\#$ *50*)

interpretation *subset-mset*: *ordered-ab-semigroup-add-imp-le op* + *op* − *op* $\subseteq\#$ *op* $\subset\#$
by *standard* (*auto simp add*: *subset-mset-def subseteq-mset-def multiset-eq-iff intro*: *order-trans antisym*)
— FIXME: avoid junk stemming from type class interpretation

lemma *mset-less-eqI*:
$(\bigwedge a.\ count\ A\ a \leq count\ B\ a) \Longrightarrow A \subseteq\# B$
by (*simp add*: *subseteq-mset-def*)

lemma *mset-less-eq-count*:
$A \subseteq\# B \Longrightarrow count\ A\ a \leq count\ B\ a$
by (*simp add*: *subseteq-mset-def*)

lemma *mset-le-exists-conv*: $(A{::}'a\ multiset) \subseteq\# B \longleftrightarrow (\exists C.\ B = A + C)$
unfolding *subseteq-mset-def*
apply (*rule iffI*)
apply (*rule exI* [**where** $x = B - A$])
apply (*auto intro*: *multiset-eq-iff* [*THEN iffD2*])
done

interpretation *subset-mset*: *ordered-cancel-comm-monoid-diff op* + *0 op* $\leq\#$ *op* $<\#$ *op* −
by *standard* (*simp*, *fact mset-le-exists-conv*)

declare *subset-mset.zero-order*[*simp del*]
— this removes some simp rules not in the usual order for multisets

lemma *mset-le-mono-add-right-cancel* [*simp*]: $(A::'a\ multiset) + C \subseteq\# B + C \longleftrightarrow A \subseteq\# B$
by (*fact subset-mset.add-le-cancel-right*)

lemma *mset-le-mono-add-left-cancel* [*simp*]: $C + (A::'a\ multiset) \subseteq\# C + B \longleftrightarrow A \subseteq\# B$
by (*fact subset-mset.add-le-cancel-left*)

lemma *mset-le-mono-add*: $(A::'a\ multiset) \subseteq\# B \Longrightarrow C \subseteq\# D \Longrightarrow A + C \subseteq\# B + D$
by (*fact subset-mset.add-mono*)

lemma *mset-le-add-left* [*simp*]: $(A::'a\ multiset) \subseteq\# A + B$
unfolding *subseteq-mset-def* **by** *auto*

lemma *mset-le-add-right* [*simp*]: $B \subseteq\# (A::'a\ multiset) + B$
unfolding *subseteq-mset-def* **by** *auto*

lemma *single-subset-iff* [*simp*]:
$\{\#a\#\} \subseteq\# M \longleftrightarrow a \in\# M$
by (*auto simp add*: *subseteq-mset-def Suc-le-eq*)

lemma *mset-le-single*: $a \in\# B \Longrightarrow \{\#a\#\} \subseteq\# B$
by (*simp add*: *subseteq-mset-def Suc-le-eq*)

lemma *multiset-diff-union-assoc*:
fixes $A\ B\ C\ D :: 'a\ multiset$
shows $C \subseteq\# B \Longrightarrow A + B - C = A + (B - C)$
by (*fact subset-mset.diff-add-assoc*)

lemma *mset-le-multiset-union-diff-commute*:
fixes $A\ B\ C\ D :: 'a\ multiset$
shows $B \subseteq\# A \Longrightarrow A - B + C = A + C - B$
by (*fact subset-mset.add-diff-assoc2*)

lemma *diff-le-self*[*simp*]:
$(M::'a\ multiset) - N \subseteq\# M$
by (*simp add*: *subseteq-mset-def*)

lemma *mset-leD*:
assumes $A \subseteq\# B$ **and** $x \in\# A$
shows $x \in\# B$
proof −
from ‹$x \in\# A$› **have** $count\ A\ x > 0$ **by** *simp*
also from ‹$A \subseteq\# B$› **have** $count\ A\ x \le count\ B\ x$

by (*simp add*: *subseteq-mset-def*)
finally show *?thesis* **by** *simp*
qed

lemma *mset-lessD*:
$A \subset\# B \Longrightarrow x \in\# A \Longrightarrow x \in\# B$
by (*auto intro*: *mset-leD* [*of A*])

lemma *set-mset-mono*:
$A \subseteq\# B \Longrightarrow$ *set-mset* $A \subseteq$ *set-mset* B
by (*metis mset-leD subsetI*)

lemma *mset-le-insertD*:
$A + \{\#x\#\} \subseteq\# B \Longrightarrow x \in\# B \wedge A \subset\# B$
apply (*rule conjI*)
apply (*simp add*: *mset-leD*)
apply (*clarsimp simp*: *subset-mset-def subseteq-mset-def*)
apply *safe*
apply (*erule-tac* $x = a$ **in** *allE*)
apply (*auto split*: *if-split-asm*)
done

lemma *mset-less-insertD*:
$A + \{\#x\#\} \subset\# B \Longrightarrow x \in\# B \wedge A \subset\# B$
by (*rule mset-le-insertD*) *simp*

lemma *mset-less-of-empty*[*simp*]: $A \subset\# \{\#\} \longleftrightarrow$ *False*
by (*auto simp add*: *subseteq-mset-def subset-mset-def multiset-eq-iff*)

lemma *empty-le* [*simp*]: $\{\#\} \subseteq\# A$
unfolding *mset-le-exists-conv* **by** *auto*

lemma *insert-subset-eq-iff*:
$\{\#a\#\} + A \subseteq\# B \longleftrightarrow a \in\# B \wedge A \subseteq\# B - \{\#a\#\}$
using *le-diff-conv2* [*of Suc 0 count B a count A a*]
apply (*auto simp add*: *subseteq-mset-def not-in-iff Suc-le-eq*)
apply (*rule ccontr*)
apply (*auto simp add*: *not-in-iff*)
done

lemma *insert-union-subset-iff*:
$\{\#a\#\} + A \subset\# B \longleftrightarrow a \in\# B \wedge A \subset\# B - \{\#a\#\}$
by (*auto simp add*: *insert-subset-eq-iff subset-mset-def insert-DiffM*)

lemma *subset-eq-diff-conv*:
$A - C \subseteq\# B \longleftrightarrow A \subseteq\# B + C$
by (*simp add*: *subseteq-mset-def le-diff-conv*)

lemma *le-empty* [*simp*]: $M \subseteq\# \{\#\} \longleftrightarrow M = \{\#\}$

unfolding *mset-le-exists-conv* **by** *auto*

lemma *multi-psub-of-add-self* [*simp*]: $A \subset\# A + \{\#x\#\}$
by (*auto simp*: *subset-mset-def subseteq-mset-def*)

lemma *multi-psub-self* [*simp*]: $(A::'a\ multiset) \subset\# A = False$
by *simp*

lemma *mset-less-add-bothsides*: $N + \{\#x\#\} \subset\# M + \{\#x\#\} \Longrightarrow N \subset\# M$
by (*fact subset-mset.add-less-imp-less-right*)

lemma *mset-less-empty-nonempty*: $\{\#\} \subset\# S \longleftrightarrow S \neq \{\#\}$
by (*fact subset-mset.zero-less-iff-neq-zero*)

lemma *mset-less-diff-self*: $c \in\# B \Longrightarrow B - \{\#c\#\} \subset\# B$
by (*auto simp*: *subset-mset-def elim*: *mset-add*)

65.3.6 Intersection

definition *inf-subset-mset* :: *'a multiset* ⇒ *'a multiset* ⇒ *'a multiset* (**infixl** #∩ *70*) **where**
multiset-inter-def: *inf-subset-mset* $A\ B = A - (A - B)$

interpretation *subset-mset*: *semilattice-inf inf-subset-mset op* ⊆# *op* ⊂#
proof −
have [*simp*]: $m \le n \Longrightarrow m \le q \Longrightarrow m \le n - (n - q)$ **for** $m\ n\ q :: nat$
by *arith*
show *class.semilattice-inf op* #∩ *op* ⊆# *op* ⊂#
by *standard* (*auto simp add*: *multiset-inter-def subseteq-mset-def*)
qed
— FIXME: avoid junk stemming from type class interpretation

lemma *multiset-inter-count* [*simp*]:
fixes $A\ B :: 'a\ multiset$
shows *count* $(A \#\cap B)\ x = min\ (count\ A\ x)\ (count\ B\ x)$
by (*simp add*: *multiset-inter-def*)

lemma *set-mset-inter* [*simp*]:
set-mset $(A \#\cap B)$ = *set-mset* $A \cap$ *set-mset* B
by (*simp only*: *set-eq-iff count-greater-zero-iff* [*symmetric*] *multiset-inter-count*) *simp*

lemma *diff-intersect-left-idem* [*simp*]:
$M - M \#\cap N = M - N$
by (*simp add*: *multiset-eq-iff min-def*)

lemma *diff-intersect-right-idem* [*simp*]:
$M - N \#\cap M = M - N$
by (*simp add*: *multiset-eq-iff min-def*)

lemma *multiset-inter-single*: $a \neq b \Longrightarrow \{\#a\#\} \ \#\cap \ \{\#b\#\} = \{\#\}$
  by (*rule multiset-eqI*) *auto*

lemma *multiset-union-diff-commute*:
  assumes $B \ \#\cap \ C = \{\#\}$
  shows $A + B - C = A - C + B$
proof (*rule multiset-eqI*)
  fix x
  from *assms* **have** *min* (*count B x*) (*count C x*) = *0*
    by (*auto simp add*: *multiset-eq-iff*)
  then have *count B x* = *0* ∨ *count C x* = *0*
    unfolding *min-def* **by** (*auto split*: *if-splits*)
  then show *count* $(A + B - C)$ *x* = *count* $(A - C + B)$ *x*
    by *auto*
qed

lemma *disjunct-not-in*:
  $A \ \#\cap \ B = \{\#\} \longleftrightarrow (\forall a. \ a \notin\# \ A \vee a \notin\# \ B)$ (**is** *?P* ⟷ *?Q*)
proof
  assume *?P*
  show *?Q*
  proof
    fix a
    from ‹*?P*› **have** *min* (*count A a*) (*count B a*) = *0*
      by (*simp add*: *multiset-eq-iff*)
    then have *count A a* = *0* ∨ *count B a* = *0*
      by (*cases count A a* ≤ *count B a*) (*simp-all add*: *min-def*)
    then show $a \notin\# \ A \vee a \notin\# \ B$
      by (*simp add*: *not-in-iff*)
  qed
next
  assume *?Q*
  show *?P*
  proof (*rule multiset-eqI*)
    fix a
    from ‹*?Q*› **have** *count A a* = *0* ∨ *count B a* = *0*
      by (*auto simp add*: *not-in-iff*)
    then show *count* $(A \ \#\cap \ B)$ *a* = *count* $\{\#\}$ *a*
      by *auto*
  qed
qed

lemma *empty-inter* [*simp*]: $\{\#\} \ \#\cap \ M = \{\#\}$
  by (*simp add*: *multiset-eq-iff*)

lemma *inter-empty* [*simp*]: $M \ \#\cap \ \{\#\} = \{\#\}$
  by (*simp add*: *multiset-eq-iff*)

lemma *inter-add-left1*: $\neg\ x \in\#\ N \Longrightarrow (M + \{\#x\#\})\ \#\cap\ N = M\ \#\cap\ N$
by (*simp add*: *multiset-eq-iff not-in-iff*)

lemma *inter-add-left2*: $x \in\#\ N \Longrightarrow (M + \{\#x\#\})\ \#\cap\ N = (M\ \#\cap\ (N - \{\#x\#\})) + \{\#x\#\}$
by (*auto simp add*: *multiset-eq-iff elim*: *mset-add*)

lemma *inter-add-right1*: $\neg\ x \in\#\ N \Longrightarrow N\ \#\cap\ (M + \{\#x\#\}) = N\ \#\cap\ M$
by (*simp add*: *multiset-eq-iff not-in-iff*)

lemma *inter-add-right2*: $x \in\#\ N \Longrightarrow N\ \#\cap\ (M + \{\#x\#\}) = ((N - \{\#x\#\})\ \#\cap\ M) + \{\#x\#\}$
by (*auto simp add*: *multiset-eq-iff elim*: *mset-add*)

lemma *disjunct-set-mset-diff*:
assumes $M\ \#\cap\ N = \{\#\}$
shows *set-mset* $(M - N) =$ *set-mset* M
proof (*rule set-eqI*)
fix a
from *assms* **have** $a \notin\#\ M \vee a \notin\#\ N$
by (*simp add*: *disjunct-not-in*)
then show $a \in\#\ M - N \longleftrightarrow a \in\#\ M$
by (*auto dest*: *in-diffD*) (*simp add*: *in-diff-count not-in-iff*)
qed

lemma *at-most-one-mset-mset-diff*:
assumes $a \notin\#\ M - \{\#a\#\}$
shows *set-mset* $(M - \{\#a\#\}) =$ *set-mset* $M - \{a\}$
using *assms* **by** (*auto simp add*: *not-in-iff in-diff-count set-eq-iff*)

lemma *more-than-one-mset-mset-diff*:
assumes $a \in\#\ M - \{\#a\#\}$
shows *set-mset* $(M - \{\#a\#\}) =$ *set-mset* M
proof (*rule set-eqI*)
fix b
have *Suc 0* $<$ *count* $M\ b \Longrightarrow$ *count* $M\ b > 0$ **by** *arith*
then show $b \in\#\ M - \{\#a\#\} \longleftrightarrow b \in\#\ M$
using *assms* **by** (*auto simp add*: *in-diff-count*)
qed

lemma *inter-iff*:
$a \in\#\ A\ \#\cap\ B \longleftrightarrow a \in\#\ A \wedge a \in\#\ B$
by *simp*

lemma *inter-union-distrib-left*:
$A\ \#\cap\ B + C = (A + C)\ \#\cap\ (B + C)$
by (*simp add*: *multiset-eq-iff min-add-distrib-left*)

lemma *inter-union-distrib-right*:

$C + A \#\cap B = (C + A) \#\cap (C + B)$
using *inter-union-distrib-left* [*of A B C*] **by** (*simp add*: *ac-simps*)

lemma *inter-subset-eq-union*:
$A \#\cap B \subseteq\# A + B$
by (*auto simp add*: *subseteq-mset-def*)

65.3.7 Bounded union

definition *sup-subset-mset* :: *'a multiset* ⇒ *'a multiset* ⇒ *'a multiset*(**infixl** #∪ *70*)
where *sup-subset-mset A B* = *A* + (*B* − *A*) — FIXME irregular fact name

interpretation *subset-mset*: *semilattice-sup sup-subset-mset op* ⊆# *op* ⊂#
proof −
have [*simp*]: $m \leq n \Longrightarrow q \leq n \Longrightarrow m + (q - m) \leq n$ **for** *m n q* :: *nat*
by *arith*
show *class.semilattice-sup op* #∪ *op* ⊆# *op* ⊂#
by *standard* (*auto simp add*: *sup-subset-mset-def subseteq-mset-def*)
qed
— FIXME: avoid junk stemming from type class interpretation

lemma *sup-subset-mset-count* [*simp*]: — FIXME irregular fact name
count (*A* #∪ *B*) *x* = *max* (*count A x*) (*count B x*)
by (*simp add*: *sup-subset-mset-def*)

lemma *set-mset-sup* [*simp*]:
set-mset (*A* #∪ *B*) = *set-mset A* ∪ *set-mset B*
by (*simp only*: *set-eq-iff count-greater-zero-iff* [*symmetric*] *sup-subset-mset-count*)
(*auto simp add*: *not-in-iff elim*: *mset-add*)

lemma *empty-sup* [*simp*]: {#} #∪ *M* = *M*
by (*simp add*: *multiset-eq-iff*)

lemma *sup-empty* [*simp*]: *M* #∪ {#} = *M*
by (*simp add*: *multiset-eq-iff*)

lemma *sup-union-left1*: ¬ *x* ∈# *N* ⟹ (*M* + {#*x*#}) #∪ *N* = (*M* #∪ *N*) + {#*x*#}
by (*simp add*: *multiset-eq-iff not-in-iff*)

lemma *sup-union-left2*: *x* ∈# *N* ⟹ (*M* + {#*x*#}) #∪ *N* = (*M* #∪ (*N* − {#*x*#})) + {#*x*#}
by (*simp add*: *multiset-eq-iff*)

lemma *sup-union-right1*: ¬ *x* ∈# *N* ⟹ *N* #∪ (*M* + {#*x*#}) = (*N* #∪ *M*) + {#*x*#}
by (*simp add*: *multiset-eq-iff not-in-iff*)

lemma *sup-union-right2*: $x \in\# N \Longrightarrow N \#\cup (M + \{\#x\#\}) = ((N - \{\#x\#\}) \#\cup M) + \{\#x\#\}$
by (*simp add*: *multiset-eq-iff*)

lemma *sup-union-distrib-left*:
$A \#\cup B + C = (A + C) \#\cup (B + C)$
by (*simp add*: *multiset-eq-iff max-add-distrib-left*)

lemma *union-sup-distrib-right*:
$C + A \#\cup B = (C + A) \#\cup (C + B)$
using *sup-union-distrib-left* [*of A B C*] **by** (*simp add*: *ac-simps*)

lemma *union-diff-inter-eq-sup*:
$A + B - A \#\cap B = A \#\cup B$
by (*auto simp add*: *multiset-eq-iff*)

lemma *union-diff-sup-eq-inter*:
$A + B - A \#\cup B = A \#\cap B$
by (*auto simp add*: *multiset-eq-iff*)

65.3.8 Subset is an order

interpretation *subset-mset*: *order op* ≤# *op* <# **by** *unfold-locales auto*

65.3.9 Filter (with comprehension syntax)

Multiset comprehension

lift-definition *filter-mset* :: ('a ⇒ *bool*) ⇒ 'a *multiset* ⇒ 'a *multiset*
is $\lambda P\ M.\ \lambda x.$ *if P x then M x else 0*
by (*rule filter-preserves-multiset*)

syntax (*ASCII*)
-MCollect :: *pttrn* ⇒ 'a *multiset* ⇒ *bool* ⇒ 'a *multiset* ((*1*{# - :# -./ -#}))
syntax
-MCollect :: *pttrn* ⇒ 'a *multiset* ⇒ *bool* ⇒ 'a *multiset* ((*1*{# - ∈# -./ -#}))
translations
{#x ∈# M. P#} == *CONST filter-mset* (λx. P) M

lemma *count-filter-mset* [*simp*]:
count (*filter-mset P M*) *a* = (*if P a then count M a else 0*)
by (*simp add*: *filter-mset.rep-eq*)

lemma *set-mset-filter* [*simp*]:
set-mset (*filter-mset P M*) = {*a* ∈ *set-mset M*. *P a*}
by (*simp only*: *set-eq-iff count-greater-zero-iff* [*symmetric*] *count-filter-mset*) *simp*

lemma *filter-empty-mset* [*simp*]: *filter-mset P* {#} = {#}
by (*rule multiset-eqI*) *simp*

lemma *filter-single-mset* [*simp*]: *filter-mset P* {#*x*#} = (*if P x then* {#*x*#} *else* {#})
  by (*rule multiset-eqI*) *simp*

lemma *filter-union-mset* [*simp*]: *filter-mset P* (*M* + *N*) = *filter-mset P M* + *filter-mset P N*
  by (*rule multiset-eqI*) *simp*

lemma *filter-diff-mset* [*simp*]: *filter-mset P* (*M* − *N*) = *filter-mset P M* − *filter-mset P N*
  by (*rule multiset-eqI*) *simp*

lemma *filter-inter-mset* [*simp*]: *filter-mset P* (*M* #∩ *N*) = *filter-mset P M* #∩ *filter-mset P N*
  by (*rule multiset-eqI*) *simp*

lemma *multiset-filter-subset*[*simp*]: *filter-mset f M* ⊆# *M*
  by (*simp add*: *mset-less-eqI*)

lemma *multiset-filter-mono*:
  assumes *A* ⊆# *B*
  shows *filter-mset f A* ⊆# *filter-mset f B*
proof −
  from *assms*[*unfolded mset-le-exists-conv*]
  obtain *C* **where** *B*: *B* = *A* + *C* **by** *auto*
  show *?thesis* **unfolding** *B* **by** *auto*
qed

lemma *filter-mset-eq-conv*:
  filter-mset P M = *N* ⟷ *N* ⊆# *M* ∧ (∀ *b*∈#*N*. *P b*) ∧ (∀ *a*∈#*M* − *N*. ¬ *P a*)
(**is** *?P* ⟷ *?Q*)
proof
  assume *?P* **then show** *?Q* **by** *auto* (*simp add*: *multiset-eq-iff in-diff-count*)
next
  assume *?Q*
  then obtain *Q* **where** *M*: *M* = *N* + *Q*
    by (*auto simp add*: *mset-le-exists-conv*)
  then have *MN*: *M* − *N* = *Q* **by** *simp*
  show *?P*
  proof (*rule multiset-eqI*)
    fix *a*
    from ‹*?Q*› *MN* **have** ∗: ¬ *P a* ⟹ *a* ∉# *N* *P a* ⟹ *a* ∉# *Q*
      by *auto*
    show *count* (*filter-mset P M*) *a* = *count N a*
    proof (*cases a* ∈# *M*)
      case *True*
      with ∗ **show** *?thesis*
        by (*simp add*: *not-in-iff M*)
    next

case *False* **then have** *count M a = 0*
by (*simp add*: *not-in-iff*)
with *M* **show** *?thesis* **by** *simp*
qed
qed
qed

65.3.10 Size

definition *wcount* **where** *wcount f M* = (λ*x*. *count M x* ∗ *Suc* (*f x*))

lemma *wcount-union*: *wcount f* (*M* + *N*) *a* = *wcount f M a* + *wcount f N a*
by (*auto simp*: *wcount-def add-mult-distrib*)

definition *size-multiset* :: (*'a* ⇒ *nat*) ⇒ *'a multiset* ⇒ *nat* **where**
size-multiset f M = *setsum* (*wcount f M*) (*set-mset M*)

lemmas *size-multiset-eq* = *size-multiset-def*[*unfolded wcount-def*]

instantiation *multiset* :: (*type*) *size*
begin

definition *size-multiset* **where**
size-multiset-overloaded-def: *size-multiset* = *Multiset.size-multiset* (λ-. *0*)
instance ..

end

lemmas *size-multiset-overloaded-eq* =
size-multiset-overloaded-def[*THEN fun-cong*, *unfolded size-multiset-eq*, *simplified*]

lemma *size-multiset-empty* [*simp*]: *size-multiset f* {#} = *0*
by (*simp add*: *size-multiset-def*)

lemma *size-empty* [*simp*]: *size* {#} = *0*
by (*simp add*: *size-multiset-overloaded-def*)

lemma *size-multiset-single* [*simp*]: *size-multiset f* {#*b*#} = *Suc* (*f b*)
by (*simp add*: *size-multiset-eq*)

lemma *size-single* [*simp*]: *size* {#*b*#} = *1*
by (*simp add*: *size-multiset-overloaded-def*)

lemma *setsum-wcount-Int*:
finite A ⟹ *setsum* (*wcount f N*) (*A* ∩ *set-mset N*) = *setsum* (*wcount f N*) *A*
by (*induct rule*: *finite-induct*)
(*simp-all add*: *Int-insert-left wcount-def count-eq-zero-iff*)

lemma *size-multiset-union* [*simp*]:

size-multiset f $(M + N::'a\ multiset)$ = *size-multiset f M* + *size-multiset f N*
apply (*simp add*: *size-multiset-def setsum-Un-nat setsum.distrib setsum-wcount-Int wcount-union*)
apply (*subst Int-commute*)
apply (*simp add*: *setsum-wcount-Int*)
done

lemma *size-union* [*simp*]: *size* $(M + N::'a\ multiset)$ = *size M* + *size N*
by (*auto simp add*: *size-multiset-overloaded-def*)

lemma *size-multiset-eq-0-iff-empty* [*iff*]:
size-multiset f M $= 0 \longleftrightarrow M = \{\#\}$
by (*auto simp add*: *size-multiset-eq count-eq-zero-iff*)

lemma *size-eq-0-iff-empty* [*iff*]: (*size* $M = 0) = (M = \{\#\})$
by (*auto simp add*: *size-multiset-overloaded-def*)

lemma *nonempty-has-size*: $(S \neq \{\#\}) = (0 <$ *size* $S)$
by (*metis gr0I gr-implies-not0 size-empty size-eq-0-iff-empty*)

lemma *size-eq-Suc-imp-elem*: *size* $M =$ *Suc* $n \Longrightarrow \exists\, a.\ a \in\# M$
apply (*unfold size-multiset-overloaded-eq*)
apply (*drule setsum-SucD*)
apply *auto*
done

lemma *size-eq-Suc-imp-eq-union*:
assumes *size* $M =$ *Suc* n
shows $\exists\, a\ N.\ M = N + \{\#a\#\}$
proof −
from *assms* **obtain** a **where** $a \in\# M$
by (*erule size-eq-Suc-imp-elem* [*THEN exE*])
then have $M = M - \{\#a\#\} + \{\#a\#\}$ **by** *simp*
then show *?thesis* **by** *blast*
qed

lemma *size-mset-mono*:
fixes $A\ B$ :: *'a multiset*
assumes $A \subseteq\# B$
shows *size* $A \leq$ *size* B
proof −
from *assms*[*unfolded mset-le-exists-conv*]
obtain C **where** B: $B = A + C$ **by** *auto*
show *?thesis* **unfolding** B **by** (*induct* C) *auto*
qed

lemma *size-filter-mset-lesseq*[*simp*]: *size* (*filter-mset f M*) $\leq$ *size M*
by (*rule size-mset-mono*[*OF multiset-filter-subset*])

lemma *size-Diff-submset*:
  $M \subseteq\# M' \Longrightarrow size\ (M' - M) = size\ M' - size(M::'a\ multiset)$
by (*metis add-diff-cancel-left' size-union mset-le-exists-conv*)

65.4 Induction and case splits

theorem *multiset-induct* [*case-names empty add*, *induct type*: *multiset*]:
  assumes *empty*: $P\ \{\#\}$
  assumes *add*: $\bigwedge M\ x.\ P\ M \Longrightarrow P\ (M + \{\#x\#\})$
  shows $P\ M$
proof (*induct* $n \equiv size\ M$ *arbitrary*: M)
  case *0* **thus** $P\ M$ **by** (*simp add*: *empty*)
next
  case (*Suc k*)
  obtain $N\ x$ **where** $M = N + \{\#x\#\}$
    using ‹*Suc k = size M*› [*symmetric*]
    using *size-eq-Suc-imp-eq-union* **by** *fast*
  with *Suc add* **show** $P\ M$ **by** *simp*
qed

lemma *multi-nonempty-split*: $M \neq \{\#\} \Longrightarrow \exists A\ a.\ M = A + \{\#a\#\}$
by (*induct M*) *auto*

lemma *multiset-cases* [*cases type*]:
  obtains (*empty*) $M = \{\#\}$
    | (*add*) $N\ x$ **where** $M = N + \{\#x\#\}$
  using *assms* **by** (*induct M*) *simp-all*

lemma *multi-drop-mem-not-eq*: $c \in\# B \Longrightarrow B - \{\#c\#\} \neq B$
by (*cases* $B = \{\#\}$) (*auto dest*: *multi-member-split*)

lemma *multiset-partition*: $M = \{\#\ x{\in}\#M.\ P\ x\ \#\} + \{\#\ x{\in}\#M.\ \neg\ P\ x\ \#\}$
apply (*subst multiset-eq-iff*)
apply *auto*
done

lemma *mset-less-size*: $(A::'a\ multiset) \subset\# B \Longrightarrow size\ A < size\ B$
proof (*induct A arbitrary*: *B*)
  case (*empty M*)
  then have $M \neq \{\#\}$ **by** (*simp add*: *mset-less-empty-nonempty*)
  then obtain $M'\ x$ **where** $M = M' + \{\#x\#\}$
    by (*blast dest*: *multi-nonempty-split*)
  then show *?case* **by** *simp*
next
  case (*add S x T*)
  have *IH*: $\bigwedge B.\ S \subset\# B \Longrightarrow size\ S < size\ B$ **by** *fact*
  have *SxsubT*: $S + \{\#x\#\} \subset\# T$ **by** *fact*
  then have $x \in\# T$ **and** $S \subset\# T$
    by (*auto dest*: *mset-less-insertD*)

then obtain T' **where** T: $T = T' + \{\#x\#\}$
  by (*blast dest*: *multi-member-split*)
then have $S \subset\# T'$ **using** *SxsubT*
  by (*blast intro*: *mset-less-add-bothsides*)
then have *size* $S <$ *size* T' **using** *IH* **by** *simp*
then show *?case* **using** T **by** *simp*
qed

lemma *size-1-singleton-mset*: *size* $M = 1 \Longrightarrow \exists\, a.\ M = \{\#a\#\}$
by (*cases M*) *auto*

65.4.1 Strong induction and subset induction for multisets

Well-foundedness of strict subset relation

lemma *wf-less-mset-rel*: *wf* $\{(M, N :: 'a\ multiset).\ M \subset\# N\}$
apply (*rule wf-measure* [*THEN wf-subset*, **where** *f1=size*])
apply (*clarsimp simp*: *measure-def inv-image-def mset-less-size*)
done

lemma *full-multiset-induct* [*case-names less*]:
assumes *ih*: $\bigwedge B.\ \forall (A::'a\ multiset).\ A \subset\# B \longrightarrow P\ A \Longrightarrow P\ B$
shows $P\ B$
apply (*rule wf-less-mset-rel* [*THEN wf-induct*])
apply (*rule ih*, *auto*)
done

lemma *multi-subset-induct* [*consumes 2*, *case-names empty add*]:
assumes $F \subseteq\# A$
  and *empty*: $P\ \{\#\}$
  and *insert*: $\bigwedge a\ F.\ a \in\# A \Longrightarrow P\ F \Longrightarrow P\ (F + \{\#a\#\})$
shows $P\ F$
proof −
  from ‹$F \subseteq\# A$›
  show *?thesis*
  proof (*induct F*)
    show $P\ \{\#\}$ **by** *fact*
  next
    fix $x\ F$
    assume P: $F \subseteq\# A \Longrightarrow P\ F$ **and** i: $F + \{\#x\#\} \subseteq\# A$
    show $P\ (F + \{\#x\#\})$
    proof (*rule insert*)
      from i **show** $x \in\# A$ **by** (*auto dest*: *mset-le-insertD*)
      from i **have** $F \subseteq\# A$ **by** (*auto dest*: *mset-le-insertD*)
      with P **show** $P\ F$ **.**
    qed
  qed
qed

65.5 The fold combinator

definition *fold-mset* :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a$ *multiset* $\Rightarrow 'b$
where
fold-mset f s M = *Finite-Set.fold* (λx. *f x* ^^ *count M x*) *s* (*set-mset M*)

lemma *fold-mset-empty* [*simp*]: *fold-mset f s* {#} = *s*
by (*simp add*: *fold-mset-def*)

context *comp-fun-commute*
begin

lemma *fold-mset-insert*: *fold-mset f s* (*M* + {#*x*#}) = *f x* (*fold-mset f s M*)
proof −
interpret *mset*: *comp-fun-commute* λy. *f y* ^^ *count M y*
by (*fact comp-fun-commute-funpow*)
interpret *mset-union*: *comp-fun-commute* λy. *f y* ^^ *count* (*M* + {#*x*#}) *y*
by (*fact comp-fun-commute-funpow*)
show *?thesis*
proof (*cases x* $\in$ *set-mset M*)
case *False*
then have ∗: *count* (*M* + {#*x*#}) *x* = *1*
by (*simp add*: *not-in-iff*)
from *False* **have** *Finite-Set.fold* (λy. *f y* ^^ *count* (*M* + {#*x*#}) *y*) *s* (*set-mset M*) =
Finite-Set.fold (λy. *f y* ^^ *count M y*) *s* (*set-mset M*)
by (*auto intro*!: *Finite-Set.fold-cong comp-fun-commute-funpow*)
with *False* ∗ **show** *?thesis*
by (*simp add*: *fold-mset-def del*: *count-union*)
next
case *True*
def *N* $\equiv$ *set-mset M* − {*x*}
from *N-def True* **have** ∗: *set-mset M* = *insert x N x* $\notin$ *N finite N* **by** *auto*
then have *Finite-Set.fold* (λy. *f y* ^^ *count* (*M* + {#*x*#}) *y*) *s N* =
Finite-Set.fold (λy. *f y* ^^ *count M y*) *s N*
by (*auto intro*!: *Finite-Set.fold-cong comp-fun-commute-funpow*)
with ∗ **show** *?thesis* **by** (*simp add*: *fold-mset-def del*: *count-union*) *simp*
qed
qed

corollary *fold-mset-single* [*simp*]: *fold-mset f s* {#*x*#} = *f x s*
proof −
have *fold-mset f s* ({#} + {#*x*#}) = *f x s* **by** (*simp only*: *fold-mset-insert*) *simp*
then show *?thesis* **by** *simp*
qed

lemma *fold-mset-fun-left-comm*: *f x* (*fold-mset f s M*) = *fold-mset f* (*f x s*) *M*
by (*induct M*) (*simp-all add*: *fold-mset-insert fun-left-comm*)

lemma *fold-mset-union* [*simp*]: *fold-mset f s* (*M* + *N*) = *fold-mset f* (*fold-mset f*

s M) *N*
proof (*induct M*)
  case *empty* **then show** *?case* **by** *simp*
next
  case (*add M x*)
  have $M + \{\#x\#\} + N = (M + N) + \{\#x\#\}$
    by (*simp add*: *ac-simps*)
  with *add* **show** *?case* **by** (*simp add*: *fold-mset-insert fold-mset-fun-left-comm*)
qed

lemma *fold-mset-fusion*:
  assumes *comp-fun-commute g*
    and ∗: $\bigwedge x\ y.\ h\ (g\ x\ y) = f\ x\ (h\ y)$
  shows *h* (*fold-mset g w A*) = *fold-mset f* (*h w*) *A*
proof −
  interpret *comp-fun-commute g* **by** (*fact assms*)
  from ∗ **show** *?thesis* **by** (*induct A*) *auto*
qed

end

A note on code generation: When defining some function containing a subterm *fold-mset F*, code generation is not automatic. When interpreting locale *left-commutative* with *F*, the would be code thms for *fold-mset* become thms like *fold-mset F z* {#} = *z* where *F* is not a pattern but contains defined symbols, i.e. is not a code thm. Hence a separate constant with its own code thms needs to be introduced for *F*. See the image operator below.

65.6 Image

definition *image-mset* :: ('*a* ⇒ '*b*) ⇒ '*a multiset* ⇒ '*b multiset* **where**
  image-mset f = *fold-mset* (*plus* ∘ *single* ∘ *f*) {#}

lemma *comp-fun-commute-mset-image*: *comp-fun-commute* (*plus* ∘ *single* ∘ *f*)
proof
qed (*simp add*: *ac-simps fun-eq-iff*)

lemma *image-mset-empty* [*simp*]: *image-mset f* {#} = {#}
  by (*simp add*: *image-mset-def*)

lemma *image-mset-single* [*simp*]: *image-mset f* {#*x*#} = {#*f x*#}
proof −
  interpret *comp-fun-commute plus* ∘ *single* ∘ *f*
    by (*fact comp-fun-commute-mset-image*)
  show *?thesis* **by** (*simp add*: *image-mset-def*)
qed

lemma *image-mset-union* [*simp*]: *image-mset f* (*M* + *N*) = *image-mset f M* + *image-mset f N*

proof −
  interpret *comp-fun-commute plus* ∘ *single* ∘ *f*
    by (*fact comp-fun-commute-mset-image*)
  show *?thesis* **by** (*induct N*) (*simp-all add*: *image-mset-def ac-simps*)
qed

corollary *image-mset-insert*: *image-mset f* (*M* + {#*a*#}) = *image-mset f M* + {#*f a*#}
  by *simp*

lemma *set-image-mset* [*simp*]: *set-mset* (*image-mset f M*) = *image f* (*set-mset M*)
  by (*induct M*) *simp-all*

lemma *size-image-mset* [*simp*]: *size* (*image-mset f M*) = *size M*
  by (*induct M*) *simp-all*

lemma *image-mset-is-empty-iff* [*simp*]: *image-mset f M* = {#} ⟷ *M* = {#}
  by (*cases M*) *auto*

syntax (*ASCII*)
  -comprehension-mset :: ′*a* ⇒ ′*b* ⇒ ′*b multiset* ⇒ ′*a multiset* (({#-/. - :# -#}))
syntax
  -comprehension-mset :: ′*a* ⇒ ′*b* ⇒ ′*b multiset* ⇒ ′*a multiset* (({#-/. - ∈# -#}))
translations
  {#*e*. *x* ∈# *M*#} ⇌ *CONST image-mset* (λ*x*. *e*) *M*

syntax (*ASCII*)
  -comprehension-mset′ :: ′*a* ⇒ ′*b* ⇒ ′*b multiset* ⇒ *bool* ⇒ ′*a multiset* (({#-/ | - :# -./ -#}))
syntax
  -comprehension-mset′ :: ′*a* ⇒ ′*b* ⇒ ′*b multiset* ⇒ *bool* ⇒ ′*a multiset* (({#-/ | - ∈# -./ -#}))
translations
  {#*e* | *x*∈#*M*. *P*#} ⇀ {#*e*. *x* ∈# {# *x*∈#*M*. *P*#}#}

This allows to write not just filters like {# *x* ∈# *M*. *x* < *c*#} but also images like {#*x* + *x*. *x* ∈# *M*#} and {#*x*+*x*|*x*∈#*M*. *x*<*c*#}, where the latter is currently displayed as {#*x* + *x*. *x* ∈# {# *x* ∈# *M*. *x* < *c*#}#}.

lemma *in-image-mset*: *y* ∈# {#*f x*. *x* ∈# *M*#} ⟷ *y* ∈ *f* ‘ *set-mset M*
by (*metis set-image-mset*)

functor *image-mset*: *image-mset*
proof −
  fix *f g* **show** *image-mset f* ∘ *image-mset g* = *image-mset* (*f* ∘ *g*)
  proof
    fix *A*
    show (*image-mset f* ∘ *image-mset g*) *A* = *image-mset* (*f* ∘ *g*) *A*
      by (*induct A*) *simp-all*

  qed
  show *image-mset id* = *id*
  proof
    fix *A*
    show *image-mset id A* = *id A*
      by (*induct A*) *simp-all*
  qed
qed

declare
  image-mset.id [*simp*]
  image-mset.identity [*simp*]

lemma *image-mset-id*[*simp*]: *image-mset id x* = *x*
  unfolding *id-def* **by** *auto*

lemma *image-mset-cong*: ($\bigwedge x.\ x \in\#\ M \Longrightarrow f\ x = g\ x$) $\Longrightarrow$ $\{\#f\ x.\ x \in\#\ M\#\}$ $= \{\#g\ x.\ x \in\#\ M\#\}$
  by (*induct M*) *auto*

lemma *image-mset-cong-pair*:
  $(\forall x\ y.\ (x, y) \in\#\ M \longrightarrow f\ x\ y = g\ x\ y) \Longrightarrow \{\#f\ x\ y.\ (x, y) \in\#\ M\#\} = \{\#g\ x\ y.\ (x, y) \in\#\ M\#\}$
  by (*metis image-mset-cong split-cong*)

65.7 Further conversions

primrec *mset* :: *'a list* ⇒ *'a multiset* **where**
  mset [] = {#} |
  mset (*a* # *x*) = *mset x* + {# *a* #}

lemma *in-multiset-in-set*:
  $x \in\#$ *mset xs* $\longleftrightarrow x \in$ *set xs*
  by (*induct xs*) *simp-all*

lemma *count-mset*:
  count (*mset xs*) *x* = *length* (*filter* ($\lambda y.\ x = y$) *xs*)
  by (*induct xs*) *simp-all*

lemma *mset-zero-iff* [*simp*]: (*mset x* = {#}) = (*x* = [])
  by (*induct x*) *auto*

lemma *mset-zero-iff-right*[*simp*]: ({#} = *mset x*) = (*x* = [])
by (*induct x*) *auto*

lemma *set-mset-mset*[*simp*]: *set-mset* (*mset x*) = *set x*
by (*induct x*) *auto*

lemma *set-mset-comp-mset* [*simp*]: *set-mset* ∘ *mset* = *set*

by (*simp add*: *fun-eq-iff*)

lemma *size-mset* [*simp*]: *size* (*mset xs*) = *length xs*
by (*induct xs*) *simp-all*

lemma *mset-append* [*simp*]: *mset* (*xs* @ *ys*) = *mset xs* + *mset ys*
by (*induct xs arbitrary*: *ys*) (*auto simp*: *ac-simps*)

lemma *mset-filter*: *mset* (*filter P xs*) = {#*x* ∈# *mset xs*. *P x* #}
by (*induct xs*) *simp-all*

lemma *mset-rev* [*simp*]:
mset (*rev xs*) = *mset xs*
by (*induct xs*) *simp-all*

lemma *surj-mset*: *surj mset*
apply (*unfold surj-def*)
apply (*rule allI*)
apply (*rule-tac M* = *y* **in** *multiset-induct*)
apply *auto*
apply (*rule-tac x* = *x* # *xa* **in** *exI*)
apply *auto*
done

lemma *distinct-count-atmost-1*:
distinct x = (∀ *a*. *count* (*mset x*) *a* = (*if a* ∈ *set x then 1 else 0*))
proof (*induct x*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons x xs*) **show** *?case* (**is** *?lhs* ⟷ *?rhs*)
proof
assume *?lhs* **then show** *?rhs* **using** *Cons* **by** *simp*
next
assume *?rhs* **then have** *x* ∉ *set xs*
by (*simp split*: *if-splits*)
moreover from ‹*?rhs*› **have** (∀ *a*. *count* (*mset xs*) *a* =
(*if a* ∈ *set xs then 1 else 0*))
by (*auto split*: *if-splits simp add*: *count-eq-zero-iff*)
ultimately show *?lhs* **using** *Cons* **by** *simp*
qed
qed

lemma *mset-eq-setD*:
assumes *mset xs* = *mset ys*
shows *set xs* = *set ys*
proof −
from *assms* **have** *set-mset* (*mset xs*) = *set-mset* (*mset ys*)
by *simp*
then show *?thesis* **by** *simp*

```
qed

lemma set-eq-iff-mset-eq-distinct:
  distinct x ⟹ distinct y ⟹
    (set x = set y) = (mset x = mset y)
by (auto simp: multiset-eq-iff distinct-count-atmost-1)

lemma set-eq-iff-mset-remdups-eq:
   (set x = set y) = (mset (remdups x) = mset (remdups y))
apply (rule iffI)
apply (simp add: set-eq-iff-mset-eq-distinct[THEN iffD1])
apply (drule distinct-remdups [THEN distinct-remdups
      [THEN set-eq-iff-mset-eq-distinct [THEN iffD2]]])
apply simp
done

lemma mset-compl-union [simp]: mset [x←xs. P x] + mset [x←xs. ¬P x] = mset
xs
  by (induct xs) (auto simp: ac-simps)

lemma nth-mem-mset: i < length ls ⟹ (ls ! i) ∈# mset ls
proof (induct ls arbitrary: i)
  case Nil
  then show ?case by simp
next
  case Cons
  then show ?case by (cases i) auto
qed

lemma mset-remove1[simp]: mset (remove1 a xs) = mset xs − {#a#}
  by (induct xs) (auto simp add: multiset-eq-iff)

lemma mset-eq-length:
  assumes mset xs = mset ys
  shows length xs = length ys
  using assms by (metis size-mset)

lemma mset-eq-length-filter:
  assumes mset xs = mset ys
  shows length (filter (λx. z = x) xs) = length (filter (λy. z = y) ys)
  using assms by (metis count-mset)

lemma fold-multiset-equiv:
  assumes f: ⋀x y. x ∈ set xs ⟹ y ∈ set xs ⟹ f x ∘ f y = f y ∘ f x
    and equiv: mset xs = mset ys
  shows List.fold f xs = List.fold f ys
  using f equiv [symmetric]
proof (induct xs arbitrary: ys)
  case Nil
```

then show *?case* **by** *simp*
next
case (*Cons x xs*)
then have ∗: *set ys* = *set* (*x* # *xs*)
by (*blast dest*: *mset-eq-setD*)
have $\bigwedge x\ y.\ x \in set\ ys \Longrightarrow y \in set\ ys \Longrightarrow f\ x \circ f\ y = f\ y \circ f\ x$
by (*rule Cons.prems*(*1*)) (*simp-all add*: ∗)
moreover from ∗ **have** $x \in set\ ys$
by *simp*
ultimately have *List.fold f ys* = *List.fold f* (*remove1 x ys*) ∘ *f x*
by (*fact fold-remove1-split*)
moreover from *Cons.prems* **have** *List.fold f xs* = *List.fold f* (*remove1 x ys*)
by (*auto intro*: *Cons.hyps*)
ultimately show *?case* **by** *simp*
qed

lemma *mset-insort* [*simp*]: *mset* (*insort x xs*) = *mset xs* + {#*x*#}
by (*induct xs*) (*simp-all add*: *ac-simps*)

lemma *mset-map*: *mset* (*map f xs*) = *image-mset f* (*mset xs*)
by (*induct xs*) *simp-all*

global-interpretation *mset-set*: *folding* λ*x M*. {#*x*#} + *M* {#}
defines *mset-set* = *folding.F* (λ*x M*. {#*x*#} + *M*) {#}
by *standard* (*simp add*: *fun-eq-iff ac-simps*)

lemma *count-mset-set* [*simp*]:
finite A ⟹ $x \in A$ ⟹ *count* (*mset-set A*) *x* = *1* (**is** *PROP ?P*)
¬ *finite A* ⟹ *count* (*mset-set A*) *x* = *0* (**is** *PROP ?Q*)
$x \notin A$ ⟹ *count* (*mset-set A*) *x* = *0* (**is** *PROP ?R*)
proof −
have ∗: *count* (*mset-set A*) *x* = *0* **if** $x \notin A$ **for** *A*
proof (*cases finite A*)
case *False* **then show** *?thesis* **by** *simp*
next
case *True* **from** *True* ‹$x \notin A$› **show** *?thesis* **by** (*induct A*) *auto*
qed
then show *PROP ?P PROP ?Q PROP ?R*
by (*auto elim*!: *Set.set-insert*)
qed — TODO: maybe define *mset-set* also in terms of *Abs-multiset*

lemma *elem-mset-set*[*simp*, *intro*]: *finite A* ⟹ *x* ∈# *mset-set A* ⟷ $x \in A$
by (*induct A rule*: *finite-induct*) *simp-all*

context *linorder*
begin

definition *sorted-list-of-multiset* :: ′*a multiset* ⇒ ′*a list*
where

sorted-list-of-multiset M = *fold-mset insort* [] *M*

lemma *sorted-list-of-multiset-empty* [*simp*]:
sorted-list-of-multiset {#} = []
by (*simp add*: *sorted-list-of-multiset-def*)

lemma *sorted-list-of-multiset-singleton* [*simp*]:
sorted-list-of-multiset {#*x*#} = [*x*]
proof −
interpret *comp-fun-commute insort* **by** (*fact comp-fun-commute-insort*)
show *?thesis* **by** (*simp add*: *sorted-list-of-multiset-def*)
qed

lemma *sorted-list-of-multiset-insert* [*simp*]:
sorted-list-of-multiset (*M* + {#*x*#}) = *List.insort x* (*sorted-list-of-multiset M*)
proof −
interpret *comp-fun-commute insort* **by** (*fact comp-fun-commute-insort*)
show *?thesis* **by** (*simp add*: *sorted-list-of-multiset-def*)
qed

end

lemma *mset-sorted-list-of-multiset* [*simp*]:
mset (*sorted-list-of-multiset M*) = *M*
by (*induct M*) *simp-all*

lemma *sorted-list-of-multiset-mset* [*simp*]:
sorted-list-of-multiset (*mset xs*) = *sort xs*
by (*induct xs*) *simp-all*

lemma *finite-set-mset-mset-set*[*simp*]:
finite A ⟹ *set-mset* (*mset-set A*) = *A*
by (*induct A rule*: *finite-induct*) *simp-all*

lemma *infinite-set-mset-mset-set*:
¬ *finite A* ⟹ *set-mset* (*mset-set A*) = {}
by *simp*

lemma *set-sorted-list-of-multiset* [*simp*]:
set (*sorted-list-of-multiset M*) = *set-mset M*
by (*induct M*) (*simp-all add*: *set-insort*)

lemma *sorted-list-of-mset-set* [*simp*]:
sorted-list-of-multiset (*mset-set A*) = *sorted-list-of-set A*
by (*cases finite A*) (*induct A rule*: *finite-induct*, *simp-all add*: *ac-simps*)

65.8 Replicate operation

definition *replicate-mset* :: *nat* ⇒ *'a* ⇒ *'a multiset* **where**

replicate-mset n x = ((*op* + {#*x*#}) ^^ *n*) {#}

lemma *replicate-mset-0*[*simp*]: *replicate-mset 0 x* = {#}
unfolding *replicate-mset-def* **by** *simp*

lemma *replicate-mset-Suc*[*simp*]: *replicate-mset* (*Suc n*) *x* = *replicate-mset n x* + {#*x*#}
unfolding *replicate-mset-def* **by** (*induct n*) (*auto intro*: *add.commute*)

lemma *in-replicate-mset*[*simp*]: *x* ∈# *replicate-mset n y* ⟷ *n* > *0* ∧ *x* = *y*
unfolding *replicate-mset-def* **by** (*induct n*) *auto*

lemma *count-replicate-mset*[*simp*]: *count* (*replicate-mset n x*) *y* = (*if y* = *x then n else 0*)
unfolding *replicate-mset-def* **by** (*induct n*) *simp-all*

lemma *set-mset-replicate-mset-subset*[*simp*]: *set-mset* (*replicate-mset n x*) = (*if n* = *0 then* {} *else* {*x*})
by (*auto split*: *if-splits*)

lemma *size-replicate-mset*[*simp*]: *size* (*replicate-mset n M*) = *n*
by (*induct n*, *simp-all*)

lemma *count-le-replicate-mset-le*: *n* ≤ *count M x* ⟷ *replicate-mset n x* ⊆# *M*
by (*auto simp add*: *assms mset-less-eqI*) (*metis count-replicate-mset subseteq-mset-def*)

lemma *filter-eq-replicate-mset*: {#*y* ∈# *D*. *y* = *x*#} = *replicate-mset* (*count D x*) *x*
by (*induct D*) *simp-all*

lemma *replicate-count-mset-eq-filter-eq*:
replicate (*count* (*mset xs*) *k*) *k* = *filter* (*HOL.eq k*) *xs*
by (*induct xs*) *auto*

lemma *replicate-mset-eq-empty-iff* [*simp*]:
replicate-mset n a = {#} ⟷ *n* = *0*
by (*induct n*) *simp-all*

lemma *replicate-mset-eq-iff*:
replicate-mset m a = *replicate-mset n b* ⟷
m = *0* ∧ *n* = *0* ∨ *m* = *n* ∧ *a* = *b*
by (*auto simp add*: *multiset-eq-iff*)

65.9 Big operators

no-notation *times* (**infixl** ∗ *70*)
no-notation *Groups.one* (*1*)

locale *comm-monoid-mset* = *comm-monoid*

begin

definition *F* :: *'a multiset* ⇒ *'a*
where *eq-fold*: *F M* = *fold-mset f 1 M*

lemma *empty* [*simp*]: *F* {#} = *1*
by (*simp add*: *eq-fold*)

lemma *singleton* [*simp*]: *F* {#*x*#} = *x*
proof −
interpret *comp-fun-commute*
by *standard* (*simp add*: *fun-eq-iff left-commute*)
show *?thesis* **by** (*simp add*: *eq-fold*)
qed

lemma *union* [*simp*]: *F* (*M* + *N*) = *F M* ∗ *F N*
proof −
interpret *comp-fun-commute f*
by *standard* (*simp add*: *fun-eq-iff left-commute*)
show *?thesis*
by (*induct N*) (*simp-all add*: *left-commute eq-fold*)
qed

end

lemma *comp-fun-commute-plus-mset*[*simp*]: *comp-fun-commute* (*op* + :: *'a multiset* ⇒ *-* ⇒ *-*)
by *standard* (*simp add*: *add-ac comp-def*)

declare *comp-fun-commute.fold-mset-insert*[*OF comp-fun-commute-plus-mset*, *simp*]

lemma *in-mset-fold-plus-iff*[*iff*]: *x* ∈# *fold-mset* (*op* +) *M NN* ⟷ *x* ∈# *M* ∨ (∃ *N*. *N* ∈# *NN* ∧ *x* ∈# *N*)
by (*induct NN*) *auto*

notation *times* (**infixl** ∗ *70*)
notation *Groups.one* (*1*)

context *comm-monoid-add*
begin

sublocale *msetsum*: *comm-monoid-mset plus 0*
defines *msetsum* = *msetsum.F* **..**

lemma (**in** *semiring-1*) *msetsum-replicate-mset* [*simp*]:
msetsum (*replicate-mset n a*) = *of-nat n* ∗ *a*
by (*induct n*) (*simp-all add*: *algebra-simps*)

lemma *setsum-unfold-msetsum*:

setsum f A = *msetsum* (*image-mset f* (*mset-set A*))
by (*cases finite A*) (*induct A rule*: *finite-induct*, *simp-all*)

end

lemma *msetsum-diff*:
fixes *M N* :: (*'a* :: *ordered-cancel-comm-monoid-diff*) *multiset*
shows *N* ⊆# *M* ⟹ *msetsum* (*M* − *N*) = *msetsum M* − *msetsum N*
by (*metis add-diff-cancel-right' msetsum.union subset-mset.diff-add*)

lemma *size-eq-msetsum*: *size M* = *msetsum* (*image-mset* (λ-. *1*) *M*)
proof (*induct M*)
case *empty* **then show** *?case* **by** *simp*
next
case (*add M x*) **then show** *?case*
by (*cases x* ∈ *set-mset M*)
(*simp-all add*: *size-multiset-overloaded-eq setsum.distrib setsum.delta' insert-absorb not-in-iff*)
qed

syntax (*ASCII*)
-msetsum-image :: *pttrn* ⇒ *'b set* ⇒ *'a* ⇒ *'a*::*comm-monoid-add* ((*3SUM* -:#-. -) [*0*, *51*, *10*] *10*)
syntax
-msetsum-image :: *pttrn* ⇒ *'b set* ⇒ *'a* ⇒ *'a*::*comm-monoid-add* ((*3*∑-∈#-. -) [*0*, *51*, *10*] *10*)
translations
∑*i* ∈# *A*. *b* ⇌ *CONST msetsum* (*CONST image-mset* (λ*i*. *b*) *A*)

abbreviation *Union-mset* :: *'a multiset multiset* ⇒ *'a multiset* (⋃#- [*900*] *900*)
where ⋃# *MM* ≡ *msetsum MM* — FIXME ambiguous notation – could likewise refer to ⨆#

lemma *set-mset-Union-mset*[*simp*]: *set-mset* (⋃# *MM*) = (⋃*M* ∈ *set-mset MM*. *set-mset M*)
by (*induct MM*) *auto*

lemma *in-Union-mset-iff* [*iff*]: *x* ∈# ⋃# *MM* ⟷ (∃*M*. *M* ∈# *MM* ∧ *x* ∈# *M*)
by (*induct MM*) *auto*

lemma *count-setsum*:
count (*setsum f A*) *x* = *setsum* (λ*a*. *count* (*f a*) *x*) *A*
by (*induct A rule*: *infinite-finite-induct*) *simp-all*

lemma *setsum-eq-empty-iff*:
assumes *finite A*
shows *setsum f A* = {#} ⟷ (∀ *a*∈*A*. *f a* = {#})
using *assms* **by** *induct simp-all*

context *comm-monoid-mult*
begin

sublocale *msetprod*: *comm-monoid-mset times 1*
defines *msetprod* = *msetprod.F* **..**

lemma *msetprod-empty*:
msetprod {#} = *1*
by (*fact msetprod.empty*)

lemma *msetprod-singleton*:
msetprod {#*x*#} = *x*
by (*fact msetprod.singleton*)

lemma *msetprod-Un*:
msetprod (*A* + *B*) = *msetprod A* ∗ *msetprod B*
by (*fact msetprod.union*)

lemma *msetprod-replicate-mset* [*simp*]:
msetprod (*replicate-mset n a*) = *a* ˆ *n*
by (*induct n*) (*simp-all add*: *ac-simps*)

lemma *setprod-unfold-msetprod*:
setprod f A = *msetprod* (*image-mset f* (*mset-set A*))
by (*cases finite A*) (*induct A rule*: *finite-induct*, *simp-all*)

lemma *msetprod-multiplicity*:
msetprod M = *setprod* (λ*x*. *x* ˆ *count M x*) (*set-mset M*)
by (*simp add*: *fold-mset-def setprod.eq-fold msetprod.eq-fold funpow-times-power comp-def*)

end

syntax (*ASCII*)
-msetprod-image :: *pttrn* ⇒ ′*b set* ⇒ ′*a* ⇒ ′*a*::*comm-monoid-mult* ((*3PROD* -:#-. -) [*0*, *51*, *10*] *10*)
syntax
-msetprod-image :: *pttrn* ⇒ ′*b set* ⇒ ′*a* ⇒ ′*a*::*comm-monoid-mult* ((*3*∏ -∈#-. -) [*0*, *51*, *10*] *10*)
translations
∏ *i* ∈# *A*. *b* ⇌ *CONST msetprod* (*CONST image-mset* (λ*i*. *b*) *A*)

lemma (**in** *comm-semiring-1*) *dvd-msetprod*:
assumes *x* ∈# *A*
shows *x dvd msetprod A*
proof −
from *assms* **have** *A* = (*A* − {#*x*#}) + {#*x*#} **by** *simp*
then obtain *B* **where** *A* = *B* + {#*x*#} **..**

then show *?thesis* **by** *simp*
qed

lemma (**in** *semidom*) *msetprod-zero-iff* [*iff*]:
msetprod $A = 0 \longleftrightarrow 0 \in\# A$
by (*induct A*) *auto*

lemma (**in** *semidom-divide*) *msetprod-diff*:
assumes $B \subseteq\# A$ **and** $0 \notin\# B$
shows *msetprod* $(A - B)$ = *msetprod A div msetprod B*
proof −
from *assms* **obtain** C **where** $A = B + C$
by (*metis subset-mset.add-diff-inverse*)
with *assms* **show** *?thesis* **by** *simp*
qed

lemma (**in** *semidom-divide*) *msetprod-minus*:
assumes $a \in\# A$ **and** $a \neq 0$
shows *msetprod* $(A - \{\#a\#\})$ = *msetprod A div a*
using *assms msetprod-diff* [*of* $\{\#a\#\}$ *A*]
by (*auto simp add*: *single-subset-iff*)

lemma (**in** *normalization-semidom*) *normalized-msetprodI*:
assumes $\bigwedge a.\ a \in\# A \Longrightarrow$ *normalize* $a = a$
shows *normalize* (*msetprod A*) = *msetprod A*
using *assms* **by** (*induct A*) (*simp-all add*: *normalize-mult*)

65.10 Alternative representations

65.10.1 Lists

context *linorder*
begin

lemma *mset-insort* [*simp*]:
mset (*insort-key k x xs*) = $\{\#x\#\}$ + *mset xs*
by (*induct xs*) (*simp-all add*: *ac-simps*)

lemma *mset-sort* [*simp*]:
mset (*sort-key k xs*) = *mset xs*
by (*induct xs*) (*simp-all add*: *ac-simps*)

This lemma shows which properties suffice to show that a function f with $f\ xs = ys$ behaves like sort.

lemma *properties-for-sort-key*:
assumes *mset ys* = *mset xs*
and $\bigwedge k.\ k \in$ *set ys* $\Longrightarrow$ *filter* $(\lambda x.\ f\ k = f\ x)$ *ys* = *filter* $(\lambda x.\ f\ k = f\ x)$ *xs*
and *sorted* (*map f ys*)
shows *sort-key f xs* = *ys*
using *assms*

proof (*induct xs arbitrary*: *ys*)
  case *Nil* **then show** *?case* **by** *simp*
next
  case (*Cons x xs*)
  from *Cons.prems*(*2*) **have**
    $\forall k \in$ *set ys*. *filter* (λx. *f k* = *f x*) (*remove1 x ys*) = *filter* (λx. *f k* = *f x*) *xs*
    by (*simp add*: *filter-remove1*)
  with *Cons.prems* **have** *sort-key f xs* = *remove1 x ys*
    by (*auto intro*!: *Cons.hyps simp add*: *sorted-map-remove1*)
  moreover from *Cons.prems* **have** *x* ∈# *mset ys*
    by *auto*
  then have *x* ∈ *set ys*
    by *simp*
  ultimately show *?case* **using** *Cons.prems* **by** (*simp add*: *insort-key-remove1*)
qed

lemma *properties-for-sort*:
  assumes *multiset*: *mset ys* = *mset xs*
    and *sorted ys*
  shows *sort xs* = *ys*
proof (*rule properties-for-sort-key*)
  from *multiset* **show** *mset ys* = *mset xs* **.**
  from ‹*sorted ys*› **show** *sorted* (*map* (λx. *x*) *ys*) **by** *simp*
  from *multiset* **have** *length* (*filter* (λy. *k* = *y*) *ys*) = *length* (*filter* (λx. *k* = *x*) *xs*) **for** *k*
    by (*rule mset-eq-length-filter*)
  then have *replicate* (*length* (*filter* (λy. *k* = *y*) *ys*)) *k* =
    replicate (*length* (*filter* (λx. *k* = *x*) *xs*)) *k* **for** *k*
    by *simp*
  then show *k* ∈ *set ys* ⟹ *filter* (λy. *k* = *y*) *ys* = *filter* (λx. *k* = *x*) *xs* **for** *k*
    by (*simp add*: *replicate-length-filter*)
qed

lemma *sort-key-inj-key-eq*:
  assumes *mset-equal*: *mset xs* = *mset ys*
    and *inj-on f* (*set xs*)
    and *sorted* (*map f ys*)
  shows *sort-key f xs* = *ys*
proof (*rule properties-for-sort-key*)
  from *mset-equal*
  show *mset ys* = *mset xs* **by** *simp*
  from ‹*sorted* (*map f ys*)›
  show *sorted* (*map f ys*) **.**
  show [*x*←*ys* . *f k* = *f x*] = [*x*←*xs* . *f k* = *f x*] **if** *k* ∈ *set ys* **for** *k*
  proof −
    from *mset-equal*
    have *set-equal*: *set xs* = *set ys* **by** (*rule mset-eq-setD*)
    with *that* **have** *insert k* (*set ys*) = *set ys* **by** *auto*
    with ‹*inj-on f* (*set xs*)› **have** *inj*: *inj-on f* (*insert k* (*set ys*))

```
      by (simp add: set-equal)
    from inj have [x←ys . f k = f x] = filter (HOL.eq k) ys
      by (auto intro!: inj-on-filter-key-eq)
    also have ... = replicate (count (mset ys) k) k
      by (simp add: replicate-count-mset-eq-filter-eq)
    also have ... = replicate (count (mset xs) k) k
      using mset-equal by simp
    also have ... = filter (HOL.eq k) xs
      by (simp add: replicate-count-mset-eq-filter-eq)
    also have ... = [x←xs . f k = f x]
     using inj by (auto intro!: inj-on-filter-key-eq [symmetric] simp add: set-equal)
    finally show ?thesis .
  qed
qed

lemma sort-key-eq-sort-key:
  assumes mset xs = mset ys
    and inj-on f (set xs)
  shows sort-key f xs = sort-key f ys
  by (rule sort-key-inj-key-eq) (simp-all add: assms)

lemma sort-key-by-quicksort:
  sort-key f xs = sort-key f [x←xs. f x < f (xs ! (length xs div 2))]
    @ [x←xs. f x = f (xs ! (length xs div 2))]
    @ sort-key f [x←xs. f x > f (xs ! (length xs div 2))] (is sort-key f ?lhs = ?rhs)
proof (rule properties-for-sort-key)
  show mset ?rhs = mset ?lhs
    by (rule multiset-eqI) (auto simp add: mset-filter)
  show sorted (map f ?rhs)
    by (auto simp add: sorted-append intro: sorted-map-same)
next
  fix l
  assume l ∈ set ?rhs
  let ?pivot = f (xs ! (length xs div 2))
  have *: ⋀x. f l = f x ⟷ f x = f l by auto
  have [x ← sort-key f xs . f x = f l] = [x ← xs. f x = f l]
   unfolding filter-sort by (rule properties-for-sort-key) (auto intro: sorted-map-same)
  with * have **: [x ← sort-key f xs . f l = f x] = [x ← xs. f l = f x] by simp
  have ⋀x P. P (f x) ?pivot ∧ f l = f x ⟷ P (f l) ?pivot ∧ f l = f x by auto
  then have ⋀P. [x ← sort-key f xs . P (f x) ?pivot ∧ f l = f x] =
    [x ← sort-key f xs. P (f l) ?pivot ∧ f l = f x] by simp
  note *** = this [of op <] this [of op >] this [of op =]
  show [x ← ?rhs. f l = f x] = [x ← ?lhs. f l = f x]
  proof (cases f l ?pivot rule: linorder-cases)
    case less
    then have f l ≠ ?pivot and ¬ f l > ?pivot by auto
    with less show ?thesis
      by (simp add: filter-sort [symmetric] ** ***)
  next
```

case *equal* **then show** *?thesis*
by (*simp add*: ∗ *less-le*)
next
case *greater*
then have *f l* ≠ *?pivot* **and** ¬ *f l* < *?pivot* **by** *auto*
with *greater* **show** *?thesis*
by (*simp add*: *filter-sort* [*symmetric*] ∗∗ ∗∗∗)
qed
qed

lemma *sort-by-quicksort*:
sort xs = *sort* [*x*←*xs*. *x* < *xs* ! (*length xs div 2*)]
@ [*x*←*xs*. *x* = *xs* ! (*length xs div 2*)]
@ *sort* [*x*←*xs*. *x* > *xs* ! (*length xs div 2*)] (**is** *sort ?lhs* = *?rhs*)
using *sort-key-by-quicksort* [*of* λ*x*. *x*, *symmetric*] **by** *simp*

A stable parametrized quicksort

definition *part* :: ('*b* ⇒ '*a*) ⇒ '*a* ⇒ '*b list* ⇒ '*b list* × '*b list* × '*b list* **where**
part f pivot xs = ([*x* ← *xs*. *f x* < *pivot*], [*x* ← *xs*. *f x* = *pivot*], [*x* ← *xs*. *pivot* < *f x*])

lemma *part-code* [*code*]:
part f pivot [] = ([], [], [])
part f pivot (*x* # *xs*) = (*let* (*lts*, *eqs*, *gts*) = *part f pivot xs*; *x'* = *f x in*
if x' < *pivot then* (*x* # *lts*, *eqs*, *gts*)
else if x' > *pivot then* (*lts*, *eqs*, *x* # *gts*)
else (*lts*, *x* # *eqs*, *gts*))
by (*auto simp add*: *part-def Let-def split-def*)

lemma *sort-key-by-quicksort-code* [*code*]:
sort-key f xs =
(*case xs of*
[] ⇒ []
| [*x*] ⇒ *xs*
| [*x*, *y*] ⇒ (*if f x* ≤ *f y then xs else* [*y*, *x*])
| - ⇒
let (*lts*, *eqs*, *gts*) = *part f* (*f* (*xs* ! (*length xs div 2*))) *xs*
in sort-key f lts @ *eqs* @ *sort-key f gts*)
proof (*cases xs*)
case *Nil* **then show** *?thesis* **by** *simp*
next
case (*Cons* - *ys*) **note** *hyps* = *Cons* **show** *?thesis*
proof (*cases ys*)
case *Nil* **with** *hyps* **show** *?thesis* **by** *simp*
next
case (*Cons* - *zs*) **note** *hyps* = *hyps Cons* **show** *?thesis*
proof (*cases zs*)
case *Nil* **with** *hyps* **show** *?thesis* **by** *auto*
next

```
      case Cons
      from sort-key-by-quicksort [of f xs]
      have sort-key f xs = (let (lts, eqs, gts) = part f (f (xs ! (length xs div 2))) xs
        in sort-key f lts @ eqs @ sort-key f gts)
      by (simp only: split-def Let-def part-def fst-conv snd-conv)
      with hyps Cons show ?thesis by (simp only: list.cases)
    qed
  qed
qed

end

hide-const (open) part

lemma mset-remdups-le: mset (remdups xs) ⊆# mset xs
  by (induct xs) (auto intro: subset-mset.order-trans)

lemma mset-update:
  i < length ls ⟹ mset (ls[i := v]) = mset ls − {#ls ! i#} + {#v#}
proof (induct ls arbitrary: i)
  case Nil then show ?case by simp
next
  case (Cons x xs)
  show ?case
  proof (cases i)
    case 0 then show ?thesis by simp
  next
    case (Suc i')
    with Cons show ?thesis
      apply simp
      apply (subst add.assoc)
      apply (subst add.commute [of {#v#} {#x#}])
      apply (subst add.assoc [symmetric])
      apply simp
      apply (rule mset-le-multiset-union-diff-commute)
      apply (simp add: mset-le-single nth-mem-mset)
      done
  qed
qed

lemma mset-swap:
  i < length ls ⟹ j < length ls ⟹
    mset (ls[j := ls ! i, i := ls ! j]) = mset ls
  by (cases i = j) (simp-all add: mset-update nth-mem-mset)
```

65.11 The multiset order

65.11.1 Well-foundedness

definition *mult1* :: ('a × 'a) set ⇒ ('a multiset × 'a multiset) set **where**

mult1 r = {(*N*, *M*). ∃ *a M0 K*. *M* = *M0* + {#*a*#} ∧ *N* = *M0* + *K* ∧
(∀ *b*. *b* ∈# *K* ⟶ (*b*, *a*) ∈ *r*)}

definition *mult* :: ('*a* × '*a*) *set* ⇒ ('*a multiset* × '*a multiset*) *set* **where**
mult r = (*mult1 r*)$^+$

lemma *mult1I*:
assumes *M* = *M0* + {#*a*#} **and** *N* = *M0* + *K* **and** ⋀*b*. *b* ∈# *K* ⟹ (*b*, *a*) ∈ *r*
shows (*N*, *M*) ∈ *mult1 r*
using *assms* **unfolding** *mult1-def* **by** *blast*

lemma *mult1E*:
assumes (*N*, *M*) ∈ *mult1 r*
obtains *a M0 K* **where** *M* = *M0* + {#*a*#} *N* = *M0* + *K* ⋀*b*. *b* ∈# *K* ⟹ (*b*, *a*) ∈ *r*
using *assms* **unfolding** *mult1-def* **by** *blast*

lemma *not-less-empty* [*iff*]: (*M*, {#}) ∉ *mult1 r*
by (*simp add*: *mult1-def*)

lemma *less-add*:
assumes *mult1*: (*N*, *M0* + {#*a*#}) ∈ *mult1 r*
shows
(∃ *M*. (*M*, *M0*) ∈ *mult1 r* ∧ *N* = *M* + {#*a*#}) ∨
(∃ *K*. (∀ *b*. *b* ∈# *K* ⟶ (*b*, *a*) ∈ *r*) ∧ *N* = *M0* + *K*)
proof −
let *?r* = λ*K a*. ∀ *b*. *b* ∈# *K* ⟶ (*b*, *a*) ∈ *r*
let *?R* = λ*N M*. ∃ *a M0 K*. *M* = *M0* + {#*a*#} ∧ *N* = *M0* + *K* ∧ *?r K a*
obtain *a' M0' K* **where** *M0*: *M0* + {#*a*#} = *M0'* + {#*a'*#}
and *N*: *N* = *M0'* + *K*
and *r*: *?r K a'*
using *mult1* **unfolding** *mult1-def* **by** *auto*
show *?thesis* (**is** *?case1* ∨ *?case2*)
proof −
from *M0* **consider** *M0* = *M0'* *a* = *a'*
| *K'* **where** *M0* = *K'* + {#*a'*#} *M0'* = *K'* + {#*a*#}
by *atomize-elim* (*simp only*: *add-eq-conv-ex*)
then show *?thesis*
proof *cases*
case *1*
with *N r* **have** *?r K a* ∧ *N* = *M0* + *K* **by** *simp*
then have *?case2* **..**
then show *?thesis* **..**
next
case *2*
from *N 2*(*2*) **have** *n*: *N* = *K'* + *K* + {#*a*#} **by** (*simp add*: *ac-simps*)
with *r 2*(*1*) **have** *?R* (*K'* + *K*) *M0* **by** *blast*
with *n* **have** *?case1* **by** (*simp add*: *mult1-def*)

```
      then show ?thesis ..
    qed
  qed
qed

lemma all-accessible:
  assumes wf r
  shows ∀ M. M ∈ Wellfounded.acc (mult1 r)
proof
  let ?R = mult1 r
  let ?W = Wellfounded.acc ?R
  {
    fix M M0 a
    assume M0: M0 ∈ ?W
      and wf-hyp: ⋀b. (b, a) ∈ r ⟹ (∀ M ∈ ?W. M + {#b#} ∈ ?W)
      and acc-hyp: ∀ M. (M, M0) ∈ ?R ⟶ M + {#a#} ∈ ?W
    have M0 + {#a#} ∈ ?W
    proof (rule accI [of M0 + {#a#}])
      fix N
      assume (N, M0 + {#a#}) ∈ ?R
      then consider M where (M, M0) ∈ ?R N = M + {#a#}
        | K where ∀ b. b ∈# K ⟶ (b, a) ∈ r N = M0 + K
        by atomize-elim (rule less-add)
      then show N ∈ ?W
      proof cases
        case 1
        from acc-hyp have (M, M0) ∈ ?R ⟶ M + {#a#} ∈ ?W ..
        from this and ‹(M, M0) ∈ ?R› have M + {#a#} ∈ ?W ..
        then show N ∈ ?W by (simp only: ‹N = M + {#a#}›)
      next
        case 2
        from this(1) have M0 + K ∈ ?W
        proof (induct K)
          case empty
          from M0 show M0 + {#} ∈ ?W by simp
        next
          case (add K x)
          from add.prems have (x, a) ∈ r by simp
          with wf-hyp have ∀ M ∈ ?W. M + {#x#} ∈ ?W by blast
          moreover from add have M0 + K ∈ ?W by simp
          ultimately have (M0 + K) + {#x#} ∈ ?W ..
          then show M0 + (K + {#x#}) ∈ ?W by (simp only: add.assoc)
        qed
        then show N ∈ ?W by (simp only: 2(2))
      qed
    qed
  } note tedious-reasoning = this

  show M ∈ ?W for M
```

proof (*induct M*)
show {#} ∈ *?W*
proof (*rule accI*)
fix *b* **assume** (*b*, {#}) ∈ *?R*
with *not-less-empty* **show** *b* ∈ *?W* **by** *contradiction*
qed

fix *M a* **assume** *M* ∈ *?W*
from ‹*wf r*› **have** ∀ *M* ∈ *?W*. *M* + {#*a*#} ∈ *?W*
proof *induct*
fix *a*
assume *r*: ⋀*b*. (*b*, *a*) ∈ *r* ⟹ (∀ *M* ∈ *?W*. *M* + {#*b*#} ∈ *?W*)
show ∀ *M* ∈ *?W*. *M* + {#*a*#} ∈ *?W*
proof
fix *M* **assume** *M* ∈ *?W*
then show *M* + {#*a*#} ∈ *?W*
by (*rule acc-induct*) (*rule tedious-reasoning* [*OF* - *r*])
qed
qed
from *this* **and** ‹*M* ∈ *?W*› **show** *M* + {#*a*#} ∈ *?W* **..**
qed
qed

theorem *wf-mult1*: *wf r* ⟹ *wf* (*mult1 r*)
by (*rule acc-wfI*) (*rule all-accessible*)

theorem *wf-mult*: *wf r* ⟹ *wf* (*mult r*)
unfolding *mult-def* **by** (*rule wf-trancl*) (*rule wf-mult1*)

65.11.2 Closure-free presentation

One direction.

lemma *mult-implies-one-step*:
trans r ⟹ (*M*, *N*) ∈ *mult r* ⟹
∃ *I J K*. *N* = *I* + *J* ∧ *M* = *I* + *K* ∧ *J* ≠ {#} ∧
(∀ *k* ∈ *set-mset K*. ∃ *j* ∈ *set-mset J*. (*k*, *j*) ∈ *r*)
apply (*unfold mult-def mult1-def*)
apply (*erule converse-trancl-induct*, *clarify*)
apply (*rule-tac x* = *M0* **in** *exI*, *simp*, *clarify*)
apply (*case-tac a* ∈# *K*)
apply (*rule-tac x* = *I* **in** *exI*)
apply (*simp* (*no-asm*))
apply (*rule-tac x* = (*K* − {#*a*#}) + *Ka* **in** *exI*)
apply (*simp* (*no-asm-simp*) *add*: *add.assoc* [*symmetric*])
apply (*drule-tac f* = λ*M*. *M* − {#*a*#} **and** *x*=*S* + *T* **for** *S T* **in** *arg-cong*)
apply (*simp add*: *diff-union-single-conv*)
apply (*simp* (*no-asm-use*) *add*: *trans-def*)
apply (*metis* (*no-types*, *hide-lams*) *Multiset.diff-right-commute Un-iff diff-single-trivial multi-drop-mem-not-eq*)

apply (*subgoal-tac* $a \in\# I$)
 apply (*rule-tac* $x = I - \{\#a\#\}$ **in** *exI*)
 apply (*rule-tac* $x = J + \{\#a\#\}$ **in** *exI*)
 apply (*rule-tac* $x = K + Ka$ **in** *exI*)
 apply (*rule conjI*)
  apply (*simp add*: *multiset-eq-iff split*: *nat-diff-split*)
 apply (*rule conjI*)
  apply (*drule-tac* $f = \lambda M.\ M - \{\#a\#\}$ **and** $x = S + T$ **for** $S\ T$ **in** *arg-cong*, *simp*)
  apply (*simp add*: *multiset-eq-iff split*: *nat-diff-split*)
 apply (*simp* (*no-asm-use*) *add*: *trans-def*)
apply (*subgoal-tac* $a \in\# (M0 + \{\#a\#\})$)
 apply (*simp-all add*: *not-in-iff*)
 apply *blast*
 apply (*metis add.comm-neutral add-diff-cancel-right′ count-eq-zero-iff diff-single-trivial multi-self-add-other-not-self plus-multiset.rep-eq*)
done

lemma *one-step-implies-mult-aux*:
 $\forall I\ J\ K.\ size\ J = n \wedge J \neq \{\#\} \wedge (\forall k \in set\text{-}mset\ K.\ \exists j \in set\text{-}mset\ J.\ (k, j) \in r)$
  $\longrightarrow (I + K, I + J) \in mult\ r$
apply (*induct n*)
 apply *auto*
apply (*frule size-eq-Suc-imp-eq-union*, *clarify*)
apply (*rename-tac* J', *simp*)
apply (*erule notE*, *auto*)
apply (*case-tac* $J' = \{\#\}$)
 apply (*simp add*: *mult-def*)
 apply (*rule r-into-trancl*)
 apply (*simp add*: *mult1-def*, *blast*)

Now we know $J' \neq \{\#\}$.

apply (*cut-tac* $M = K$ **and** $P = \lambda x.\ (x, a) \in r$ **in** *multiset-partition*)
apply (*erule-tac* $P = \forall k \in set\text{-}mset\ K.\ P\ k$ **for** P **in** *rev-mp*)
apply (*erule ssubst*)
apply (*simp add*: *Ball-def*, *auto*)
apply (*subgoal-tac*
 $((I + \{\#\ x \in\# K.\ (x, a) \in r\ \#\}) + \{\#\ x \in\# K.\ (x, a) \notin r\ \#\},$
  $(I + \{\#\ x \in\# K.\ (x, a) \in r\ \#\}) + J') \in mult\ r$)
 prefer *2*
 apply *force*
apply (*simp* (*no-asm-use*) *add*: *add.assoc* [*symmetric*] *mult-def*)
apply (*erule trancl-trans*)
apply (*rule r-into-trancl*)
apply (*simp add*: *mult1-def*)
apply (*rule-tac* $x = a$ **in** *exI*)
apply (*rule-tac* $x = I + J'$ **in** *exI*)
apply (*simp add*: *ac-simps*)
done

lemma *one-step-implies-mult*:
  $J \neq \{\#\} \Longrightarrow \forall k \in \textit{set-mset } K.\ \exists j \in \textit{set-mset } J.\ (k,\ j) \in r$
    $\Longrightarrow (I + K,\ I + J) \in \textit{mult } r$
using *one-step-implies-mult-aux* **by** *blast*

65.11.3 Partial-order properties

lemma (**in** *order*) *mult1-lessE*:
  assumes $(N,\ M) \in \textit{mult1 } \{(a,\ b).\ a < b\}$
  obtains $a\ M0\ K$ **where** $M = M0 + \{\#a\#\}\ N = M0 + K$
    $a \notin\#\ K\ \bigwedge b.\ b \in\#\ K \Longrightarrow b < a$
proof −
  from *assms* **obtain** $a\ M0\ K$ **where** $M = M0 + \{\#a\#\}\ N = M0 + K$
    $\bigwedge b.\ b \in\#\ K \Longrightarrow b < a$ **by** (*blast elim*: *mult1E*)
  moreover from *this*(*3*) [*of a*] **have** $a \notin\#\ K$ **by** *auto*
  ultimately show *thesis* **by** (*auto intro*: *that*)
qed

definition *less-multiset* :: *'a::order multiset ⇒ 'a multiset ⇒ bool* (**infix** #⊂# *50*)
  where $M' \#\subset\# M \longleftrightarrow (M',\ M) \in \textit{mult } \{(x',\ x).\ x' < x\}$

definition *le-multiset* :: *'a::order multiset ⇒ 'a multiset ⇒ bool* (**infix** #⊆# *50*)
  where $M' \#\subseteq\# M \longleftrightarrow M' \#\subset\# M \vee M' = M$

notation (*ASCII*)
  less-multiset (**infix** #<# *50*) **and**
  le-multiset (**infix** #<=# *50*)

interpretation *multiset-order*: *order le-multiset less-multiset*
proof −
  have *irrefl*: $\neg\ M \#\subset\# M$ **for** *M* :: *'a multiset*
  proof
    assume $M \#\subset\# M$
    then have *MM*: $(M,\ M) \in \textit{mult } \{(x,\ y).\ x < y\}$ **by** (*simp add*: *less-multiset-def*)
    have *trans* $\{(x'::'a,\ x).\ x' < x\}$
      by (*rule transI*) *simp*
    moreover note *MM*
    ultimately have $\exists I\ J\ K.\ M = I + J \wedge M = I + K$
      $\wedge\ J \neq \{\#\} \wedge (\forall k{\in}\textit{set-mset } K.\ \exists j{\in}\textit{set-mset } J.\ (k,\ j) \in \{(x,\ y).\ x < y\})$
      by (*rule mult-implies-one-step*)
    then obtain *I J K* **where** $M = I + J$ **and** $M = I + K$
      and $J \neq \{\#\}$ **and** $(\forall k{\in}\textit{set-mset } K.\ \exists j{\in}\textit{set-mset } J.\ (k,\ j) \in \{(x,\ y).\ x < y\})$ **by** *blast*
    then have $*$: $K \neq \{\#\}$ **and** $**$: $\forall k{\in}\textit{set-mset } K.\ \exists j{\in}\textit{set-mset } K.\ k < j$ **by** *auto*
    have *finite* (*set-mset K*) **by** *simp*
    moreover note $**$

ultimately have *set-mset K* = {}
by (*induct rule*: *finite-induct*) (*auto intro*: *order-less-trans*)
with ∗ **show** *False* **by** *simp*
qed
have *trans*: *K* #⊂# *M* ⟹ *M* #⊂# *N* ⟹ *K* #⊂# *N* **for** *K M N* :: *'a multiset*
unfolding *less-multiset-def mult-def* **by** (*blast intro*: *trancl-trans*)
show *class.order* (*le-multiset* :: *'a multiset* ⇒ -) *less-multiset*
by *standard* (*auto simp add*: *le-multiset-def irrefl dest*: *trans*)
qed — FIXME avoid junk stemming from type class interpretation

lemma *mult-less-irrefl* [*elim*!]:
fixes *M* :: *'a*::*order multiset*
shows *M* #⊂# *M* ⟹ *R*
by *simp*

65.11.4 Monotonicity of multiset union

lemma *mult1-union*: (*B*, *D*) ∈ *mult1 r* ⟹ (*C* + *B*, *C* + *D*) ∈ *mult1 r*
apply (*unfold mult1-def*)
apply *auto*
apply (*rule-tac x* = *a* **in** *exI*)
apply (*rule-tac x* = *C* + *M0* **in** *exI*)
apply (*simp add*: *add.assoc*)
done

lemma *union-less-mono2*: *B* #⊂# *D* ⟹ *C* + *B* #⊂# *C* + (*D*::*'a*::*order multiset*)
apply (*unfold less-multiset-def mult-def*)
apply (*erule trancl-induct*)
apply (*blast intro*: *mult1-union*)
apply (*blast intro*: *mult1-union trancl-trans*)
done

lemma *union-less-mono1*: *B* #⊂# *D* ⟹ *B* + *C* #⊂# *D* + (*C*::*'a*::*order multiset*)
apply (*subst add.commute* [*of B C*])
apply (*subst add.commute* [*of D C*])
apply (*erule union-less-mono2*)
done

lemma *union-less-mono*:
fixes *A B C D* :: *'a*::*order multiset*
shows *A* #⊂# *C* ⟹ *B* #⊂# *D* ⟹ *A* + *B* #⊂# *C* + *D*
by (*blast intro*!: *union-less-mono1 union-less-mono2 multiset-order.less-trans*)

interpretation *multiset-order*: *ordered-ab-semigroup-add plus le-multiset less-multiset*
by *standard* (*auto simp add*: *le-multiset-def intro*: *union-less-mono2*)

65.11.5 Termination proofs with multiset orders

lemma *multi-member-skip*: $x \in\# XS \Longrightarrow x \in\# \{\# y \#\} + XS$
and *multi-member-this*: $x \in\# \{\# x \#\} + XS$
and *multi-member-last*: $x \in\# \{\# x \#\}$
by *auto*

definition *ms-strict* = *mult pair-less*
definition *ms-weak* = *ms-strict* ∪ *Id*

lemma *ms-reduction-pair*: *reduction-pair* (*ms-strict*, *ms-weak*)
unfolding *reduction-pair-def ms-strict-def ms-weak-def pair-less-def*
by (*auto intro*: *wf-mult1 wf-trancl simp*: *mult-def*)

lemma *smsI*:
(*set-mset A*, *set-mset B*) ∈ *max-strict* ⟹ (Z + A, Z + B) ∈ *ms-strict*
unfolding *ms-strict-def*
by (*rule one-step-implies-mult*) (*auto simp add*: *max-strict-def pair-less-def elim*!:*max-ext.cases*)

lemma *wmsI*:
(*set-mset A*, *set-mset B*) ∈ *max-strict* ∨ $A = \{\#\} \wedge B = \{\#\}$
⟹ (Z + A, Z + B) ∈ *ms-weak*
unfolding *ms-weak-def ms-strict-def*
by (*auto simp add*: *pair-less-def max-strict-def elim*!:*max-ext.cases intro*: *one-step-implies-mult*)

inductive *pw-leq*
where
pw-leq-empty: *pw-leq* {#} {#}
| *pw-leq-step*: ⟦(x,y) ∈ *pair-leq*; *pw-leq X Y* ⟧ ⟹ *pw-leq* ({#x#} + X) ({#y#} + Y)

lemma *pw-leq-lstep*:
(x, y) ∈ *pair-leq* ⟹ *pw-leq* {#x#} {#y#}
by (*drule pw-leq-step*) (*rule pw-leq-empty*, *simp*)

lemma *pw-leq-split*:
assumes *pw-leq X Y*
shows ∃ A B Z. $X = A + Z \wedge Y = B + Z \wedge$ ((*set-mset A*, *set-mset B*) ∈ *max-strict* ∨ ($B = \{\#\} \wedge A = \{\#\}$))
using *assms*
proof *induct*
case *pw-leq-empty* **thus** *?case* **by** *auto*
next
case (*pw-leq-step x y X Y*)
then obtain A B Z **where**
[*simp*]: $X = A + Z$ $Y = B + Z$
and *1*[*simp*]: (*set-mset A*, *set-mset B*) ∈ *max-strict* ∨ ($B = \{\#\} \wedge A = \{\#\}$)
by *auto*
from *pw-leq-step* **consider** $x = y$ | (x, y) ∈ *pair-less*
unfolding *pair-leq-def* **by** *auto*

thus *?case*
proof *cases*
case [*simp*]: *1*
have $\{\#x\#\} + X = A + (\{\#y\#\}+Z) \wedge \{\#y\#\} + Y = B + (\{\#y\#\}+Z) \wedge$
$((\textit{set-mset } A, \textit{set-mset } B) \in \textit{max-strict} \vee (B = \{\#\} \wedge A = \{\#\}))$
by (*auto simp*: *ac-simps*)
thus *?thesis* **by** *blast*
next
case *2*
let $?A' = \{\#x\#\} + A$ **and** $?B' = \{\#y\#\} + B$
have $\{\#x\#\} + X = ?A' + Z$
$\{\#y\#\} + Y = ?B' + Z$
by (*auto simp add*: *ac-simps*)
moreover have
$(\textit{set-mset } ?A', \textit{set-mset } ?B') \in \textit{max-strict}$
using *1 2* **unfolding** *max-strict-def*
by (*auto elim*!: *max-ext.cases*)
ultimately show *?thesis* **by** *blast*
qed
qed

lemma
assumes *pwleq*: *pw-leq* Z Z'
shows *ms-strictI*: $(\textit{set-mset } A, \textit{set-mset } B) \in \textit{max-strict} \Longrightarrow (Z + A, Z' + B)$
$\in$ *ms-strict*
and *ms-weakI1*: $(\textit{set-mset } A, \textit{set-mset } B) \in \textit{max-strict} \Longrightarrow (Z + A, Z' + B)$
$\in$ *ms-weak*
and *ms-weakI2*: $(Z + \{\#\}, Z' + \{\#\}) \in \textit{ms-weak}$
proof –
from *pw-leq-split*[*OF pwleq*]
obtain A' B' Z''
where [*simp*]: $Z = A' + Z''$ $Z' = B' + Z''$
and *mx-or-empty*: $(\textit{set-mset } A', \textit{set-mset } B') \in \textit{max-strict} \vee (A' = \{\#\} \wedge B'$
$= \{\#\})$
by *blast*
{
assume *max*: $(\textit{set-mset } A, \textit{set-mset } B) \in \textit{max-strict}$
from *mx-or-empty*
have $(Z'' + (A + A'), Z'' + (B + B')) \in \textit{ms-strict}$
proof
assume *max'*: $(\textit{set-mset } A', \textit{set-mset } B') \in \textit{max-strict}$
with *max* **have** $(\textit{set-mset } (A + A'), \textit{set-mset } (B + B')) \in \textit{max-strict}$
by (*auto simp*: *max-strict-def intro*: *max-ext-additive*)
thus *?thesis* **by** (*rule smsI*)
next
assume [*simp*]: $A' = \{\#\} \wedge B' = \{\#\}$
show *?thesis* **by** (*rule smsI*) (*auto intro*: *max*)
qed
thus $(Z + A, Z' + B) \in \textit{ms-strict}$ **by** (*simp add*: *ac-simps*)

thus $(Z + A, Z' + B) \in$ *ms-weak* **by** (*simp add*: *ms-weak-def*)
}
from *mx-or-empty*
have $(Z'' + A', Z'' + B') \in$ *ms-weak* **by** (*rule wmsI*)
thus $(Z + \{\#\}, Z' + \{\#\}) \in$ *ms-weak* **by** (*simp add*:*ac-simps*)
qed

lemma *empty-neutral*: $\{\#\} + x = x\ x + \{\#\} = x$
and *nonempty-plus*: $\{\#\ x\ \#\} + rs \neq \{\#\}$
and *nonempty-single*: $\{\#\ x\ \#\} \neq \{\#\}$
by *auto*

setup ‹

```
let
  fun msetT T = Type (@{type-name multiset}, [T]);

  fun mk-mset T [] = Const (@{const-abbrev Mempty}, msetT T)
    | mk-mset T [x] = Const (@{const-name single}, T --> msetT T) $ x
    | mk-mset T (x :: xs) =
          Const (@{const-name plus}, msetT T --> msetT T --> msetT T) $
                mk-mset T [x] $ mk-mset T xs

  fun mset-member-tac ctxt m i =
    if m <= 0 then
      resolve-tac ctxt @{thms multi-member-this} i ORELSE
      resolve-tac ctxt @{thms multi-member-last} i
    else
      resolve-tac ctxt @{thms multi-member-skip} i THEN mset-member-tac ctxt
(m - 1) i

  fun mset-nonempty-tac ctxt =
    resolve-tac ctxt @{thms nonempty-plus} ORELSE'
    resolve-tac ctxt @{thms nonempty-single}

  fun regroup-munion-conv ctxt =
    Function-Lib.regroup-conv ctxt @{const-abbrev Mempty} @{const-name plus}
    (map (fn t => t RS eq-reflection) (@{thms ac-simps} @ @{thms empty-neutral}))

  fun unfold-pwleq-tac ctxt i =
    (resolve-tac ctxt @{thms pw-leq-step} i THEN (fn st => unfold-pwleq-tac ctxt
(i + 1) st))
      ORELSE (resolve-tac ctxt @{thms pw-leq-lstep} i)
      ORELSE (resolve-tac ctxt @{thms pw-leq-empty} i)

  val set-mset-simps = [@{thm set-mset-empty}, @{thm set-mset-single}, @{thm
set-mset-union},
                        @{thm Un-insert-left}, @{thm Un-empty-left}]
in
  ScnpReconstruct.multiset-setup (ScnpReconstruct.Multiset
```

{
msetT=msetT, *mk-mset=mk-mset*, *mset-regroup-conv=regroup-munion-conv*,
mset-member-tac=mset-member-tac, *mset-nonempty-tac=mset-nonempty-tac*,
mset-pwleq-tac=unfold-pwleq-tac, *set-of-simps=set-mset-simps*,
smsI'= @{*thm ms-strictI*}, *wmsI2''*= @{*thm ms-weakI2*}, *wmsI1*= @{*thm ms-weakI1*},
reduction-pair = @{*thm ms-reduction-pair*}
})
end
⟩

65.12 Legacy theorem bindings

lemmas *multi-count-eq* = *multiset-eq-iff* [*symmetric*]

lemma *union-commute*: $M + N = N + (M::'a\ multiset)$
by (*fact add.commute*)

lemma *union-assoc*: $(M + N) + K = M + (N + (K::'a\ multiset))$
by (*fact add.assoc*)

lemma *union-lcomm*: $M + (N + K) = N + (M + (K::'a\ multiset))$
by (*fact add.left-commute*)

lemmas *union-ac* = *union-assoc union-commute union-lcomm*

lemma *union-right-cancel*: $M + K = N + K \longleftrightarrow M = (N::'a\ multiset)$
by (*fact add-right-cancel*)

lemma *union-left-cancel*: $K + M = K + N \longleftrightarrow M = (N::'a\ multiset)$
by (*fact add-left-cancel*)

lemma *multi-union-self-other-eq*: $(A::'a\ multiset) + X = A + Y \Longrightarrow X = Y$
by (*fact add-left-imp-eq*)

lemma *mset-less-trans*: $(M::'a\ multiset) \subset\# K \Longrightarrow K \subset\# N \Longrightarrow M \subset\# N$
by (*fact subset-mset.less-trans*)

lemma *multiset-inter-commute*: $A \#\cap B = B \#\cap A$
by (*fact subset-mset.inf.commute*)

lemma *multiset-inter-assoc*: $A \#\cap (B \#\cap C) = A \#\cap B \#\cap C$
by (*fact subset-mset.inf.assoc* [*symmetric*])

lemma *multiset-inter-left-commute*: $A \#\cap (B \#\cap C) = B \#\cap (A \#\cap C)$
by (*fact subset-mset.inf.left-commute*)

lemmas *multiset-inter-ac* =
multiset-inter-commute

multiset-inter-assoc
multiset-inter-left-commute

lemma *mult-less-not-refl*: $\neg$ *M* #⊂# (*M*::′*a*::*order multiset*)
by (*fact multiset-order.less-irrefl*)

lemma *mult-less-trans*: *K* #⊂# *M* ⟹ *M* #⊂# *N* ⟹ *K* #⊂# (*N*::′*a*::*order multiset*)
by (*fact multiset-order.less-trans*)

lemma *mult-less-not-sym*: *M* #⊂# *N* ⟹ $\neg$ *N* #⊂# (*M*::′*a*::*order multiset*)
by (*fact multiset-order.less-not-sym*)

lemma *mult-less-asym*: *M* #⊂# *N* ⟹ ($\neg$ *P* ⟹ *N* #⊂# (*M*::′*a*::*order multiset*)) ⟹ *P*
by (*fact multiset-order.less-asym*)

declaration ‹

```
let
  fun multiset-postproc - maybe-name all-values (T as Type (-, [elem-T])) (Const - $ t') =
      let
        val (maybe-opt, ps) =
          Nitpick-Model.dest-plain-fun t'
          ||> op ~~
          ||> map (apsnd (snd o HOLogic.dest-number))
        fun elems-for t =
          (case AList.lookup (op =) ps t of
            SOME n => replicate n t
          | NONE => [Const (maybe-name, elem-T --> elem-T) $ t])
      in
        (case maps elems-for (all-values elem-T) @
              (if maybe-opt then [Const (Nitpick-Model.unrep-mixfix (), elem-T)] else []) of
          [] => Const (@{const-name zero-class.zero}, T)
        | ts =>
            foldl1 (fn (t1, t2) =>
                Const (@{const-name plus-class.plus}, T --> T --> T) $ t1 $ t2)
              (map (curry (op $) (Const (@{const-name single}, elem-T --> T))) ts))
      end
    | multiset-postproc - - - - t = t
  in Nitpick-Model.register-term-postprocessor @{typ 'a multiset} multiset-postproc end
```

›

65.13 Naive implementation using lists

code-datatype *mset*

lemma [*code*]: {#} = *mset* []
  by *simp*

lemma [*code*]: {#*x*#} = *mset* [*x*]
  by *simp*

lemma *union-code* [*code*]: *mset xs* + *mset ys* = *mset* (*xs* @ *ys*)
  by *simp*

lemma [*code*]: *image-mset f* (*mset xs*) = *mset* (*map f xs*)
  by (*simp add*: *mset-map*)

lemma [*code*]: *filter-mset f* (*mset xs*) = *mset* (*filter f xs*)
  by (*simp add*: *mset-filter*)

lemma [*code*]: *mset xs* − *mset ys* = *mset* (*fold remove1 ys xs*)
  by (*rule sym*, *induct ys arbitrary*: *xs*) (*simp-all add*: *diff-add diff-right-commute*)

lemma [*code*]:
  mset xs #∩ *mset ys* =
    mset (*snd* (*fold* (λ*x* (*ys*, *zs*).
      if x ∈ *set ys then* (*remove1 x ys*, *x* # *zs*) *else* (*ys*, *zs*)) *xs* (*ys*, [])))
proof −
  have ⋀*zs*. *mset* (*snd* (*fold* (λ*x* (*ys*, *zs*).
    if x ∈ *set ys then* (*remove1 x ys*, *x* # *zs*) *else* (*ys*, *zs*)) *xs* (*ys*, *zs*))) =
      (*mset xs* #∩ *mset ys*) + *mset zs*
    by (*induct xs arbitrary*: *ys*)
      (*auto simp add*: *inter-add-right1 inter-add-right2 ac-simps*)
  then show *?thesis* **by** *simp*
qed

lemma [*code*]:
  mset xs #∪ *mset ys* =
    mset (*case-prod append* (*fold* (λ*x* (*ys*, *zs*). (*remove1 x ys*, *x* # *zs*)) *xs* (*ys*, [])))
proof −
  have ⋀*zs*. *mset* (*case-prod append* (*fold* (λ*x* (*ys*, *zs*). (*remove1 x ys*, *x* # *zs*)) *xs* (*ys*, *zs*))) =
      (*mset xs* #∪ *mset ys*) + *mset zs*
    by (*induct xs arbitrary*: *ys*) (*simp-all add*: *multiset-eq-iff*)
  then show *?thesis* **by** *simp*
qed

declare *in-multiset-in-set* [*code-unfold*]

lemma [*code*]: *count* (*mset xs*) *x* = *fold* (λ*y*. *if x* = *y then Suc else id*) *xs 0*
proof −

have $\bigwedge n.$ *fold* ($\lambda y.$ *if* $x = y$ *then Suc else id*) *xs n* = *count* (*mset xs*) $x + n$
by (*induct xs*) *simp-all*
then show *?thesis* **by** *simp*
qed

declare *set-mset-mset* [*code*]

declare *sorted-list-of-multiset-mset* [*code*]

lemma [*code*]: — not very efficient, but representation-ignorant!
mset-set A = *mset* (*sorted-list-of-set A*)
apply (*cases finite A*)
apply *simp-all*
apply (*induct A rule*: *finite-induct*)
apply (*simp-all add*: *add.commute*)
done

declare *size-mset* [*code*]

fun *ms-lesseq-impl* :: *'a list* ⇒ *'a list* ⇒ *bool option* **where**
ms-lesseq-impl [] *ys* = *Some* (*ys* ≠ [])
| *ms-lesseq-impl* (*Cons x xs*) *ys* = (*case List.extract* (*op* = *x*) *ys of*
None ⇒ *None*
| *Some* (*ys1*,-,*ys2*) ⇒ *ms-lesseq-impl xs* (*ys1* @ *ys2*))

lemma *ms-lesseq-impl*: (*ms-lesseq-impl xs ys* = *None* ⟷ ¬ *mset xs* ⊆# *mset ys*)
∧
(*ms-lesseq-impl xs ys* = *Some True* ⟷ *mset xs* ⊂# *mset ys*) ∧
(*ms-lesseq-impl xs ys* = *Some False* ⟶ *mset xs* = *mset ys*)
proof (*induct xs arbitrary*: *ys*)
case (*Nil ys*)
show *?case* **by** (*auto simp*: *mset-less-empty-nonempty*)
next
case (*Cons x xs ys*)
show *?case*
proof (*cases List.extract* (*op* = *x*) *ys*)
case *None*
hence *x*: *x* ∉ *set ys* **by** (*simp add*: *extract-None-iff*)
{
assume *mset* (*x* # *xs*) ⊆# *mset ys*
from *set-mset-mono*[*OF this*] *x* **have** *False* **by** *simp*
} **note** *nle* = *this*
moreover
{
assume *mset* (*x* # *xs*) ⊂# *mset ys*
hence *mset* (*x* # *xs*) ⊆# *mset ys* **by** *auto*
from *nle*[*OF this*] **have** *False* **.**
}
ultimately show *?thesis* **using** *None* **by** *auto*

```
  next
    case (Some res)
    obtain ys1 y ys2 where res: res = (ys1,y,ys2) by (cases res, auto)
    note Some = Some[unfolded res]
    from extract-SomeE[OF Some] have ys = ys1 @ x # ys2 by simp
    hence id: mset ys = mset (ys1 @ ys2) + {#x#}
      by (auto simp: ac-simps)
    show ?thesis unfolding ms-lesseq-impl.simps
      unfolding Some option.simps split
      unfolding id
      using Cons[of ys1 @ ys2]
      unfolding subset-mset-def subseteq-mset-def by auto
  qed
qed

lemma [code]: mset xs ⊆# mset ys ⟷ ms-lesseq-impl xs ys ≠ None
  using ms-lesseq-impl[of xs ys] by (cases ms-lesseq-impl xs ys, auto)

lemma [code]: mset xs ⊂# mset ys ⟷ ms-lesseq-impl xs ys = Some True
  using ms-lesseq-impl[of xs ys] by (cases ms-lesseq-impl xs ys, auto)

instantiation multiset :: (equal) equal
begin

definition
  [code del]: HOL.equal A (B :: 'a multiset) ⟷ A = B
lemma [code]: HOL.equal (mset xs) (mset ys) ⟷ ms-lesseq-impl xs ys = Some
False
  unfolding equal-multiset-def
  using ms-lesseq-impl[of xs ys] by (cases ms-lesseq-impl xs ys, auto)

instance
  by standard (simp add: equal-multiset-def)

end

lemma [code]: msetsum (mset xs) = listsum xs
  by (induct xs) (simp-all add: add.commute)

lemma [code]: msetprod (mset xs) = fold times xs 1
proof −
  have ⋀x. fold times xs x = msetprod (mset xs) ∗ x
    by (induct xs) (simp-all add: mult.assoc)
  then show ?thesis by simp
qed
```

Exercise for the casual reader: add implementations for *op* #⊆# and *op* #⊂# (multiset order).

Quickcheck generators

definition (**in** *term-syntax*)
msetify :: *'a::typerep list* × (*unit* ⇒ *Code-Evaluation.term*)
⇒ *'a multiset* × (*unit* ⇒ *Code-Evaluation.term*) **where**
[*code-unfold*]: *msetify xs* = *Code-Evaluation.valtermify mset* {·} *xs*

notation *fcomp* (**infixl** ∘> *60*)
notation *scomp* (**infixl** ∘→ *60*)

instantiation *multiset* :: (*random*) *random*
begin

definition
Quickcheck-Random.random i = *Quickcheck-Random.random i* ∘→ (λ*xs*. *Pair* (*msetify xs*))

instance ..

end

no-notation *fcomp* (**infixl** ∘> *60*)
no-notation *scomp* (**infixl** ∘→ *60*)

instantiation *multiset* :: (*full-exhaustive*) *full-exhaustive*
begin

definition *full-exhaustive-multiset* :: (*'a multiset* × (*unit* ⇒ *term*) ⇒ (*bool* × *term list*) *option*) ⇒ *natural* ⇒ (*bool* × *term list*) *option*
where
full-exhaustive-multiset f i = *Quickcheck-Exhaustive.full-exhaustive* (λ*xs*. *f* (*msetify xs*)) *i*

instance ..

end

hide-const (**open**) *msetify*

65.14 BNF setup

definition *rel-mset* **where**
rel-mset R X Y ⟷ (∃ *xs ys*. *mset xs* = *X* ∧ *mset ys* = *Y* ∧ *list-all2 R xs ys*)

lemma *mset-zip-take-Cons-drop-twice*:
assumes *length xs* = *length ys j* ≤ *length xs*
shows *mset* (*zip* (*take j xs* @ *x* # *drop j xs*) (*take j ys* @ *y* # *drop j ys*)) =
mset (*zip xs ys*) + {#(*x*, *y*)#}
using *assms*
proof (*induct xs ys arbitrary*: *x y j rule*: *list-induct2*)
case *Nil*

```
  thus ?case
    by simp
next
  case (Cons x xs y ys)
  thus ?case
  proof (cases j = 0)
    case True
    thus ?thesis
      by simp
  next
    case False
    then obtain k where k: j = Suc k
      by (cases j) simp
    hence k ≤ length xs
      using Cons.prems by auto
    hence mset (zip (take k xs @ x # drop k xs) (take k ys @ y # drop k ys)) =
      mset (zip xs ys) + {#(x, y)#}
      by (rule Cons.hyps(2))
    thus ?thesis
      unfolding k by (auto simp: add.commute union-lcomm)
  qed
qed

lemma ex-mset-zip-left:
  assumes length xs = length ys mset xs' = mset xs
  shows ∃ys'. length ys' = length xs' ∧ mset (zip xs' ys') = mset (zip xs ys)
using assms
proof (induct xs ys arbitrary: xs' rule: list-induct2)
  case Nil
  thus ?case
    by auto
next
  case (Cons x xs y ys xs')
  obtain j where j-len: j < length xs' and nth-j: xs' ! j = x
    by (metis Cons.prems in-set-conv-nth list.set-intros(1) mset-eq-setD)

  def xsa ≡ take j xs' @ drop (Suc j) xs'
  have mset xs' = {#x#} + mset xsa
    unfolding xsa-def using j-len nth-j
      by (metis (no-types) ab-semigroup-add-class.add-ac(1) append-take-drop-id
Cons-nth-drop-Suc
      mset.simps(2) union-code add.commute)
  hence ms-x: mset xsa = mset xs
    by (metis Cons.prems add.commute add-right-imp-eq mset.simps(2))
  then obtain ysa where
    len-a: length ysa = length xsa and ms-a: mset (zip xsa ysa) = mset (zip xs ys)
    using Cons.hyps(2) by blast

  def ys' ≡ take j ysa @ y # drop j ysa
```

have *xs'*: *xs' = take j xsa @ x # drop j xsa*
using *ms-x j-len nth-j Cons.prems xsa-def*
by (*metis append-eq-append-conv append-take-drop-id diff-Suc-Suc Cons-nth-drop-Suc length-Cons*
length-drop size-mset)
have *j-len'*: $j \leq$ *length xsa*
using *j-len xs' xsa-def*
by (*metis add-Suc-right append-take-drop-id length-Cons length-append less-eq-Suc-le not-less*)
have *length ys' = length xs'*
unfolding *ys'-def* **using** *Cons.prems len-a ms-x*
by (*metis add-Suc-right append-take-drop-id length-Cons length-append mset-eq-length*)
moreover have *mset* (*zip xs' ys'*) = *mset* (*zip* (*x # xs*) (*y # ys*))
unfolding *xs' ys'-def*
by (*rule trans*[*OF mset-zip-take-Cons-drop-twice*])
(*auto simp*: *len-a ms-a j-len' add.commute*)
ultimately show *?case*
by *blast*
qed

lemma *list-all2-reorder-left-invariance*:
assumes *rel*: *list-all2 R xs ys* **and** *ms-x*: *mset xs' = mset xs*
shows $\exists$ *ys'*. *list-all2 R xs' ys'* $\wedge$ *mset ys' = mset ys*
proof −
have *len*: *length xs = length ys*
using *rel list-all2-conv-all-nth* **by** *auto*
obtain *ys'* **where**
len': *length xs' = length ys'* **and** *ms-xy*: *mset* (*zip xs' ys'*) = *mset* (*zip xs ys*)
using *len ms-x* **by** (*metis ex-mset-zip-left*)
have *list-all2 R xs' ys'*
using *assms*(*1*) *len' ms-xy* **unfolding** *list-all2-iff* **by** (*blast dest*: *mset-eq-setD*)
moreover have *mset ys' = mset ys*
using *len len' ms-xy map-snd-zip mset-map* **by** *metis*
ultimately show *?thesis*
by *blast*
qed

lemma *ex-mset*: $\exists$ *xs*. *mset xs = X*
by (*induct X*) (*simp*, *metis mset.simps*(*2*))

inductive *pred-mset* :: (*'a* $\Rightarrow$ *bool*) $\Rightarrow$ *'a multiset* $\Rightarrow$ *bool*
where
pred-mset P {#}
| $[\![$*P a*; *pred-mset P M*$]\!] \Longrightarrow$ *pred-mset P* (*M* + {#*a*#})

bnf *'a multiset*
map: *image-mset*
sets: *set-mset*
bd: *natLeq*

wits: {#}
rel: *rel-mset*
pred: *pred-mset*
proof −
show *image-mset id* = *id*
by (*rule image-mset.id*)
show *image-mset* (*g* ∘ *f*) = *image-mset g* ∘ *image-mset f* **for** *f g*
unfolding *comp-def* **by** (*rule ext*) (*simp add*: *comp-def image-mset.compositionality*)
show ($\bigwedge$*z*. *z* ∈ *set-mset X* ⟹ *f z* = *g z*) ⟹ *image-mset f X* = *image-mset g X* **for** *f g X*
by (*induct X*) *simp-all*
show *set-mset* ∘ *image-mset f* = *op* ' *f* ∘ *set-mset* **for** *f*
by *auto*
show *card-order natLeq*
by (*rule natLeq-card-order*)
show *BNF-Cardinal-Arithmetic.cinfinite natLeq*
by (*rule natLeq-cinfinite*)
show *ordLeq3* (*card-of* (*set-mset X*)) *natLeq* **for** *X*
by *transfer*
(*auto intro*!: *ordLess-imp-ordLeq simp*: *finite-iff-ordLess-natLeq*[*symmetric*] *multiset-def*)
show *rel-mset R OO rel-mset S* ≤ *rel-mset* (*R OO S*) **for** *R S*
unfolding *rel-mset-def*[*abs-def*] *OO-def*
apply *clarify*
subgoal for *X Z Y xs ys' ys zs*
apply (*drule list-all2-reorder-left-invariance* [**where** *xs* = *ys'* **and** *ys* = *zs* **and** *xs'* = *ys*])
apply (*auto intro*: *list-all2-trans*)
done
done
show *rel-mset R* =
(λ*x y*. ∃ *z*. *set-mset z* ⊆ {(*x*, *y*). *R x y*} ∧
image-mset fst z = *x* ∧ *image-mset snd z* = *y*) **for** *R*
unfolding *rel-mset-def*[*abs-def*]
apply (*rule ext*)+
apply *safe*
apply (*rule-tac x* = *mset* (*zip xs ys*) **in** *exI*;
auto simp: *in-set-zip list-all2-iff mset-map*[*symmetric*])
apply (*rename-tac XY*)
apply (*cut-tac X* = *XY* **in** *ex-mset*)
apply (*erule exE*)
apply (*rename-tac xys*)
apply (*rule-tac x* = *map fst xys* **in** *exI*)
apply (*auto simp*: *mset-map*)
apply (*rule-tac x* = *map snd xys* **in** *exI*)
apply (*auto simp*: *mset-map list-all2I subset-eq zip-map-fst-snd*)
done
show *z* ∈ *set-mset* {#} ⟹ *False* **for** *z*
by *auto*

show *pred-mset P* = (λ*x*. *Ball* (*set-mset x*) *P*) **for** *P*
proof (*intro ext iffI*)
fix *x*
assume *pred-mset P x*
then show *Ball* (*set-mset x*) *P* **by** (*induct pred*: *pred-mset*; *simp*)
next
fix *x*
assume *Ball* (*set-mset x*) *P*
then show *pred-mset P x* **by** (*induct x*; *auto intro*: *pred-mset.intros*)
qed
qed

inductive *rel-mset′*
where
Zero[*intro*]: *rel-mset′ R* {#} {#}
| *Plus*[*intro*]: ⟦*R a b*; *rel-mset′ R M N*⟧ ⟹ *rel-mset′ R* (*M* + {#*a*#}) (*N* + {#*b*#})

lemma *rel-mset-Zero*: *rel-mset R* {#} {#}
unfolding *rel-mset-def Grp-def* **by** *auto*

declare *multiset.count*[*simp*]
declare *Abs-multiset-inverse*[*simp*]
declare *multiset.count-inverse*[*simp*]
declare *union-preserves-multiset*[*simp*]

lemma *rel-mset-Plus*:
assumes *ab*: *R a b*
and *MN*: *rel-mset R M N*
shows *rel-mset R* (*M* + {#*a*#}) (*N* + {#*b*#})
proof −
have ∃ *ya*. *image-mset fst y* + {#*a*#} = *image-mset fst ya* ∧
image-mset snd y + {#*b*#} = *image-mset snd ya* ∧
set-mset ya ⊆ {(*x*, *y*). *R x y*}
if *R a b* **and** *set-mset y* ⊆ {(*x*, *y*). *R x y*} **for** *y*
using *that* **by** (*intro exI*[*of - y* + {#(*a*,*b*)#}]) *auto*
thus *?thesis*
using *assms*
unfolding *multiset.rel-compp-Grp Grp-def* **by** *blast*
qed

lemma *rel-mset′-imp-rel-mset*: *rel-mset′ R M N* ⟹ *rel-mset R M N*
by (*induct rule*: *rel-mset′.induct*) (*auto simp*: *rel-mset-Zero rel-mset-Plus*)

lemma *rel-mset-size*: *rel-mset R M N* ⟹ *size M* = *size N*
unfolding *multiset.rel-compp-Grp Grp-def* **by** *auto*

lemma *multiset-induct2*[*case-names empty addL addR*]:
assumes *empty*: *P* {#} {#}

```
    and addL: ⋀M N a. P M N ⟹ P (M + {#a#}) N
    and addR: ⋀M N a. P M N ⟹ P M (N + {#a#})
  shows P M N
apply(induct N rule: multiset-induct)
  apply(induct M rule: multiset-induct, rule empty, erule addL)
  apply(induct M rule: multiset-induct, erule addR, erule addR)
done

lemma multiset-induct2-size[consumes 1, case-names empty add]:
  assumes c: size M = size N
    and empty: P {#} {#}
    and add: ⋀M N a b. P M N ⟹ P (M + {#a#}) (N + {#b#})
  shows P M N
  using c
proof (induct M arbitrary: N rule: measure-induct-rule[of size])
  case (less M)
  show ?case
  proof(cases M = {#})
    case True hence N = {#} using less.prems by auto
    thus ?thesis using True empty by auto
  next
     case False then obtain M1 a where M: M = M1 + {#a#} by (metis
multi-nonempty-split)
    have N ≠ {#} using False less.prems by auto
   then obtain N1 b where N: N = N1 + {#b#} by (metis multi-nonempty-split)
    have size M1 = size N1 using less.prems unfolding M N by auto
    thus ?thesis using M N less.hyps add by auto
  qed
qed

lemma msed-map-invL:
  assumes image-mset f (M + {#a#}) = N
  shows ∃N1. N = N1 + {#f a#} ∧ image-mset f M = N1
proof −
  have f a ∈# N
    using assms multiset.set-map[of f M + {#a#}] by auto
  then obtain N1 where N: N = N1 + {#f a#} using multi-member-split by
metis
  have image-mset f M = N1 using assms unfolding N by simp
  thus ?thesis using N by blast
qed

lemma msed-map-invR:
  assumes image-mset f M = N + {#b#}
  shows ∃M1 a. M = M1 + {#a#} ∧ f a = b ∧ image-mset f M1 = N
proof −
  obtain a where a: a ∈# M and fa: f a = b
    using multiset.set-map[of f M] unfolding assms
    by (metis image-iff union-single-eq-member)
```

then obtain *M1* **where** *M*: *M* = *M1* + {#*a*#} **using** *multi-member-split* **by** *metis*
have *image-mset f M1* = *N* **using** *assms* **unfolding** *M fa*[*symmetric*] **by** *simp*
thus *?thesis* **using** *M fa* **by** *blast*
qed

lemma *msed-rel-invL*:
assumes *rel-mset R* (*M* + {#*a*#}) *N*
shows ∃ *N1 b*. *N* = *N1* + {#*b*#} ∧ *R a b* ∧ *rel-mset R M N1*
proof −
obtain *K* **where** *KM*: *image-mset fst K* = *M* + {#*a*#}
and *KN*: *image-mset snd K* = *N* **and** *sK*: *set-mset K* ⊆ {(*a*, *b*). *R a b*}
using *assms*
unfolding *multiset.rel-compp-Grp Grp-def* **by** *auto*
obtain *K1 ab* **where** *K*: *K* = *K1* + {#*ab*#} **and** *a*: *fst ab* = *a*
and *K1M*: *image-mset fst K1* = *M* **using** *msed-map-invR*[*OF KM*] **by** *auto*
obtain *N1* **where** *N*: *N* = *N1* + {#*snd ab*#} **and** *K1N1*: *image-mset snd K1* = *N1*
using *msed-map-invL*[*OF KN*[*unfolded K*]] **by** *auto*
have *Rab*: *R a* (*snd ab*) **using** *sK a* **unfolding** *K* **by** *auto*
have *rel-mset R M N1* **using** *sK K1M K1N1*
unfolding *K multiset.rel-compp-Grp Grp-def* **by** *auto*
thus *?thesis* **using** *N Rab* **by** *auto*
qed

lemma *msed-rel-invR*:
assumes *rel-mset R M* (*N* + {#*b*#})
shows ∃ *M1 a*. *M* = *M1* + {#*a*#} ∧ *R a b* ∧ *rel-mset R M1 N*
proof −
obtain *K* **where** *KN*: *image-mset snd K* = *N* + {#*b*#}
and *KM*: *image-mset fst K* = *M* **and** *sK*: *set-mset K* ⊆ {(*a*, *b*). *R a b*}
using *assms*
unfolding *multiset.rel-compp-Grp Grp-def* **by** *auto*
obtain *K1 ab* **where** *K*: *K* = *K1* + {#*ab*#} **and** *b*: *snd ab* = *b*
and *K1N*: *image-mset snd K1* = *N* **using** *msed-map-invR*[*OF KN*] **by** *auto*
obtain *M1* **where** *M*: *M* = *M1* + {#*fst ab*#} **and** *K1M1*: *image-mset fst K1* = *M1*
using *msed-map-invL*[*OF KM*[*unfolded K*]] **by** *auto*
have *Rab*: *R* (*fst ab*) *b* **using** *sK b* **unfolding** *K* **by** *auto*
have *rel-mset R M1 N* **using** *sK K1N K1M1*
unfolding *K multiset.rel-compp-Grp Grp-def* **by** *auto*
thus *?thesis* **using** *M Rab* **by** *auto*
qed

lemma *rel-mset-imp-rel-mset'*:
assumes *rel-mset R M N*
shows *rel-mset' R M N*
using *assms* **proof**(*induct M arbitrary*: *N rule*: *measure-induct-rule*[*of size*])
case (*less M*)

have *c*: *size M* = *size N* **using** *rel-mset-size*[*OF less.prems*] **.**
show *?case*
proof(*cases M* = {#})
case *True* **hence** *N* = {#} **using** *c* **by** *simp*
thus *?thesis* **using** *True rel-mset'.Zero* **by** *auto*
next
case *False* **then obtain** *M1 a* **where** *M*: *M* = *M1* + {#*a*#} **by** (*metis multi-nonempty-split*)
obtain *N1 b* **where** *N*: *N* = *N1* + {#*b*#} **and** *R*: *R a b* **and** *ms*: *rel-mset R M1 N1*
using *msed-rel-invL*[*OF less.prems*[*unfolded M*]] **by** *auto*
have *rel-mset' R M1 N1* **using** *less.hyps*[*of M1 N1*] *ms* **unfolding** *M* **by** *simp*
thus *?thesis* **using** *rel-mset'.Plus*[*of R a b*, *OF R*] **unfolding** *M N* **by** *simp*
qed
qed

lemma *rel-mset-rel-mset'*: *rel-mset R M N* = *rel-mset' R M N*
using *rel-mset-imp-rel-mset' rel-mset'-imp-rel-mset* **by** *auto*

The main end product for *rel-mset*: inductive characterization:

lemmas *rel-mset-induct*[*case-names empty add*, *induct pred*: *rel-mset*] =
rel-mset'.induct[*unfolded rel-mset-rel-mset'*[*symmetric*]]

65.15 Size setup

lemma *multiset-size-o-map*: *size-multiset g* ∘ *image-mset f* = *size-multiset* (*g* ∘ *f*)
apply (*rule ext*)
subgoal for *x* **by** (*induct x*) *auto*
done

setup ‹
BNF-LFP-Size.register-size-global @{*type-name multiset*} @{*const-name size-multiset*}
@{*thm size-multiset-overloaded-def*}
@{*thms size-multiset-empty size-multiset-single size-multiset-union size-empty size-single*
size-union}
@{*thms multiset-size-o-map*}
›

hide-const (**open**) *wcount*

end

66 More Theorems about the Multiset Order

theory *Multiset-Order*
imports *Multiset*
begin

66.0.1 Alternative characterizations

context *order*
begin

lemma *reflp-le*: *reflp* (*op* ≤)
  unfolding *reflp-def* **by** *simp*

lemma *antisymP-le*: *antisymP* (*op* ≤)
  unfolding *antisym-def* **by** *auto*

lemma *transp-le*: *transp* (*op* ≤)
  unfolding *transp-def* **by** *auto*

lemma *irreflp-less*: *irreflp* (*op* <)
  unfolding *irreflp-def* **by** *simp*

lemma *antisymP-less*: *antisymP* (*op* <)
  unfolding *antisym-def* **by** *auto*

lemma *transp-less*: *transp* (*op* <)
  unfolding *transp-def* **by** *auto*

lemmas *le-trans* = *transp-le*[*unfolded transp-def*, *rule-format*]

lemma *order-mult*: *class.order*
  $(\lambda M\ N.\ (M, N) \in mult\ \{(x, y).\ x < y\} \lor M = N)$
  $(\lambda M\ N.\ (M, N) \in mult\ \{(x, y).\ x < y\})$
  (**is** *class.order ?le ?less*)
proof −
  have *irrefl*: $\bigwedge M :: 'a\ multiset.\ \neg\ ?less\ M\ M$
  proof
    fix $M :: 'a\ multiset$
    have *trans* $\{(x'::'a, x).\ x' < x\}$
      by (*rule transI*) *simp*
    moreover
    assume $(M, M) \in mult\ \{(x, y).\ x < y\}$
    ultimately have $\exists I\ J\ K.\ M = I + J \land M = I + K$
      $\land J \neq \{\#\} \land (\forall k \in set\text{-}mset\ K.\ \exists j \in set\text{-}mset\ J.\ (k, j) \in \{(x, y).\ x < y\})$
      by (*rule mult-implies-one-step*)
    then obtain *I J K* **where** $M = I + J$ **and** $M = I + K$
       and $J \neq \{\#\}$ **and** $(\forall k \in set\text{-}mset\ K.\ \exists j \in set\text{-}mset\ J.\ (k, j) \in \{(x, y).\ x <$
$y\})$ **by** *blast*
    then have *aux1*: $K \neq \{\#\}$ **and** *aux2*: $\forall k \in set\text{-}mset\ K.\ \exists j \in set\text{-}mset\ K.\ k < j$
by *auto*
    have *finite* (*set-mset K*) **by** *simp*
    moreover note *aux2*
    ultimately have *set-mset K* = {}
      by (*induct rule*: *finite-induct*)
       (*simp*, *metis* (*mono-tags*) *insert-absorb insert-iff insert-not-empty less-irrefl*

less-trans)
  with *aux1* **show** *False* **by** *simp*
 qed
 have *trans*: $\bigwedge K\ M\ N$ :: *'a multiset*. *?less K M* $\Longrightarrow$ *?less M N* $\Longrightarrow$ *?less K N*
  unfolding *mult-def* **by** (*blast intro*: *trancl-trans*)
 show *class.order ?le ?less*
  by *standard* (*auto simp add*: *le-multiset-def irrefl dest*: *trans*)
qed

The Dershowitz–Manna ordering:

definition $\textit{less-multiset}_{DM}$ **where**
 $\textit{less-multiset}_{DM}$ $M\ N \longleftrightarrow$
 $(\exists X\ Y.\ X \neq \{\#\} \wedge X \leq\# N \wedge M = (N - X) + Y \wedge (\forall k.\ k \in\# Y \longrightarrow (\exists a.\ a \in\# X \wedge k < a)))$

The Huet–Oppen ordering:

definition $\textit{less-multiset}_{HO}$ **where**
 $\textit{less-multiset}_{HO}$ $M\ N \longleftrightarrow M \neq N \wedge (\forall y.\ \textit{count}\ N\ y < \textit{count}\ M\ y \longrightarrow (\exists x.\ y < x \wedge \textit{count}\ M\ x < \textit{count}\ N\ x))$

lemma $\textit{mult-imp-less-multiset}_{HO}$:
 $(M, N) \in \textit{mult}\ \{(x, y).\ x < y\} \Longrightarrow \textit{less-multiset}_{HO}\ M\ N$
proof (*unfold mult-def*, *induct rule*: *trancl-induct*)
 case (*base P*)
 then show *?case*
  by (*auto elim*!: *mult1-lessE simp add*: *count-eq-zero-iff* $\textit{less-multiset}_{HO}$*-def split*: *if-splits dest*!: *Suc-lessD*)
next
 case (*step N P*)
 from *step*(*3*) **have** $M \neq N$ **and**
  $**$: $\bigwedge y.\ \textit{count}\ N\ y < \textit{count}\ M\ y \Longrightarrow (\exists x{>}y.\ \textit{count}\ M\ x < \textit{count}\ N\ x)$
  by (*simp-all add*: $\textit{less-multiset}_{HO}$*-def*)
 from *step*(*2*) **obtain** *M0 a K* **where**
  $*$: $P = M0 + \{\#a\#\}\ N = M0 + K\ a \notin\# K\ \bigwedge b.\ b \in\# K \Longrightarrow b < a$
  by (*blast elim*: *mult1-lessE*)
 from ‹$M \neq N$› $**$ $*(1,2,3)$ **have** $M \neq P$ **by** (*force dest*: $*(4)$ *split*: *if-splits*)
 moreover
 { assume $\textit{count}\ P\ a \leq \textit{count}\ M\ a$
  with ‹$a \notin\# K$› **have** $\textit{count}\ N\ a < \textit{count}\ M\ a$ **unfolding** $*(1,2)$
   by (*auto simp add*: *not-in-iff*)
   with $**$ **obtain** *z* **where** *z*: $z > a$ $\textit{count}\ M\ z < \textit{count}\ N\ z$
    by *blast*
   with $*$ **have** $\textit{count}\ N\ z \leq \textit{count}\ P\ z$
    by (*force simp add*: *not-in-iff*)
   with *z* **have** $\exists z > a.\ \textit{count}\ M\ z < \textit{count}\ P\ z$ **by** *auto*
 } note *count-a* = *this*
 { fix *y*
  assume *count-y*: $\textit{count}\ P\ y < \textit{count}\ M\ y$
  have $\exists x{>}y.\ \textit{count}\ M\ x < \textit{count}\ P\ x$

proof (*cases* $y = a$)
case *True*
with *count-y count-a* **show** *?thesis* **by** *auto*
next
case *False*
show *?thesis*
proof (*cases* $y \in\# K$)
case *True*
with $*(4)$ **have** $y < a$ **by** *simp*
then show *?thesis* **by** (*cases count* $P\ a \leq$ *count* $M\ a$) (*auto dest*: *count-a intro*: *less-trans*)
next
case *False*
with ‹$y \neq a$› **have** *count* $P\ y =$ *count* $N\ y$ **unfolding** $*(1,2)$
by (*simp add*: *not-in-iff*)
with *count-y* $**$ **obtain** z **where** z: $z > y$ *count* $M\ z <$ *count* $N\ z$ **by** *auto*
show *?thesis*
proof (*cases* $z \in\# K$)
case *True*
with $*(4)$ **have** $z < a$ **by** *simp*
with $z(1)$ **show** *?thesis*
by (*cases count* $P\ a \leq$ *count* $M\ a$) (*auto dest*!: *count-a intro*: *less-trans*)
next
case *False*
with ‹$a \notin\# K$› **have** *count* $N\ z \leq$ *count* $P\ z$ **unfolding** $*$
by (*auto simp add*: *not-in-iff*)
with z **show** *?thesis* **by** *auto*
qed
qed
qed
}
ultimately show *?case* **unfolding** *less-multiset*$_{HO}$*-def* **by** *blast*
qed

lemma *less-multiset*$_{DM}$*-imp-mult*:
less-multiset$_{DM}$ $M\ N \Longrightarrow (M, N) \in$ *mult* $\{(x, y).\ x < y\}$
proof −
assume *less-multiset*$_{DM}$ $M\ N$
then obtain $X\ Y$ **where**
$X \neq \{\#\}$ **and** $X \leq\# N$ **and** $M = N - X + Y$ **and** $\forall k.\ k \in\# Y \longrightarrow (\exists a.\ a \in\# X \wedge k < a)$
unfolding *less-multiset*$_{DM}$*-def* **by** *blast*
then have $(N - X + Y, N - X + X) \in$ *mult* $\{(x, y).\ x < y\}$
by (*intro one-step-implies-mult*) (*auto simp*: *Bex-def trans-def*)
with ‹$M = N - X + Y$› ‹$X \leq\# N$› **show** $(M, N) \in$ *mult* $\{(x, y).\ x < y\}$
by (*metis subset-mset.diff-add*)
qed

lemma *less-multiset*$_{HO}$*-imp-less-multiset*$_{DM}$: *less-multiset*$_{HO}$ $M\ N \Longrightarrow$ *less-multiset*$_{DM}$

M N
unfolding *less-multiset$_{DM}$-def*
proof (*intro iffI exI conjI*)
assume *less-multiset$_{HO}$* M N
then obtain z **where** z: *count* M z < *count* N z
unfolding *less-multiset$_{HO}$-def* **by** (*auto simp*: *multiset-eq-iff nat-neq-iff*)
def $X \equiv N - M$
def $Y \equiv M - N$
from z **show** $X \neq \{\#\}$ **unfolding** *X-def* **by** (*auto simp*: *multiset-eq-iff not-less-eq-eq Suc-le-eq*)
from z **show** $X \leq\# N$ **unfolding** *X-def* **by** *auto*
show $M = (N - X) + Y$ **unfolding** *X-def Y-def multiset-eq-iff count-union count-diff* **by** *force*
show $\forall k.\ k \in\# Y \longrightarrow (\exists a.\ a \in\# X \wedge k < a)$
proof (*intro allI impI*)
fix k
assume $k \in\# Y$
then have *count* N k < *count* M k **unfolding** *Y-def*
by (*auto simp add*: *in-diff-count*)
with ‹*less-multiset$_{HO}$* M N› **obtain** a **where** $k < a$ **and** *count* M a < *count* N a
unfolding *less-multiset$_{HO}$-def* **by** *blast*
then show $\exists a.\ a \in\# X \wedge k < a$ **unfolding** *X-def*
by (*auto simp add*: *in-diff-count*)
qed
qed

lemma *mult-less-multiset$_{DM}$*: $(M, N) \in$ *mult* $\{(x, y).\ x < y\} \longleftrightarrow$ *less-multiset$_{DM}$* M N
by (*metis less-multiset$_{DM}$-imp-mult less-multiset$_{HO}$-imp-less-multiset$_{DM}$ mult-imp-less-multiset$_{HO}$*)

lemma *mult-less-multiset$_{HO}$*: $(M, N) \in$ *mult* $\{(x, y).\ x < y\} \longleftrightarrow$ *less-multiset$_{HO}$* M N
by (*metis less-multiset$_{DM}$-imp-mult less-multiset$_{HO}$-imp-less-multiset$_{DM}$ mult-imp-less-multiset$_{HO}$*)

lemmas *mult$_{DM}$* = *mult-less-multiset$_{DM}$*[*unfolded less-multiset$_{DM}$-def*]
lemmas *mult$_{HO}$* = *mult-less-multiset$_{HO}$*[*unfolded less-multiset$_{HO}$-def*]

end

context *linorder*
begin

lemma *total-le*: *total* $\{(a :: 'a, b).\ a \leq b\}$
unfolding *total-on-def* **by** *auto*

lemma *total-less*: *total* $\{(a :: 'a, b).\ a < b\}$
unfolding *total-on-def* **by** *auto*

lemma *linorder-mult*: *class.linorder*
($\lambda M\ N.\ (M,\ N) \in mult\ \{(x,\ y).\ x < y\} \vee M = N$)
($\lambda M\ N.\ (M,\ N) \in mult\ \{(x,\ y).\ x < y\}$)
proof –
interpret *o*: *order*
($\lambda M\ N.\ (M,\ N) \in mult\ \{(x,\ y).\ x < y\} \vee M = N$)
($\lambda M\ N.\ (M,\ N) \in mult\ \{(x,\ y).\ x < y\}$)
by (*rule order-mult*)
show *?thesis* **by** *unfold-locales* (*auto 0 3 simp*: $mult_{HO}$ *not-less-iff-gr-or-eq*)
qed

end

lemma *less-multiset-less-multiset*$_{HO}$:
$M\ \#{\subset}\#\ N \longleftrightarrow less\text{-}multiset_{HO}\ M\ N$
unfolding *less-multiset-def* $mult_{HO}$ *less-multiset*$_{HO}$*-def* **..**

lemmas $less\text{-}multiset_{DM} = mult_{DM}$[*folded less-multiset-def*]
lemmas $less\text{-}multiset_{HO} = mult_{HO}$[*folded less-multiset-def*]

lemma *le-multiset*$_{HO}$:
fixes *M N* :: (*'a* :: *linorder*) *multiset*
shows $M\ \#{\subseteq}\#\ N \longleftrightarrow (\forall y.\ count\ N\ y < count\ M\ y \longrightarrow (\exists x.\ y < x \wedge count\ M\ x < count\ N\ x))$
by (*auto simp*: *le-multiset-def less-multiset*$_{HO}$)

lemma *wf-less-multiset*: *wf* $\{(M :: ('a :: wellorder)\ multiset,\ N).\ M\ \#{\subset}\#\ N\}$
unfolding *less-multiset-def* **by** (*auto intro*: *wf-mult wf*)

lemma *order-multiset*: *class.order*
(*le-multiset* :: (*'a* :: *order*) *multiset* ⇒ (*'a* :: *order*) *multiset* ⇒ *bool*)
(*less-multiset* :: (*'a* :: *order*) *multiset* ⇒ (*'a* :: *order*) *multiset* ⇒ *bool*)
by *unfold-locales*

lemma *linorder-multiset*: *class.linorder*
(*le-multiset* :: (*'a* :: *linorder*) *multiset* ⇒ (*'a* :: *linorder*) *multiset* ⇒ *bool*)
(*less-multiset* :: (*'a* :: *linorder*) *multiset* ⇒ (*'a* :: *linorder*) *multiset* ⇒ *bool*)
by *unfold-locales* (*fastforce simp add*: *less-multiset*$_{HO}$ *le-multiset-def not-less-iff-gr-or-eq*)

interpretation *multiset-linorder*: *linorder*
le-multiset :: (*'a* :: *linorder*) *multiset* ⇒ (*'a* :: *linorder*) *multiset* ⇒ *bool*
less-multiset :: (*'a* :: *linorder*) *multiset* ⇒ (*'a* :: *linorder*) *multiset* ⇒ *bool*
by (*rule linorder-multiset*)

interpretation *multiset-wellorder*: *wellorder*
le-multiset :: (*'a* :: *wellorder*) *multiset* ⇒ (*'a* :: *wellorder*) *multiset* ⇒ *bool*
less-multiset :: (*'a* :: *wellorder*) *multiset* ⇒ (*'a* :: *wellorder*) *multiset* ⇒ *bool*
by *unfold-locales* (*blast intro*: *wf-less-multiset* [*unfolded wf-def*, *simplified*, *rule-format*])

lemma *le-multiset-total*:
fixes M N :: $('a$ :: *linorder*) *multiset*
shows $\neg$ M $\#\subseteq\#$ $N \Longrightarrow N$ $\#\subseteq\#$ M
by (*metis multiset-linorder.le-cases*)

lemma *less-eq-imp-le-multiset*:
fixes M N :: $('a$ :: *linorder*) *multiset*
shows $M \leq\#$ $N \Longrightarrow M$ $\#\subseteq\#$ N
unfolding *le-multiset-def less-multiset*$_{HO}$
by (*simp add*: *less-le-not-le subseteq-mset-def*)

lemma *less-multiset-right-total*:
fixes M :: $('a$ :: *linorder*) *multiset*
shows M $\#\subset\#$ M + {#*undefined*#}
unfolding *le-multiset-def less-multiset*$_{HO}$ **by** *simp*

lemma *le-multiset-empty-left*[*simp*]:
fixes M :: $('a$ :: *linorder*) *multiset*
shows {#} $\#\subseteq\#$ M
by (*simp add*: *less-eq-imp-le-multiset*)

lemma *le-multiset-empty-right*[*simp*]:
fixes M :: $('a$ :: *linorder*) *multiset*
shows $M \neq$ {#} $\Longrightarrow \neg$ M $\#\subseteq\#$ {#}
by (*metis le-multiset-empty-left multiset-order.antisym*)

lemma *less-multiset-empty-left*[*simp*]:
fixes M :: $('a$ :: *linorder*) *multiset*
shows $M \neq$ {#} $\Longrightarrow$ {#} $\#\subset\#$ M
by (*simp add*: *less-multiset*$_{HO}$)

lemma *less-multiset-empty-right*[*simp*]:
fixes M :: $('a$ :: *linorder*) *multiset*
shows $\neg$ M $\#\subset\#$ {#}
using *le-empty less-multiset*$_{DM}$ **by** *blast*

lemma
fixes M N :: $('a$ :: *linorder*) *multiset*
shows
le-multiset-plus-left[*simp*]: N $\#\subseteq\#$ $(M + N)$ **and**
le-multiset-plus-right[*simp*]: M $\#\subseteq\#$ $(M + N)$
using [[*metis-verbose* = *false*]] **by** (*metis less-eq-imp-le-multiset mset-le-add-left add.commute*)+

lemma
fixes M N :: $('a$ :: *linorder*) *multiset*
shows
less-multiset-plus-plus-left-iff[*simp*]: $M + N$ $\#\subset\#$ $M' + N \longleftrightarrow M$ $\#\subset\#$ M'
and

less-multiset-plus-plus-right-iff [*simp*]: $M + N \#{\subset}\# M + N' \longleftrightarrow N \#{\subset}\# N'$
unfolding *less-multiset*$_{HO}$ **by** *auto*

lemma *add-eq-self-empty-iff*: $M + N = M \longleftrightarrow N = \{\#\}$
by (*metis add.commute add-diff-cancel-right′ monoid-add-class.add.left-neutral*)

lemma
fixes *M N* :: (*′a* :: *linorder*) *multiset*
shows
less-multiset-plus-left-nonempty[*simp*]: $M \neq \{\#\} \Longrightarrow N \#{\subset}\# M + N$ **and**
less-multiset-plus-right-nonempty[*simp*]: $N \neq \{\#\} \Longrightarrow M \#{\subset}\# M + N$
using [[*metis-verbose* = *false*]]
by (*metis add.right-neutral less-multiset-empty-left less-multiset-plus-plus-right-iff add.commute*)+

lemma *ex-gt-imp-less-multiset*: $(\exists\, y :: 'a :: linorder.\ y \in\# N \wedge (\forall\, x.\ x \in\# M \longrightarrow x < y)) \Longrightarrow M \#{\subset}\# N$
unfolding *less-multiset*$_{HO}$
by (*metis count-eq-zero-iff count-greater-zero-iff less-le-not-le*)

lemma *ex-gt-count-imp-less-multiset*:
$(\forall\, y :: 'a :: linorder.\ y \in\# M + N \longrightarrow y \leq x) \Longrightarrow count\ M\ x < count\ N\ x \Longrightarrow M \#{\subset}\# N$
unfolding *less-multiset*$_{HO}$
by (*metis add-gr-0 count-union mem-Collect-eq not-gr0 not-le not-less-iff-gr-or-eq set-mset-def*)

lemma *union-less-diff-plus*: $P \leq\# M \Longrightarrow N \#{\subset}\# P \Longrightarrow M - P + N \#{\subset}\# M$
by (*drule subset-mset.diff-add*[*symmetric*]) (*metis union-less-mono2*)

end

67 Numeral Syntax for Types

theory *Numeral-Type*
imports *Cardinality*
begin

67.1 Numeral Types

typedef *num0* = *UNIV* :: *nat set* **..**
typedef *num1* = *UNIV* :: *unit set* **..**

typedef *′a bit0* = $\{0 ..< 2 * int\ CARD('a::finite)\}$
proof
show $0 \in \{0 ..< 2 * int\ CARD('a)\}$
by *simp*
qed

typedef *'a bit1* = {*0* ..< *1* + *2* ∗ *int CARD*(*'a*::*finite*)}
proof
  show *0* ∈ {*0* ..< *1* + *2* ∗ *int CARD*(*'a*)}
    by *simp*
qed

lemma *card-num0* [*simp*]: *CARD* (*num0*) = *0*
  unfolding *type-definition.card* [*OF type-definition-num0*]
  by *simp*

lemma *infinite-num0*: ¬ *finite* (*UNIV* :: *num0 set*)
  using *card-num0*[*unfolded card-eq-0-iff*]
  by *simp*

lemma *card-num1* [*simp*]: *CARD*(*num1*) = *1*
  unfolding *type-definition.card* [*OF type-definition-num1*]
  by (*simp only*: *card-UNIV-unit*)

lemma *card-bit0* [*simp*]: *CARD*(*'a bit0*) = *2* ∗ *CARD*(*'a*::*finite*)
  unfolding *type-definition.card* [*OF type-definition-bit0*]
  by *simp*

lemma *card-bit1* [*simp*]: *CARD*(*'a bit1*) = *Suc* (*2* ∗ *CARD*(*'a*::*finite*))
  unfolding *type-definition.card* [*OF type-definition-bit1*]
  by *simp*

instance *num1* :: *finite*
proof
  show *finite* (*UNIV*::*num1 set*)
    unfolding *type-definition.univ* [*OF type-definition-num1*]
    using *finite* **by** (*rule finite-imageI*)
qed

instance *bit0* :: (*finite*) *card2*
proof
  show *finite* (*UNIV*::*'a bit0 set*)
    unfolding *type-definition.univ* [*OF type-definition-bit0*]
    by *simp*
  show *2* ≤ *CARD*(*'a bit0*)
    by *simp*
qed

instance *bit1* :: (*finite*) *card2*
proof
  show *finite* (*UNIV*::*'a bit1 set*)
    unfolding *type-definition.univ* [*OF type-definition-bit1*]
    by *simp*
  show *2* ≤ *CARD*(*'a bit1*)
    by *simp*

qed

67.2 Locales for for modular arithmetic subtypes

locale *mod-type* =
  fixes *n* :: *int*
  and *Rep* :: *'a*::{*zero*,*one*,*plus*,*times*,*uminus*,*minus*} ⇒ *int*
  and *Abs* :: *int* ⇒ *'a*::{*zero*,*one*,*plus*,*times*,*uminus*,*minus*}
  assumes *type*: *type-definition Rep Abs* {*0*..<*n*}
  and *size1*: *1* < *n*
  and *zero-def*: *0* = *Abs 0*
  and *one-def*: *1* = *Abs 1*
  and *add-def*: *x* + *y* = *Abs* ((*Rep x* + *Rep y*) *mod n*)
  and *mult-def*: *x* ∗ *y* = *Abs* ((*Rep x* ∗ *Rep y*) *mod n*)
  and *diff-def*: *x* − *y* = *Abs* ((*Rep x* − *Rep y*) *mod n*)
  and *minus-def*: − *x* = *Abs* ((− *Rep x*) *mod n*)
begin

lemma *size0*: *0* < *n*
using *size1* **by** *simp*

lemmas *definitions* =
  zero-def one-def add-def mult-def minus-def diff-def

lemma *Rep-less-n*: *Rep x* < *n*
by (*rule type-definition.Rep* [*OF type*, *simplified*, *THEN conjunct2*])

lemma *Rep-le-n*: *Rep x* ≤ *n*
by (*rule Rep-less-n* [*THEN order-less-imp-le*])

lemma *Rep-inject-sym*: *x* = *y* ⟷ *Rep x* = *Rep y*
by (*rule type-definition.Rep-inject* [*OF type*, *symmetric*])

lemma *Rep-inverse*: *Abs* (*Rep x*) = *x*
by (*rule type-definition.Rep-inverse* [*OF type*])

lemma *Abs-inverse*: *m* ∈ {*0*..<*n*} ⟹ *Rep* (*Abs m*) = *m*
by (*rule type-definition.Abs-inverse* [*OF type*])

lemma *Rep-Abs-mod*: *Rep* (*Abs* (*m mod n*)) = *m mod n*
by (*simp add*: *Abs-inverse pos-mod-conj* [*OF size0*])

lemma *Rep-Abs-0*: *Rep* (*Abs 0*) = *0*
by (*simp add*: *Abs-inverse size0*)

lemma *Rep-0*: *Rep 0* = *0*
by (*simp add*: *zero-def Rep-Abs-0*)

lemma *Rep-Abs-1*: *Rep* (*Abs 1*) = *1*

by (*simp add*: *Abs-inverse size1*)

lemma *Rep-1*: *Rep 1 = 1*
by (*simp add*: *one-def Rep-Abs-1*)

lemma *Rep-mod*: *Rep x mod n = Rep x*
apply (*rule-tac x=x* **in** *type-definition.Abs-cases* [*OF type*])
apply (*simp add*: *type-definition.Abs-inverse* [*OF type*])
apply (*simp add*: *mod-pos-pos-trivial*)
done

lemmas *Rep-simps* =
Rep-inject-sym Rep-inverse Rep-Abs-mod Rep-mod Rep-Abs-0 Rep-Abs-1

lemma *comm-ring-1*: *OFCLASS('a, comm-ring-1-class)*
apply (*intro-classes*, *unfold definitions*)
apply (*simp-all add*: *Rep-simps zmod-simps field-simps*)
done

end

locale *mod-ring* = *mod-type n Rep Abs*
for *n* :: *int*
and *Rep* :: *'a::{comm-ring-1} ⇒ int*
and *Abs* :: *int ⇒ 'a::{comm-ring-1}*
begin

lemma *of-nat-eq*: *of-nat k = Abs (int k mod n)*
apply (*induct k*)
apply (*simp add*: *zero-def*)
apply (*simp add*: *Rep-simps add-def one-def zmod-simps ac-simps*)
done

lemma *of-int-eq*: *of-int z = Abs (z mod n)*
apply (*cases z rule*: *int-diff-cases*)
apply (*simp add*: *Rep-simps of-nat-eq diff-def zmod-simps*)
done

lemma *Rep-numeral*:
Rep (numeral w) = numeral w mod n
using *of-int-eq* [*of numeral w*]
by (*simp add*: *Rep-inject-sym Rep-Abs-mod*)

lemma *iszero-numeral*:
iszero (numeral w::'a) ⟷ numeral w mod n = 0
by (*simp add*: *Rep-inject-sym Rep-numeral Rep-0 iszero-def*)

lemma *cases*:
assumes *1*: ⋀*z*. ⟦*(x::'a) = of-int z*; *0 ≤ z*; *z < n*⟧ ⟹ *P*

shows *P*
apply (*cases x rule*: *type-definition.Abs-cases* [*OF type*])
apply (*rule-tac z=y* **in** *1*)
apply (*simp-all add*: *of-int-eq mod-pos-pos-trivial*)
done

lemma *induct*:
$(\bigwedge z.\ [\![0 \leq z;\ z < n]\!] \Longrightarrow P\ (of\text{-}int\ z)) \Longrightarrow P\ (x{::}'a)$
by (*cases x rule*: *cases*) *simp*

end

67.3 Ring class instances

Unfortunately *ring-1* instance is not possible for *num1*, since 0 and 1 are not distinct.

instantiation *num1* :: {*comm-ring*,*comm-monoid-mult*,*numeral*}
begin

lemma *num1-eq-iff*: $(x{::}num1) = (y{::}num1) \longleftrightarrow True$
by (*induct x*, *induct y*) *simp*

instance
by *standard* (*simp-all add*: *num1-eq-iff*)

end

instantiation
bit0 **and** *bit1* :: (*finite*) {*zero*,*one*,*plus*,*times*,*uminus*,*minus*}
begin

definition *Abs-bit0'* :: *int* ⇒ *'a bit0* **where**
Abs-bit0' x = *Abs-bit0* (*x mod int CARD*(*'a bit0*))

definition *Abs-bit1'* :: *int* ⇒ *'a bit1* **where**
Abs-bit1' x = *Abs-bit1* (*x mod int CARD*(*'a bit1*))

definition *0* = *Abs-bit0 0*
definition *1* = *Abs-bit0 1*
definition *x* + *y* = *Abs-bit0'* (*Rep-bit0 x* + *Rep-bit0 y*)
definition *x* ∗ *y* = *Abs-bit0'* (*Rep-bit0 x* ∗ *Rep-bit0 y*)
definition *x* − *y* = *Abs-bit0'* (*Rep-bit0 x* − *Rep-bit0 y*)
definition − *x* = *Abs-bit0'* (− *Rep-bit0 x*)

definition *0* = *Abs-bit1 0*
definition *1* = *Abs-bit1 1*
definition *x* + *y* = *Abs-bit1'* (*Rep-bit1 x* + *Rep-bit1 y*)
definition *x* ∗ *y* = *Abs-bit1'* (*Rep-bit1 x* ∗ *Rep-bit1 y*)
definition *x* − *y* = *Abs-bit1'* (*Rep-bit1 x* − *Rep-bit1 y*)

definition $-$ *x* = *Abs-bit1'* ($-$ *Rep-bit1 x*)

instance ..

end

interpretation *bit0*:
mod-type int CARD('*a*::*finite bit0*)
Rep-bit0 :: '*a*::*finite bit0* ⇒ *int*
Abs-bit0 :: *int* ⇒ '*a*::*finite bit0*
apply (*rule mod-type.intro*)
apply (*simp add*: *of-nat-mult type-definition-bit0*)
apply (*rule one-less-int-card*)
apply (*rule zero-bit0-def*)
apply (*rule one-bit0-def*)
apply (*rule plus-bit0-def* [*unfolded Abs-bit0'-def*])
apply (*rule times-bit0-def* [*unfolded Abs-bit0'-def*])
apply (*rule minus-bit0-def* [*unfolded Abs-bit0'-def*])
apply (*rule uminus-bit0-def* [*unfolded Abs-bit0'-def*])
done

interpretation *bit1*:
mod-type int CARD('*a*::*finite bit1*)
Rep-bit1 :: '*a*::*finite bit1* ⇒ *int*
Abs-bit1 :: *int* ⇒ '*a*::*finite bit1*
apply (*rule mod-type.intro*)
apply (*simp add*: *of-nat-mult type-definition-bit1*)
apply (*rule one-less-int-card*)
apply (*rule zero-bit1-def*)
apply (*rule one-bit1-def*)
apply (*rule plus-bit1-def* [*unfolded Abs-bit1'-def*])
apply (*rule times-bit1-def* [*unfolded Abs-bit1'-def*])
apply (*rule minus-bit1-def* [*unfolded Abs-bit1'-def*])
apply (*rule uminus-bit1-def* [*unfolded Abs-bit1'-def*])
done

instance *bit0* :: (*finite*) *comm-ring-1*
by (*rule bit0.comm-ring-1*)

instance *bit1* :: (*finite*) *comm-ring-1*
by (*rule bit1.comm-ring-1*)

interpretation *bit0*:
mod-ring int CARD('*a*::*finite bit0*)
Rep-bit0 :: '*a*::*finite bit0* ⇒ *int*
Abs-bit0 :: *int* ⇒ '*a*::*finite bit0*
..

interpretation *bit1*:

mod-ring int CARD('a::finite bit1)
Rep-bit1 :: *'a::finite bit1* ⇒ *int*
Abs-bit1 :: *int* ⇒ *'a::finite bit1*
..

Set up cases, induction, and arithmetic

lemmas *bit0-cases* [*case-names of-int*, *cases type*: *bit0*] = *bit0.cases*
lemmas *bit1-cases* [*case-names of-int*, *cases type*: *bit1*] = *bit1.cases*

lemmas *bit0-induct* [*case-names of-int*, *induct type*: *bit0*] = *bit0.induct*
lemmas *bit1-induct* [*case-names of-int*, *induct type*: *bit1*] = *bit1.induct*

lemmas *bit0-iszero-numeral* [*simp*] = *bit0.iszero-numeral*
lemmas *bit1-iszero-numeral* [*simp*] = *bit1.iszero-numeral*

lemmas [*simp*] = *eq-numeral-iff-iszero* [**where** *'a='a bit0*] **for** *dummy* :: *'a::finite*
lemmas [*simp*] = *eq-numeral-iff-iszero* [**where** *'a='a bit1*] **for** *dummy* :: *'a::finite*

67.4 Order instances

instantiation *bit0* **and** *bit1* :: (*finite*) *linorder* **begin**
definition $a < b \longleftrightarrow$ *Rep-bit0* $a <$ *Rep-bit0* b
definition $a \leq b \longleftrightarrow$ *Rep-bit0* $a \leq$ *Rep-bit0* b
definition $a < b \longleftrightarrow$ *Rep-bit1* $a <$ *Rep-bit1* b
definition $a \leq b \longleftrightarrow$ *Rep-bit1* $a \leq$ *Rep-bit1* b

instance
by(*intro-classes*)
(*auto simp add*: *less-eq-bit0-def less-bit0-def less-eq-bit1-def less-bit1-def Rep-bit0-inject Rep-bit1-inject*)
end

lemma (**in** *preorder*) *tranclp-less*: *op* $<^{++}$ = *op* <
by(*auto simp add*: *fun-eq-iff intro*: *less-trans elim*: *tranclp.induct*)

instance *bit0* **and** *bit1* :: (*finite*) *wellorder*
proof −
have *wf* {(*x* :: *'a bit0*, *y*). *x* < *y*}
by(*auto simp add*: *trancl-def tranclp-less intro*!: *finite-acyclic-wf acyclicI*)
thus *OFCLASS*(*'a bit0*, *wellorder-class*)
by(*rule wf-wellorderI*) *intro-classes*
next
have *wf* {(*x* :: *'a bit1*, *y*). *x* < *y*}
by(*auto simp add*: *trancl-def tranclp-less intro*!: *finite-acyclic-wf acyclicI*)
thus *OFCLASS*(*'a bit1*, *wellorder-class*)
by(*rule wf-wellorderI*) *intro-classes*
qed

67.5 Code setup and type classes for code generation

Code setup for *num0* and *num1*

definition *Num0* :: *num0* **where** *Num0* = *Abs-num0 0*
code-datatype *Num0*

instantiation *num0* :: *equal* **begin**
definition *equal-num0* :: *num0* ⇒ *num0* ⇒ *bool*
  where *equal-num0* = *op* =
instance by *intro-classes* (*simp add*: *equal-num0-def*)
end

lemma *equal-num0-code* [*code*]:
  equal-class.equal Num0 Num0 = *True*
by(*rule equal-refl*)

code-datatype *1* :: *num1*

instantiation *num1* :: *equal* **begin**
definition *equal-num1* :: *num1* ⇒ *num1* ⇒ *bool*
  where *equal-num1* = *op* =
instance by *intro-classes* (*simp add*: *equal-num1-def*)
end

lemma *equal-num1-code* [*code*]:
  equal-class.equal (*1* :: *num1*) *1* = *True*
by(*rule equal-refl*)

instantiation *num1* :: *enum* **begin**
definition *enum-class.enum* = [*1* :: *num1*]
definition *enum-class.enum-all P* = *P* (*1* :: *num1*)
definition *enum-class.enum-ex P* = *P* (*1* :: *num1*)
instance
  by *intro-classes*
    (*auto simp add*: *enum-num1-def enum-all-num1-def enum-ex-num1-def num1-eq-iff Ball-def*,
      (*metis* (*full-types*) *num1-eq-iff*)+)
end

instantiation *num0* **and** *num1* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*num0*) *False*
definition *card-UNIV* = *Phantom*(*num0*) *0*
definition *finite-UNIV* = *Phantom*(*num1*) *True*
definition *card-UNIV* = *Phantom*(*num1*) *1*
instance
  by *intro-classes*
    (*simp-all add*: *finite-UNIV-num0-def card-UNIV-num0-def infinite-num0 finite-UNIV-num1-def card-UNIV-num1-def*)
end

Code setup for $'a$ *bit0* and $'a$ *bit1*

declare
bit0.Rep-inverse[*code abstype*]
bit0.Rep-0[*code abstract*]
bit0.Rep-1[*code abstract*]

lemma *Abs-bit0'-code* [*code abstract*]:
Rep-bit0 (*Abs-bit0' x* :: *'a* :: *finite bit0*) = *x mod int* (*CARD*(*'a bit0*))
by(*auto simp add*: *Abs-bit0'-def intro*!: *Abs-bit0-inverse*)

lemma *inj-on-Abs-bit0*:
inj-on (*Abs-bit0* :: *int* ⇒ *'a bit0*) {*0..<2* ∗ *int CARD*(*'a* :: *finite*)}
by(*auto intro*: *inj-onI simp add*: *Abs-bit0-inject*)

declare
bit1.Rep-inverse[*code abstype*]
bit1.Rep-0[*code abstract*]
bit1.Rep-1[*code abstract*]

lemma *Abs-bit1'-code* [*code abstract*]:
Rep-bit1 (*Abs-bit1' x* :: *'a* :: *finite bit1*) = *x mod int* (*CARD*(*'a bit1*))
by(*auto simp add*: *Abs-bit1'-def intro*!: *Abs-bit1-inverse*)

lemma *inj-on-Abs-bit1*:
inj-on (*Abs-bit1* :: *int* ⇒ *'a bit1*) {*0..<1* + *2* ∗ *int CARD*(*'a* :: *finite*)}
by(*auto intro*: *inj-onI simp add*: *Abs-bit1-inject*)

instantiation *bit0* **and** *bit1* :: (*finite*) *equal* **begin**

definition *equal-class.equal x y* ⟷ *Rep-bit0 x* = *Rep-bit0 y*
definition *equal-class.equal x y* ⟷ *Rep-bit1 x* = *Rep-bit1 y*

instance
by *intro-classes* (*simp-all add*: *equal-bit0-def equal-bit1-def Rep-bit0-inject Rep-bit1-inject*)

end

instantiation *bit0* :: (*finite*) *enum* **begin**
definition (*enum-class.enum* :: *'a bit0 list*) = *map* (*Abs-bit0'* ∘ *int*) (*upt 0* (*CARD*(*'a bit0*)))
definition *enum-class.enum-all P* = (∀ *b* :: *'a bit0* ∈ *set enum-class.enum*. *P b*)
definition *enum-class.enum-ex P* = (∃ *b* :: *'a bit0* ∈ *set enum-class.enum*. *P b*)

instance
proof(*intro-classes*)
show *distinct* (*enum-class.enum* :: *'a bit0 list*)
by (*simp add*: *enum-bit0-def distinct-map inj-on-def Abs-bit0'-def Abs-bit0-inject mod-pos-pos-trivial*)

show *univ-eq*: (*UNIV* :: *'a bit0 set*) = *set enum-class.enum*
unfolding *enum-bit0-def type-definition.Abs-image*[*OF type-definition-bit0*, *symmetric*]
by(*simp add*: *image-comp* [*symmetric*] *inj-on-Abs-bit0 card-image image-int-atLeastLessThan*)
(*auto intro*!: *image-cong*[*OF refl*] *simp add*: *Abs-bit0'-def mod-pos-pos-trivial*)

fix *P* :: *'a bit0* ⇒ *bool*
show *enum-class.enum-all P* = *Ball UNIV P*
and *enum-class.enum-ex P* = *Bex UNIV P*
by(*simp-all add*: *enum-all-bit0-def enum-ex-bit0-def univ-eq*)
qed

end

instantiation *bit1* :: (*finite*) *enum* **begin**
definition (*enum-class.enum* :: *'a bit1 list*) = *map* (*Abs-bit1'* ∘ *int*) (*upt 0* (*CARD*(*'a bit1*)))
definition *enum-class.enum-all P* = (∀ *b* :: *'a bit1* ∈ *set enum-class.enum*. *P b*)
definition *enum-class.enum-ex P* = (∃ *b* :: *'a bit1* ∈ *set enum-class.enum*. *P b*)

instance
proof(*intro-classes*)
show *distinct* (*enum-class.enum* :: *'a bit1 list*)
by(*simp only*: *Abs-bit1'-def zmod-int*[*symmetric*] *enum-bit1-def distinct-map Suc-eq-plus1 card-bit1 o-apply inj-on-def*)
(*clarsimp simp add*: *Abs-bit1-inject*)

show *univ-eq*: (*UNIV* :: *'a bit1 set*) = *set enum-class.enum*
unfolding *enum-bit1-def type-definition.Abs-image*[*OF type-definition-bit1*, *symmetric*]
by(*simp add*: *image-comp* [*symmetric*] *inj-on-Abs-bit1 card-image image-int-atLeastLessThan*)
(*auto intro*!: *image-cong*[*OF refl*] *simp add*: *Abs-bit1'-def mod-pos-pos-trivial*)

fix *P* :: *'a bit1* ⇒ *bool*
show *enum-class.enum-all P* = *Ball UNIV P*
and *enum-class.enum-ex P* = *Bex UNIV P*
by(*simp-all add*: *enum-all-bit1-def enum-ex-bit1-def univ-eq*)
qed

end

instantiation *bit0* **and** *bit1* :: (*finite*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*'a bit0*) *True*
definition *finite-UNIV* = *Phantom*(*'a bit1*) *True*
instance by *intro-classes* (*simp-all add*: *finite-UNIV-bit0-def finite-UNIV-bit1-def*)
end

instantiation *bit0* **and** *bit1* :: ({*finite*,*card-UNIV*}) *card-UNIV* **begin**
definition *card-UNIV* = *Phantom*(*'a bit0*) (*2* ∗ *of-phantom* (*card-UNIV* :: *'a*

card-UNIV))
definition *card-UNIV* = *Phantom*(*'a bit1*) (*1* + *2* ∗ *of-phantom* (*card-UNIV* :: *'a card-UNIV*))
instance by *intro-classes* (*simp-all add*: *card-UNIV-bit0-def card-UNIV-bit1-def card-UNIV*)
end

67.6 Syntax

```
syntax
  -NumeralType :: num-token => type  (-)
  -NumeralType0 :: type (0)
  -NumeralType1 :: type (1)

translations
  (type) 1 == (type) num1
  (type) 0 == (type) num0

parse-translation ‹
  let
    fun mk-bintype n =
      let
        fun mk-bit 0 = Syntax.const @{type-syntax bit0}
          | mk-bit 1 = Syntax.const @{type-syntax bit1};
        fun bin-of n =
          if n = 1 then Syntax.const @{type-syntax num1}
          else if n = 0 then Syntax.const @{type-syntax num0}
          else if n = ~1 then raise TERM (negative type numeral, [])
          else
            let val (q, r) = Integer.div-mod n 2;
            in mk-bit r $ bin-of q end;
      in bin-of n end;

    fun numeral-tr [Free (str, -)] = mk-bintype (the (Int.fromString str))
      | numeral-tr ts = raise TERM (numeral-tr, ts);

  in [(@{syntax-const -NumeralType}, K numeral-tr)] end;
›

print-translation ‹
  let
    fun int-of [] = 0
      | int-of (b :: bs) = b + 2 * int-of bs;

    fun bin-of (Const (@{type-syntax num0}, -)) = []
      | bin-of (Const (@{type-syntax num1}, -)) = [1]
      | bin-of (Const (@{type-syntax bit0}, -) $ bs) = 0 :: bin-of bs
      | bin-of (Const (@{type-syntax bit1}, -) $ bs) = 1 :: bin-of bs
      | bin-of t = raise TERM (bin-of, [t]);
```

```
  fun bit-tr' b [t] =
      let
        val rev-digs = b :: bin-of t handle TERM - => raise Match
        val i = int-of rev-digs;
        val num = string-of-int (abs i);
      in
        Syntax.const @{syntax-const -NumeralType} $ Syntax.free num
      end
    | bit-tr' b - = raise Match;
 in
  [(@{type-syntax bit0}, K (bit-tr' 0)),
   (@{type-syntax bit1}, K (bit-tr' 1))]
 end;
⟩
```

67.7 Examples

lemma *CARD(0) = 0* **by** *simp*
lemma *CARD(17) = 17* **by** *simp*
lemma $8 * 11$ ^ $3 - 6 = (2::5)$ **by** *simp*

end

68 ω-words

theory *Omega-Words-Fun*

imports *Infinite-Set*
begin

Note: This theory is based on Stefan Merz's work.

Automata recognize languages, which are sets of words. For the theory of ω-automata, we are mostly interested in ω-words, but it is sometimes useful to reason about finite words, too. We are modeling finite words as lists; this lets us benefit from the existing library. Other formalizations could be investigated, such as representing words as functions whose domains are initial intervals of the natural numbers.

68.1 Type declaration and elementary operations

We represent ω-words as functions from the natural numbers to the alphabet type. Other possible formalizations include a coinductive definition or a uniform encoding of finite and infinite words, as studied by Müller et al.

type-synonym
 'a word = nat ⇒ 'a

We can prefix a finite word to an ω-word, and a way to obtain an ω-word from a finite, non-empty word is by ω-iteration.

definition
conc :: [*'a list*, *'a word*] ⇒ *'a word* (**infixr** ⌢ *65*)
where $w \frown x == \lambda n.\ if\ n < length\ w\ then\ w!n\ else\ x\ (n - length\ w)$

definition
iter :: *'a list* ⇒ *'a word* ((-$^\omega$) [*1000*])
where *iter w* == *if w* = [] *then undefined else* (λn. *w*!(*n mod* (*length w*)))

lemma *conc-empty*[*simp*]: [] ⌢ *w* = *w*
unfolding *conc-def* **by** *auto*

lemma *conc-fst*[*simp*]: *n* < *length w* ⟹ (*w* ⌢ *x*) *n* = *w*!*n*
by (*simp add*: *conc-def*)

lemma *conc-snd*[*simp*]: ¬(*n* < *length w*) ⟹ (*w* ⌢ *x*) *n* = *x* (*n* − *length w*)
by (*simp add*: *conc-def*)

lemma *iter-nth* [*simp*]: *0* < *length w* ⟹ w^ω *n* = *w*!(*n mod* (*length w*))
by (*simp add*: *iter-def*)

lemma *conc-conc*[*simp*]: *u* ⌢ *v* ⌢ *w* = (*u* @ *v*) ⌢ *w* (**is** *?lhs* = *?rhs*)
proof
fix *n*
have *u*: *n* < *length u* ⟹ *?lhs n* = *?rhs n*
by (*simp add*: *conc-def nth-append*)
have *v*: ⟦ ¬(*n* < *length u*); *n* < *length u* + *length v* ⟧ ⟹ *?lhs n* = *?rhs n*
by (*simp add*: *conc-def nth-append*, *arith*)
have *w*: ¬(*n* < *length u* + *length v*) ⟹ *?lhs n* = *?rhs n*
by (*simp add*: *conc-def nth-append*, *arith*)
from *u v w* **show** *?lhs n* = *?rhs n* **by** *blast*
qed

lemma *range-conc*[*simp*]: *range* ($w_1 \frown w_2$) = *set* w_1 ∪ *range* w_2
proof (*intro equalityI subsetI*)
fix *a*
assume *a* ∈ *range* ($w_1 \frown w_2$)
then obtain *i* **where** *1*: *a* = ($w_1 \frown w_2$) *i* **by** *auto*
then show *a* ∈ *set* w_1 ∪ *range* w_2
unfolding *1* **by** (*cases i* < *length* w_1) *simp-all*
next
fix *a*
assume *a*: *a* ∈ *set* w_1 ∪ *range* w_2
then show *a* ∈ *range* ($w_1 \frown w_2$)
proof
assume *a* ∈ *set* w_1
then obtain *i* **where** *1*: *i* < *length* w_1 *a* = w_1 ! *i*
using *in-set-conv-nth* **by** *metis*

    show *?thesis*
    proof
      show $a = (w_1 \frown w_2)\ i$ **using** *1* **by** *auto*
      show $i \in UNIV$ **by** *rule*
    qed
  next
    assume $a \in range\ w_2$
    then obtain i **where** *1*: $a = w_2\ i$ **by** *auto*
    show *?thesis*
    proof
      show $a = (w_1 \frown w_2)\ (length\ w_1 + i)$ **using** *1* **by** *simp*
      show $length\ w_1 + i \in UNIV$ **by** *rule*
    qed
  qed
qed

lemma *iter-unroll*: $0 < length\ w \Longrightarrow w^\omega = w \frown w^\omega$
  by (*rule ext*) (*simp add*: *conc-def mod-geq*)

68.2 Subsequence, Prefix, and Suffix

definition *suffix* :: [*nat*, *'a word*] ⇒ *'a word*
  where *suffix k x* ≡ λ*n*. *x* (*k*+*n*)

definition *subsequence* :: *'a word* ⇒ *nat* ⇒ *nat* ⇒ *'a list* (- [- → -] *900*)
  where *subsequence w i j* ≡ *map w* [*i*..<*j*]

abbreviation *prefix* :: *nat* ⇒ *'a word* ⇒ *'a list*
  where *prefix n w* ≡ *subsequence w 0 n*

lemma *suffix-nth* [*simp*]: (*suffix k x*) *n* = *x* (*k*+*n*)
  by (*simp add*: *suffix-def*)

lemma *suffix-0* [*simp*]: *suffix 0 x* = *x*
  by (*simp add*: *suffix-def*)

lemma *suffix-suffix* [*simp*]: *suffix m* (*suffix k x*) = *suffix* (*k*+*m*) *x*
  by (*rule ext*) (*simp add*: *suffix-def add.assoc*)

lemma *subsequence-append*: *prefix* (*i* + *j*) *w* = *prefix i w* @ (*w* [*i* → *i* + *j*])
  unfolding *map-append*[*symmetric*] *upt-add-eq-append*[*OF le0*] *subsequence-def*
..

lemma *subsequence-drop*[*simp*]: *drop i* (*w* [*j* → *k*]) = *w* [*j* + *i* → *k*]
  by (*simp add*: *subsequence-def drop-map*)

lemma *subsequence-empty*[*simp*]: *w* [*i* → *j*] = [] ⟷ *j* ≤ *i*
  by (*auto simp add*: *subsequence-def*)

lemma *subsequence-length*[*simp*]: *length* (*subsequence w i j*) = *j* − *i*
  by (*simp add*: *subsequence-def*)

lemma *subsequence-nth*[*simp*]: *k* < *j* − *i* ⟹ (*w* [*i* → *j*]) ! *k* = *w* (*i* + *k*)
  unfolding *subsequence-def*
  by *auto*

lemma *subseq-to-zero*[*simp*]: *w*[*i*→*0*] = []
  by *simp*

lemma *subseq-to-smaller*[*simp*]: *i*≥*j* ⟹ *w*[*i*→*j*] = []
  by *simp*

lemma *subseq-to-Suc*[*simp*]: *i*≤*j* ⟹ *w* [*i* → *Suc j*] = *w* [*i* → *j*] @ [*w j*]
  by (*auto simp*: *subsequence-def*)

lemma *subsequence-singleton*[*simp*]: *w* [*i* → *Suc i*] = [*w i*]
  by (*auto simp*: *subsequence-def*)

lemma *subsequence-prefix-suffix*: *prefix* (*j* − *i*) (*suffix i w*) = *w* [*i* → *j*]
proof (*cases i* ≤ *j*)
  case *True*
  have *w* [*i* → *j*] = *map w* (*map* (λ*n*. *n* + *i*) [*0*..<*j* − *i*])
    unfolding *map-add-upt subsequence-def*
    using *le-add-diff-inverse2*[*OF True*] **by** *force*
  also
  have ... = *map* (λ*n*. *w* (*n* + *i*)) [*0*..<*j* − *i*]
    unfolding *map-map comp-def* **by** *blast*
  finally
  show *?thesis*
    unfolding *subsequence-def suffix-def add.commute*[*of i*] **by** *simp*
next
  case *False*
  then show *?thesis*
    by (*simp add*: *subsequence-def*)
qed

lemma *prefix-suffix*: *x* = *prefix n x* ⌢ (*suffix n x*)
  by (*rule ext*) (*simp add*: *subsequence-def conc-def*)

declare *prefix-suffix*[*symmetric*, *simp*]

lemma *word-split*: **obtains** v_1 v_2 **where** $v = v_1 \frown v_2$ *length* $v_1 = k$
proof
  show *v* = *prefix k v* ⌢ *suffix k v*
    by (*rule prefix-suffix*)

show *length* (*prefix k v*) = *k*
by *simp*
qed

lemma *set-subsequence*[*simp*]: *set* (*w*[*i*→*j*]) = *w*'{*i*..<*j*}
unfolding *subsequence-def* **by** *auto*

lemma *subsequence-take*[*simp*]: *take i* (*w* [*j* → *k*]) = *w* [*j* → *min* (*j* + *i*) *k*]
by (*simp add*: *subsequence-def take-map min-def*)

lemma *subsequence-shift*[*simp*]: (*suffix i w*) [*j* → *k*] = *w* [*i* + *j* → *i* + *k*]
by (*metis add-diff-cancel-left subsequence-prefix-suffix suffix-suffix*)

lemma *suffix-subseq-join*[*simp*]: *i* ≤ *j* ⟹ *v* [*i* → *j*] ⌢ *suffix j v* = *suffix i v*
by (*metis* (*no-types*, *lifting*) *Nat.add-0-right le-add-diff-inverse prefix-suffix subsequence-shift suffix-suffix*)

lemma *prefix-conc-fst*[*simp*]:
assumes *j* ≤ *length w*
shows *prefix j* (*w* ⌢ *w*′) = *take j w*
proof −
have ∀ *i* < *j*. (*prefix j* (*w* ⌢ *w*′)) ! *i* = (*take j w*) ! *i*
using *assms* **by** (*simp add*: *conc-fst subsequence-def*)
thus *?thesis*
by (*simp add*: *assms list-eq-iff-nth-eq min.absorb2*)
qed

lemma *prefix-conc-snd*[*simp*]:
assumes *n* ≥ *length u*
shows *prefix n* (*u* ⌢ *v*) = *u* @ *prefix* (*n* − *length u*) *v*
proof (*intro nth-equalityI allI impI*)
show *length* (*prefix n* (*u* ⌢ *v*)) = *length* (*u* @ *prefix* (*n* − *length u*) *v*)
using *assms* **by** *simp*
fix *i*
assume *i* < *length* (*prefix n* (*u* ⌢ *v*))
then show *prefix n* (*u* ⌢ *v*) ! *i* = (*u* @ *prefix* (*n* − *length u*) *v*) ! *i*
by (*cases i* < *length u*) (*auto simp*: *nth-append*)
qed

lemma *prefix-conc-length*[*simp*]: *prefix* (*length w*) (*w* ⌢ *w*′) = *w*
by *simp*

lemma *suffix-conc-fst*[*simp*]:
assumes *n* ≤ *length u*
shows *suffix n* (*u* ⌢ *v*) = *drop n u* ⌢ *v*
proof
show *suffix n* (*u* ⌢ *v*) *i* = (*drop n u* ⌢ *v*) *i* **for** *i*
using *assms* **by** (*cases n* + *i* < *length u*) (*auto simp*: *algebra-simps*)

qed

lemma *suffix-conc-snd*[*simp*]:
 assumes $n \geq length\ u$
 shows $suffix\ n\ (u \frown v) = suffix\ (n - length\ u)\ v$
proof
 show $suffix\ n\ (u \frown v)\ i = suffix\ (n - length\ u)\ v\ i$ **for** i
  using *assms* **by** *simp*
qed

lemma *suffix-conc-length*[*simp*]: $suffix\ (length\ w)\ (w \frown w') = w'$
 unfolding *conc-def* **by** *force*

lemma *concat-eq*[*iff*]:
 assumes $length\ v_1 = length\ v_2$
 shows $v_1 \frown u_1 = v_2 \frown u_2 \longleftrightarrow v_1 = v_2 \wedge u_1 = u_2$
 (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
 assume *?lhs*
 then have *1*: $(v_1 \frown u_1)\ i = (v_2 \frown u_2)\ i$ **for** i **by** *auto*
 show *?rhs*
 proof (*intro conjI ext nth-equalityI allI impI*)
  show $length\ v_1 = length\ v_2$ **by** (*rule assms*(*1*))
 next
  fix i
  assume *2*: $i < length\ v_1$
  have *3*: $i < length\ v_2$ **using** *assms*(*1*) *2* **by** *simp*
  show $v_1\ !\ i = v_2\ !\ i$ **using** *1*[*of i*] *2 3* **by** *simp*
 next
  show $u_1\ i = u_2\ i$ **for** i
   using *1*[*of length* $v_1 + i$] *assms*(*1*) **by** *simp*
 qed
next
 assume *?rhs*
 then show *?lhs* **by** *simp*
qed

lemma *same-concat-eq*[*iff*]: $u \frown v = u \frown w \longleftrightarrow v = w$
 by *simp*

lemma *comp-concat*[*simp*]: $f \circ u \frown v = map\ f\ u \frown (f \circ v)$
proof
 fix i
 show $(f \circ u \frown v)\ i = (map\ f\ u \frown (f \circ v))\ i$
  by (*cases* $i < length\ u$) *simp-all*
qed

68.3 Prepending

primrec *build* :: $'a \Rightarrow 'a\ word \Rightarrow 'a\ word$ (**infixr** *##* *65*)
  where $(a\ \#\#\ w)\ 0 = a \mid (a\ \#\#\ w)\ (Suc\ i) = w\ i$

lemma *build-eq*[*iff*]: $a_1\ \#\#\ w_1 = a_2\ \#\#\ w_2 \longleftrightarrow a_1 = a_2 \wedge w_1 = w_2$
proof
  assume *1*: $a_1\ \#\#\ w_1 = a_2\ \#\#\ w_2$
  have *2*: $(a_1\ \#\#\ w_1)\ i = (a_2\ \#\#\ w_2)\ i$ **for** i
    using *1* **by** *auto*
  show $a_1 = a_2 \wedge w_1 = w_2$
  proof (*intro conjI ext*)
    show $a_1 = a_2$
      using *2*[*of 0*] **by** *simp*
    show $w_1\ i = w_2\ i$ **for** i
      using *2*[*of Suc i*] **by** *simp*
  qed
next
  assume *1*: $a_1 = a_2 \wedge w_1 = w_2$
  show $a_1\ \#\#\ w_1 = a_2\ \#\#\ w_2$ **using** *1* **by** *simp*
qed

lemma *build-cons*[*simp*]: $(a\ \#\ u) \frown v = a\ \#\#\ u \frown v$
proof
  fix i
  show $((a\ \#\ u) \frown v)\ i = (a\ \#\#\ u \frown v)\ i$
  proof (*cases i*)
    case *0*
    show *?thesis* **unfolding** *0* **by** *simp*
  next
    case (*Suc j*)
    show *?thesis* **unfolding** *Suc* **by** (*cases* $j < length\ u$, *simp+*)
  qed
qed

lemma *build-append*[*simp*]: $(w\ @\ a\ \#\ u) \frown v = w \frown a\ \#\#\ u \frown v$
  unfolding *conc-conc*[*symmetric*] **by** *simp*

lemma *build-first*[*simp*]: $w\ 0\ \#\#\ suffix\ (Suc\ 0)\ w = w$
proof
  show $(w\ 0\ \#\#\ suffix\ (Suc\ 0)\ w)\ i = w\ i$ **for** i
    by (*cases i*) *simp-all*
qed

lemma *build-split*[*intro*]: $w = w\ 0\ \#\#\ suffix\ 1\ w$
  by *simp*

lemma *build-range*[*simp*]: $range\ (a\ \#\#\ w) = insert\ a\ (range\ w)$
proof *safe*
  show $(a\ \#\#\ w)\ i \notin range\ w \Longrightarrow (a\ \#\#\ w)\ i = a$ **for** i

```
    by (cases i) auto
  show a ∈ range (a ## w)
  proof (rule range-eqI)
    show a = (a ## w) 0 by simp
  qed
  show w i ∈ range (a ## w) for i
  proof (rule range-eqI)
    show w i = (a ## w) (Suc i) by simp
  qed
qed

lemma suffix-singleton-suffix[simp]: w i ## suffix (Suc i) w = suffix i w
  using suffix-subseq-join[of i Suc i w]
  by simp
```

Find the first occurrence of a letter from a given set

```
lemma word-first-split-set:
  assumes A ∩ range w ≠ {}
  obtains u a v where w = u ⌢ [a] ⌢ v A ∩ set u = {} a ∈ A
proof −
  def i ≡ LEAST i. w i ∈ A
  show ?thesis
  proof
    show w = prefix i w ⌢ [w i] ⌢ suffix (Suc i) w
      by simp
    show A ∩ set (prefix i w) = {}
      apply safe
      subgoal premises prems for a
      proof −
        from prems obtain k where 3: k < i w k = a
          by auto
        have 4: w k ∉ A
          using not-less-Least 3(1) unfolding i-def .
        show ?thesis
          using prems(1) 3(2) 4 by auto
      qed
      done
    show w i ∈ A
      using LeastI assms(1) unfolding i-def by fast
  qed
qed
```

68.4 The limit set of an ω-word

The limit set (also called infinity set) of an ω-word is the set of letters that appear infinitely often in the word. This set plays an important role in defining acceptance conditions of ω-automata.

```
definition limit :: 'a word ⇒ 'a set
  where limit x ≡ {a . ∃∞n . x n = a}
```

lemma *limit-iff-frequent*: $a \in limit\ x \longleftrightarrow (\exists_\infty n\ .\ x\ n = a)$
  by (*simp add*: *limit-def*)

The following is a different way to define the limit, using the reverse image, making the laws about reverse image applicable to the limit set. (Might want to change the definition above?)

lemma *limit-vimage*: $(a \in limit\ x) = infinite\ (x -` \{a\})$
  by (*simp add*: *limit-def Inf-many-def vimage-def*)

lemma *two-in-limit-iff*:
  $(\{a, b\} \subseteq limit\ x) =$
    $((\exists n.\ x\ n = a\ ) \wedge (\forall n.\ x\ n = a \longrightarrow (\exists m>n.\ x\ m = b)) \wedge (\forall m.\ x\ m = b \longrightarrow (\exists n>m.\ x\ n = a)))$
  (**is** *?lhs* = (*?r1* ∧ *?r2* ∧ *?r3*))
proof
  assume *lhs*: *?lhs*
  hence *1*: *?r1* **by** (*auto simp*: *limit-def elim*: *INFM-EX*)
  from *lhs* **have** $\forall n.\ \exists m>n.\ x\ m = b$ **by** (*auto simp*: *limit-def INFM-nat*)
  hence *2*: *?r2* **by** *simp*
  from *lhs* **have** $\forall m.\ \exists n>m.\ x\ n = a$ **by** (*auto simp*: *limit-def INFM-nat*)
  hence *3*: *?r3* **by** *simp*
  from *1 2 3* **show** *?r1* ∧ *?r2* ∧ *?r3* **by** *simp*
next
  assume *?r1* ∧ *?r2* ∧ *?r3*
  hence *1*: *?r1* **and** *2*: *?r2* **and** *3*: *?r3* **by** *simp+*
  have *infa*: $\forall m.\ \exists n\geq m.\ x\ n = a$
  proof
    fix *m*
    show $\exists n\geq m.\ x\ n = a$ (**is** *?A m*)
    proof (*induct m*)
      from *1* **show** *?A 0* **by** *simp*
    next
      fix *m*
      assume *ih*: *?A m*
      then obtain *n* **where** *n*: $n \geq m\ x\ n = a$ **by** *auto*
      with *2* **obtain** *k* **where** *k*: $k>n\ x\ k = b$ **by** *auto*
      with *3* **obtain** *l* **where** *l*: $l>k\ x\ l = a$ **by** *auto*
      from *n k l* **have** $l \geq Suc\ m$ **by** *auto*
      with *l* **show** *?A* (*Suc m*) **by** *auto*
    qed
  qed
  hence *infa'*: $\exists_\infty n.\ x\ n = a$ **by** (*simp add*: *INFM-nat-le*)
  have $\forall n.\ \exists m>n.\ x\ m = b$
  proof
    fix *n*
    from *infa* **obtain** *k* **where** *k1*: $k\geq n$ **and** *k2*: $x\ k = a$ **by** *auto*
    from *2 k2* **obtain** *l* **where** *l1*: $l>k$ **and** *l2*: $x\ l = b$ **by** *auto*
    from *k1 l1* **have** $l > n$ **by** *auto*

with *l2* **show** $\exists\, m>n.\ x\ m = b$ **by** *auto*
qed
hence $\exists_\infty m.\ x\ m = b$ **by** (*simp add*: *INFM-nat*)
with *infa'* **show** *?lhs* **by** (*auto simp*: *limit-def*)
qed

For ω-words over a finite alphabet, the limit set is non-empty. Moreover, from some position onward, any such word contains only letters from its limit set.

lemma *limit-nonempty*:
assumes *fin*: *finite* (*range x*)
shows $\exists\, a.\ a \in$ *limit x*
proof −
from *fin* **obtain** *a* **where** $a \in$ *range x* $\wedge$ *infinite* (x −' $\{a\}$)
by (*rule inf-img-fin-domE*) *auto*
hence $a \in$ *limit x*
by (*auto simp add*: *limit-vimage*)
thus *?thesis* **..**
qed

lemmas *limit-nonemptyE* = *limit-nonempty*[*THEN exE*]

lemma *limit-inter-INF*:
assumes *hyp*: *limit w* $\cap$ $S \neq \{\}$
shows $\exists_\infty\ n.\ w\ n \in S$
proof −
from *hyp* **obtain** *x* **where** $\exists_\infty\ n.\ w\ n = x$ **and** $x \in S$
by (*auto simp add*: *limit-def*)
thus *?thesis*
by (*auto elim*: *INFM-mono*)
qed

The reverse implication is true only if S is finite.

lemma *INF-limit-inter*:
assumes *hyp*: $\exists_\infty\ n.\ w\ n \in\ S$
and *fin*: *finite* ($S \cap$ *range w*)
shows $\exists\, a.\ a \in$ *limit w* $\cap\ S$
proof (*rule ccontr*)
assume *contra*: $\neg(\exists\, a.\ a \in$ *limit w* $\cap\ S)$
hence $\forall\, a{\in}S.$ *finite* $\{n.\ w\ n = a\}$
by (*auto simp add*: *limit-def Inf-many-def*)
with *fin* **have** *finite* (*UN a*:*S* $\cap$ *range w*. $\{n.\ w\ n = a\}$)
by *auto*
moreover
have (*UN a*:*S* $\cap$ *range w*. $\{n.\ w\ n = a\}$) = $\{n.\ w\ n \in S\}$
by *auto*
moreover
note *hyp*
ultimately show *False*

by (*simp add*: *Inf-many-def*)
qed

lemma *fin-ex-inf-eq-limit*: *finite A* $\Longrightarrow$ ($\exists_\infty i.\ w\ i \in A$) $\longleftrightarrow$ *limit w* $\cap$ *A* $\neq$ {}
by (*metis INF-limit-inter equals0D finite-Int limit-inter-INF*)

lemma *limit-in-range-suffix*: *limit x* $\subseteq$ *range* (*suffix k x*)
proof
fix *a*
assume *a* $\in$ *limit x*
then obtain *l* **where**
kl: $k < l$ **and** *xl*: $x\ l = a$
by (*auto simp add*: *limit-def INFM-nat*)
from *kl* **obtain** *m* **where** $l = k+m$
by (*auto simp add*: *less-iff-Suc-add*)
with *xl* **show** *a* $\in$ *range* (*suffix k x*)
by *auto*
qed

lemma *limit-in-range*: *limit r* $\subseteq$ *range r*
using *limit-in-range-suffix*[*of r 0*] **by** *simp*

lemmas *limit-in-range-suffixD* = *limit-in-range-suffix*[*THEN subsetD*]

lemma *limit-subset*: *limit f* $\subseteq$ *f* ‘ {*n*..}
using *limit-in-range-suffix*[*of f n*] **unfolding** *suffix-def* **by** *auto*

theorem *limit-is-suffix*:
assumes *fin*: *finite* (*range x*)
shows $\exists k$. *limit x* = *range* (*suffix k x*)
proof −
have $\exists k$. *range* (*suffix k x*) $\subseteq$ *limit x*
proof −
— The set of letters that are not in the limit is certainly finite.
from *fin* **have** *finite* (*range x* − *limit x*)
by *simp*
— Moreover, any such letter occurs only finitely often
moreover
have $\forall a \in$ *range x* − *limit x*. *finite* (*x* −‘ {*a*})
by (*auto simp add*: *limit-vimage*)
— Thus, there are only finitely many occurrences of such letters.
ultimately have *finite* (*UN a* : *range x* − *limit x*. *x* −‘ {*a*})
by (*blast intro*: *finite-UN-I*)
— Therefore these occurrences are within some initial interval.
then obtain *k* **where** (*UN a* : *range x* − *limit x*. *x* −‘ {*a*}) $\subseteq$ {..<*k*}
by (*blast dest*: *finite-nat-bounded*)
— This is just the bound we are looking for.
hence $\forall m.\ k \leq m \longrightarrow x\ m \in$ *limit x*
by (*auto simp add*: *limit-vimage*)

```
    hence range (suffix k x) ⊆ limit x
      by auto
    thus ?thesis ..
  qed
  then obtain k where range (suffix k x) ⊆ limit x ..
  with limit-in-range-suffix
  have limit x = range (suffix k x)
    by (rule subset-antisym)
  thus ?thesis ..
qed
```

lemmas *limit-is-suffixE* = *limit-is-suffix*[*THEN exE*]

The limit set enjoys some simple algebraic laws with respect to concatenation, suffixes, iteration, and renaming.

```
theorem limit-conc [simp]: limit (w ⌢ x) = limit x
proof (auto)
  fix a assume a: a ∈ limit (w ⌢ x)
  have ∀ m. ∃ n. m<n ∧ x n = a
  proof
    fix m
    from a obtain n where m + length w < n ∧ (w ⌢ x) n = a
      by (auto simp add: limit-def Inf-many-def infinite-nat-iff-unbounded)
    hence m < n − length w ∧ x (n − length w) = a
      by (auto simp add: conc-def)
    thus ∃ n. m<n ∧ x n = a ..
  qed
  hence infinite {n . x n = a}
    by (simp add: infinite-nat-iff-unbounded)
  thus a ∈ limit x
    by (simp add: limit-def Inf-many-def)
next
  fix a assume a: a ∈ limit x
  have ∀ m. length w < m ⟶ (∃ n. m<n ∧ (w ⌢ x) n = a)
  proof (clarify)
    fix m
    assume m: length w < m
    with a obtain n where m − length w < n ∧ x n = a
      by (auto simp add: limit-def Inf-many-def infinite-nat-iff-unbounded)
    with m have m < n + length w ∧ (w ⌢ x) (n + length w) = a
      by (simp add: conc-def, arith)
    thus ∃ n. m<n ∧ (w ⌢ x) n = a ..
  qed
  hence infinite {n . (w ⌢ x) n = a}
    by (simp add: unbounded-k-infinite)
  thus a ∈ limit (w ⌢ x)
    by (simp add: limit-def Inf-many-def)
qed
```

theorem *limit-suffix* [*simp*]: *limit* (*suffix n x*) = *limit x*
proof –
  have $x = (prefix\ n\ x) \frown (suffix\ n\ x)$
    by (*simp add*: *prefix-suffix*)
  hence $limit\ x = limit\ (prefix\ n\ x \frown suffix\ n\ x)$
    by *simp*
  also have ... = *limit* (*suffix n x*)
    by (*rule limit-conc*)
  finally show *?thesis*
    by (*rule sym*)
qed

theorem *limit-iter* [*simp*]:
  assumes *nempty*: $0 < length\ w$
  shows $limit\ w^\omega = set\ w$
proof
  have $limit\ w^\omega \subseteq range\ w^\omega$
    by (*auto simp add*: *limit-def dest*: *INFM-EX*)
  also from *nempty* **have** $\ldots \subseteq set\ w$
    by *auto*
  finally show $limit\ w^\omega \subseteq set\ w$ **.**
next
  {
    fix *a* **assume** *a*: $a \in set\ w$
    then obtain *k* **where** *k*: $k < length\ w \wedge w!k = a$
      by (*auto simp add*: *set-conv-nth*)
    — the following bound is terrible, but it simplifies the proof
    from *nempty k* **have** $\forall m.\ w^\omega\ ((Suc\ m)*(length\ w) + k) = a$
      by (*simp add*: *mod-add-left-eq*)
    moreover
    — why is the following so hard to prove??
    have $\forall m.\ m < (Suc\ m)*(length\ w) + k$
    proof
      fix *m*
      from *nempty* **have** $1 \leq length\ w$ **by** *arith*
      hence $m*1 \leq m*length\ w$ **by** *simp*
      hence $m \leq m*length\ w$ **by** *simp*
      with *nempty* **have** $m < length\ w + (m*length\ w) + k$ **by** *arith*
      thus $m < (Suc\ m)*(length\ w) + k$ **by** *simp*
    qed
    moreover note *nempty*
    ultimately have $a \in limit\ w^\omega$
      by (*auto simp add*: *limit-iff-frequent INFM-nat*)
  }
  then show $set\ w \subseteq limit\ w^\omega$ **by** *auto*
qed

lemma *limit-o* [*simp*]:
  assumes *a*: $a \in limit\ w$

shows $f\ a \in limit\ (f \circ w)$
proof −
from a
have $\exists_\infty n.\ w\ n = a$
by (*simp add*: *limit-iff-frequent*)
hence $\exists_\infty n.\ f\ (w\ n) = f\ a$
by (*rule INFM-mono*, *simp*)
thus $f\ a \in limit\ (f \circ w)$
by (*simp add*: *limit-iff-frequent*)
qed

The converse relation is not true in general: $f(a)$ can be in the limit of $f \circ w$ even though a is not in the limit of w. However, *limit* commutes with renaming if the function is injective. More generally, if $f(a)$ is the image of only finitely many elements, some of these must be in the limit of w.

lemma *limit-o-inv*:
assumes *fin*: $finite\ (f\ -`\ \{x\})$
and x: $x \in limit\ (f \circ w)$
shows $\exists\, a \in (f\ -`\ \{x\}).\ a \in limit\ w$
proof (*rule ccontr*)
assume *contra*: $\neg$ *?thesis*
— hence, every element in the pre-image occurs only finitely often
then have $\forall\, a \in (f\ -`\ \{x\}).\ finite\ \{n.\ w\ n = a\}$
by (*simp add*: *limit-def Inf-many-def*)
— so there are only finitely many occurrences of any such element
with *fin* **have** $finite\ (\bigcup\ a \in (f\ -`\ \{x\}).\ \{n.\ w\ n = a\})$
by *auto*
— these are precisely those positions where x occurs in $f \circ w$
moreover
have $(\bigcup\ a \in (f\ -`\ \{x\}).\ \{n.\ w\ n = a\}) = \{n.\ f(w\ n) = x\}$
by *auto*
ultimately
— so x can occur only finitely often in the translated word
have $finite\ \{n.\ f(w\ n) = x\}$
by *simp*
— ... which yields a contradiction
with x **show** *False*
by (*simp add*: *limit-def Inf-many-def*)
qed

theorem *limit-inj* [*simp*]:
assumes *inj*: *inj f*
shows $limit\ (f \circ w) = f\ `\ (limit\ w)$
proof
show $f\ `\ limit\ w \subseteq limit\ (f \circ w)$
by *auto*
show $limit\ (f \circ w) \subseteq f\ `\ limit\ w$
proof
fix x

```
    assume x: x ∈ limit (f ∘ w)
    from inj have finite (f −‘ {x})
      by (blast intro: finite-vimageI)
    with x obtain a where a: a ∈ (f −‘ {x}) ∧ a ∈ limit w
      by (blast dest: limit-o-inv)
    thus x ∈ f ‘ (limit w)
      by auto
  qed
qed

lemma limit-inter-empty:
  assumes fin: finite (range w)
  assumes hyp: limit w ∩ S = {}
  shows ∀∞n. w n ∉ S
proof −
  from fin obtain k where k-def: limit w = range (suffix k w)
    using limit-is-suffix by blast
  have w (k + k′) ∉ S for k′
    using hyp unfolding k-def suffix-def image-def by blast
  thus ?thesis
    unfolding MOST-nat-le using le-Suc-ex by blast
qed
```

If the limit is the suffix of the sequence's range, we may increase the suffix index arbitrarily

```
lemma limit-range-suffix-incr:
  assumes limit r = range (suffix i r)
  assumes j≥i
  shows limit r = range (suffix j r)
    (is ?lhs = ?rhs)
proof −
  have ?lhs = range (suffix i r)
    using assms by simp
  moreover
  have . . . ⊇ ?rhs using ‹j≥i›
    by (metis (mono-tags, lifting) assms(2)
        image-subsetI le-Suc-ex range-eqI suffix-def suffix-suffix)
  moreover
  have . . . ⊇ ?lhs by (rule limit-in-range-suffix)
  ultimately
  show ?lhs = ?rhs
    by (metis antisym-conv limit-in-range-suffix)
qed
```

For two finite sequences, we can find a common suffix index such that the limits can be represented as these suffixes' ranges.

```
lemma common-range-limit:
  assumes finite (range x)
    and finite (range y)
```

obtains *i* **where** *limit x = range (suffix i x)*
and *limit y = range (suffix i y)*
proof −
obtain *i j* **where** *1*: *limit x = range (suffix i x)*
and *2*: *limit y = range (suffix j y)*
using *assms limit-is-suffix* **by** *metis*
have *limit x = range (suffix (max i j) x)*
and *limit y = range (suffix (max i j) y)*
using *limit-range-suffix-incr[OF 1] limit-range-suffix-incr[OF 2]*
by *auto*
thus *?thesis*
using *that* **by** *metis*
qed

68.5 Index sequences and piecewise definitions

A word can be defined piecewise: given a sequence of words $w_0, w_1, \ldots$ and a strictly increasing sequence of integers $i_0, i_1, \ldots$ where $i_0 = 0$, a single word is obtained by concatenating subwords of the w_n as given by the integers: the resulting word is

$$(w_0)_{i_0} \ldots (w_0)_{i_1-1}(w_1)_{i_1} \ldots (w_1)_{i_2-1} \ldots$$

We prepare the field by proving some trivial facts about such sequences of indexes.

definition *idx-sequence* :: *nat word* ⇒ *bool*
where *idx-sequence idx* ≡ (*idx 0 = 0*) ∧ (∀ *n. idx n < idx (Suc n)*)

lemma *idx-sequence-less*:
assumes *iseq*: *idx-sequence idx*
shows *idx n < idx (Suc(n+k))*
proof (*induct k*)
from *iseq* **show** *idx n < idx (Suc (n + 0))*
by (*simp add*: *idx-sequence-def*)
next
fix *k*
assume *ih*: *idx n < idx (Suc(n+k))*
from *iseq* **have** *idx (Suc(n+k)) < idx (Suc(n + Suc k))*
by (*simp add*: *idx-sequence-def*)
with *ih* **show** *idx n < idx (Suc(n + Suc k))*
by (*rule less-trans*)
qed

lemma *idx-sequence-inj*:
assumes *iseq*: *idx-sequence idx*
and *eq*: *idx m = idx n*
shows *m = n*
proof (*rule nat-less-cases*)
assume *n<m*

```
  then obtain k where m = Suc(n+k)
    by (auto simp add: less-iff-Suc-add)
  with iseq have idx n < idx m
    by (simp add: idx-sequence-less)
  with eq show ?thesis
    by simp
next
  assume m<n
  then obtain k where n = Suc(m+k)
    by (auto simp add: less-iff-Suc-add)
  with iseq have idx m < idx n
    by (simp add: idx-sequence-less)
  with eq show ?thesis
    by simp
qed (simp)

lemma idx-sequence-mono:
  assumes iseq: idx-sequence idx
    and m: m ≤ n
  shows idx m ≤ idx n
proof (cases m=n)
  case True
  thus ?thesis by simp
next
  case False
  with m have m < n by simp
  then obtain k where n = Suc(m+k)
    by (auto simp add: less-iff-Suc-add)
  with iseq have idx m < idx n
    by (simp add: idx-sequence-less)
  thus ?thesis by simp
qed
```

Given an index sequence, every natural number is contained in the interval defined by two adjacent indexes, and in fact this interval is determined uniquely.

```
lemma idx-sequence-idx:
  assumes idx-sequence idx
  shows idx k ∈ {idx k ..< idx (Suc k)}
using assms by (auto simp add: idx-sequence-def)

lemma idx-sequence-interval:
  assumes iseq: idx-sequence idx
  shows ∃k. n ∈ {idx k ..< idx (Suc k) }
    (is ?P n is ∃k. ?in n k)
proof (induct n)
  from iseq have 0 = idx 0
    by (simp add: idx-sequence-def)
  moreover
```

from *iseq* **have** *idx 0* ∈ {*idx 0* ..< *idx* (*Suc 0*) }
  by (*rule idx-sequence-idx*)
ultimately
show *?P 0* **by** *auto*
next
fix *n*
assume *?P n*
then obtain *k* **where** *k*: *?in n k* **..**
show *?P* (*Suc n*)
proof (*cases Suc n* < *idx* (*Suc k*))
  case *True*
  with *k* **have** *?in* (*Suc n*) *k*
    by *simp*
  thus *?thesis* **..**
next
  case *False*
  with *k* **have** *Suc n* = *idx* (*Suc k*)
    by *auto*
  with *iseq* **have** *?in* (*Suc n*) (*Suc k*)
    by (*simp add*: *idx-sequence-def*)
  thus *?thesis* **..**
qed
qed

lemma *idx-sequence-interval-unique*:
assumes *iseq*: *idx-sequence idx*
  and *k*: *n* ∈ {*idx k* ..< *idx* (*Suc k*)}
  and *m*: *n* ∈ {*idx m* ..< *idx* (*Suc m*)}
shows *k* = *m*
proof (*rule nat-less-cases*)
assume *k* < *m*
hence *Suc k* ≤ *m* **by** *simp*
with *iseq* **have** *idx* (*Suc k*) ≤ *idx m*
  by (*rule idx-sequence-mono*)
with *m* **have** *idx* (*Suc k*) ≤ *n*
  by *auto*
with *k* **have** *False*
  by *simp*
thus *?thesis* **..**
next
assume *m* < *k*
hence *Suc m* ≤ *k* **by** *simp*
with *iseq* **have** *idx* (*Suc m*) ≤ *idx k*
  by (*rule idx-sequence-mono*)
with *k* **have** *idx* (*Suc m*) ≤ *n*
  by *auto*
with *m* **have** *False*
  by *simp*
thus *?thesis* **..**

qed (*simp*)

lemma *idx-sequence-unique-interval*:
  assumes *iseq*: *idx-sequence idx*
  shows $\exists!$ *k*. $n \in \{idx\ k\ ..<\ idx\ (Suc\ k)\ \}$
proof (*rule ex-ex1I*)
  from *iseq* **show** $\exists k.\ n \in \{idx\ k\ ..<\ idx\ (Suc\ k)\}$
    by (*rule idx-sequence-interval*)
next
  fix *k y*
  assume $n \in \{idx\ k..<idx\ (Suc\ k)\}$ **and** $n \in \{idx\ y..<idx\ (Suc\ y)\}$
  with *iseq* **show** $k = y$ **by** (*auto elim*: *idx-sequence-interval-unique*)
qed

Now we can define the piecewise construction of a word using an index sequence.

definition *merge* :: *'a word word* $\Rightarrow$ *nat word* $\Rightarrow$ *'a word*
  where *merge ws idx* $\equiv \lambda n.$ *let* $i =$ *THE* $i.\ n \in \{idx\ i\ ..<\ idx\ (Suc\ i)\ \}$ *in ws i n*

lemma *merge*:
  assumes *idx*: *idx-sequence idx*
    and *n*: $n \in \{idx\ i\ ..<\ idx\ (Suc\ i)\}$
  shows *merge ws idx n* = *ws i n*
proof −
  from *n* **have** (*THE k*. $n \in \{idx\ k\ ..<\ idx\ (Suc\ k)\ \}) = i$
    by (*rule the-equality*[*OF* - *sym*[*OF idx-sequence-interval-unique*[*OF idx n*]]]) *simp*
  thus *?thesis*
    by (*simp add*: *merge-def Let-def*)
qed

lemma *merge0*:
  assumes *idx*: *idx-sequence idx*
  shows *merge ws idx 0* = *ws 0 0*
proof (*rule merge*[*OF idx*])
  from *idx* **have** *idx 0* < *idx* (*Suc 0*)
    unfolding *idx-sequence-def* **by** *blast*
  with *idx* **show** $0 \in \{idx\ 0\ ..<\ idx\ (Suc\ 0)\}$
    by (*simp add*: *idx-sequence-def*)
qed

lemma *merge-Suc*:
  assumes *idx*: *idx-sequence idx*
    and *n*: $n \in \{idx\ i\ ..<\ idx\ (Suc\ i)\}$
  shows *merge ws idx* (*Suc n*) = (*if Suc n* = *idx* (*Suc i*) *then ws* (*Suc i*) *else ws i*) (*Suc n*)
proof *auto*
  assume *eq*: *Suc n* = *idx* (*Suc i*)
  from *idx* **have** *idx* (*Suc i*) < *idx* (*Suc*(*Suc i*))

unfolding *idx-sequence-def* **by** *blast*
with *eq idx* **show** *merge ws idx* (*idx* (*Suc i*)) = *ws* (*Suc i*) (*idx* (*Suc i*))
by (*simp add*: *merge*)
next
assume *neq*: *Suc n* ≠ *idx* (*Suc i*)
with *n* **have** *Suc n* ∈ {*idx i* ..< *idx* (*Suc i*) }
by *auto*
with *idx* **show** *merge ws idx* (*Suc n*) = *ws i* (*Suc n*)
by (*rule merge*)
qed

end

69 Canonical order on option type

theory *Option-ord*
imports *Option Main*
begin

notation
bot (⊥) **and**
top (⊤) **and**
inf (**infixl** ⊓ *70*) **and**
sup (**infixl** ⊔ *65*) **and**
Inf (⨅- [*900*] *900*) **and**
Sup (⨆- [*900*] *900*)

syntax
-INF1 :: *pttrns* ⇒ *'b* ⇒ *'b* ((*3*⨅-./ -) [*0*, *10*] *10*)
-INF :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b* ((*3*⨅-∈-./ -) [*0*, *0*, *10*] *10*)
-SUP1 :: *pttrns* ⇒ *'b* ⇒ *'b* ((*3*⨆-./ -) [*0*, *10*] *10*)
-SUP :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b* ((*3*⨆-∈-./ -) [*0*, *0*, *10*] *10*)

instantiation *option* :: (*preorder*) *preorder*
begin

definition *less-eq-option* **where**
$x \leq y \longleftrightarrow$ (*case x of None* ⇒ *True* | *Some x* ⇒ (*case y of None* ⇒ *False* | *Some y* ⇒ $x \leq y$))

definition *less-option* **where**
$x < y \longleftrightarrow$ (*case y of None* ⇒ *False* | *Some y* ⇒ (*case x of None* ⇒ *True* | *Some x* ⇒ $x < y$))

lemma *less-eq-option-None* [*simp*]: *None* ≤ *x*
by (*simp add*: *less-eq-option-def*)

lemma *less-eq-option-None-code* [*code*]: *None* $\leq$ *x* $\longleftrightarrow$ *True*
by *simp*

lemma *less-eq-option-None-is-None*: *x* $\leq$ *None* $\Longrightarrow$ *x* = *None*
by (*cases x*) (*simp-all add*: *less-eq-option-def*)

lemma *less-eq-option-Some-None* [*simp*, *code*]: *Some x* $\leq$ *None* $\longleftrightarrow$ *False*
by (*simp add*: *less-eq-option-def*)

lemma *less-eq-option-Some* [*simp*, *code*]: *Some x* $\leq$ *Some y* $\longleftrightarrow$ *x* $\leq$ *y*
by (*simp add*: *less-eq-option-def*)

lemma *less-option-None* [*simp*, *code*]: *x* < *None* $\longleftrightarrow$ *False*
by (*simp add*: *less-option-def*)

lemma *less-option-None-is-Some*: *None* < *x* $\Longrightarrow$ $\exists$ *z*. *x* = *Some z*
by (*cases x*) (*simp-all add*: *less-option-def*)

lemma *less-option-None-Some* [*simp*]: *None* < *Some x*
by (*simp add*: *less-option-def*)

lemma *less-option-None-Some-code* [*code*]: *None* < *Some x* $\longleftrightarrow$ *True*
by *simp*

lemma *less-option-Some* [*simp*, *code*]: *Some x* < *Some y* $\longleftrightarrow$ *x* < *y*
by (*simp add*: *less-option-def*)

instance
by *standard*
(*auto simp add*: *less-eq-option-def less-option-def less-le-not-le*
elim: *order-trans split*: *option.splits*)

end

instance *option* :: (*order*) *order*
by *standard* (*auto simp add*: *less-eq-option-def less-option-def split*: *option.splits*)

instance *option* :: (*linorder*) *linorder*
by *standard* (*auto simp add*: *less-eq-option-def less-option-def split*: *option.splits*)

instantiation *option* :: (*order*) *order-bot*
begin

definition *bot-option* **where** $\bot$ = *None*

instance
by *standard* (*simp add*: *bot-option-def*)

end

instantiation *option* :: (*order-top*) *order-top*
begin

definition *top-option* **where** $\top = Some\ \top$

instance
  by *standard* (*simp add*: *top-option-def less-eq-option-def split*: *option.split*)

end

instance *option* :: (*wellorder*) *wellorder*
proof
  fix P :: *'a option* $\Rightarrow$ *bool*
  fix z :: *'a option*
  assume H: $\bigwedge x.\ (\bigwedge y.\ y < x \Longrightarrow P\ y) \Longrightarrow P\ x$
  have *P None* **by** (*rule H*) *simp*
  then have *P-Some* [*case-names Some*]: $P\ z$ **if** $\bigwedge x.\ z = Some\ x \Longrightarrow (P\ o\ Some)\ x$ **for** z
    using ‹*P None*› *that* **by** (*cases z*) *simp-all*
  show *P z*
  proof (*cases z rule*: *P-Some*)
    case (*Some w*)
    show (*P o Some*) *w*
    proof (*induct rule*: *less-induct*)
      case (*less x*)
      have *P* (*Some x*)
      proof (*rule H*)
        fix y :: *'a option*
        assume $y < Some\ x$
        show *P y*
        proof (*cases y rule*: *P-Some*)
          case (*Some v*)
          with ‹$y < Some\ x$› **have** $v < x$ **by** *simp*
          with *less* **show** (*P o Some*) *v* **.**
        qed
      qed
      then show *?case* **by** *simp*
    qed
  qed
qed

instantiation *option* :: (*inf*) *inf*
begin

definition *inf-option* **where**
  $x \sqcap y = (case\ x\ of\ None \Rightarrow None \mid Some\ x \Rightarrow (case\ y\ of\ None \Rightarrow None \mid Some\ y \Rightarrow Some\ (x \sqcap y)))$

lemma *inf-None-1* [*simp*, *code*]: *None* ⊓ *y* = *None*
  by (*simp add*: *inf-option-def*)

lemma *inf-None-2* [*simp*, *code*]: *x* ⊓ *None* = *None*
  by (*cases x*) (*simp-all add*: *inf-option-def*)

lemma *inf-Some* [*simp*, *code*]: *Some x* ⊓ *Some y* = *Some* (*x* ⊓ *y*)
  by (*simp add*: *inf-option-def*)

instance ..

end

instantiation *option* :: (*sup*) *sup*
begin

definition *sup-option* **where**
  x ⊔ *y* = (*case x of None* ⇒ *y* | *Some x'* ⇒ (*case y of None* ⇒ *x* | *Some y* ⇒ *Some* (*x'* ⊔ *y*)))

lemma *sup-None-1* [*simp*, *code*]: *None* ⊔ *y* = *y*
  by (*simp add*: *sup-option-def*)

lemma *sup-None-2* [*simp*, *code*]: *x* ⊔ *None* = *x*
  by (*cases x*) (*simp-all add*: *sup-option-def*)

lemma *sup-Some* [*simp*, *code*]: *Some x* ⊔ *Some y* = *Some* (*x* ⊔ *y*)
  by (*simp add*: *sup-option-def*)

instance ..

end

instance *option* :: (*semilattice-inf*) *semilattice-inf*
proof
  fix *x y z* :: *'a option*
  show *x* ⊓ *y* ≤ *x*
    by (*cases x*, *simp-all*, *cases y*, *simp-all*)
  show *x* ⊓ *y* ≤ *y*
    by (*cases x*, *simp-all*, *cases y*, *simp-all*)
  show *x* ≤ *y* ⟹ *x* ≤ *z* ⟹ *x* ≤ *y* ⊓ *z*
    by (*cases x*, *simp-all*, *cases y*, *simp-all*, *cases z*, *simp-all*)
qed

instance *option* :: (*semilattice-sup*) *semilattice-sup*
proof
  fix *x y z* :: *'a option*
  show *x* ≤ *x* ⊔ *y*

by (*cases x*, *simp-all*, *cases y*, *simp-all*)
show $y \leq x \sqcup y$
by (*cases x*, *simp-all*, *cases y*, *simp-all*)
fix $x\ y\ z$ :: *'a option*
show $y \leq x \Longrightarrow z \leq x \Longrightarrow y \sqcup z \leq x$
by (*cases y*, *simp-all*, *cases z*, *simp-all*, *cases x*, *simp-all*)
qed

instance *option* :: (*lattice*) *lattice* **..**

instance *option* :: (*lattice*) *bounded-lattice-bot* **..**

instance *option* :: (*bounded-lattice-top*) *bounded-lattice-top* **..**

instance *option* :: (*bounded-lattice-top*) *bounded-lattice* **..**

instance *option* :: (*distrib-lattice*) *distrib-lattice*
proof
fix $x\ y\ z$ :: *'a option*
show $x \sqcup y \sqcap z = (x \sqcup y) \sqcap (x \sqcup z)$
by (*cases x*, *simp-all*, *cases y*, *simp-all*, *cases z*, *simp-all add*: *sup-inf-distrib1 inf-commute*)
qed

instantiation *option* :: (*complete-lattice*) *complete-lattice*
begin

definition *Inf-option* :: *'a option set* ⇒ *'a option* **where**
$\bigsqcap A$ = (*if None* ∈ *A then None else Some* ($\bigsqcap$ *Option.these A*))

lemma *None-in-Inf* [*simp*]: *None* ∈ *A* ⟹ $\bigsqcap A$ = *None*
by (*simp add*: *Inf-option-def*)

definition *Sup-option* :: *'a option set* ⇒ *'a option* **where**
$\bigsqcup A$ = (*if A* = {} ∨ *A* = {*None*} *then None else Some* ($\bigsqcup$ *Option.these A*))

lemma *empty-Sup* [*simp*]: $\bigsqcup${} = *None*
by (*simp add*: *Sup-option-def*)

lemma *singleton-None-Sup* [*simp*]: $\bigsqcup${*None*} = *None*
by (*simp add*: *Sup-option-def*)

instance
proof
fix *x* :: *'a option* **and** *A*
assume *x* ∈ *A*
then show $\bigsqcap A \leq x$
by (*cases x*) (*auto simp add*: *Inf-option-def in-these-eq intro*: *Inf-lower*)
next

fix z :: $'a$ *option* **and** A
assume $*$: $\bigwedge x.\ x \in A \Longrightarrow z \le x$
show $z \le \bigsqcap A$
proof (*cases z*)
case *None* **then show** *?thesis* **by** *simp*
next
case (*Some y*)
show *?thesis*
by (*auto simp add*: *Inf-option-def in-these-eq Some intro*!: *Inf-greatest dest*!:
$*$)
qed
next
fix x :: $'a$ *option* **and** A
assume $x \in A$
then show $x \le \bigsqcup A$
by (*cases x*) (*auto simp add*: *Sup-option-def in-these-eq intro*: *Sup-upper*)
next
fix z :: $'a$ *option* **and** A
assume $*$: $\bigwedge x.\ x \in A \Longrightarrow x \le z$
show $\bigsqcup A \le z$
proof (*cases z*)
case *None*
with $*$ **have** $\bigwedge x.\ x \in A \Longrightarrow x = None$ **by** (*auto dest*: *less-eq-option-None-is-None*)
then have $A = \{\} \vee A = \{None\}$ **by** *blast*
then show *?thesis* **by** (*simp add*: *Sup-option-def*)
next
case (*Some y*)
from $*$ **have** $\bigwedge w.\ Some\ w \in A \Longrightarrow Some\ w \le z$ **.**
with *Some* **have** $\bigwedge w.\ w \in Option.these\ A \Longrightarrow w \le y$
by (*simp add*: *in-these-eq*)
then have $\bigsqcup Option.these\ A \le y$ **by** (*rule Sup-least*)
with *Some* **show** *?thesis* **by** (*simp add*: *Sup-option-def*)
qed
next
show $\bigsqcup \{\} = (\bot::'a\ option)$
by (*auto simp*: *bot-option-def*)
show $\bigsqcap \{\} = (\top::'a\ option)$
by (*auto simp*: *top-option-def Inf-option-def*)
qed

end

lemma *Some-Inf*:
$Some\ (\bigsqcap A) = \bigsqcap (Some\ `\ A)$
by (*auto simp add*: *Inf-option-def*)

lemma *Some-Sup*:
$A \ne \{\} \Longrightarrow Some\ (\bigsqcup A) = \bigsqcup (Some\ `\ A)$
by (*auto simp add*: *Sup-option-def*)

lemma *Some-INF*:
  Some ($\bigsqcap x{\in}A.\ f\ x$) = ($\bigsqcap x{\in}A.$ *Some* ($f\ x$))
  using *Some-Inf* [*of f ' A*] **by** (*simp add*: *comp-def*)

lemma *Some-SUP*:
  $A \neq \{\} \Longrightarrow$ *Some* ($\bigsqcup x{\in}A.\ f\ x$) = ($\bigsqcup x{\in}A.$ *Some* ($f\ x$))
  using *Some-Sup* [*of f ' A*] **by** (*simp add*: *comp-def*)

instance *option* :: (*complete-distrib-lattice*) *complete-distrib-lattice*
proof
  fix *a* :: *'a option* **and** *B*
  show $a \sqcup \bigsqcap B = (\bigsqcap b{\in}B.\ a \sqcup b)$
  proof (*cases a*)
    case *None*
    then show *?thesis* **by** *simp*
  next
    case (*Some c*)
    show *?thesis*
    proof (*cases None* $\in$ *B*)
      case *True*
      then have *Some c* = ($\bigsqcap b{\in}B.$ *Some c* $\sqcup b$)
        by (*auto intro*!: *antisym INF-lower2 INF-greatest*)
      with *True Some* **show** *?thesis* **by** *simp*
    next
      case *False* **then have** *B*: {$x \in B.\ \exists y.\ x =$ *Some y*} = *B* **by** *auto* (*metis not-Some-eq*)
      from *sup-Inf* **have** *Some c* $\sqcup$ *Some* ($\bigsqcap$ *Option.these B*) = *Some* ($\bigsqcap b{\in}$*Option.these B*. $c \sqcup b$) **by** *simp*
      then have *Some c* $\sqcup \bigsqcap$(*Some ' Option.these B*) = ($\bigsqcap x{\in}$*Some ' Option.these B*. *Some c* $\sqcup x$)
        by (*simp add*: *Some-INF Some-Inf comp-def*)
      with *Some B* **show** *?thesis* **by** (*simp add*: *Some-image-these-eq cong del*: *strong-INF-cong*)
    qed
  qed
  show $a \sqcap \bigsqcup B = (\bigsqcup b{\in}B.\ a \sqcap b)$
  proof (*cases a*)
    case *None*
    then show *?thesis* **by** (*simp add*: *image-constant-conv bot-option-def cong del*: *strong-SUP-cong*)
  next
    case (*Some c*)
    show *?thesis*
    proof (*cases B* = {} $\vee$ *B* = {*None*})
      case *True*
      then show *?thesis* **by** *auto*
    next
      have *B*: *B* = {$x \in B.\ \exists y.\ x =$ *Some y*} $\cup$ {$x \in B.\ x =$ *None*}

by *auto*
then have *Sup-B*: $\bigsqcup B = \bigsqcup(\{x \in B.\ \exists y.\ x = Some\ y\} \cup \{x \in B.\ x = None\})$
and *SUP-B*: $\bigwedge f.\ (\bigsqcup x \in B.\ f\ x) = (\bigsqcup x \in \{x \in B.\ \exists y.\ x = Some\ y\} \cup \{x \in B.\ x = None\}.\ f\ x)$
by *simp-all*
have *Sup-None*: $\bigsqcup\{x.\ x = None \wedge x \in B\} = None$
by (*simp add*: *bot-option-def* [*symmetric*])
have *SUP-None*: $(\bigsqcup x{\in}\{x.\ x = None \wedge x \in B\}.\ Some\ c \sqcap x) = None$
by (*simp add*: *bot-option-def* [*symmetric*])
case *False* **then have** $Option.these\ B \neq \{\}$ **by** (*simp add*: *these-not-empty-eq*)
moreover from *inf-Sup* **have** $Some\ c \sqcap Some\ (\bigsqcup Option.these\ B) = Some\ (\bigsqcup b{\in}Option.these\ B.\ c \sqcap b)$
by *simp*
ultimately have $Some\ c \sqcap \bigsqcup(Some\ `\ Option.these\ B) = (\bigsqcup x{\in}Some\ `\ Option.these\ B.\ Some\ c \sqcap x)$
by (*simp add*: *Some-SUP Some-Sup comp-def*)
with *Some* **show** *?thesis*
by (*simp add*: *Some-image-these-eq Sup-B SUP-B Sup-None SUP-None SUP-union Sup-union-distrib cong del*: *strong-SUP-cong*)
qed
qed
qed

instance *option* :: (*complete-linorder*) *complete-linorder* **..**

no-notation
bot (⊥) **and**
top (⊤) **and**
inf (**infixl** ⊓ *70*) **and**
sup (**infixl** ⊔ *65*) **and**
Inf (⨅- [*900*] *900*) **and**
Sup (⨆- [*900*] *900*)

no-syntax
-INF1 :: *pttrns* ⇒ *'b* ⇒ *'b* ((*3*⨅-./ -) [*0*, *10*] *10*)
-INF :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b* ((*3*⨅-∈-./ -) [*0*, *0*, *10*] *10*)
-SUP1 :: *pttrns* ⇒ *'b* ⇒ *'b* ((*3*⨆-./ -) [*0*, *10*] *10*)
-SUP :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b* ((*3*⨆-∈-./ -) [*0*, *0*, *10*] *10*)

end

70 Futures and parallel lists for code generated towards Isabelle/ML

theory *Parallel*
imports *Main*

begin

70.1 Futures

datatype *'a future* = *fork unit* ⇒ *'a*

primrec *join* :: *'a future* ⇒ *'a* **where**
join (*fork f*) = *f* ()

lemma *future-eqI* [*intro*!]:
assumes *join f* = *join g*
shows *f* = *g*
using *assms* **by** (*cases f*, *cases g*) (*simp add*: *ext*)

code-printing
type-constructor *future* ⇀ (*Eval*) - *future*
| **constant** *fork* ⇀ (*Eval*) *Future.fork*
| **constant** *join* ⇀ (*Eval*) *Future.join*

code-reserved *Eval Future future*

70.2 Parallel lists

definition *map* :: (*'a* ⇒ *'b*) ⇒ *'a list* ⇒ *'b list* **where**
[*simp*]: *map* = *List.map*

definition *forall* :: (*'a* ⇒ *bool*) ⇒ *'a list* ⇒ *bool* **where**
forall = *list-all*

lemma *forall-all* [*simp*]:
forall P xs ⟷ (∀ *x*∈*set xs*. *P x*)
by (*simp add*: *forall-def list-all-iff*)

definition *exists* :: (*'a* ⇒ *bool*) ⇒ *'a list* ⇒ *bool* **where**
exists = *list-ex*

lemma *exists-ex* [*simp*]:
exists P xs ⟷ (∃ *x*∈*set xs*. *P x*)
by (*simp add*: *exists-def list-ex-iff*)

code-printing
constant *map* ⇀ (*Eval*) *Par'-List.map*
| **constant** *forall* ⇀ (*Eval*) *Par'-List.forall*
| **constant** *exists* ⇀ (*Eval*) *Par'-List.exists*

code-reserved *Eval Par-List*

hide-const (**open**) *fork join map exists forall*

end

71 Permutations

theory *Permutation*
imports *Multiset*
begin

inductive *perm* :: *'a list ⇒ 'a list ⇒ bool* (- <~~> - [*50*, *50*] *50*)
where
Nil [*intro*!]: [] <~~> []
| *swap* [*intro*!]: *y # x # l* <~~> *x # y # l*
| *Cons* [*intro*!]: *xs* <~~> *ys* ⟹ *z # xs* <~~> *z # ys*
| *trans* [*intro*]: *xs* <~~> *ys* ⟹ *ys* <~~> *zs* ⟹ *xs* <~~> *zs*

proposition *perm-refl* [*iff*]: *l* <~~> *l*
by (*induct l*) *auto*

71.1 Some examples of rule induction on permutations

proposition *xperm-empty-imp*: [] <~~> *ys* ⟹ *ys* = []
by (*induct xs* == [] :: *'a list ys pred*: *perm*) *simp-all*

This more general theorem is easier to understand!

proposition *perm-length*: *xs* <~~> *ys* ⟹ *length xs* = *length ys*
by (*induct pred*: *perm*) *simp-all*

proposition *perm-empty-imp*: [] <~~> *xs* ⟹ *xs* = []
by (*drule perm-length*) *auto*

proposition *perm-sym*: *xs* <~~> *ys* ⟹ *ys* <~~> *xs*
by (*induct pred*: *perm*) *auto*

71.2 Ways of making new permutations

We can insert the head anywhere in the list.

proposition *perm-append-Cons*: *a # xs @ ys* <~~> *xs @ a # ys*
by (*induct xs*) *auto*

proposition *perm-append-swap*: *xs @ ys* <~~> *ys @ xs*
by (*induct xs*) (*auto intro*: *perm-append-Cons*)

proposition *perm-append-single*: *a # xs* <~~> *xs @* [*a*]
by (*rule perm.trans* [*OF* - *perm-append-swap*]) *simp*

proposition *perm-rev*: *rev xs* <~~> *xs*
by (*induct xs*) (*auto intro*!: *perm-append-single intro*: *perm-sym*)

proposition *perm-append1*: *xs* <~~> *ys* ⟹ *l* @ *xs* <~~> *l* @ *ys*
by (*induct l*) *auto*

proposition *perm-append2*: *xs* <~~> *ys* ⟹ *xs* @ *l* <~~> *ys* @ *l*
by (*blast intro*!: *perm-append-swap perm-append1*)

71.3 Further results

proposition *perm-empty* [*iff*]: [] <~~> *xs* ⟷ *xs* = []
by (*blast intro*: *perm-empty-imp*)

proposition *perm-empty2* [*iff*]: *xs* <~~> [] ⟷ *xs* = []
apply *auto*
apply (*erule perm-sym* [*THEN perm-empty-imp*])
done

proposition *perm-sing-imp*: *ys* <~~> *xs* ⟹ *xs* = [*y*] ⟹ *ys* = [*y*]
by (*induct pred*: *perm*) *auto*

proposition *perm-sing-eq* [*iff*]: *ys* <~~> [*y*] ⟷ *ys* = [*y*]
by (*blast intro*: *perm-sing-imp*)

proposition *perm-sing-eq2* [*iff*]: [*y*] <~~> *ys* ⟷ *ys* = [*y*]
by (*blast dest*: *perm-sym*)

71.4 Removing elements

proposition *perm-remove*: *x* ∈ *set ys* ⟹ *ys* <~~> *x* # *remove1 x ys*
by (*induct ys*) *auto*

Congruence rule

proposition *perm-remove-perm*: *xs* <~~> *ys* ⟹ *remove1 z xs* <~~> *remove1 z ys*
by (*induct pred*: *perm*) *auto*

proposition *remove-hd* [*simp*]: *remove1 z* (*z* # *xs*) = *xs*
by *auto*

proposition *cons-perm-imp-perm*: *z* # *xs* <~~> *z* # *ys* ⟹ *xs* <~~> *ys*
by (*drule-tac z* = *z* **in** *perm-remove-perm*) *auto*

proposition *cons-perm-eq* [*iff*]: *z*#*xs* <~~> *z*#*ys* ⟷ *xs* <~~> *ys*
by (*blast intro*: *cons-perm-imp-perm*)

proposition *append-perm-imp-perm*: *zs* @ *xs* <~~> *zs* @ *ys* ⟹ *xs* <~~> *ys*
by (*induct zs arbitrary*: *xs ys rule*: *rev-induct*) *auto*

proposition *perm-append1-eq* [*iff*]: *zs* @ *xs* <~~> *zs* @ *ys* ⟷ *xs* <~~> *ys*
by (*blast intro*: *append-perm-imp-perm perm-append1*)

proposition *perm-append2-eq* [*iff*]: *xs* @ *zs* <~~> *ys* @ *zs* ⟷ *xs* <~~> *ys*
apply (*safe intro*!: *perm-append2*)
apply (*rule append-perm-imp-perm*)
apply (*rule perm-append-swap* [*THEN perm.trans*])
— the previous step helps this *blast* call succeed quickly
apply (*blast intro*: *perm-append-swap*)
done

theorem *mset-eq-perm*: *mset xs* = *mset ys* ⟷ *xs* <~~> *ys*
apply (*rule iffI*)
apply (*erule-tac* [*2*] *perm.induct*)
apply (*simp-all add*: *union-ac*)
apply (*erule rev-mp*)
apply (*rule-tac x*=*ys* **in** *spec*)
apply (*induct-tac xs*)
apply *auto*
apply (*erule-tac x* = *remove1 a x* **in** *allE*)
apply (*drule sym*)
apply *simp*
apply (*subgoal-tac a* ∈ *set x*)
apply (*drule-tac z* = *a* **in** *perm.Cons*)
apply (*erule perm.trans*)
apply (*rule perm-sym*)
apply (*erule perm-remove*)
apply (*drule-tac f*=*set-mset* **in** *arg-cong*)
apply *simp*
done

proposition *mset-le-perm-append*: *mset xs* ≤# *mset ys* ⟷ (∃ *zs*. *xs* @ *zs* <~~> *ys*)
apply (*auto simp*: *mset-eq-perm*[*THEN sym*] *mset-le-exists-conv*)
apply (*insert surj-mset*)
apply (*drule surjD*)
apply (*blast intro*: *sym*)+
done

proposition *perm-set-eq*: *xs* <~~> *ys* ⟹ *set xs* = *set ys*
by (*metis mset-eq-perm mset-eq-setD*)

proposition *perm-distinct-iff*: *xs* <~~> *ys* ⟹ *distinct xs* = *distinct ys*
apply (*induct pred*: *perm*)
apply *simp-all*
apply *fastforce*
apply (*metis perm-set-eq*)
done

theorem *eq-set-perm-remdups*: *set xs* = *set ys* ⟹ *remdups xs* <~~> *remdups ys*
apply (*induct xs arbitrary*: *ys rule*: *length-induct*)

apply (*case-tac remdups xs*)
apply *simp-all*
apply (*subgoal-tac a* ∈ *set* (*remdups ys*))
prefer *2* **apply** (*metis list.set*(*2*) *insert-iff set-remdups*)
apply (*drule split-list*) **apply** (*elim exE conjE*)
apply (*drule-tac x* = *list* **in** *spec*) **apply** (*erule impE*) **prefer** *2*
apply (*drule-tac x* = *ysa* @ *zs* **in** *spec*) **apply** (*erule impE*) **prefer** *2*
apply *simp*
apply (*subgoal-tac a* # *list* <~~> *a* # *ysa* @ *zs*)
apply (*metis Cons-eq-appendI perm-append-Cons trans*)
apply (*metis Cons Cons-eq-appendI distinct.simps*(*2*)
distinct-remdups distinct-remdups-id perm-append-swap perm-distinct-iff)
apply (*subgoal-tac set* (*a* # *list*) =
set (*ysa* @ *a* # *zs*) ∧ *distinct* (*a* # *list*) ∧ *distinct* (*ysa* @ *a* # *zs*))
apply (*fastforce simp add*: *insert-ident*)
apply (*metis distinct-remdups set-remdups*)
apply (*subgoal-tac length* (*remdups xs*) < *Suc* (*length xs*))
apply *simp*
apply (*subgoal-tac length* (*remdups xs*) ≤ *length xs*)
apply *simp*
apply (*rule length-remdups-leq*)
done

proposition *perm-remdups-iff-eq-set*: *remdups x* <~~> *remdups y* ⟷ *set x* = *set y*
by (*metis List.set-remdups perm-set-eq eq-set-perm-remdups*)

theorem *permutation-Ex-bij*:
assumes *xs* <~~> *ys*
shows ∃*f*. *bij-betw f* {..<*length xs*} {..<*length ys*} ∧ (∀ *i*<*length xs*. *xs* ! *i* = *ys* ! (*f i*))
using *assms*
proof *induct*
case *Nil*
then show *?case*
unfolding *bij-betw-def* **by** *simp*
next
case (*swap y x l*)
show *?case*
proof (*intro exI*[*of* - *Fun.swap 0 1 id*] *conjI allI impI*)
show *bij-betw* (*Fun.swap 0 1 id*) {..<*length* (*y* # *x* # *l*)} {..<*length* (*x* # *y* # *l*)}
by (*auto simp*: *bij-betw-def*)
fix *i*
assume *i* < *length* (*y* # *x* # *l*)
show (*y* # *x* # *l*) ! *i* = (*x* # *y* # *l*) ! (*Fun.swap 0 1 id*) *i*
by (*cases i*) (*auto simp*: *Fun.swap-def gr0-conv-Suc*)
qed
next

```
  case (Cons xs ys z)
  then obtain f where bij: bij-betw f {..<length xs} {..<length ys}
    and perm: ∀ i<length xs. xs ! i = ys ! (f i)
    by blast
  let ?f = λi. case i of Suc n ⇒ Suc (f n) | 0 ⇒ 0
  show ?case
  proof (intro exI[of - ?f] allI conjI impI)
    have *: {..<length (z#xs)} = {0} ∪ Suc ‘ {..<length xs}
            {..<length (z#ys)} = {0} ∪ Suc ‘ {..<length ys}
      by (simp-all add: lessThan-Suc-eq-insert-0)
    show bij-betw ?f {..<length (z#xs)} {..<length (z#ys)}
      unfolding *
    proof (rule bij-betw-combine)
      show bij-betw ?f (Suc ‘ {..<length xs}) (Suc ‘ {..<length ys})
        using bij unfolding bij-betw-def
        by (auto intro!: inj-onI imageI dest: inj-onD simp: image-comp comp-def)
    qed (auto simp: bij-betw-def)
    fix i
    assume i < length (z # xs)
    then show (z # xs) ! i = (z # ys) ! (?f i)
      using perm by (cases i) auto
  qed
next
  case (trans xs ys zs)
  then obtain f g
    where bij: bij-betw f {..<length xs} {..<length ys} bij-betw g {..<length ys}
{..<length zs}
    and perm: ∀ i<length xs. xs ! i = ys ! (f i) ∀ i<length ys. ys ! i = zs ! (g i)
    by blast
  show ?case
  proof (intro exI[of - g ∘ f] conjI allI impI)
    show bij-betw (g ∘ f) {..<length xs} {..<length zs}
      using bij by (rule bij-betw-trans)
    fix i
    assume i < length xs
    with bij have f i < length ys
      unfolding bij-betw-def by force
    with ‹i < length xs› show xs ! i = zs ! (g ∘ f) i
      using trans(1,3)[THEN perm-length] perm by auto
  qed
qed

proposition perm-finite: finite {B. B <~~> A}
proof (rule finite-subset[where B={xs. set xs ⊆ set A ∧ length xs ≤ length A}])
 show finite {xs. set xs ⊆ set A ∧ length xs ≤ length A}
   apply (cases A, simp)
   apply (rule card-ge-0-finite)
   apply (auto simp: card-lists-length-le)
   done
```

next
 show {*B*. *B* <~~> *A*} ⊆ {*xs*. *set xs* ⊆ *set A* ∧ *length xs* ≤ *length A*}
  by (*clarsimp simp add*: *perm-length perm-set-eq*)
qed

proposition *perm-swap*:
  assumes *i* < *length xs j* < *length xs*
  shows *xs*[*i* := *xs* ! *j*, *j* := *xs* ! *i*] <~~> *xs*
 using *assms* **by** (*simp add*: *mset-eq-perm*[*symmetric*] *mset-swap*)

end

72 Permutations, both general and specifically on finite sets.

theory *Permutations*
imports *Binomial*
begin

72.1 Transpositions

lemma *swap-id-idempotent* [*simp*]:
 Fun.swap a b id ∘ *Fun.swap a b id* = *id*
 by (*rule ext*, *auto simp add*: *Fun.swap-def*)

lemma *inv-swap-id*:
 inv (*Fun.swap a b id*) = *Fun.swap a b id*
 by (*rule inv-unique-comp*) *simp-all*

lemma *swap-id-eq*:
 Fun.swap a b id x = (*if x* = *a then b else if x* = *b then a else x*)
 by (*simp add*: *Fun.swap-def*)

72.2 Basic consequences of the definition

definition *permutes* (**infixr** *permutes 41*)
 where (*p permutes S*) ⟷ (∀ *x*. *x* ∉ *S* ⟶ *p x* = *x*) ∧ (∀ *y*. ∃!*x*. *p x* = *y*)

lemma *permutes-in-image*: *p permutes S* ⟹ *p x* ∈ *S* ⟷ *x* ∈ *S*
 unfolding *permutes-def* **by** *metis*

lemma *permutes-image*: *p permutes S* ⟹ *p* ` *S* = *S*
 unfolding *permutes-def*
 apply (*rule set-eqI*)
 apply (*simp add*: *image-iff*)
 apply *metis*
 done

lemma *permutes-inj*: *p permutes S* ⟹ *inj p*

unfolding *permutes-def inj-on-def* **by** *blast*

lemma *permutes-surj*: p *permutes* $s \Longrightarrow$ *surj* p
unfolding *permutes-def surj-def* **by** *metis*

lemma *permutes-bij*: p *permutes* $s \Longrightarrow$ *bij* p
unfolding *bij-def* **by** (*metis permutes-inj permutes-surj*)

lemma *permutes-imp-bij*: p *permutes* $S \Longrightarrow$ *bij-betw* p S S
by (*metis UNIV-I bij-betw-subset permutes-bij permutes-image subsetI*)

lemma *bij-imp-permutes*: *bij-betw* p S $S \Longrightarrow (\bigwedge x.\ x \notin S \Longrightarrow p\ x = x) \Longrightarrow p$ *permutes* S
unfolding *permutes-def bij-betw-def inj-on-def*
by *auto* (*metis image-iff*)+

lemma *permutes-inv-o*:
assumes *pS*: p *permutes* S
shows $p \circ$ *inv* p = *id*
and *inv* $p \circ p$ = *id*
using *permutes-inj*[*OF pS*] *permutes-surj*[*OF pS*]
unfolding *inj-iff*[*symmetric*] *surj-iff*[*symmetric*] **by** *blast*+

lemma *permutes-inverses*:
fixes $p :: 'a \Rightarrow 'a$
assumes *pS*: p *permutes* S
shows p (*inv* p x) = x
and *inv* p (p x) = x
using *permutes-inv-o*[*OF pS*, *unfolded fun-eq-iff o-def*] **by** *auto*

lemma *permutes-subset*: p *permutes* $S \Longrightarrow S \subseteq T \Longrightarrow p$ *permutes* T
unfolding *permutes-def* **by** *blast*

lemma *permutes-empty*[*simp*]: p *permutes* $\{\} \longleftrightarrow p = id$
unfolding *fun-eq-iff permutes-def* **by** *simp metis*

lemma *permutes-sing*[*simp*]: p *permutes* $\{a\} \longleftrightarrow p = id$
unfolding *fun-eq-iff permutes-def* **by** *simp metis*

lemma *permutes-univ*: p *permutes* *UNIV* $\longleftrightarrow (\forall y.\ \exists!x.\ p\ x = y)$
unfolding *permutes-def* **by** *simp*

lemma *permutes-inv-eq*: p *permutes* $S \Longrightarrow$ *inv* $p\ y = x \longleftrightarrow p\ x = y$
unfolding *permutes-def inv-def*
apply *auto*
apply (*erule allE*[**where** *x=y*])
apply (*erule allE*[**where** *x=y*])
apply (*rule someI-ex*)
apply *blast*

apply (*rule some1-equality*)
apply *blast*
apply *blast*
done

lemma *permutes-swap-id*: $a \in S \Longrightarrow b \in S \Longrightarrow$ *Fun.swap a b id permutes S*
unfolding *permutes-def Fun.swap-def fun-upd-def* **by** *auto metis*

lemma *permutes-superset*: *p permutes S* $\Longrightarrow (\forall x \in S - T.\ p\ x = x) \Longrightarrow$ *p permutes T*
by (*simp add*: *Ball-def permutes-def*) *metis*

72.3 Group properties

lemma *permutes-id*: *id permutes S*
unfolding *permutes-def* **by** *simp*

lemma *permutes-compose*: *p permutes S* $\Longrightarrow$ *q permutes S* $\Longrightarrow q \circ p$ *permutes S*
unfolding *permutes-def o-def* **by** *metis*

lemma *permutes-inv*:
assumes *pS*: *p permutes S*
shows *inv p permutes S*
using *pS* **unfolding** *permutes-def permutes-inv-eq*[*OF pS*] **by** *metis*

lemma *permutes-inv-inv*:
assumes *pS*: *p permutes S*
shows *inv* (*inv p*) = *p*
unfolding *fun-eq-iff permutes-inv-eq*[*OF pS*] *permutes-inv-eq*[*OF permutes-inv*[*OF pS*]]
by *blast*

72.4 The number of permutations on a finite set

lemma *permutes-insert-lemma*:
assumes *pS*: *p permutes* (*insert a S*)
shows *Fun.swap a* (*p a*) *id* $\circ$ *p permutes S*
apply (*rule permutes-superset*[**where** *S* = *insert a S*])
apply (*rule permutes-compose*[*OF pS*])
apply (*rule permutes-swap-id*, *simp*)
using *permutes-in-image*[*OF pS*, *of a*]
apply *simp*
apply (*auto simp add*: *Ball-def Fun.swap-def*)
done

lemma *permutes-insert*: $\{p.\ p$ *permutes* (*insert a S*)$\}$ =
$(\lambda(b,p).$ *Fun.swap a b id* $\circ\ p)$ ‘ $\{(b,p).\ b \in$ *insert a S* $\wedge\ p \in \{p.\ p$ *permutes S*$\}\}$
proof −
{
fix *p*

```
  {
    assume pS: p permutes insert a S
    let ?b = p a
    let ?q = Fun.swap a (p a) id ∘ p
    have th0: p = Fun.swap a ?b id ∘ ?q
      unfolding fun-eq-iff o-assoc by simp
    have th1: ?b ∈ insert a S
      unfolding permutes-in-image[OF pS] by simp
    from permutes-insert-lemma[OF pS] th0 th1
     have ∃ b q. p = Fun.swap a b id ∘ q ∧ b ∈ insert a S ∧ q permutes S by
blast
  }
  moreover
  {
    fix b q
    assume bq: p = Fun.swap a b id ∘ q b ∈ insert a S q permutes S
    from permutes-subset[OF bq(3), of insert a S]
    have qS: q permutes insert a S
      by auto
    have aS: a ∈ insert a S
      by simp
    from bq(1) permutes-compose[OF qS permutes-swap-id[OF aS bq(2)]]
    have p permutes insert a S
      by simp
  }
  ultimately have p permutes insert a S ⟷
      (∃ b q. p = Fun.swap a b id ∘ q ∧ b ∈ insert a S ∧ q permutes S)
    by blast
 }
 then show ?thesis
   by auto
qed

lemma card-permutations:
 assumes Sn: card S = n
   and fS: finite S
 shows card {p. p permutes S} = fact n
 using fS Sn
proof (induct arbitrary: n)
 case empty
 then show ?case by simp
next
 case (insert x F)
 {
   fix n
   assume H0: card (insert x F) = n
   let ?xF = {p. p permutes insert x F}
   let ?pF = {p. p permutes F}
   let ?pF' = {(b, p). b ∈ insert x F ∧ p ∈ ?pF}
```

let *?g* = (λ(*b*, *p*). *Fun.swap x b id* ∘ *p*)
from *permutes-insert*[*of x F*]
have *xfgpF′*: *?xF* = *?g* ' *?pF′* **.**
have *Fs*: *card F* = *n* − *1*
  using ‹*x* ∉ *F*› *H0* ‹*finite F*› **by** *auto*
from *insert.hyps Fs* **have** *pFs*: *card ?pF* = *fact* (*n* − *1*)
  using ‹*finite F*› **by** *auto*
then have *finite ?pF*
  by (*auto intro*: *card-ge-0-finite*)
then have *pF′f*: *finite ?pF′*
  using *H0* ‹*finite F*›
  apply (*simp only*: *Collect-case-prod Collect-mem-eq*)
  apply (*rule finite-cartesian-product*)
  apply *simp-all*
  done

have *ginj*: *inj-on ?g ?pF′*
proof −
  {
    fix *b p c q*
    assume *bp*: (*b*,*p*) ∈ *?pF′*
    assume *cq*: (*c*,*q*) ∈ *?pF′*
    assume *eq*: *?g* (*b*,*p*) = *?g* (*c*,*q*)
    from *bp cq* **have** *ths*: *b* ∈ *insert x F c* ∈ *insert x F x* ∈ *insert x F*
      p permutes F q permutes F
      by *auto*
    from *ths*(*4*) ‹*x* ∉ *F*› *eq* **have** *b* = *?g* (*b*,*p*) *x*
      unfolding *permutes-def*
      by (*auto simp add*: *Fun.swap-def fun-upd-def fun-eq-iff*)
    also have ... = *?g* (*c*,*q*) *x*
      using *ths*(*5*) ‹*x* ∉ *F*› *eq*
      by (*auto simp add*: *swap-def fun-upd-def fun-eq-iff*)
    also have ... = *c*
      using *ths*(*5*) ‹*x* ∉ *F*›
      unfolding *permutes-def*
      by (*auto simp add*: *Fun.swap-def fun-upd-def fun-eq-iff*)
    finally have *bc*: *b* = *c* **.**
    then have *Fun.swap x b id* = *Fun.swap x c id*
      by *simp*
    with *eq* **have** *Fun.swap x b id* ∘ *p* = *Fun.swap x b id* ∘ *q*
      by *simp*
    then have *Fun.swap x b id* ∘ (*Fun.swap x b id* ∘ *p*) =
      Fun.swap x b id ∘ (*Fun.swap x b id* ∘ *q*)
      by *simp*
    then have *p* = *q*
      by (*simp add*: *o-assoc*)
    with *bc* **have** (*b*, *p*) = (*c*, *q*)
      by *simp*
  }

then show *?thesis*
unfolding *inj-on-def* **by** *blast*
qed
from ⟨$x \notin F$⟩ *H0* **have** *n0*: $n \neq 0$
using ⟨*finite F*⟩ **by** *auto*
then have $\exists m.\ n = Suc\ m$
by *presburger*
then obtain *m* **where** *n*[*simp*]: $n = Suc\ m$
by *blast*
from *pFs H0* **have** *xFc*: *card ?xF* = *fact n*
unfolding *xfgpF′ card-image*[*OF ginj*]
using ⟨*finite F*⟩ ⟨*finite ?pF*⟩
apply (*simp only*: *Collect-case-prod Collect-mem-eq card-cartesian-product*)
apply *simp*
done
from *finite-imageI*[*OF pF′f*, *of ?g*] **have** *xFf*: *finite ?xF*
unfolding *xfgpF′* **by** *simp*
have *card ?xF* = *fact n*
using *xFf xFc* **unfolding** *xFf* **by** *blast*
}
then show *?case*
using *insert* **by** *simp*
qed

lemma *finite-permutations*:
assumes *fS*: *finite S*
shows *finite* {*p*. *p permutes S*}
using *card-permutations*[*OF refl fS*]
by (*auto intro*: *card-ge-0-finite*)

72.5 Permutations of index set for iterated operations

lemma (**in** *comm-monoid-set*) *permute*:
assumes *p permutes S*
shows $F\ g\ S = F\ (g \circ p)\ S$
proof −
from ⟨*p permutes S*⟩ **have** *inj p*
by (*rule permutes-inj*)
then have *inj-on p S*
by (*auto intro*: *subset-inj-on*)
then have $F\ g\ (p\ `\ S) = F\ (g \circ p)\ S$
by (*rule reindex*)
moreover from ⟨*p permutes S*⟩ **have** $p\ `\ S = S$
by (*rule permutes-image*)
ultimately show *?thesis*
by *simp*
qed

72.6 Various combinations of transpositions with 2, 1 and 0 common elements

lemma *swap-id-common*: $a \neq c \Longrightarrow b \neq c \Longrightarrow$
Fun.swap a b id ∘ *Fun.swap a c id* = *Fun.swap b c id* ∘ *Fun.swap a b id*
by (*simp add*: *fun-eq-iff Fun.swap-def*)

lemma *swap-id-common'*: $a \neq b \Longrightarrow a \neq c \Longrightarrow$
Fun.swap a c id ∘ *Fun.swap b c id* = *Fun.swap b c id* ∘ *Fun.swap a b id*
by (*simp add*: *fun-eq-iff Fun.swap-def*)

lemma *swap-id-independent*: $a \neq c \Longrightarrow a \neq d \Longrightarrow b \neq c \Longrightarrow b \neq d \Longrightarrow$
Fun.swap a b id ∘ *Fun.swap c d id* = *Fun.swap c d id* ∘ *Fun.swap a b id*
by (*simp add*: *fun-eq-iff Fun.swap-def*)

72.7 Permutations as transposition sequences

inductive *swapidseq* :: *nat* ⇒ (*'a* ⇒ *'a*) ⇒ *bool*
where
id[*simp*]: *swapidseq 0 id*
| *comp-Suc*: *swapidseq n p* ⟹ $a \neq b$ ⟹ *swapidseq* (*Suc n*) (*Fun.swap a b id* ∘ *p*)

declare *id*[*unfolded id-def*, *simp*]

definition *permutation p* ⟷ (∃ *n*. *swapidseq n p*)

72.8 Some closure properties of the set of permutations, with lengths

lemma *permutation-id*[*simp*]: *permutation id*
unfolding *permutation-def* **by** (*rule exI*[**where** *x=0*]) *simp*

declare *permutation-id*[*unfolded id-def*, *simp*]

lemma *swapidseq-swap*: *swapidseq* (*if a = b then 0 else 1*) (*Fun.swap a b id*)
apply *clarsimp*
using *comp-Suc*[*of 0 id a b*]
apply *simp*
done

lemma *permutation-swap-id*: *permutation* (*Fun.swap a b id*)
apply (*cases a = b*)
apply *simp-all*
unfolding *permutation-def*
using *swapidseq-swap*[*of a b*]
apply *blast*
done

lemma *swapidseq-comp-add*: *swapidseq n p* $\Longrightarrow$ *swapidseq m q* $\Longrightarrow$ *swapidseq* (*n* + *m*) (*p* ∘ *q*)
proof (*induct n p arbitrary*: *m q rule*: *swapidseq.induct*)
  case (*id m q*)
  then show *?case* **by** *simp*
next
  case (*comp-Suc n p a b m q*)
  have *th*: *Suc n* + *m* = *Suc* (*n* + *m*)
    by *arith*
  show *?case*
    unfolding *th comp-assoc*
    apply (*rule swapidseq.comp-Suc*)
    using *comp-Suc.hyps*(*2*)[*OF comp-Suc.prems*] *comp-Suc.hyps*(*3*)
    apply *blast+*
    done
qed

lemma *permutation-compose*: *permutation p* $\Longrightarrow$ *permutation q* $\Longrightarrow$ *permutation* (*p* ∘ *q*)
  unfolding *permutation-def* **using** *swapidseq-comp-add*[*of - p - q*] **by** *metis*

lemma *swapidseq-endswap*: *swapidseq n p* $\Longrightarrow$ *a* ≠ *b* $\Longrightarrow$ *swapidseq* (*Suc n*) (*p* ∘ *Fun.swap a b id*)
  apply (*induct n p rule*: *swapidseq.induct*)
  using *swapidseq-swap*[*of a b*]
  apply (*auto simp add*: *comp-assoc intro*: *swapidseq.comp-Suc*)
  done

lemma *swapidseq-inverse-exists*: *swapidseq n p* $\Longrightarrow$ ∃ *q*. *swapidseq n q* ∧ *p* ∘ *q* = *id* ∧ *q* ∘ *p* = *id*
proof (*induct n p rule*: *swapidseq.induct*)
  case *id*
  then show *?case*
    by (*rule exI*[**where** *x*=*id*]) *simp*
next
  case (*comp-Suc n p a b*)
  from *comp-Suc.hyps* **obtain** *q* **where** *q*: *swapidseq n q p* ∘ *q* = *id q* ∘ *p* = *id*
    by *blast*
  let *?q* = *q* ∘ *Fun.swap a b id*
  note *H* = *comp-Suc.hyps*
  from *swapidseq-swap*[*of a b*] *H*(*3*) **have** *th0*: *swapidseq 1* (*Fun.swap a b id*)
    by *simp*
  from *swapidseq-comp-add*[*OF q*(*1*) *th0*] **have** *th1*: *swapidseq* (*Suc n*) *?q*
    by *simp*
  have *Fun.swap a b id* ∘ *p* ∘ *?q* = *Fun.swap a b id* ∘ (*p* ∘ *q*) ∘ *Fun.swap a b id*
    by (*simp add*: *o-assoc*)
  also have ... = *id*
    by (*simp add*: *q*(*2*))
  finally have *th2*: *Fun.swap a b id* ∘ *p* ∘ *?q* = *id* **.**

have *?q* ∘ (*Fun.swap a b id* ∘ *p*) = *q* ∘ (*Fun.swap a b id* ∘ *Fun.swap a b id*) ∘ *p*
by (*simp only*: *o-assoc*)
then have *?q* ∘ (*Fun.swap a b id* ∘ *p*) = *id*
by (*simp add*: *q*(*3*))
with *th1 th2* **show** *?case*
by *blast*
qed

lemma *swapidseq-inverse*:
assumes *H*: *swapidseq n p*
shows *swapidseq n* (*inv p*)
using *swapidseq-inverse-exists*[*OF H*] *inv-unique-comp*[*of p*] **by** *auto*

lemma *permutation-inverse*: *permutation p* ⟹ *permutation* (*inv p*)
using *permutation-def swapidseq-inverse* **by** *blast*

72.9 The identity map only has even transposition sequences

lemma *symmetry-lemma*:
assumes ⋀*a b c d*. *P a b c d* ⟹ *P a b d c*
and ⋀*a b c d*. *a* ≠ *b* ⟹ *c* ≠ *d* ⟹
a = *c* ∧ *b* = *d* ∨ *a* = *c* ∧ *b* ≠ *d* ∨ *a* ≠ *c* ∧ *b* = *d* ∨ *a* ≠ *c* ∧ *a* ≠ *d* ∧ *b* ≠ *c* ∧ *b* ≠ *d* ⟹
P a b c d
shows ⋀*a b c d*. *a* ≠ *b* ⟶ *c* ≠ *d* ⟶ *P a b c d*
using *assms* **by** *metis*

lemma *swap-general*: *a* ≠ *b* ⟹ *c* ≠ *d* ⟹
Fun.swap a b id ∘ *Fun.swap c d id* = *id* ∨
(∃ *x y z*. *x* ≠ *a* ∧ *y* ≠ *a* ∧ *z* ≠ *a* ∧ *x* ≠ *y* ∧
Fun.swap a b id ∘ *Fun.swap c d id* = *Fun.swap x y id* ∘ *Fun.swap a z id*)
proof −
assume *H*: *a* ≠ *b c* ≠ *d*
have *a* ≠ *b* ⟶ *c* ≠ *d* ⟶
(*Fun.swap a b id* ∘ *Fun.swap c d id* = *id* ∨
(∃ *x y z*. *x* ≠ *a* ∧ *y* ≠ *a* ∧ *z* ≠ *a* ∧ *x* ≠ *y* ∧
Fun.swap a b id ∘ *Fun.swap c d id* = *Fun.swap x y id* ∘ *Fun.swap a z id*))
apply (*rule symmetry-lemma*[**where** *a*=*a* **and** *b*=*b* **and** *c*=*c* **and** *d*=*d*])
apply (*simp-all only*: *swap-commute*)
apply (*case-tac a* = *c* ∧ *b* = *d*)
apply (*clarsimp simp only*: *swap-commute swap-id-idempotent*)
apply (*case-tac a* = *c* ∧ *b* ≠ *d*)
apply (*rule disjI2*)
apply (*rule-tac x*=*b* **in** *exI*)
apply (*rule-tac x*=*d* **in** *exI*)
apply (*rule-tac x*=*b* **in** *exI*)
apply (*clarsimp simp add*: *fun-eq-iff Fun.swap-def*)
apply (*case-tac a* ≠ *c* ∧ *b* = *d*)
apply (*rule disjI2*)

apply (*rule-tac x=c* **in** *exI*)
apply (*rule-tac x=d* **in** *exI*)
apply (*rule-tac x=c* **in** *exI*)
apply (*clarsimp simp add*: *fun-eq-iff Fun.swap-def*)
apply (*rule disjI2*)
apply (*rule-tac x=c* **in** *exI*)
apply (*rule-tac x=d* **in** *exI*)
apply (*rule-tac x=b* **in** *exI*)
apply (*clarsimp simp add*: *fun-eq-iff Fun.swap-def*)
done
with *H* **show** *?thesis* **by** *metis*
qed

lemma *swapidseq-id-iff* [*simp*]: *swapidseq 0 p* ⟷ *p* = *id*
using *swapidseq.cases*[*of 0 p p* = *id*]
by *auto*

lemma *swapidseq-cases*: *swapidseq n p* ⟷
n = *0* ∧ *p* = *id* ∨ (∃ *a b q m*. *n* = *Suc m* ∧ *p* = *Fun.swap a b id* ∘ *q* ∧ *swapidseq m q* ∧ *a* ≠ *b*)
apply (*rule iffI*)
apply (*erule swapidseq.cases*[*of n p*])
apply *simp*
apply (*rule disjI2*)
apply (*rule-tac x*= *a* **in** *exI*)
apply (*rule-tac x*= *b* **in** *exI*)
apply (*rule-tac x*= *pa* **in** *exI*)
apply (*rule-tac x*= *na* **in** *exI*)
apply *simp*
apply *auto*
apply (*rule comp-Suc*, *simp-all*)
done

lemma *fixing-swapidseq-decrease*:
assumes *spn*: *swapidseq n p*
and *ab*: *a* ≠ *b*
and *pa*: (*Fun.swap a b id* ∘ *p*) *a* = *a*
shows *n* ≠ *0* ∧ *swapidseq* (*n* − *1*) (*Fun.swap a b id* ∘ *p*)
using *spn ab pa*
proof (*induct n arbitrary*: *p a b*)
case *0*
then show *?case*
by (*auto simp add*: *Fun.swap-def fun-upd-def*)
next
case (*Suc n p a b*)
from *Suc.prems*(*1*) *swapidseq-cases*[*of Suc n p*]
obtain *c d q m* **where**
cdqm: *Suc n* = *Suc m p* = *Fun.swap c d id* ∘ *q swapidseq m q c* ≠ *d n* = *m*
by *auto*

```
  {
    assume H: Fun.swap a b id ∘ Fun.swap c d id = id
    have ?case by (simp only: cdqm o-assoc H) (simp add: cdqm)
  }
  moreover
  {
    fix x y z
    assume H: x ≠ a y ≠ a z ≠ a x ≠ y
      Fun.swap a b id ∘ Fun.swap c d id = Fun.swap x y id ∘ Fun.swap a z id
    from H have az: a ≠ z
      by simp

    {
      fix h
      have (Fun.swap x y id ∘ h) a = a ⟷ h a = a
        using H by (simp add: Fun.swap-def)
    }
    note th3 = this
    from cdqm(2) have Fun.swap a b id ∘ p = Fun.swap a b id ∘ (Fun.swap c d
id ∘ q)
      by simp
    then have Fun.swap a b id ∘ p = Fun.swap x y id ∘ (Fun.swap a z id ∘ q)
      by (simp add: o-assoc H)
    then have (Fun.swap a b id ∘ p) a = (Fun.swap x y id ∘ (Fun.swap a z id ∘
q)) a
      by simp
    then have (Fun.swap x y id ∘ (Fun.swap a z id ∘ q)) a = a
      unfolding Suc by metis
    then have th1: (Fun.swap a z id ∘ q) a = a
      unfolding th3 .
    from Suc.hyps[OF cdqm(3)[ unfolded cdqm(5)[symmetric]] az th1]
    have th2: swapidseq (n − 1) (Fun.swap a z id ∘ q) n ≠ 0
      by blast+
    have th: Suc n − 1 = Suc (n − 1)
      using th2(2) by auto
    have ?case
      unfolding cdqm(2) H o-assoc th
      apply (simp only: Suc-not-Zero simp-thms comp-assoc)
      apply (rule comp-Suc)
      using th2 H
      apply blast+
      done
  }
  ultimately show ?case
    using swap-general[OF Suc.prems(2) cdqm(4)] by metis
qed

lemma swapidseq-identity-even:
  assumes swapidseq n (id :: 'a ⇒ 'a)
```

shows *even n*
using ⟨*swapidseq n id*⟩
proof (*induct n rule*: *nat-less-induct*)
fix *n*
assume *H*: $\forall m<n.$ *swapidseq m* (*id*::'*a* ⇒ '*a*) ⟶ *even m swapidseq n* (*id* :: '*a* ⇒ '*a*)
{
assume *n* = *0*
then have *even n* **by** *presburger*
}
moreover
{
fix *a b* :: '*a* **and** *q m*
assume *h*: *n* = *Suc m* (*id* :: '*a* ⇒ '*a*) = *Fun.swap a b id* ∘ *q swapidseq m q a* ≠ *b*
from *fixing-swapidseq-decrease*[*OF h*(*3*,*4*), *unfolded h*(*2*)[*symmetric*]]
have *m*: *m* ≠ *0 swapidseq* (*m* − *1*) (*id* :: '*a* ⇒ '*a*)
by *auto*
from *h m* **have** *mn*: *m* − *1* < *n*
by *arith*
from *H*(*1*)[*rule-format*, *OF mn m*(*2*)] *h*(*1*) *m*(*1*) **have** *even n*
by *presburger*
}
ultimately show *even n*
using *H*(*2*)[*unfolded swapidseq-cases*[*of n id*]] **by** *auto*
qed

72.10 Therefore we have a welldefined notion of parity

definition *evenperm p* = *even* (*SOME n*. *swapidseq n p*)

lemma *swapidseq-even-even*:
assumes *m*: *swapidseq m p*
and *n*: *swapidseq n p*
shows *even m* ⟷ *even n*
proof −
from *swapidseq-inverse-exists*[*OF n*]
obtain *q* **where** *q*: *swapidseq n q p* ∘ *q* = *id q* ∘ *p* = *id*
by *blast*
from *swapidseq-identity-even*[*OF swapidseq-comp-add*[*OF m q*(*1*), *unfolded q*]]
show *?thesis*
by *arith*
qed

lemma *evenperm-unique*:
assumes *p*: *swapidseq n p*
and *n*:*even n* = *b*
shows *evenperm p* = *b*
unfolding *n*[*symmetric*] *evenperm-def*

apply (*rule swapidseq-even-even*[**where** *p* = *p*])
apply (*rule someI*[**where** *x* = *n*])
using *p*
apply *blast+*
done

72.11 And it has the expected composition properties

lemma *evenperm-id*[*simp*]: *evenperm id* = *True*
by (*rule evenperm-unique*[**where** *n* = *0*]) *simp-all*

lemma *evenperm-swap*: *evenperm* (*Fun.swap a b id*) = (*a* = *b*)
by (*rule evenperm-unique*[**where** *n*=*if a* = *b then 0 else 1*]) (*simp-all add*: *swapidseq-swap*)

lemma *evenperm-comp*:
assumes *p*: *permutation p*
and *q*:*permutation q*
shows *evenperm* (*p* ∘ *q*) = (*evenperm p* = *evenperm q*)
proof −
from *p q* **obtain** *n m* **where** *n*: *swapidseq n p* **and** *m*: *swapidseq m q*
unfolding *permutation-def* **by** *blast*
note *nm* = *swapidseq-comp-add*[*OF n m*]
have *th*: *even* (*n* + *m*) = (*even n* ⟷ *even m*)
by *arith*
from *evenperm-unique*[*OF n refl*] *evenperm-unique*[*OF m refl*]
evenperm-unique[*OF nm th*]
show *?thesis*
by *blast*
qed

lemma *evenperm-inv*:
assumes *p*: *permutation p*
shows *evenperm* (*inv p*) = *evenperm p*
proof −
from *p* **obtain** *n* **where** *n*: *swapidseq n p*
unfolding *permutation-def* **by** *blast*
from *evenperm-unique*[*OF swapidseq-inverse*[*OF n*] *evenperm-unique*[*OF n refl*, *symmetric*]]
show *?thesis* **.**
qed

72.12 A more abstract characterization of permutations

lemma *bij-iff*: *bij f* ⟷ (∀ *x*. ∃!*y*. *f y* = *x*)
unfolding *bij-def inj-on-def surj-def*
apply *auto*
apply *metis*
apply *metis*
done

lemma *permutation-bijective*:
  assumes *p*: *permutation p*
  shows *bij p*
proof −
  from *p* **obtain** *n* **where** *n*: *swapidseq n p*
    unfolding *permutation-def* **by** *blast*
  from *swapidseq-inverse-exists*[*OF n*]
  obtain *q* **where** *q*: *swapidseq n q p* ∘ *q* = *id q* ∘ *p* = *id*
    by *blast*
  then show *?thesis* **unfolding** *bij-iff*
    apply (*auto simp add*: *fun-eq-iff*)
    apply *metis*
    done
qed

lemma *permutation-finite-support*:
  assumes *p*: *permutation p*
  shows *finite* {*x*. *p x* ≠ *x*}
proof −
  from *p* **obtain** *n* **where** *n*: *swapidseq n p*
    unfolding *permutation-def* **by** *blast*
  from *n* **show** *?thesis*
  proof (*induct n p rule*: *swapidseq.induct*)
    case *id*
    then show *?case* **by** *simp*
  next
    case (*comp-Suc n p a b*)
    let *?S* = *insert a* (*insert b* {*x*. *p x* ≠ *x*})
    from *comp-Suc.hyps*(*2*) **have** *fS*: *finite ?S*
      by *simp*
    from ‹*a* ≠ *b*› **have** *th*: {*x*. (*Fun.swap a b id* ∘ *p*) *x* ≠ *x*} ⊆ *?S*
      by (*auto simp add*: *Fun.swap-def*)
    from *finite-subset*[*OF th fS*] **show** *?case* **.**
  qed
qed

lemma *bij-inv-eq-iff*: *bij p* ⟹ *x* = *inv p y* ⟷ *p x* = *y*
  using *surj-f-inv-f*[*of p*] **by** (*auto simp add*: *bij-def*)

lemma *bij-swap-comp*:
  assumes *bp*: *bij p*
  shows *Fun.swap a b id* ∘ *p* = *Fun.swap* (*inv p a*) (*inv p b*) *p*
  using *surj-f-inv-f*[*OF bij-is-surj*[*OF bp*]]
  by (*simp add*: *fun-eq-iff Fun.swap-def bij-inv-eq-iff*[*OF bp*])

lemma *bij-swap-ompose-bij*: *bij p* ⟹ *bij* (*Fun.swap a b id* ∘ *p*)
proof −
  assume *H*: *bij p*

show *?thesis*
  unfolding *bij-swap-comp*[*OF H*] *bij-swap-iff*
  using *H* **.**
qed

lemma *permutation-lemma*:
  assumes *fS*: *finite S*
    and *p*: *bij p*
    and *pS*: $\forall x.\ x \notin S \longrightarrow p\ x = x$
  shows *permutation p*
  using *fS p pS*
proof (*induct S arbitrary*: *p rule*: *finite-induct*)
  case (*empty p*)
  then show *?case* **by** *simp*
next
  case (*insert a F p*)
  let *?r* = *Fun.swap a* (*p a*) *id* ∘ *p*
  let *?q* = *Fun.swap a* (*p a*) *id* ∘ *?r*
  have *raa*: *?r a* = *a*
    by (*simp add*: *Fun.swap-def*)
  from *bij-swap-ompose-bij*[*OF insert*(*4*)]
  have *br*: *bij ?r* **.**

  from *insert raa* **have** *th*: $\forall x.\ x \notin F \longrightarrow$ *?r x* = *x*
    apply (*clarsimp simp add*: *Fun.swap-def*)
    apply (*erule-tac x*=*x* **in** *allE*)
    apply *auto*
    unfolding *bij-iff*
    apply *metis*
    done
  from *insert*(*3*)[*OF br th*]
  have *rp*: *permutation ?r* **.**
  have *permutation ?q*
    by (*simp add*: *permutation-compose permutation-swap-id rp*)
  then show *?case*
    by (*simp add*: *o-assoc*)
qed

lemma *permutation*: *permutation p* ⟷ *bij p* ∧ *finite* {*x*. *p x* ≠ *x*}
  (**is** *?lhs* ⟷ *?b* ∧ *?f*)
proof
  assume *p*: *?lhs*
  from *p permutation-bijective permutation-finite-support* **show** *?b* ∧ *?f*
    by *auto*
next
  assume *?b* ∧ *?f*
  then have *?f ?b* **by** *blast+*
  from *permutation-lemma*[*OF this*] **show** *?lhs*
    by *blast*

qed

lemma *permutation-inverse-works*:
  assumes *p*: *permutation p*
  shows *inv p* ∘ *p* = *id*
    and *p* ∘ *inv p* = *id*
  using *permutation-bijective* [*OF p*]
  unfolding *bij-def inj-iff surj-iff* **by** *auto*

lemma *permutation-inverse-compose*:
  assumes *p*: *permutation p*
    and *q*: *permutation q*
  shows *inv* (*p* ∘ *q*) = *inv q* ∘ *inv p*
proof −
  note *ps* = *permutation-inverse-works*[*OF p*]
  note *qs* = *permutation-inverse-works*[*OF q*]
  have *p* ∘ *q* ∘ (*inv q* ∘ *inv p*) = *p* ∘ (*q* ∘ *inv q*) ∘ *inv p*
    by (*simp add*: *o-assoc*)
  also have ... = *id*
    by (*simp add*: *ps qs*)
  finally have *th0*: *p* ∘ *q* ∘ (*inv q* ∘ *inv p*) = *id* **.**
  have *inv q* ∘ *inv p* ∘ (*p* ∘ *q*) = *inv q* ∘ (*inv p* ∘ *p*) ∘ *q*
    by (*simp add*: *o-assoc*)
  also have ... = *id*
    by (*simp add*: *ps qs*)
  finally have *th1*: *inv q* ∘ *inv p* ∘ (*p* ∘ *q*) = *id* **.**
  from *inv-unique-comp*[*OF th0 th1*] **show** *?thesis* **.**
qed

72.13 Relation to "permutes"

lemma *permutation-permutes*: *permutation p* ⟷ (∃ *S*. *finite S* ∧ *p permutes S*)
  unfolding *permutation permutes-def bij-iff* [*symmetric*]
  apply (*rule iffI*, *clarify*)
  apply (*rule exI*[**where** *x*={*x*. *p x* ≠ *x*}])
  apply *simp*
  apply *clarsimp*
  apply (*rule-tac B*=*S* **in** *finite-subset*)
  apply *auto*
  done

72.14 Hence a sort of induction principle composing by swaps

lemma *permutes-induct*: *finite S* ⟹ *P id* ⟹
  (⋀ *a b p*. *a* ∈ *S* ⟹ *b* ∈ *S* ⟹ *P p* ⟹ *P p* ⟹ *permutation p* ⟹ *P* (*Fun.swap a b id* ∘ *p*)) ⟹
  (⋀*p*. *p permutes S* ⟹ *P p*)
proof (*induct S rule*: *finite-induct*)
  case *empty*
  then show *?case* **by** *auto*

next
  case (*insert x F p*)
  let *?r* = *Fun.swap x* (*p x*) *id* ∘ *p*
  let *?q* = *Fun.swap x* (*p x*) *id* ∘ *?r*
  have *qp*: *?q* = *p*
    by (*simp add*: *o-assoc*)
  from *permutes-insert-lemma*[*OF insert.prems*(*3*)] *insert* **have** *Pr*: *P ?r*
    by *blast*
  from *permutes-in-image*[*OF insert.prems*(*3*), *of x*]
  have *pxF*: *p x* ∈ *insert x F*
    by *simp*
  have *xF*: *x* ∈ *insert x F*
    by *simp*
  have *rp*: *permutation ?r*
    unfolding *permutation-permutes* **using** *insert.hyps*(*1*)
      permutes-insert-lemma[*OF insert.prems*(*3*)]
    by *blast*
  from *insert.prems*(*2*)[*OF xF pxF Pr Pr rp*]
  show *?case*
    unfolding *qp* **.**
qed

72.15 Sign of a permutation as a real number

definition *sign p* = (*if evenperm p then* (*1*::*int*) *else* −*1*)

lemma *sign-nz*: *sign p* ≠ *0*
  by (*simp add*: *sign-def*)

lemma *sign-id*: *sign id* = *1*
  by (*simp add*: *sign-def*)

lemma *sign-inverse*: *permutation p* ⟹ *sign* (*inv p*) = *sign p*
  by (*simp add*: *sign-def evenperm-inv*)

lemma *sign-compose*: *permutation p* ⟹ *permutation q* ⟹ *sign* (*p* ∘ *q*) = *sign p* ∗ *sign q*
  by (*simp add*: *sign-def evenperm-comp*)

lemma *sign-swap-id*: *sign* (*Fun.swap a b id*) = (*if a* = *b then 1 else* −*1*)
  by (*simp add*: *sign-def evenperm-swap*)

lemma *sign-idempotent*: *sign p* ∗ *sign p* = *1*
  by (*simp add*: *sign-def*)

72.16 More lemmas about permutations

lemma *permutes-natset-le*:
  fixes *S* :: *'a*::*wellorder set*
  assumes *p*: *p permutes S*

```
    and le: ∀ i ∈ S. p i ≤ i
  shows p = id
proof −
  {
    fix n
    have p n = n
      using p le
    proof (induct n arbitrary: S rule: less-induct)
      fix n S
      assume H:
        ⋀m S. m < n ⟹ p permutes S ⟹ ∀ i∈S. p i ≤ i ⟹ p m = m
        p permutes S ∀ i ∈S. p i ≤ i
      {
        assume n ∉ S
        with H(2) have p n = n
          unfolding permutes-def by metis
      }
      moreover
      {
        assume ns: n ∈ S
        from H(3)  ns have p n < n ∨ p n = n
          by auto
        moreover {
          assume h: p n < n
          from H h have p (p n) = p n
            by metis
          with permutes-inj[OF H(2)] have p n = n
            unfolding inj-on-def by blast
          with h have False
            by simp
        }
        ultimately have p n = n
          by blast
      }
      ultimately show p n = n
        by blast
    qed
  }
  then show ?thesis
    by (auto simp add: fun-eq-iff)
qed

lemma permutes-natset-ge:
  fixes S :: 'a::wellorder set
  assumes p: p permutes S
    and le: ∀ i ∈ S. p i ≥ i
  shows p = id
proof −
  {
```

fix *i*
assume *i*: $i \in S$
from *i permutes-in-image*[*OF permutes-inv*[*OF p*]] **have** $inv\ p\ i \in S$
by *simp*
with *le* **have** $p\ (inv\ p\ i) \geq inv\ p\ i$
by *blast*
with *permutes-inverses*[*OF p*] **have** $i \geq inv\ p\ i$
by *simp*
}
then have *th*: $\forall\, i{\in}S.\ inv\ p\ i \leq i$
by *blast*
from *permutes-natset-le*[*OF permutes-inv*[*OF p*] *th*]
have $inv\ p = inv\ id$
by *simp*
then show *?thesis*
apply (*subst permutes-inv-inv*[*OF p*, *symmetric*])
apply (*rule inv-unique-comp*)
apply *simp-all*
done
qed

lemma *image-inverse-permutations*: $\{inv\ p\ |p.\ p\ permutes\ S\} = \{p.\ p\ permutes\ S\}$
apply (*rule set-eqI*)
apply *auto*
using *permutes-inv-inv permutes-inv*
apply *auto*
apply (*rule-tac x=inv x* **in** *exI*)
apply *auto*
done

lemma *image-compose-permutations-left*:
assumes *q*: *q permutes S*
shows $\{q \circ p\ |\ p.\ p\ permutes\ S\} = \{p\ .\ p\ permutes\ S\}$
apply (*rule set-eqI*)
apply *auto*
apply (*rule permutes-compose*)
using *q*
apply *auto*
apply (*rule-tac* $x = inv\ q \circ x$ **in** *exI*)
apply (*simp add*: *o-assoc permutes-inv permutes-compose permutes-inv-o*)
done

lemma *image-compose-permutations-right*:
assumes *q*: *q permutes S*
shows $\{p \circ q\ |\ p.\ p\ permutes\ S\} = \{p\ .\ p\ permutes\ S\}$
apply (*rule set-eqI*)
apply *auto*
apply (*rule permutes-compose*)

using *q*
apply *auto*
apply (*rule-tac x* = *x* ∘ *inv q* **in** *exI*)
apply (*simp add*: *o-assoc permutes-inv permutes-compose permutes-inv-o comp-assoc*)
done

lemma *permutes-in-seg*: *p permutes* {*1* ..*n*} ⟹ *i* ∈ {*1*..*n*} ⟹ *1* ≤ *p i* ∧ *p i* ≤ *n*
by (*simp add*: *permutes-def*) *metis*

lemma *setsum-permutations-inverse*:
setsum f {*p*. *p permutes S*} = *setsum* (λ*p*. *f*(*inv p*)) {*p*. *p permutes S*}
(**is** *?lhs* = *?rhs*)
proof −
let *?S* = {*p* . *p permutes S*}
have *th0*: *inj-on inv ?S*
proof (*auto simp add*: *inj-on-def*)
fix *q r*
assume *q*: *q permutes S*
and *r*: *r permutes S*
and *qr*: *inv q* = *inv r*
then have *inv* (*inv q*) = *inv* (*inv r*)
by *simp*
with *permutes-inv-inv*[*OF q*] *permutes-inv-inv*[*OF r*] **show** *q* = *r*
by *metis*
qed
have *th1*: *inv* ' *?S* = *?S*
using *image-inverse-permutations* **by** *blast*
have *th2*: *?rhs* = *setsum* (*f* ∘ *inv*) *?S*
by (*simp add*: *o-def*)
from *setsum.reindex*[*OF th0*, *of f*] **show** *?thesis* **unfolding** *th1 th2* **.**
qed

lemma *setum-permutations-compose-left*:
assumes *q*: *q permutes S*
shows *setsum f* {*p*. *p permutes S*} = *setsum* (λ*p*. *f*(*q* ∘ *p*)) {*p*. *p permutes S*}
(**is** *?lhs* = *?rhs*)
proof −
let *?S* = {*p*. *p permutes S*}
have *th0*: *?rhs* = *setsum* (*f* ∘ (*op* ∘ *q*)) *?S*
by (*simp add*: *o-def*)
have *th1*: *inj-on* (*op* ∘ *q*) *?S*
proof (*auto simp add*: *inj-on-def*)
fix *p r*
assume *p permutes S*
and *r*: *r permutes S*
and *rp*: *q* ∘ *p* = *q* ∘ *r*
then have *inv q* ∘ *q* ∘ *p* = *inv q* ∘ *q* ∘ *r*
by (*simp add*: *comp-assoc*)

with *permutes-inj*[*OF q*, *unfolded inj-iff*] **show** $p = r$
by *simp*
qed
have *th3*: $(op \circ q)$ ' *?S* = *?S*
using *image-compose-permutations-left*[*OF q*] **by** *auto*
from *setsum.reindex*[*OF th1*, *of f*] **show** *?thesis* **unfolding** *th0 th1 th3* **.**
qed

lemma *sum-permutations-compose-right*:
assumes *q*: *q permutes S*
shows *setsum f* {*p*. *p permutes S*} = *setsum* (λp. $f(p \circ q)$) {*p*. *p permutes S*}
(**is** *?lhs* = *?rhs*)
proof −
let *?S* = {*p*. *p permutes S*}
have *th0*: *?rhs* = *setsum* ($f \circ (\lambda p.\ p \circ q)$) *?S*
by (*simp add*: *o-def*)
have *th1*: *inj-on* ($\lambda p.\ p \circ q$) *?S*
proof (*auto simp add*: *inj-on-def*)
fix *p r*
assume *p permutes S*
and *r*: *r permutes S*
and *rp*: $p \circ q = r \circ q$
then have $p \circ (q \circ inv\ q) = r \circ (q \circ inv\ q)$
by (*simp add*: *o-assoc*)
with *permutes-surj*[*OF q*, *unfolded surj-iff*] **show** $p = r$
by *simp*
qed
have *th3*: ($\lambda p.\ p \circ q$) ' *?S* = *?S*
using *image-compose-permutations-right*[*OF q*] **by** *auto*
from *setsum.reindex*[*OF th1*, *of f*]
show *?thesis* **unfolding** *th0 th1 th3* **.**
qed

72.17 Sum over a set of permutations (could generalize to iteration)

lemma *setsum-over-permutations-insert*:
assumes *fS*: *finite S*
and *aS*: $a \notin S$
shows *setsum f* {*p*. *p permutes* (*insert a S*)} =
setsum (λb. *setsum* (λq. *f* (*Fun.swap a b id* $\circ$ *q*)) {*p*. *p permutes S*}) (*insert a S*)
proof −
have *th0*: $\bigwedge f\ a\ b$. ($\lambda(b,p)$. *f* (*Fun.swap a b id* $\circ$ *p*)) = $f \circ$ ($\lambda(b,p)$. *Fun.swap a b id* $\circ$ *p*)
by (*simp add*: *fun-eq-iff*)
have *th1*: $\bigwedge P\ Q.\ P \times Q = \{(a,b).\ a \in P \wedge b \in Q\}$
by *blast*
have *th2*: $\bigwedge P\ Q.\ P \Longrightarrow (P \Longrightarrow Q) \Longrightarrow P \wedge Q$

```
    by blast
  show ?thesis
    unfolding permutes-insert
    unfolding setsum.cartesian-product
    unfolding th1[symmetric]
    unfolding th0
  proof (rule setsum.reindex)
    let ?f = (λ(b, y). Fun.swap a b id ∘ y)
    let ?P = {p. p permutes S}
    {
      fix b c p q
      assume b: b ∈ insert a S
      assume c: c ∈ insert a S
      assume p: p permutes S
      assume q: q permutes S
      assume eq: Fun.swap a b id ∘ p = Fun.swap a c id ∘ q
      from p q aS have pa: p a = a and qa: q a = a
        unfolding permutes-def by metis+
      from eq have (Fun.swap a b id ∘ p) a  = (Fun.swap a c id ∘ q) a
        by simp
      then have bc: b = c
        by (simp add: permutes-def pa qa o-def fun-upd-def Fun.swap-def id-def
            cong del: if-weak-cong split: if-split-asm)
      from eq[unfolded bc] have (λp. Fun.swap a c id ∘ p) (Fun.swap a c id ∘ p)
=
        (λp. Fun.swap a c id ∘ p) (Fun.swap a c id ∘ q) by simp
      then have p = q
        unfolding o-assoc swap-id-idempotent
        by (simp add: o-def)
      with bc have b = c ∧ p = q
        by blast
    }
    then show inj-on ?f (insert a S × ?P)
      unfolding inj-on-def by clarify metis
  qed
qed

end
```

73 Roots of real quadratics

```
theory Quadratic-Discriminant
imports Complex-Main
begin

definition discrim :: [real,real,real] ⇒ real where
  discrim a b c ≡ b^2 − 4 ∗ a ∗ c

lemma complete-square:
```

fixes $a\ b\ c\ x :: real$
assumes $a \neq 0$
shows $a * x^2 + b * x + c = 0 \longleftrightarrow (2 * a * x + b)^2 = discrim\ a\ b\ c$
proof −
have $4 * a^2 * x^2 + 4 * a * b * x + 4 * a * c = 4 * a * (a * x^2 + b * x + c)$
by (*simp add*: *algebra-simps power2-eq-square*)
with ‹$a \neq 0$›
have $a * x^2 + b * x + c = 0 \longleftrightarrow 4 * a^2 * x^2 + 4 * a * b * x + 4 * a * c = 0$
by *simp*
thus $a * x^2 + b * x + c = 0 \longleftrightarrow (2 * a * x + b)^2 = discrim\ a\ b\ c$
unfolding *discrim-def*
by (*simp add*: *power2-eq-square algebra-simps*)
qed

lemma *discriminant-negative*:
fixes $a\ b\ c\ x :: real$
assumes $a \neq 0$
and $discrim\ a\ b\ c < 0$
shows $a * x^2 + b * x + c \neq 0$
proof −
have $(2 * a * x + b)^2 \geq 0$ **by** *simp*
with ‹$discrim\ a\ b\ c < 0$› **have** $(2 * a * x + b)^2 \neq discrim\ a\ b\ c$ **by** *arith*
with *complete-square* **and** ‹$a \neq 0$› **show** $a * x^2 + b * x + c \neq 0$ **by** *simp*
qed

lemma *plus-or-minus-sqrt*:
fixes $x\ y :: real$
assumes $y \geq 0$
shows $x^2 = y \longleftrightarrow x = sqrt\ y \vee x = -\ sqrt\ y$
proof
assume $x^2 = y$
hence $sqrt\ (x^2) = sqrt\ y$ **by** *simp*
hence $sqrt\ y = |x|$ **by** *simp*
thus $x = sqrt\ y \vee x = -\ sqrt\ y$ **by** *auto*
next
assume $x = sqrt\ y \vee x = -\ sqrt\ y$
hence $x^2 = (sqrt\ y)^2 \vee x^2 = (-\ sqrt\ y)^2$ **by** *auto*
with ‹$y \geq 0$› **show** $x^2 = y$ **by** *simp*
qed

lemma *divide-non-zero*:
fixes $x\ y\ z :: real$
assumes $x \neq 0$
shows $x * y = z \longleftrightarrow y = z\ /\ x$
proof
assume $x * y = z$
with ‹$x \neq 0$› **show** $y = z\ /\ x$ **by** (*simp add*: *field-simps*)
next
assume $y = z\ /\ x$

with ‹$x \neq 0$› **show** $x * y = z$ **by** *simp*
qed

lemma *discriminant-nonneg*:
fixes $a\ b\ c\ x :: real$
assumes $a \neq 0$
and $discrim\ a\ b\ c \geq 0$
shows $a * x^2 + b * x + c = 0 \longleftrightarrow$
$x = (-b + sqrt\ (discrim\ a\ b\ c))\ /\ (2 * a) \vee$
$x = (-b - sqrt\ (discrim\ a\ b\ c))\ /\ (2 * a)$
proof −
from *complete-square* **and** *plus-or-minus-sqrt* **and** *assms*
have $a * x^2 + b * x + c = 0 \longleftrightarrow$
$(2 * a) * x + b = sqrt\ (discrim\ a\ b\ c) \vee$
$(2 * a) * x + b = -\ sqrt\ (discrim\ a\ b\ c)$
by *simp*
also have ... $\longleftrightarrow (2 * a) * x = (-b + sqrt\ (discrim\ a\ b\ c)) \vee$
$(2 * a) * x = (-b - sqrt\ (discrim\ a\ b\ c))$
by *auto*
also from ‹$a \neq 0$› **and** *divide-non-zero* [*of* $2 * a\ x$]
have ... $\longleftrightarrow x = (-b + sqrt\ (discrim\ a\ b\ c))\ /\ (2 * a) \vee$
$x = (-b - sqrt\ (discrim\ a\ b\ c))\ /\ (2 * a)$
by *simp*
finally show $a * x^2 + b * x + c = 0 \longleftrightarrow$
$x = (-b + sqrt\ (discrim\ a\ b\ c))\ /\ (2 * a) \vee$
$x = (-b - sqrt\ (discrim\ a\ b\ c))\ /\ (2 * a)$ **.**
qed

lemma *discriminant-zero*:
fixes $a\ b\ c\ x :: real$
assumes $a \neq 0$
and $discrim\ a\ b\ c = 0$
shows $a * x^2 + b * x + c = 0 \longleftrightarrow x = -b\ /\ (2 * a)$
using *discriminant-nonneg* **and** *assms*
by *simp*

theorem *discriminant-iff*:
fixes $a\ b\ c\ x :: real$
assumes $a \neq 0$
shows $a * x^2 + b * x + c = 0 \longleftrightarrow$
$discrim\ a\ b\ c \geq 0 \wedge$
$(x = (-b + sqrt\ (discrim\ a\ b\ c))\ /\ (2 * a) \vee$
$x = (-b - sqrt\ (discrim\ a\ b\ c))\ /\ (2 * a))$
proof
assume $a * x^2 + b * x + c = 0$
with *discriminant-negative* **and** ‹$a \neq 0$› **have** $\neg(discrim\ a\ b\ c < 0)$ **by** *auto*
hence $discrim\ a\ b\ c \geq 0$ **by** *simp*
with *discriminant-nonneg* **and** ‹$a * x^2 + b * x + c = 0$› **and** ‹$a \neq 0$›
have $x = (-b + sqrt\ (discrim\ a\ b\ c))\ /\ (2 * a) \vee$

$x = (-b - sqrt\ (discrim\ a\ b\ c))\ /\ (2 * a)$
by *simp*
with ⟨$discrim\ a\ b\ c \geq 0$⟩
show $discrim\ a\ b\ c \geq 0\ \wedge$
$(x = (-b + sqrt\ (discrim\ a\ b\ c))\ /\ (2 * a)\ \vee$
$x = (-b - sqrt\ (discrim\ a\ b\ c))\ /\ (2 * a))$ **..**
next
assume $discrim\ a\ b\ c \geq 0\ \wedge$
$(x = (-b + sqrt\ (discrim\ a\ b\ c))\ /\ (2 * a)\ \vee$
$x = (-b - sqrt\ (discrim\ a\ b\ c))\ /\ (2 * a))$
hence $discrim\ a\ b\ c \geq 0$ **and**
$x = (-b + sqrt\ (discrim\ a\ b\ c))\ /\ (2 * a)\ \vee$
$x = (-b - sqrt\ (discrim\ a\ b\ c))\ /\ (2 * a)$
by *simp-all*
with *discriminant-nonneg* **and** ⟨$a \neq 0$⟩ **show** $a * x^2 + b * x + c = 0$ **by** *simp*
qed

lemma *discriminant-nonneg-ex*:
fixes $a\ b\ c$ :: *real*
assumes $a \neq 0$
and $discrim\ a\ b\ c \geq 0$
shows $\exists\ x.\ a * x^2 + b * x + c = 0$
using *discriminant-nonneg* **and** *assms*
by *auto*

lemma *discriminant-pos-ex*:
fixes $a\ b\ c$ :: *real*
assumes $a \neq 0$
and $discrim\ a\ b\ c > 0$
shows $\exists\ x\ y.\ x \neq y \wedge a * x^2 + b * x + c = 0 \wedge a * y^2 + b * y + c = 0$
proof −
let $?x = (-b + sqrt\ (discrim\ a\ b\ c))\ /\ (2 * a)$
let $?y = (-b - sqrt\ (discrim\ a\ b\ c))\ /\ (2 * a)$
from ⟨$discrim\ a\ b\ c > 0$⟩ **have** $sqrt\ (discrim\ a\ b\ c) \neq 0$ **by** *simp*
hence $sqrt\ (discrim\ a\ b\ c) \neq -\ sqrt\ (discrim\ a\ b\ c)$ **by** *arith*
with ⟨$a \neq 0$⟩ **have** $?x \neq ?y$ **by** *simp*
moreover
from *discriminant-nonneg* [*of a b c ?x*]
and *discriminant-nonneg* [*of a b c ?y*]
and *assms*
have $a * ?x^2 + b * ?x + c = 0$ **and** $a * ?y^2 + b * ?y + c = 0$ **by** *simp-all*
ultimately
show $\exists\ x\ y.\ x \neq y \wedge a * x^2 + b * x + c = 0 \wedge a * y^2 + b * y + c = 0$ **by** *blast*
qed

lemma *discriminant-pos-distinct*:
fixes $a\ b\ c\ x$ :: *real*
assumes $a \neq 0$ **and** $discrim\ a\ b\ c > 0$

shows ∃ *y*. $x \neq y \wedge a * y^2 + b * y + c = 0$
proof −
from *discriminant-pos-ex* **and** ‹$a \neq 0$› **and** ‹*discrim a b c* > *0*›
obtain *w* **and** *z* **where** $w \neq z$
and $a * w^2 + b * w + c = 0$ **and** $a * z^2 + b * z + c = 0$
by *blast*
show ∃ *y*. $x \neq y \wedge a * y^2 + b * y + c = 0$
proof *cases*
assume $x = w$
with ‹$w \neq z$› **have** $x \neq z$ **by** *simp*
with ‹$a * z^2 + b * z + c = 0$›
show ∃ *y*. $x \neq y \wedge a * y^2 + b * y + c = 0$ **by** *auto*
next
assume $x \neq w$
with ‹$a * w^2 + b * w + c = 0$›
show ∃ *y*. $x \neq y \wedge a * y^2 + b * y + c = 0$ **by** *auto*
qed
qed

end

74 Pretty syntax for Quotient operations

theory *Quotient-Syntax*
imports *Main*
begin

notation
rel-conj (**infixr** *OOO 75*) **and**
map-fun (**infixr** −−−> *55*) **and**
rel-fun (**infixr** ===> *55*)

end

75 Quotient infrastructure for the set type

theory *Quotient-Set*
imports *Quotient-Syntax*
begin

75.1 Contravariant set map (vimage) and set relator, rules for the Quotient package

definition *rel-vset R xs ys* ≡ ∀ *x y*. *R x y* ⟶ *x* ∈ *xs* ⟷ *y* ∈ *ys*

lemma *rel-vset-eq* [*id-simps*]:
rel-vset op = = *op* =
by (*subst fun-eq-iff*, *subst fun-eq-iff*) (*simp add*: *set-eq-iff rel-vset-def*)

lemma *rel-vset-equivp*:
  assumes *e*: *equivp R*
  shows *rel-vset R xs ys* ⟷ *xs* = *ys* ∧ (∀ *x y*. *x* ∈ *xs* ⟶ *R x y* ⟶ *y* ∈ *xs*)
  unfolding *rel-vset-def*
  using *equivp-reflp*[*OF e*]
  by *auto* (*metis*, *metis equivp-symp*[*OF e*])

lemma *set-quotient* [*quot-thm*]:
  assumes *Quotient3 R Abs Rep*
  shows *Quotient3* (*rel-vset R*) (*vimage Rep*) (*vimage Abs*)
proof (*rule Quotient3I*)
  from *assms* **have** ⋀*x*. *Abs* (*Rep x*) = *x* **by** (*rule Quotient3-abs-rep*)
  then show ⋀*xs*. *Rep* −‘ (*Abs* −‘ *xs*) = *xs*
    unfolding *vimage-def* **by** *auto*
next
  show ⋀*xs*. *rel-vset R* (*Abs* −‘ *xs*) (*Abs* −‘ *xs*)
    unfolding *rel-vset-def vimage-def*
    by *auto* (*metis Quotient3-rel-abs*[*OF assms*])+
next
  fix *r s*
  show *rel-vset R r s* = (*rel-vset R r r* ∧ *rel-vset R s s* ∧ *Rep* −‘ *r* = *Rep* −‘ *s*)
    unfolding *rel-vset-def vimage-def set-eq-iff*
    by *auto* (*metis rep-abs-rsp*[*OF assms*] *assms*[*simplified Quotient3-def*])+
qed

declare [[*mapQ3 set* = (*rel-vset*, *set-quotient*)]]

lemma *empty-set-rsp*[*quot-respect*]:
  rel-vset R {} {}
  unfolding *rel-vset-def* **by** *simp*

lemma *collect-rsp*[*quot-respect*]:
  assumes *Quotient3 R Abs Rep*
  shows ((*R* ===> *op* =) ===> *rel-vset R*) *Collect Collect*
  by (*intro rel-funI*) (*simp add*: *rel-fun-def rel-vset-def*)

lemma *collect-prs*[*quot-preserve*]:
  assumes *Quotient3 R Abs Rep*
  shows ((*Abs* −−−> *id*) −−−> *op* −‘ *Rep*) *Collect* = *Collect*
  unfolding *fun-eq-iff*
  by (*simp add*: *Quotient3-abs-rep*[*OF assms*])

lemma *union-rsp*[*quot-respect*]:
  assumes *Quotient3 R Abs Rep*
  shows (*rel-vset R* ===> *rel-vset R* ===> *rel-vset R*) *op* ∪ *op* ∪
  by (*intro rel-funI*) (*simp add*: *rel-vset-def*)

lemma *union-prs*[*quot-preserve*]:

assumes *Quotient3 R Abs Rep*
shows (*op* −' *Abs* −−−> *op* −' *Abs* −−−> *op* −' *Rep*) *op* ∪ = *op* ∪
unfolding *fun-eq-iff*
by (*simp add*: *Quotient3-abs-rep*[*OF set-quotient*[*OF assms*]])

lemma *diff-rsp*[*quot-respect*]:
assumes *Quotient3 R Abs Rep*
shows (*rel-vset R* ===> *rel-vset R* ===> *rel-vset R*) *op* − *op* −
by (*intro rel-funI*) (*simp add*: *rel-vset-def*)

lemma *diff-prs*[*quot-preserve*]:
assumes *Quotient3 R Abs Rep*
shows (*op* −' *Abs* −−−> *op* −' *Abs* −−−> *op* −' *Rep*) *op* − = *op* −
unfolding *fun-eq-iff*
by (*simp add*: *Quotient3-abs-rep*[*OF set-quotient*[*OF assms*]] *vimage-Diff*)

lemma *inter-rsp*[*quot-respect*]:
assumes *Quotient3 R Abs Rep*
shows (*rel-vset R* ===> *rel-vset R* ===> *rel-vset R*) *op* ∩ *op* ∩
by (*intro rel-funI*) (*auto simp add*: *rel-vset-def*)

lemma *inter-prs*[*quot-preserve*]:
assumes *Quotient3 R Abs Rep*
shows (*op* −' *Abs* −−−> *op* −' *Abs* −−−> *op* −' *Rep*) *op* ∩ = *op* ∩
unfolding *fun-eq-iff*
by (*simp add*: *Quotient3-abs-rep*[*OF set-quotient*[*OF assms*]])

lemma *mem-prs*[*quot-preserve*]:
assumes *Quotient3 R Abs Rep*
shows (*Rep* −−−> *op* −' *Abs* −−−> *id*) *op* ∈ = *op* ∈
by (*simp add*: *fun-eq-iff Quotient3-abs-rep*[*OF assms*])

lemma *mem-rsp*[*quot-respect*]:
shows (*R* ===> *rel-vset R* ===> *op* =) *op* ∈ *op* ∈
by (*intro rel-funI*) (*simp add*: *rel-vset-def*)

end

76 Quotient infrastructure for the product type

theory *Quotient-Product*
imports *Quotient-Syntax*
begin

76.1 Rules for the Quotient package

lemma *map-prod-id* [*id-simps*]:
shows *map-prod id id* = *id*
by (*simp add*: *fun-eq-iff*)

lemma *rel-prod-eq* [*id-simps*]:
shows *rel-prod* (*op* =) (*op* =) = (*op* =)
by (*simp add*: *fun-eq-iff*)

lemma *prod-equivp* [*quot-equiv*]:
assumes *equivp R1*
assumes *equivp R2*
shows *equivp* (*rel-prod R1 R2*)
using *assms* **by** (*auto intro*!: *equivpI reflpI sympI transpI elim*!: *equivpE elim*: *reflpE sympE transpE*)

lemma *prod-quotient* [*quot-thm*]:
assumes *Quotient3 R1 Abs1 Rep1*
assumes *Quotient3 R2 Abs2 Rep2*
shows *Quotient3* (*rel-prod R1 R2*) (*map-prod Abs1 Abs2*) (*map-prod Rep1 Rep2*)
apply (*rule Quotient3I*)
apply (*simp add*: *map-prod.compositionality comp-def map-prod.identity Quotient3-abs-rep* [*OF assms*(*1*)] *Quotient3-abs-rep* [*OF assms*(*2*)])
apply (*simp add*: *split-paired-all Quotient3-rel-rep* [*OF assms*(*1*)] *Quotient3-rel-rep* [*OF assms*(*2*)])
using *Quotient3-rel* [*OF assms*(*1*)] *Quotient3-rel* [*OF assms*(*2*)]
apply (*auto simp add*: *split-paired-all*)
done

declare [[*mapQ3 prod* = (*rel-prod*, *prod-quotient*)]]

lemma *Pair-rsp* [*quot-respect*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
assumes *q2*: *Quotient3 R2 Abs2 Rep2*
shows (*R1* ===> *R2* ===> *rel-prod R1 R2*) *Pair Pair*
by (*rule Pair-transfer*)

lemma *Pair-prs* [*quot-preserve*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
assumes *q2*: *Quotient3 R2 Abs2 Rep2*
shows (*Rep1* −−−> *Rep2* −−−> (*map-prod Abs1 Abs2*)) *Pair* = *Pair*
apply(*simp add*: *fun-eq-iff*)
apply(*simp add*: *Quotient3-abs-rep*[*OF q1*] *Quotient3-abs-rep*[*OF q2*])
done

lemma *fst-rsp* [*quot-respect*]:
assumes *Quotient3 R1 Abs1 Rep1*
assumes *Quotient3 R2 Abs2 Rep2*
shows (*rel-prod R1 R2* ===> *R1*) *fst fst*
by *auto*

lemma *fst-prs* [*quot-preserve*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*

assumes *q2*: *Quotient3 R2 Abs2 Rep2*
shows (*map-prod Rep1 Rep2* −−−> *Abs1*) *fst* = *fst*
by (*simp add*: *fun-eq-iff Quotient3-abs-rep*[*OF q1*])

lemma *snd-rsp* [*quot-respect*]:
assumes *Quotient3 R1 Abs1 Rep1*
assumes *Quotient3 R2 Abs2 Rep2*
shows (*rel-prod R1 R2* ===> *R2*) *snd snd*
by *auto*

lemma *snd-prs* [*quot-preserve*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
assumes *q2*: *Quotient3 R2 Abs2 Rep2*
shows (*map-prod Rep1 Rep2* −−−> *Abs2*) *snd* = *snd*
by (*simp add*: *fun-eq-iff Quotient3-abs-rep*[*OF q2*])

lemma *case-prod-rsp* [*quot-respect*]:
shows ((*R1* ===> *R2* ===> (*op* =)) ===> (*rel-prod R1 R2*) ===> (*op* =)) *case-prod case-prod*
by (*rule case-prod-transfer*)

lemma *split-prs* [*quot-preserve*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
and *q2*: *Quotient3 R2 Abs2 Rep2*
shows (((*Abs1* −−−> *Abs2* −−−> *id*) −−−> *map-prod Rep1 Rep2* −−−> *id*) *case-prod*) = *case-prod*
by (*simp add*: *fun-eq-iff Quotient3-abs-rep*[*OF q1*] *Quotient3-abs-rep*[*OF q2*])

lemma [*quot-respect*]:
shows ((*R2* ===> *R2* ===> *op* =) ===> (*R1* ===> *R1* ===> *op* =) ===>
rel-prod R2 R1 ===> *rel-prod R2 R1* ===> *op* =) *rel-prod rel-prod*
by (*rule prod.rel-transfer*)

lemma [*quot-preserve*]:
assumes *q1*: *Quotient3 R1 abs1 rep1*
and *q2*: *Quotient3 R2 abs2 rep2*
shows ((*abs1* −−−> *abs1* −−−> *id*) −−−> (*abs2* −−−> *abs2* −−−> *id*) −−−>
map-prod rep1 rep2 −−−> *map-prod rep1 rep2* −−−> *id*) *rel-prod* = *rel-prod*
by (*simp add*: *fun-eq-iff Quotient3-abs-rep*[*OF q1*] *Quotient3-abs-rep*[*OF q2*])

lemma [*quot-preserve*]:
shows(*rel-prod* ((*rep1* −−−> *rep1* −−−> *id*) *R1*) ((*rep2* −−−> *rep2* −−−> *id*) *R2*)
(*l1*, *l2*) (*r1*, *r2*)) = (*R1* (*rep1 l1*) (*rep1 r1*) ∧ *R2* (*rep2 l2*) (*rep2 r2*))
by *simp*

declare *prod.inject*[*quot-preserve*]

end

77 Quotient infrastructure for the option type

theory *Quotient-Option*
imports *Quotient-Syntax*
begin

77.1 Rules for the Quotient package

lemma *rel-option-map1*:
rel-option R (*map-option f x*) *y* ⟷ *rel-option* (λ*x*. *R* (*f x*)) *x y*
by (*simp add*: *rel-option-iff split*: *option.split*)

lemma *rel-option-map2*:
rel-option R x (*map-option f y*) ⟷ *rel-option* (λ*x y*. *R x* (*f y*)) *x y*
by (*simp add*: *rel-option-iff split*: *option.split*)

declare
map-option.id [*id-simps*]
option.rel-eq [*id-simps*]

lemma *reflp-rel-option*:
reflp R ⟹ *reflp* (*rel-option R*)
unfolding *reflp-def split-option-all* **by** *simp*

lemma *option-symp*:
symp R ⟹ *symp* (*rel-option R*)
unfolding *symp-def split-option-all*
by (*simp only*: *option.rel-inject option.rel-distinct*) *fast*

lemma *option-transp*:
transp R ⟹ *transp* (*rel-option R*)
unfolding *transp-def split-option-all*
by (*simp only*: *option.rel-inject option.rel-distinct*) *fast*

lemma *option-equivp* [*quot-equiv*]:
equivp R ⟹ *equivp* (*rel-option R*)
by (*blast intro*: *equivpI reflp-rel-option option-symp option-transp elim*: *equivpE*)

lemma *option-quotient* [*quot-thm*]:
assumes *Quotient3 R Abs Rep*
shows *Quotient3* (*rel-option R*) (*map-option Abs*) (*map-option Rep*)
apply (*rule Quotient3I*)
apply (*simp-all add*: *option.map-comp comp-def option.map-id*[*unfolded id-def*] *option.rel-eq rel-option-map1 rel-option-map2 Quotient3-abs-rep* [*OF assms*] *Quotient3-rel-rep* [*OF assms*])
using *Quotient3-rel* [*OF assms*]

apply (*simp add*: *rel-option-iff split*: *option.split*)
done

declare [[*mapQ3 option* = (*rel-option*, *option-quotient*)]]

lemma *option-None-rsp* [*quot-respect*]:
assumes *q*: *Quotient3 R Abs Rep*
shows *rel-option R None None*
by (*rule option.ctr-transfer*(*1*))

lemma *option-Some-rsp* [*quot-respect*]:
assumes *q*: *Quotient3 R Abs Rep*
shows (*R* ===> *rel-option R*) *Some Some*
by (*rule option.ctr-transfer*(*2*))

lemma *option-None-prs* [*quot-preserve*]:
assumes *q*: *Quotient3 R Abs Rep*
shows *map-option Abs None* = *None*
by (*rule Option.option.map*(*1*))

lemma *option-Some-prs* [*quot-preserve*]:
assumes *q*: *Quotient3 R Abs Rep*
shows (*Rep* −−−> *map-option Abs*) *Some* = *Some*
apply(*simp add*: *fun-eq-iff*)
apply(*simp add*: *Quotient3-abs-rep*[*OF q*])
done

end

78 Quotient infrastructure for the list type

theory *Quotient-List*
imports *Quotient-Set Quotient-Product Quotient-Option*
begin

78.1 Rules for the Quotient package

lemma *map-id* [*id-simps*]:
map id = *id*
by (*fact List.map.id*)

lemma *list-all2-eq* [*id-simps*]:
list-all2 (*op* =) = (*op* =)
proof (*rule ext*)+
fix *xs ys*
show *list-all2* (*op* =) *xs ys* ⟷ *xs* = *ys*
by (*induct xs ys rule*: *list-induct2'*) *simp-all*
qed

lemma *reflp-list-all2*:
  assumes *reflp R*
  shows *reflp* (*list-all2 R*)
proof (*rule reflpI*)
  from *assms* **have** ∗: $\bigwedge$*xs*. *R xs xs* **by** (*rule reflpE*)
  fix *xs*
  show *list-all2 R xs xs*
    by (*induct xs*) (*simp-all add*: ∗)
qed

lemma *list-symp*:
  assumes *symp R*
  shows *symp* (*list-all2 R*)
proof (*rule sympI*)
  from *assms* **have** ∗: $\bigwedge$*xs ys*. *R xs ys* $\Longrightarrow$ *R ys xs* **by** (*rule sympE*)
  fix *xs ys*
  assume *list-all2 R xs ys*
  then show *list-all2 R ys xs*
    by (*induct xs ys rule*: *list-induct2'*) (*simp-all add*: ∗)
qed

lemma *list-transp*:
  assumes *transp R*
  shows *transp* (*list-all2 R*)
proof (*rule transpI*)
  from *assms* **have** ∗: $\bigwedge$*xs ys zs*. *R xs ys* $\Longrightarrow$ *R ys zs* $\Longrightarrow$ *R xs zs* **by** (*rule transpE*)
  fix *xs ys zs*
  assume *list-all2 R xs ys* **and** *list-all2 R ys zs*
  then show *list-all2 R xs zs*
    by (*induct arbitrary*: *zs*) (*auto simp*: *list-all2-Cons1 intro*: ∗)
qed

lemma *list-equivp* [*quot-equiv*]:
  equivp R $\Longrightarrow$ *equivp* (*list-all2 R*)
  by (*blast intro*: *equivpI reflp-list-all2 list-symp list-transp elim*: *equivpE*)

lemma *list-quotient3* [*quot-thm*]:
  assumes *Quotient3 R Abs Rep*
  shows *Quotient3* (*list-all2 R*) (*map Abs*) (*map Rep*)
proof (*rule Quotient3I*)
  from *assms* **have** $\bigwedge$*x*. *Abs* (*Rep x*) = *x* **by** (*rule Quotient3-abs-rep*)
  then show $\bigwedge$*xs*. *map Abs* (*map Rep xs*) = *xs* **by** (*simp add*: *comp-def*)
next
  from *assms* **have** $\bigwedge$*x y*. *R* (*Rep x*) (*Rep y*) $\longleftrightarrow$ *x* = *y* **by** (*rule Quotient3-rel-rep*)
  then show $\bigwedge$*xs*. *list-all2 R* (*map Rep xs*) (*map Rep xs*)
    by (*simp add*: *list-all2-map1 list-all2-map2 list-all2-eq*)
next
  fix *xs ys*
  from *assms* **have** $\bigwedge$*x y*. *R x x* $\wedge$ *R y y* $\wedge$ *Abs x* = *Abs y* $\longleftrightarrow$ *R x y* **by** (*rule*

Quotient3-rel)
  then show *list-all2 R xs ys* $\longleftrightarrow$ *list-all2 R xs xs* $\wedge$ *list-all2 R ys ys* $\wedge$ *map Abs xs* = *map Abs ys*
    by (*induct xs ys rule*: *list-induct2'*) *auto*
qed

declare [[*mapQ3 list* = (*list-all2*, *list-quotient3*)]]

lemma *cons-prs* [*quot-preserve*]:
  assumes *q*: *Quotient3 R Abs Rep*
  shows (*Rep* ---> (*map Rep*) ---> (*map Abs*)) (*op* #) = (*op* #)
  by (*auto simp add*: *fun-eq-iff comp-def Quotient3-abs-rep* [*OF q*])

lemma *cons-rsp* [*quot-respect*]:
  assumes *q*: *Quotient3 R Abs Rep*
  shows (*R* ===> *list-all2 R* ===> *list-all2 R*) (*op* #) (*op* #)
  by *auto*

lemma *nil-prs* [*quot-preserve*]:
  assumes *q*: *Quotient3 R Abs Rep*
  shows *map Abs* [] = []
  by *simp*

lemma *nil-rsp* [*quot-respect*]:
  assumes *q*: *Quotient3 R Abs Rep*
  shows *list-all2 R* [] []
  by *simp*

lemma *map-prs-aux*:
  assumes *a*: *Quotient3 R1 abs1 rep1*
  and *b*: *Quotient3 R2 abs2 rep2*
  shows (*map abs2*) (*map* ((*abs1* ---> *rep2*) *f*) (*map rep1 l*)) = *map f l*
  by (*induct l*)
    (*simp-all add*: *Quotient3-abs-rep*[*OF a*] *Quotient3-abs-rep*[*OF b*])

lemma *map-prs* [*quot-preserve*]:
  assumes *a*: *Quotient3 R1 abs1 rep1*
  and *b*: *Quotient3 R2 abs2 rep2*
  shows ((*abs1* ---> *rep2*) ---> (*map rep1*) ---> (*map abs2*)) *map* = *map*
  and ((*abs1* ---> *id*) ---> *map rep1* ---> *id*) *map* = *map*
  by (*simp-all only*: *fun-eq-iff map-prs-aux*[*OF a b*] *comp-def*)
    (*simp-all add*: *Quotient3-abs-rep*[*OF a*] *Quotient3-abs-rep*[*OF b*])

lemma *map-rsp* [*quot-respect*]:
  assumes *q1*: *Quotient3 R1 Abs1 Rep1*
  and *q2*: *Quotient3 R2 Abs2 Rep2*
  shows ((*R1* ===> *R2*) ===> (*list-all2 R1*) ===> *list-all2 R2*) *map map*
  and ((*R1* ===> *op* =) ===> (*list-all2 R1*) ===> *op* =) *map map*
  unfolding *list-all2-eq* [*symmetric*] **by** (*rule list.map-transfer*)+

lemma *foldr-prs-aux*:
assumes *a*: *Quotient3 R1 abs1 rep1*
and *b*: *Quotient3 R2 abs2 rep2*
shows *abs2* (*foldr* ((*abs1* −−−> *abs2* −−−> *rep2*) *f*) (*map rep1 l*) (*rep2 e*)) = *foldr f l e*
by (*induct l*) (*simp-all add*: *Quotient3-abs-rep*[*OF a*] *Quotient3-abs-rep*[*OF b*])

lemma *foldr-prs* [*quot-preserve*]:
assumes *a*: *Quotient3 R1 abs1 rep1*
and *b*: *Quotient3 R2 abs2 rep2*
shows ((*abs1* −−−> *abs2* −−−> *rep2*) −−−> (*map rep1*) −−−> *rep2* −−−> *abs2*) *foldr* = *foldr*
apply (*simp add*: *fun-eq-iff*)
by (*simp only*: *fun-eq-iff foldr-prs-aux*[*OF a b*])
(*simp*)

lemma *foldl-prs-aux*:
assumes *a*: *Quotient3 R1 abs1 rep1*
and *b*: *Quotient3 R2 abs2 rep2*
shows *abs1* (*foldl* ((*abs1* −−−> *abs2* −−−> *rep1*) *f*) (*rep1 e*) (*map rep2 l*)) = *foldl f e l*
by (*induct l arbitrary*:*e*) (*simp-all add*: *Quotient3-abs-rep*[*OF a*] *Quotient3-abs-rep*[*OF b*])

lemma *foldl-prs* [*quot-preserve*]:
assumes *a*: *Quotient3 R1 abs1 rep1*
and *b*: *Quotient3 R2 abs2 rep2*
shows ((*abs1* −−−> *abs2* −−−> *rep1*) −−−> *rep1* −−−> (*map rep2*) −−−> *abs1*) *foldl* = *foldl*
by (*simp add*: *fun-eq-iff foldl-prs-aux* [*OF a b*])

lemma *foldl-rsp*[*quot-respect*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
and *q2*: *Quotient3 R2 Abs2 Rep2*
shows ((*R1* ===> *R2* ===> *R1*) ===> *R1* ===> *list-all2 R2* ===> *R1*) *foldl foldl*
by (*rule foldl-transfer*)

lemma *foldr-rsp*[*quot-respect*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
and *q2*: *Quotient3 R2 Abs2 Rep2*
shows ((*R1* ===> *R2* ===> *R2*) ===> *list-all2 R1* ===> *R2* ===> *R2*) *foldr foldr*
by (*rule foldr-transfer*)

lemma *list-all2-rsp*:
assumes *r*: $\forall x\ y.\ R\ x\ y \longrightarrow (\forall a\ b.\ R\ a\ b \longrightarrow S\ x\ a = T\ y\ b)$

and *l1*: *list-all2 R x y*
and *l2*: *list-all2 R a b*
shows *list-all2 S x a = list-all2 T y b*
using *l1 l2*
by (*induct arbitrary*: *a b rule*: *list-all2-induct*,
auto simp: *list-all2-Cons1 list-all2-Cons2 r*)

lemma [*quot-respect*]:
((*R* ===> *R* ===> *op* =) ===> *list-all2 R* ===> *list-all2 R* ===> *op* =) *list-all2 list-all2*
by (*rule list.rel-transfer*)

lemma [*quot-preserve*]:
assumes *a*: *Quotient3 R abs1 rep1*
shows ((*abs1* ---> *abs1* ---> *id*) ---> *map rep1* ---> *map rep1* ---> *id*) *list-all2* = *list-all2*
apply (*simp add*: *fun-eq-iff*)
apply *clarify*
apply (*induct-tac xa xb rule*: *list-induct2'*)
apply (*simp-all add*: *Quotient3-abs-rep*[*OF a*])
done

lemma [*quot-preserve*]:
assumes *a*: *Quotient3 R abs1 rep1*
shows (*list-all2* ((*rep1* ---> *rep1* ---> *id*) *R*) *l m*) = (*l* = *m*)
by (*induct l m rule*: *list-induct2'*) (*simp-all add*: *Quotient3-rel-rep*[*OF a*])

lemma *list-all2-find-element*:
assumes *a*: $x \in set\ a$
and *b*: *list-all2 R a b*
shows $\exists y. (y \in set\ b \wedge R\ x\ y)$
using *b a* **by** *induct auto*

lemma *list-all2-refl*:
assumes *a*: $\bigwedge x\ y.\ R\ x\ y = (R\ x = R\ y)$
shows *list-all2 R x x*
by (*induct x*) (*auto simp add*: *a*)

end

79 Quotient infrastructure for the sum type

theory *Quotient-Sum*
imports *Quotient-Syntax*
begin

79.1 Rules for the Quotient package

lemma *rel-sum-map1*:

rel-sum R1 R2 (map-sum f1 f2 x) y ⟷ *rel-sum (λx. R1 (f1 x)) (λx. R2 (f2 x)) x y*
by (*rule sum.rel-map*(*1*))

lemma *rel-sum-map2*:
rel-sum R1 R2 x (map-sum f1 f2 y) ⟷ *rel-sum (λx y. R1 x (f1 y)) (λx y. R2 x (f2 y)) x y*
by (*rule sum.rel-map*(*2*))

lemma *map-sum-id* [*id-simps*]:
map-sum id id = id
by (*simp add*: *id-def map-sum.identity fun-eq-iff*)

lemma *rel-sum-eq* [*id-simps*]:
rel-sum (op =) (op =) = (op =)
by (*rule sum.rel-eq*)

lemma *reflp-rel-sum*:
reflp R1 ⟹ *reflp R2* ⟹ *reflp (rel-sum R1 R2)*
unfolding *reflp-def split-sum-all rel-sum-simps* **by** *fast*

lemma *sum-symp*:
symp R1 ⟹ *symp R2* ⟹ *symp (rel-sum R1 R2)*
unfolding *symp-def split-sum-all rel-sum-simps* **by** *fast*

lemma *sum-transp*:
transp R1 ⟹ *transp R2* ⟹ *transp (rel-sum R1 R2)*
unfolding *transp-def split-sum-all rel-sum-simps* **by** *fast*

lemma *sum-equivp* [*quot-equiv*]:
equivp R1 ⟹ *equivp R2* ⟹ *equivp (rel-sum R1 R2)*
by (*blast intro*: *equivpI reflp-rel-sum sum-symp sum-transp elim*: *equivpE*)

lemma *sum-quotient* [*quot-thm*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
assumes *q2*: *Quotient3 R2 Abs2 Rep2*
shows *Quotient3 (rel-sum R1 R2) (map-sum Abs1 Abs2) (map-sum Rep1 Rep2)*
apply (*rule Quotient3I*)
apply (*simp-all add*: *map-sum.compositionality comp-def map-sum.identity rel-sum-eq rel-sum-map1 rel-sum-map2*
Quotient3-abs-rep [*OF q1*] *Quotient3-rel-rep* [*OF q1*] *Quotient3-abs-rep* [*OF q2*] *Quotient3-rel-rep* [*OF q2*])
using *Quotient3-rel* [*OF q1*] *Quotient3-rel* [*OF q2*]
apply (*fastforce elim!*: *rel-sum.cases simp add*: *comp-def split*: *sum.split*)
done

declare [[*mapQ3 sum* = (*rel-sum*, *sum-quotient*)]]

lemma *sum-Inl-rsp* [*quot-respect*]:

assumes *q1*: *Quotient3 R1 Abs1 Rep1*
assumes *q2*: *Quotient3 R2 Abs2 Rep2*
shows (*R1* ===> *rel-sum R1 R2*) *Inl Inl*
by *auto*

lemma *sum-Inr-rsp* [*quot-respect*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
assumes *q2*: *Quotient3 R2 Abs2 Rep2*
shows (*R2* ===> *rel-sum R1 R2*) *Inr Inr*
by *auto*

lemma *sum-Inl-prs* [*quot-preserve*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
assumes *q2*: *Quotient3 R2 Abs2 Rep2*
shows (*Rep1* −−−> *map-sum Abs1 Abs2*) *Inl* = *Inl*
apply(*simp add*: *fun-eq-iff*)
apply(*simp add*: *Quotient3-abs-rep*[*OF q1*])
done

lemma *sum-Inr-prs* [*quot-preserve*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
assumes *q2*: *Quotient3 R2 Abs2 Rep2*
shows (*Rep2* −−−> *map-sum Abs1 Abs2*) *Inr* = *Inr*
apply(*simp add*: *fun-eq-iff*)
apply(*simp add*: *Quotient3-abs-rep*[*OF q2*])
done

end

80 Quotient types

theory *Quotient-Type*
imports *Main*
begin

We introduce the notion of quotient types over equivalence relations via type classes.

80.1 Equivalence relations and quotient types

Type class *equiv* models equivalence relations $\sim$ :: $'a \Rightarrow 'a \Rightarrow bool$.

class *eqv* =
fixes *eqv* :: $'a \Rightarrow 'a \Rightarrow bool$ (**infixl** $\sim$ *50*)

class *equiv* = *eqv* +
assumes *equiv-refl* [*intro*]: $x \sim x$
and *equiv-trans* [*trans*]: $x \sim y \Longrightarrow y \sim z \Longrightarrow x \sim z$
and *equiv-sym* [*sym*]: $x \sim y \Longrightarrow y \sim x$

begin

lemma *equiv-not-sym* [*sym*]: $\neg\ x \sim y \Longrightarrow \neg\ y \sim x$
proof –
  assume $\neg\ x \sim y$
  then show $\neg\ y \sim x$ **by** (*rule contrapos-nn*) (*rule equiv-sym*)
qed

lemma *not-equiv-trans1* [*trans*]: $\neg\ x \sim y \Longrightarrow y \sim z \Longrightarrow \neg\ x \sim z$
proof –
  assume $\neg\ x \sim y$ **and** $y \sim z$
  show $\neg\ x \sim z$
  proof
    assume $x \sim z$
    also from ‹$y \sim z$› **have** $z \sim y$ **..**
    finally have $x \sim y$ **.**
    with ‹$\neg\ x \sim y$› **show** *False* **by** *contradiction*
  qed
qed

lemma *not-equiv-trans2* [*trans*]: $x \sim y \Longrightarrow \neg\ y \sim z \Longrightarrow \neg\ x \sim z$
proof –
  assume $\neg\ y \sim z$
  then have $\neg\ z \sim y$ **..**
  also
  assume $x \sim y$
  then have $y \sim x$ **..**
  finally have $\neg\ z \sim x$ **.**
  then show $\neg\ x \sim z$ **..**
qed

end

The quotient type *'a quot* consists of all *equivalence classes* over elements of the base type *'a*.

definition (**in** *eqv*) *quot* = $\{\{x.\ a \sim x\} \mid a.\ True\}$

typedef (**overloaded**) *'a quot* = *quot* :: *'a*::*eqv set set*
  unfolding *quot-def* **by** *blast*

lemma *quotI* [*intro*]: $\{x.\ a \sim x\} \in quot$
  unfolding *quot-def* **by** *blast*

lemma *quotE* [*elim*]:
  assumes $R \in quot$
  obtains a **where** $R = \{x.\ a \sim x\}$
  using *assms* **unfolding** *quot-def* **by** *blast*

Abstracted equivalence classes are the canonical representation of elements of a quotient type.

```
definition class :: 'a::equiv ⇒ 'a quot  (⌊-⌋)
  where ⌊a⌋ = Abs-quot {x. a ∼ x}

theorem quot-exhaust: ∃ a. A = ⌊a⌋
proof (cases A)
  fix R
  assume R: A = Abs-quot R
  assume R ∈ quot
  then have ∃ a. R = {x. a ∼ x} by blast
  with R have ∃ a. A = Abs-quot {x. a ∼ x} by blast
  then show ?thesis unfolding class-def .
qed

lemma quot-cases [cases type: quot]:
  obtains a where A = ⌊a⌋
  using quot-exhaust by blast
```

80.2 Equality on quotients

Equality of canonical quotient elements coincides with the original relation.

```
theorem quot-equality [iff?]: ⌊a⌋ = ⌊b⌋ ⟷ a ∼ b
proof
  assume eq: ⌊a⌋ = ⌊b⌋
  show a ∼ b
  proof −
    from eq have {x. a ∼ x} = {x. b ∼ x}
      by (simp only: class-def Abs-quot-inject quotI)
    moreover have a ∼ a ..
    ultimately have a ∈ {x. b ∼ x} by blast
    then have b ∼ a by blast
    then show ?thesis ..
  qed
next
  assume ab: a ∼ b
  show ⌊a⌋ = ⌊b⌋
  proof −
    have {x. a ∼ x} = {x. b ∼ x}
    proof (rule Collect-cong)
      fix x show (a ∼ x) = (b ∼ x)
      proof
        from ab have b ∼ a ..
        also assume a ∼ x
        finally show b ∼ x .
      next
        note ab
        also assume b ∼ x
        finally show a ∼ x .
      qed
    qed
```

```
    then show ?thesis by (simp only: class-def)
  qed
qed
```

80.3 Picking representing elements

```
definition pick :: 'a::equiv quot ⇒ 'a
  where pick A = (SOME a. A = ⌊a⌋)

theorem pick-equiv [intro]: pick ⌊a⌋ ∼ a
proof (unfold pick-def)
  show (SOME x. ⌊a⌋ = ⌊x⌋) ∼ a
  proof (rule someI2)
    show ⌊a⌋ = ⌊a⌋ ..
    fix x assume ⌊a⌋ = ⌊x⌋
    then have a ∼ x ..
    then show x ∼ a ..
  qed
qed

theorem pick-inverse [intro]: ⌊pick A⌋ = A
proof (cases A)
  fix a assume a: A = ⌊a⌋
  then have pick A ∼ a by (simp only: pick-equiv)
  then have ⌊pick A⌋ = ⌊a⌋ ..
  with a show ?thesis by simp
qed
```

The following rules support canonical function definitions on quotient types (with up to two arguments). Note that the stripped-down version without additional conditions is sufficient most of the time.

```
theorem quot-cond-function:
  assumes eq: ⋀X Y. P X Y ⟹ f X Y ≡ g (pick X) (pick Y)
    and cong: ⋀x x' y y'. ⌊x⌋ = ⌊x'⌋ ⟹ ⌊y⌋ = ⌊y'⌋
      ⟹ P ⌊x⌋ ⌊y⌋ ⟹ P ⌊x'⌋ ⌊y'⌋ ⟹ g x y = g x' y'
    and P: P ⌊a⌋ ⌊b⌋
  shows f ⌊a⌋ ⌊b⌋ = g a b
proof −
  from eq and P have f ⌊a⌋ ⌊b⌋ = g (pick ⌊a⌋) (pick ⌊b⌋) by (simp only:)
  also have ... = g a b
  proof (rule cong)
    show ⌊pick ⌊a⌋⌋ = ⌊a⌋ ..
    moreover
    show ⌊pick ⌊b⌋⌋ = ⌊b⌋ ..
    moreover
    show P ⌊a⌋ ⌊b⌋ by (rule P)
    ultimately show P ⌊pick ⌊a⌋⌋ ⌊pick ⌊b⌋⌋ by (simp only:)
  qed
  finally show ?thesis .
```

qed

theorem *quot-function*:
  assumes $\bigwedge X\ Y.\ f\ X\ Y \equiv g\ (pick\ X)\ (pick\ Y)$
    and $\bigwedge x\ x'\ y\ y'.\ \lfloor x\rfloor = \lfloor x'\rfloor \Longrightarrow \lfloor y\rfloor = \lfloor y'\rfloor \Longrightarrow g\ x\ y = g\ x'\ y'$
  shows $f\ \lfloor a\rfloor\ \lfloor b\rfloor = g\ a\ b$
  using *assms* **and** *TrueI*
  by (*rule quot-cond-function*)

theorem *quot-function'*:
  $(\bigwedge X\ Y.\ f\ X\ Y \equiv g\ (pick\ X)\ (pick\ Y)) \Longrightarrow$
    $(\bigwedge x\ x'\ y\ y'.\ x \sim x' \Longrightarrow y \sim y' \Longrightarrow g\ x\ y = g\ x'\ y') \Longrightarrow$
    $f\ \lfloor a\rfloor\ \lfloor b\rfloor = g\ a\ b$
  by (*rule quot-function*) (*simp-all only*: *quot-equality*)

end

81 Ramsey's Theorem

theory *Ramsey*
imports *Main Infinite-Set*
begin

81.1 Finite Ramsey theorem(s)

To distinguish the finite and infinite ones, lower and upper case names are used.

This is the most basic version in terms of cliques and independent sets, i.e. the version for graphs and 2 colours.

definition $clique\ V\ E = (\forall v\in V.\ \forall w\in V.\ v\neq w \longrightarrow \{v,w\} : E)$
definition $indep\ V\ E = (\forall v\in V.\ \forall w\in V.\ v\neq w \longrightarrow \neg\ \{v,w\} : E)$

lemma *ramsey2*:
  $\exists r\geq 1.\ \forall\ (V::'a\ set)\ (E::'a\ set\ set).\ finite\ V \wedge card\ V \geq r \longrightarrow$
  $(\exists\ R \subseteq V.\ card\ R = m \wedge clique\ R\ E \vee card\ R = n \wedge indep\ R\ E)$
  (**is** $\exists r\geq 1.\ ?R\ m\ n\ r$)
proof(*induct* $k == m+n$ *arbitrary*: $m\ n$)
  case *0*
  show *?case* (**is** $EX\ r.\ ?R\ r$)
  proof
    show *?R 1* **using** *0*
      by (*clarsimp simp*: *indep-def*)(*metis card.empty emptyE empty-subsetI*)
  qed
next
  case (*Suc k*)
  { assume $m=0$
    have *?case* (**is** $EX\ r.\ ?R\ r$)
    proof

```
    show ?R 1 using ‹m=0›
      by (simp add:clique-def)(metis card.empty emptyE empty-subsetI)
  qed
} moreover
{ assume n=0
  have ?case (is EX r. ?R r)
  proof
    show ?R 1 using ‹n=0›
      by (simp add:indep-def)(metis card.empty emptyE empty-subsetI)
  qed
} moreover
{ assume m≠0 n≠0
 then have k = (m − 1) + n k = m + (n − 1) using ‹Suc k = m+n› by auto
  from Suc(1)[OF this(1)] Suc(1)[OF this(2)]
  obtain r1 r2 where r1≥1 r2≥1 ?R (m − 1) n r1 ?R m (n − 1) r2
    by auto
  then have r1+r2 ≥ 1 by arith
  moreover
  have ?R m n (r1+r2) (is ALL V E. - ⟶ ?EX V E m n)
  proof clarify
    fix V :: 'a set and E :: 'a set set
    assume finite V r1+r2 ≤ card V
    with ‹r1≥1› have V ≠ {} by auto
    then obtain v where v : V by blast
    let ?M = {w : V. w≠v & {v,w} : E}
    let ?N = {w : V. w≠v & {v,w} ~: E}
    have V = insert v (?M ∪ ?N) using ‹v : V› by auto
    then have card V = card(insert v (?M ∪ ?N)) by metis
    also have . . . = card ?M + card ?N + 1 using ‹finite V›
      by(fastforce intro: card-Un-disjoint)
    finally have card V = card ?M + card ?N + 1 .
    then have r1+r2 ≤ card ?M + card ?N + 1 using ‹r1+r2 ≤ card V› by
simp
    then have r1 ≤ card ?M ∨ r2 ≤ card ?N by arith
    moreover
    { assume r1 ≤ card ?M
      moreover have finite ?M using ‹finite V› by auto
     ultimately have ?EX ?M E (m − 1) n using ‹?R (m − 1) n r1› by blast
      then obtain R where R ⊆ ?M v ~: R and
        CI: card R = m − 1 ∧ clique R E ∨
             card R = n ∧ indep R E (is ?C ∨ ?I)
        by blast
      have R <= V using ‹R <= ?M› by auto
      have finite R using ‹finite V› ‹R ⊆ V› by (metis finite-subset)
      { assume ?I
        with ‹R <= V› have ?EX V E m n by blast
      } moreover
      { assume ?C
        then have clique (insert v R) E using ‹R <= ?M›
```

```
          by(auto simp:clique-def insert-commute)
        moreover have card(insert v R) = m
          using ‹?C› ‹finite R› ‹v ~: R› ‹m≠0› by simp
          ultimately have ?EX V E m n using ‹R <= V› ‹v : V› by (metis
insert-subset)
      } ultimately have ?EX V E m n using CI by blast
    } moreover
    { assume r2 ≤ card ?N
      moreover have finite ?N using ‹finite V› by auto
      ultimately have ?EX ?N E m (n - 1) using ‹?R m (n - 1) r2› by blast
      then obtain R where R ⊆ ?N v ~: R and
        CI: card R = m ∧ clique R E ∨
            card R = n - 1 ∧ indep R E (is ?C ∨ ?I)
        by blast
      have R <= V using ‹R <= ?N› by auto
      have finite R using ‹finite V› ‹R ⊆ V› by (metis finite-subset)
      { assume ?C
        with ‹R <= V› have ?EX V E m n by blast
      } moreover
      { assume ?I
        then have indep (insert v R) E using ‹R <= ?N›
          by(auto simp:indep-def insert-commute)
        moreover have card(insert v R) = n
          using ‹?I› ‹finite R› ‹v ~: R› ‹n≠0› by simp
          ultimately have ?EX V E m n using ‹R <= V› ‹v : V› by (metis
insert-subset)
      } ultimately have ?EX V E m n using CI by blast
    } ultimately show ?EX V E m n by blast
  qed
  ultimately have ?case by blast
 } ultimately show ?case by blast
qed
```

81.2 Preliminaries

81.2.1 "Axiom" of Dependent Choice

```
primrec choice :: ('a => bool) => ('a * 'a) set => nat => 'a where
 — An integer-indexed chain of choices
   choice-0:   choice P r 0 = (SOME x. P x)
 | choice-Suc: choice P r (Suc n) = (SOME y. P y & (choice P r n, y) ∈ r)

lemma choice-n:
 assumes P0: P x0
     and Pstep: !!x. P x ==> ∃ y. P y & (x,y) ∈ r
 shows P (choice P r n)
proof (induct n)
 case 0 show ?case by (force intro: someI P0)
next
 case Suc then show ?case by (auto intro: someI2-ex [OF Pstep])
```

qed

lemma *dependent-choice*:
  assumes *trans*: *trans r*
    and *P0*: *P x0*
    and *Pstep*: !!*x*. *P x* ==> ∃ *y*. *P y* & (*x*,*y*) ∈ *r*
  obtains *f* :: *nat* => *'a* **where**
    !!*n*. *P* (*f n*) **and** !!*n m*. *n* < *m* ==> (*f n*, *f m*) ∈ *r*
proof
  fix *n*
  show *P* (*choice P r n*) **by** (*blast intro*: *choice-n* [*OF P0 Pstep*])
next
  have *PSuc*: ∀ *n*. (*choice P r n*, *choice P r* (*Suc n*)) ∈ *r*
    using *Pstep* [*OF choice-n* [*OF P0 Pstep*]]
    by (*auto intro*: *someI2-ex*)
  fix *n m* :: *nat*
  assume *less*: *n* < *m*
  show (*choice P r n*, *choice P r m*) ∈ *r* **using** *PSuc*
    by (*auto intro*: *less-Suc-induct* [*OF less*] *transD* [*OF trans*])
qed

81.2.2 Partitions of a Set

definition *part* :: *nat* => *nat* => *'a set* => (*'a set* => *nat*) => *bool*
  — the function *f* partitions the *r*-subsets of the typically infinite set *Y* into *s* distinct categories.
where
  part r s Y f = (∀ *X*. *X* ⊆ *Y* & *finite X* & *card X* = *r* −−> *f X* < *s*)

For induction, we decrease the value of *r* in partitions.

lemma *part-Suc-imp-part*:
    [| *infinite Y*; *part* (*Suc r*) *s Y f*; *y* ∈ *Y* |]
    ==> *part r s* (*Y* − {*y*}) (%*u*. *f* (*insert y u*))
  apply(*simp add*: *part-def*, *clarify*)
  apply(*drule-tac x*=*insert y X* **in** *spec*)
  apply(*force*)
  done

lemma *part-subset*: *part r s YY f* ==> *Y* ⊆ *YY* ==> *part r s Y f*
  unfolding *part-def* **by** *blast*

81.3 Ramsey's Theorem: Infinitary Version

lemma *Ramsey-induction*:
  fixes *s* **and** *r*::*nat*
  shows
  !!(*YY*::*'a set*) (*f*::*'a set* => *nat*).
    [|*infinite YY*; *part r s YY f*|]
    ==> ∃ *Y' t'*. *Y'* ⊆ *YY* & *infinite Y'* & *t'* < *s* &
          (∀ *X*. *X* ⊆ *Y'* & *finite X* & *card X* = *r* −−> *f X* = *t'*)

proof (*induct r*)
  case *0*
  then show *?case* **by** (*auto simp add*: *part-def card-eq-0-iff cong*: *conj-cong*)
next
  case (*Suc r*)
  show *?case*
  proof −
    from *Suc.prems infinite-imp-nonempty* **obtain** *yy* **where** *yy*: $yy \in YY$ **by** *blast*
    let *?ramr* = $\{((y,Y,t),(y',Y',t')).\ y' \in Y\ \&\ Y' \subseteq Y\}$
    let *?propr* = $\%(y,Y,t).$
        $y \in YY\ \&\ y \notin Y\ \&\ Y \subseteq YY\ \&$ *infinite* $Y\ \&\ t < s$
        $\&\ (\forall X.\ X \subseteq Y\ \&$ *finite* $X\ \&$ *card* $X = r \dashrightarrow (f$ *o insert* $y)\ X = t)$
    have *infYY'*: *infinite* $(YY - \{yy\})$ **using** *Suc.prems* **by** *auto*
    have *partf'*: *part r s* $(YY - \{yy\})$ $(f \circ$ *insert yy*$)$
      by (*simp add*: *o-def part-Suc-imp-part yy Suc.prems*)
    have *transr*: *trans ?ramr* **by** (*force simp add*: *trans-def*)
    from *Suc.hyps* [*OF infYY' partf'*]
    obtain *Y0* **and** *t0*
    where $Y0 \subseteq YY - \{yy\}$ *infinite Y0* $t0 < s$
        $\forall X.\ X \subseteq Y0 \wedge$ *finite* $X \wedge$ *card* $X = r \longrightarrow (f \circ$ *insert yy*$)\ X = t0$
      by *blast*
    with *yy* **have** *propr0*: *?propr*$(yy,Y0,t0)$ **by** *blast*
    have *proprstep*: $\bigwedge x.$ *?propr* $x \Longrightarrow \exists y.$ *?propr* $y \wedge (x,\ y) \in$ *?ramr*
    proof −
      fix *x*
      assume *px*: *?propr x* **then show** *?thesis x*
      proof (*cases x*)
        case (*fields yx Yx tx*)
        then obtain *yx'* **where** *yx'*: $yx' \in Yx$ **using** *px*
            by (*blast dest*: *infinite-imp-nonempty*)
        have *infYx'*: *infinite* $(Yx - \{yx'\})$ **using** *fields px* **by** *auto*
        with *fields px yx' Suc.prems*
        have *partfx'*: *part r s* $(Yx - \{yx'\})$ $(f \circ$ *insert yx'*$)$
          by (*simp add*: *o-def part-Suc-imp-part part-subset* [**where** $YY = YY$ **and** $Y = Yx$])
        from *Suc.hyps* [*OF infYx' partfx'*]
        obtain *Y'* **and** *t'*
        where *Y'*: $Y' \subseteq Yx - \{yx'\}$ *infinite Y'* $t' < s$
            $\forall X.\ X \subseteq Y' \wedge$ *finite* $X \wedge$ *card* $X = r \longrightarrow (f \circ$ *insert yx'*$)\ X = t'$
          by *blast*
        show *?thesis*
        proof
          show *?propr* $(yx',Y',t')\ \&\ (x,\ (yx',Y',t')) \in$ *?ramr*
            using *fields Y' yx' px* **by** *blast*
        qed
      qed
    qed
    from *dependent-choice* [*OF transr propr0 proprstep*]

```
obtain g where pg: !!n::nat.  ?propr (g n)
  and rg: !!n m. n<m ==> (g n, g m) ∈ ?ramr by blast
let ?gy = fst o g
let ?gt = snd o snd o g
have rangeg: ∃ k. range ?gt ⊆ {..<k}
proof (intro exI subsetI)
  fix x
  assume x ∈ range ?gt
  then obtain n where x = ?gt n ..
  with pg [of n] show x ∈ {..<s} by (cases g n) auto
qed
have finite (range ?gt)
  by (simp add: finite-nat-iff-bounded rangeg)
then obtain s′ and n′
  where s′: s′ = ?gt n′
    and infeqs′: infinite {n. ?gt n = s′}
 by (rule inf-img-fin-domE) (auto simp add: vimage-def intro: infinite-UNIV-nat)
with pg [of n′] have less′: s′<s by (cases g n′) auto
have inj-gy: inj ?gy
proof (rule linorder-injI)
  fix m m′ :: nat assume less: m < m′ show ?gy m ≠ ?gy m′
    using rg [OF less] pg [of m] by (cases g m, cases g m′) auto
qed
show ?thesis
proof (intro exI conjI)
  show ?gy ‘ {n. ?gt n = s′} ⊆ YY using pg
    by (auto simp add: Let-def split-beta)
  show infinite (?gy ‘ {n. ?gt n = s′}) using infeqs′
    by (blast intro: inj-gy [THEN subset-inj-on] dest: finite-imageD)
  show s′ < s by (rule less′)
  show ∀ X. X ⊆ ?gy ‘ {n. ?gt n = s′} & finite X & card X = Suc r
      −−> f X = s′
  proof −
    {fix X
     assume X ⊆ ?gy ‘ {n. ?gt n = s′}
        and cardX: finite X card X = Suc r
    then obtain AA where AA: AA ⊆ {n. ?gt n = s′} and Xeq: X = ?gy‘AA
        by (auto simp add: subset-image-iff)
     with cardX have AA≠{} by auto
    then have AAleast: (LEAST x. x ∈ AA) ∈ AA by (auto intro: LeastI-ex)
     have f X = s′
     proof (cases g (LEAST x. x ∈ AA))
       case (fields ya Ya ta)
       with AAleast Xeq
       have ya: ya ∈ X by (force intro!: rev-image-eqI)
      then have f X = f (insert ya (X − {ya})) by (simp add: insert-absorb)
       also have ... = ta
       proof −
         have X − {ya} ⊆ Ya
```

```
          proof
            fix x assume x: x ∈ X − {ya}
            then obtain a′ where xeq: x = ?gy a′ and a′: a′ ∈ AA
              by (auto simp add: Xeq)
            then have a′ ≠ (LEAST x. x ∈ AA) using x fields by auto
            then have lessa′: (LEAST x. x ∈ AA) < a′
              using Least-le [of %x. x ∈ AA, OF a′] by arith
            show x ∈ Ya using xeq fields rg [OF lessa′] by auto
          qed
          moreover
          have card (X − {ya}) = r
            by (simp add: cardX ya)
          ultimately show ?thesis
            using pg [of LEAST x. x ∈ AA] fields cardX
            by (clarsimp simp del:insert-Diff-single)
        qed
        also have ... = s′ using AA AAleast fields by auto
        finally show ?thesis .
      qed}
     then show ?thesis by blast
    qed
  qed
 qed
qed


theorem Ramsey:
 fixes s r :: nat and Z::′a set and f::′a set => nat
 shows
  [|infinite Z;
    ∀X. X ⊆ Z & finite X & card X = r −−> f X < s|]
 ==> ∃Y t. Y ⊆ Z & infinite Y & t < s
          & (∀X. X ⊆ Y & finite X & card X = r −−> f X = t)
by (blast intro: Ramsey-induction [unfolded part-def])


corollary Ramsey2:
 fixes s::nat and Z::′a set and f::′a set => nat
 assumes infZ: infinite Z
    and part: ∀x∈Z. ∀y∈Z. x≠y −−> f{x,y} < s
 shows
  ∃Y t. Y ⊆ Z & infinite Y & t < s & (∀x∈Y. ∀y∈Y. x≠y −−> f{x,y} = t)
proof −
 have part2: ∀X. X ⊆ Z & finite X & card X = 2 −−> f X < s
   using part by (fastforce simp add: eval-nat-numeral card-Suc-eq)
 obtain Y t
   where *: Y ⊆ Z infinite Y t < s
        (∀X. X ⊆ Y & finite X & card X = 2 −−> f X = t)
   by (insert Ramsey [OF infZ part2]) auto
```

```
  then have ∀x∈Y. ∀y∈Y. x ≠ y ⟶ f {x, y} = t by auto
  with * show ?thesis by iprover
qed
```

81.4 Disjunctive Well-Foundedness

An application of Ramsey's theorem to program termination. See [2].

```
definition disj-wf :: ('a * 'a)set => bool
  where disj-wf r = (∃T. ∃n::nat. (∀i<n. wf(T i)) & r = (⋃i<n. T i))

definition transition-idx :: [nat => 'a, nat => ('a*'a)set, nat set] => nat
  where
    transition-idx s T A =
      (LEAST k. ∃i j. A = {i,j} & i<j & (s j, s i) ∈ T k)


lemma transition-idx-less:
    [|i<j; (s j, s i) ∈ T k; k<n|] ==> transition-idx s T {i,j} < n
apply (subgoal-tac transition-idx s T {i, j} ≤ k, simp)
apply (simp add: transition-idx-def, blast intro: Least-le)
done

lemma transition-idx-in:
    [|i<j; (s j, s i) ∈ T k|] ==> (s j, s i) ∈ T (transition-idx s T {i,j})
apply (simp add: transition-idx-def doubleton-eq-iff conj-disj-distribR
            cong: conj-cong)
apply (erule LeastI)
done
```

To be equal to the union of some well-founded relations is equivalent to being the subset of such a union.

```
lemma disj-wf:
     disj-wf(r) = (∃T. ∃n::nat. (∀i<n. wf(T i)) & r ⊆ (⋃i<n. T i))
apply (auto simp add: disj-wf-def)
apply (rule-tac x=%i. T i Int r in exI)
apply (rule-tac x=n in exI)
apply (force simp add: wf-Int1)
done

theorem trans-disj-wf-implies-wf:
  assumes transr: trans r
      and dwf:    disj-wf(r)
  shows wf r
proof (simp only: wf-iff-no-infinite-down-chain, rule notI)
  assume ∃s. ∀i. (s (Suc i), s i) ∈ r
  then obtain s where sSuc: ∀i. (s (Suc i), s i) ∈ r ..
  have s: !!i j. i < j ==> (s j, s i) ∈ r
  proof −
    fix i and j::nat
```

assume *less*: $i<j$
then show $(s\ j,\ s\ i) \in r$
proof (*rule less-Suc-induct*)
show $\bigwedge i.\ (s\ (Suc\ i),\ s\ i) \in r$ **by** (*simp add*: *sSuc*)
show $\bigwedge i\ j\ k.\ [\![(s\ j,\ s\ i) \in r;\ (s\ k,\ s\ j) \in r]\!] \Longrightarrow (s\ k,\ s\ i) \in r$
using *transr* **by** (*unfold trans-def*, *blast*)
qed
qed
from *dwf*
obtain T **and** $n{::}nat$ **where** *wfT*: $\forall k<n.\ wf(T\ k)$ **and** r: $r = (\bigcup k<n.\ T\ k)$
by (*auto simp add*: *disj-wf-def*)
have *s-in-T*: $\bigwedge i\ j.\ i<j\ ==>\ \exists k.\ (s\ j,\ s\ i) \in T\ k\ \&\ k<n$
proof −
fix i **and** $j{::}nat$
assume *less*: $i<j$
then have $(s\ j,\ s\ i) \in r$ **by** (*rule s* [*of i j*])
then show $\exists k.\ (s\ j,\ s\ i) \in T\ k\ \&\ k<n$ **by** (*auto simp add*: r)
qed
have *trless*: $!!i\ j.\ i\neq j\ ==>\ transition\text{-}idx\ s\ T\ \{i,j\} < n$
apply (*auto simp add*: *linorder-neq-iff*)
apply (*blast dest*: *s-in-T transition-idx-less*)
apply (*subst insert-commute*)
apply (*blast dest*: *s-in-T transition-idx-less*)
done
have
$\exists K\ k.\ K \subseteq UNIV\ \&\ infinite\ K\ \&\ k < n\ \&$
$(\forall i{\in}K.\ \forall j{\in}K.\ i\neq j\ -->\ transition\text{-}idx\ s\ T\ \{i,j\} = k)$
by (*rule Ramsey2*) (*auto intro*: *trless infinite-UNIV-nat*)
then obtain K **and** k
where *infK*: *infinite* K **and** *less*: $k < n$ **and**
allk: $\forall i{\in}K.\ \forall j{\in}K.\ i\neq j\ -->\ transition\text{-}idx\ s\ T\ \{i,j\} = k$
by *auto*
have $\forall m.\ (s\ (enumerate\ K\ (Suc\ m)),\ s(enumerate\ K\ m)) \in T\ k$
proof
fix $m{::}nat$
let $?j = enumerate\ K\ (Suc\ m)$
let $?i = enumerate\ K\ m$
have *jK*: $?j \in K$ **by** (*simp add*: *enumerate-in-set infK*)
have *iK*: $?i \in K$ **by** (*simp add*: *enumerate-in-set infK*)
have *ij*: $?i < ?j$ **by** (*simp add*: *enumerate-step infK*)
have *ijk*: $transition\text{-}idx\ s\ T\ \{?i,?j\} = k$ **using** *iK jK ij*
by (*simp add*: *allk*)
obtain k' **where** $(s\ ?j,\ s\ ?i) \in T\ k'\ k'<n$
using *s-in-T* [*OF ij*] **by** *blast*
then show $(s\ ?j,\ s\ ?i) \in T\ k$
by (*simp add*: *ijk* [*symmetric*] *transition-idx-in ij*)
qed
then have $\sim\ wf(T\ k)$ **by** (*force simp add*: *wf-iff-no-infinite-down-chain*)
then show *False* **using** *wfT less* **by** *blast*

qed

end

82 Generic reflection and reification

theory *Reflection*
imports *Main*
begin

ML-file ~~/*src*/*HOL*/*Tools*/*reflection.ML*

method-setup *reify* = ‹
```
  Attrib.thms --
    Scan.option (Scan.lift (Args.$$$ () |-- Args.term --| Scan.lift (Args.$$$ )))
>>
    (fn (user-eqs, to) => fn ctxt => SIMPLE-METHOD' (Reflection.default-reify-tac
ctxt user-eqs to))
```
› *partial automatic reification*

method-setup *reflection* = ‹
```
let
  fun keyword k = Scan.lift (Args.$$$ k -- Args.colon) >> K ();
  val onlyN = only;
  val rulesN = rules;
  val any-keyword = keyword onlyN || keyword rulesN;
  val thms = Scan.repeats (Scan.unless any-keyword Attrib.multi-thm);
  val terms = thms >> map (Thm.term-of o Drule.dest-term);
in
  thms -- Scan.optional (keyword rulesN |-- thms) [] --
    Scan.option (keyword onlyN |-- Args.term) >>
  (fn ((user-eqs, user-thms), to) => fn ctxt =>
      SIMPLE-METHOD' (Reflection.default-reflection-tac ctxt user-thms user-eqs
to))
end
```
› *partial automatic reflection*

end

83 Assigning lengths to types by typeclasses

theory *Type-Length*
imports ~~/*src*/*HOL*/*Library*/*Numeral-Type*
begin

The aim of this is to allow any type as index type, but to provide a default instantiation for numeral types. This independence requires some duplication with the definitions in *Numeral-Type*.

class *len0* =
fixes *len-of* :: *'a itself* ⇒ *nat*

Some theorems are only true on words with length greater 0.

class *len* = *len0* +
assumes *len-gt-0* [*iff*]: *0* < *len-of TYPE* (*'a*)

instantiation *num0* **and** *num1* :: *len0*
begin

definition
len-num0: *len-of* (*x*::*num0 itself*) = *0*

definition
len-num1: *len-of* (*x*::*num1 itself*) = *1*

instance ..

end

instantiation *bit0* **and** *bit1* :: (*len0*) *len0*
begin

definition
len-bit0: *len-of* (*x*::*'a*::*len0 bit0 itself*) = *2* ∗ *len-of TYPE* (*'a*)

definition
len-bit1: *len-of* (*x*::*'a*::*len0 bit1 itself*) = *2* ∗ *len-of TYPE* (*'a*) + *1*

instance ..

end

lemmas *len-of-numeral-defs* [*simp*] = *len-num0 len-num1 len-bit0 len-bit1*

instance *num1* :: *len* **proof qed** *simp*
instance *bit0* :: (*len*) *len* **proof qed** *simp*
instance *bit1* :: (*len0*) *len* **proof qed** *simp*

end

84 Saturated arithmetic

theory *Saturated*
imports *Numeral-Type* ~~/*src*/*HOL*/*Word*/*Type-Length*
begin

84.1 The type of saturated naturals

typedef (**overloaded**) (*'a::len*) *sat* = {.. *len-of TYPE('a)*}
morphisms *nat-of Abs-sat*
by *auto*

lemma *sat-eqI*:
nat-of m = *nat-of n* ⟹ *m* = *n*
by (*simp add*: *nat-of-inject*)

lemma *sat-eq-iff*:
m = *n* ⟷ *nat-of m* = *nat-of n*
by (*simp add*: *nat-of-inject*)

lemma *Abs-sat-nat-of* [*code abstype*]:
Abs-sat (*nat-of n*) = *n*
by (*fact nat-of-inverse*)

definition *Abs-sat'* :: *nat* ⇒ *'a::len sat* **where**
Abs-sat' n = *Abs-sat* (*min* (*len-of TYPE('a)*) *n*)

lemma *nat-of-Abs-sat'* [*simp*]:
nat-of (*Abs-sat' n* :: (*'a::len*) *sat*) = *min* (*len-of TYPE('a)*) *n*
unfolding *Abs-sat'-def* **by** (*rule Abs-sat-inverse*) *simp*

lemma *nat-of-le-len-of* [*simp*]:
nat-of (*n* :: (*'a::len*) *sat*) ≤ *len-of TYPE('a)*
using *nat-of* [**where** *x* = *n*] **by** *simp*

lemma *min-len-of-nat-of* [*simp*]:
min (*len-of TYPE('a)*) (*nat-of* (*n*::(*'a::len*) *sat*)) = *nat-of n*
by (*rule min.absorb2* [*OF nat-of-le-len-of*])

lemma *min-nat-of-len-of* [*simp*]:
min (*nat-of* (*n*::(*'a::len*) *sat*)) (*len-of TYPE('a)*) = *nat-of n*
by (*subst min.commute*) *simp*

lemma *Abs-sat'-nat-of* [*simp*]:
Abs-sat' (*nat-of n*) = *n*
by (*simp add*: *Abs-sat'-def nat-of-inverse*)

instantiation *sat* :: (*len*) *linorder*
begin

definition
less-eq-sat-def: *x* ≤ *y* ⟷ *nat-of x* ≤ *nat-of y*

definition
less-sat-def: *x* < *y* ⟷ *nat-of x* < *nat-of y*

instance
 by *standard*
 (*auto simp add*: *less-eq-sat-def less-sat-def not-le sat-eq-iff min.coboundedI1 mult.commute*)

end

instantiation *sat* :: (*len*) {*minus*, *comm-semiring-1*}
begin

definition
 0 = *Abs-sat′ 0*

definition
 1 = *Abs-sat′ 1*

lemma *nat-of-zero-sat* [*simp*, *code abstract*]:
 nat-of 0 = *0*
 by (*simp add*: *zero-sat-def*)

lemma *nat-of-one-sat* [*simp*, *code abstract*]:
 nat-of 1 = *min 1* (*len-of TYPE*(*′a*))
 by (*simp add*: *one-sat-def*)

definition
 $x + y$ = *Abs-sat′* (*nat-of x* + *nat-of y*)

lemma *nat-of-plus-sat* [*simp*, *code abstract*]:
 nat-of ($x + y$) = *min* (*nat-of x* + *nat-of y*) (*len-of TYPE*(*′a*))
 by (*simp add*: *plus-sat-def*)

definition
 $x - y$ = *Abs-sat′* (*nat-of x* − *nat-of y*)

lemma *nat-of-minus-sat* [*simp*, *code abstract*]:
 nat-of ($x - y$) = *nat-of x* − *nat-of y*
proof −
 from *nat-of-le-len-of* [*of x*] **have** *nat-of x* − *nat-of y* ≤ *len-of TYPE*(*′a*) **by** *arith*
 then show *?thesis* **by** (*simp add*: *minus-sat-def*)
qed

definition
 $x * y$ = *Abs-sat′* (*nat-of x* ∗ *nat-of y*)

lemma *nat-of-times-sat* [*simp*, *code abstract*]:
 nat-of ($x * y$) = *min* (*nat-of x* ∗ *nat-of y*) (*len-of TYPE*(*′a*))
 by (*simp add*: *times-sat-def*)

```
instance
proof
  fix a b c :: 'a::len sat
  show a * b * c = a * (b * c)
  proof(cases a = 0)
    case True thus ?thesis by (simp add: sat-eq-iff)
  next
    case False show ?thesis
    proof(cases c = 0)
      case True thus ?thesis by (simp add: sat-eq-iff)
    next
      case False with ‹a ≠ 0› show ?thesis
          by (simp add: sat-eq-iff nat-mult-min-left nat-mult-min-right mult.assoc
min.assoc min.absorb2)
    qed
  qed
  show 1 * a = a
    apply (simp add: sat-eq-iff)
   apply (metis One-nat-def len-gt-0 less-Suc0 less-zeroE linorder-not-less min.absorb-iff1
min-nat-of-len-of nat-mult-1-right mult.commute)
    done
  show (a + b) * c = a * c + b * c
  proof(cases c = 0)
    case True thus ?thesis by (simp add: sat-eq-iff)
  next
    case False thus ?thesis
     by (simp add: sat-eq-iff nat-mult-min-left add-mult-distrib min-add-distrib-left
min-add-distrib-right min.assoc min.absorb2)
  qed
qed (simp-all add: sat-eq-iff mult.commute)

end

instantiation sat :: (len) ordered-comm-semiring
begin

instance
  by standard
     (auto simp add: less-eq-sat-def less-sat-def not-le sat-eq-iff min.coboundedI1
mult.commute)

end

lemma Abs-sat'-eq-of-nat: Abs-sat' n = of-nat n
  by (rule sat-eqI, induct n, simp-all)

abbreviation Sat :: nat ⇒ 'a::len sat where
  Sat ≡ of-nat
```

lemma *nat-of-Sat* [*simp*]:
nat-of (*Sat n* :: (′*a*::*len*) *sat*) = *min* (*len-of TYPE*(′*a*)) *n*
by (*rule nat-of-Abs-sat′* [*unfolded Abs-sat′-eq-of-nat*])

lemma [*code-abbrev*]:
of-nat (*numeral k*) = (*numeral k* :: ′*a*::*len sat*)
by *simp*

context
begin

qualified definition *sat-of-nat* :: *nat* ⇒ (′*a*::*len*) *sat*
where [*code-abbrev*]: *sat-of-nat* = *of-nat*

lemma [*code abstract*]:
nat-of (*sat-of-nat n* :: (′*a*::*len*) *sat*) = *min* (*len-of TYPE*(′*a*)) *n*
by (*simp add*: *sat-of-nat-def*)

end

instance *sat* :: (*len*) *finite*
proof
show *finite* (*UNIV*::′*a sat set*)
unfolding *type-definition.univ* [*OF type-definition-sat*]
using *finite* **by** *simp*
qed

instantiation *sat* :: (*len*) *equal*
begin

definition *HOL.equal A B* ⟷ *nat-of A* = *nat-of B*

instance
by *standard* (*simp add*: *equal-sat-def nat-of-inject*)

end

instantiation *sat* :: (*len*) {*bounded-lattice*, *distrib-lattice*}
begin

definition (*inf* :: ′*a sat* ⇒ ′*a sat* ⇒ ′*a sat*) = *min*
definition (*sup* :: ′*a sat* ⇒ ′*a sat* ⇒ ′*a sat*) = *max*
definition *bot* = (*0* :: ′*a sat*)
definition *top* = *Sat* (*len-of TYPE*(′*a*))

instance
by *standard*
(*simp-all add*: *inf-sat-def sup-sat-def bot-sat-def top-sat-def max-min-distrib2*,
simp-all add: *less-eq-sat-def*)

end

instantiation *sat* :: (*len*) {*Inf*, *Sup*}
begin

definition *Inf* = (*semilattice-neutr-set.F min top* :: *'a sat set* ⇒ *'a sat*)
definition *Sup* = (*semilattice-neutr-set.F max bot* :: *'a sat set* ⇒ *'a sat*)

instance ..

end

interpretation *Inf-sat*: *semilattice-neutr-set min top* :: *'a::len sat*
rewrites
  semilattice-neutr-set.F min (*top* :: *'a sat*) = *Inf*
proof −
  show *semilattice-neutr-set min* (*top* :: *'a sat*)
    by *standard* (*simp add*: *min-def*)
  show *semilattice-neutr-set.F min* (*top* :: *'a sat*) = *Inf*
    by (*simp add*: *Inf-sat-def*)
qed

interpretation *Sup-sat*: *semilattice-neutr-set max bot* :: *'a::len sat*
rewrites
  semilattice-neutr-set.F max (*bot* :: *'a sat*) = *Sup*
proof −
  show *semilattice-neutr-set max* (*bot* :: *'a sat*)
    by *standard* (*simp add*: *max-def bot.extremum-unique*)
  show *semilattice-neutr-set.F max* (*bot* :: *'a sat*) = *Sup*
    by (*simp add*: *Sup-sat-def*)
qed

instance *sat* :: (*len*) *complete-lattice*
proof
  fix *x* :: *'a sat*
  fix *A* :: *'a sat set*
  note *finite*
  moreover assume $x \in A$
  ultimately show *Inf A* ≤ *x*
    by (*induct A*) (*auto intro*: *min.coboundedI2*)
next
  fix *z* :: *'a sat*
  fix *A* :: *'a sat set*
  note *finite*
  moreover assume *z*: $\bigwedge x.\ x \in A \Longrightarrow z \le x$
  ultimately show *z* ≤ *Inf A* **by** (*induct A*) *simp-all*
next
  fix *x* :: *'a sat*

```
  fix A :: 'a sat set
  note finite
  moreover assume x ∈ A
  ultimately show x ≤ Sup A
    by (induct A) (auto intro: max.coboundedI2)
next
  fix z :: 'a sat
  fix A :: 'a sat set
  note finite
  moreover assume z: ⋀x. x ∈ A ⟹ x ≤ z
  ultimately show Sup A ≤ z by (induct A) auto
next
  show Inf {} = (top::'a sat)
    by (auto simp: top-sat-def)
  show Sup {} = (bot::'a sat)
    by (auto simp: bot-sat-def)
qed

end
```

85 Combinator syntax for generic, open state monads (single-threaded monads)

```
theory State-Monad
imports Main Monad-Syntax
begin
```

85.1 Motivation

The logic HOL has no notion of constructor classes, so it is not possible to model monads the Haskell way in full genericity in Isabelle/HOL.

However, this theory provides substantial support for a very common class of monads: *state monads* (or *single-threaded monads*, since a state is transformed single-threadedly).

To enter from the Haskell world, http://www.engr.mun.ca/~theo/Misc/haskell_and_monads.htm makes a good motivating start. Here we just sketch briefly how those monads enter the game of Isabelle/HOL.

85.2 State transformations and combinators

We classify functions operating on states into two categories:

transformations with type signature $\sigma \Rightarrow \sigma'$, transforming a state.

"yielding" transformations with type signature $\sigma \Rightarrow \alpha \times \sigma'$, "yielding" a side result while transforming a state.

queries with type signature $\sigma \Rightarrow \alpha$, computing a result dependent on a state.

By convention we write σ for types representing states and α, β, γ, ... for types representing side results. Type changes due to transformations are not excluded in our scenario.

We aim to assert that values of any state type σ are used in a single-threaded way: after application of a transformation on a value of type σ, the former value should not be used again. To achieve this, we use a set of monad combinators:

notation *fcomp* (**infixl** $\circ>$ *60*)
notation *scomp* (**infixl** $\circ\rightarrow$ *60*)

Given two transformations f and g, they may be directly composed using the *op* $\circ>$ combinator, forming a forward composition: $(f \circ> g)\ s = f\ (g\ s)$.

After any yielding transformation, we bind the side result immediately using a lambda abstraction. This is the purpose of the *op* $\circ\rightarrow$ combinator: $(f \circ\rightarrow (\lambda x.\ g))\ s = (let\ (x,\ s') = f\ s\ in\ g\ s')$.

For queries, the existing *Let* is appropriate.

Naturally, a computation may yield a side result by pairing it to the state from the left; we introduce the suggestive abbreviation *return* for this purpose.

The most crucial distinction to Haskell is that we do not need to introduce distinguished type constructors for different kinds of state. This has two consequences:

- The monad model does not state anything about the kind of state; the model for the state is completely orthogonal and may be specified completely independently.
- There is no distinguished type constructor encapsulating away the state transformation, i.e. transformations may be applied directly without using any lifting or providing and dropping units ("open monad").
- The type of states may change due to a transformation.

85.3 Monad laws

The common monadic laws hold and may also be used as normalization rules for monadic expressions:

lemmas *monad-simp* = *Pair-scomp scomp-Pair id-fcomp fcomp-id scomp-scomp scomp-fcomp fcomp-scomp fcomp-assoc*

Evaluation of monadic expressions by force:

lemmas *monad-collapse* = *monad-simp fcomp-apply scomp-apply split-beta*

85.4 Do-syntax

nonterminal *sdo-binds* **and** *sdo-bind*

syntax
  -sdo-block :: *sdo-binds* ⇒ *'a* (*exec* {//(*2* -)//} [*12*] *62*)
  -sdo-bind :: [*pttrn*, *'a*] ⇒ *sdo-bind* ((- ←/ -) *13*)
  -sdo-let :: [*pttrn*, *'a*] ⇒ *sdo-bind* ((*2let* - =/ -) [*1000*, *13*] *13*)
  -sdo-then :: *'a* ⇒ *sdo-bind* (- [*14*] *13*)
  -sdo-final :: *'a* ⇒ *sdo-binds* (-)
  -sdo-cons :: [*sdo-bind*, *sdo-binds*] ⇒ *sdo-binds* (-;//- [*13*, *12*] *12*)

syntax (*ASCII*)
  -sdo-bind :: [*pttrn*, *'a*] ⇒ *sdo-bind* ((- <−/ -) *13*)

translations
  -sdo-block (*-sdo-cons* (*-sdo-bind p t*) (*-sdo-final e*))
    == *CONST scomp t* (λ*p*. *e*)
  -sdo-block (*-sdo-cons* (*-sdo-then t*) (*-sdo-final e*))
    => *CONST fcomp t e*
  -sdo-final (*-sdo-block* (*-sdo-cons* (*-sdo-then t*) (*-sdo-final e*)))
    <= *-sdo-final* (*CONST fcomp t e*)
  -sdo-block (*-sdo-cons* (*-sdo-then t*) *e*)
    <= *CONST fcomp t* (*-sdo-block e*)
  -sdo-block (*-sdo-cons* (*-sdo-let p t*) *bs*)
    == *let p* = *t in -sdo-block bs*
  -sdo-block (*-sdo-cons b* (*-sdo-cons c cs*))
    == *-sdo-block* (*-sdo-cons b* (*-sdo-final* (*-sdo-block* (*-sdo-cons c cs*))))
  -sdo-cons (*-sdo-let p t*) (*-sdo-final s*)
    == *-sdo-final* (*let p* = *t in s*)
  -sdo-block (*-sdo-final e*) => *e*

For an example, see `~~/src/HOL/Proofs/Extraction/Higman.thy`.

end

86 A decision procedure for universal multivariate real arithmetic with addition, multiplication and ordering using semidefinite programming

theory *Sum-of-Squares*
imports *Complex-Main*
begin

ML-file *positivstellensatz.ML*
ML-file *Sum-of-Squares/sum-of-squares.ML*
ML-file *Sum-of-Squares/positivstellensatz-tools.ML*
ML-file *Sum-of-Squares/sos-wrapper.ML*

end

87 A table-based implementation of the reflexive transitive closure

```
theory Transitive-Closure-Table
imports Main
begin

inductive rtrancl-path :: ('a ⇒ 'a ⇒ bool) ⇒ 'a ⇒ 'a list ⇒ 'a ⇒ bool
  for r :: 'a ⇒ 'a ⇒ bool
where
  base: rtrancl-path r x [] x
| step: r x y ⟹ rtrancl-path r y ys z ⟹ rtrancl-path r x (y # ys) z

lemma rtranclp-eq-rtrancl-path: r** x y ⟷ (∃ xs. rtrancl-path r x xs y)
proof
  show ∃ xs. rtrancl-path r x xs y if r** x y
    using that
  proof (induct rule: converse-rtranclp-induct)
    case base
    have rtrancl-path r y [] y by (rule rtrancl-path.base)
    then show ?case ..
  next
    case (step x z)
    from ‹∃ xs. rtrancl-path r z xs y›
    obtain xs where rtrancl-path r z xs y ..
    with ‹r x z› have rtrancl-path r x (z # xs) y
      by (rule rtrancl-path.step)
    then show ?case ..
  qed
  show r** x y if ∃ xs. rtrancl-path r x xs y
  proof −
    from that obtain xs where rtrancl-path r x xs y ..
    then show ?thesis
    proof induct
      case (base x)
      show ?case
        by (rule rtranclp.rtrancl-refl)
    next
      case (step x y ys z)
      from ‹r x y› ‹r** y z› show ?case
        by (rule converse-rtranclp-into-rtranclp)
    qed
  qed
qed
```

lemma *rtrancl-path-trans*:
  assumes *xy*: *rtrancl-path r x xs y*
    and *yz*: *rtrancl-path r y ys z*
  shows *rtrancl-path r x* (*xs* @ *ys*) *z* **using** *xy yz*
proof (*induct arbitrary*: *z*)
  case (*base x*)
  then show *?case* **by** *simp*
next
  case (*step x y xs*)
  then have *rtrancl-path r y* (*xs* @ *ys*) *z*
    by *simp*
  with ‹*r x y*› **have** *rtrancl-path r x* (*y* # (*xs* @ *ys*)) *z*
    by (*rule rtrancl-path.step*)
  then show *?case* **by** *simp*
qed

lemma *rtrancl-path-appendE*:
  assumes *xz*: *rtrancl-path r x* (*xs* @ *y* # *ys*) *z*
  obtains *rtrancl-path r x* (*xs* @ [*y*]) *y* **and** *rtrancl-path r y ys z*
  using *xz*
proof (*induct xs arbitrary*: *x*)
  case *Nil*
  then have *rtrancl-path r x* (*y* # *ys*) *z* **by** *simp*
  then obtain *xy*: *r x y* **and** *yz*: *rtrancl-path r y ys z*
    by *cases auto*
  from *xy* **have** *rtrancl-path r x* [*y*] *y*
    by (*rule rtrancl-path.step* [*OF* - *rtrancl-path.base*])
  then have *rtrancl-path r x* ([] @ [*y*]) *y* **by** *simp*
  then show *thesis* **using** *yz* **by** (*rule Nil*)
next
  case (*Cons a as*)
  then have *rtrancl-path r x* (*a* # (*as* @ *y* # *ys*)) *z* **by** *simp*
  then obtain *xa*: *r x a* **and** *az*: *rtrancl-path r a* (*as* @ *y* # *ys*) *z*
    by *cases auto*
  show *thesis*
  proof (*rule Cons*(*1*) [*OF* - *az*])
    assume *rtrancl-path r y ys z*
    assume *rtrancl-path r a* (*as* @ [*y*]) *y*
    with *xa* **have** *rtrancl-path r x* (*a* # (*as* @ [*y*])) *y*
      by (*rule rtrancl-path.step*)
    then have *rtrancl-path r x* ((*a* # *as*) @ [*y*]) *y*
      by *simp*
    then show *thesis* **using** ‹*rtrancl-path r y ys z*›
      by (*rule Cons*(*2*))
  qed
qed

lemma *rtrancl-path-distinct*:
  assumes *xy*: *rtrancl-path r x xs y*

obtains *xs'* **where** *rtrancl-path r x xs' y* **and** *distinct* (*x* # *xs'*) **and** *set xs'* ⊆ *set xs*
using *xy*
proof (*induct xs rule*: *measure-induct-rule* [*of length*])
case (*less xs*)
show *?case*
proof (*cases distinct* (*x* # *xs*))
case *True*
with ‹*rtrancl-path r x xs y*› **show** *?thesis* **by** (*rule less*) *simp*
next
case *False*
then have ∃ *as bs cs a*. *x* # *xs* = *as* @ [*a*] @ *bs* @ [*a*] @ *cs*
by (*rule not-distinct-decomp*)
then obtain *as bs cs a* **where** *xxs*: *x* # *xs* = *as* @ [*a*] @ *bs* @ [*a*] @ *cs*
by *iprover*
show *?thesis*
proof (*cases as*)
case *Nil*
with *xxs* **have** *x*: *x* = *a* **and** *xs*: *xs* = *bs* @ *a* # *cs*
by *auto*
from *x xs* ‹*rtrancl-path r x xs y*› **have** *cs*: *rtrancl-path r x cs y set cs* ⊆ *set xs*
by (*auto elim*: *rtrancl-path-appendE*)
from *xs* **have** *length cs* < *length xs* **by** *simp*
then show *?thesis*
by (*rule less*(*1*))(*blast intro*: *cs less*(*2*) *order-trans del*: *subsetI*)+
next
case (*Cons d ds*)
with *xxs* **have** *xs*: *xs* = *ds* @ *a* # (*bs* @ [*a*] @ *cs*)
by *auto*
with ‹*rtrancl-path r x xs y*› **obtain** *xa*: *rtrancl-path r x* (*ds* @ [*a*]) *a*
and *ay*: *rtrancl-path r a* (*bs* @ *a* # *cs*) *y*
by (*auto elim*: *rtrancl-path-appendE*)
from *ay* **have** *rtrancl-path r a cs y* **by** (*auto elim*: *rtrancl-path-appendE*)
with *xa* **have** *xy*: *rtrancl-path r x* ((*ds* @ [*a*]) @ *cs*) *y*
by (*rule rtrancl-path-trans*)
from *xs* **have** *set*: *set* ((*ds* @ [*a*]) @ *cs*) ⊆ *set xs* **by** *auto*
from *xs* **have** *length* ((*ds* @ [*a*]) @ *cs*) < *length xs* **by** *simp*
then show *?thesis*
by (*rule less*(*1*))(*blast intro*: *xy less*(*2*) *set*[*THEN subsetD*])+
qed
qed
qed

inductive *rtrancl-tab* :: ('*a* ⇒ '*a* ⇒ *bool*) ⇒ '*a list* ⇒ '*a* ⇒ '*a* ⇒ *bool*
for *r* :: '*a* ⇒ '*a* ⇒ *bool*
where
base: *rtrancl-tab r xs x x*
| *step*: *x* ∉ *set xs* ⟹ *r x y* ⟹ *rtrancl-tab r* (*x* # *xs*) *y z* ⟹ *rtrancl-tab r xs x z*

lemma *rtrancl-path-imp-rtrancl-tab*:
  assumes *path*: *rtrancl-path r x xs y*
    and *x*: *distinct* (*x* # *xs*)
    and *ys*: ({*x*} ∪ *set xs*) ∩ *set ys* = {}
  shows *rtrancl-tab r ys x y*
  using *path x ys*
proof (*induct arbitrary*: *ys*)
  case *base*
  show *?case*
    by (*rule rtrancl-tab.base*)
next
  case (*step x y zs z*)
  then have *x* ∉ *set ys*
    by *auto*
  from *step* **have** *distinct* (*y* # *zs*)
    by *simp*
  moreover from *step* **have** ({*y*} ∪ *set zs*) ∩ *set* (*x* # *ys*) = {}
    by *auto*
  ultimately have *rtrancl-tab r* (*x* # *ys*) *y z*
    by (*rule step*)
  with ‹*x* ∉ *set ys*› ‹*r x y*› **show** *?case*
    by (*rule rtrancl-tab.step*)
qed

lemma *rtrancl-tab-imp-rtrancl-path*:
  assumes *tab*: *rtrancl-tab r ys x y*
  obtains *xs* **where** *rtrancl-path r x xs y*
  using *tab*
proof *induct*
  case *base*
  from *rtrancl-path.base* **show** *?case*
    by (*rule base*)
next
  case *step*
  show *?case*
    by (*iprover intro*: *step rtrancl-path.step*)
qed

lemma *rtranclp-eq-rtrancl-tab-nil*: r^{**} *x y* ⟷ *rtrancl-tab r* [] *x y*
proof
  show *rtrancl-tab r* [] *x y* **if** r^{**} *x y*
  proof −
    from *that* **obtain** *xs* **where** *rtrancl-path r x xs y*
      by (*auto simp add*: *rtranclp-eq-rtrancl-path*)
    then obtain *xs'* **where** *xs'*: *rtrancl-path r x xs' y* **and** *distinct*: *distinct* (*x* #
xs')
      by (*rule rtrancl-path-distinct*)
    have ({*x*} ∪ *set xs'*) ∩ *set* [] = {}
      by *simp*

```
    with xs' distinct show ?thesis
      by (rule rtrancl-path-imp-rtrancl-tab)
  qed
  show r** x y if rtrancl-tab r [] x y
  proof −
    from that obtain xs where rtrancl-path r x xs y
      by (rule rtrancl-tab-imp-rtrancl-path)
    then show ?thesis
      by (auto simp add: rtranclp-eq-rtrancl-path)
  qed
qed

declare rtranclp-rtrancl-eq [code del]
declare rtranclp-eq-rtrancl-tab-nil [THEN iffD2, code-pred-intro]

code-pred rtranclp
  using rtranclp-eq-rtrancl-tab-nil [THEN iffD1] by fastforce

lemma rtrancl-path-Range: ⟦ rtrancl-path R x xs y; z ∈ set xs ⟧ ⟹ RangeP R z
by(induction rule: rtrancl-path.induct) auto

lemma rtrancl-path-Range-end: ⟦ rtrancl-path R x xs y; xs ≠ [] ⟧ ⟹ RangeP R y
by(induction rule: rtrancl-path.induct)(auto elim: rtrancl-path.cases)

lemma rtrancl-path-nth:
  ⟦ rtrancl-path R x xs y; i < length xs ⟧ ⟹ R ((x # xs) ! i) (xs ! i)
proof(induction arbitrary: i rule: rtrancl-path.induct)
  case step thus ?case by(cases i) simp-all
qed simp

lemma rtrancl-path-last: ⟦ rtrancl-path R x xs y; xs ≠ [] ⟧ ⟹ last xs = y
by(induction rule: rtrancl-path.induct)(auto elim: rtrancl-path.cases)

lemma rtrancl-path-mono:
  ⟦ rtrancl-path R x p y; ⋀x y. R x y ⟹ S x y ⟧ ⟹ rtrancl-path S x p y
by(induction rule: rtrancl-path.induct)(auto intro: rtrancl-path.intros)

end
```

88 Binary Tree

```
theory Tree
imports Main
begin

datatype 'a tree =
  is-Leaf: Leaf (⟨⟩) |
  Node (left: 'a tree) (val: 'a) (right: 'a tree) ((1⟨-,/ -,/ -⟩))
  where
```

left Leaf = *Leaf*
| *right Leaf* = *Leaf*
datatype-compat *tree*

Can be seen as counting the number of leaves rather than nodes:

definition *size1* :: *'a tree* ⇒ *nat* **where**
size1 t = *size t* + *1*

lemma *size1-simps*[*simp*]:
size1 ⟨⟩ = *1*
size1 ⟨*l*, *x*, *r*⟩ = *size1 l* + *size1 r*
by (*simp-all add*: *size1-def*)

lemma *size1-ge0*[*simp*]: *0* < *size1 t*
by (*simp add*: *size1-def*)

lemma *size-0-iff-Leaf*: *size t* = *0* ⟷ *t* = *Leaf*
by(*cases t*) *auto*

lemma *neq-Leaf-iff*: (*t* ≠ ⟨⟩) = (∃ *l a r*. *t* = ⟨*l*, *a*, *r*⟩)
by (*cases t*) *auto*

lemma *finite-set-tree*[*simp*]: *finite*(*set-tree t*)
by(*induction t*) *auto*

lemma *size-map-tree*[*simp*]: *size* (*map-tree f t*) = *size t*
by (*induction t*) *auto*

lemma *size1-map-tree*[*simp*]: *size1* (*map-tree f t*) = *size1 t*
by (*simp add*: *size1-def*)

88.1 The Height

class *height* = **fixes** *height* :: *'a* ⇒ *nat*

instantiation *tree* :: (*type*)*height*
begin

fun *height-tree* :: *'a tree* => *nat* **where**
height Leaf = *0* |
height (*Node t1 a t2*) = *max* (*height t1*) (*height t2*) + *1*

instance ..

end

lemma *height-map-tree*[*simp*]: *height* (*map-tree f t*) = *height t*
by (*induction t*) *auto*

lemma *size1-height*: *size t* + *1* ≤ *2* ^ *height* (*t*::*'a tree*)

proof(*induction t*)
  case (*Node l a r*)
  show *?case*
  proof (*cases height l* ≤ *height r*)
    case *True*
    have *size*(*Node l a r*) + *1* = (*size l* + *1*) + (*size r* + *1*) **by** *simp*
    also have *size l* + *1* ≤ *2* ^ *height l* **by**(*rule Node.IH*(*1*))
    also have *size r* + *1* ≤ *2* ^ *height r* **by**(*rule Node.IH*(*2*))
    also have (*2*::*nat*) ^ *height l* ≤ *2* ^ *height r* **using** *True* **by** *simp*
    finally show *?thesis* **using** *True* **by** (*auto simp*: *max-def mult-2*)
  next
    case *False*
    have *size*(*Node l a r*) + *1* = (*size l* + *1*) + (*size r* + *1*) **by** *simp*
    also have *size l* + *1* ≤ *2* ^ *height l* **by**(*rule Node.IH*(*1*))
    also have *size r* + *1* ≤ *2* ^ *height r* **by**(*rule Node.IH*(*2*))
    also have (*2*::*nat*) ^ *height r* ≤ *2* ^ *height l* **using** *False* **by** *simp*
    finally show *?thesis* **using** *False* **by** (*auto simp*: *max-def mult-2*)
  qed
qed *simp*

88.2 The set of subtrees

fun *subtrees* :: *'a tree* ⇒ *'a tree set* **where**
subtrees ⟨⟩ = {⟨⟩} |
subtrees (⟨*l*, *a*, *r*⟩) = *insert* ⟨*l*, *a*, *r*⟩ (*subtrees l* ∪ *subtrees r*)

lemma *set-treeE*: *a* ∈ *set-tree t* ⟹ ∃ *l r*. ⟨*l*, *a*, *r*⟩ ∈ *subtrees t*
by (*induction t*)(*auto*)

lemma *Node-notin-subtrees-if*[*simp*]: *a* ∉ *set-tree t* ⟹ *Node l a r* ∉ *subtrees t*
by (*induction t*) *auto*

lemma *in-set-tree-if*: ⟨*l*, *a*, *r*⟩ ∈ *subtrees t* ⟹ *a* ∈ *set-tree t*
by (*metis Node-notin-subtrees-if*)

88.3 List of entries

fun *preorder* :: *'a tree* ⇒ *'a list* **where**
preorder ⟨⟩ = [] |
preorder ⟨*l*, *x*, *r*⟩ = *x* # *preorder l* @ *preorder r*

fun *inorder* :: *'a tree* ⇒ *'a list* **where**
inorder ⟨⟩ = [] |
inorder ⟨*l*, *x*, *r*⟩ = *inorder l* @ [*x*] @ *inorder r*

lemma *set-inorder*[*simp*]: *set* (*inorder t*) = *set-tree t*
by (*induction t*) *auto*

lemma *set-preorder*[*simp*]: *set* (*preorder t*) = *set-tree t*
by (*induction t*) *auto*

lemma *length-preorder*[*simp*]: *length* (*preorder t*) = *size t*
by (*induction t*) *auto*

lemma *length-inorder*[*simp*]: *length* (*inorder t*) = *size t*
by (*induction t*) *auto*

lemma *preorder-map*: *preorder* (*map-tree f t*) = *map f* (*preorder t*)
by (*induction t*) *auto*

lemma *inorder-map*: *inorder* (*map-tree f t*) = *map f* (*inorder t*)
by (*induction t*) *auto*

88.4 Binary Search Tree predicate

fun (**in** *linorder*) *bst* :: *'a tree* ⇒ *bool* **where**
bst ⟨⟩ ⟷ *True* |
bst ⟨*l*, *a*, *r*⟩ ⟷ *bst l* ∧ *bst r* ∧ (∀ *x*∈*set-tree l*. *x* < *a*) ∧ (∀ *x*∈*set-tree r*. *a* < *x*)

In case there are duplicates:

fun (**in** *linorder*) *bst-eq* :: *'a tree* ⇒ *bool* **where**
bst-eq ⟨⟩ ⟷ *True* |
bst-eq ⟨*l*,*a*,*r*⟩ ⟷
bst-eq l ∧ *bst-eq r* ∧ (∀ *x*∈*set-tree l*. *x* ≤ *a*) ∧ (∀ *x*∈*set-tree r*. *a* ≤ *x*)

lemma (**in** *linorder*) *bst-eq-if-bst*: *bst t* ⟹ *bst-eq t*
by (*induction t*) (*auto*)

lemma (**in** *linorder*) *bst-eq-imp-sorted*: *bst-eq t* ⟹ *sorted* (*inorder t*)
apply (*induction t*)
apply(*simp*)
by (*fastforce simp*: *sorted-append sorted-Cons intro*: *less-imp-le less-trans*)

lemma (**in** *linorder*) *distinct-preorder-if-bst*: *bst t* ⟹ *distinct* (*preorder t*)
apply (*induction t*)
apply *simp*
apply(*fastforce elim*: *order.asym*)
done

lemma (**in** *linorder*) *distinct-inorder-if-bst*: *bst t* ⟹ *distinct* (*inorder t*)
apply (*induction t*)
apply *simp*
apply(*fastforce elim*: *order.asym*)
done

88.5 The heap predicate

fun *heap* :: *'a::linorder tree* ⇒ *bool* **where**
heap Leaf = *True* |
heap (*Node l m r*) =

$(heap\ l \wedge heap\ r \wedge (\forall x \in set\text{-}tree\ l \cup set\text{-}tree\ r.\ m \leq x))$

88.6 Function *mirror*

fun *mirror* :: *'a tree* ⇒ *'a tree* **where**
mirror ⟨⟩ = *Leaf* |
mirror ⟨*l*,*x*,*r*⟩ = ⟨*mirror r*, *x*, *mirror l*⟩

lemma *mirror-Leaf*[*simp*]: *mirror t* = ⟨⟩ ⟷ *t* = ⟨⟩
by (*induction t*) *simp-all*

lemma *size-mirror*[*simp*]: *size*(*mirror t*) = *size t*
by (*induction t*) *simp-all*

lemma *size1-mirror*[*simp*]: *size1* (*mirror t*) = *size1 t*
by (*simp add*: *size1-def*)

lemma *height-mirror*[*simp*]: *height*(*mirror t*) = *height t*
by (*induction t*) *simp-all*

lemma *inorder-mirror*: *inorder*(*mirror t*) = *rev*(*inorder t*)
by (*induction t*) *simp-all*

lemma *map-mirror*: *map-tree f* (*mirror t*) = *mirror* (*map-tree f t*)
by (*induction t*) *simp-all*

lemma *mirror-mirror*[*simp*]: *mirror*(*mirror t*) = *t*
by (*induction t*) *simp-all*

end

89 Multiset of Elements of Binary Tree

theory *Tree-Multiset*
imports *Multiset Tree*
begin

Kept separate from theory *Tree* to avoid importing all of theory *Multiset* into *Tree*. Should be merged if *Multiset* ever becomes part of *Main*.

fun *mset-tree* :: *'a tree* ⇒ *'a multiset* **where**
mset-tree Leaf = {#} |
mset-tree (*Node l a r*) = {#*a*#} + *mset-tree l* + *mset-tree r*

lemma *set-mset-tree*[*simp*]: *set-mset* (*mset-tree t*) = *set-tree t*
by(*induction t*) *auto*

lemma *size-mset-tree*[*simp*]: *size*(*mset-tree t*) = *size t*
by(*induction t*) *auto*

```
lemma mset-map-tree: mset-tree (map-tree f t) = image-mset f (mset-tree t)
by (induction t) auto

lemma mset-iff-set-tree: x ∈# mset-tree t ⟷ x ∈ set-tree t
by(induction t arbitrary: x) auto

lemma mset-preorder[simp]: mset (preorder t) = mset-tree t
by (induction t) (auto simp: ac-simps)

lemma mset-inorder[simp]: mset (inorder t) = mset-tree t
by (induction t) (auto simp: ac-simps)

lemma map-mirror: mset-tree (mirror t) = mset-tree t
by (induction t) (simp-all add: ac-simps)

end
```

90 A general "while" combinator

```
theory While-Combinator
imports Main
begin
```

90.1 Partial version

```
definition while-option :: ('a ⇒ bool) ⇒ ('a ⇒ 'a) ⇒ 'a ⇒ 'a option where
while-option b c s = (if (∃k. ~ b ((c ^^ k) s))
  then Some ((c ^^ (LEAST k. ~ b ((c ^^ k) s))) s)
  else None)

theorem while-option-unfold[code]:
while-option b c s = (if b s then while-option b c (c s) else Some s)
proof cases
 assume b s
 show ?thesis
 proof (cases ∃k. ~ b ((c ^^ k) s))
  case True
  then obtain k where 1: ~ b ((c ^^ k) s) ..
  with ‹b s› obtain l where k = Suc l by (cases k) auto
  with 1 have ~ b ((c ^^ l) (c s)) by (auto simp: funpow-swap1)
  then have 2: ∃l. ~ b ((c ^^ l) (c s)) ..
  from 1
  have (LEAST k. ~ b ((c ^^ k) s)) = Suc (LEAST l. ~ b ((c ^^ Suc l) s))
   by (rule Least-Suc) (simp add: ‹b s›)
  also have ... = Suc (LEAST l. ~ b ((c ^^ l) (c s)))
   by (simp add: funpow-swap1)
  finally
  show ?thesis
   using True 2 ‹b s› by (simp add: funpow-swap1 while-option-def)
```

```
  next
    case False
    then have ~ (∃ l. ~ b ((c ^^ Suc l) s)) by blast
    then have ~ (∃ l. ~ b ((c ^^ l) (c s)))
      by (simp add: funpow-swap1)
    with False ‹b s› show ?thesis by (simp add: while-option-def)
  qed
next
  assume [simp]: ~ b s
  have least: (LEAST k. ~ b ((c ^^ k) s)) = 0
    by (rule Least-equality) auto
  moreover
  have ∃ k. ~ b ((c ^^ k) s) by (rule exI[of - 0::nat]) auto
  ultimately show ?thesis unfolding while-option-def by auto
qed

lemma while-option-stop2:
 while-option b c s = Some t ⟹ EX k. t = (c^^k) s ∧ ¬ b t
apply(simp add: while-option-def split: if-splits)
by (metis (lifting) LeastI-ex)

lemma while-option-stop: while-option b c s = Some t ⟹ ~ b t
by(metis while-option-stop2)

theorem while-option-rule:
assumes step: !!s. P s ==> b s ==> P (c s)
and result: while-option b c s = Some t
and init: P s
shows P t
proof −
  def k == LEAST k. ~ b ((c ^^ k) s)
  from assms have t: t = (c ^^ k) s
    by (simp add: while-option-def k-def split: if-splits)
  have 1: ALL i<k. b ((c ^^ i) s)
    by (auto simp: k-def dest: not-less-Least)

  { fix i assume i <= k then have P ((c ^^ i) s)
      by (induct i) (auto simp: init step 1) }
  thus P t by (auto simp: t)
qed

lemma funpow-commute:
  ⟦∀ k' < k. f (c ((c^^k') s)) = c' (f ((c^^k') s))⟧ ⟹ f ((c^^k) s) = (c'^^k) (f s)
by (induct k arbitrary: s) auto

lemma while-option-commute-invariant:
assumes Invariant: ⋀s. P s ⟹ b s ⟹ P (c s)
assumes TestCommute: ⋀s. P s ⟹ b s = b' (f s)
assumes BodyCommute: ⋀s. P s ⟹ b s ⟹ f (c s) = c' (f s)
```

assumes *Initial*: *P s*
shows *map-option f* (*while-option b c s*) = *while-option b′ c′* (*f s*)
unfolding *while-option-def*
proof (*rule trans*[*OF if-distrib if-cong*], *safe*, *unfold option.inject*)
  fix *k*
  assume ¬ *b* ((*c* ^^ *k*) *s*)
  with *Initial* **show** ∃ *k*. ¬ *b′* ((*c′* ^^ *k*) (*f s*))
  proof (*induction k arbitrary*: *s*)
    case *0* **thus** *?case* **by** (*auto simp*: *TestCommute intro*: *exI*[*of* - *0*])
  next
    case (*Suc k*) **thus** *?case*
    proof (*cases b s*)
      assume *b s*
      with *Suc.IH*[*of c s*] *Suc.prems* **show** *?thesis*
      by (*metis BodyCommute Invariant comp-apply funpow.simps*(*2*) *funpow-swap1*)
    next
      assume ¬ *b s*
      with *Suc* **show** *?thesis* **by** (*auto simp*: *TestCommute intro*: *exI* [*of* - *0*])
    qed
  qed
next
  fix *k*
  assume ¬ *b′* ((*c′* ^^ *k*) (*f s*))
  with *Initial* **show** ∃ *k*. ¬ *b* ((*c* ^^ *k*) *s*)
  proof (*induction k arbitrary*: *s*)
    case *0* **thus** *?case* **by** (*auto simp*: *TestCommute intro*: *exI*[*of* - *0*])
  next
    case (*Suc k*) **thus** *?case*
    proof (*cases b s*)
      assume *b s*
      with *Suc.IH*[*of c s*] *Suc.prems* **show** *?thesis*
      by (*metis BodyCommute Invariant comp-apply funpow.simps*(*2*) *funpow-swap1*)
    next
      assume ¬ *b s*
      with *Suc* **show** *?thesis* **by** (*auto simp*: *TestCommute intro*: *exI* [*of* - *0*])
    qed
  qed
next
  fix *k*
  assume *k*: ¬ *b′* ((*c′* ^^ *k*) (*f s*))
  have ∗: (*LEAST k*. ¬ *b′* ((*c′* ^^ *k*) (*f s*))) = (*LEAST k*. ¬ *b* ((*c* ^^ *k*) *s*))
        (**is** *?k′* = *?k*)
  proof (*cases ?k′*)
    case *0*
    have ¬ *b′* ((*c′* ^^ *0*) (*f s*))
      unfolding *0*[*symmetric*] **by** (*rule LeastI*[*of* - *k*]) (*rule k*)
    hence ¬ *b s* **by** (*auto simp*: *TestCommute Initial*)
    hence *?k* = *0* **by** (*intro Least-equality*) *auto*
    with *0* **show** *?thesis* **by** *auto*

```
next
  case (Suc k')
  have ¬ b' ((c' ^^ Suc k') (f s))
    unfolding Suc[symmetric] by (rule LeastI) (rule k)
  moreover
  { fix k assume k ≤ k'
    hence k < ?k' unfolding Suc by simp
    hence b' ((c' ^^ k) (f s)) by (rule iffD1[OF not-not, OF not-less-Least])
  }
  note b' = this
  { fix k assume k ≤ k'
    hence f ((c ^^ k) s) = (c' ^^ k) (f s)
    and b ((c ^^ k) s) = b' ((c' ^^ k) (f s))
    and P ((c ^^ k) s)
      by (induct k) (auto simp: b' assms)
    with ‹k ≤ k'›
    have b ((c ^^ k) s)
    and f ((c ^^ k) s) = (c' ^^ k) (f s)
    and P ((c ^^ k) s)
      by (auto simp: b')
  }
  note b = this(1) and body = this(2) and inv = this(3)
  hence k': f ((c ^^ k') s) = (c' ^^ k') (f s) by auto
  ultimately show ?thesis unfolding Suc using b
  proof (intro Least-equality[symmetric], goal-cases)
    case 1
    hence Test: ¬ b' (f ((c ^^ Suc k') s))
      by (auto simp: BodyCommute inv b)
    have P ((c ^^ Suc k') s) by (auto simp: Invariant inv b)
    with Test show ?case by (auto simp: TestCommute)
  next
    case 2
    thus ?case by (metis not-less-eq-eq)
  qed
qed
have f ((c ^^ ?k) s) = (c' ^^ ?k') (f s) unfolding *
proof (rule funpow-commute, clarify)
  fix k assume k < ?k
  hence TestTrue: b ((c ^^ k) s) by (auto dest: not-less-Least)
  from ‹k < ?k› have P ((c ^^ k) s)
  proof (induct k)
    case 0 thus ?case by (auto simp: assms)
  next
    case (Suc h)
    hence P ((c ^^ h) s) by auto
    with Suc show ?case
      by (auto, metis (lifting, no-types) Invariant Suc-lessD not-less-Least)
  qed
  with TestTrue show f (c ((c ^^ k) s)) = c' (f ((c ^^ k) s))
```

```
    by (metis BodyCommute)
  qed
  thus ∃ z. (c ^^ ?k) s = z ∧ f z = (c′ ^^ ?k′) (f s) by blast
qed

lemma while-option-commute:
  assumes ⋀s. b s = b′ (f s) ⋀s. ⟦b s⟧ ⟹ f (c s) = c′ (f s)
  shows map-option f (while-option b c s) = while-option b′ c′ (f s)
by(rule while-option-commute-invariant[where P = λ-. True])
  (auto simp add: assms)
```

90.2 Total version

```
definition while :: ('a ⇒ bool) ⇒ ('a ⇒ 'a) ⇒ 'a ⇒ 'a
where while b c s = the (while-option b c s)

lemma while-unfold [code]:
  while b c s = (if b s then while b c (c s) else s)
unfolding while-def by (subst while-option-unfold) simp

lemma def-while-unfold:
  assumes fdef: f == while test do
  shows f x = (if test x then f(do x) else x)
unfolding fdef by (fact while-unfold)
```

The proof rule for *while*, where P is the invariant.

```
theorem while-rule-lemma:
  assumes invariant: !!s. P s ==> b s ==> P (c s)
    and terminate: !!s. P s ==> ¬ b s ==> Q s
    and wf: wf {(t, s). P s ∧ b s ∧ t = c s}
  shows P s ⟹ Q (while b c s)
  using wf
  apply (induct s)
  apply simp
  apply (subst while-unfold)
  apply (simp add: invariant terminate)
  done

theorem while-rule:
  [| P s;
     !!s. [| P s; b s |] ==> P (c s);
     !!s. [| P s; ¬ b s |] ==> Q s;
     wf r;
     !!s. [| P s; b s |] ==> (c s, s) ∈ r |] ==>
   Q (while b c s)
  apply (rule while-rule-lemma)
     prefer 4 apply assumption
    apply blast
   apply blast
  apply (erule wf-subset)
```

apply *blast*
done

Proving termination:

theorem *wf-while-option-Some*:
assumes *wf* $\{(t, s). (P\ s \wedge b\ s) \wedge t = c\ s\}$
and $!!s.\ P\ s \Longrightarrow b\ s \Longrightarrow P(c\ s)$ **and** $P\ s$
shows *EX t. while-option b c s = Some t*
using *assms(1,3)*
proof (*induction s*)
case *less* **thus** *?case* **using** *assms(2)*
by (*subst while-option-unfold*) *simp*
qed

lemma *wf-rel-while-option-Some*:
assumes *wf*: *wf R*
assumes *smaller*: $\bigwedge s.\ P\ s \wedge b\ s \Longrightarrow (c\ s,\ s) \in R$
assumes *inv*: $\bigwedge s.\ P\ s \wedge b\ s \Longrightarrow P(c\ s)$
assumes *init*: *P s*
shows $\exists\, t.$ *while-option b c s = Some t*
proof −
from *smaller* **have** $\{(t,s).\ P\ s \wedge b\ s \wedge t = c\ s\} \subseteq R$ **by** *auto*
with *wf* **have** *wf* $\{(t,s).\ P\ s \wedge b\ s \wedge t = c\ s\}$ **by** (*auto simp*: *wf-subset*)
with *inv init* **show** *?thesis* **by** (*auto simp*: *wf-while-option-Some*)
qed

theorem *measure-while-option-Some*: **fixes** $f :: {'s} \Rightarrow nat$
shows $(!!s.\ P\ s \Longrightarrow b\ s \Longrightarrow P(c\ s) \wedge f(c\ s) < f\ s)$
$\Longrightarrow P\ s \Longrightarrow$ *EX t. while-option b c s = Some t*
by(*blast intro*: *wf-while-option-Some*[*OF wf-if-measure*, *of P b f*])

Kleene iteration starting from the empty set and assuming some finite bounding set:

lemma *while-option-finite-subset-Some*: **fixes** *C* :: *'a set*
assumes *mono f* **and** $!!X.\ X \subseteq C \Longrightarrow f\ X \subseteq C$ **and** *finite C*
shows $\exists\, P.$ *while-option* $(\lambda A.\ f\ A \neq A)\ f\ \{\} = Some\ P$
proof(*rule measure-while-option-Some*[**where**
f= %*A*::*'a set*. *card C* − *card A* **and** *P*= %*A*. $A \subseteq C \wedge A \subseteq f\ A$ **and** *s*= {}])
fix *A* **assume** *A*: $A \subseteq C \wedge A \subseteq f\ A\ f\ A \neq A$
show $(f\ A \subseteq C \wedge f\ A \subseteq f\ (f\ A)) \wedge card\ C - card\ (f\ A) < card\ C - card\ A$
(**is** *?L* ∧ *?R*)
proof
show *?L* **by**(*metis A(1) assms(2) monoD*[*OF* ‹*mono f*›])
show *?R* **by** (*metis A assms(2,3) card-seteq diff-less-mono2 equalityI linorder-le-less-linear rev-finite-subset*)
qed
qed *simp*

lemma *lfp-the-while-option*:

assumes *mono f* **and** !!X. $X \subseteq C \Longrightarrow f\, X \subseteq C$ **and** *finite C*
shows *lfp f* = *the*(*while-option* (λA. f A ≠ A) f {})
proof−
obtain P **where** *while-option* (λA. f A ≠ A) f {} = *Some* P
using *while-option-finite-subset-Some*[*OF assms*] **by** *blast*
with *while-option-stop2*[*OF this*] *lfp-Kleene-iter*[*OF assms*(*1*)]
show *?thesis* **by** *auto*
qed

lemma *lfp-while*:
assumes *mono f* **and** !!X. $X \subseteq C \Longrightarrow f\, X \subseteq C$ **and** *finite C*
shows *lfp f* = *while* (λA. f A ≠ A) f {}
unfolding *while-def* **using** *assms* **by** (*rule lfp-the-while-option*) *blast*

Computing the reflexive, transitive closure by iterating a successor function. Stops when an element is found that dos not satisfy the test.

More refined (and hence more efficient) versions can be found in ITP 2011 paper by Nipkow (the theories are in the AFP entry Flyspeck by Nipkow) and the AFP article Executable Transitive Closures by René Thiemann.

context
fixes *p* :: 'a ⇒ *bool*
and *f* :: 'a ⇒ 'a *list*
and *x* :: 'a
begin

qualified fun *rtrancl-while-test* :: 'a *list* × 'a *set* ⇒ *bool*
where *rtrancl-while-test* (*ws*,-) = (*ws* ≠ [] ∧ *p*(*hd ws*))

qualified fun *rtrancl-while-step* :: 'a *list* × 'a *set* ⇒ 'a *list* × 'a *set*
where *rtrancl-while-step* (*ws*, *Z*) =
(*let x* = *hd ws*; *new* = *remdups* (*filter* (λy. y ∉ Z) (f x))
in (*new* @ *tl ws*, *set new* ∪ *Z*))

definition *rtrancl-while* :: ('a *list* ∗ 'a *set*) *option*
where *rtrancl-while* = *while-option rtrancl-while-test rtrancl-while-step* ([x],{x})

qualified fun *rtrancl-while-invariant* :: 'a *list* × 'a *set* ⇒ *bool*
where *rtrancl-while-invariant* (*ws*, *Z*) =
(x ∈ Z ∧ *set ws* ⊆ Z ∧ *distinct ws* ∧ {(x,y). y ∈ *set*(f x)} `` (Z − *set ws*) ⊆ Z ∧
Z ⊆ {(x,y). y ∈ *set*(f x)}^∗ `` {x} ∧ (∀ z∈Z − *set ws*. p z))

qualified lemma *rtrancl-while-invariant*:
assumes *inv*: *rtrancl-while-invariant st* **and** *test*: *rtrancl-while-test st*
shows *rtrancl-while-invariant* (*rtrancl-while-step st*)
proof (*cases st*)
fix *ws Z* **assume** *st*: *st* = (*ws*, *Z*)
with *test* **obtain** *h t* **where** *ws* = *h* # *t p h* **by** (*cases ws*) *auto*
with *inv st* **show** *?thesis* **by** (*auto intro*: *rtrancl.rtrancl-into-rtrancl*)

qed

lemma *rtrancl-while-Some*: **assumes** *rtrancl-while* = *Some*(*ws*,*Z*)
shows *if ws* = []
then Z = {(*x*,*y*). *y* ∈ *set*(*f x*)} ^∗ `` {*x*} ∧ (∀ *z*∈*Z*. *p z*)
else ¬*p*(*hd ws*) ∧ *hd ws* ∈ {(*x*,*y*). *y* ∈ *set*(*f x*)} ^∗ `` {*x*}
proof −
have *rtrancl-while-invariant* ([*x*],{*x*}) **by** *simp*
with *rtrancl-while-invariant* **have** *I*: *rtrancl-while-invariant* (*ws*,*Z*)
by (*rule while-option-rule*[*OF - assms*[*unfolded rtrancl-while-def*]])
{ assume *ws* = []
hence *?thesis* **using** *I*
by (*auto simp del*:*Image-Collect-case-prod dest*: *Image-closed-trancl*)
} moreover
{ assume *ws* ≠ []
hence *?thesis* **using** *I while-option-stop*[*OF assms*[*unfolded rtrancl-while-def*]]
by (*simp add*: *subset-iff*)
}
ultimately show *?thesis* **by** *simp*
qed

lemma *rtrancl-while-finite-Some*:
assumes *finite* ({(*x*, *y*). *y* ∈ *set* (*f x*)}* `` {*x*}) (**is** *finite ?Cl*)
shows ∃ *y*. *rtrancl-while* = *Some y*
proof −
let *?R* = (λ(-, *Z*). *card* (*?Cl* − *Z*)) <∗*mlex*∗> (λ(*ws*, -). *length ws*) <∗*mlex*∗> {}
have *wf ?R* **by** (*blast intro*: *wf-mlex*)
then show *?thesis* **unfolding** *rtrancl-while-def*
proof (*rule wf-rel-while-option-Some*[*of ?R rtrancl-while-invariant*])
fix *st* **assume** ∗: *rtrancl-while-invariant st* ∧ *rtrancl-while-test st*
hence *I*: *rtrancl-while-invariant* (*rtrancl-while-step st*)
by (*blast intro*: *rtrancl-while-invariant*)
show (*rtrancl-while-step st*, *st*) ∈ *?R*
proof (*cases st*)
fix *ws Z* **let** *?ws* = *fst* (*rtrancl-while-step st*) **and** *?Z* = *snd* (*rtrancl-while-step st*)
assume *st*: *st* = (*ws*, *Z*)
with ∗ **obtain** *h t* **where** *ws*: *ws* = *h* # *t p h* **by** (*cases ws*) *auto*
{ assume *remdups* (*filter* (λ*y*. *y* ∉ *Z*) (*f h*)) ≠ []
then obtain *z* **where** *z* ∈ *set* (*remdups* (*filter* (λ*y*. *y* ∉ *Z*) (*f h*))) **by** *fastforce*
with *st ws I* **have** *Z* ⊂ *?Z Z* ⊆ *?Cl ?Z* ⊆ *?Cl* **by** *auto*
with *assms* **have** *card* (*?Cl* − *?Z*) < *card* (*?Cl* − *Z*) **by** (*blast intro*: *psubset-card-mono*)
with *st ws* **have** *?thesis* **unfolding** *mlex-prod-def* **by** *simp*
}
moreover
{ assume *remdups* (*filter* (λ*y*. *y* ∉ *Z*) (*f h*)) = []

```
      with st ws have ?Z = Z ?ws = t  by (auto simp: filter-empty-conv)
      with st ws have ?thesis unfolding mlex-prod-def by simp
    }
    ultimately show ?thesis by blast
  qed
 qed (simp-all add: rtrancl-while-invariant)
qed

end

end


theory Rewrite
imports Main
begin

consts rewrite-HOLE :: 'a::{}  (⌑)

lemma eta-expand:
 fixes f :: 'a::{} ⇒ 'b::{}
 shows f ≡ λx. f x .

lemma rewr-imp:
 assumes PROP A ≡ PROP B
 shows (PROP A ⟹ PROP C) ≡ (PROP B ⟹ PROP C)
 apply (rule Pure.equal-intr-rule)
 apply (drule equal-elim-rule2[OF assms]; assumption)
 apply (drule equal-elim-rule1[OF assms]; assumption)
 done

lemma imp-cong-eq:
 (PROP A ⟹ (PROP B ⟹ PROP C) ≡ (PROP B' ⟹ PROP C')) ≡
   ((PROP B ⟹ PROP A ⟹ PROP C) ≡ (PROP B' ⟹ PROP A ⟹ PROP
C'))
 apply (intro Pure.equal-intr-rule)
     apply (drule (1) cut-rl; drule Pure.equal-elim-rule1 Pure.equal-elim-rule2;
assumption)+
  apply (drule Pure.equal-elim-rule1 Pure.equal-elim-rule2; assumption)+
 done

ML-file cconv.ML
ML-file rewrite.ML

end
```

91 Lexicographic order on lists

theory *List-lexord*

imports *Main*
begin

instantiation *list* :: (*ord*) *ord*
begin

definition
list-less-def: $xs < ys \longleftrightarrow (xs, ys) \in lexord \{(u, v). u < v\}$

definition
list-le-def: $(xs :: - list) \leq ys \longleftrightarrow xs < ys \lor xs = ys$

instance ..

end

instance *list* :: (*order*) *order*
proof
  fix *xs* :: *'a list*
  show $xs \leq xs$ **by** (*simp add*: *list-le-def*)
next
  fix *xs ys zs* :: *'a list*
  assume $xs \leq ys$ **and** $ys \leq zs$
  then show $xs \leq zs$
    apply (*auto simp add*: *list-le-def list-less-def*)
    apply (*rule lexord-trans*)
    apply (*auto intro*: *transI*)
    done
next
  fix *xs ys* :: *'a list*
  assume $xs \leq ys$ **and** $ys \leq xs$
  then show $xs = ys$
    apply (*auto simp add*: *list-le-def list-less-def*)
    apply (*rule lexord-irreflexive* [*THEN notE*])
    defer
    apply (*rule lexord-trans*)
    apply (*auto intro*: *transI*)
    done
next
  fix *xs ys* :: *'a list*
  show $xs < ys \longleftrightarrow xs \leq ys \land \neg ys \leq xs$
    apply (*auto simp add*: *list-less-def list-le-def*)
    defer
    apply (*rule lexord-irreflexive* [*THEN notE*])
    apply *auto*
    apply (*rule lexord-irreflexive* [*THEN notE*])
    defer
    apply (*rule lexord-trans*)
    apply (*auto intro*: *transI*)

  done
qed

instance *list* :: (*linorder*) *linorder*
proof
  fix *xs ys* :: *'a list*
  have (*xs*, *ys*) ∈ *lexord* {(*u*, *v*). *u* < *v*} ∨ *xs* = *ys* ∨ (*ys*, *xs*) ∈ *lexord* {(*u*, *v*). *u* < *v*}
    by (*rule lexord-linear*) *auto*
  then show *xs* ≤ *ys* ∨ *ys* ≤ *xs*
    by (*auto simp add*: *list-le-def list-less-def*)
qed

instantiation *list* :: (*linorder*) *distrib-lattice*
begin

definition (*inf* :: *'a list* ⇒ -) = *min*

definition (*sup* :: *'a list* ⇒ -) = *max*

instance
  by *standard* (*auto simp add*: *inf-list-def sup-list-def max-min-distrib2*)

end

lemma *not-less-Nil* [*simp*]: ¬ *x* < []
  by (*simp add*: *list-less-def*)

lemma *Nil-less-Cons* [*simp*]: [] < *a* # *x*
  by (*simp add*: *list-less-def*)

lemma *Cons-less-Cons* [*simp*]: *a* # *x* < *b* # *y* ⟷ *a* < *b* ∨ *a* = *b* ∧ *x* < *y*
  by (*simp add*: *list-less-def*)

lemma *le-Nil* [*simp*]: *x* ≤ [] ⟷ *x* = []
  unfolding *list-le-def* **by** (*cases x*) *auto*

lemma *Nil-le-Cons* [*simp*]: [] ≤ *x*
  unfolding *list-le-def* **by** (*cases x*) *auto*

lemma *Cons-le-Cons* [*simp*]: *a* # *x* ≤ *b* # *y* ⟷ *a* < *b* ∨ *a* = *b* ∧ *x* ≤ *y*
  unfolding *list-le-def* **by** *auto*

instantiation *list* :: (*order*) *order-bot*
begin

definition *bot* = []

instance

by *standard* (*simp add*: *bot-list-def*)

end

lemma *less-list-code* [*code*]:
$xs < ([]::'a::\{equal, order\}\ list) \longleftrightarrow False$
$[] < (x::'a::\{equal, order\})\ \#\ xs \longleftrightarrow True$
$(x::'a::\{equal, order\})\ \#\ xs < y\ \#\ ys \longleftrightarrow x < y \lor x = y \land xs < ys$
by *simp-all*

lemma *less-eq-list-code* [*code*]:
$x\ \#\ xs \leq ([]::'a::\{equal, order\}\ list) \longleftrightarrow False$
$[] \leq (xs::'a::\{equal, order\}\ list) \longleftrightarrow True$
$(x::'a::\{equal, order\})\ \#\ xs \leq y\ \#\ ys \longleftrightarrow x < y \lor x = y \land xs \leq ys$
by *simp-all*

end

92 Sublist Ordering

theory *Sublist-Order*
imports *Sublist*
begin

This theory defines sublist ordering on lists. A list *ys* is a sublist of a list *xs*, iff one obtains *ys* by erasing some elements from *xs*.

92.1 Definitions and basic lemmas

instantiation *list* :: (*type*) *ord*
begin

definition
$(xs :: 'a\ list) \leq ys \longleftrightarrow sublisteq\ xs\ ys$

definition
$(xs :: 'a\ list) < ys \longleftrightarrow xs \leq ys \land \neg\ ys \leq xs$

instance ..

end

instance *list* :: (*type*) *order*
proof
fix *xs ys* :: *'a list*
show $xs < ys \longleftrightarrow xs \leq ys \land \neg\ ys \leq xs$ **unfolding** *less-list-def* **..**
next
fix *xs* :: *'a list*
show $xs \leq xs$ **by** (*simp add*: *less-eq-list-def*)

next
 fix *xs ys* :: *'a list*
 assume *xs* <= *ys* **and** *ys* <= *xs*
 thus *xs* = *ys* **by** (*unfold less-eq-list-def*) (*rule sublisteq-antisym*)
next
 fix *xs ys zs* :: *'a list*
 assume *xs* <= *ys* **and** *ys* <= *zs*
 thus *xs* <= *zs* **by** (*unfold less-eq-list-def*) (*rule sublisteq-trans*)
qed

lemmas *less-eq-list-induct* [*consumes 1*, *case-names empty drop take*] =
 list-emb.induct [*of op* =, *folded less-eq-list-def*]
lemmas *less-eq-list-drop* = *list-emb.list-emb-Cons* [*of op* =, *folded less-eq-list-def*]
lemmas *le-list-Cons2-iff* [*simp*, *code*] = *sublisteq-Cons2-iff* [*folded less-eq-list-def*]
lemmas *le-list-map* = *sublisteq-map* [*folded less-eq-list-def*]
lemmas *le-list-filter* = *sublisteq-filter* [*folded less-eq-list-def*]
lemmas *le-list-length* = *list-emb-length* [*of op* =, *folded less-eq-list-def*]

lemma *less-list-length*: *xs* < *ys* ⟹ *length xs* < *length ys*
 by (*metis list-emb-length sublisteq-same-length le-neq-implies-less less-list-def less-eq-list-def*)

lemma *less-list-empty* [*simp*]: [] < *xs* ⟷ *xs* ≠ []
 by (*metis less-eq-list-def list-emb-Nil order-less-le*)

lemma *less-list-below-empty* [*simp*]: *xs* < [] ⟷ *False*
 by (*metis list-emb-Nil less-eq-list-def less-list-def*)

lemma *less-list-drop*: *xs* < *ys* ⟹ *xs* < *x* # *ys*
 by (*unfold less-le less-eq-list-def*) (*auto*)

lemma *less-list-take-iff*: *x* # *xs* < *x* # *ys* ⟷ *xs* < *ys*
 by (*metis sublisteq-Cons2-iff less-list-def less-eq-list-def*)

lemma *less-list-drop-many*: *xs* < *ys* ⟹ *xs* < *zs* @ *ys*
 by (*metis sublisteq-append-le-same-iff sublisteq-drop-many order-less-le self-append-conv2 less-eq-list-def*)

lemma *less-list-take-many-iff*: *zs* @ *xs* < *zs* @ *ys* ⟷ *xs* < *ys*
 by (*metis less-list-def less-eq-list-def sublisteq-append'*)

lemma *less-list-rev-take*: *xs* @ *zs* < *ys* @ *zs* ⟷ *xs* < *ys*
 by (*unfold less-le less-eq-list-def*) *auto*

end

93 Lexicographic order on product types

theory *Product-Lexorder*
imports *Main*

begin

instantiation *prod* :: (*ord*, *ord*) *ord*
begin

definition
$x \leq y \longleftrightarrow fst\ x < fst\ y \lor fst\ x \leq fst\ y \land snd\ x \leq snd\ y$

definition
$x < y \longleftrightarrow fst\ x < fst\ y \lor fst\ x \leq fst\ y \land snd\ x < snd\ y$

instance ..

end

lemma *less-eq-prod-simp* [*simp*, *code*]:
$(x1, y1) \leq (x2, y2) \longleftrightarrow x1 < x2 \lor x1 \leq x2 \land y1 \leq y2$
by (*simp add*: *less-eq-prod-def*)

lemma *less-prod-simp* [*simp*, *code*]:
$(x1, y1) < (x2, y2) \longleftrightarrow x1 < x2 \lor x1 \leq x2 \land y1 < y2$
by (*simp add*: *less-prod-def*)

A stronger version for partial orders.

lemma *less-prod-def'*:
fixes $x\ y$ :: *'a::order* × *'b::ord*
shows $x < y \longleftrightarrow fst\ x < fst\ y \lor fst\ x = fst\ y \land snd\ x < snd\ y$
by (*auto simp add*: *less-prod-def le-less*)

instance *prod* :: (*preorder*, *preorder*) *preorder*
by *standard* (*auto simp*: *less-eq-prod-def less-prod-def less-le-not-le intro*: *order-trans*)

instance *prod* :: (*order*, *order*) *order*
by *standard* (*auto simp add*: *less-eq-prod-def*)

instance *prod* :: (*linorder*, *linorder*) *linorder*
by *standard* (*auto simp*: *less-eq-prod-def*)

instantiation *prod* :: (*linorder*, *linorder*) *distrib-lattice*
begin

definition
(*inf* :: *'a* × *'b* ⇒ - ⇒ -) = *min*

definition
(*sup* :: *'a* × *'b* ⇒ - ⇒ -) = *max*

instance
by *standard* (*auto simp add*: *inf-prod-def sup-prod-def max-min-distrib2*)

end

instantiation *prod* :: (*bot*, *bot*) *bot*
begin

definition
bot = (*bot*, *bot*)

instance ..

end

instance *prod* :: (*order-bot*, *order-bot*) *order-bot*
by *standard* (*auto simp add*: *bot-prod-def*)

instantiation *prod* :: (*top*, *top*) *top*
begin

definition
top = (*top*, *top*)

instance ..

end

instance *prod* :: (*order-top*, *order-top*) *order-top*
by *standard* (*auto simp add*: *top-prod-def*)

instance *prod* :: (*wellorder*, *wellorder*) *wellorder*
proof
fix $P :: 'a \times 'b \Rightarrow bool$ **and** $z :: 'a \times 'b$
assume P: $\bigwedge x. (\bigwedge y. y < x \Longrightarrow P\ y) \Longrightarrow P\ x$
show $P\ z$
proof (*induct z*)
case (*Pair a b*)
show $P\ (a,\ b)$
proof (*induct a arbitrary*: *b rule*: *less-induct*)
case (*less* a_1) **note** a_1 = *this*
show $P\ (a_1,\ b)$
proof (*induct b rule*: *less-induct*)
case (*less* b_1) **note** b_1 = *this*
show $P\ (a_1,\ b_1)$
proof (*rule P*)
fix p **assume** p: $p < (a_1,\ b_1)$
show $P\ p$
proof (*cases fst* $p < a_1$)
case *True*
then have P (*fst p*, *snd p*) **by** (*rule* a_1)

```
          then show ?thesis by simp
        next
          case False
          with p have 1: a₁ = fst p and 2: snd p < b₁
            by (simp-all add: less-prod-def')
          from 2 have P (a₁, snd p) by (rule b₁)
          with 1 show ?thesis by simp
        qed
      qed
    qed
  qed
 qed
qed
```

Legacy lemma bindings

lemmas *prod-le-def* = *less-eq-prod-def*
lemmas *prod-less-def* = *less-prod-def*
lemmas *prod-less-eq* = *less-prod-def'*

end

94 Pointwise order on product types

theory *Product-Order*
imports *Product-plus Conditionally-Complete-Lattices*
begin

94.1 Pointwise ordering

instantiation *prod* :: (*ord*, *ord*) *ord*
begin

definition
$x \le y \longleftrightarrow fst\ x \le fst\ y \wedge snd\ x \le snd\ y$

definition
$(x::'a \times 'b) < y \longleftrightarrow x \le y \wedge \neg\ y \le x$

instance ..

end

lemma *fst-mono*: $x \le y \Longrightarrow fst\ x \le fst\ y$
unfolding *less-eq-prod-def* **by** *simp*

lemma *snd-mono*: $x \le y \Longrightarrow snd\ x \le snd\ y$
unfolding *less-eq-prod-def* **by** *simp*

lemma *Pair-mono*: $x \le x' \Longrightarrow y \le y' \Longrightarrow (x, y) \le (x', y')$

unfolding *less-eq-prod-def* **by** *simp*

lemma *Pair-le* [*simp*]: $(a, b) \leq (c, d) \longleftrightarrow a \leq c \wedge b \leq d$
unfolding *less-eq-prod-def* **by** *simp*

instance *prod* :: (*preorder*, *preorder*) *preorder*
proof
fix $x\ y\ z :: 'a \times 'b$
show $x < y \longleftrightarrow x \leq y \wedge \neg\ y \leq x$
by (*rule less-prod-def*)
show $x \leq x$
unfolding *less-eq-prod-def*
by *fast*
assume $x \leq y$ **and** $y \leq z$ **thus** $x \leq z$
unfolding *less-eq-prod-def*
by (*fast elim*: *order-trans*)
qed

instance *prod* :: (*order*, *order*) *order*
by *standard auto*

94.2 Binary infimum and supremum

instantiation *prod* :: (*inf*, *inf*) *inf*
begin

definition *inf x y* = (*inf* (*fst x*) (*fst y*), *inf* (*snd x*) (*snd y*))

lemma *inf-Pair-Pair* [*simp*]: *inf* (*a*, *b*) (*c*, *d*) = (*inf a c*, *inf b d*)
unfolding *inf-prod-def* **by** *simp*

lemma *fst-inf* [*simp*]: *fst* (*inf x y*) = *inf* (*fst x*) (*fst y*)
unfolding *inf-prod-def* **by** *simp*

lemma *snd-inf* [*simp*]: *snd* (*inf x y*) = *inf* (*snd x*) (*snd y*)
unfolding *inf-prod-def* **by** *simp*

instance ..

end

instance *prod* :: (*semilattice-inf*, *semilattice-inf*) *semilattice-inf*
by *standard auto*

instantiation *prod* :: (*sup*, *sup*) *sup*
begin

definition

sup x y = (*sup* (*fst x*) (*fst y*), *sup* (*snd x*) (*snd y*))

lemma *sup-Pair-Pair* [*simp*]: *sup* (*a*, *b*) (*c*, *d*) = (*sup a c*, *sup b d*)
unfolding *sup-prod-def* **by** *simp*

lemma *fst-sup* [*simp*]: *fst* (*sup x y*) = *sup* (*fst x*) (*fst y*)
unfolding *sup-prod-def* **by** *simp*

lemma *snd-sup* [*simp*]: *snd* (*sup x y*) = *sup* (*snd x*) (*snd y*)
unfolding *sup-prod-def* **by** *simp*

instance ..

end

instance *prod* :: (*semilattice-sup*, *semilattice-sup*) *semilattice-sup*
by *standard auto*

instance *prod* :: (*lattice*, *lattice*) *lattice* **..**

instance *prod* :: (*distrib-lattice*, *distrib-lattice*) *distrib-lattice*
by *standard* (*auto simp add*: *sup-inf-distrib1*)

94.3 Top and bottom elements

instantiation *prod* :: (*top*, *top*) *top*
begin

definition
top = (*top*, *top*)

instance ..

end

lemma *fst-top* [*simp*]: *fst top* = *top*
unfolding *top-prod-def* **by** *simp*

lemma *snd-top* [*simp*]: *snd top* = *top*
unfolding *top-prod-def* **by** *simp*

lemma *Pair-top-top*: (*top*, *top*) = *top*
unfolding *top-prod-def* **by** *simp*

instance *prod* :: (*order-top*, *order-top*) *order-top*
by *standard* (*auto simp add*: *top-prod-def*)

instantiation *prod* :: (*bot*, *bot*) *bot*
begin

definition
bot = (*bot*, *bot*)

instance ..

end

lemma *fst-bot* [*simp*]: *fst bot* = *bot*
unfolding *bot-prod-def* **by** *simp*

lemma *snd-bot* [*simp*]: *snd bot* = *bot*
unfolding *bot-prod-def* **by** *simp*

lemma *Pair-bot-bot*: (*bot*, *bot*) = *bot*
unfolding *bot-prod-def* **by** *simp*

instance *prod* :: (*order-bot*, *order-bot*) *order-bot*
by *standard* (*auto simp add*: *bot-prod-def*)

instance *prod* :: (*bounded-lattice*, *bounded-lattice*) *bounded-lattice* **..**

instance *prod* :: (*boolean-algebra*, *boolean-algebra*) *boolean-algebra*
by *standard* (*auto simp add*: *prod-eqI diff-eq*)

94.4 Complete lattice operations

instantiation *prod* :: (*Inf*, *Inf*) *Inf*
begin

definition *Inf A* = (*INF x*:*A*. *fst x*, *INF x*:*A*. *snd x*)

instance ..

end

instantiation *prod* :: (*Sup*, *Sup*) *Sup*
begin

definition *Sup A* = (*SUP x*:*A*. *fst x*, *SUP x*:*A*. *snd x*)

instance ..

end

instance *prod* :: (*conditionally-complete-lattice*, *conditionally-complete-lattice*)
conditionally-complete-lattice
by *standard* (*force simp*: *less-eq-prod-def Inf-prod-def Sup-prod-def bdd-below-def bdd-above-def*

intro!: *cInf-lower cSup-upper cInf-greatest cSup-least*)+

instance *prod* :: (*complete-lattice*, *complete-lattice*) *complete-lattice*
by *standard* (*simp-all add*: *less-eq-prod-def Inf-prod-def Sup-prod-def INF-lower SUP-upper le-INF-iff SUP-le-iff bot-prod-def top-prod-def*)

lemma *fst-Sup*: *fst* (*Sup A*) = (*SUP x*:*A*. *fst x*)
unfolding *Sup-prod-def* **by** *simp*

lemma *snd-Sup*: *snd* (*Sup A*) = (*SUP x*:*A*. *snd x*)
unfolding *Sup-prod-def* **by** *simp*

lemma *fst-Inf*: *fst* (*Inf A*) = (*INF x*:*A*. *fst x*)
unfolding *Inf-prod-def* **by** *simp*

lemma *snd-Inf*: *snd* (*Inf A*) = (*INF x*:*A*. *snd x*)
unfolding *Inf-prod-def* **by** *simp*

lemma *fst-SUP*: *fst* (*SUP x*:*A*. *f x*) = (*SUP x*:*A*. *fst* (*f x*))
using *fst-Sup* [*of f ' A*, *symmetric*] **by** (*simp add*: *comp-def*)

lemma *snd-SUP*: *snd* (*SUP x*:*A*. *f x*) = (*SUP x*:*A*. *snd* (*f x*))
using *snd-Sup* [*of f ' A*, *symmetric*] **by** (*simp add*: *comp-def*)

lemma *fst-INF*: *fst* (*INF x*:*A*. *f x*) = (*INF x*:*A*. *fst* (*f x*))
using *fst-Inf* [*of f ' A*, *symmetric*] **by** (*simp add*: *comp-def*)

lemma *snd-INF*: *snd* (*INF x*:*A*. *f x*) = (*INF x*:*A*. *snd* (*f x*))
using *snd-Inf* [*of f ' A*, *symmetric*] **by** (*simp add*: *comp-def*)

lemma *SUP-Pair*: (*SUP x*:*A*. (*f x*, *g x*)) = (*SUP x*:*A*. *f x*, *SUP x*:*A*. *g x*)
unfolding *Sup-prod-def* **by** (*simp add*: *comp-def*)

lemma *INF-Pair*: (*INF x*:*A*. (*f x*, *g x*)) = (*INF x*:*A*. *f x*, *INF x*:*A*. *g x*)
unfolding *Inf-prod-def* **by** (*simp add*: *comp-def*)

Alternative formulations for set infima and suprema over the product of two complete lattices:

lemma *INF-prod-alt-def*:
INFIMUM A f = (*INFIMUM A* (*fst o f*), *INFIMUM A* (*snd o f*))
unfolding *Inf-prod-def* **by** *simp*

lemma *SUP-prod-alt-def*:
SUPREMUM A f = (*SUPREMUM A* (*fst o f*), *SUPREMUM A* (*snd o f*))
unfolding *Sup-prod-def* **by** *simp*

94.5 Complete distributive lattices

instance *prod* :: (*complete-distrib-lattice*, *complete-distrib-lattice*) *complete-distrib-lattice*

proof (*standard*, *goal-cases*)
case *1*
then show *?case*
by (*auto simp*: *sup-prod-def Inf-prod-def INF-prod-alt-def sup-Inf sup-INF comp-def*)
next
case *2*
then show *?case*
by (*auto simp*: *inf-prod-def Sup-prod-def SUP-prod-alt-def inf-Sup inf-SUP comp-def*)
qed

end

theory *Finite-Lattice*
imports *Product-Order*
begin

A non-empty finite lattice is a complete lattice. Since types are never empty in Isabelle/HOL, a type of classes *finite* and *lattice* should also have class *complete-lattice*. A type class is defined that extends classes *finite* and *lattice* with the operators *bot*, *top*, *Inf*, and *Sup*, along with assumptions that define these operators in terms of the ones of classes *finite* and *lattice*. The resulting class is a subclass of *complete-lattice*.

class *finite-lattice-complete* = *finite* + *lattice* + *bot* + *top* + *Inf* + *Sup* +
assumes *bot-def*: *bot* = *Inf-fin UNIV*
assumes *top-def*: *top* = *Sup-fin UNIV*
assumes *Inf-def*: *Inf A* = *Finite-Set.fold inf top A*
assumes *Sup-def*: *Sup A* = *Finite-Set.fold sup bot A*

The definitional assumptions on the operators *bot* and *top* of class *finite-lattice-complete* ensure that they yield bottom and top.

lemma *finite-lattice-complete-bot-least*: $(bot::'a::finite\text{-}lattice\text{-}complete) \leq x$
by (*auto simp*: *bot-def intro*: *Inf-fin.coboundedI*)

instance *finite-lattice-complete* ⊆ *order-bot*
by *standard* (*auto simp*: *finite-lattice-complete-bot-least*)

lemma *finite-lattice-complete-top-greatest*: $(top::'a::finite\text{-}lattice\text{-}complete) \geq x$
by (*auto simp*: *top-def Sup-fin.coboundedI*)

instance *finite-lattice-complete* ⊆ *order-top*
by *standard* (*auto simp*: *finite-lattice-complete-top-greatest*)

instance *finite-lattice-complete* ⊆ *bounded-lattice* **..**

The definitional assumptions on the operators *Inf* and *Sup* of class *finite-lattice-complete* ensure that they yield infimum and supremum.

lemma *finite-lattice-complete-Inf-empty*: *Inf* {} = (*top* :: ′*a*::*finite-lattice-complete*)
by (*simp add*: *Inf-def*)

lemma *finite-lattice-complete-Sup-empty*: *Sup* {} = (*bot* :: ′*a*::*finite-lattice-complete*)
by (*simp add*: *Sup-def*)

lemma *finite-lattice-complete-Inf-insert*:
fixes *A* :: ′*a*::*finite-lattice-complete set*
shows *Inf* (*insert x A*) = *inf x* (*Inf A*)
proof −
interpret *comp-fun-idem inf* :: ′*a* ⇒ -
by (*fact comp-fun-idem-inf*)
show *?thesis* **by** (*simp add*: *Inf-def*)
qed

lemma *finite-lattice-complete-Sup-insert*:
fixes *A* :: ′*a*::*finite-lattice-complete set*
shows *Sup* (*insert x A*) = *sup x* (*Sup A*)
proof −
interpret *comp-fun-idem sup* :: ′*a* ⇒ -
by (*fact comp-fun-idem-sup*)
show *?thesis* **by** (*simp add*: *Sup-def*)
qed

lemma *finite-lattice-complete-Inf-lower*:
(*x*::′*a*::*finite-lattice-complete*) ∈ *A* ⟹ *Inf A* ≤ *x*
using *finite* [*of A*]
by (*induct A*) (*auto simp add*: *finite-lattice-complete-Inf-insert intro*: *le-infI2*)

lemma *finite-lattice-complete-Inf-greatest*:
∀ *x*::′*a*::*finite-lattice-complete* ∈ *A*. *z* ≤ *x* ⟹ *z* ≤ *Inf A*
using *finite* [*of A*]
by (*induct A*) (*auto simp add*: *finite-lattice-complete-Inf-empty finite-lattice-complete-Inf-insert*)

lemma *finite-lattice-complete-Sup-upper*:
(*x*::′*a*::*finite-lattice-complete*) ∈ *A* ⟹ *Sup A* ≥ *x*
using *finite* [*of A*]
by (*induct A*) (*auto simp add*: *finite-lattice-complete-Sup-insert intro*: *le-supI2*)

lemma *finite-lattice-complete-Sup-least*:
∀ *x*::′*a*::*finite-lattice-complete* ∈ *A*. *z* ≥ *x* ⟹ *z* ≥ *Sup A*
using *finite* [*of A*]
by (*induct A*) (*auto simp add*: *finite-lattice-complete-Sup-empty finite-lattice-complete-Sup-insert*)

instance *finite-lattice-complete* ⊆ *complete-lattice*
proof
qed (*auto simp*:
finite-lattice-complete-Inf-lower
finite-lattice-complete-Inf-greatest

finite-lattice-complete-Sup-upper
finite-lattice-complete-Sup-least
finite-lattice-complete-Inf-empty
finite-lattice-complete-Sup-empty)

The product of two finite lattices is already a finite lattice.

lemma *finite-bot-prod*:
(*bot* :: (*'a*::*finite-lattice-complete* × *'b*::*finite-lattice-complete*)) =
Inf-fin UNIV
by (*metis Inf-fin.coboundedI UNIV-I bot.extremum-uniqueI finite-UNIV*)

lemma *finite-top-prod*:
(*top* :: (*'a*::*finite-lattice-complete* × *'b*::*finite-lattice-complete*)) =
Sup-fin UNIV
by (*metis Sup-fin.coboundedI UNIV-I top.extremum-uniqueI finite-UNIV*)

lemma *finite-Inf-prod*:
Inf(*A* :: (*'a*::*finite-lattice-complete* × *'b*::*finite-lattice-complete*) *set*) =
Finite-Set.fold inf top A
by (*metis Inf-fold-inf finite*)

lemma *finite-Sup-prod*:
Sup (*A* :: (*'a*::*finite-lattice-complete* × *'b*::*finite-lattice-complete*) *set*) =
Finite-Set.fold sup bot A
by (*metis Sup-fold-sup finite*)

instance *prod* :: (*finite-lattice-complete*, *finite-lattice-complete*) *finite-lattice-complete*
by *standard* (*auto simp*: *finite-bot-prod finite-top-prod finite-Inf-prod finite-Sup-prod*)

Functions with a finite domain and with a finite lattice as codomain already form a finite lattice.

lemma *finite-bot-fun*: (*bot* :: (*'a*::*finite* ⇒ *'b*::*finite-lattice-complete*)) = *Inf-fin UNIV*
by (*metis Inf-UNIV Inf-fin-Inf empty-not-UNIV finite*)

lemma *finite-top-fun*: (*top* :: (*'a*::*finite* ⇒ *'b*::*finite-lattice-complete*)) = *Sup-fin UNIV*
by (*metis Sup-UNIV Sup-fin-Sup empty-not-UNIV finite*)

lemma *finite-Inf-fun*:
Inf (*A*::(*'a*::*finite* ⇒ *'b*::*finite-lattice-complete*) *set*) =
Finite-Set.fold inf top A
by (*metis Inf-fold-inf finite*)

lemma *finite-Sup-fun*:
Sup (*A*::(*'a*::*finite* ⇒ *'b*::*finite-lattice-complete*) *set*) =
Finite-Set.fold sup bot A
by (*metis Sup-fold-sup finite*)

instance *fun* :: (*finite*, *finite-lattice-complete*) *finite-lattice-complete*
by *standard* (*auto simp*: *finite-bot-fun finite-top-fun finite-Inf-fun finite-Sup-fun*)

94.6 Finite Distributive Lattices

A finite distributive lattice is a complete lattice whose *inf* and *sup* operators distribute over *Sup* and *Inf*.

class *finite-distrib-lattice-complete* =
distrib-lattice + *finite-lattice-complete*

lemma *finite-distrib-lattice-complete-sup-Inf*:
sup (*x*::'*a*::*finite-distrib-lattice-complete*) (*Inf A*) = (*INF y*:*A*. *sup x y*)
using *finite*
by (*induct A rule*: *finite-induct*) (*simp-all add*: *sup-inf-distrib1*)

lemma *finite-distrib-lattice-complete-inf-Sup*:
inf (*x*::'*a*::*finite-distrib-lattice-complete*) (*Sup A*) = (*SUP y*:*A*. *inf x y*)
using *finite* [*of A*] **by** *induct* (*simp-all add*: *inf-sup-distrib1*)

instance *finite-distrib-lattice-complete* ⊆ *complete-distrib-lattice*
proof
qed (*auto simp*:
finite-distrib-lattice-complete-sup-Inf
finite-distrib-lattice-complete-inf-Sup)

The product of two finite distributive lattices is already a finite distributive lattice.

instance *prod* ::
(*finite-distrib-lattice-complete*, *finite-distrib-lattice-complete*)
finite-distrib-lattice-complete
..

Functions with a finite domain and with a finite distributive lattice as codomain already form a finite distributive lattice.

instance *fun* ::
(*finite*, *finite-distrib-lattice-complete*) *finite-distrib-lattice-complete*
..

94.7 Linear Orders

A linear order is a distributive lattice. A type class is defined that extends class *linorder* with the operators *inf* and *sup*, along with assumptions that define these operators in terms of the ones of class *linorder*. The resulting class is a subclass of *distrib-lattice*.

class *linorder-lattice* = *linorder* + *inf* + *sup* +
assumes *inf-def*: *inf x y* = (*if x* ≤ *y then x else y*)
assumes *sup-def*: *sup x y* = (*if x* ≥ *y then x else y*)

The definitional assumptions on the operators *inf* and *sup* of class *linorder-lattice* ensure that they yield infimum and supremum and that they distribute over each other.

lemma *linorder-lattice-inf-le1*: $inf\ (x{::}'a{::}linorder\text{-}lattice)\ y \leq x$
unfolding *inf-def* **by** (*metis* (*full-types*) *linorder-linear*)

lemma *linorder-lattice-inf-le2*: $inf\ (x{::}'a{::}linorder\text{-}lattice)\ y \leq y$
unfolding *inf-def* **by** (*metis* (*full-types*) *linorder-linear*)

lemma *linorder-lattice-inf-greatest*:
$(x{::}'a{::}linorder\text{-}lattice) \leq y \Longrightarrow x \leq z \Longrightarrow x \leq inf\ y\ z$
unfolding *inf-def* **by** (*metis* (*full-types*))

lemma *linorder-lattice-sup-ge1*: $sup\ (x{::}'a{::}linorder\text{-}lattice)\ y \geq x$
unfolding *sup-def* **by** (*metis* (*full-types*) *linorder-linear*)

lemma *linorder-lattice-sup-ge2*: $sup\ (x{::}'a{::}linorder\text{-}lattice)\ y \geq y$
unfolding *sup-def* **by** (*metis* (*full-types*) *linorder-linear*)

lemma *linorder-lattice-sup-least*:
$(x{::}'a{::}linorder\text{-}lattice) \geq y \Longrightarrow x \geq z \Longrightarrow x \geq sup\ y\ z$
by (*auto simp*: *sup-def*)

lemma *linorder-lattice-sup-inf-distrib1*:
$sup\ (x{::}'a{::}linorder\text{-}lattice)\ (inf\ y\ z) = inf\ (sup\ x\ y)\ (sup\ x\ z)$
by (*auto simp*: *inf-def sup-def*)

instance *linorder-lattice* ⊆ *distrib-lattice*
proof
qed (*auto simp*:
linorder-lattice-inf-le1
linorder-lattice-inf-le2
linorder-lattice-inf-greatest
linorder-lattice-sup-ge1
linorder-lattice-sup-ge2
linorder-lattice-sup-least
linorder-lattice-sup-inf-distrib1)

94.8 Finite Linear Orders

A (non-empty) finite linear order is a complete linear order.

class *finite-linorder-complete* = *linorder-lattice* + *finite-lattice-complete*

instance *finite-linorder-complete* ⊆ *complete-linorder* **..**

A (non-empty) finite linear order is a complete lattice whose *inf* and *sup* operators distribute over *Sup* and *Inf*.

instance *finite-linorder-complete* ⊆ *finite-distrib-lattice-complete* **..**

end

95 GCD and LCM on polynomials over a field

theory *Polynomial-GCD-euclidean*
imports *Main ~~/src/HOL/GCD ~~/src/HOL/Library/Polynomial*
begin

95.1 GCD of polynomials

instantiation *poly* :: (*field*) *gcd*
begin

function *gcd-poly* :: *'a::field poly ⇒ 'a poly ⇒ 'a poly*
where
  gcd (*x*::*'a poly*) *0* = *smult* (*inverse* (*coeff x* (*degree x*))) *x*
| *y* ≠ *0* ⟹ *gcd* (*x*::*'a poly*) *y* = *gcd y* (*x mod y*)
by *auto*

termination *gcd* :: *- poly ⇒ -*
by (*relation measure* (λ(*x*, *y*). *if y* = *0 then 0 else Suc* (*degree y*)))
  (*auto dest*: *degree-mod-less*)

declare *gcd-poly.simps* [*simp del*]

definition *lcm-poly* :: *'a::field poly ⇒ 'a poly ⇒ 'a poly*
where
  lcm-poly a b = *a* ∗ *b div smult* (*coeff a* (*degree a*) ∗ *coeff b* (*degree b*)) (*gcd a b*)

instance ..

end

lemma
  fixes *x y* :: *- poly*
  shows *poly-gcd-dvd1* [*iff*]: *gcd x y dvd x*
    and *poly-gcd-dvd2* [*iff*]: *gcd x y dvd y*
  apply (*induct x y rule*: *gcd-poly.induct*)
  apply (*simp-all add*: *gcd-poly.simps*)
  apply (*fastforce simp add*: *smult-dvd-iff dest*: *inverse-zero-imp-zero*)
  apply (*blast dest*: *dvd-mod-imp-dvd*)
  done

lemma *poly-gcd-greatest*:
  fixes *k x y* :: *- poly*
  shows *k dvd x* ⟹ *k dvd y* ⟹ *k dvd gcd x y*
  by (*induct x y rule*: *gcd-poly.induct*)
    (*simp-all add*: *gcd-poly.simps dvd-mod dvd-smult*)

lemma *dvd-poly-gcd-iff* [*iff*]:
 fixes *k x y* :: *- poly*
 shows *k dvd gcd x y* ⟷ *k dvd x* ∧ *k dvd y*
 by (*auto intro*!: *poly-gcd-greatest intro*: *dvd-trans* [*of - gcd x y*])

lemma *poly-gcd-monic*:
 fixes *x y* :: *- poly*
 shows *coeff* (*gcd x y*) (*degree* (*gcd x y*)) =
  (*if x* = *0* ∧ *y* = *0 then 0 else 1*)
 by (*induct x y rule*: *gcd-poly.induct*)
  (*simp-all add*: *gcd-poly.simps nonzero-imp-inverse-nonzero*)

lemma *poly-gcd-zero-iff* [*simp*]:
 fixes *x y* :: *- poly*
 shows *gcd x y* = *0* ⟷ *x* = *0* ∧ *y* = *0*
 by (*simp only*: *dvd-0-left-iff* [*symmetric*] *dvd-poly-gcd-iff*)

lemma *poly-gcd-0-0* [*simp*]:
 gcd (*0*::*- poly*) *0* = *0*
 by *simp*

lemma *poly-dvd-antisym*:
 fixes *p q* :: *'a*::*idom poly*
 assumes *coeff*: *coeff p* (*degree p*) = *coeff q* (*degree q*)
 assumes *dvd1*: *p dvd q* **and** *dvd2*: *q dvd p* **shows** *p* = *q*
proof (*cases p* = *0*)
 case *True* **with** *coeff* **show** *p* = *q* **by** *simp*
next
 case *False* **with** *coeff* **have** *q* ≠ *0* **by** *auto*
 have *degree*: *degree p* = *degree q*
  using ‹*p dvd q*› ‹*q dvd p*› ‹*p* ≠ *0*› ‹*q* ≠ *0*›
  by (*intro order-antisym dvd-imp-degree-le*)

 from ‹*p dvd q*› **obtain** *a* **where** *a*: *q* = *p* ∗ *a* **..**
 with ‹*q* ≠ *0*› **have** *a* ≠ *0* **by** *auto*
 with *degree a* ‹*p* ≠ *0*› **have** *degree a* = *0*
  by (*simp add*: *degree-mult-eq*)
 with *coeff a* **show** *p* = *q*
  by (*cases a*, *auto split*: *if-splits*)
qed

lemma *poly-gcd-unique*:
 fixes *d x y* :: *- poly*
 assumes *dvd1*: *d dvd x* **and** *dvd2*: *d dvd y*
  and *greatest*: ⋀*k*. *k dvd x* ⟹ *k dvd y* ⟹ *k dvd d*
  and *monic*: *coeff d* (*degree d*) = (*if x* = *0* ∧ *y* = *0 then 0 else 1*)
 shows *gcd x y* = *d*
proof −

have *coeff* (*gcd x y*) (*degree* (*gcd x y*)) = *coeff d* (*degree d*)
by (*simp-all add*: *poly-gcd-monic monic*)
moreover have *gcd x y dvd d*
using *poly-gcd-dvd1 poly-gcd-dvd2* **by** (*rule greatest*)
moreover have *d dvd gcd x y*
using *dvd1 dvd2* **by** (*rule poly-gcd-greatest*)
ultimately show *?thesis*
by (*rule poly-dvd-antisym*)
qed

instance *poly* :: (*field*) *semiring-gcd*
proof
fix *p q* :: *'a::field poly*
show *normalize* (*gcd p q*) = *gcd p q*
by (*induct p q rule*: *gcd-poly.induct*)
(*simp-all add*: *gcd-poly.simps normalize-poly-def*)
show *lcm p q* = *normalize* (*p* ∗ *q*) *div gcd p q*
by (*simp add*: *coeff-degree-mult div-smult-left div-smult-right lcm-poly-def normalize-poly-def*)
(*metis* (*no-types*, *lifting*) *div-smult-right inverse-mult-distrib inverse-zero mult.commute pdivmod-rel pdivmod-rel-def smult-eq-0-iff*)
qed *simp-all*

lemma *poly-gcd-1-left* [*simp*]: *gcd 1 y* = (*1* :: - *poly*)
by (*rule poly-gcd-unique*) *simp-all*

lemma *poly-gcd-1-right* [*simp*]: *gcd x 1* = (*1* :: - *poly*)
by (*rule poly-gcd-unique*) *simp-all*

lemma *poly-gcd-minus-left* [*simp*]: *gcd* (− *x*) *y* = *gcd x* (*y* :: - *poly*)
by (*rule poly-gcd-unique*) (*simp-all add*: *poly-gcd-monic*)

lemma *poly-gcd-minus-right* [*simp*]: *gcd x* (− *y*) = *gcd x* (*y* :: - *poly*)
by (*rule poly-gcd-unique*) (*simp-all add*: *poly-gcd-monic*)

lemma *poly-gcd-code* [*code*]:
gcd x y = (*if y* = *0 then normalize x else gcd y* (*x mod* (*y* :: - *poly*)))
by (*simp add*: *gcd-poly.simps*)

end

96 Implementation of mappings with Association Lists

theory *AList-Mapping*
imports *AList Mapping*
begin

lift-definition *Mapping* :: (*'a* × *'b*) *list* ⇒ (*'a*, *'b*) *mapping* **is** *map-of* **.**

code-datatype *Mapping*

lemma *lookup-Mapping* [*simp*, *code*]:
Mapping.lookup (*Mapping xs*) = *map-of xs*
by *transfer rule*

lemma *keys-Mapping* [*simp*, *code*]:
Mapping.keys (*Mapping xs*) = *set* (*map fst xs*)
by *transfer* (*simp add*: *dom-map-of-conv-image-fst*)

lemma *empty-Mapping* [*code*]:
Mapping.empty = *Mapping* []
by *transfer simp*

lemma *is-empty-Mapping* [*code*]:
Mapping.is-empty (*Mapping xs*) ⟷ *List.null xs*
by (*case-tac xs*) (*simp-all add*: *is-empty-def null-def*)

lemma *update-Mapping* [*code*]:
Mapping.update k v (*Mapping xs*) = *Mapping* (*AList.update k v xs*)
by *transfer* (*simp add*: *update-conv′*)

lemma *delete-Mapping* [*code*]:
Mapping.delete k (*Mapping xs*) = *Mapping* (*AList.delete k xs*)
by *transfer* (*simp add*: *delete-conv′*)

lemma *ordered-keys-Mapping* [*code*]:
Mapping.ordered-keys (*Mapping xs*) = *sort* (*remdups* (*map fst xs*))
by (*simp only*: *ordered-keys-def keys-Mapping sorted-list-of-set-sort-remdups*) *simp*

lemma *size-Mapping* [*code*]:
Mapping.size (*Mapping xs*) = *length* (*remdups* (*map fst xs*))
by (*simp add*: *size-def length-remdups-card-conv dom-map-of-conv-image-fst*)

lemma *tabulate-Mapping* [*code*]:
Mapping.tabulate ks f = *Mapping* (*map* (λ*k*. (*k*, *f k*)) *ks*)
by *transfer* (*simp add*: *map-of-map-restrict*)

lemma *bulkload-Mapping* [*code*]:
Mapping.bulkload vs = *Mapping* (*map* (λ*n*. (*n*, *vs* ! *n*)) [*0..<length vs*])
by *transfer* (*simp add*: *map-of-map-restrict fun-eq-iff*)

lemma *equal-Mapping* [*code*]:
HOL.equal (*Mapping xs*) (*Mapping ys*) ⟷
(*let ks* = *map fst xs*; *ls* = *map fst ys*
in (∀ *l*∈*set ls*. *l* ∈ *set ks*) ∧ (∀ *k*∈*set ks*. *k* ∈ *set ls* ∧ *map-of xs k* = *map-of ys k*))

proof –
  have *aux*: $\bigwedge a\ b\ xs.\ (a,\ b) \in set\ xs \Longrightarrow a \in fst\ `\ set\ xs$
    by (*auto simp add*: *image-def intro*!: *bexI*)
  show *?thesis* **apply** *transfer*
    by (*auto intro*!: *map-of-eqI*) (*auto dest*!: *map-of-eq-dom intro*: *aux*)
qed

lemma [*code nbe*]:
  $HOL.equal\ (x :: ('a,\ 'b)\ mapping)\ x \longleftrightarrow True$
  by (*fact equal-refl*)

end

97 Avoidance of pattern matching on natural numbers

theory *Code-Abstract-Nat*
imports *Main*
begin

When natural numbers are implemented in another than the conventional inductive *0/Suc* representation, it is necessary to avoid all pattern matching on natural numbers altogether. This is accomplished by this theory (up to a certain extent).

97.1 Case analysis

Case analysis on natural numbers is rephrased using a conditional expression:

lemma [*code*, *code-unfold*]:
  $case\text{-}nat = (\lambda f\ g\ n.\ if\ n = 0\ then\ f\ else\ g\ (n - 1))$
  by (*auto simp add*: *fun-eq-iff dest*!: *gr0-implies-Suc*)

97.2 Preprocessors

The term *Suc n* is no longer a valid pattern. Therefore, all occurrences of this term in a position where a pattern is expected (i.e. on the left-hand side of a code equation) must be eliminated. This can be accomplished – as far as possible – by applying the following transformation rule:

lemma *Suc-if-eq*:
  assumes $\bigwedge n.\ f\ (Suc\ n) \equiv h\ n$
  assumes $f\ 0 \equiv g$
  shows $f\ n \equiv if\ n = 0\ then\ g\ else\ h\ (n - 1)$
  by (*rule eq-reflection*) (*cases n*, *insert assms*, *simp-all*)

The rule above is built into a preprocessor that is plugged into the code generator.

```
setup ‹
let

val Suc-if-eq = Thm.incr-indexes 1 @{thm Suc-if-eq};

fun remove-suc ctxt thms =
  let
    val vname = singleton (Name.variant-list (map fst
      (fold (Term.add-var-names o Thm.full-prop-of) thms []))) n;
    val cv = Thm.cterm-of ctxt (Var ((vname, 0), HOLogic.natT));
    val lhs-of = snd o Thm.dest-comb o fst o Thm.dest-comb o Thm.cprop-of;
    val rhs-of = snd o Thm.dest-comb o Thm.cprop-of;
    fun find-vars ct = (case Thm.term-of ct of
        (Const (@{const-name Suc}, -) $ Var -) => [(cv, snd (Thm.dest-comb ct))]
      | - $ - =>
        let val (ct1, ct2) = Thm.dest-comb ct
        in
          map (apfst (fn ct => Thm.apply ct ct2)) (find-vars ct1) @
          map (apfst (Thm.apply ct1)) (find-vars ct2)
        end
      | - => []);
    val eqs = maps
      (fn thm => map (pair thm) (find-vars (lhs-of thm))) thms;
    fun mk-thms (thm, (ct, cv')) =
      let
        val thm' =
          Thm.implies-elim
           (Conv.fconv-rule (Thm.beta-conversion true)
             (Thm.instantiate'
               [SOME (Thm.ctyp-of-cterm ct)] [SOME (Thm.lambda cv ct),
                 SOME (Thm.lambda cv' (rhs-of thm)), NONE, SOME cv']
              Suc-if-eq)) (Thm.forall-intr cv' thm)
      in
        case map-filter (fn thm'' =>
            SOME (thm'', singleton
              (Variable.trade (K (fn [thm'''] => [thm''' RS thm']))
                (Variable.declare-thm thm'' ctxt)) thm'')
          handle THM - => NONE) thms of
            [] => NONE
          | thmps =>
              let val (thms1, thms2) = split-list thmps
              in SOME (subtract Thm.eq-thm (thm :: thms1) thms @ thms2) end
      end
  in get-first mk-thms eqs end;

fun eqn-suc-base-preproc ctxt thms =
  let
    val dest = fst o Logic.dest-equals o Thm.prop-of;
    val contains-suc = exists-Const (fn (c, -) => c = @{const-name Suc});
```

```
  in
    if forall (can dest) thms andalso exists (contains-suc o dest) thms
    then thms |> perhaps-loop (remove-suc ctxt) |> (Option.map o map) Drule.zero-var-indexes
      else NONE
  end;

val eqn-suc-preproc = Code-Preproc.simple-functrans eqn-suc-base-preproc;

in

  Code-Preproc.add-functrans (eqn-Suc, eqn-suc-preproc)

end;
⟩
```

end

98 Implementation of natural numbers as binary numerals

theory *Code-Binary-Nat*
imports *Code-Abstract-Nat*
begin

When generating code for functions on natural numbers, the canonical representation using *0* and *Suc* is unsuitable for computations involving large numbers. This theory refines the representation of natural numbers for code generation to use binary numerals, which do not grow linear in size but logarithmic.

98.1 Representation

code-datatype *0::nat nat-of-num*

lemma [*code*]:
  num-of-nat 0 = Num.One
  num-of-nat (nat-of-num k) = k
  by (*simp-all add: nat-of-num-inverse*)

lemma [*code*]:
  (*1::nat*) = *Numeral1*
  by *simp*

lemma [*code-abbrev*]: *Numeral1* = (*1::nat*)
  by *simp*

lemma [*code*]:
  Suc n = n + 1

by *simp*

98.2 Basic arithmetic

context
begin

lemma [*code*, *code del*]:
(*plus* :: *nat* ⇒ -) = *plus* **..**

lemma *plus-nat-code* [*code*]:
nat-of-num k + *nat-of-num l* = *nat-of-num* (*k* + *l*)
m + *0* = (*m*::*nat*)
0 + *n* = (*n*::*nat*)
by (*simp-all add*: *nat-of-num-numeral*)

Bounded subtraction needs some auxiliary

qualified definition *dup* :: *nat* ⇒ *nat* **where**
dup n = *n* + *n*

lemma *dup-code* [*code*]:
dup 0 = *0*
dup (*nat-of-num k*) = *nat-of-num* (*Num.Bit0 k*)
by (*simp-all add*: *dup-def numeral-Bit0*)

qualified definition *sub* :: *num* ⇒ *num* ⇒ *nat option* **where**
sub k l = (*if k* ≥ *l then Some* (*numeral k* − *numeral l*) *else None*)

lemma *sub-code* [*code*]:
sub Num.One Num.One = *Some 0*
sub (*Num.Bit0 m*) *Num.One* = *Some* (*nat-of-num* (*Num.BitM m*))
sub (*Num.Bit1 m*) *Num.One* = *Some* (*nat-of-num* (*Num.Bit0 m*))
sub Num.One (*Num.Bit0 n*) = *None*
sub Num.One (*Num.Bit1 n*) = *None*
sub (*Num.Bit0 m*) (*Num.Bit0 n*) = *map-option dup* (*sub m n*)
sub (*Num.Bit1 m*) (*Num.Bit1 n*) = *map-option dup* (*sub m n*)
sub (*Num.Bit1 m*) (*Num.Bit0 n*) = *map-option* (λ*q*. *dup q* + *1*) (*sub m n*)
sub (*Num.Bit0 m*) (*Num.Bit1 n*) = (*case sub m n of None* ⇒ *None*
| *Some q* ⇒ *if q* = *0 then None else Some* (*dup q* − *1*))
apply (*auto simp add*: *nat-of-num-numeral*
Num.dbl-def Num.dbl-inc-def Num.dbl-dec-def
Let-def le-imp-diff-is-add BitM-plus-one sub-def dup-def)
apply (*simp-all add*: *sub-non-positive*)
apply (*simp-all add*: *sub-non-negative* [*symmetric*, **where** *?'a* = *int*])
done

lemma [*code*, *code del*]:
(*minus* :: *nat* ⇒ -) = *minus* **..**

lemma *minus-nat-code* [*code*]:

nat-of-num k $-$ *nat-of-num l* = (*case sub k l of None* ⇒ *0* | *Some j* ⇒ *j*)
m $-$ *0* = (*m*::*nat*)
0 $-$ *n* = (*0*::*nat*)
by (*simp-all add*: *nat-of-num-numeral sub-non-positive sub-def*)

lemma [*code*, *code del*]:
(*times* :: *nat* ⇒ -) = *times* **..**

lemma *times-nat-code* [*code*]:
nat-of-num k $*$ *nat-of-num l* = *nat-of-num* (*k* $*$ *l*)
m $*$ *0* = (*0*::*nat*)
0 $*$ *n* = (*0*::*nat*)
by (*simp-all add*: *nat-of-num-numeral*)

lemma [*code*, *code del*]:
(*HOL.equal* :: *nat* ⇒ -) = *HOL.equal* **..**

lemma *equal-nat-code* [*code*]:
HOL.equal 0 (*0*::*nat*) ⟷ *True*
HOL.equal 0 (*nat-of-num l*) ⟷ *False*
HOL.equal (*nat-of-num k*) *0* ⟷ *False*
HOL.equal (*nat-of-num k*) (*nat-of-num l*) ⟷ *HOL.equal k l*
by (*simp-all add*: *nat-of-num-numeral equal*)

lemma *equal-nat-refl* [*code nbe*]:
HOL.equal (*n*::*nat*) *n* ⟷ *True*
by (*rule equal-refl*)

lemma [*code*, *code del*]:
(*less-eq* :: *nat* ⇒ -) = *less-eq* **..**

lemma *less-eq-nat-code* [*code*]:
0 ≤ (*n*::*nat*) ⟷ *True*
nat-of-num k ≤ *0* ⟷ *False*
nat-of-num k ≤ *nat-of-num l* ⟷ *k* ≤ *l*
by (*simp-all add*: *nat-of-num-numeral*)

lemma [*code*, *code del*]:
(*less* :: *nat* ⇒ -) = *less* **..**

lemma *less-nat-code* [*code*]:
(*m*::*nat*) < *0* ⟷ *False*
0 < *nat-of-num l* ⟷ *True*
nat-of-num k < *nat-of-num l* ⟷ *k* < *l*
by (*simp-all add*: *nat-of-num-numeral*)

lemma [*code*, *code del*]:
Divides.divmod-nat = *Divides.divmod-nat* **..**

lemma *divmod-nat-code* [*code*]:
  Divides.divmod-nat (*nat-of-num k*) (*nat-of-num l*) = *divmod k l*
  Divides.divmod-nat m 0 = (*0*, *m*)
  Divides.divmod-nat 0 n = (*0*, *0*)
  by (*simp-all add*: *prod-eq-iff nat-of-num-numeral*)

end

98.3 Conversions

lemma [*code*, *code del*]:
  of-nat = *of-nat* **..**

lemma *of-nat-code* [*code*]:
  of-nat 0 = *0*
  of-nat (*nat-of-num k*) = *numeral k*
  by (*simp-all add*: *nat-of-num-numeral*)

code-identifier
  code-module *Code-Binary-Nat* ⇀
    (*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

end

99 Code generation of pretty characters (and strings)

theory *Code-Char*
imports *Main Char-ord*
begin

code-printing
  type-constructor *char* ⇀
    (*SML*) *char*
    and (*OCaml*) *char*
    and (*Haskell*) *Prelude.Char*
    and (*Scala*) *Char*

setup ‹
  fold String-Code.add-literal-char [*SML*, *OCaml*, *Haskell*, *Scala*]
  #> *String-Code.add-literal-list-string Haskell*
›

code-printing
  constant *integer-of-char* ⇀
    (*SML*) !(*IntInf.fromInt o Char.ord*)
    and (*OCaml*) *Big'-int.big'-int'-of'-int* (*Char.code* -)
    and (*Haskell*) *Prelude.toInteger* (*Prelude.fromEnum* (- :: *Prelude.Char*))
    and (*Scala*) *BigInt*(-.*toInt*)

| **constant** *char-of-integer* ⇀
(*SML*) !(*Char.chr o IntInf.toInt*)
and (*OCaml*) *Char.chr* (*Big'-int.int'-of'-big'-int* -)
and (*Haskell*) !(*let chr k* | (*0* <= *k* && *k* < *256*) = *Prelude.toEnum k* :: *Prelude.Char in chr . Prelude.fromInteger*)
and (*Scala*) !((*k*: *BigInt*) => *if* (*BigInt*(*0*) <= *k* && *k* < *BigInt*(*256*)) *k.charValue else error*(*character value out of range*))
| **class-instance** *char* :: *equal* ⇀
(*Haskell*) −
| **constant** *HOL.equal* :: *char* ⇒ *char* ⇒ *bool* ⇀
(*SML*) !((- : *char*) = -)
and (*OCaml*) !((- : *char*) = -)
and (*Haskell*) **infix** *4* ==
and (*Scala*) **infixl** *5* ==
| **constant** *Code-Evaluation.term-of* :: *char* ⇒ *term* ⇀
(*Eval*) *HOLogic.mk'-char*/ (*IntInf.fromInt*/ (*Char.ord*/ -))

code-reserved *SML*
char

code-reserved *OCaml*
char

code-reserved *Scala*
char

code-reserved *SML String*

code-printing
constant *String.implode* ⇀
(*SML*) *String.implode*
and (*OCaml*) !(*let l* = - *in let res* = *String.create* (*List.length l*) *in let rec imp i* = *function* | [] −> *res* | *c* :: *l* −> *String.set res i c*; *imp* (*i* + *1*) *l in imp 0 l*)
and (*Haskell*) -
and (*Scala*) !(++/ -)
| **constant** *String.explode* ⇀
(*SML*) *String.explode*
and (*OCaml*) !(*let s* = - *in let rec exp i l* = *if i* < *0 then l else exp* (*i* − *1*) (*String.get s i* :: *l*) *in exp* (*String.length s* − *1*) [])
and (*Haskell*) -
and (*Scala*) !(-.*toList*)

code-printing
constant *Orderings.less-eq* :: *char* ⇒ *char* ⇒ *bool* ⇀
(*SML*) !((- : *char*) <= -)
and (*OCaml*) !((- : *char*) <= -)
and (*Haskell*) **infix** *4* <=
and (*Scala*) **infixl** *4* <=
and (*Eval*) **infixl** *6* <=

| **constant** *Orderings.less* :: *char* ⇒ *char* ⇒ *bool* ⇀
(*SML*) !((- : *char*) < -)
and (*OCaml*) !((- : *char*) < -)
and (*Haskell*) **infix** *4* <
and (*Scala*) **infixl** *4* <
and (*Eval*) **infixl** *6* <
| **constant** *Orderings.less-eq* :: *String.literal* ⇒ *String.literal* ⇒ *bool* ⇀
(*SML*) !((- : *string*) <= -)
and (*OCaml*) !((- : *string*) <= -)
— Order operations for *String.literal* work in Haskell only if no type class instance needs to be generated, because String = [Char] in Haskell and *char list* need not have the same order as *String.literal*.
and (*Haskell*) **infix** *4* <=
and (*Scala*) **infixl** *4* <=
and (*Eval*) **infixl** *6* <=
| **constant** *Orderings.less* :: *String.literal* ⇒ *String.literal* ⇒ *bool* ⇀
(*SML*) !((- : *string*) < -)
and (*OCaml*) !((- : *string*) < -)
and (*Haskell*) **infix** *4* <
and (*Scala*) **infixl** *4* <
and (*Eval*) **infixl** *6* <

end

100 Code generation of prolog programs

theory *Code-Prolog*
imports *Main*
keywords *values-prolog* :: *diag*
begin

ML-file ~~/*src*/*HOL*/*Tools*/*Predicate-Compile*/*code-prolog.ML*

101 Setup for Numerals

setup ‹*Predicate-Compile-Data.ignore-consts* [@{*const-name numeral*}]›

setup ‹*Predicate-Compile-Data.keep-functions* [@{*const-name numeral*}]›

end

102 Implementation of integer numbers by target-language integers

theory *Code-Target-Int*
imports ../*GCD*
begin

code-datatype *int-of-integer*

declare [[*code drop*: *integer-of-int*]]

context
includes *integer.lifting*
begin

lemma [*code*]:
integer-of-int (*int-of-integer k*) = *k*
by *transfer rule*

lemma [*code*]:
Int.Pos = *int-of-integer* ∘ *integer-of-num*
by *transfer* (*simp add*: *fun-eq-iff*)

lemma [*code*]:
Int.Neg = *int-of-integer* ∘ *uminus* ∘ *integer-of-num*
by *transfer* (*simp add*: *fun-eq-iff*)

lemma [*code-abbrev*]:
int-of-integer (*numeral k*) = *Int.Pos k*
by *transfer simp*

lemma [*code-abbrev*]:
int-of-integer (− *numeral k*) = *Int.Neg k*
by *transfer simp*

lemma [*code*, *symmetric*, *code-post*]:
0 = *int-of-integer 0*
by *transfer simp*

lemma [*code*, *symmetric*, *code-post*]:
1 = *int-of-integer 1*
by *transfer simp*

lemma [*code-post*]:
int-of-integer (− *1*) = − *1*
by *simp*

lemma [*code*]:
k + *l* = *int-of-integer* (*of-int k* + *of-int l*)
by *transfer simp*

lemma [*code*]:
− *k* = *int-of-integer* (− *of-int k*)
by *transfer simp*

lemma [*code*]:
k − *l* = *int-of-integer* (*of-int k* − *of-int l*)
by *transfer simp*

lemma [*code*]:
Int.dup k = *int-of-integer* (*Code-Numeral.dup* (*of-int k*))
by *transfer simp*

declare [[*code drop*: *Int.sub*]]

lemma [*code*]:
k ∗ *l* = *int-of-integer* (*of-int k* ∗ *of-int l*)
by *simp*

lemma [*code*]:
k div l = *int-of-integer* (*of-int k div of-int l*)
by *simp*

lemma [*code*]:
k mod l = *int-of-integer* (*of-int k mod of-int l*)
by *simp*

lemma [*code*]:
divmod m n = *map-prod int-of-integer int-of-integer* (*divmod m n*)
unfolding *prod-eq-iff divmod-def map-prod-def case-prod-beta fst-conv snd-conv*
by *transfer simp*

lemma [*code*]:
HOL.equal k l = *HOL.equal* (*of-int k* :: *integer*) (*of-int l*)
by *transfer* (*simp add*: *equal*)

lemma [*code*]:
k ≤ *l* ⟷ (*of-int k* :: *integer*) ≤ *of-int l*
by *transfer rule*

lemma [*code*]:
k < *l* ⟷ (*of-int k* :: *integer*) < *of-int l*
by *transfer rule*

lemma *gcd-int-of-integer* [*code*]:
gcd (*int-of-integer x*) (*int-of-integer y*) = *int-of-integer* (*gcd x y*)
by *transfer rule*

lemma *lcm-int-of-integer* [*code*]:
lcm (*int-of-integer x*) (*int-of-integer y*) = *int-of-integer* (*lcm x y*)
by *transfer rule*

end

lemma (**in** *ring-1*) *of-int-code-if*:
  of-int k = (*if k* = *0 then 0*
    else if k < *0 then* − *of-int* (− *k*)
    else let
      l = *2* ∗ *of-int* (*k div 2*);
      j = *k mod 2*
    in if j = *0 then l else l* + *1*)
proof −
  from *mod-div-equality* **have** ∗: *of-int k* = *of-int* (*k div 2* ∗ *2* + *k mod 2*) **by** *simp*
  show *?thesis*
    by (*simp add*: *Let-def of-int-add* [*symmetric*]) (*simp add*: ∗ *mult.commute*)
qed

declare *of-int-code-if* [*code*]

lemma [*code*]:
  nat = *nat-of-integer* ∘ *of-int*
  including *integer.lifting* **by** *transfer* (*simp add*: *fun-eq-iff*)

code-identifier
  code-module *Code-Target-Int* ⇀
    (*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

end

theory *Code-Real-Approx-By-Float*
imports *Complex-Main Code-Target-Int*
begin

WARNING This theory implements mathematical reals by machine reals (floats). This is inconsistent. See the proof of False at the end of the theory, where an equality on mathematical reals is (incorrectly) disproved by mapping it to machine reals.

The value command cannot display real results yet.

The only legitimate use of this theory is as a tool for code generation purposes.

code-printing
  type-constructor *real* ⇀
    (*SML*) *real*
    and (*OCaml*) *float*

code-printing
  constant *Ratreal* ⇀
    (*SML*) *error/ Bad constant*: *Ratreal*

code-printing
  constant *0* :: *real* ⇀

(*SML*) *0.0*
and (*OCaml*) *0.0*
declare *zero-real-code*[*code-unfold del*]

code-printing
constant *1* :: *real* ⇀
(*SML*) *1.0*
and (*OCaml*) *1.0*
declare *one-real-code*[*code-unfold del*]

code-printing
constant *HOL.equal* :: *real* ⇒ *real* ⇒ *bool* ⇀
(*SML*) *Real.==* ((-), (-))
and (*OCaml*) *Pervasives.*(=)

code-printing
constant *Orderings.less-eq* :: *real* ⇒ *real* ⇒ *bool* ⇀
(*SML*) *Real.*<= ((-), (-))
and (*OCaml*) *Pervasives.*(<=)

code-printing
constant *Orderings.less* :: *real* ⇒ *real* ⇒ *bool* ⇀
(*SML*) *Real.*< ((-), (-))
and (*OCaml*) *Pervasives.*(<)

code-printing
constant *op* + :: *real* ⇒ *real* ⇒ *real* ⇀
(*SML*) *Real.*+ ((-), (-))
and (*OCaml*) *Pervasives.*(+.)

code-printing
constant *op* ∗ :: *real* ⇒ *real* ⇒ *real* ⇀
(*SML*) *Real.*∗ ((-), (-))
and (*OCaml*) *Pervasives.*(∗.)

code-printing
constant *op* − :: *real* ⇒ *real* ⇒ *real* ⇀
(*SML*) *Real.*− ((-), (-))
and (*OCaml*) *Pervasives.*(−.)

code-printing
constant *uminus* :: *real* ⇒ *real* ⇀
(*SML*) *Real.*~
and (*OCaml*) *Pervasives.*(~−.)

code-printing
constant *op* / :: *real* ⇒ *real* ⇒ *real* ⇀
(*SML*) *Real.*'/ ((-), (-))
and (*OCaml*) *Pervasives.*('/.)

code-printing
  constant *HOL.equal* :: *real* ⇒ *real* ⇒ *bool* ⇀
    (*SML*) *Real.==* ((-:*real*), (-))

code-printing
  constant *sqrt* :: *real* ⇒ *real* ⇀
    (*SML*) *Math.sqrt*
    and (*OCaml*) *Pervasives.sqrt*
declare *sqrt-def* [*code del*]

context
begin

qualified definition *real-exp* :: *real* ⇒ *real* **where** *real-exp* = *exp*

lemma *exp-eq-real-exp*[*code-unfold*]: *exp* = *real-exp*
  unfolding *real-exp-def* **..**

end

code-printing
  constant *Code-Real-Approx-By-Float.real-exp* ⇀
    (*SML*) *Math.exp*
    and (*OCaml*) *Pervasives.exp*
declare *Code-Real-Approx-By-Float.real-exp-def* [*code del*]
declare *exp-def* [*code del*]

code-printing
  constant *ln* ⇀
    (*SML*) *Math.ln*
    and (*OCaml*) *Pervasives.ln*
declare *ln-real-def* [*code del*]

code-printing
  constant *cos* ⇀
    (*SML*) *Math.cos*
    and (*OCaml*) *Pervasives.cos*
declare *cos-def* [*code del*]

code-printing
  constant *sin* ⇀
    (*SML*) *Math.sin*
    and (*OCaml*) *Pervasives.sin*
declare *sin-def* [*code del*]

code-printing
  constant *pi* ⇀
    (*SML*) *Math.pi*

and (*OCaml*) *Pervasives.pi*
declare *pi-def* [*code del*]

code-printing
constant *arctan* ⇀
(*SML*) *Math.atan*
and (*OCaml*) *Pervasives.atan*
declare *arctan-def* [*code del*]

code-printing
constant *arccos* ⇀
(*SML*) *Math.scos*
and (*OCaml*) *Pervasives.acos*
declare *arccos-def* [*code del*]

code-printing
constant *arcsin* ⇀
(*SML*) *Math.asin*
and (*OCaml*) *Pervasives.asin*
declare *arcsin-def* [*code del*]

definition *real-of-integer* :: *integer* ⇒ *real* **where**
real-of-integer = *of-int* ∘ *int-of-integer*

code-printing
constant *real-of-integer* ⇀
(*SML*) *Real.fromInt*
and (*OCaml*) *Pervasives.float* (*Big′-int.int′-of′-big′-int* (-))

context
begin

qualified definition *real-of-int* :: *int* ⇒ *real* **where**
[*code-abbrev*]: *real-of-int* = *of-int*

lemma [*code*]:
real-of-int = *real-of-integer* ∘ *integer-of-int*
by (*simp add*: *fun-eq-iff real-of-integer-def real-of-int-def*)

lemma [*code-unfold del*]:
0 ≡ (*of-rat 0* :: *real*)
by *simp*

lemma [*code-unfold del*]:
1 ≡ (*of-rat 1* :: *real*)
by *simp*

lemma [*code-unfold del*]:
numeral k ≡ (*of-rat* (*numeral k*) :: *real*)

by *simp*

lemma [*code-unfold del*]:
− *numeral k* ≡ (*of-rat* (− *numeral k*) :: *real*)
by *simp*

end

code-printing
constant *Ratreal* ⇀ (*SML*)

definition *Realfract* :: *int* => *int* => *real*
where
Realfract p q = *of-int p* / *of-int q*

code-datatype *Realfract*

code-printing
constant *Realfract* ⇀ (*SML*) *Real.fromInt* -/ ′// *Real.fromInt* -

lemma [*code*]:
Ratreal r = *case-prod Realfract* (*quotient-of r*)
by (*cases r*) (*simp add*: *Realfract-def quotient-of-Fract of-rat-rat*)

lemma [*code*, *code del*]:
(*HOL.equal* :: *real*=>*real*=>*bool*) = (*HOL.equal* :: *real* => *real* => *bool*)
..

lemma [*code*, *code del*]:
(*plus* :: *real* => *real* => *real*) = *plus*
..

lemma [*code*, *code del*]:
(*uminus* :: *real* => *real*) = *uminus*
..

lemma [*code*, *code del*]:
(*minus* :: *real* => *real* => *real*) = *minus*
..

lemma [*code*, *code del*]:
(*times* :: *real* => *real* => *real*) = *times*
..

lemma [*code*, *code del*]:
(*divide* :: *real* => *real* => *real*) = *divide*
..

lemma [*code*]:

fixes *r* :: *real*
shows *inverse r* = *1* / *r*
by (*fact inverse-eq-divide*)

notepad
begin
have *cos* (*pi*/*2*) = *0* **by** (*rule cos-pi-half*)
moreover have *cos* (*pi*/*2*) ≠ *0* **by** *eval*
ultimately have *False* **by** *blast*
end

end

103 Implementation of natural numbers by target-language integers

theory *Code-Target-Nat*
imports *Code-Abstract-Nat*
begin

103.1 Implementation for *nat*

context
includes *natural.lifting integer.lifting*
begin

lift-definition *Nat* :: *integer* ⇒ *nat*
is *nat*
.

lemma [*code-post*]:
Nat 0 = *0*
Nat 1 = *1*
Nat (*numeral k*) = *numeral k*
by (*transfer*, *simp*)+

lemma [*code-abbrev*]:
integer-of-nat = *of-nat*
by *transfer rule*

lemma [*code-unfold*]:
Int.nat (*int-of-integer k*) = *nat-of-integer k*
by *transfer rule*

lemma [*code abstype*]:
Code-Target-Nat.Nat (*integer-of-nat n*) = *n*
by *transfer simp*

lemma [*code abstract*]:
integer-of-nat (*nat-of-integer k*) = *max 0 k*
by *transfer auto*

lemma [*code-abbrev*]:
nat-of-integer (*numeral k*) = *nat-of-num k*
by *transfer* (*simp add*: *nat-of-num-numeral*)

lemma [*code abstract*]:
integer-of-nat (*nat-of-num n*) = *integer-of-num n*
by *transfer* (*simp add*: *nat-of-num-numeral*)

lemma [*code abstract*]:
integer-of-nat 0 = *0*
by *transfer simp*

lemma [*code abstract*]:
integer-of-nat 1 = *1*
by *transfer simp*

lemma [*code*]:
Suc n = *n* + *1*
by *simp*

lemma [*code abstract*]:
integer-of-nat (*m* + *n*) = *of-nat m* + *of-nat n*
by *transfer simp*

lemma [*code abstract*]:
integer-of-nat (*m* − *n*) = *max 0* (*of-nat m* − *of-nat n*)
by *transfer simp*

lemma [*code abstract*]:
integer-of-nat (*m* ∗ *n*) = *of-nat m* ∗ *of-nat n*
by *transfer* (*simp add*: *of-nat-mult*)

lemma [*code abstract*]:
integer-of-nat (*m div n*) = *of-nat m div of-nat n*
by *transfer* (*simp add*: *zdiv-int*)

lemma [*code abstract*]:
integer-of-nat (*m mod n*) = *of-nat m mod of-nat n*
by *transfer* (*simp add*: *zmod-int*)

lemma [*code*]:
Divides.divmod-nat m n = (*m div n*, *m mod n*)
by (*fact divmod-nat-div-mod*)

lemma [*code*]:

divmod m n = map-prod nat-of-integer nat-of-integer (divmod m n)
by (*simp only*: *prod-eq-iff divmod-def map-prod-def case-prod-beta fst-conv snd-conv*)
(*transfer*, *simp-all only*: *nat-div-distrib nat-mod-distrib*
zero-le-numeral nat-numeral)

lemma [*code*]:
HOL.equal m n = HOL.equal (*of-nat m* :: *integer*) (*of-nat n*)
by *transfer* (*simp add*: *equal*)

lemma [*code*]:
$m \leq n \longleftrightarrow$ (*of-nat m* :: *integer*) $\leq$ *of-nat n*
by *simp*

lemma [*code*]:
$m < n \longleftrightarrow$ (*of-nat m* :: *integer*) $<$ *of-nat n*
by *simp*

lemma *num-of-nat-code* [*code*]:
num-of-nat = num-of-integer $\circ$ *of-nat*
by *transfer* (*simp add*: *fun-eq-iff*)

end

lemma (**in** *semiring-1*) *of-nat-code-if*:
of-nat n = (*if n = 0 then 0*
else let
(m, q) = *Divides.divmod-nat n 2*;
$m' = 2 *$ *of-nat m*
in if $q = 0$ *then* m' *else* $m' + 1$)
proof −
from *mod-div-equality* **have** $*$: *of-nat n = of-nat* (n *div* $2 * 2 + n$ *mod* 2) **by** *simp*
show *?thesis*
by (*simp add*: *Let-def divmod-nat-div-mod of-nat-add* [*symmetric*])
(*simp add*: $*$ *mult.commute of-nat-mult add.commute*)
qed

declare *of-nat-code-if* [*code*]

definition *int-of-nat* :: *nat* $\Rightarrow$ *int* **where**
[*code-abbrev*]: *int-of-nat = of-nat*

lemma [*code*]:
int-of-nat n = int-of-integer (*of-nat n*)
by (*simp add*: *int-of-nat-def*)

lemma [*code abstract*]:
integer-of-nat (*nat k*) = *max 0* (*integer-of-int k*)
including *integer.lifting* **by** *transfer auto*

lemma *term-of-nat-code* [*code*]:
— Use *nat-of-integer* in term reconstruction instead of *Code-Target-Nat.Nat* such that reconstructed terms can be fed back to the code generator
term-of-class.term-of n =
Code-Evaluation.App
(*Code-Evaluation.Const* (*STR ''Code-Numeral.nat-of-integer''*)
(*typerep.Typerep* (*STR ''fun''*)
[*typerep.Typerep* (*STR ''Code-Numeral.integer''*) [],
typerep.Typerep (*STR ''Nat.nat''*) []]))
(*term-of-class.term-of* (*integer-of-nat n*))
by (*simp add*: *term-of-anything*)

lemma *nat-of-integer-code-post* [*code-post*]:
nat-of-integer 0 = 0
nat-of-integer 1 = 1
nat-of-integer (*numeral k*) = *numeral k*
including *integer.lifting* **by** (*transfer*, *simp*)+

code-identifier
code-module *Code-Target-Nat* ⇀
(*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

end

104 Implementation of natural and integer numbers by target-language integers

theory *Code-Target-Numeral*
imports *Code-Target-Int Code-Target-Nat*
begin

end

105 Abstract type of association lists with unique keys

theory *DAList*
imports *AList*
begin

This was based on some existing fragments in the AFP-Collection framework.

105.1 Preliminaries

lemma *distinct-map-fst-filter*:

distinct (*map fst xs*) $\Longrightarrow$ *distinct* (*map fst* (*List.filter P xs*))
by (*induct xs*) *auto*

105.2 Type ('*key*, '*value*) *alist*

typedef ('*key*, '*value*) *alist* = {*xs* :: ('*key* × '*value*) *list*. (*distinct* ∘ *map fst*) *xs*}
morphisms *impl-of Alist*
proof
show [] ∈ {*xs*. (*distinct o map fst*) *xs*}
by *simp*
qed

setup-lifting *type-definition-alist*

lemma *alist-ext*: *impl-of xs* = *impl-of ys* $\Longrightarrow$ *xs* = *ys*
by (*simp add*: *impl-of-inject*)

lemma *alist-eq-iff*: *xs* = *ys* $\longleftrightarrow$ *impl-of xs* = *impl-of ys*
by (*simp add*: *impl-of-inject*)

lemma *impl-of-distinct* [*simp*, *intro*]: *distinct* (*map fst* (*impl-of xs*))
using *impl-of*[*of xs*] **by** *simp*

lemma *Alist-impl-of* [*code abstype*]: *Alist* (*impl-of xs*) = *xs*
by (*rule impl-of-inverse*)

105.3 Primitive operations

lift-definition *lookup* :: ('*key*, '*value*) *alist* ⇒ '*key* ⇒ '*value option* **is** *map-of* **.**

lift-definition *empty* :: ('*key*, '*value*) *alist* **is** []
by *simp*

lift-definition *update* :: '*key* ⇒ '*value* ⇒ ('*key*, '*value*) *alist* ⇒ ('*key*, '*value*) *alist*
is *AList.update*
by (*simp add*: *distinct-update*)

lift-definition *delete* :: '*key* ⇒ ('*key*, '*value*) *alist* ⇒ ('*key*, '*value*) *alist*
is *AList.delete*
by (*simp add*: *distinct-delete*)

lift-definition *map-entry* ::
'*key* ⇒ ('*value* ⇒ '*value*) ⇒ ('*key*, '*value*) *alist* ⇒ ('*key*, '*value*) *alist*
is *AList.map-entry*
by (*simp add*: *distinct-map-entry*)

lift-definition *filter* :: ('*key* × '*value* ⇒ *bool*) ⇒ ('*key*, '*value*) *alist* ⇒ ('*key*, '*value*) *alist*
is *List.filter*

by (*simp add*: *distinct-map-fst-filter*)

lift-definition *map-default* ::
'key ⇒ *'value* ⇒ (*'value* ⇒ *'value*) ⇒ (*'key*, *'value*) *alist* ⇒ (*'key*, *'value*) *alist*
is *AList.map-default*
by (*simp add*: *distinct-map-default*)

105.4 Abstract operation properties

lemma *lookup-empty* [*simp*]: *lookup empty k* = *None*
by (*simp add*: *empty-def lookup-def Alist-inverse*)

lemma *lookup-delete* [*simp*]: *lookup* (*delete k al*) = (*lookup al*)(*k* := *None*)
by (*simp add*: *lookup-def delete-def Alist-inverse distinct-delete delete-conv'*)

105.5 Further operations

105.5.1 Equality

instantiation *alist* :: (*equal*, *equal*) *equal*
begin

definition *HOL.equal* (*xs* :: (*'a*, *'b*) *alist*) *ys* == *impl-of xs* = *impl-of ys*

instance
by *standard* (*simp add*: *equal-alist-def impl-of-inject*)

end

105.5.2 Size

instantiation *alist* :: (*type*, *type*) *size*
begin

definition *size* (*al* :: (*'a*, *'b*) *alist*) = *length* (*impl-of al*)

instance ..

end

105.6 Quickcheck generators

notation *fcomp* (**infixl** ∘> *60*)
notation *scomp* (**infixl** ∘→ *60*)

definition (**in** *term-syntax*)
valterm-empty :: (*'key* :: *typerep*, *'value* :: *typerep*) *alist* × (*unit* ⇒ *Code-Evaluation.term*)
where *valterm-empty* = *Code-Evaluation.valtermify empty*

definition (**in** *term-syntax*)

valterm-update :: *'key* :: *typerep* × (*unit* ⇒ *Code-Evaluation.term*) ⇒
'value :: *typerep* × (*unit* ⇒ *Code-Evaluation.term*) ⇒
(*'key*, *'value*) *alist* × (*unit* ⇒ *Code-Evaluation.term*) ⇒
(*'key*, *'value*) *alist* × (*unit* ⇒ *Code-Evaluation.term*) **where**
[*code-unfold*]: *valterm-update k v a* = *Code-Evaluation.valtermify update* {·} *k* {·}
v {·}*a*

fun (**in** *term-syntax*) *random-aux-alist*
where
random-aux-alist i j =
(*if i* = *0 then Pair valterm-empty*
else Quickcheck-Random.collapse
(*Random.select-weight*
[(*i*, *Quickcheck-Random.random j* ∘→ (λ*k*. *Quickcheck-Random.random j*
∘→
(λ*v*. *random-aux-alist* (*i* − *1*) *j* ∘→ (λ*a*. *Pair* (*valterm-update k v a*))))),
(*1*, *Pair valterm-empty*)]))

instantiation *alist* :: (*random*, *random*) *random*
begin

definition *random-alist*
where
random-alist i = *random-aux-alist i i*

instance ..

end

no-notation *fcomp* (**infixl** ∘> *60*)
no-notation *scomp* (**infixl** ∘→ *60*)

instantiation *alist* :: (*exhaustive*, *exhaustive*) *exhaustive*
begin

fun *exhaustive-alist* ::
((*'a*, *'b*) *alist* ⇒ (*bool* × *term list*) *option*) ⇒ *natural* ⇒ (*bool* × *term list*) *option*
where
exhaustive-alist f i =
(*if i* = *0 then None*
else
case f empty of
Some ts ⇒ *Some ts*
| *None* ⇒
exhaustive-alist
(λ*a*. *Quickcheck-Exhaustive.exhaustive*
(λ*k*. *Quickcheck-Exhaustive.exhaustive* (λ*v*. *f* (*update k v a*)) (*i* − *1*))
(*i* − *1*))
(*i* − *1*))

instance ..

end

instantiation *alist* :: (*full-exhaustive*, *full-exhaustive*) *full-exhaustive*
begin

fun *full-exhaustive-alist* ::
 (('a, 'b) alist × (unit ⇒ term) ⇒ (bool × term list) option) ⇒ natural ⇒
  (bool × term list) option
where
 full-exhaustive-alist f i =
  (*if i* = *0 then None*
   else
    case f valterm-empty of
     Some ts ⇒ *Some ts*
    | *None* ⇒
      full-exhaustive-alist
       (λ*a*.
        Quickcheck-Exhaustive.full-exhaustive
         (λ*k*. *Quickcheck-Exhaustive.full-exhaustive* (λ*v*. *f* (*valterm-update k v a*)) (*i* − *1*))
        (*i* − *1*))
       (*i* − *1*))

instance ..

end

106 alist is a BNF

lift-bnf (*dead* 'k, *set*: 'v) *alist* [*wits*: [] :: ('k × 'v) *list*] **for** *map*: *map rel*: *rel*
 by *auto*

hide-const *valterm-empty valterm-update random-aux-alist*

hide-fact (**open**) *lookup-def empty-def update-def delete-def map-entry-def filter-def map-default-def*
hide-const (**open**) *impl-of lookup empty update delete map-entry filter map-default map set rel*

end

107 Multisets partially implemented by association lists

theory *DAList-Multiset*

imports *Multiset DAList*
begin

Delete prexisting code equations

lemma [*code*, *code del*]: {#} = {#} ..

lemma [*code*, *code del*]: *single* = *single* ..

lemma [*code*, *code del*]: *plus* = (*plus* :: *'a multiset* ⇒ -) ..

lemma [*code*, *code del*]: *minus* = (*minus* :: *'a multiset* ⇒ -) ..

lemma [*code*, *code del*]: *inf-subset-mset* = (*inf-subset-mset* :: *'a multiset* ⇒ -) ..

lemma [*code*, *code del*]: *sup-subset-mset* = (*sup-subset-mset* :: *'a multiset* ⇒ -) ..

lemma [*code*, *code del*]: *image-mset* = *image-mset* ..

lemma [*code*, *code del*]: *filter-mset* = *filter-mset* ..

lemma [*code*, *code del*]: *count* = *count* ..

lemma [*code*, *code del*]: *size* = (*size* :: - *multiset* ⇒ *nat*) ..

lemma [*code*, *code del*]: *msetsum* = *msetsum* ..

lemma [*code*, *code del*]: *msetprod* = *msetprod* ..

lemma [*code*, *code del*]: *set-mset* = *set-mset* ..

lemma [*code*, *code del*]: *sorted-list-of-multiset* = *sorted-list-of-multiset* ..

lemma [*code*, *code del*]: *subset-mset* = *subset-mset* ..

lemma [*code*, *code del*]: *subseteq-mset* = *subseteq-mset* ..

lemma [*code*, *code del*]: *equal-multiset-inst.equal-multiset* = *equal-multiset-inst.equal-multiset* ..

Raw operations on lists

definition *join-raw* ::
  (*'key* ⇒ *'val* × *'val* ⇒ *'val*) ⇒
    (*'key* × *'val*) *list* ⇒ (*'key* × *'val*) *list* ⇒ (*'key* × *'val*) *list*
  where *join-raw f xs ys* = *foldr* (λ(*k*, *v*). *map-default k v* (λ*v'*. *f k* (*v'*, *v*))) *ys xs*

lemma *join-raw-Nil* [*simp*]: *join-raw f xs* [] = *xs*
  by (*simp add*: *join-raw-def*)

lemma *join-raw-Cons* [*simp*]:

join-raw f xs ((*k*, *v*) # *ys*) = *map-default k v* (λ*v'*. *f k* (*v'*, *v*)) (*join-raw f xs ys*)
by (*simp add*: *join-raw-def*)

lemma *map-of-join-raw*:
assumes *distinct* (*map fst ys*)
shows *map-of* (*join-raw f xs ys*) *x* =
(*case map-of xs x of*
None ⇒ *map-of ys x*
| *Some v* ⇒ (*case map-of ys x of None* ⇒ *Some v* | *Some v'* ⇒ *Some* (*f x* (*v*, *v'*))))
using *assms*
apply (*induct ys*)
apply (*auto simp add*: *map-of-map-default split*: *option.split*)
apply (*metis map-of-eq-None-iff option.simps*(*2*) *weak-map-of-SomeI*)
apply (*metis Some-eq-map-of-iff map-of-eq-None-iff option.simps*(*2*))
done

lemma *distinct-join-raw*:
assumes *distinct* (*map fst xs*)
shows *distinct* (*map fst* (*join-raw f xs ys*))
using *assms*
proof (*induct ys*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons y ys*)
then show *?case* **by** (*cases y*) (*simp add*: *distinct-map-default*)
qed

definition *subtract-entries-raw xs ys* = *foldr* (λ(*k*, *v*). *AList.map-entry k* (λ*v'*. *v'* − *v*)) *ys xs*

lemma *map-of-subtract-entries-raw*:
assumes *distinct* (*map fst ys*)
shows *map-of* (*subtract-entries-raw xs ys*) *x* =
(*case map-of xs x of*
None ⇒ *None*
| *Some v* ⇒ (*case map-of ys x of None* ⇒ *Some v* | *Some v'* ⇒ *Some* (*v* − *v'*)))
using *assms*
unfolding *subtract-entries-raw-def*
apply (*induct ys*)
apply *auto*
apply (*simp split*: *option.split*)
apply (*simp add*: *map-of-map-entry*)
apply (*auto split*: *option.split*)
apply (*metis map-of-eq-None-iff option.simps*(*3*) *option.simps*(*4*))
apply (*metis map-of-eq-None-iff option.simps*(*4*) *option.simps*(*5*))
done

lemma *distinct-subtract-entries-raw*:
assumes *distinct* (*map fst xs*)
shows *distinct* (*map fst* (*subtract-entries-raw xs ys*))
using *assms*
unfolding *subtract-entries-raw-def*
by (*induct ys*) (*auto simp add*: *distinct-map-entry*)

Operations on alists with distinct keys

lift-definition *join* :: ($'a \Rightarrow 'b \times 'b \Rightarrow 'b$) $\Rightarrow$ ($'a$, $'b$) *alist* $\Rightarrow$ ($'a$, $'b$) *alist* $\Rightarrow$ ($'a$, $'b$) *alist*
is *join-raw*
by (*simp add*: *distinct-join-raw*)

lift-definition *subtract-entries* :: ($'a$, ($'b$ :: *minus*)) *alist* $\Rightarrow$ ($'a$, $'b$) *alist* $\Rightarrow$ ($'a$, $'b$) *alist*
is *subtract-entries-raw*
by (*simp add*: *distinct-subtract-entries-raw*)

Implementing multisets by means of association lists

definition *count-of* :: ($'a \times nat$) *list* $\Rightarrow$ $'a \Rightarrow nat$
where *count-of xs x* = (*case map-of xs x of None* $\Rightarrow$ *0* | *Some n* $\Rightarrow$ *n*)

lemma *count-of-multiset*: *count-of xs* $\in$ *multiset*
proof −
let *?A* = {*x*::$'a$. *0* < (*case map-of xs x of None* $\Rightarrow$ *0*::*nat* | *Some n* $\Rightarrow$ *n*)}
have *?A* $\subseteq$ *dom* (*map-of xs*)
proof
fix *x*
assume *x* $\in$ *?A*
then have *0* < (*case map-of xs x of None* $\Rightarrow$ *0*::*nat* | *Some n* $\Rightarrow$ *n*)
by *simp*
then have *map-of xs x* $\neq$ *None*
by (*cases map-of xs x*) *auto*
then show *x* $\in$ *dom* (*map-of xs*)
by *auto*
qed
with *finite-dom-map-of* [*of xs*] **have** *finite ?A*
by (*auto intro*: *finite-subset*)
then show *?thesis*
by (*simp add*: *count-of-def fun-eq-iff multiset-def*)
qed

lemma *count-simps* [*simp*]:
count-of [] = (λ-. *0*)
count-of ((*x*, *n*) # *xs*) = (λ*y*. *if x* = *y then n else count-of xs y*)
by (*simp-all add*: *count-of-def fun-eq-iff*)

lemma *count-of-empty*: *x* $\notin$ *fst* ` *set xs* $\Longrightarrow$ *count-of xs x* = *0*
by (*induct xs*) (*simp-all add*: *count-of-def*)

lemma *count-of-filter*: *count-of* (*List.filter* ($P \circ$ *fst*) *xs*) x = (*if P x then count-of xs x else 0*)
  by (*induct xs*) *auto*

lemma *count-of-map-default* [*simp*]:
  count-of (*map-default x b* ($\lambda x.\ x + b$) *xs*) y =
    (*if* $x = y$ *then count-of xs x* + *b else count-of xs y*)
  unfolding *count-of-def* **by** (*simp add*: *map-of-map-default split*: *option.split*)

lemma *count-of-join-raw*:
  distinct (*map fst ys*) $\Longrightarrow$
    count-of xs x + *count-of ys x* = *count-of* (*join-raw* ($\lambda x\ (x, y).\ x + y$) *xs ys*) x
  unfolding *count-of-def* **by** (*simp add*: *map-of-join-raw split*: *option.split*)

lemma *count-of-subtract-entries-raw*:
  distinct (*map fst ys*) $\Longrightarrow$
    count-of xs x − *count-of ys x* = *count-of* (*subtract-entries-raw xs ys*) x
  unfolding *count-of-def* **by** (*simp add*: *map-of-subtract-entries-raw split*: *option.split*)

Code equations for multiset operations

definition *Bag* :: ('*a*, *nat*) *alist* ⇒ '*a multiset*
  where *Bag xs* = *Abs-multiset* (*count-of* (*DAList.impl-of xs*))

code-datatype *Bag*

lemma *count-Bag* [*simp*, *code*]: *count* (*Bag xs*) = *count-of* (*DAList.impl-of xs*)
  by (*simp add*: *Bag-def count-of-multiset*)

lemma *Mempty-Bag* [*code*]: {#} = *Bag* (*DAList.empty*)
  by (*simp add*: *multiset-eq-iff alist.Alist-inverse DAList.empty-def*)

lemma *single-Bag* [*code*]: {#*x*#} = *Bag* (*DAList.update x 1 DAList.empty*)
  by (*simp add*: *multiset-eq-iff alist.Alist-inverse update.rep-eq empty.rep-eq*)

lemma *union-Bag* [*code*]: *Bag xs* + *Bag ys* = *Bag* (*join* ($\lambda x\ (n1, n2).\ n1 + n2$) *xs ys*)
  by (*rule multiset-eqI*)
    (*simp add*: *count-of-join-raw alist.Alist-inverse distinct-join-raw join-def*)

lemma *minus-Bag* [*code*]: *Bag xs* − *Bag ys* = *Bag* (*subtract-entries xs ys*)
  by (*rule multiset-eqI*)
    (*simp add*: *count-of-subtract-entries-raw alist.Alist-inverse*
      distinct-subtract-entries-raw subtract-entries-def)

lemma *filter-Bag* [*code*]: *filter-mset P* (*Bag xs*) = *Bag* (*DAList.filter* ($P \circ$ *fst*) *xs*)
  by (*rule multiset-eqI*) (*simp add*: *count-of-filter DAList.filter.rep-eq*)

lemma *mset-eq* [*code*]: *HOL.equal* (*m1*::′*a*::*equal multiset*) *m2* ⟷ *m1* ≤# *m2* ∧ *m2* ≤# *m1*
 by (*metis equal-multiset-def subset-mset.eq-iff*)

By default the code for < is ($xs < ys$) = ($xs \leq ys \wedge xs \neq ys$). With equality implemented by ≤, this leads to three calls of ≤. Here is a more efficient version:

lemma *mset-less*[*code*]: *xs* <# (*ys* :: ′*a multiset*) ⟷ *xs* ≤# *ys* ∧ ¬ *ys* ≤# *xs*
 by (*rule subset-mset.less-le-not-le*)

lemma *mset-less-eq-Bag0*:
 Bag xs ≤# *A* ⟷ (∀ (*x*, *n*) ∈ *set* (*DAList.impl-of xs*). *count-of* (*DAList.impl-of xs*) *x* ≤ *count A x*)
  (**is** *?lhs* ⟷ *?rhs*)
proof
 assume *?lhs*
 then show *?rhs* **by** (*auto simp add*: *subseteq-mset-def*)
next
 assume *?rhs*
 show *?lhs*
 proof (*rule mset-less-eqI*)
  fix *x*
  from ‹*?rhs*› **have** *count-of* (*DAList.impl-of xs*) *x* ≤ *count A x*
   by (*cases x* ∈ *fst* ‘ *set* (*DAList.impl-of xs*)) (*auto simp add*: *count-of-empty*)
  then show *count* (*Bag xs*) *x* ≤ *count A x* **by** (*simp add*: *subset-mset-def*)
 qed
qed

lemma *mset-less-eq-Bag* [*code*]:
 Bag xs ≤# (*A* :: ′*a multiset*) ⟷ (∀ (*x*, *n*) ∈ *set* (*DAList.impl-of xs*). *n* ≤ *count A x*)
proof −
 {
  fix *x n*
  assume (*x*,*n*) ∈ *set* (*DAList.impl-of xs*)
  then have *count-of* (*DAList.impl-of xs*) *x* = *n*
  proof *transfer*
   fix *x n*
   fix *xs* :: (′*a* × *nat*) *list*
   show (*distinct* ∘ *map fst*) *xs* ⟹ (*x*, *n*) ∈ *set xs* ⟹ *count-of xs x* = *n*
   proof (*induct xs*)
    case *Nil*
    then show *?case* **by** *simp*
   next
    case (*Cons ym ys*)
    obtain *y m* **where** *ym*: *ym* = (*y*,*m*) **by** *force*
    note *Cons* = *Cons*[*unfolded ym*]
    show *?case*

```
        proof (cases x = y)
          case False
          with Cons show ?thesis
            unfolding ym by auto
        next
          case True
          with Cons(2−3) have m = n by force
          with True show ?thesis
            unfolding ym by auto
        qed
      qed
    qed
  }
  then show ?thesis
    unfolding mset-less-eq-Bag0 by auto
qed

declare multiset-inter-def [code]
declare sup-subset-mset-def [code]
declare mset.simps [code]


fun fold-impl :: ('a ⇒ nat ⇒ 'b ⇒ 'b) ⇒ 'b ⇒ ('a × nat) list ⇒ 'b
where
  fold-impl fn e ((a,n) # ms) = (fold-impl fn ((fn a n) e) ms)
| fold-impl fn e [] = e

context
begin

qualified definition fold :: ('a ⇒ nat ⇒ 'b ⇒ 'b) ⇒ 'b ⇒ ('a, nat) alist ⇒ 'b
  where fold f e al = fold-impl f e (DAList.impl-of al)

end

context comp-fun-commute
begin

lemma DAList-Multiset-fold:
  assumes fn: ⋀a n x. fn a n x = (f a ^^ n) x
  shows fold-mset f e (Bag al) = DAList-Multiset.fold fn e al
  unfolding DAList-Multiset.fold-def
proof (induct al)
  fix ys
  let ?inv = {xs :: ('a × nat) list. (distinct ∘ map fst) xs}
  note cs[simp del] = count-simps
  have count[simp]: ⋀x. count (Abs-multiset (count-of x)) = count-of x
    by (rule Abs-multiset-inverse[OF count-of-multiset])
  assume ys: ys ∈ ?inv
```

then show *fold-mset f e* (*Bag* (*Alist ys*)) = *fold-impl fn e* (*DAList.impl-of* (*Alist ys*))
  unfolding *Bag-def* **unfolding** *Alist-inverse*[*OF ys*]
 proof (*induct ys arbitrary*: *e rule*: *list.induct*)
  case *Nil*
  show *?case*
   by (*rule trans*[*OF arg-cong*[*of* - {#} *fold-mset f e*, *OF multiset-eqI*]])
    (*auto*, *simp add*: *cs*)
 next
  case (*Cons pair ys e*)
  obtain *a n* **where** *pair*: *pair* = (*a*,*n*)
   by *force*
  from *fn*[*of a n*] **have** [*simp*]: *fn a n* = (*f a* ^^ *n*)
   by *auto*
  have *inv*: *ys* ∈ *?inv*
   using *Cons*(*2*) **by** *auto*
  note *IH* = *Cons*(*1*)[*OF inv*]
  def *Ys* ≡ *Abs-multiset* (*count-of ys*)
  have *id*: *Abs-multiset* (*count-of* ((*a*, *n*) # *ys*)) = ((*op* + {# *a* #}) ^^ *n*) *Ys*
   unfolding *Ys-def*
  proof (*rule multiset-eqI*, *unfold count*)
   fix *c*
   show *count-of* ((*a*, *n*) # *ys*) *c* =
    count ((*op* + {#*a*#} ^^ *n*) (*Abs-multiset* (*count-of ys*))) *c* (**is** *?l* = *?r*)
   proof (*cases c* = *a*)
    case *False*
    then show *?thesis*
     unfolding *cs* **by** (*induct n*) *auto*
   next
    case *True*
    then have *?l* = *n* **by** (*simp add*: *cs*)
    also have *n* = *?r* **unfolding** *True*
    proof (*induct n*)
     case *0*
     from *Cons*(*2*)[*unfolded pair*] **have** *a* ∉ *fst* ' *set ys* **by** *auto*
     then show *?case* **by** (*induct ys*) (*simp*, *auto simp*: *cs*)
    next
     case *Suc*
     then show *?case* **by** *simp*
    qed
    finally show *?thesis* **.**
   qed
  qed
  show *?case*
   unfolding *pair*
   apply (*simp add*: *IH*[*symmetric*])
   unfolding *id Ys-def*[*symmetric*]
   apply (*induct n*)
   apply (*auto simp*: *fold-mset-fun-left-comm*[*symmetric*])

done
qed
qed

end

context
begin

private lift-definition *single-alist-entry* :: $'a \Rightarrow 'b \Rightarrow ('a, 'b)$ *alist* **is** $\lambda a\ b.\ [(a, b)]$
by *auto*

lemma *image-mset-Bag* [*code*]:
image-mset f (*Bag ms*) =
DAList-Multiset.fold ($\lambda a\ n\ m$. *Bag* (*single-alist-entry* (*f a*) *n*) + *m*) {#} *ms*
unfolding *image-mset-def*
proof (*rule comp-fun-commute.DAList-Multiset-fold*, *unfold-locales*, (*auto simp*: *ac-simps*)[*1*])
fix *a n m*
show *Bag* (*single-alist-entry* (*f a*) *n*) + *m* = ((*op* + ∘ *single* ∘ *f*) *a* ^^ *n*) *m* (**is** *?l* = *?r*)
proof (*rule multiset-eqI*)
fix *x*
have *count ?r x* = (*if x* = *f a then n* + *count m x else count m x*)
by (*induct n*) *auto*
also have ... = *count ?l x*
by (*simp add*: *single-alist-entry.rep-eq*)
finally show *count ?l x* = *count ?r x* **..**
qed
qed

end

lemma *msetsum-Bag*[*code*]: *msetsum* (*Bag ms*) = *DAList-Multiset.fold* ($\lambda a\ n$. ((*op* + *a*) ^^ *n*)) *0 ms*
unfolding *msetsum.eq-fold*
apply (*rule comp-fun-commute.DAList-Multiset-fold*)
apply *unfold-locales*
apply (*auto simp*: *ac-simps*)
done

lemma *msetprod-Bag*[*code*]: *msetprod* (*Bag ms*) = *DAList-Multiset.fold* ($\lambda a\ n$. ((*op* ∗ *a*) ^^ *n*)) *1 ms*
unfolding *msetprod.eq-fold*
apply (*rule comp-fun-commute.DAList-Multiset-fold*)
apply *unfold-locales*

apply (*auto simp*: *ac-simps*)
done

lemma *size-fold*: *size A* = *fold-mset* (λ-. *Suc*) *0 A* (**is** - = *fold-mset ?f* - -)
proof –
interpret *comp-fun-commute ?f* **by** *standard auto*
show *?thesis* **by** (*induct A*) *auto*
qed

lemma *size-Bag*[*code*]: *size* (*Bag ms*) = *DAList-Multiset.fold* (λ*a n*. *op* + *n*) *0* *ms*
unfolding *size-fold*
proof (*rule comp-fun-commute.DAList-Multiset-fold*, *unfold-locales*, *simp*)
fix *a n x*
show *n* + *x* = (*Suc* ^^ *n*) *x*
by (*induct n*) *auto*
qed

lemma *set-mset-fold*: *set-mset A* = *fold-mset insert* {} *A* (**is** - = *fold-mset ?f* - -)
proof –
interpret *comp-fun-commute ?f* **by** *standard auto*
show *?thesis* **by** (*induct A*) *auto*
qed

lemma *set-mset-Bag*[*code*]:
set-mset (*Bag ms*) = *DAList-Multiset.fold* (λ*a n*. (*if n* = *0 then* (λ*m*. *m*) *else* *insert a*)) {} *ms*
unfolding *set-mset-fold*
proof (*rule comp-fun-commute.DAList-Multiset-fold*, *unfold-locales*, (*auto simp*: *ac-simps*)[*1*])
fix *a n x*
show (*if n* = *0 then* λ*m*. *m else insert a*) *x* = (*insert a* ^^ *n*) *x* (**is** *?l n* = *?r n*)
proof (*cases n*)
case *0*
then show *?thesis* **by** *simp*
next
case (*Suc m*)
then have *?l n* = *insert a x* **by** *simp*
moreover have *?r n* = *insert a x* **unfolding** *Suc* **by** (*induct m*) *auto*
ultimately show *?thesis* **by** *auto*
qed
qed

instantiation *multiset* :: (*exhaustive*) *exhaustive*
begin

definition *exhaustive-multiset* ::

('*a multiset* ⇒ (*bool* × *term list*) *option*) ⇒ *natural* ⇒ (*bool* × *term list*) *option*
where *exhaustive-multiset f i* = *Quickcheck-Exhaustive.exhaustive* (λ*xs*. *f* (*Bag xs*)) *i*

instance ..

end

end

108 Implementation of Red-Black Trees

theory *RBT-Impl*
imports *Main*
begin

For applications, you should use theory *RBT* which defines an abstract type of red-black tree obeying the invariant.

108.1 Datatype of RB trees

datatype *color* = *R* | *B*
datatype ('*a*, '*b*) *rbt* = *Empty* | *Branch color* ('*a*, '*b*) *rbt* '*a* '*b* ('*a*, '*b*) *rbt*

lemma *rbt-cases*:
obtains (*Empty*) *t* = *Empty*
| (*Red*) *l k v r* **where** *t* = *Branch R l k v r*
| (*Black*) *l k v r* **where** *t* = *Branch B l k v r*
proof (*cases t*)
case *Empty* **with** *that* **show** *thesis* **by** *blast*
next
case (*Branch c*) **with** *that* **show** *thesis* **by** (*cases c*) *blast*+
qed

108.2 Tree properties

108.2.1 Content of a tree

primrec *entries* :: ('*a*, '*b*) *rbt* ⇒ ('*a* × '*b*) *list*
where
entries Empty = []
| *entries* (*Branch* - *l k v r*) = *entries l* @ (*k*,*v*) # *entries r*

abbreviation (*input*) *entry-in-tree* :: '*a* ⇒ '*b* ⇒ ('*a*, '*b*) *rbt* ⇒ *bool*
where
entry-in-tree k v t ≡ (*k*, *v*) ∈ *set* (*entries t*)

definition *keys* :: ('*a*, '*b*) *rbt* ⇒ '*a list* **where**
keys t = *map fst* (*entries t*)

lemma *keys-simps* [*simp*, *code*]:
keys Empty = []
keys (*Branch c l k v r*) = *keys l* @ *k* # *keys r*
by (*simp-all add*: *keys-def*)

lemma *entry-in-tree-keys*:
assumes (k, v) ∈ *set* (*entries t*)
shows k ∈ *set* (*keys t*)
proof −
from *assms* **have** *fst* (k, v) ∈ *fst* ‘ *set* (*entries t*) **by** (*rule imageI*)
then show *?thesis* **by** (*simp add*: *keys-def*)
qed

lemma *keys-entries*:
k ∈ *set* (*keys t*) ⟷ (∃ v. (k, v) ∈ *set* (*entries t*))
by (*auto intro*: *entry-in-tree-keys*) (*auto simp add*: *keys-def*)

lemma *non-empty-rbt-keys*:
t ≠ *rbt.Empty* ⟹ *keys t* ≠ []
by (*cases t*) *simp-all*

108.2.2 Search tree properties

context *ord* **begin**

definition *rbt-less* :: ′*a* ⇒ (′*a*, ′*b*) *rbt* ⇒ *bool*
where
rbt-less-prop: *rbt-less k t* ⟷ (∀ *x*∈*set* (*keys t*). $x < k$)

abbreviation *rbt-less-symbol* (**infix** |≪ *50*)
where *t* |≪ *x* ≡ *rbt-less x t*

definition *rbt-greater* :: ′*a* ⇒ (′*a*, ′*b*) *rbt* ⇒ *bool* (**infix** ≪| *50*)
where
rbt-greater-prop: *rbt-greater k t* = (∀ *x*∈*set* (*keys t*). $k < x$)

lemma *rbt-less-simps* [*simp*]:
Empty |≪ *k* = *True*
Branch c lt kt v rt |≪ *k* ⟷ *kt* < *k* ∧ *lt* |≪ *k* ∧ *rt* |≪ *k*
by (*auto simp add*: *rbt-less-prop*)

lemma *rbt-greater-simps* [*simp*]:
k ≪| *Empty* = *True*
k ≪| (*Branch c lt kt v rt*) ⟷ *k* < *kt* ∧ *k* ≪| *lt* ∧ *k* ≪| *rt*
by (*auto simp add*: *rbt-greater-prop*)

lemmas *rbt-ord-props* = *rbt-less-prop rbt-greater-prop*

lemmas *rbt-greater-nit* = *rbt-greater-prop entry-in-tree-keys*
lemmas *rbt-less-nit* = *rbt-less-prop entry-in-tree-keys*

lemma (**in** *order*)
  shows *rbt-less-eq-trans*: $l \mid\ll u \Longrightarrow u \le v \Longrightarrow l \mid\ll v$
  and *rbt-less-trans*: $t \mid\ll x \Longrightarrow x < y \Longrightarrow t \mid\ll y$
  and *rbt-greater-eq-trans*: $u \le v \Longrightarrow v \ll\mid r \Longrightarrow u \ll\mid r$
  and *rbt-greater-trans*: $x < y \Longrightarrow y \ll\mid t \Longrightarrow x \ll\mid t$
  by (*auto simp*: *rbt-ord-props*)

primrec *rbt-sorted* :: ('*a*, '*b*) *rbt* ⇒ *bool*
where
  rbt-sorted Empty = *True*
| *rbt-sorted* (*Branch c l k v r*) = ($l \mid\ll k \wedge k \ll\mid r \wedge$ *rbt-sorted l* ∧ *rbt-sorted r*)

end

context *linorder* **begin**

lemma *rbt-sorted-entries*:
  rbt-sorted t ⟹ *List.sorted* (*map fst* (*entries t*))
by (*induct t*)
  (*force simp*: *sorted-append sorted-Cons rbt-ord-props*
    dest!: *entry-in-tree-keys*)+

lemma *distinct-entries*:
  rbt-sorted t ⟹ *distinct* (*map fst* (*entries t*))
by (*induct t*)
  (*force simp*: *sorted-append sorted-Cons rbt-ord-props*
    dest!: *entry-in-tree-keys*)+

lemma *distinct-keys*:
  rbt-sorted t ⟹ *distinct* (*keys t*)
  by (*simp add*: *distinct-entries keys-def*)

108.2.3 Tree lookup

primrec (**in** *ord*) *rbt-lookup* :: ('*a*, '*b*) *rbt* ⇒ '*a* ⇀ '*b*
where
  rbt-lookup Empty k = *None*
| *rbt-lookup* (*Branch - l x y r*) *k* =
   (*if k* < *x then rbt-lookup l k else if x* < *k then rbt-lookup r k else Some y*)

lemma *rbt-lookup-keys*: *rbt-sorted t* ⟹ *dom* (*rbt-lookup t*) = *set* (*keys t*)
  by (*induct t*) (*auto simp*: *dom-def rbt-greater-prop rbt-less-prop*)

lemma *dom-rbt-lookup-Branch*:
  rbt-sorted (*Branch c t1 k v t2*) ⟹
    dom (*rbt-lookup* (*Branch c t1 k v t2*))

= *Set.insert k* (*dom* (*rbt-lookup t1*) ∪ *dom* (*rbt-lookup t2*))
proof −
assume *rbt-sorted* (*Branch c t1 k v t2*)
then show *?thesis* **by** (*simp add*: *rbt-lookup-keys*)
qed

lemma *finite-dom-rbt-lookup* [*simp*, *intro*!]: *finite* (*dom* (*rbt-lookup t*))
proof (*induct t*)
case *Empty* **then show** *?case* **by** *simp*
next
case (*Branch color t1 a b t2*)
let *?A* = *Set.insert a* (*dom* (*rbt-lookup t1*) ∪ *dom* (*rbt-lookup t2*))
have *dom* (*rbt-lookup* (*Branch color t1 a b t2*)) ⊆ *?A* **by** (*auto split*: *if-split-asm*)
moreover from *Branch* **have** *finite* (*insert a* (*dom* (*rbt-lookup t1*) ∪ *dom* (*rbt-lookup t2*))) **by** *simp*
ultimately show *?case* **by** (*rule finite-subset*)
qed

end

context *ord* **begin**

lemma *rbt-lookup-rbt-less*[*simp*]: *t* |≪ *k* ⟹ *rbt-lookup t k* = *None*
by (*induct t*) *auto*

lemma *rbt-lookup-rbt-greater*[*simp*]: *k* ≪| *t* ⟹ *rbt-lookup t k* = *None*
by (*induct t*) *auto*

lemma *rbt-lookup-Empty*: *rbt-lookup Empty* = *empty*
by (*rule ext*) *simp*

end

context *linorder* **begin**

lemma *map-of-entries*:
rbt-sorted t ⟹ *map-of* (*entries t*) = *rbt-lookup t*
proof (*induct t*)
case *Empty* **thus** *?case* **by** (*simp add*: *rbt-lookup-Empty*)
next
case (*Branch c t1 k v t2*)
have *rbt-lookup* (*Branch c t1 k v t2*) = *rbt-lookup t2* ++ [*k*↦*v*] ++ *rbt-lookup t1*
proof (*rule ext*)
fix *x*
from *Branch* **have** *RBT-SORTED*: *rbt-sorted* (*Branch c t1 k v t2*) **by** *simp*
let *?thesis* = *rbt-lookup* (*Branch c t1 k v t2*) *x* = (*rbt-lookup t2* ++ [*k* ↦ *v*] ++ *rbt-lookup t1*) *x*

have *DOM-T1*: !!k'. $k' \in dom$ (*rbt-lookup t1*) $\Longrightarrow$ $k > k'$
proof –
fix k'
from *RBT-SORTED* **have** *t1* |≪ *k* **by** *simp*
with *rbt-less-prop* **have** $\forall k' \in set$ (*keys t1*). $k > k'$ **by** *auto*
moreover assume $k' \in dom$ (*rbt-lookup t1*)
ultimately show $k > k'$ **using** *rbt-lookup-keys RBT-SORTED* **by** *auto*
qed

have *DOM-T2*: !!k'. $k' \in dom$ (*rbt-lookup t2*) $\Longrightarrow$ $k < k'$
proof –
fix k'
from *RBT-SORTED* **have** *k* ≪| *t2* **by** *simp*
with *rbt-greater-prop* **have** $\forall k' \in set$ (*keys t2*). $k < k'$ **by** *auto*
moreover assume $k' \in dom$ (*rbt-lookup t2*)
ultimately show $k < k'$ **using** *rbt-lookup-keys RBT-SORTED* **by** *auto*
qed

{
assume *C*: $x < k$
hence *rbt-lookup* (*Branch c t1 k v t2*) x = *rbt-lookup t1 x* **by** *simp*
moreover from *C* **have** $x \notin dom$ [$k \mapsto v$] **by** *simp*
moreover have $x \notin$ *dom* (*rbt-lookup t2*)
proof
assume $x \in$ *dom* (*rbt-lookup t2*)
with *DOM-T2* **have** $k < x$ **by** *blast*
with *C* **show** *False* **by** *simp*
qed
ultimately have *?thesis* **by** (*simp add*: *map-add-upd-left map-add-dom-app-simps*)
} moreover {
assume [*simp*]: $x = k$
hence *rbt-lookup* (*Branch c t1 k v t2*) x = [$k \mapsto v$] x **by** *simp*
moreover have $x \notin$ *dom* (*rbt-lookup t1*)
proof
assume $x \in$ *dom* (*rbt-lookup t1*)
with *DOM-T1* **have** $k > x$ **by** *blast*
thus *False* **by** *simp*
qed
ultimately have *?thesis* **by** (*simp add*: *map-add-upd-left map-add-dom-app-simps*)
} moreover {
assume *C*: $x > k$
hence *rbt-lookup* (*Branch c t1 k v t2*) x = *rbt-lookup t2 x* **by** (*simp add*: *less-not-sym*[*of k x*])
moreover from *C* **have** $x \notin dom$ [$k \mapsto v$] **by** *simp*
moreover have $x \notin dom$ (*rbt-lookup t1*) **proof**
assume $x \in dom$ (*rbt-lookup t1*)
with *DOM-T1* **have** $k > x$ **by** *simp*
with *C* **show** *False* **by** *simp*
qed

ultimately have *?thesis* **by** (*simp add*: *map-add-upd-left map-add-dom-app-simps*)
} **ultimately show** *?thesis* **using** *less-linear* **by** *blast*
qed
also from *Branch*
have *rbt-lookup t2* ++ [*k* ↦ *v*] ++ *rbt-lookup t1* = *map-of* (*entries* (*Branch c t1 k v t2*)) **by** *simp*
finally show *?case* **by** *simp*
qed

lemma *rbt-lookup-in-tree*: *rbt-sorted t* ⟹ *rbt-lookup t k* = *Some v* ⟷ (*k*, *v*) ∈ *set* (*entries t*)
by (*simp add*: *map-of-entries* [*symmetric*] *distinct-entries*)

lemma *set-entries-inject*:
assumes *rbt-sorted*: *rbt-sorted t1 rbt-sorted t2*
shows *set* (*entries t1*) = *set* (*entries t2*) ⟷ *entries t1* = *entries t2*
proof −
from *rbt-sorted* **have** *distinct* (*map fst* (*entries t1*))
distinct (*map fst* (*entries t2*))
by (*auto intro*: *distinct-entries*)
with *rbt-sorted* **show** *?thesis*
by (*auto intro*: *map-sorted-distinct-set-unique rbt-sorted-entries simp add*: *distinct-map*)
qed

lemma *entries-eqI*:
assumes *rbt-sorted*: *rbt-sorted t1 rbt-sorted t2*
assumes *rbt-lookup*: *rbt-lookup t1* = *rbt-lookup t2*
shows *entries t1* = *entries t2*
proof −
from *rbt-sorted rbt-lookup* **have** *map-of* (*entries t1*) = *map-of* (*entries t2*)
by (*simp add*: *map-of-entries*)
with *rbt-sorted* **have** *set* (*entries t1*) = *set* (*entries t2*)
by (*simp add*: *map-of-inject-set distinct-entries*)
with *rbt-sorted* **show** *?thesis* **by** (*simp add*: *set-entries-inject*)
qed

lemma *entries-rbt-lookup*:
assumes *rbt-sorted t1 rbt-sorted t2*
shows *entries t1* = *entries t2* ⟷ *rbt-lookup t1* = *rbt-lookup t2*
using *assms* **by** (*auto intro*: *entries-eqI simp add*: *map-of-entries* [*symmetric*])

lemma *rbt-lookup-from-in-tree*:
assumes *rbt-sorted t1 rbt-sorted t2*
and ⋀*v*. (*k*, *v*) ∈ *set* (*entries t1*) ⟷ (*k*, *v*) ∈ *set* (*entries t2*)
shows *rbt-lookup t1 k* = *rbt-lookup t2 k*
proof −
from *assms* **have** *k* ∈ *dom* (*rbt-lookup t1*) ⟷ *k* ∈ *dom* (*rbt-lookup t2*)
by (*simp add*: *keys-entries rbt-lookup-keys*)

with *assms* **show** *?thesis* **by** (*auto simp add*: *rbt-lookup-in-tree* [*symmetric*])
qed

end

108.2.4 Red-black properties

primrec *color-of* :: ('*a*, '*b*) *rbt* ⇒ *color*
where
color-of Empty = *B*
| *color-of* (*Branch c - - - -*) = *c*

primrec *bheight* :: ('*a*,'*b*) *rbt* ⇒ *nat*
where
bheight Empty = *0*
| *bheight* (*Branch c lt k v rt*) = (*if c* = *B then Suc* (*bheight lt*) *else bheight lt*)

primrec *inv1* :: ('*a*, '*b*) *rbt* ⇒ *bool*
where
inv1 Empty = *True*
| *inv1* (*Branch c lt k v rt*) ⟷ *inv1 lt* ∧ *inv1 rt* ∧ (*c* = *B* ∨ *color-of lt* = *B* ∧ *color-of rt* = *B*)

primrec *inv1l* :: ('*a*, '*b*) *rbt* ⇒ *bool* — Weaker version
where
inv1l Empty = *True*
| *inv1l* (*Branch c l k v r*) = (*inv1 l* ∧ *inv1 r*)
lemma [*simp*]: *inv1 t* ⟹ *inv1l t* **by** (*cases t*) *simp+*

primrec *inv2* :: ('*a*, '*b*) *rbt* ⇒ *bool*
where
inv2 Empty = *True*
| *inv2* (*Branch c lt k v rt*) = (*inv2 lt* ∧ *inv2 rt* ∧ *bheight lt* = *bheight rt*)

context *ord* **begin**

definition *is-rbt* :: ('*a*, '*b*) *rbt* ⇒ *bool* **where**
is-rbt t ⟷ *inv1 t* ∧ *inv2 t* ∧ *color-of t* = *B* ∧ *rbt-sorted t*

lemma *is-rbt-rbt-sorted* [*simp*]:
is-rbt t ⟹ *rbt-sorted t* **by** (*simp add*: *is-rbt-def*)

theorem *Empty-is-rbt* [*simp*]:
is-rbt Empty **by** (*simp add*: *is-rbt-def*)

end

108.3 Insertion

The function definitions are based on the book by Okasaki.

fun
balance :: ('a,'b) *rbt* ⇒ 'a ⇒ 'b ⇒ ('a,'b) *rbt* ⇒ ('a,'b) *rbt*
where
balance (*Branch R a w x b*) *s t* (*Branch R c y z d*) = *Branch R* (*Branch B a w x b*) *s t* (*Branch B c y z d*) |
balance (*Branch R* (*Branch R a w x b*) *s t c*) *y z d* = *Branch R* (*Branch B a w x b*) *s t* (*Branch B c y z d*) |
balance (*Branch R a w x* (*Branch R b s t c*)) *y z d* = *Branch R* (*Branch B a w x b*) *s t* (*Branch B c y z d*) |
balance a w x (*Branch R b s t* (*Branch R c y z d*)) = *Branch R* (*Branch B a w x b*) *s t* (*Branch B c y z d*) |
balance a w x (*Branch R* (*Branch R b s t c*) *y z d*) = *Branch R* (*Branch B a w x b*) *s t* (*Branch B c y z d*) |
balance a s t b = *Branch B a s t b*

lemma *balance-inv1*: ⟦*inv1l l*; *inv1l r*⟧ ⟹ *inv1* (*balance l k v r*)
by (*induct l k v r rule*: *balance.induct*) *auto*

lemma *balance-bheight*: *bheight l* = *bheight r* ⟹ *bheight* (*balance l k v r*) = *Suc* (*bheight l*)
by (*induct l k v r rule*: *balance.induct*) *auto*

lemma *balance-inv2*:
assumes *inv2 l inv2 r bheight l* = *bheight r*
shows *inv2* (*balance l k v r*)
using *assms*
by (*induct l k v r rule*: *balance.induct*) *auto*

context *ord* **begin**

lemma *balance-rbt-greater*[*simp*]: (*v* ≪| *balance a k x b*) = (*v* ≪| *a* ∧ *v* ≪| *b* ∧ *v* < *k*)
by (*induct a k x b rule*: *balance.induct*) *auto*

lemma *balance-rbt-less*[*simp*]: (*balance a k x b* |≪ *v*) = (*a* |≪ *v* ∧ *b* |≪ *v* ∧ *k* < *v*)
by (*induct a k x b rule*: *balance.induct*) *auto*

end

lemma (**in** *linorder*) *balance-rbt-sorted*:
fixes *k* :: 'a
assumes *rbt-sorted l rbt-sorted r l* |≪ *k k* ≪| *r*
shows *rbt-sorted* (*balance l k v r*)
using *assms* **proof** (*induct l k v r rule*: *balance.induct*)
case (*2-2 a x w b y t c z s va vb vd vc*)
hence *y* < *z* ∧ *z* ≪| *Branch B va vb vd vc*
by (*auto simp add*: *rbt-ord-props*)
hence *y* ≪| (*Branch B va vb vd vc*) **by** (*blast dest*: *rbt-greater-trans*)
with *2-2* **show** *?case* **by** *simp*

next
  case (*3-2 va vb vd vc x w b y s c z*)
  from *3-2* **have** $x < y \wedge$ *Branch B va vb vd vc* $|\ll x$
    by *simp*
  hence *Branch B va vb vd vc* $|\ll y$ **by** (*blast dest*: *rbt-less-trans*)
  with *3-2* **show** *?case* **by** *simp*
next
  case (*3-3 x w b y s c z t va vb vd vc*)
  from *3-3* **have** $y < z \wedge z \ll|$ *Branch B va vb vd vc* **by** *simp*
  hence $y \ll|$ *Branch B va vb vd vc* **by** (*blast dest*: *rbt-greater-trans*)
  with *3-3* **show** *?case* **by** *simp*
next
  case (*3-4 vd ve vg vf x w b y s c z t va vb vii vc*)
  hence $x < y \wedge$ *Branch B vd ve vg vf* $|\ll x$ **by** *simp*
  hence *1*: *Branch B vd ve vg vf* $|\ll y$ **by** (*blast dest*: *rbt-less-trans*)
  from *3-4* **have** $y < z \wedge z \ll|$ *Branch B va vb vii vc* **by** *simp*
  hence $y \ll|$ *Branch B va vb vii vc* **by** (*blast dest*: *rbt-greater-trans*)
  with *1 3-4* **show** *?case* **by** *simp*
next
  case (*4-2 va vb vd vc x w b y s c z t dd*)
  hence $x < y \wedge$ *Branch B va vb vd vc* $|\ll x$ **by** *simp*
  hence *Branch B va vb vd vc* $|\ll y$ **by** (*blast dest*: *rbt-less-trans*)
  with *4-2* **show** *?case* **by** *simp*
next
  case (*5-2 x w b y s c z t va vb vd vc*)
  hence $y < z \wedge z \ll|$ *Branch B va vb vd vc* **by** *simp*
  hence $y \ll|$ *Branch B va vb vd vc* **by** (*blast dest*: *rbt-greater-trans*)
  with *5-2* **show** *?case* **by** *simp*
next
  case (*5-3 va vb vd vc x w b y s c z t*)
  hence $x < y \wedge$ *Branch B va vb vd vc* $|\ll x$ **by** *simp*
  hence *Branch B va vb vd vc* $|\ll y$ **by** (*blast dest*: *rbt-less-trans*)
  with *5-3* **show** *?case* **by** *simp*
next
  case (*5-4 va vb vg vc x w b y s c z t vd ve vii vf*)
  hence $x < y \wedge$ *Branch B va vb vg vc* $|\ll x$ **by** *simp*
  hence *1*: *Branch B va vb vg vc* $|\ll y$ **by** (*blast dest*: *rbt-less-trans*)
  from *5-4* **have** $y < z \wedge z \ll|$ *Branch B vd ve vii vf* **by** *simp*
  hence $y \ll|$ *Branch B vd ve vii vf* **by** (*blast dest*: *rbt-greater-trans*)
  with *1 5-4* **show** *?case* **by** *simp*
qed *simp+*

lemma *entries-balance* [*simp*]:
  entries (*balance l k v r*) = *entries l* @ (*k*, *v*) # *entries r*
  by (*induct l k v r rule*: *balance.induct*) *auto*

lemma *keys-balance* [*simp*]:
  keys (*balance l k v r*) = *keys l* @ *k* # *keys r*
  by (*simp add*: *keys-def*)

lemma *balance-in-tree*:
entry-in-tree k x (*balance l v y r*) $\longleftrightarrow$ *entry-in-tree k x l* $\vee$ $k = v \wedge x = y \vee$ *entry-in-tree k x r*
by (*auto simp add*: *keys-def*)

lemma (**in** *linorder*) *rbt-lookup-balance*[*simp*]:
fixes *k* :: *'a*
assumes *rbt-sorted l rbt-sorted r l* |≪ *k k* ≪| *r*
shows *rbt-lookup* (*balance l k v r*) *x* = *rbt-lookup* (*Branch B l k v r*) *x*
by (*rule rbt-lookup-from-in-tree*) (*auto simp*:*assms balance-in-tree balance-rbt-sorted*)

primrec *paint* :: *color* ⇒ (*'a*,*'b*) *rbt* ⇒ (*'a*,*'b*) *rbt*
where
paint c Empty = *Empty*
| *paint c* (*Branch - l k v r*) = *Branch c l k v r*

lemma *paint-inv1l*[*simp*]: *inv1l t* ⟹ *inv1l* (*paint c t*) **by** (*cases t*) *auto*
lemma *paint-inv1*[*simp*]: *inv1l t* ⟹ *inv1* (*paint B t*) **by** (*cases t*) *auto*
lemma *paint-inv2*[*simp*]: *inv2 t* ⟹ *inv2* (*paint c t*) **by** (*cases t*) *auto*
lemma *paint-color-of*[*simp*]: *color-of* (*paint B t*) = *B* **by** (*cases t*) *auto*
lemma *paint-in-tree*[*simp*]: *entry-in-tree k x* (*paint c t*) = *entry-in-tree k x t* **by** (*cases t*) *auto*

context *ord* **begin**

lemma *paint-rbt-sorted*[*simp*]: *rbt-sorted t* ⟹ *rbt-sorted* (*paint c t*) **by** (*cases t*) *auto*
lemma *paint-rbt-lookup*[*simp*]: *rbt-lookup* (*paint c t*) = *rbt-lookup t* **by** (*rule ext*) (*cases t*, *auto*)
lemma *paint-rbt-greater*[*simp*]: (*v* ≪| *paint c t*) = (*v* ≪| *t*) **by** (*cases t*) *auto*
lemma *paint-rbt-less*[*simp*]: (*paint c t* |≪ *v*) = (*t* |≪ *v*) **by** (*cases t*) *auto*

fun
rbt-ins :: (*'a* ⇒ *'b* ⇒ *'b* ⇒ *'b*) ⇒ *'a* ⇒ *'b* ⇒ (*'a*,*'b*) *rbt* ⇒ (*'a*,*'b*) *rbt*
where
rbt-ins f k v Empty = *Branch R Empty k v Empty* |
rbt-ins f k v (*Branch B l x y r*) = (*if k* < *x then balance* (*rbt-ins f k v l*) *x y r*
else if k > *x then balance l x y* (*rbt-ins f k v r*)
else Branch B l x (*f k y v*) *r*) |
rbt-ins f k v (*Branch R l x y r*) = (*if k* < *x then Branch R* (*rbt-ins f k v l*) *x y r*
else if k > *x then Branch R l x y* (*rbt-ins f k v r*)
else Branch R l x (*f k y v*) *r*)

lemma *ins-inv1-inv2*:
assumes *inv1 t inv2 t*
shows *inv2* (*rbt-ins f k x t*) *bheight* (*rbt-ins f k x t*) = *bheight t*
color-of t = *B* ⟹ *inv1* (*rbt-ins f k x t*) *inv1l* (*rbt-ins f k x t*)
using *assms*

by (*induct f k x t rule*: *rbt-ins.induct*) (*auto simp*: *balance-inv1 balance-inv2 balance-bheight*)

end

context *linorder* **begin**

lemma *ins-rbt-greater*[*simp*]: $(v \ll| \textit{rbt-ins } f\ (k :: 'a)\ x\ t) = (v \ll| t \wedge k > v)$
by (*induct f k x t rule*: *rbt-ins.induct*) *auto*
lemma *ins-rbt-less*[*simp*]: $(\textit{rbt-ins } f\ k\ x\ t\ |\ll v) = (t\ |\ll v \wedge k < v)$
by (*induct f k x t rule*: *rbt-ins.induct*) *auto*
lemma *ins-rbt-sorted*[*simp*]: *rbt-sorted t* $\Longrightarrow$ *rbt-sorted* (*rbt-ins f k x t*)
by (*induct f k x t rule*: *rbt-ins.induct*) (*auto simp*: *balance-rbt-sorted*)

lemma *keys-ins*: *set* (*keys* (*rbt-ins f k v t*)) = { *k* } ∪ *set* (*keys t*)
by (*induct f k v t rule*: *rbt-ins.induct*) *auto*

lemma *rbt-lookup-ins*:
fixes *k* :: *'a*
assumes *rbt-sorted t*
shows *rbt-lookup* (*rbt-ins f k v t*) *x* = ((*rbt-lookup t*)(*k* |−> *case rbt-lookup t k of None* ⇒ *v*
| *Some w* ⇒ *f k w v*)) *x*
using *assms* **by** (*induct f k v t rule*: *rbt-ins.induct*) *auto*

end

context *ord* **begin**

definition *rbt-insert-with-key* :: (*'a* ⇒ *'b* ⇒ *'b* ⇒ *'b*) ⇒ *'a* ⇒ *'b* ⇒ (*'a*,*'b*) *rbt* ⇒ (*'a*,*'b*) *rbt*
where *rbt-insert-with-key f k v t* = *paint B* (*rbt-ins f k v t*)

definition *rbt-insertw-def*: *rbt-insert-with f* = *rbt-insert-with-key* (λ-. *f*)

definition *rbt-insert* :: *'a* ⇒ *'b* ⇒ (*'a*, *'b*) *rbt* ⇒ (*'a*, *'b*) *rbt* **where**
rbt-insert = *rbt-insert-with-key* (λ- - *nv*. *nv*)

end

context *linorder* **begin**

lemma *rbt-insertwk-rbt-sorted*: *rbt-sorted t* $\Longrightarrow$ *rbt-sorted* (*rbt-insert-with-key f* (*k* :: *'a*) *x t*)
by (*auto simp*: *rbt-insert-with-key-def*)

theorem *rbt-insertwk-is-rbt*:
assumes *inv*: *is-rbt t*
shows *is-rbt* (*rbt-insert-with-key f k x t*)

using *assms*
unfolding *rbt-insert-with-key-def is-rbt-def*
by (*auto simp*: *ins-inv1-inv2*)

lemma *rbt-lookup-rbt-insertwk*:
assumes *rbt-sorted t*
shows *rbt-lookup* (*rbt-insert-with-key f k v t*) *x* = ((*rbt-lookup t*)(*k* |−> *case rbt-lookup t k of None* ⇒ *v*
| *Some w* ⇒ *f k w v*)) *x*
unfolding *rbt-insert-with-key-def* **using** *assms*
by (*simp add*:*rbt-lookup-ins*)

lemma *rbt-insertw-rbt-sorted*: *rbt-sorted t* ⟹ *rbt-sorted* (*rbt-insert-with f k v t*)
by (*simp add*: *rbt-insertwk-rbt-sorted rbt-insertw-def*)
theorem *rbt-insertw-is-rbt*: *is-rbt t* ⟹ *is-rbt* (*rbt-insert-with f k v t*)
by (*simp add*: *rbt-insertwk-is-rbt rbt-insertw-def*)

lemma *rbt-lookup-rbt-insertw*:
assumes *is-rbt t*
shows *rbt-lookup* (*rbt-insert-with f k v t*) = (*rbt-lookup t*)(*k* ↦ (*if k*:*dom* (*rbt-lookup t*) *then f* (*the* (*rbt-lookup t k*)) *v else v*))
using *assms*
unfolding *rbt-insertw-def*
by (*rule-tac ext*) (*cases rbt-lookup t k*, *auto simp*:*rbt-lookup-rbt-insertwk dom-def*)

lemma *rbt-insert-rbt-sorted*: *rbt-sorted t* ⟹ *rbt-sorted* (*rbt-insert k v t*)
by (*simp add*: *rbt-insertwk-rbt-sorted rbt-insert-def*)
theorem *rbt-insert-is-rbt* [*simp*]: *is-rbt t* ⟹ *is-rbt* (*rbt-insert k v t*)
by (*simp add*: *rbt-insertwk-is-rbt rbt-insert-def*)

lemma *rbt-lookup-rbt-insert*:
assumes *is-rbt t*
shows *rbt-lookup* (*rbt-insert k v t*) = (*rbt-lookup t*)(*k*↦*v*)
unfolding *rbt-insert-def*
using *assms*
by (*rule-tac ext*) (*simp add*: *rbt-lookup-rbt-insertwk split*:*option.split*)

end

108.4 Deletion

lemma *bheight-paintR'*[*simp*]: *color-of t* = *B* ⟹ *bheight* (*paint R t*) = *bheight t* − *1*
by (*cases t rule*: *rbt-cases*) *auto*

The function definitions are based on the Haskell code by Stefan Kahrs at http://www.cs.ukc.ac.uk/people/staff/smk/redblack/rb.html .

fun
balance-left :: ('*a*,'*b*) *rbt* ⇒ '*a* ⇒ '*b* ⇒ ('*a*,'*b*) *rbt* ⇒ ('*a*,'*b*) *rbt*

where
balance-left (*Branch R a k x b*) *s y c* = *Branch R* (*Branch B a k x b*) *s y c* |
balance-left bl k x (*Branch B a s y b*) = *balance bl k x* (*Branch R a s y b*) |
balance-left bl k x (*Branch R* (*Branch B a s y b*) *t z c*) = *Branch R* (*Branch B bl k x a*) *s y* (*balance b t z* (*paint R c*)) |
balance-left t k x s = *Empty*

lemma *balance-left-inv2-with-inv1*:
assumes *inv2 lt inv2 rt bheight lt* + *1* = *bheight rt inv1 rt*
shows *bheight* (*balance-left lt k v rt*) = *bheight lt* + *1*
and *inv2* (*balance-left lt k v rt*)
using *assms*
by (*induct lt k v rt rule*: *balance-left.induct*) (*auto simp*: *balance-inv2 balance-bheight*)

lemma *balance-left-inv2-app*:
assumes *inv2 lt inv2 rt bheight lt* + *1* = *bheight rt color-of rt* = *B*
shows *inv2* (*balance-left lt k v rt*)
bheight (*balance-left lt k v rt*) = *bheight rt*
using *assms*
by (*induct lt k v rt rule*: *balance-left.induct*) (*auto simp add*: *balance-inv2 balance-bheight*)+

lemma *balance-left-inv1*: ⟦*inv1l a*; *inv1 b*; *color-of b* = *B*⟧ ⟹ *inv1* (*balance-left a k x b*)
by (*induct a k x b rule*: *balance-left.induct*) (*simp add*: *balance-inv1*)+

lemma *balance-left-inv1l*: ⟦ *inv1l lt*; *inv1 rt* ⟧ ⟹ *inv1l* (*balance-left lt k x rt*)
by (*induct lt k x rt rule*: *balance-left.induct*) (*auto simp*: *balance-inv1*)

lemma (**in** *linorder*) *balance-left-rbt-sorted*:
⟦ *rbt-sorted l*; *rbt-sorted r*; *rbt-less k l*; *k* ≪| *r* ⟧ ⟹ *rbt-sorted* (*balance-left l k v r*)
apply (*induct l k v r rule*: *balance-left.induct*)
apply (*auto simp*: *balance-rbt-sorted*)
apply (*unfold rbt-greater-prop rbt-less-prop*)
by *force*+

context *order* **begin**

lemma *balance-left-rbt-greater*:
fixes *k* :: *'a*
assumes *k* ≪| *a k* ≪| *b k* < *x*
shows *k* ≪| *balance-left a x t b*
using *assms*
by (*induct a x t b rule*: *balance-left.induct*) *auto*

lemma *balance-left-rbt-less*:
fixes *k* :: *'a*
assumes *a* |≪ *k b* |≪ *k x* < *k*

shows *balance-left a x t b* $|\ll$ *k*
using *assms*
by (*induct a x t b rule*: *balance-left.induct*) *auto*

end

lemma *balance-left-in-tree*:
assumes *inv1l l inv1 r bheight l* + *1* = *bheight r*
shows *entry-in-tree k v* (*balance-left l a b r*) = (*entry-in-tree k v l* ∨ *k* = *a* ∧ *v* = *b* ∨ *entry-in-tree k v r*)
using *assms*
by (*induct l k v r rule*: *balance-left.induct*) (*auto simp*: *balance-in-tree*)

fun
balance-right :: ('*a*,'*b*) *rbt* ⇒ '*a* ⇒ '*b* ⇒ ('*a*,'*b*) *rbt* ⇒ ('*a*,'*b*) *rbt*
where
balance-right a k x (*Branch R b s y c*) = *Branch R a k x* (*Branch B b s y c*) |
balance-right (*Branch B a k x b*) *s y bl* = *balance* (*Branch R a k x b*) *s y bl* |
balance-right (*Branch R a k x* (*Branch B b s y c*)) *t z bl* = *Branch R* (*balance* (*paint R a*) *k x b*) *s y* (*Branch B c t z bl*) |
balance-right t k x s = *Empty*

lemma *balance-right-inv2-with-inv1*:
assumes *inv2 lt inv2 rt bheight lt* = *bheight rt* + *1 inv1 lt*
shows *inv2* (*balance-right lt k v rt*) ∧ *bheight* (*balance-right lt k v rt*) = *bheight lt*
using *assms*
by (*induct lt k v rt rule*: *balance-right.induct*) (*auto simp*: *balance-inv2 balance-bheight*)

lemma *balance-right-inv1*: ⟦*inv1 a*; *inv1l b*; *color-of a* = *B*⟧ ⟹ *inv1* (*balance-right a k x b*)
by (*induct a k x b rule*: *balance-right.induct*) (*simp add*: *balance-inv1*)+

lemma *balance-right-inv1l*: ⟦ *inv1 lt*; *inv1l rt* ⟧ ⟹*inv1l* (*balance-right lt k x rt*)
by (*induct lt k x rt rule*: *balance-right.induct*) (*auto simp*: *balance-inv1*)

lemma (**in** *linorder*) *balance-right-rbt-sorted*:
⟦ *rbt-sorted l*; *rbt-sorted r*; *rbt-less k l*; *k* $\ll|$ *r* ⟧ ⟹ *rbt-sorted* (*balance-right l k v r*)
apply (*induct l k v r rule*: *balance-right.induct*)
apply (*auto simp*:*balance-rbt-sorted*)
apply (*unfold rbt-less-prop rbt-greater-prop*)
by *force*+

context *order* **begin**

lemma *balance-right-rbt-greater*:
fixes *k* :: '*a*
assumes *k* $\ll|$ *a k* $\ll|$ *b k* < *x*

shows *k* ≪| *balance-right a x t b*
using *assms* **by** (*induct a x t b rule*: *balance-right.induct*) *auto*

lemma *balance-right-rbt-less*:
fixes *k* :: *'a*
assumes *a* |≪ *k b* |≪ *k x* < *k*
shows *balance-right a x t b* |≪ *k*
using *assms* **by** (*induct a x t b rule*: *balance-right.induct*) *auto*

end

lemma *balance-right-in-tree*:
assumes *inv1 l inv1l r bheight l* = *bheight r* + *1 inv2 l inv2 r*
shows *entry-in-tree x y* (*balance-right l k v r*) = (*entry-in-tree x y l* ∨ *x* = *k* ∧ *y* = *v* ∨ *entry-in-tree x y r*)
using *assms* **by** (*induct l k v r rule*: *balance-right.induct*) (*auto simp*: *balance-in-tree*)

fun
combine :: (*'a*,*'b*) *rbt* ⇒ (*'a*,*'b*) *rbt* ⇒ (*'a*,*'b*) *rbt*
where
combine Empty x = *x*
| *combine x Empty* = *x*
| *combine* (*Branch R a k x b*) (*Branch R c s y d*) = (*case* (*combine b c*) *of*
Branch R b2 t z c2 ⇒ (*Branch R* (*Branch R a k x b2*) *t z* (*Branch R c2 s y d*)) |
bc ⇒ *Branch R a k x* (*Branch R bc s y d*))
| *combine* (*Branch B a k x b*) (*Branch B c s y d*) = (*case* (*combine b c*) *of*
Branch R b2 t z c2 ⇒ *Branch R* (*Branch B a k x b2*) *t z* (*Branch B c2 s y d*) |
bc ⇒ *balance-left a k x* (*Branch B bc s y d*))
| *combine a* (*Branch R b k x c*) = *Branch R* (*combine a b*) *k x c*
| *combine* (*Branch R a k x b*) *c* = *Branch R a k x* (*combine b c*)

lemma *combine-inv2*:
assumes *inv2 lt inv2 rt bheight lt* = *bheight rt*
shows *bheight* (*combine lt rt*) = *bheight lt inv2* (*combine lt rt*)
using *assms*
by (*induct lt rt rule*: *combine.induct*)
(*auto simp*: *balance-left-inv2-app split*: *rbt.splits color.splits*)

lemma *combine-inv1*:
assumes *inv1 lt inv1 rt*
shows *color-of lt* = *B* ⟹ *color-of rt* = *B* ⟹ *inv1* (*combine lt rt*)
inv1l (*combine lt rt*)
using *assms*
by (*induct lt rt rule*: *combine.induct*)
(*auto simp*: *balance-left-inv1 split*: *rbt.splits color.splits*)

context *linorder* **begin**

lemma *combine-rbt-greater*[*simp*]:
  fixes k :: $'a$
  assumes $k \ll| l$ $k \ll| r$
  shows $k \ll|$ *combine* l r
using *assms*
by (*induct l r rule*: *combine.induct*)
  (*auto simp*: *balance-left-rbt-greater split*:*rbt.splits color.splits*)

lemma *combine-rbt-less*[*simp*]:
  fixes k :: $'a$
  assumes $l |\ll k$ $r |\ll k$
  shows *combine* l r $|\ll k$
using *assms*
by (*induct l r rule*: *combine.induct*)
  (*auto simp*: *balance-left-rbt-less split*:*rbt.splits color.splits*)

lemma *combine-rbt-sorted*:
  fixes k :: $'a$
  assumes *rbt-sorted* l *rbt-sorted* r $l |\ll k$ $k \ll| r$
  shows *rbt-sorted* (*combine* l r)
using *assms* **proof** (*induct l r rule*: *combine.induct*)
  case (*3 a x v b c y w d*)
  hence *ineqs*: $a |\ll x$ $x \ll| b$ $b |\ll k$ $k \ll| c$ $c |\ll y$ $y \ll| d$
    by *auto*
  with *3*
  show *?case*
    by (*cases combine b c rule*: *rbt-cases*)
      (*auto*, (*metis combine-rbt-greater combine-rbt-less ineqs ineqs rbt-less-simps*(*2*)
rbt-greater-simps(*2*) *rbt-greater-trans rbt-less-trans*)+)
next
  case (*4 a x v b c y w d*)
  hence $x < k \wedge$ *rbt-greater* k c **by** *simp*
  hence *rbt-greater* x c **by** (*blast dest*: *rbt-greater-trans*)
  with *4* **have** *2*: *rbt-greater* x (*combine* b c) **by** (*simp add*: *combine-rbt-greater*)
  from *4* **have** $k < y \wedge$ *rbt-less* k b **by** *simp*
  hence *rbt-less* y b **by** (*blast dest*: *rbt-less-trans*)
  with *4* **have** *3*: *rbt-less* y (*combine* b c) **by** (*simp add*: *combine-rbt-less*)
  show *?case*
  proof (*cases combine b c rule*: *rbt-cases*)
    case *Empty*
    from *4* **have** $x < y \wedge$ *rbt-greater* y d **by** *auto*
    hence *rbt-greater* x d **by** (*blast dest*: *rbt-greater-trans*)
    with *4 Empty* **have** *rbt-sorted* a **and** *rbt-sorted* (*Branch B Empty y w d*)
      and *rbt-less* x a **and** *rbt-greater* x (*Branch B Empty y w d*) **by** *auto*
    with *Empty* **show** *?thesis* **by** (*simp add*: *balance-left-rbt-sorted*)
  next
    case (*Red lta va ka rta*)
    with *2 4* **have** $x < va \wedge$ *rbt-less* x a **by** *simp*

hence *5*: *rbt-less va a* **by** (*blast dest*: *rbt-less-trans*)
from *Red 3 4* **have** *va* < *y* ∧ *rbt-greater y d* **by** *simp*
hence *rbt-greater va d* **by** (*blast dest*: *rbt-greater-trans*)
with *Red 2 3 4 5* **show** *?thesis* **by** *simp*
next
case (*Black lta va ka rta*)
from *4* **have** *x* < *y* ∧ *rbt-greater y d* **by** *auto*
hence *rbt-greater x d* **by** (*blast dest*: *rbt-greater-trans*)
with *Black 2 3 4* **have** *rbt-sorted a* **and** *rbt-sorted* (*Branch B* (*combine b c*) *y w d*)
and *rbt-less x a* **and** *rbt-greater x* (*Branch B* (*combine b c*) *y w d*) **by** *auto*
with *Black* **show** *?thesis* **by** (*simp add*: *balance-left-rbt-sorted*)
qed
next
case (*5 va vb vd vc b x w c*)
hence *k* < *x* ∧ *rbt-less k* (*Branch B va vb vd vc*) **by** *simp*
hence *rbt-less x* (*Branch B va vb vd vc*) **by** (*blast dest*: *rbt-less-trans*)
with *5* **show** *?case* **by** (*simp add*: *combine-rbt-less*)
next
case (*6 a x v b va vb vd vc*)
hence *x* < *k* ∧ *rbt-greater k* (*Branch B va vb vd vc*) **by** *simp*
hence *rbt-greater x* (*Branch B va vb vd vc*) **by** (*blast dest*: *rbt-greater-trans*)
with *6* **show** *?case* **by** (*simp add*: *combine-rbt-greater*)
qed *simp+*

end

lemma *combine-in-tree*:
assumes *inv2 l inv2 r bheight l* = *bheight r inv1 l inv1 r*
shows *entry-in-tree k v* (*combine l r*) = (*entry-in-tree k v l* ∨ *entry-in-tree k v r*)
using *assms*
proof (*induct l r rule*: *combine.induct*)
case (*4 - - - b c*)
hence *a*: *bheight* (*combine b c*) = *bheight b* **by** (*simp add*: *combine-inv2*)
from *4* **have** *b*: *inv1l* (*combine b c*) **by** (*simp add*: *combine-inv1*)

show *?case*
proof (*cases combine b c rule*: *rbt-cases*)
case *Empty*
with *4 a* **show** *?thesis* **by** (*auto simp*: *balance-left-in-tree*)
next
case (*Red lta ka va rta*)
with *4* **show** *?thesis* **by** *auto*
next
case (*Black lta ka va rta*)
with *a b 4* **show** *?thesis* **by** (*auto simp*: *balance-left-in-tree*)
qed
qed (*auto split*: *rbt.splits color.splits*)

context *ord* **begin**

fun
rbt-del-from-left :: $'a \Rightarrow ('a,'b)$ *rbt* $\Rightarrow 'a \Rightarrow 'b \Rightarrow ('a,'b)$ *rbt* $\Rightarrow ('a,'b)$ *rbt* **and**
rbt-del-from-right :: $'a \Rightarrow ('a,'b)$ *rbt* $\Rightarrow 'a \Rightarrow 'b \Rightarrow ('a,'b)$ *rbt* $\Rightarrow ('a,'b)$ *rbt* **and**
rbt-del :: $'a \Rightarrow ('a,'b)$ *rbt* $\Rightarrow ('a,'b)$ *rbt*
where
rbt-del x Empty = Empty |
rbt-del x (Branch c a y s b) =
(if x < y then rbt-del-from-left x a y s b
else (if x > y then rbt-del-from-right x a y s b else combine a b)) |
rbt-del-from-left x (Branch B lt z v rt) y s b = balance-left (rbt-del x (Branch B lt z v rt)) y s b |
rbt-del-from-left x a y s b = Branch R (rbt-del x a) y s b |
rbt-del-from-right x a y s (Branch B lt z v rt) = balance-right a y s (rbt-del x (Branch B lt z v rt)) |
rbt-del-from-right x a y s b = Branch R a y s (rbt-del x b)

end

context *linorder* **begin**

lemma
assumes *inv2 lt inv1 lt*
shows
⟦*inv2 rt*; *bheight lt = bheight rt*; *inv1 rt*⟧ $\Longrightarrow$
inv2 (rbt-del-from-left x lt k v rt) $\wedge$
bheight (rbt-del-from-left x lt k v rt) = bheight lt $\wedge$
(color-of lt = B $\wedge$ *color-of rt = B* $\wedge$ *inv1 (rbt-del-from-left x lt k v rt)* $\vee$
(color-of lt $\neq$ *B* $\vee$ *color-of rt* $\neq$ *B)* $\wedge$ *inv1l (rbt-del-from-left x lt k v rt))*
and ⟦*inv2 rt*; *bheight lt = bheight rt*; *inv1 rt*⟧ $\Longrightarrow$
inv2 (rbt-del-from-right x lt k v rt) $\wedge$
bheight (rbt-del-from-right x lt k v rt) = bheight lt $\wedge$
(color-of lt = B $\wedge$ *color-of rt = B* $\wedge$ *inv1 (rbt-del-from-right x lt k v rt)* $\vee$
(color-of lt $\neq$ *B* $\vee$ *color-of rt* $\neq$ *B)* $\wedge$ *inv1l (rbt-del-from-right x lt k v rt))*
and *rbt-del-inv1-inv2*: *inv2 (rbt-del x lt)* $\wedge$ *(color-of lt = R* $\wedge$ *bheight (rbt-del x lt) = bheight lt* $\wedge$ *inv1 (rbt-del x lt)*
$\vee$ *color-of lt = B* $\wedge$ *bheight (rbt-del x lt) = bheight lt − 1* $\wedge$ *inv1l (rbt-del x lt))*
using *assms*
proof *(induct x lt k v rt* **and** *x lt k v rt* **and** *x lt rule: rbt-del-from-left-rbt-del-from-right-rbt-del.induct)*
case *(2 y c - y')*
have $y = y' \vee y < y' \vee y > y'$ **by** *auto*
thus *?case* **proof** *(elim disjE)*
assume $y = y'$
with *2* **show** *?thesis* **by** *(cases c) (simp add: combine-inv2 combine-inv1)+*
next
assume $y < y'$
with *2* **show** *?thesis* **by** *(cases c) auto*

next
assume *y′ < y*
with *2* **show** *?thesis* **by** (*cases c*) *auto*
qed
next
case (*3 y lt z v rta y′ ss bb*)
thus *?case* **by** (*cases color-of* (*Branch B lt z v rta*) = *B* ∧ *color-of bb* = *B*) (*simp add*: *balance-left-inv2-with-inv1 balance-left-inv1 balance-left-inv1l*)+
next
case (*5 y a y′ ss lt z v rta*)
thus *?case* **by** (*cases color-of a* = *B* ∧ *color-of* (*Branch B lt z v rta*) = *B*) (*simp add*: *balance-right-inv2-with-inv1 balance-right-inv1 balance-right-inv1l*)+
next
case (*6-1 y a y′ ss*) **thus** *?case* **by** (*cases color-of a* = *B* ∧ *color-of Empty* = *B*) *simp*+
qed *auto*

lemma
rbt-del-from-left-rbt-less: ⟦ *lt* |≪ *v*; *rt* |≪ *v*; *k* < *v*⟧ ⟹ *rbt-del-from-left x lt k y rt* |≪ *v*
and *rbt-del-from-right-rbt-less*: ⟦*lt* |≪ *v*; *rt* |≪ *v*; *k* < *v*⟧ ⟹ *rbt-del-from-right x lt k y rt* |≪ *v*
and *rbt-del-rbt-less*: *lt* |≪ *v* ⟹ *rbt-del x lt* |≪ *v*
by (*induct x lt k y rt* **and** *x lt k y rt* **and** *x lt rule*: *rbt-del-from-left-rbt-del-from-right-rbt-del.induct*)

(*auto simp*: *balance-left-rbt-less balance-right-rbt-less*)

lemma *rbt-del-from-left-rbt-greater*: ⟦*v* ≪| *lt*; *v* ≪| *rt*; *k* > *v*⟧ ⟹ *v* ≪| *rbt-del-from-left x lt k y rt*
and *rbt-del-from-right-rbt-greater*: ⟦*v* ≪| *lt*; *v* ≪| *rt*; *k* > *v*⟧ ⟹ *v* ≪| *rbt-del-from-right x lt k y rt*
and *rbt-del-rbt-greater*: *v* ≪| *lt* ⟹ *v* ≪| *rbt-del x lt*
by (*induct x lt k y rt* **and** *x lt k y rt* **and** *x lt rule*: *rbt-del-from-left-rbt-del-from-right-rbt-del.induct*)
(*auto simp*: *balance-left-rbt-greater balance-right-rbt-greater*)

lemma ⟦*rbt-sorted lt*; *rbt-sorted rt*; *lt* |≪ *k*; *k* ≪| *rt*⟧ ⟹ *rbt-sorted* (*rbt-del-from-left x lt k y rt*)
and ⟦*rbt-sorted lt*; *rbt-sorted rt*; *lt* |≪ *k*; *k* ≪| *rt*⟧ ⟹ *rbt-sorted* (*rbt-del-from-right x lt k y rt*)
and *rbt-del-rbt-sorted*: *rbt-sorted lt* ⟹ *rbt-sorted* (*rbt-del x lt*)
proof (*induct x lt k y rt* **and** *x lt k y rt* **and** *x lt rule*: *rbt-del-from-left-rbt-del-from-right-rbt-del.induct*)
case (*3 x lta zz v rta yy ss bb*)
from *3* **have** *Branch B lta zz v rta* |≪ *yy* **by** *simp*
hence *rbt-del x* (*Branch B lta zz v rta*) |≪ *yy* **by** (*rule rbt-del-rbt-less*)
with *3* **show** *?case* **by** (*simp add*: *balance-left-rbt-sorted*)
next
case (*4-2 x vaa vbb vdd vc yy ss bb*)
hence *Branch R vaa vbb vdd vc* |≪ *yy* **by** *simp*
hence *rbt-del x* (*Branch R vaa vbb vdd vc*) |≪ *yy* **by** (*rule rbt-del-rbt-less*)

with *4-2* **show** *?case* **by** *simp*
next
case (*5 x aa yy ss lta zz v rta*)
hence *yy* ≪| *Branch B lta zz v rta* **by** *simp*
hence *yy* ≪| *rbt-del x* (*Branch B lta zz v rta*) **by** (*rule rbt-del-rbt-greater*)
with *5* **show** *?case* **by** (*simp add*: *balance-right-rbt-sorted*)
next
case (*6-2 x aa yy ss vaa vbb vdd vc*)
hence *yy* ≪| *Branch R vaa vbb vdd vc* **by** *simp*
hence *yy* ≪| *rbt-del x* (*Branch R vaa vbb vdd vc*) **by** (*rule rbt-del-rbt-greater*)
with *6-2* **show** *?case* **by** *simp*
qed (*auto simp*: *combine-rbt-sorted*)

lemma ⟦*rbt-sorted lt*; *rbt-sorted rt*; *lt* |≪ *kt*; *kt* ≪| *rt*; *inv1 lt*; *inv1 rt*; *inv2 lt*; *inv2 rt*; *bheight lt* = *bheight rt*; *x* < *kt*⟧ ⟹ *entry-in-tree k v* (*rbt-del-from-left x lt kt y rt*) = (*False* ∨ (*x* ≠ *k* ∧ *entry-in-tree k v* (*Branch c lt kt y rt*)))
and ⟦*rbt-sorted lt*; *rbt-sorted rt*; *lt* |≪ *kt*; *kt* ≪| *rt*; *inv1 lt*; *inv1 rt*; *inv2 lt*; *inv2 rt*; *bheight lt* = *bheight rt*; *x* > *kt*⟧ ⟹ *entry-in-tree k v* (*rbt-del-from-right x lt kt y rt*) = (*False* ∨ (*x* ≠ *k* ∧ *entry-in-tree k v* (*Branch c lt kt y rt*)))
and *rbt-del-in-tree*: ⟦*rbt-sorted t*; *inv1 t*; *inv2 t*⟧ ⟹ *entry-in-tree k v* (*rbt-del x t*) = (*False* ∨ (*x* ≠ *k* ∧ *entry-in-tree k v t*))
proof (*induct x lt kt y rt* **and** *x lt kt y rt* **and** *x t rule*: *rbt-del-from-left-rbt-del-from-right-rbt-del.induct*)
case (*2 xx c aa yy ss bb*)
have *xx* = *yy* ∨ *xx* < *yy* ∨ *xx* > *yy* **by** *auto*
from *this 2* **show** *?case* **proof** (*elim disjE*)
assume *xx* = *yy*
with *2* **show** *?thesis* **proof** (*cases xx* = *k*)
case *True*
from *2* ‹*xx* = *yy*› ‹*xx* = *k*› **have** *rbt-sorted* (*Branch c aa yy ss bb*) ∧ *k* = *yy* **by** *simp*
hence ¬ *entry-in-tree k v aa* ¬ *entry-in-tree k v bb* **by** (*auto simp*: *rbt-less-nit rbt-greater-prop*)
with ‹*xx* = *yy*› *2* ‹*xx* = *k*› **show** *?thesis* **by** (*simp add*: *combine-in-tree*)
qed (*simp add*: *combine-in-tree*)
qed *simp*+
next
case (*3 xx lta zz vv rta yy ss bb*)
def *mt*[*simp*]: *mt* == *Branch B lta zz vv rta*
from *3* **have** *inv2 mt* ∧ *inv1 mt* **by** *simp*
hence *inv2* (*rbt-del xx mt*) ∧ (*color-of mt* = *R* ∧ *bheight* (*rbt-del xx mt*) = *bheight mt* ∧ *inv1* (*rbt-del xx mt*) ∨ *color-of mt* = *B* ∧ *bheight* (*rbt-del xx mt*) = *bheight mt* − *1* ∧ *inv1l* (*rbt-del xx mt*)) **by** (*blast dest*: *rbt-del-inv1-inv2*)
with *3* **have** *4*: *entry-in-tree k v* (*rbt-del-from-left xx mt yy ss bb*) = (*False* ∨ *xx* ≠ *k* ∧ *entry-in-tree k v mt* ∨ (*k* = *yy* ∧ *v* = *ss*) ∨ *entry-in-tree k v bb*) **by** (*simp add*: *balance-left-in-tree*)
thus *?case* **proof** (*cases xx* = *k*)
case *True*
from *3 True* **have** *yy* ≪| *bb* ∧ *yy* > *k* **by** *simp*
hence *k* ≪| *bb* **by** (*blast dest*: *rbt-greater-trans*)

with *3 4 True* **show** *?thesis* **by** (*auto simp*: *rbt-greater-nit*)
qed *auto*
next
case (*4-1 xx yy ss bb*)
show *?case* **proof** (*cases xx* = *k*)
case *True*
with *4-1* **have** *yy* ≪| *bb* ∧ *k* < *yy* **by** *simp*
hence *k* ≪| *bb* **by** (*blast dest*: *rbt-greater-trans*)
with *4-1* ‹*xx* = *k*›
have *entry-in-tree k v* (*Branch R Empty yy ss bb*) = *entry-in-tree k v Empty* **by** (*auto simp*: *rbt-greater-nit*)
thus *?thesis* **by** *auto*
qed *simp+*
next
case (*4-2 xx vaa vbb vdd vc yy ss bb*)
thus *?case* **proof** (*cases xx* = *k*)
case *True*
with *4-2* **have** *k* < *yy* ∧ *yy* ≪| *bb* **by** *simp*
hence *k* ≪| *bb* **by** (*blast dest*: *rbt-greater-trans*)
with *True 4-2* **show** *?thesis* **by** (*auto simp*: *rbt-greater-nit*)
qed *auto*
next
case (*5 xx aa yy ss lta zz vv rta*)
def *mt*[*simp*]: *mt* == *Branch B lta zz vv rta*
from *5* **have** *inv2 mt* ∧ *inv1 mt* **by** *simp*
hence *inv2* (*rbt-del xx mt*) ∧ (*color-of mt* = *R* ∧ *bheight* (*rbt-del xx mt*) = *bheight mt* ∧ *inv1* (*rbt-del xx mt*) ∨ *color-of mt* = *B* ∧ *bheight* (*rbt-del xx mt*) = *bheight mt* − *1* ∧ *inv1l* (*rbt-del xx mt*)) **by** (*blast dest*: *rbt-del-inv1-inv2*)
with *5* **have** *3*: *entry-in-tree k v* (*rbt-del-from-right xx aa yy ss mt*) = (*entry-in-tree k v aa* ∨ (*k* = *yy* ∧ *v* = *ss*) ∨ *False* ∨ *xx* ≠ *k* ∧ *entry-in-tree k v mt*) **by** (*simp add*: *balance-right-in-tree*)
thus *?case* **proof** (*cases xx* = *k*)
case *True*
from *5 True* **have** *aa* |≪ *yy* ∧ *yy* < *k* **by** *simp*
hence *aa* |≪ *k* **by** (*blast dest*: *rbt-less-trans*)
with *3 5 True* **show** *?thesis* **by** (*auto simp*: *rbt-less-nit*)
qed *auto*
next
case (*6-1 xx aa yy ss*)
show *?case* **proof** (*cases xx* = *k*)
case *True*
with *6-1* **have** *aa* |≪ *yy* ∧ *k* > *yy* **by** *simp*
hence *aa* |≪ *k* **by** (*blast dest*: *rbt-less-trans*)
with *6-1* ‹*xx* = *k*› **show** *?thesis* **by** (*auto simp*: *rbt-less-nit*)
qed *simp*
next
case (*6-2 xx aa yy ss vaa vbb vdd vc*)
thus *?case* **proof** (*cases xx* = *k*)
case *True*

with *6-2* **have** $k > yy \wedge aa \mid\ll yy$ **by** *simp*
hence $aa \mid\ll k$ **by** (*blast dest*: *rbt-less-trans*)
with *True 6-2* **show** *?thesis* **by** (*auto simp*: *rbt-less-nit*)
qed *auto*
qed *simp*

definition (**in** *ord*) *rbt-delete* **where**
rbt-delete k t = *paint B* (*rbt-del k t*)

theorem *rbt-delete-is-rbt* [*simp*]: **assumes** *is-rbt t* **shows** *is-rbt* (*rbt-delete k t*)
proof −
from *assms* **have** *inv2 t* **and** *inv1 t* **unfolding** *is-rbt-def* **by** *auto*
hence *inv2* (*rbt-del k t*) ∧ (*color-of t* = *R* ∧ *bheight* (*rbt-del k t*) = *bheight t* ∧ *inv1* (*rbt-del k t*) ∨ *color-of t* = *B* ∧ *bheight* (*rbt-del k t*) = *bheight t* − *1* ∧ *inv1l* (*rbt-del k t*)) **by** (*rule rbt-del-inv1-inv2*)
hence *inv2* (*rbt-del k t*) ∧ *inv1l* (*rbt-del k t*) **by** (*cases color-of t*) *auto*
with *assms* **show** *?thesis*
unfolding *is-rbt-def rbt-delete-def*
by (*auto intro*: *paint-rbt-sorted rbt-del-rbt-sorted*)
qed

lemma *rbt-delete-in-tree*:
assumes *is-rbt t*
shows *entry-in-tree k v* (*rbt-delete x t*) = ($x \neq k$ ∧ *entry-in-tree k v t*)
using *assms* **unfolding** *is-rbt-def rbt-delete-def*
by (*auto simp*: *rbt-del-in-tree*)

lemma *rbt-lookup-rbt-delete*:
assumes *is-rbt*: *is-rbt t*
shows *rbt-lookup* (*rbt-delete k t*) = (*rbt-lookup t*)|‘(−{*k*})
proof
fix *x*
show *rbt-lookup* (*rbt-delete k t*) *x* = (*rbt-lookup t* |‘ (−{*k*})) *x*
proof (*cases* $x = k$)
assume $x = k$
with *is-rbt* **show** *?thesis*
by (*cases rbt-lookup* (*rbt-delete k t*) *k*) (*auto simp*: *rbt-lookup-in-tree rbt-delete-in-tree*)
next
assume $x \neq k$
thus *?thesis*
by *auto* (*metis is-rbt rbt-delete-is-rbt rbt-delete-in-tree is-rbt-rbt-sorted rbt-lookup-from-in-tree*)
qed
qed

end

108.5 Modifying existing entries

context *ord* **begin**

primrec
rbt-map-entry :: $'a \Rightarrow ('b \Rightarrow 'b) \Rightarrow ('a, 'b)$ *rbt* $\Rightarrow ('a, 'b)$ *rbt*
where
rbt-map-entry k f Empty = *Empty*
| *rbt-map-entry k f* (*Branch c lt x v rt*) =
(*if k* < *x then Branch c* (*rbt-map-entry k f lt*) *x v rt*
else if k > *x then* (*Branch c lt x v* (*rbt-map-entry k f rt*))
else Branch c lt x (*f v*) *rt*)

lemma *rbt-map-entry-color-of*: *color-of* (*rbt-map-entry k f t*) = *color-of t* **by** (*induct t*) *simp+*
lemma *rbt-map-entry-inv1*: *inv1* (*rbt-map-entry k f t*) = *inv1 t* **by** (*induct t*) (*simp add*: *rbt-map-entry-color-of*)+
lemma *rbt-map-entry-inv2*: *inv2* (*rbt-map-entry k f t*) = *inv2 t bheight* (*rbt-map-entry k f t*) = *bheight t* **by** (*induct t*) *simp+*
lemma *rbt-map-entry-rbt-greater*: *rbt-greater a* (*rbt-map-entry k f t*) = *rbt-greater a t* **by** (*induct t*) *simp+*
lemma *rbt-map-entry-rbt-less*: *rbt-less a* (*rbt-map-entry k f t*) = *rbt-less a t* **by** (*induct t*) *simp+*
lemma *rbt-map-entry-rbt-sorted*: *rbt-sorted* (*rbt-map-entry k f t*) = *rbt-sorted t*
by (*induct t*) (*simp-all add*: *rbt-map-entry-rbt-less rbt-map-entry-rbt-greater*)

theorem *rbt-map-entry-is-rbt* [*simp*]: *is-rbt* (*rbt-map-entry k f t*) = *is-rbt t*
unfolding *is-rbt-def* **by** (*simp add*: *rbt-map-entry-inv2 rbt-map-entry-color-of rbt-map-entry-rbt-sorted rbt-map-entry-inv1*)

end

theorem (**in** *linorder*) *rbt-lookup-rbt-map-entry*:
rbt-lookup (*rbt-map-entry k f t*) = (*rbt-lookup t*)(*k* := *map-option f* (*rbt-lookup t k*))
by (*induct t*) (*auto split*: *option.splits simp add*: *fun-eq-iff*)

108.6 Mapping all entries

primrec
map :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('a, 'b)$ *rbt* $\Rightarrow ('a, 'c)$ *rbt*
where
map f Empty = *Empty*
| *map f* (*Branch c lt k v rt*) = *Branch c* (*map f lt*) *k* (*f k v*) (*map f rt*)

lemma *map-entries* [*simp*]: *entries* (*map f t*) = *List.map* (λ(*k*, *v*). (*k*, *f k v*)) (*entries t*)
by (*induct t*) *auto*
lemma *map-keys* [*simp*]: *keys* (*map f t*) = *keys t* **by** (*simp add*: *keys-def split-def*)
lemma *map-color-of*: *color-of* (*map f t*) = *color-of t* **by** (*induct t*) *simp+*
lemma *map-inv1*: *inv1* (*map f t*) = *inv1 t* **by** (*induct t*) (*simp add*: *map-color-of*)+

lemma *map-inv2*: *inv2* (*map f t*) = *inv2 t bheight* (*map f t*) = *bheight t* **by** (*induct t*) *simp+*

context *ord* **begin**

lemma *map-rbt-greater*: *rbt-greater k* (*map f t*) = *rbt-greater k t* **by** (*induct t*) *simp+*
lemma *map-rbt-less*: *rbt-less k* (*map f t*) = *rbt-less k t* **by** (*induct t*) *simp+*
lemma *map-rbt-sorted*: *rbt-sorted* (*map f t*) = *rbt-sorted t* **by** (*induct t*) (*simp add*: *map-rbt-less map-rbt-greater*)+
theorem *map-is-rbt* [*simp*]: *is-rbt* (*map f t*) = *is-rbt t*
unfolding *is-rbt-def* **by** (*simp add*: *map-inv1 map-inv2 map-rbt-sorted map-color-of*)

end

theorem (**in** *linorder*) *rbt-lookup-map*: *rbt-lookup* (*map f t*) *x* = *map-option* (*f x*) (*rbt-lookup t x*)
  apply(*induct t*)
  apply *auto*
  apply(*rename-tac a b c*, *subgoal-tac x* = *a*)
  apply *auto*
  done

hide-const (**open**) *map*

108.7 Folding over entries

definition *fold* :: ('*a* ⇒ '*b* ⇒ '*c* ⇒ '*c*) ⇒ ('*a*, '*b*) *rbt* ⇒ '*c* ⇒ '*c* **where**
  fold f t = *List.fold* (*case-prod f*) (*entries t*)

lemma *fold-simps* [*simp*]:
  fold f Empty = *id*
  fold f (*Branch c lt k v rt*) = *fold f rt* ∘ *f k v* ∘ *fold f lt*
  by (*simp-all add*: *fold-def fun-eq-iff*)

lemma *fold-code* [*code*]:
  fold f Empty x = *x*
  fold f (*Branch c lt k v rt*) *x* = *fold f rt* (*f k v* (*fold f lt x*))
by(*simp-all*)

fun *foldi* :: ('*c* ⇒ *bool*) ⇒ ('*a* ⇒ '*b* ⇒ '*c* ⇒ '*c*) ⇒ ('*a* :: *linorder*, '*b*) *rbt* ⇒ '*c* ⇒ '*c*
  where
  foldi c f Empty s = *s* |
  foldi c f (*Branch col l k v r*) *s* = (
    if (*c s*) *then*

```
    let s' = foldi c f l s in
      if (c s') then
        foldi c f r (f k v s')
      else s'
  else
    s
)
```

108.8 Bulkloading a tree

definition (**in** *ord*) *rbt-bulkload* :: ('a × 'b) *list* ⇒ ('a, 'b) *rbt* **where**
rbt-bulkload xs = *foldr* (λ(k, v). *rbt-insert k v*) *xs Empty*

context *linorder* **begin**

lemma *rbt-bulkload-is-rbt* [*simp*, *intro*]:
is-rbt (*rbt-bulkload xs*)
unfolding *rbt-bulkload-def* **by** (*induct xs*) *auto*

lemma *rbt-lookup-rbt-bulkload*:
rbt-lookup (*rbt-bulkload xs*) = *map-of xs*
proof −
obtain *ys* **where** *ys* = *rev xs* **by** *simp*
have ⋀t. *is-rbt t* ⟹
rbt-lookup (*List.fold* (*case-prod rbt-insert*) *ys t*) = *rbt-lookup t* ++ *map-of* (*rev ys*)
by (*induct ys*) (*simp-all add*: *rbt-bulkload-def rbt-lookup-rbt-insert case-prod-beta*)
from *this Empty-is-rbt* **have**
rbt-lookup (*List.fold* (*case-prod rbt-insert*) (*rev xs*) *Empty*) = *rbt-lookup Empty* ++ *map-of xs*
by (*simp add*: ‹*ys* = *rev xs*›)
then show *?thesis* **by** (*simp add*: *rbt-bulkload-def rbt-lookup-Empty foldr-conv-fold*)
qed

end

108.9 Building a RBT from a sorted list

These functions have been adapted from Andrew W. Appel, Efficient Verified Red-Black Trees (September 2011)

fun *rbtreeify-f* :: *nat* ⇒ ('a × 'b) *list* ⇒ ('a, 'b) *rbt* × ('a × 'b) *list*
and *rbtreeify-g* :: *nat* ⇒ ('a × 'b) *list* ⇒ ('a, 'b) *rbt* × ('a × 'b) *list*
where

```
rbtreeify-f n kvs =
 (if n = 0 then (Empty, kvs)
  else if n = 1 then
    case kvs of (k, v) # kvs' ⇒ (Branch R Empty k v Empty, kvs')
  else if (n mod 2 = 0) then
    case rbtreeify-f (n div 2) kvs of (t1, (k, v) # kvs') ⇒
```

apfst (Branch B t1 k v) (rbtreeify-g (n div 2) kvs′)
else case rbtreeify-f (n div 2) kvs of (t1, (k, v) # kvs′) ⇒
apfst (Branch B t1 k v) (rbtreeify-f (n div 2) kvs′))

| *rbtreeify-g n kvs =*
(if n = 0 ∨ n = 1 then (Empty, kvs)
else if n mod 2 = 0 then
case rbtreeify-g (n div 2) kvs of (t1, (k, v) # kvs′) ⇒
apfst (Branch B t1 k v) (rbtreeify-g (n div 2) kvs′)
else case rbtreeify-f (n div 2) kvs of (t1, (k, v) # kvs′) ⇒
apfst (Branch B t1 k v) (rbtreeify-g (n div 2) kvs′))

definition *rbtreeify* :: (′*a* × ′*b*) *list* ⇒ (′*a*, ′*b*) *rbt*
where *rbtreeify kvs = fst (rbtreeify-g (Suc (length kvs)) kvs)*

declare *rbtreeify-f.simps* [*simp del*] *rbtreeify-g.simps* [*simp del*]

lemma *rbtreeify-f-code* [*code*]:
rbtreeify-f n kvs =
(if n = 0 then (Empty, kvs)
else if n = 1 then
case kvs of (k, v) # kvs′ ⇒
(Branch R Empty k v Empty, kvs′)
else let (n′, r) = Divides.divmod-nat n 2 in
if r = 0 then
case rbtreeify-f n′ kvs of (t1, (k, v) # kvs′) ⇒
apfst (Branch B t1 k v) (rbtreeify-g n′ kvs′)
else case rbtreeify-f n′ kvs of (t1, (k, v) # kvs′) ⇒
apfst (Branch B t1 k v) (rbtreeify-f n′ kvs′))
by (*subst rbtreeify-f.simps*) (*simp only*: *Let-def divmod-nat-div-mod prod.case*)

lemma *rbtreeify-g-code* [*code*]:
rbtreeify-g n kvs =
(if n = 0 ∨ n = 1 then (Empty, kvs)
else let (n′, r) = Divides.divmod-nat n 2 in
if r = 0 then
case rbtreeify-g n′ kvs of (t1, (k, v) # kvs′) ⇒
apfst (Branch B t1 k v) (rbtreeify-g n′ kvs′)
else case rbtreeify-f n′ kvs of (t1, (k, v) # kvs′) ⇒
apfst (Branch B t1 k v) (rbtreeify-g n′ kvs′))
by(*subst rbtreeify-g.simps*)(*simp only*: *Let-def divmod-nat-div-mod prod.case*)

lemma *Suc-double-half*: *Suc (2 ∗ n) div 2 = n*
by *simp*

lemma *div2-plus-div2*: *n div 2 + n div 2 = (n :: nat) − n mod 2*
by *arith*

lemma *rbtreeify-f-rec-aux-lemma*:

$[\![k - n\ div\ 2 = Suc\ k'; n \leq k; n\ mod\ 2 = Suc\ 0]\!]$
$\Longrightarrow k' - n\ div\ 2 = k - n$
apply(*rule add-right-imp-eq*[**where** $a = n - n\ div\ 2$])
apply(*subst add-diff-assoc2*, *arith*)
apply(*simp add*: *div2-plus-div2*)
done

lemma *rbtreeify-f-simps*:
rbtreeify-f 0 kvs = (*Empty*, *kvs*)
rbtreeify-f (*Suc 0*) ((*k*, *v*) # *kvs*) =
(*Branch R Empty k v Empty*, *kvs*)
0 < *n* ⟹ *rbtreeify-f* (*2* ∗ *n*) *kvs* =
(*case rbtreeify-f n kvs of* (*t1*, (*k*, *v*) # *kvs'*) ⇒
apfst (*Branch B t1 k v*) (*rbtreeify-g n kvs'*))
0 < *n* ⟹ *rbtreeify-f* (*Suc* (*2* ∗ *n*)) *kvs* =
(*case rbtreeify-f n kvs of* (*t1*, (*k*, *v*) # *kvs'*) ⇒
apfst (*Branch B t1 k v*) (*rbtreeify-f n kvs'*))
by(*subst* (*1*) *rbtreeify-f.simps*, *simp add*: *Suc-double-half*)+

lemma *rbtreeify-g-simps*:
rbtreeify-g 0 kvs = (*Empty*, *kvs*)
rbtreeify-g (*Suc 0*) *kvs* = (*Empty*, *kvs*)
0 < *n* ⟹ *rbtreeify-g* (*2* ∗ *n*) *kvs* =
(*case rbtreeify-g n kvs of* (*t1*, (*k*, *v*) # *kvs'*) ⇒
apfst (*Branch B t1 k v*) (*rbtreeify-g n kvs'*))
0 < *n* ⟹ *rbtreeify-g* (*Suc* (*2* ∗ *n*)) *kvs* =
(*case rbtreeify-f n kvs of* (*t1*, (*k*, *v*) # *kvs'*) ⇒
apfst (*Branch B t1 k v*) (*rbtreeify-g n kvs'*))
by(*subst* (*1*) *rbtreeify-g.simps*, *simp add*: *Suc-double-half*)+

declare *rbtreeify-f-simps*[*simp*] *rbtreeify-g-simps*[*simp*]

lemma *length-rbtreeify-f*: *n* ≤ *length kvs*
⟹ *length* (*snd* (*rbtreeify-f n kvs*)) = *length kvs* − *n*
and *length-rbtreeify-g*:⟦ *0* < *n*; *n* ≤ *Suc* (*length kvs*) ⟧
⟹ *length* (*snd* (*rbtreeify-g n kvs*)) = *Suc* (*length kvs*) − *n*
proof(*induction n kvs* **and** *n kvs rule*: *rbtreeify-f-rbtreeify-g.induct*)
case (*1 n kvs*)
show *?case*
proof(*cases n* ≤ *1*)
case *True* **thus** *?thesis* **using** *1.prems*
by(*cases n kvs rule*: *nat.exhaust*[*case-product list.exhaust*]) *auto*
next
case *False*
hence *n* ≠ *0* *n* ≠ *1* **by** *simp-all*
note *IH* = *1.IH*[*OF this*]
show *?thesis*
proof(*cases n mod 2* = *0*)
case *True*

```
  hence length (snd (rbtreeify-f n kvs)) =
    length (snd (rbtreeify-f (2 * (n div 2)) kvs))
    by(metis minus-nat.diff-0 mult-div-cancel)
  also from 1.prems False obtain k v kvs'
    where kvs: kvs = (k, v) # kvs' by(cases kvs) auto
  also have 0 < n div 2 using False by(simp)
  note rbtreeify-f-simps(3)[OF this]
  also note kvs[symmetric]
  also let ?rest1 = snd (rbtreeify-f (n div 2) kvs)
  from 1.prems have n div 2 ≤ length kvs by simp
  with True have len: length ?rest1 = length kvs − n div 2 by(rule IH)
  with 1.prems False obtain t1 k' v' kvs''
    where kvs'': rbtreeify-f (n div 2) kvs = (t1, (k', v') # kvs'')
     by(cases ?rest1)(auto simp add: snd-def split: prod.split-asm)
  note this also note prod.case also note list.simps(5)
  also note prod.case also note snd-apfst
  also have 0 < n div 2 n div 2 ≤ Suc (length kvs'')
    using len 1.prems False unfolding kvs'' by simp-all
  with True kvs''[symmetric] refl refl
  have length (snd (rbtreeify-g (n div 2) kvs'')) =
    Suc (length kvs'') − n div 2 by(rule IH)
  finally show ?thesis using len[unfolded kvs''] 1.prems True
    by(simp add: Suc-diff-le[symmetric] mult-2[symmetric] mult-div-cancel)
next
  case False
  hence length (snd (rbtreeify-f n kvs)) =
    length (snd (rbtreeify-f (Suc (2 * (n div 2))) kvs))
    by (simp add: mod-eq-0-iff-dvd)
  also from 1.prems ‹¬ n ≤ 1› obtain k v kvs'
    where kvs: kvs = (k, v) # kvs' by(cases kvs) auto
  also have 0 < n div 2 using ‹¬ n ≤ 1› by(simp)
  note rbtreeify-f-simps(4)[OF this]
  also note kvs[symmetric]
  also let ?rest1 = snd (rbtreeify-f (n div 2) kvs)
  from 1.prems have n div 2 ≤ length kvs by simp
  with False have len: length ?rest1 = length kvs − n div 2 by(rule IH)
  with 1.prems ‹¬ n ≤ 1› obtain t1 k' v' kvs''
    where kvs'': rbtreeify-f (n div 2) kvs = (t1, (k', v') # kvs'')
    by(cases ?rest1)(auto simp add: snd-def split: prod.split-asm)
  note this also note prod.case also note list.simps(5)
  also note prod.case also note snd-apfst
  also have n div 2 ≤ length kvs''
    using len 1.prems False unfolding kvs'' by simp arith
  with False kvs''[symmetric] refl refl
  have length (snd (rbtreeify-f (n div 2) kvs'')) = length kvs'' − n div 2
    by(rule IH)
  finally show ?thesis using len[unfolded kvs''] 1.prems False
    by simp(rule rbtreeify-f-rec-aux-lemma[OF sym])
qed
```

qed
next
case (*2 n kvs*)
show *?case*
proof(*cases n > 1*)
case *False* **with** ‹*0 < n*› **show** *?thesis*
by(*cases n kvs rule*: *nat.exhaust*[*case-product list.exhaust*]) *simp-all*
next
case *True*
hence ¬ (*n* = *0* ∨ *n* = *1*) **by** *simp*
note *IH* = *2.IH*[*OF this*]
show *?thesis*
proof(*cases n mod 2* = *0*)
case *True*
hence *length* (*snd* (*rbtreeify-g n kvs*)) =
length (*snd* (*rbtreeify-g* (*2* ∗ (*n div 2*)) *kvs*))
by(*metis minus-nat.diff-0 mult-div-cancel*)
also from *2.prems True* **obtain** *k v kvs'*
where *kvs*: *kvs* = (*k*, *v*) # *kvs'* **by**(*cases kvs*) *auto*
also have *0 < n div 2* **using** ‹*1 < n*› **by**(*simp*)
note *rbtreeify-g-simps*(*3*)[*OF this*]
also note *kvs*[*symmetric*]
also let *?rest1* = *snd* (*rbtreeify-g* (*n div 2*) *kvs*)
from *2.prems* ‹*1 < n*›
have *0 < n div 2 n div 2* ≤ *Suc* (*length kvs*) **by** *simp-all*
with *True* **have** *len*: *length ?rest1* = *Suc* (*length kvs*) − *n div 2* **by**(*rule IH*)
with *2.prems* **obtain** *t1 k' v' kvs''*
where *kvs''*: *rbtreeify-g* (*n div 2*) *kvs* = (*t1*, (*k'*, *v'*) # *kvs''*)
by(*cases ?rest1*)(*auto simp add*: *snd-def split*: *prod.split-asm*)
note *this* **also note** *prod.case* **also note** *list.simps*(*5*)
also note *prod.case* **also note** *snd-apfst*
also have *n div 2* ≤ *Suc* (*length kvs''*)
using *len 2.prems* **unfolding** *kvs''* **by** *simp*
with *True kvs''*[*symmetric*] *refl refl* ‹*0 < n div 2*›
have *length* (*snd* (*rbtreeify-g* (*n div 2*) *kvs''*)) = *Suc* (*length kvs''*) − *n div 2*
by(*rule IH*)
finally show *?thesis* **using** *len*[*unfolded kvs''*] *2.prems True*
by(*simp add*: *Suc-diff-le*[*symmetric*] *mult-2*[*symmetric*] *mult-div-cancel*)
next
case *False*
hence *length* (*snd* (*rbtreeify-g n kvs*)) =
length (*snd* (*rbtreeify-g* (*Suc* (*2* ∗ (*n div 2*))) *kvs*))
by (*simp add*: *mod-eq-0-iff-dvd*)
also from *2.prems* ‹*1 < n*› **obtain** *k v kvs'*
where *kvs*: *kvs* = (*k*, *v*) # *kvs'* **by**(*cases kvs*) *auto*
also have *0 < n div 2* **using** ‹*1 < n*› **by**(*simp*)
note *rbtreeify-g-simps*(*4*)[*OF this*]
also note *kvs*[*symmetric*]
also let *?rest1* = *snd* (*rbtreeify-f* (*n div 2*) *kvs*)

from *2.prems* **have** *n div 2 ≤ length kvs* **by** *simp*
with *False* **have** *len*: *length ?rest1 = length kvs − n div 2* **by**(*rule IH*)
with *2.prems* ‹*1 < n*› *False* **obtain** *t1 k′ v′ kvs″*
where *kvs″*: *rbtreeify-f (n div 2) kvs = (t1, (k′, v′) # kvs″)*
by(*cases ?rest1*)(*auto simp add*: *snd-def split*: *prod.split-asm*, *arith*)
note *this* **also note** *prod.case* **also note** *list.simps*(*5*)
also note *prod.case* **also note** *snd-apfst*
also have *n div 2 ≤ Suc (length kvs″)*
using *len 2.prems False* **unfolding** *kvs″* **by** *simp arith*
with *False kvs″*[*symmetric*] *refl refl* ‹*0 < n div 2*›
have *length (snd (rbtreeify-g (n div 2) kvs″)) = Suc (length kvs″) − n div 2*
by(*rule IH*)
finally show *?thesis* **using** *len*[*unfolded kvs″*] *2.prems False*
by(*simp add*: *div2-plus-div2*)
qed
qed
qed

lemma *rbtreeify-induct* [*consumes 1*, *case-names f-0 f-1 f-even f-odd g-0 g-1 g-even g-odd*]:
fixes *P Q*
defines *f0 == (⋀kvs. P 0 kvs)*
and *f1 == (⋀k v kvs. P (Suc 0) ((k, v) # kvs))*
and *feven ==*
(⋀n kvs t k v kvs′. ⟦ n > 0; n ≤ length kvs; P n kvs;
rbtreeify-f n kvs = (t, (k, v) # kvs′); n ≤ Suc (length kvs′); Q n kvs′ ⟧
⟹ P (2 ∗ n) kvs)
and *fodd ==*
(⋀n kvs t k v kvs′. ⟦ n > 0; n ≤ length kvs; P n kvs;
rbtreeify-f n kvs = (t, (k, v) # kvs′); n ≤ length kvs′; P n kvs′ ⟧
⟹ P (Suc (2 ∗ n)) kvs)
and *g0 == (⋀kvs. Q 0 kvs)*
and *g1 == (⋀kvs. Q (Suc 0) kvs)*
and *geven ==*
(⋀n kvs t k v kvs′. ⟦ n > 0; n ≤ Suc (length kvs); Q n kvs;
rbtreeify-g n kvs = (t, (k, v) # kvs′); n ≤ Suc (length kvs′); Q n kvs′ ⟧
⟹ Q (2 ∗ n) kvs)
and *godd ==*
(⋀n kvs t k v kvs′. ⟦ n > 0; n ≤ length kvs; P n kvs;
rbtreeify-f n kvs = (t, (k, v) # kvs′); n ≤ Suc (length kvs′); Q n kvs′ ⟧
⟹ Q (Suc (2 ∗ n)) kvs)
shows *⟦ n ≤ length kvs;*
PROP f0; PROP f1; PROP feven; PROP fodd;
PROP g0; PROP g1; PROP geven; PROP godd ⟧
⟹ P n kvs
and *⟦ n ≤ Suc (length kvs);*
PROP f0; PROP f1; PROP feven; PROP fodd;
PROP g0; PROP g1; PROP geven; PROP godd ⟧
⟹ Q n kvs

proof −
assume *f0*: *PROP f0* **and** *f1*: *PROP f1* **and** *feven*: *PROP feven* **and** *fodd*: *PROP fodd*
and *g0*: *PROP g0* **and** *g1*: *PROP g1* **and** *geven*: *PROP geven* **and** *godd*: *PROP godd*
show $n \le$ *length kvs* $\Longrightarrow$ *P n kvs* **and** $n \le$ *Suc* (*length kvs*) $\Longrightarrow$ *Q n kvs*
proof(*induction rule*: *rbtreeify-f-rbtreeify-g.induct*)
case (*1 n kvs*)
show *?case*
proof(*cases* $n \le 1$)
case *True* **thus** *?thesis* **using** *1.prems*
by(*cases n kvs rule*: *nat.exhaust*[*case-product list.exhaust*])
(*auto simp add*: *f0*[*unfolded f0-def*] *f1*[*unfolded f1-def*])
next
case *False*
hence *ns*: $n \neq 0$ $n \neq 1$ **by** *simp-all*
hence *ge0*: *n div 2* > 0 **by** *simp*
note *IH* = *1.IH*[*OF ns*]
show *?thesis*
proof(*cases n mod 2 = 0*)
case *True* **note** *ge0*
moreover from *1.prems* **have** *n2*: *n div 2* $\le$ *length kvs* **by** *simp*
moreover from *True n2* **have** *P* (*n div 2*) *kvs* **by**(*rule IH*)
moreover from *length-rbtreeify-f*[*OF n2*] *ge0 1.prems* **obtain** *t k v kvs'*
where *kvs'*: *rbtreeify-f* (*n div 2*) *kvs* = (*t*, (*k*, *v*) # *kvs'*)
by(*cases snd* (*rbtreeify-f* (*n div 2*) *kvs*))
(*auto simp add*: *snd-def split*: *prod.split-asm*)
moreover from *1.prems length-rbtreeify-f*[*OF n2*] *ge0*
have *n2'*: *n div 2* $\le$ *Suc* (*length kvs'*) **by**(*simp add*: *kvs'*)
moreover from *True kvs'*[*symmetric*] *refl refl n2'*
have *Q* (*n div 2*) *kvs'* **by**(*rule IH*)
moreover note *feven*[*unfolded feven-def*]

ultimately have *P* (*2* ∗ (*n div 2*)) *kvs* **by** −
thus *?thesis* **using** *True* **by** (*metis div-mod-equality' minus-nat.diff-0 mult.commute*)
next
case *False* **note** *ge0*
moreover from *1.prems* **have** *n2*: *n div 2* $\le$ *length kvs* **by** *simp*
moreover from *False n2* **have** *P* (*n div 2*) *kvs* **by**(*rule IH*)
moreover from *length-rbtreeify-f*[*OF n2*] *ge0 1.prems* **obtain** *t k v kvs'*
where *kvs'*: *rbtreeify-f* (*n div 2*) *kvs* = (*t*, (*k*, *v*) # *kvs'*)
by(*cases snd* (*rbtreeify-f* (*n div 2*) *kvs*))
(*auto simp add*: *snd-def split*: *prod.split-asm*)
moreover from *1.prems length-rbtreeify-f*[*OF n2*] *ge0 False*
have *n2'*: *n div 2* $\le$ *length kvs'* **by**(*simp add*: *kvs'*) *arith*
moreover from *False kvs'*[*symmetric*] *refl refl n2'* **have** *P* (*n div 2*) *kvs'* **by**(*rule IH*)
moreover note *fodd*[*unfolded fodd-def*]

```
      ultimately have P (Suc (2 * (n div 2))) kvs by -
      thus ?thesis using False
     by simp (metis One-nat-def Suc-eq-plus1-left le-add-diff-inverse mod-less-eq-dividend
mult-div-cancel)
    qed
  qed
 next
  case (2 n kvs)
  show ?case
  proof(cases n ≤ 1)
    case True thus ?thesis using 2.prems
      by(cases n kvs rule: nat.exhaust[case-product list.exhaust])
        (auto simp add: g0[unfolded g0-def] g1[unfolded g1-def])
  next
    case False
    hence ns: ¬ (n = 0 ∨ n = 1) by simp
    hence ge0: n div 2 > 0 by simp
    note IH = 2.IH[OF ns]
    show ?thesis
    proof(cases n mod 2 = 0)
      case True note ge0
      moreover from 2.prems have n2: n div 2 ≤ Suc (length kvs) by simp
      moreover from True n2 have Q (n div 2) kvs by(rule IH)
       moreover from length-rbtreeify-g[OF ge0 n2] ge0 2.prems obtain t k v
kvs'
        where kvs': rbtreeify-g (n div 2) kvs = (t, (k, v) # kvs')
        by(cases snd (rbtreeify-g (n div 2) kvs))
          (auto simp add: snd-def split: prod.split-asm)
      moreover from 2.prems length-rbtreeify-g[OF ge0 n2] ge0
      have n2': n div 2 ≤ Suc (length kvs') by(simp add: kvs')
      moreover from True kvs'[symmetric] refl refl n2'
      have Q (n div 2) kvs' by(rule IH)
      moreover note geven[unfolded geven-def]
      ultimately have Q (2 * (n div 2)) kvs by -
      thus ?thesis using True
        by(metis div-mod-equality' minus-nat.diff-0 mult.commute)
    next
      case False note ge0
      moreover from 2.prems have n2: n div 2 ≤ length kvs by simp
      moreover from False n2 have P (n div 2) kvs by(rule IH)
      moreover from length-rbtreeify-f[OF n2] ge0 2.prems False obtain t k v
kvs'
        where kvs': rbtreeify-f (n div 2) kvs = (t, (k, v) # kvs')
        by(cases snd (rbtreeify-f (n div 2) kvs))
          (auto simp add: snd-def split: prod.split-asm, arith)
      moreover from 2.prems length-rbtreeify-f[OF n2] ge0 False
      have n2': n div 2 ≤ Suc (length kvs') by(simp add: kvs') arith
      moreover from False kvs'[symmetric] refl refl n2'
      have Q (n div 2) kvs' by(rule IH)
```

moreover note *godd*[*unfolded godd-def*]
ultimately have *Q* (*Suc* (*2* ∗ (*n div 2*))) *kvs* **by** −
thus *?thesis* **using** *False*
by *simp* (*metis One-nat-def Suc-eq-plus1-left le-add-diff-inverse mod-less-eq-dividend mult-div-cancel*)
qed
qed
qed
qed

lemma *inv1-rbtreeify-f*: *n* ≤ *length kvs*
⟹ *inv1* (*fst* (*rbtreeify-f n kvs*))
and *inv1-rbtreeify-g*: *n* ≤ *Suc* (*length kvs*)
⟹ *inv1* (*fst* (*rbtreeify-g n kvs*))
by(*induct n kvs* **and** *n kvs rule*: *rbtreeify-induct*) *simp-all*

fun *plog2* :: *nat* ⇒ *nat*
where *plog2 n* = (*if n* ≤ *1 then 0 else plog2* (*n div 2*) + *1*)

declare *plog2.simps* [*simp del*]

lemma *plog2-simps* [*simp*]:
plog2 0 = *0 plog2* (*Suc 0*) = *0*
0 < *n* ⟹ *plog2* (*2* ∗ *n*) = *1* + *plog2 n*
0 < *n* ⟹ *plog2* (*Suc* (*2* ∗ *n*)) = *1* + *plog2 n*
by(*subst plog2.simps*, *simp add*: *Suc-double-half*)+

lemma *bheight-rbtreeify-f*: *n* ≤ *length kvs*
⟹ *bheight* (*fst* (*rbtreeify-f n kvs*)) = *plog2 n*
and *bheight-rbtreeify-g*: *n* ≤ *Suc* (*length kvs*)
⟹ *bheight* (*fst* (*rbtreeify-g n kvs*)) = *plog2 n*
by(*induct n kvs* **and** *n kvs rule*: *rbtreeify-induct*) *simp-all*

lemma *bheight-rbtreeify-f-eq-plog2I*:
⟦ *rbtreeify-f n kvs* = (*t*, *kvs'*); *n* ≤ *length kvs* ⟧
⟹ *bheight t* = *plog2 n*
using *bheight-rbtreeify-f*[*of n kvs*] **by** *simp*

lemma *bheight-rbtreeify-g-eq-plog2I*:
⟦ *rbtreeify-g n kvs* = (*t*, *kvs'*); *n* ≤ *Suc* (*length kvs*) ⟧
⟹ *bheight t* = *plog2 n*
using *bheight-rbtreeify-g*[*of n kvs*] **by** *simp*

hide-const (**open**) *plog2*

lemma *inv2-rbtreeify-f*: *n* ≤ *length kvs*
⟹ *inv2* (*fst* (*rbtreeify-f n kvs*))
and *inv2-rbtreeify-g*: *n* ≤ *Suc* (*length kvs*)
⟹ *inv2* (*fst* (*rbtreeify-g n kvs*))

by(*induct n kvs* **and** *n kvs rule*: *rbtreeify-induct*)
(*auto simp add*: *bheight-rbtreeify-f bheight-rbtreeify-g*
intro: *bheight-rbtreeify-f-eq-plog2I bheight-rbtreeify-g-eq-plog2I*)

lemma $n \leq$ *length kvs* $\Longrightarrow$ *True*
and *color-of-rbtreeify-g*:
$[\![n \leq Suc\ (length\ kvs);\ 0 < n]\!]$
$\Longrightarrow$ *color-of* (*fst* (*rbtreeify-g n kvs*)) = *B*
by(*induct n kvs* **and** *n kvs rule*: *rbtreeify-induct*) *simp-all*

lemma *entries-rbtreeify-f-append*:
$n \leq$ *length kvs*
$\Longrightarrow$ *entries* (*fst* (*rbtreeify-f n kvs*)) @ *snd* (*rbtreeify-f n kvs*) = *kvs*
and *entries-rbtreeify-g-append*:
$n \leq$ *Suc* (*length kvs*)
$\Longrightarrow$ *entries* (*fst* (*rbtreeify-g n kvs*)) @ *snd* (*rbtreeify-g n kvs*) = *kvs*
by(*induction rule*: *rbtreeify-induct*) *simp-all*

lemma *length-entries-rbtreeify-f*:
$n \leq$ *length kvs* $\Longrightarrow$ *length* (*entries* (*fst* (*rbtreeify-f n kvs*))) = *n*
and *length-entries-rbtreeify-g*:
$n \leq$ *Suc* (*length kvs*) $\Longrightarrow$ *length* (*entries* (*fst* (*rbtreeify-g n kvs*))) = $n - 1$
by(*induct rule*: *rbtreeify-induct*) *simp-all*

lemma *rbtreeify-f-conv-drop*:
$n \leq$ *length kvs* $\Longrightarrow$ *snd* (*rbtreeify-f n kvs*) = *drop n kvs*
using *entries-rbtreeify-f-append*[*of n kvs*]
by(*simp add*: *append-eq-conv-conj length-entries-rbtreeify-f*)

lemma *rbtreeify-g-conv-drop*:
$n \leq$ *Suc* (*length kvs*) $\Longrightarrow$ *snd* (*rbtreeify-g n kvs*) = *drop* ($n - 1$) *kvs*
using *entries-rbtreeify-g-append*[*of n kvs*]
by(*simp add*: *append-eq-conv-conj length-entries-rbtreeify-g*)

lemma *entries-rbtreeify-f* [*simp*]:
$n \leq$ *length kvs* $\Longrightarrow$ *entries* (*fst* (*rbtreeify-f n kvs*)) = *take n kvs*
using *entries-rbtreeify-f-append*[*of n kvs*]
by(*simp add*: *append-eq-conv-conj length-entries-rbtreeify-f*)

lemma *entries-rbtreeify-g* [*simp*]:
$n \leq$ *Suc* (*length kvs*) $\Longrightarrow$
entries (*fst* (*rbtreeify-g n kvs*)) = *take* ($n - 1$) *kvs*
using *entries-rbtreeify-g-append*[*of n kvs*]
by(*simp add*: *append-eq-conv-conj length-entries-rbtreeify-g*)

lemma *keys-rbtreeify-f* [*simp*]: $n \leq$ *length kvs*
$\Longrightarrow$ *keys* (*fst* (*rbtreeify-f n kvs*)) = *take n* (*map fst kvs*)
by(*simp add*: *keys-def take-map*)

lemma *keys-rbtreeify-g* [*simp*]: *n* ≤ *Suc* (*length kvs*)
⟹ *keys* (*fst* (*rbtreeify-g n kvs*)) = *take* (*n* − *1*) (*map fst kvs*)
by(*simp add*: *keys-def take-map*)

lemma *rbtreeify-fD*:
⟦ *rbtreeify-f n kvs* = (*t*, *kvs'*); *n* ≤ *length kvs* ⟧
⟹ *entries t* = *take n kvs* ∧ *kvs'* = *drop n kvs*
using *rbtreeify-f-conv-drop*[*of n kvs*] *entries-rbtreeify-f*[*of n kvs*] **by** *simp*

lemma *rbtreeify-gD*:
⟦ *rbtreeify-g n kvs* = (*t*, *kvs'*); *n* ≤ *Suc* (*length kvs*) ⟧
⟹ *entries t* = *take* (*n* − *1*) *kvs* ∧ *kvs'* = *drop* (*n* − *1*) *kvs*
using *rbtreeify-g-conv-drop*[*of n kvs*] *entries-rbtreeify-g*[*of n kvs*] **by** *simp*

lemma *entries-rbtreeify* [*simp*]: *entries* (*rbtreeify kvs*) = *kvs*
by(*simp add*: *rbtreeify-def entries-rbtreeify-g*)

context *linorder* **begin**

lemma *rbt-sorted-rbtreeify-f*:
⟦ *n* ≤ *length kvs*; *sorted* (*map fst kvs*); *distinct* (*map fst kvs*) ⟧
⟹ *rbt-sorted* (*fst* (*rbtreeify-f n kvs*))
and *rbt-sorted-rbtreeify-g*:
⟦ *n* ≤ *Suc* (*length kvs*); *sorted* (*map fst kvs*); *distinct* (*map fst kvs*) ⟧
⟹ *rbt-sorted* (*fst* (*rbtreeify-g n kvs*))
proof(*induction n kvs* **and** *n kvs rule*: *rbtreeify-induct*)
case (*f-even n kvs t k v kvs'*)
from *rbtreeify-fD*[*OF* ‹*rbtreeify-f n kvs* = (*t*, (*k*, *v*) # *kvs'*)› ‹*n* ≤ *length kvs*›]
have *entries t* = *take n kvs*
and *kvs'*: *drop n kvs* = (*k*, *v*) # *kvs'* **by** *simp-all*
hence *unfold*: *kvs* = *take n kvs* @ (*k*, *v*) # *kvs'* **by**(*metis append-take-drop-id*)
from ‹*sorted* (*map fst kvs*)› *kvs'*
have (∀ (*x*, *y*) ∈ *set* (*take n kvs*). *x* ≤ *k*) ∧ (∀ (*x*, *y*) ∈ *set kvs'*. *k* ≤ *x*)
by(*subst* (*asm*) *unfold*)(*auto simp add*: *sorted-append sorted-Cons*)
moreover from ‹*distinct* (*map fst kvs*)› *kvs'*
have (∀ (*x*, *y*) ∈ *set* (*take n kvs*). *x* ≠ *k*) ∧ (∀ (*x*, *y*) ∈ *set kvs'*. *x* ≠ *k*)
by(*subst* (*asm*) *unfold*)(*auto intro*: *rev-image-eqI*)
ultimately have (∀ (*x*, *y*) ∈ *set* (*take n kvs*). *x* < *k*) ∧ (∀ (*x*, *y*) ∈ *set kvs'*. *k* <
x)
by *fastforce*
hence *fst* (*rbtreeify-f n kvs*) |« *k k* «| *fst* (*rbtreeify-g n kvs'*)
using ‹*n* ≤ *Suc* (*length kvs'*)› ‹*n* ≤ *length kvs*› *set-take-subset*[*of n* − *1 kvs'*]
by(*auto simp add*: *ord.rbt-greater-prop ord.rbt-less-prop take-map split-def*)
moreover from ‹*sorted* (*map fst kvs*)› ‹*distinct* (*map fst kvs*)›
have *rbt-sorted* (*fst* (*rbtreeify-f n kvs*)) **by**(*rule f-even.IH*)
moreover have *sorted* (*map fst kvs'*) *distinct* (*map fst kvs'*)
using ‹*sorted* (*map fst kvs*)› ‹*distinct* (*map fst kvs*)›
by(*subst* (*asm*) (*1 2*) *unfold*, *simp add*: *sorted-append sorted-Cons*)+
hence *rbt-sorted* (*fst* (*rbtreeify-g n kvs'*)) **by**(*rule f-even.IH*)

ultimately show *?case*
  using ‹*0* < *n*› ‹*rbtreeify-f n kvs* = (*t*, (*k*, *v*) # *kvs'*)› **by** *simp*
next
  case (*f-odd n kvs t k v kvs'*)
  from *rbtreeify-fD*[*OF* ‹*rbtreeify-f n kvs* = (*t*, (*k*, *v*) # *kvs'*)› ‹*n* ≤ *length kvs*›]
  have *entries t* = *take n kvs*
    and *kvs'*: *drop n kvs* = (*k*, *v*) # *kvs'* **by** *simp-all*
  hence *unfold*: *kvs* = *take n kvs* @ (*k*, *v*) # *kvs'* **by**(*metis append-take-drop-id*)
  from ‹*sorted* (*map fst kvs*)› *kvs'*
  have (∀(*x*, *y*) ∈ *set* (*take n kvs*). *x* ≤ *k*) ∧ (∀(*x*, *y*) ∈ *set kvs'*. *k* ≤ *x*)
    by(*subst* (*asm*) *unfold*)(*auto simp add*: *sorted-append sorted-Cons*)
  moreover from ‹*distinct* (*map fst kvs*)› *kvs'*
  have (∀(*x*, *y*) ∈ *set* (*take n kvs*). *x* ≠ *k*) ∧ (∀(*x*, *y*) ∈ *set kvs'*. *x* ≠ *k*)
    by(*subst* (*asm*) *unfold*)(*auto intro*: *rev-image-eqI*)
  ultimately have (∀(*x*, *y*) ∈ *set* (*take n kvs*). *x* < *k*) ∧ (∀(*x*, *y*) ∈ *set kvs'*. *k* < *x*)
    by *fastforce*
  hence *fst* (*rbtreeify-f n kvs*) |≪ *k* *k* ≪| *fst* (*rbtreeify-f n kvs'*)
    using ‹*n* ≤ *length kvs'*› ‹*n* ≤ *length kvs*› *set-take-subset*[*of n kvs'*]
    by(*auto simp add*: *rbt-greater-prop rbt-less-prop take-map split-def*)
  moreover from ‹*sorted* (*map fst kvs*)› ‹*distinct* (*map fst kvs*)›
  have *rbt-sorted* (*fst* (*rbtreeify-f n kvs*)) **by**(*rule f-odd.IH*)
  moreover have *sorted* (*map fst kvs'*) *distinct* (*map fst kvs'*)
    using ‹*sorted* (*map fst kvs*)› ‹*distinct* (*map fst kvs*)›
    by(*subst* (*asm*) (*1 2*) *unfold*, *simp add*: *sorted-append sorted-Cons*)+
  hence *rbt-sorted* (*fst* (*rbtreeify-f n kvs'*)) **by**(*rule f-odd.IH*)
  ultimately show *?case*
    using ‹*0* < *n*› ‹*rbtreeify-f n kvs* = (*t*, (*k*, *v*) # *kvs'*)› **by** *simp*
next
  case (*g-even n kvs t k v kvs'*)
  from *rbtreeify-gD*[*OF* ‹*rbtreeify-g n kvs* = (*t*, (*k*, *v*) # *kvs'*)› ‹*n* ≤ *Suc* (*length kvs*)›]
  have *t*: *entries t* = *take* (*n* − *1*) *kvs*
    and *kvs'*: *drop* (*n* − *1*) *kvs* = (*k*, *v*) # *kvs'* **by** *simp-all*
  hence *unfold*: *kvs* = *take* (*n* − *1*) *kvs* @ (*k*, *v*) # *kvs'* **by**(*metis append-take-drop-id*)
  from ‹*sorted* (*map fst kvs*)› *kvs'*
  have (∀(*x*, *y*) ∈ *set* (*take* (*n* − *1*) *kvs*). *x* ≤ *k*) ∧ (∀(*x*, *y*) ∈ *set kvs'*. *k* ≤ *x*)
    by(*subst* (*asm*) *unfold*)(*auto simp add*: *sorted-append sorted-Cons*)
  moreover from ‹*distinct* (*map fst kvs*)› *kvs'*
  have (∀(*x*, *y*) ∈ *set* (*take* (*n* − *1*) *kvs*). *x* ≠ *k*) ∧ (∀(*x*, *y*) ∈ *set kvs'*. *x* ≠ *k*)
    by(*subst* (*asm*) *unfold*)(*auto intro*: *rev-image-eqI*)
  ultimately have (∀(*x*, *y*) ∈ *set* (*take* (*n* − *1*) *kvs*). *x* < *k*) ∧ (∀(*x*, *y*) ∈ *set kvs'*. *k* < *x*)
    by *fastforce*
  hence *fst* (*rbtreeify-g n kvs*) |≪ *k* *k* ≪| *fst* (*rbtreeify-g n kvs'*)
    using ‹*n* ≤ *Suc* (*length kvs'*)› ‹*n* ≤ *Suc* (*length kvs*)› *set-take-subset*[*of n* − *1* *kvs'*]
    by(*auto simp add*: *rbt-greater-prop rbt-less-prop take-map split-def*)
  moreover from ‹*sorted* (*map fst kvs*)› ‹*distinct* (*map fst kvs*)›

have *rbt-sorted* (*fst* (*rbtreeify-g n kvs*)) **by**(*rule g-even.IH*)
moreover have *sorted* (*map fst kvs'*) *distinct* (*map fst kvs'*)
using ‹*sorted* (*map fst kvs*)› ‹*distinct* (*map fst kvs*)›
by(*subst* (*asm*) (*1 2*) *unfold*, *simp add*: *sorted-append sorted-Cons*)+
hence *rbt-sorted* (*fst* (*rbtreeify-g n kvs'*)) **by**(*rule g-even.IH*)
ultimately show *?case* **using** ‹*0* < *n*› ‹*rbtreeify-g n kvs* = (*t*, (*k*, *v*) # *kvs'*)›
by *simp*
next
case (*g-odd n kvs t k v kvs'*)
from *rbtreeify-fD*[*OF* ‹*rbtreeify-f n kvs* = (*t*, (*k*, *v*) # *kvs'*)› ‹*n* ≤ *length kvs*›]
have *entries t* = *take n kvs*
and *kvs'*: *drop n kvs* = (*k*, *v*) # *kvs'* **by** *simp-all*
hence *unfold*: *kvs* = *take n kvs* @ (*k*, *v*) # *kvs'* **by**(*metis append-take-drop-id*)
from ‹*sorted* (*map fst kvs*)› *kvs'*
have (∀ (*x*, *y*) ∈ *set* (*take n kvs*). *x* ≤ *k*) ∧ (∀ (*x*, *y*) ∈ *set kvs'*. *k* ≤ *x*)
by(*subst* (*asm*) *unfold*)(*auto simp add*: *sorted-append sorted-Cons*)
moreover from ‹*distinct* (*map fst kvs*)› *kvs'*
have (∀ (*x*, *y*) ∈ *set* (*take n kvs*). *x* ≠ *k*) ∧ (∀ (*x*, *y*) ∈ *set kvs'*. *x* ≠ *k*)
by(*subst* (*asm*) *unfold*)(*auto intro*: *rev-image-eqI*)
ultimately have (∀ (*x*, *y*) ∈ *set* (*take n kvs*). *x* < *k*) ∧ (∀ (*x*, *y*) ∈ *set kvs'*. *k* < *x*)
by *fastforce*
hence *fst* (*rbtreeify-f n kvs*) |≪ *k k* ≪| *fst* (*rbtreeify-g n kvs'*)
using ‹*n* ≤ *Suc* (*length kvs'*)› ‹*n* ≤ *length kvs*› *set-take-subset*[*of n* − *1 kvs'*]
by(*auto simp add*: *rbt-greater-prop rbt-less-prop take-map split-def*)
moreover from ‹*sorted* (*map fst kvs*)› ‹*distinct* (*map fst kvs*)›
have *rbt-sorted* (*fst* (*rbtreeify-f n kvs*)) **by**(*rule g-odd.IH*)
moreover have *sorted* (*map fst kvs'*) *distinct* (*map fst kvs'*)
using ‹*sorted* (*map fst kvs*)› ‹*distinct* (*map fst kvs*)›
by(*subst* (*asm*) (*1 2*) *unfold*, *simp add*: *sorted-append sorted-Cons*)+
hence *rbt-sorted* (*fst* (*rbtreeify-g n kvs'*)) **by**(*rule g-odd.IH*)
ultimately show *?case*
using ‹*0* < *n*› ‹*rbtreeify-f n kvs* = (*t*, (*k*, *v*) # *kvs'*)› **by** *simp*
qed *simp-all*

lemma *rbt-sorted-rbtreeify*:
⟦ *sorted* (*map fst kvs*); *distinct* (*map fst kvs*) ⟧ ⟹ *rbt-sorted* (*rbtreeify kvs*)
by(*simp add*: *rbtreeify-def rbt-sorted-rbtreeify-g*)

lemma *is-rbt-rbtreeify*:
⟦ *sorted* (*map fst kvs*); *distinct* (*map fst kvs*) ⟧
⟹ *is-rbt* (*rbtreeify kvs*)
by(*simp add*: *is-rbt-def rbtreeify-def inv1-rbtreeify-g inv2-rbtreeify-g rbt-sorted-rbtreeify-g color-of-rbtreeify-g*)

lemma *rbt-lookup-rbtreeify*:
⟦ *sorted* (*map fst kvs*); *distinct* (*map fst kvs*) ⟧ ⟹
rbt-lookup (*rbtreeify kvs*) = *map-of kvs*
by(*simp add*: *map-of-entries*[*symmetric*] *rbt-sorted-rbtreeify*)

end

Functions to compare the height of two rbt trees, taken from Andrew W. Appel, Efficient Verified Red-Black Trees (September 2011)

fun *skip-red* :: ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt*
where
skip-red (*Branch color.R l k v r*) = *l*
| *skip-red t* = *t*

definition *skip-black* :: ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt*
where
skip-black t = (*let t*' = *skip-red t in case t*' *of Branch color.B l k v r* ⇒ *l* | - ⇒ *t*')

datatype *compare* = *LT* | *GT* | *EQ*

partial-function (*tailrec*) *compare-height* :: ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt* ⇒ *compare*
where
compare-height sx s t tx =
(*case* (*skip-red sx*, *skip-red s*, *skip-red t*, *skip-red tx*) *of*
(*Branch* - *sx*' - - -, *Branch* - *s*' - - -, *Branch* - *t*' - - -, *Branch* - *tx*' - - -) ⇒
compare-height (*skip-black sx*') *s*' *t*' (*skip-black tx*')
| (-, *rbt.Empty*, -, *Branch* - - - - -) ⇒ *LT*
| (*Branch* - - - - -, -, *rbt.Empty*, -) ⇒ *GT*
| (*Branch* - *sx*' - - -, *Branch* - *s*' - - -, *Branch* - *t*' - - -, *rbt.Empty*) ⇒
compare-height (*skip-black sx*') *s*' *t*' *rbt.Empty*
| (*rbt.Empty*, *Branch* - *s*' - - -, *Branch* - *t*' - - -, *Branch* - *tx*' - - -) ⇒
compare-height rbt.Empty s' *t*' (*skip-black tx*')
| - ⇒ *EQ*)

declare *compare-height.simps* [*code*]

hide-type (**open**) *compare*
hide-const (**open**)
compare-height skip-black skip-red LT GT EQ case-compare rec-compare
Abs-compare Rep-compare
hide-fact (**open**)
Abs-compare-cases Abs-compare-induct Abs-compare-inject Abs-compare-inverse
Rep-compare Rep-compare-cases Rep-compare-induct Rep-compare-inject Rep-compare-inverse
compare.simps compare.exhaust compare.induct compare.rec compare.simps
compare.size compare.case-cong compare.case-cong-weak compare.case
compare.nchotomy compare.split compare.split-asm compare.eq.refl compare.eq.simps
equal-compare-def
skip-red.simps skip-red.cases skip-red.induct
skip-black-def
compare-height.simps

108.10 union and intersection of sorted associative lists

context *ord* **begin**

function *sunion-with* :: $('a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow ('a \times 'b)$ *list* $\Rightarrow ('a \times 'b)$ *list* $\Rightarrow$ $('a \times 'b)$ *list*
where
sunion-with f $((k, v) \# as)$ $((k', v') \# bs) =$
$($*if* $k > k'$ *then* $(k', v') \#$ *sunion-with f* $((k, v) \# as)$ *bs*
else if $k < k'$ *then* $(k, v) \#$ *sunion-with f as* $((k', v') \# bs)$
else $(k, f\ k\ v\ v') \#$ *sunion-with f as bs*$)$
| *sunion-with f* [] *bs* = *bs*
| *sunion-with f as* [] = *as*
by *pat-completeness auto*
termination by *lexicographic-order*

function *sinter-with* :: $('a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow ('a \times 'b)$ *list* $\Rightarrow ('a \times 'b)$ *list* $\Rightarrow$ $('a \times 'b)$ *list*
where
sinter-with f $((k, v) \# as)$ $((k', v') \# bs) =$
$($*if* $k > k'$ *then sinter-with f* $((k, v) \# as)$ *bs*
else if $k < k'$ *then sinter-with f as* $((k', v') \# bs)$
else $(k, f\ k\ v\ v') \#$ *sinter-with f as bs*$)$
| *sinter-with f* [] - = []
| *sinter-with f* - [] = []
by *pat-completeness auto*
termination by *lexicographic-order*

end

declare *ord.sunion-with.simps* [*code*] *ord.sinter-with.simps*[*code*]

context *linorder* **begin**

lemma *set-fst-sunion-with*:
set (*map fst* (*sunion-with f xs ys*)) = *set* (*map fst xs*) ∪ *set* (*map fst ys*)
by(*induct f xs ys rule*: *sunion-with.induct*) *auto*

lemma *sorted-sunion-with* [*simp*]:
⟦ *sorted* (*map fst xs*); *sorted* (*map fst ys*) ⟧
⟹ *sorted* (*map fst* (*sunion-with f xs ys*))
by(*induct f xs ys rule*: *sunion-with.induct*)
(*auto simp add*: *sorted-Cons set-fst-sunion-with simp del*: *set-map*)

lemma *distinct-sunion-with* [*simp*]:
⟦ *distinct* (*map fst xs*); *distinct* (*map fst ys*); *sorted* (*map fst xs*); *sorted* (*map fst ys*) ⟧
⟹ *distinct* (*map fst* (*sunion-with f xs ys*))
proof(*induct f xs ys rule*: *sunion-with.induct*)
case (*1 f k v xs k′ v′ ys*)

have ⟦ ¬ $k < k'$; ¬ $k' < k$ ⟧ $\Longrightarrow k = k'$ **by** *simp*
thus *?case* **using** *1*
by(*auto simp add*: *set-fst-sunion-with sorted-Cons simp del*: *set-map*)
qed *simp-all*

lemma *map-of-sunion-with*:
⟦ *sorted* (*map fst xs*); *sorted* (*map fst ys*) ⟧
$\Longrightarrow$ *map-of* (*sunion-with f xs ys*) *k* =
(*case map-of xs k of None* ⇒ *map-of ys k*
| *Some v* ⇒ *case map-of ys k of None* ⇒ *Some v*
| *Some w* ⇒ *Some* (*f k v w*))
by(*induct f xs ys rule*: *sunion-with.induct*)(*auto simp add*: *sorted-Cons split*: *option.split dest*: *map-of-SomeD bspec*)

lemma *set-fst-sinter-with* [*simp*]:
⟦ *sorted* (*map fst xs*); *sorted* (*map fst ys*) ⟧
$\Longrightarrow$ *set* (*map fst* (*sinter-with f xs ys*)) = *set* (*map fst xs*) ∩ *set* (*map fst ys*)
by(*induct f xs ys rule*: *sinter-with.induct*)(*auto simp add*: *sorted-Cons simp del*: *set-map*)

lemma *set-fst-sinter-with-subset1*:
set (*map fst* (*sinter-with f xs ys*)) ⊆ *set* (*map fst xs*)
by(*induct f xs ys rule*: *sinter-with.induct*) *auto*

lemma *set-fst-sinter-with-subset2*:
set (*map fst* (*sinter-with f xs ys*)) ⊆ *set* (*map fst ys*)
by(*induct f xs ys rule*: *sinter-with.induct*)(*auto simp del*: *set-map*)

lemma *sorted-sinter-with* [*simp*]:
⟦ *sorted* (*map fst xs*); *sorted* (*map fst ys*) ⟧
$\Longrightarrow$ *sorted* (*map fst* (*sinter-with f xs ys*))
by(*induct f xs ys rule*: *sinter-with.induct*)(*auto simp add*: *sorted-Cons simp del*: *set-map*)

lemma *distinct-sinter-with* [*simp*]:
⟦ *distinct* (*map fst xs*); *distinct* (*map fst ys*) ⟧
$\Longrightarrow$ *distinct* (*map fst* (*sinter-with f xs ys*))
proof(*induct f xs ys rule*: *sinter-with.induct*)
case (*1 f k v as k' v' bs*)
have ⟦ ¬ $k < k'$; ¬ $k' < k$ ⟧ $\Longrightarrow k = k'$ **by** *simp*
thus *?case* **using** *1 set-fst-sinter-with-subset1*[*of f as bs*]
set-fst-sinter-with-subset2[*of f as bs*]
by(*auto simp del*: *set-map*)
qed *simp-all*

lemma *map-of-sinter-with*:
⟦ *sorted* (*map fst xs*); *sorted* (*map fst ys*) ⟧
$\Longrightarrow$ *map-of* (*sinter-with f xs ys*) *k* =
(*case map-of xs k of None* ⇒ *None* | *Some v* ⇒ *map-option* (*f k v*) (*map-of ys*

k))
apply(*induct f xs ys rule*: *sinter-with.induct*)
apply(*auto simp add*: *sorted-Cons map-option-case split*: *option.splits dest*: *map-of-SomeD bspec*)
done

end

lemma *distinct-map-of-rev*: *distinct* (*map fst xs*) $\Longrightarrow$ *map-of* (*rev xs*) = *map-of xs*
by(*induct xs*)(*auto 4 3 simp add*: *map-add-def intro*!: *ext split*: *option.split intro*: *rev-image-eqI*)

lemma *map-map-filter*:
map f (*List.map-filter g xs*) = *List.map-filter* (*map-option f* $\circ$ *g*) *xs*
by(*auto simp add*: *List.map-filter-def*)

lemma *map-filter-map-option-const*:
List.map-filter (λx. *map-option* (λy. *f x*) (*g* (*f x*))) *xs* = *filter* (λx. *g x* $\neq$ *None*) (*map f xs*)
by(*auto simp add*: *map-filter-def filter-map o-def*)

lemma *set-map-filter*: *set* (*List.map-filter P xs*) = *the* ' (*P* ' *set xs* − {*None*})
by(*auto simp add*: *List.map-filter-def intro*: *rev-image-eqI*)

context *ord* **begin**

definition *rbt-union-with-key* :: ('*a* ⇒ '*b* ⇒ '*b* ⇒ '*b*) ⇒ ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt*
where
rbt-union-with-key f t1 t2 =
(*case RBT-Impl.compare-height t1 t1 t2 t2*
of compare.EQ ⇒ *rbtreeify* (*sunion-with f* (*entries t1*) (*entries t2*))
| *compare.LT* ⇒ *fold* (*rbt-insert-with-key* (λ*k v w*. *f k w v*)) *t1 t2*
| *compare.GT* ⇒ *fold* (*rbt-insert-with-key f*) *t2 t1*)

definition *rbt-union-with* **where**
rbt-union-with f = *rbt-union-with-key* (λ-. *f*)

definition *rbt-union* **where**
rbt-union = *rbt-union-with-key* (%- - *rv*. *rv*)

definition *rbt-inter-with-key* :: ('*a* ⇒ '*b* ⇒ '*b* ⇒ '*b*) ⇒ ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt*
where
rbt-inter-with-key f t1 t2 =
(*case RBT-Impl.compare-height t1 t1 t2 t2*
of compare.EQ ⇒ *rbtreeify* (*sinter-with f* (*entries t1*) (*entries t2*))
| *compare.LT* ⇒ *rbtreeify* (*List.map-filter* (λ(*k*, *v*). *map-option* (λ*w*. (*k*, *f k v*

w)) (*rbt-lookup t2 k*)) (*entries t1*))
  | *compare.GT* ⇒ *rbtreeify* (*List.map-filter* (λ(*k*, *v*). *map-option* (λ*w*. (*k*, *f k w v*)) (*rbt-lookup t1 k*)) (*entries t2*)))

definition *rbt-inter-with* **where**
rbt-inter-with f = *rbt-inter-with-key* (λ-. *f*)

definition *rbt-inter* **where**
rbt-inter = *rbt-inter-with-key* (λ- - *rv*. *rv*)

end

context *linorder* **begin**

lemma *rbt-sorted-entries-right-unique*:
⟦ (*k*, *v*) ∈ *set* (*entries t*); (*k*, *v'*) ∈ *set* (*entries t*);
  rbt-sorted t ⟧ ⟹ *v* = *v'*
by(*auto dest*!: *distinct-entries inj-onD*[**where** *x*=(*k*, *v*) **and** *y*=(*k*, *v'*)] *simp add*: *distinct-map*)

lemma *rbt-sorted-fold-rbt-insertwk*:
rbt-sorted t ⟹ *rbt-sorted* (*List.fold* (λ(*k*, *v*). *rbt-insert-with-key f k v*) *xs t*)
by(*induct xs rule*: *rev-induct*)(*auto simp add*: *rbt-insertwk-rbt-sorted*)

lemma *is-rbt-fold-rbt-insertwk*:
assumes *is-rbt t1*
shows *is-rbt* (*fold* (*rbt-insert-with-key f*) *t2 t1*)
proof −
def *xs* ≡ *entries t2*
from *assms* **show** *?thesis* **unfolding** *fold-def xs-def*[*symmetric*]
  by(*induct xs rule*: *rev-induct*)(*auto simp add*: *rbt-insertwk-is-rbt*)
qed

lemma *rbt-lookup-fold-rbt-insertwk*:
assumes *t1*: *rbt-sorted t1* **and** *t2*: *rbt-sorted t2*
shows *rbt-lookup* (*fold* (*rbt-insert-with-key f*) *t1 t2*) *k* =
(*case rbt-lookup t1 k of None* ⇒ *rbt-lookup t2 k*
 | *Some v* ⇒ *case rbt-lookup t2 k of None* ⇒ *Some v*
   | *Some w* ⇒ *Some* (*f k w v*))
proof −
def *xs* ≡ *entries t1*
hence *dt1*: *distinct* (*map fst xs*) **using** *t1* **by**(*simp add*: *distinct-entries*)
with *t2* **show** *?thesis*
  unfolding *fold-def map-of-entries*[*OF t1*, *symmetric*]
    xs-def[*symmetric*] *distinct-map-of-rev*[*OF dt1*, *symmetric*]
  apply(*induct xs rule*: *rev-induct*)
  apply(*auto simp add*: *rbt-lookup-rbt-insertwk rbt-sorted-fold-rbt-insertwk split*: *option.splits*)
  apply(*auto simp add*: *distinct-map-of-rev intro*: *rev-image-eqI*)

done
qed

lemma *is-rbt-rbt-unionwk* [*simp*]:
⟦ *is-rbt t1*; *is-rbt t2* ⟧ ⟹ *is-rbt* (*rbt-union-with-key f t1 t2*)
by(*simp add*: *rbt-union-with-key-def Let-def is-rbt-fold-rbt-insertwk is-rbt-rbtreeify rbt-sorted-entries distinct-entries split*: *compare.split*)

lemma *rbt-lookup-rbt-unionwk*:
⟦ *rbt-sorted t1*; *rbt-sorted t2* ⟧
⟹ *rbt-lookup* (*rbt-union-with-key f t1 t2*) *k* =
(*case rbt-lookup t1 k of None* ⇒ *rbt-lookup t2 k*
| *Some v* ⇒ *case rbt-lookup t2 k of None* ⇒ *Some v*
| *Some w* ⇒ *Some* (*f k v w*))
by(*auto simp add*: *rbt-union-with-key-def Let-def rbt-lookup-fold-rbt-insertwk rbt-sorted-entries distinct-entries map-of-sunion-with map-of-entries rbt-lookup-rbtreeify split*: *option.split compare.split*)

lemma *rbt-unionw-is-rbt*: ⟦ *is-rbt lt*; *is-rbt rt* ⟧ ⟹ *is-rbt* (*rbt-union-with f lt rt*)
by(*simp add*: *rbt-union-with-def*)

lemma *rbt-union-is-rbt*: ⟦ *is-rbt lt*; *is-rbt rt* ⟧ ⟹ *is-rbt* (*rbt-union lt rt*)
by(*simp add*: *rbt-union-def*)

lemma *rbt-lookup-rbt-union*:
⟦ *rbt-sorted s*; *rbt-sorted t* ⟧ ⟹
rbt-lookup (*rbt-union s t*) = *rbt-lookup s* ++ *rbt-lookup t*
by(*rule ext*)(*simp add*: *rbt-lookup-rbt-unionwk rbt-union-def map-add-def split*: *option.split*)

lemma *rbt-interwk-is-rbt* [*simp*]:
⟦ *rbt-sorted t1*; *rbt-sorted t2* ⟧ ⟹ *is-rbt* (*rbt-inter-with-key f t1 t2*)
by(*auto simp add*: *rbt-inter-with-key-def Let-def map-map-filter split-def o-def option.map-comp map-filter-map-option-const sorted-filter*[**where** *f*=*id*, *simplified*] *rbt-sorted-entries distinct-entries intro*: *is-rbt-rbtreeify split*: *compare.split*)

lemma *rbt-interw-is-rbt*:
⟦ *rbt-sorted t1*; *rbt-sorted t2* ⟧ ⟹ *is-rbt* (*rbt-inter-with f t1 t2*)
by(*simp add*: *rbt-inter-with-def*)

lemma *rbt-inter-is-rbt*:
⟦ *rbt-sorted t1*; *rbt-sorted t2* ⟧ ⟹ *is-rbt* (*rbt-inter t1 t2*)
by(*simp add*: *rbt-inter-def*)

lemma *rbt-lookup-rbt-interwk*:
⟦ *rbt-sorted t1*; *rbt-sorted t2* ⟧
⟹ *rbt-lookup* (*rbt-inter-with-key f t1 t2*) *k* =
(*case rbt-lookup t1 k of None* ⇒ *None*
| *Some v* ⇒ *case rbt-lookup t2 k of None* ⇒ *None*

| *Some w* ⇒ *Some* (*f k v w*))
by(*auto 4 3 simp add*: *rbt-inter-with-key-def Let-def map-of-entries*[*symmetric*] *rbt-lookup-rbtreeify map-map-filter split-def o-def option.map-comp map-filter-map-option-const sorted-filter*[**where** *f*=*id*, *simplified*] *rbt-sorted-entries distinct-entries map-of-sinter-with map-of-eq-None-iff set-map-filter split*: *option.split compare.split intro*: *rev-image-eqI dest*: *rbt-sorted-entries-right-unique*)

lemma *rbt-lookup-rbt-inter*:
⟦ *rbt-sorted t1*; *rbt-sorted t2* ⟧
⟹ *rbt-lookup* (*rbt-inter t1 t2*) = *rbt-lookup t2* |' *dom* (*rbt-lookup t1*)
by(*auto simp add*: *rbt-inter-def rbt-lookup-rbt-interwk restrict-map-def split*: *option.split*)

end

108.11 Code generator setup

lemmas [*code*] =
ord.rbt-less-prop
ord.rbt-greater-prop
ord.rbt-sorted.simps
ord.rbt-lookup.simps
ord.is-rbt-def
ord.rbt-ins.simps
ord.rbt-insert-with-key-def
ord.rbt-insertw-def
ord.rbt-insert-def
ord.rbt-del-from-left.simps
ord.rbt-del-from-right.simps
ord.rbt-del.simps
ord.rbt-delete-def
ord.sunion-with.simps
ord.sinter-with.simps
ord.rbt-union-with-key-def
ord.rbt-union-with-def
ord.rbt-union-def
ord.rbt-inter-with-key-def
ord.rbt-inter-with-def
ord.rbt-inter-def
ord.rbt-map-entry.simps
ord.rbt-bulkload-def

More efficient implementations for *entries* and *keys*

definition *gen-entries* ::
(('*a* × '*b*) × ('*a*, '*b*) *rbt*) *list* ⇒ ('*a*, '*b*) *rbt* ⇒ ('*a* × '*b*) *list*
where
gen-entries kvts t = *entries t* @ *concat* (*map* (λ(*kv*, *t*). *kv* # *entries t*) *kvts*)

lemma *gen-entries-simps* [*simp*, *code*]:
gen-entries [] *Empty* = []

gen-entries ((*kv*, *t*) # *kvts*) *Empty* = *kv* # *gen-entries kvts t*
gen-entries kvts (*Branch c l k v r*) = *gen-entries* (((*k*, *v*), *r*) # *kvts*) *l*
by(*simp-all add*: *gen-entries-def*)

lemma *entries-code* [*code*]:
entries = *gen-entries* []
by(*simp add*: *gen-entries-def fun-eq-iff*)

definition *gen-keys* :: ('*a* × ('*a*, '*b*) *rbt*) *list* ⇒ ('*a*, '*b*) *rbt* ⇒ '*a list*
where *gen-keys kts t* = *RBT-Impl.keys t* @ *concat* (*List.map* (λ(*k*, *t*). *k* # *keys t*) *kts*)

lemma *gen-keys-simps* [*simp*, *code*]:
gen-keys [] *Empty* = []
gen-keys ((*k*, *t*) # *kts*) *Empty* = *k* # *gen-keys kts t*
gen-keys kts (*Branch c l k v r*) = *gen-keys* ((*k*, *r*) # *kts*) *l*
by(*simp-all add*: *gen-keys-def*)

lemma *keys-code* [*code*]:
keys = *gen-keys* []
by(*simp add*: *gen-keys-def fun-eq-iff*)

Restore original type constraints for constants

setup ‹
fold Sign.add-const-constraint
[(@{*const-name rbt-less*}, *SOME* @{*typ* ('*a* :: *order*) ⇒ ('*a*, '*b*) *rbt* ⇒ *bool*}),
(@{*const-name rbt-greater*}, *SOME* @{*typ* ('*a* :: *order*) ⇒ ('*a*, '*b*) *rbt* ⇒ *bool*}),
(@{*const-name rbt-sorted*}, *SOME* @{*typ* ('*a* :: *linorder*, '*b*) *rbt* ⇒ *bool*}),
(@{*const-name rbt-lookup*}, *SOME* @{*typ* ('*a* :: *linorder*, '*b*) *rbt* ⇒ '*a* ⇀ '*b*}),
(@{*const-name is-rbt*}, *SOME* @{*typ* ('*a* :: *linorder*, '*b*) *rbt* ⇒ *bool*}),
(@{*const-name rbt-ins*}, *SOME* @{*typ* ('*a*::*linorder* ⇒ '*b* ⇒ '*b* ⇒ '*b*) ⇒ '*a* ⇒ '*b* ⇒ ('*a*,'*b*) *rbt* ⇒ ('*a*,'*b*) *rbt*}),
(@{*const-name rbt-insert-with-key*}, *SOME* @{*typ* ('*a*::*linorder* ⇒ '*b* ⇒ '*b* ⇒ '*b*) ⇒ '*a* ⇒ '*b* ⇒ ('*a*,'*b*) *rbt* ⇒ ('*a*,'*b*) *rbt*}),
(@{*const-name rbt-insert-with*}, *SOME* @{*typ* ('*b* ⇒ '*b* ⇒ '*b*) ⇒ ('*a* :: *linorder*) ⇒ '*b* ⇒ ('*a*,'*b*) *rbt* ⇒ ('*a*,'*b*) *rbt*}),
(@{*const-name rbt-insert*}, *SOME* @{*typ* ('*a* :: *linorder*) ⇒ '*b* ⇒ ('*a*,'*b*) *rbt* ⇒ ('*a*,'*b*) *rbt*}),
(@{*const-name rbt-del-from-left*}, *SOME* @{*typ* ('*a*::*linorder*) ⇒ ('*a*,'*b*) *rbt* ⇒ '*a* ⇒ '*b* ⇒ ('*a*,'*b*) *rbt* ⇒ ('*a*,'*b*) *rbt*}),
(@{*const-name rbt-del-from-right*}, *SOME* @{*typ* ('*a*::*linorder*) ⇒ ('*a*,'*b*) *rbt* ⇒ '*a* ⇒ '*b* ⇒ ('*a*,'*b*) *rbt* ⇒ ('*a*,'*b*) *rbt*}),
(@{*const-name rbt-del*}, *SOME* @{*typ* ('*a*::*linorder*) ⇒ ('*a*,'*b*) *rbt* ⇒ ('*a*,'*b*) *rbt*}),
(@{*const-name rbt-delete*}, *SOME* @{*typ* ('*a*::*linorder*) ⇒ ('*a*,'*b*) *rbt* ⇒ ('*a*,'*b*) *rbt*}),
(@{*const-name rbt-union-with-key*}, *SOME* @{*typ* ('*a*::*linorder* ⇒ '*b* ⇒ '*b* ⇒ '*b*) ⇒ ('*a*,'*b*) *rbt* ⇒ ('*a*,'*b*) *rbt* ⇒ ('*a*,'*b*) *rbt*}),
(@{*const-name rbt-union-with*}, *SOME* @{*typ* ('*b* ⇒ '*b* ⇒ '*b*) ⇒ ('*a*::*linorder*,'*b*)

rbt ⇒ ('*a*,'*b*) *rbt* ⇒ ('*a*,'*b*) *rbt*}),
(@{*const-name rbt-union*}, *SOME* @{*typ* ('*a*::*linorder*,'*b*) *rbt* ⇒ ('*a*,'*b*) *rbt* ⇒ ('*a*,'*b*) *rbt*}),
(@{*const-name rbt-map-entry*}, *SOME* @{*typ* '*a*::*linorder* ⇒ ('*b* ⇒ '*b*) ⇒ ('*a*,'*b*) *rbt* ⇒ ('*a*,'*b*) *rbt*}),
(@{*const-name rbt-bulkload*}, *SOME* @{*typ* ('*a* × '*b*) *list* ⇒ ('*a*::*linorder*,'*b*) *rbt*})]
⟩

hide-const (**open**) *R B Empty entries keys fold gen-keys gen-entries*

end

109 Abstract type of RBT trees

theory *RBT*
imports *Main RBT-Impl*
begin

109.1 Type definition

typedef (**overloaded**) ('*a*, '*b*) *rbt* = {*t* :: ('*a*::*linorder*, '*b*) *RBT-Impl.rbt*. *is-rbt t*}
morphisms *impl-of RBT*
proof −
have *RBT-Impl.Empty* ∈ *?rbt* **by** *simp*
then show *?thesis* **..**
qed

lemma *rbt-eq-iff*:
t1 = *t2* ⟷ *impl-of t1* = *impl-of t2*
by (*simp add*: *impl-of-inject*)

lemma *rbt-eqI*:
impl-of t1 = *impl-of t2* ⟹ *t1* = *t2*
by (*simp add*: *rbt-eq-iff*)

lemma *is-rbt-impl-of* [*simp*, *intro*]:
is-rbt (*impl-of t*)
using *impl-of* [*of t*] **by** *simp*

lemma *RBT-impl-of* [*simp*, *code abstype*]:
RBT (*impl-of t*) = *t*
by (*simp add*: *impl-of-inverse*)

109.2 Primitive operations

setup-lifting *type-definition-rbt*

lift-definition *lookup* :: ('*a*::*linorder*, '*b*) *rbt* ⇒ '*a* ⇀ '*b* **is** *rbt-lookup* **.**

lift-definition *empty* :: ('*a*::*linorder*, '*b*) *rbt* **is** *RBT-Impl.Empty*
by (*simp add*: *empty-def*)

lift-definition *insert* :: '*a*::*linorder* ⇒ '*b* ⇒ ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt* **is** *rbt-insert*

by *simp*

lift-definition *delete* :: '*a*::*linorder* ⇒ ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt* **is** *rbt-delete*
by *simp*

lift-definition *entries* :: ('*a*::*linorder*, '*b*) *rbt* ⇒ ('*a* × '*b*) *list* **is** *RBT-Impl.entries*
.

lift-definition *keys* :: ('*a*::*linorder*, '*b*) *rbt* ⇒ '*a list* **is** *RBT-Impl.keys* **.**

lift-definition *bulkload* :: ('*a*::*linorder* × '*b*) *list* ⇒ ('*a*, '*b*) *rbt* **is** *rbt-bulkload* **..**

lift-definition *map-entry* :: '*a* ⇒ ('*b* ⇒ '*b*) ⇒ ('*a*::*linorder*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt*
is *rbt-map-entry*
by *simp*

lift-definition *map* :: ('*a* ⇒ '*b* ⇒ '*c*) ⇒ ('*a*::*linorder*, '*b*) *rbt* ⇒ ('*a*, '*c*) *rbt* **is**
RBT-Impl.map
by *simp*

lift-definition *fold* :: ('*a* ⇒ '*b* ⇒ '*c* ⇒ '*c*) ⇒ ('*a*::*linorder*, '*b*) *rbt* ⇒ '*c* ⇒ '*c* **is**
RBT-Impl.fold **.**

lift-definition *union* :: ('*a*::*linorder*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt* **is**
rbt-union
by (*simp add*: *rbt-union-is-rbt*)

lift-definition *foldi* :: ('*c* ⇒ *bool*) ⇒ ('*a* ⇒ '*b* ⇒ '*c* ⇒ '*c*) ⇒ ('*a* :: *linorder*, '*b*)
rbt ⇒ '*c* ⇒ '*c*
is *RBT-Impl.foldi* **.**

109.3 Derived operations

definition *is-empty* :: ('*a*::*linorder*, '*b*) *rbt* ⇒ *bool* **where**
[*code*]: *is-empty t* = (*case impl-of t of RBT-Impl.Empty* ⇒ *True* | - ⇒ *False*)

109.4 Abstract lookup properties

lemma *lookup-RBT*:
is-rbt t ⟹ *lookup* (*RBT t*) = *rbt-lookup t*
by (*simp add*: *lookup-def RBT-inverse*)

lemma *lookup-impl-of*:

rbt-lookup (*impl-of t*) = *lookup t*
by *transfer* (*rule refl*)

lemma *entries-impl-of*:
RBT-Impl.entries (*impl-of t*) = *entries t*
by *transfer* (*rule refl*)

lemma *keys-impl-of*:
RBT-Impl.keys (*impl-of t*) = *keys t*
by *transfer* (*rule refl*)

lemma *lookup-keys*:
dom (*lookup t*) = *set* (*keys t*)
by *transfer* (*simp add*: *rbt-lookup-keys*)

lemma *lookup-empty* [*simp*]:
lookup empty = *Map.empty*
by (*simp add*: *empty-def lookup-RBT fun-eq-iff*)

lemma *lookup-insert* [*simp*]:
lookup (*insert k v t*) = (*lookup t*)(*k* $\mapsto$ *v*)
by *transfer* (*rule rbt-lookup-rbt-insert*)

lemma *lookup-delete* [*simp*]:
lookup (*delete k t*) = (*lookup t*)(*k* := *None*)
by *transfer* (*simp add*: *rbt-lookup-rbt-delete restrict-complement-singleton-eq*)

lemma *map-of-entries* [*simp*]:
map-of (*entries t*) = *lookup t*
by *transfer* (*simp add*: *map-of-entries*)

lemma *entries-lookup*:
entries t1 = *entries t2* $\longleftrightarrow$ *lookup t1* = *lookup t2*
by *transfer* (*simp add*: *entries-rbt-lookup*)

lemma *lookup-bulkload* [*simp*]:
lookup (*bulkload xs*) = *map-of xs*
by *transfer* (*rule rbt-lookup-rbt-bulkload*)

lemma *lookup-map-entry* [*simp*]:
lookup (*map-entry k f t*) = (*lookup t*)(*k* := *map-option f* (*lookup t k*))
by *transfer* (*rule rbt-lookup-rbt-map-entry*)

lemma *lookup-map* [*simp*]:
lookup (*map f t*) *k* = *map-option* (*f k*) (*lookup t k*)
by *transfer* (*rule rbt-lookup-map*)

lemma *fold-fold*:
fold f t = *List.fold* (*case-prod f*) (*entries t*)

by *transfer* (*rule RBT-Impl.fold-def*)

lemma *impl-of-empty*:
impl-of empty = *RBT-Impl.Empty*
by *transfer* (*rule refl*)

lemma *is-empty-empty* [*simp*]:
is-empty t ⟷ *t* = *empty*
unfolding *is-empty-def* **by** *transfer* (*simp split*: *rbt.split*)

lemma *RBT-lookup-empty* [*simp*]:
rbt-lookup t = *Map.empty* ⟷ *t* = *RBT-Impl.Empty*
by (*cases t*) (*auto simp add*: *fun-eq-iff*)

lemma *lookup-empty-empty* [*simp*]:
lookup t = *Map.empty* ⟷ *t* = *empty*
by *transfer* (*rule RBT-lookup-empty*)

lemma *sorted-keys* [*iff*]:
sorted (*keys t*)
by *transfer* (*simp add*: *RBT-Impl.keys-def rbt-sorted-entries*)

lemma *distinct-keys* [*iff*]:
distinct (*keys t*)
by *transfer* (*simp add*: *RBT-Impl.keys-def distinct-entries*)

lemma *finite-dom-lookup* [*simp*, *intro*!]: *finite* (*dom* (*lookup t*))
by *transfer simp*

lemma *lookup-union*: *lookup* (*union s t*) = *lookup s* ++ *lookup t*
by *transfer* (*simp add*: *rbt-lookup-rbt-union*)

lemma *lookup-in-tree*: (*lookup t k* = *Some v*) = ((*k*, *v*) ∈ *set* (*entries t*))
by *transfer* (*simp add*: *rbt-lookup-in-tree*)

lemma *keys-entries*: (*k* ∈ *set* (*keys t*)) = (∃ *v*. (*k*, *v*) ∈ *set* (*entries t*))
by *transfer* (*simp add*: *keys-entries*)

lemma *fold-def-alt*:
fold f t = *List.fold* (*case-prod f*) (*entries t*)
by *transfer* (*auto simp*: *RBT-Impl.fold-def*)

lemma *distinct-entries*: *distinct* (*List.map fst* (*entries t*))
by *transfer* (*simp add*: *distinct-entries*)

lemma *non-empty-keys*: *t* ≠ *empty* ⟹ *keys t* ≠ []
by *transfer* (*simp add*: *non-empty-rbt-keys*)

lemma *keys-def-alt*:

keys t = *List.map fst* (*entries t*)
by *transfer* (*simp add*: *RBT-Impl.keys-def*)

109.5 Quickcheck generators

quickcheck-generator *rbt predicate*: *is-rbt constructors*: *empty*, *insert*

109.6 Hide implementation details

lifting-update *rbt.lifting*
lifting-forget *rbt.lifting*

hide-const (**open**) *impl-of empty lookup keys entries bulkload delete map fold union insert map-entry foldi*
is-empty
hide-fact (**open**) *empty-def lookup-def keys-def entries-def bulkload-def delete-def map-def fold-def*
union-def insert-def map-entry-def foldi-def is-empty-def

end

110 Implementation of mappings with Red-Black Trees

This theory defines abstract red-black trees as an efficient representation of finite maps, backed by the implementation in *RBT-Impl*.

110.1 Data type and invariant

The type ('*k*, '*v*) *RBT-Impl.rbt* denotes red-black trees with keys of type '*k* and values of type '*v*. To function properly, the key type musorted belong to the *linorder* class.

A value *t* of this type is a valid red-black tree if it satisfies the invariant *is-rbt t*. The abstract type ('*k*, '*v*) *RBT.rbt* always obeys this invariant, and for this reason you should only use this in our application. Going back to ('*k*, '*v*) *RBT-Impl.rbt* may be necessary in proofs if not yet proven properties about the operations must be established.

The interpretation function *RBT.lookup* returns the partial map represented by a red-black tree:

RBT.lookup::('*a*, '*b*) *RBT.rbt* ⇒ '*a* ⇒ '*b option*

This function should be used for reasoning about the semantics of the RBT operations. Furthermore, it implements the lookup functionality for the data structure: It is executable and the lookup is performed in $O(\log n)$.

110.2 Operations

Currently, the following operations are supported:

$RBT.empty::('a, 'b)\ RBT.rbt$

Returns the empty tree. $O(1)$

$RBT.insert::'a \Rightarrow 'b \Rightarrow ('a, 'b)\ RBT.rbt \Rightarrow ('a, 'b)\ RBT.rbt$

Updates the map at a given position. $O(\log n)$

$RBT.delete::'a \Rightarrow ('a, 'b)\ RBT.rbt \Rightarrow ('a, 'b)\ RBT.rbt$

Deletes a map entry at a given position. $O(\log n)$

$RBT.entries::('a, 'b)\ RBT.rbt \Rightarrow ('a \times 'b)\ list$

Return a corresponding key-value list for a tree.

$RBT.bulkload::('a \times 'b)\ list \Rightarrow ('a, 'b)\ RBT.rbt$

Builds a tree from a key-value list.

$RBT.map\text{-}entry::'a \Rightarrow ('b \Rightarrow 'b) \Rightarrow ('a, 'b)\ RBT.rbt \Rightarrow ('a, 'b)\ RBT.rbt$

Maps a single entry in a tree.

$RBT.map::('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('a, 'b)\ RBT.rbt \Rightarrow ('a, 'c)\ RBT.rbt$

Maps all values in a tree. $O(n)$

$RBT.fold::('a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'c) \Rightarrow ('a, 'b)\ RBT.rbt \Rightarrow 'c \Rightarrow 'c$

Folds over all entries in a tree. $O(n)$

110.3 Invariant preservation

is-rbt rbt.Empty (*Empty-is-rbt*)
is-rbt ?t $\Longrightarrow$ *is-rbt* (*rbt-insert ?k ?v ?t*) (*rbt-insert-is-rbt*)
is-rbt ?t $\Longrightarrow$ *is-rbt* (*rbt-delete ?k ?t*) (*delete-is-rbt*)
is-rbt (*rbt-bulkload ?xs*) (*bulkload-is-rbt*)
is-rbt (*rbt-map-entry ?k ?f ?t*) = *is-rbt ?t* (*map-entry-is-rbt*)
is-rbt (*RBT-Impl.map ?f ?t*) = *is-rbt ?t* (*map-is-rbt*)
⟦*is-rbt ?lt*; *is-rbt ?rt*⟧ $\Longrightarrow$ *is-rbt* (*rbt-union ?lt ?rt*) (*union-is-rbt*)

110.4 Map Semantics

lookup-empty

Mapping.lookup Mapping.empty ?k = None

lookup-insert

RBT.lookup (RBT.insert ?k ?v ?t) = RBT.lookup ?t(?k ↦ ?v)

lookup-delete

RBT.lookup (RBT.delete ?k ?t) = (RBT.lookup ?t)(?k := None)

lookup-bulkload

RBT.lookup (RBT.bulkload ?xs) = map-of ?xs

lookup-map

RBT.lookup (RBT.map ?f ?t) ?k = map-option (?f ?k) (RBT.lookup ?t ?k)

end

111 Implementation of sets using RBT trees

theory *RBT-Set*
imports *RBT Product-Lexorder*
begin

112 Definition of code datatype constructors

definition *Set* :: ('*a*::*linorder*, *unit*) *rbt* ⇒ '*a set*
where *Set t* = {*x* . *RBT.lookup t x* = *Some* ()}

definition *Coset* :: ('*a*::*linorder*, *unit*) *rbt* ⇒ '*a set*
where [*simp*]: *Coset t* = − *Set t*

113 Deletion of already existing code equations

lemma [*code*, *code del*]:
Set.empty = *Set.empty* **..**

lemma [*code*, *code del*]:
Set.is-empty = *Set.is-empty* **..**

lemma [*code*, *code del*]:
uminus-set-inst.uminus-set = *uminus-set-inst.uminus-set* **..**

lemma [*code*, *code del*]:
Set.member = *Set.member* ..

lemma [*code*, *code del*]:
Set.insert = *Set.insert* ..

lemma [*code*, *code del*]:
Set.remove = *Set.remove* ..

lemma [*code*, *code del*]:
UNIV = *UNIV* ..

lemma [*code*, *code del*]:
Set.filter = *Set.filter* ..

lemma [*code*, *code del*]:
image = *image* ..

lemma [*code*, *code del*]:
Set.subset-eq = *Set.subset-eq* ..

lemma [*code*, *code del*]:
Ball = *Ball* ..

lemma [*code*, *code del*]:
Bex = *Bex* ..

lemma [*code*, *code del*]:
can-select = *can-select* ..

lemma [*code*, *code del*]:
Set.union = *Set.union* ..

lemma [*code*, *code del*]:
minus-set-inst.minus-set = *minus-set-inst.minus-set* ..

lemma [*code*, *code del*]:
Set.inter = *Set.inter* ..

lemma [*code*, *code del*]:
card = *card* ..

lemma [*code*, *code del*]:
the-elem = *the-elem* ..

lemma [*code*, *code del*]:
Pow = *Pow* ..

lemma [*code*, *code del*]:
setsum = *setsum* ..

lemma [*code*, *code del*]:
setprod = *setprod* ..

lemma [*code*, *code del*]:
Product-Type.product = *Product-Type.product* ..

lemma [*code*, *code del*]:
Id-on = *Id-on* ..

lemma [*code*, *code del*]:
Image = *Image* ..

lemma [*code*, *code del*]:
trancl = *trancl* ..

lemma [*code*, *code del*]:
relcomp = *relcomp* ..

lemma [*code*, *code del*]:
wf = *wf* ..

lemma [*code*, *code del*]:
Min = *Min* ..

lemma [*code*, *code del*]:
Inf-fin = *Inf-fin* ..

lemma [*code*, *code del*]:
INFIMUM = *INFIMUM* ..

lemma [*code*, *code del*]:
Max = *Max* ..

lemma [*code*, *code del*]:
Sup-fin = *Sup-fin* ..

lemma [*code*, *code del*]:
SUPREMUM = *SUPREMUM* ..

lemma [*code*, *code del*]:
(*Inf* :: *'a set set* ⇒ *'a set*) = *Inf* ..

lemma [*code*, *code del*]:
(*Sup* :: *'a set set* ⇒ *'a set*) = *Sup* ..

lemma [*code*, *code del*]:

sorted-list-of-set = *sorted-list-of-set* **..**

lemma [*code*, *code del*]:
List.map-project = *List.map-project* **..**

lemma [*code*, *code del*]:
List.Bleast = *List.Bleast* **..**

114 Lemmas

114.1 Auxiliary lemmas

lemma [*simp*]: $x \neq Some\ () \longleftrightarrow x = None$
by (*auto simp*: *not-Some-eq*[*THEN iffD1*])

lemma *Set-set-keys*: *Set x* = *dom* (*RBT.lookup x*)
by (*auto simp*: *Set-def*)

lemma *finite-Set* [*simp*, *intro*!]: *finite* (*Set x*)
by (*simp add*: *Set-set-keys*)

lemma *set-keys*: *Set t* = *set*(*RBT.keys t*)
by (*simp add*: *Set-set-keys lookup-keys*)

114.2 fold and filter

lemma *finite-fold-rbt-fold-eq*:
assumes *comp-fun-commute f*
shows *Finite-Set.fold f A* (*set* (*RBT.entries t*)) = *RBT.fold* (*curry f*) *t A*
proof −
have ∗: *remdups* (*RBT.entries t*) = *RBT.entries t*
using *distinct-entries distinct-map* **by** (*auto intro*: *distinct-remdups-id*)
show *?thesis* **using** *assms* **by** (*auto simp*: *fold-def-alt comp-fun-commute.fold-set-fold-remdups* ∗)
qed

definition *fold-keys* :: ($'a$ :: *linorder* ⇒ $'b$ ⇒ $'b$) ⇒ ($'a$, -) *rbt* ⇒ $'b$ ⇒ $'b$
where [*code-unfold*]:*fold-keys f t A* = *RBT.fold* (λ*k* - *t*. *f k t*) *t A*

lemma *fold-keys-def-alt*:
fold-keys f t s = *List.fold f* (*RBT.keys t*) *s*
by (*auto simp*: *fold-map o-def split-def fold-def-alt keys-def-alt fold-keys-def*)

lemma *finite-fold-fold-keys*:
assumes *comp-fun-commute f*
shows *Finite-Set.fold f A* (*Set t*) = *fold-keys f t A*
using *assms*
proof −
interpret *comp-fun-commute f* **by** *fact*

have *set* (*RBT.keys t*) = *fst* ' (*set* (*RBT.entries t*)) **by** (*auto simp*: *fst-eq-Domain keys-entries*)
moreover have *inj-on fst* (*set* (*RBT.entries t*)) **using** *distinct-entries distinct-map* **by** *auto*
ultimately show *?thesis*
by (*auto simp add*: *set-keys fold-keys-def curry-def fold-image finite-fold-rbt-fold-eq*

comp-comp-fun-commute)
qed

definition *rbt-filter* :: ($'a$ :: *linorder* ⇒ *bool*) ⇒ ($'a$, $'b$) *rbt* ⇒ $'a$ *set* **where**
rbt-filter P t = *RBT.fold* (λ*k* - *A'*. *if P k then Set.insert k A' else A'*) *t* {}

lemma *Set-filter-rbt-filter*:
Set.filter P (*Set t*) = *rbt-filter P t*
by (*simp add*: *fold-keys-def Set-filter-fold rbt-filter-def*
finite-fold-fold-keys[*OF comp-fun-commute-filter-fold*])

114.3 foldi and Ball

lemma *Ball-False*: *RBT-Impl.fold* (λ*k v s. s* ∧ *P k*) *t False* = *False*
by (*induction t*) *auto*

lemma *rbt-foldi-fold-conj*:
RBT-Impl.foldi (λ*s. s* = *True*) (λ*k v s. s* ∧ *P k*) *t val* = *RBT-Impl.fold* (λ*k v s. s* ∧ *P k*) *t val*
proof (*induction t arbitrary*: *val*)
case (*Branch c t1*) **then show** *?case*
by (*cases RBT-Impl.fold* (λ*k v s. s* ∧ *P k*) *t1 True*) (*simp-all add*: *Ball-False*)
qed *simp*

lemma *foldi-fold-conj*: *RBT.foldi* (λ*s. s* = *True*) (λ*k v s. s* ∧ *P k*) *t val* = *fold-keys* (λ*k s. s* ∧ *P k*) *t val*
unfolding *fold-keys-def* **including** *rbt.lifting* **by** *transfer* (*rule rbt-foldi-fold-conj*)

114.4 foldi and Bex

lemma *Bex-True*: *RBT-Impl.fold* (λ*k v s. s* ∨ *P k*) *t True* = *True*
by (*induction t*) *auto*

lemma *rbt-foldi-fold-disj*:
RBT-Impl.foldi (λ*s. s* = *False*) (λ*k v s. s* ∨ *P k*) *t val* = *RBT-Impl.fold* (λ*k v s. s* ∨ *P k*) *t val*
proof (*induction t arbitrary*: *val*)
case (*Branch c t1*) **then show** *?case*
by (*cases RBT-Impl.fold* (λ*k v s. s* ∨ *P k*) *t1 False*) (*simp-all add*: *Bex-True*)
qed *simp*

lemma *foldi-fold-disj*: *RBT.foldi* (λ*s. s* = *False*) (λ*k v s. s* ∨ *P k*) *t val* = *fold-keys* (λ*k s. s* ∨ *P k*) *t val*

unfolding *fold-keys-def* **including** *rbt.lifting* **by** *transfer* (*rule rbt-foldi-fold-disj*)

114.5 folding over non empty trees and selecting the minimal and maximal element

definition *rbt-fold1-keys* :: ($'a \Rightarrow 'a \Rightarrow 'a$) $\Rightarrow$ ($'a$::*linorder*, $'b$) *RBT-Impl.rbt* $\Rightarrow$ $'a$
 where *rbt-fold1-keys f t* = *List.fold f* (*tl*(*RBT-Impl.keys t*)) (*hd*(*RBT-Impl.keys t*))

definition *rbt-min* :: ($'a$::*linorder*, *unit*) *RBT-Impl.rbt* $\Rightarrow$ $'a$
 where *rbt-min t* = *rbt-fold1-keys min t*

lemma *key-le-right*: *rbt-sorted* (*Branch c lt k v rt*) $\Longrightarrow$ ($\bigwedge x.\ x \in set$ (*RBT-Impl.keys rt*) $\Longrightarrow k \leq x$)
by (*auto simp*: *rbt-greater-prop less-imp-le*)

lemma *left-le-key*: *rbt-sorted* (*Branch c lt k v rt*) $\Longrightarrow$ ($\bigwedge x.\ x \in set$ (*RBT-Impl.keys lt*) $\Longrightarrow x \leq k$)
by (*auto simp*: *rbt-less-prop less-imp-le*)

lemma *fold-min-triv*:
 fixes *k* :: - :: *linorder*
 shows ($\forall x \in set\ xs.\ k \leq x$) $\Longrightarrow$ *List.fold min xs k* = *k*
by (*induct xs*) (*auto simp add*: *min-def*)

lemma *rbt-min-simps*:
 is-rbt (*Branch c RBT-Impl.Empty k v rt*) $\Longrightarrow$ *rbt-min* (*Branch c RBT-Impl.Empty k v rt*) = *k*
by (*auto intro*: *fold-min-triv dest*: *key-le-right is-rbt-rbt-sorted simp*: *rbt-fold1-keys-def rbt-min-def*)

fun *rbt-min-opt* **where**
 rbt-min-opt (*Branch c RBT-Impl.Empty k v rt*) = *k* |
 rbt-min-opt (*Branch c* (*Branch lc llc lk lv lrt*) *k v rt*) = *rbt-min-opt* (*Branch lc llc lk lv lrt*)

lemma *rbt-min-opt-Branch*:
 t1 $\neq$ *rbt.Empty* $\Longrightarrow$ *rbt-min-opt* (*Branch c t1 k* () *t2*) = *rbt-min-opt t1*
by (*cases t1*) *auto*

lemma *rbt-min-opt-induct* [*case-names empty left-empty left-non-empty*]:
 fixes *t* :: ($'a$ :: *linorder*, *unit*) *RBT-Impl.rbt*
 assumes *P rbt.Empty*
 assumes $\bigwedge$*color t1 a b t2*. *P t1* $\Longrightarrow$ *P t2* $\Longrightarrow$ *t1* = *rbt.Empty* $\Longrightarrow$ *P* (*Branch color t1 a b t2*)
 assumes $\bigwedge$*color t1 a b t2*. *P t1* $\Longrightarrow$ *P t2* $\Longrightarrow$ *t1* $\neq$ *rbt.Empty* $\Longrightarrow$ *P* (*Branch*

color t1 a b t2)
  shows *P t*
using *assms*
  apply (*induction t*)
  apply *simp*
  apply (*case-tac t1* = *rbt.Empty*)
  apply *simp-all*
done

lemma *rbt-min-opt-in-set*:
  fixes *t* :: (*'a* :: *linorder*, *unit*) *RBT-Impl.rbt*
  assumes $t \neq$ *rbt.Empty*
  shows *rbt-min-opt t* ∈ *set* (*RBT-Impl.keys t*)
using *assms* **by** (*induction t rule*: *rbt-min-opt.induct*) (*auto*)

lemma *rbt-min-opt-is-min*:
  fixes *t* :: (*'a* :: *linorder*, *unit*) *RBT-Impl.rbt*
  assumes *rbt-sorted t*
  assumes $t \neq$ *rbt.Empty*
  shows ⋀*y*. *y* ∈ *set* (*RBT-Impl.keys t*) ⟹ *y* ≥ *rbt-min-opt t*
using *assms*
proof (*induction t rule*: *rbt-min-opt-induct*)
  case *empty*
  then show *?case* **by** *simp*
next
  case *left-empty*
  then show *?case* **by** (*auto intro*: *key-le-right simp del*: *rbt-sorted.simps*)
next
  case (*left-non-empty c t1 k v t2 y*)
  then consider *y* = *k* | *y* ∈ *set* (*RBT-Impl.keys t1*) | *y* ∈ *set* (*RBT-Impl.keys t2*)
    by *auto*
  then show *?case*
  proof *cases*
    case *1*
    with *left-non-empty* **show** *?thesis*
      by (*auto simp add*: *rbt-min-opt-Branch intro*: *left-le-key rbt-min-opt-in-set*)
  next
    case *2*
    with *left-non-empty* **show** *?thesis*
      by (*auto simp add*: *rbt-min-opt-Branch*)
  next
    case *y*: *3*
    have *rbt-min-opt t1* ≤ *k*
      using *left-non-empty* **by** (*simp add*: *left-le-key rbt-min-opt-in-set*)
    moreover have *k* ≤ *y*
      using *left-non-empty y* **by** (*simp add*: *key-le-right*)
    ultimately show *?thesis*
      using *left-non-empty y* **by** (*simp add*: *rbt-min-opt-Branch*)

qed
qed

lemma *rbt-min-eq-rbt-min-opt*:
assumes $t \neq$ *RBT-Impl.Empty*
assumes *is-rbt t*
shows *rbt-min t* = *rbt-min-opt t*
proof −
from *assms* **have** *hd* (*RBT-Impl.keys t*) # *tl* (*RBT-Impl.keys t*) = *RBT-Impl.keys t* **by** (*cases t*) *simp-all*
with *assms* **show** *?thesis*
by (*simp add*: *rbt-min-def rbt-fold1-keys-def rbt-min-opt-is-min Min.set-eq-fold* [*symmetric*] *Min-eqI rbt-min-opt-in-set*)
qed

definition *rbt-max* :: ('*a*::*linorder*, *unit*) *RBT-Impl.rbt* ⇒ '*a*
where *rbt-max t* = *rbt-fold1-keys max t*

lemma *fold-max-triv*:
fixes *k* :: - :: *linorder*
shows (∀ *x*∈*set xs*. *x* ≤ *k*) ⟹ *List.fold max xs k* = *k*
by (*induct xs*) (*auto simp add*: *max-def*)

lemma *fold-max-rev-eq*:
fixes *xs* :: ('*a* :: *linorder*) *list*
assumes *xs* ≠ []
shows *List.fold max* (*tl xs*) (*hd xs*) = *List.fold max* (*tl* (*rev xs*)) (*hd* (*rev xs*))
using *assms* **by** (*simp add*: *Max.set-eq-fold* [*symmetric*])

lemma *rbt-max-simps*:
assumes *is-rbt* (*Branch c lt k v RBT-Impl.Empty*)
shows *rbt-max* (*Branch c lt k v RBT-Impl.Empty*) = *k*
proof −
have *List.fold max* (*tl* (*rev*(*RBT-Impl.keys lt* @ [*k*]))) (*hd* (*rev*(*RBT-Impl.keys lt* @ [*k*]))) = *k*
using *assms* **by** (*auto intro*!: *fold-max-triv dest*!: *left-le-key is-rbt-rbt-sorted*)
then show *?thesis* **by** (*auto simp add*: *rbt-max-def rbt-fold1-keys-def fold-max-rev-eq*)
qed

fun *rbt-max-opt* **where**
rbt-max-opt (*Branch c lt k v RBT-Impl.Empty*) = *k* |
rbt-max-opt (*Branch c lt k v* (*Branch rc rlc rk rv rrt*)) = *rbt-max-opt* (*Branch rc rlc rk rv rrt*)

lemma *rbt-max-opt-Branch*:
t2 ≠ *rbt.Empty* ⟹ *rbt-max-opt* (*Branch c t1 k* () *t2*) = *rbt-max-opt t2*
by (*cases t2*) *auto*

lemma *rbt-max-opt-induct* [*case-names empty right-empty right-non-empty*]:
fixes *t* :: (*'a* :: *linorder*, *unit*) *RBT-Impl.rbt*
assumes *P rbt.Empty*
assumes $\bigwedge$*color t1 a b t2*. *P t1* $\Longrightarrow$ *P t2* $\Longrightarrow$ *t2* = *rbt.Empty* $\Longrightarrow$ *P* (*Branch color t1 a b t2*)
assumes $\bigwedge$*color t1 a b t2*. *P t1* $\Longrightarrow$ *P t2* $\Longrightarrow$ *t2* $\neq$ *rbt.Empty* $\Longrightarrow$ *P* (*Branch color t1 a b t2*)
shows *P t*
using *assms*
apply (*induction t*)
apply *simp*
apply (*case-tac t2* = *rbt.Empty*)
apply *simp-all*
done

lemma *rbt-max-opt-in-set*:
fixes *t* :: (*'a* :: *linorder*, *unit*) *RBT-Impl.rbt*
assumes *t* $\neq$ *rbt.Empty*
shows *rbt-max-opt t* $\in$ *set* (*RBT-Impl.keys t*)
using *assms* **by** (*induction t rule*: *rbt-max-opt.induct*) (*auto*)

lemma *rbt-max-opt-is-max*:
fixes *t* :: (*'a* :: *linorder*, *unit*) *RBT-Impl.rbt*
assumes *rbt-sorted t*
assumes *t* $\neq$ *rbt.Empty*
shows $\bigwedge$*y*. *y* $\in$ *set* (*RBT-Impl.keys t*) $\Longrightarrow$ *y* $\leq$ *rbt-max-opt t*
using *assms*
proof (*induction t rule*: *rbt-max-opt-induct*)
case *empty*
then show *?case* **by** *simp*
next
case *right-empty*
then show *?case* **by** (*auto intro*: *left-le-key simp del*: *rbt-sorted.simps*)
next
case (*right-non-empty c t1 k v t2 y*)
then consider *y* = *k* | *y* $\in$ *set* (*RBT-Impl.keys t2*) | *y* $\in$ *set* (*RBT-Impl.keys t1*)
by *auto*
then show *?case*
proof *cases*
case *1*
with *right-non-empty* **show** *?thesis*
by (*auto simp add*: *rbt-max-opt-Branch intro*: *key-le-right rbt-max-opt-in-set*)
next
case *2*
with *right-non-empty* **show** *?thesis*
by (*auto simp add*: *rbt-max-opt-Branch*)
next

case *y*: *3*
have *rbt-max-opt t2* ≥ *k*
using *right-non-empty* **by** (*simp add*: *key-le-right rbt-max-opt-in-set*)
moreover have *y* ≤ *k*
using *right-non-empty y* **by** (*simp add*: *left-le-key*)
ultimately show *?thesis*
using *right-non-empty* **by** (*simp add*: *rbt-max-opt-Branch*)
qed
qed

lemma *rbt-max-eq-rbt-max-opt*:
assumes *t* ≠ *RBT-Impl.Empty*
assumes *is-rbt t*
shows *rbt-max t* = *rbt-max-opt t*
proof −
from *assms* **have** *hd* (*RBT-Impl.keys t*) # *tl* (*RBT-Impl.keys t*) = *RBT-Impl.keys t* **by** (*cases t*) *simp-all*
with *assms* **show** *?thesis*
by (*simp add*: *rbt-max-def rbt-fold1-keys-def rbt-max-opt-is-max Max.set-eq-fold* [*symmetric*] *Max-eqI rbt-max-opt-in-set*)
qed

context includes *rbt.lifting* **begin**
lift-definition *fold1-keys* :: (′*a* ⇒ ′*a* ⇒ ′*a*) ⇒ (′*a*::*linorder*, ′*b*) *rbt* ⇒ ′*a*
is *rbt-fold1-keys* **.**

lemma *fold1-keys-def-alt*:
fold1-keys f t = *List.fold f* (*tl* (*RBT.keys t*)) (*hd* (*RBT.keys t*))
by *transfer* (*simp add*: *rbt-fold1-keys-def*)

lemma *finite-fold1-fold1-keys*:
assumes *semilattice f*
assumes ¬ *RBT.is-empty t*
shows *semilattice-set.F f* (*Set t*) = *fold1-keys f t*
proof −
from ‹*semilattice f*› **interpret** *semilattice-set f* **by** (*rule semilattice-set.intro*)
show *?thesis* **using** *assms*
by (*auto simp*: *fold1-keys-def-alt set-keys fold-def-alt non-empty-keys set-eq-fold* [*symmetric*])
qed

lift-definition *r-min* :: (′*a* :: *linorder*, *unit*) *rbt* ⇒ ′*a* **is** *rbt-min* **.**

lift-definition *r-min-opt* :: (′*a* :: *linorder*, *unit*) *rbt* ⇒ ′*a* **is** *rbt-min-opt* **.**

lemma *r-min-alt-def*: *r-min t* = *fold1-keys min t*
by *transfer* (*simp add*: *rbt-min-def*)

lemma *r-min-eq-r-min-opt*:
  assumes ¬ (*RBT.is-empty t*)
  shows *r-min t* = *r-min-opt t*
using *assms* **unfolding** *is-empty-empty* **by** *transfer* (*auto intro*: *rbt-min-eq-rbt-min-opt*)

lemma *fold-keys-min-top-eq*:
  fixes *t* :: (*'a* :: {*linorder*, *bounded-lattice-top*}, *unit*) *rbt*
  assumes ¬ (*RBT.is-empty t*)
  shows *fold-keys min t top* = *fold1-keys min t*
proof −
  have ∗: $\bigwedge$*t*. *RBT-Impl.keys t* ≠ [] ⟹ *List.fold min* (*RBT-Impl.keys t*) *top* =
    List.fold min (*hd*(*RBT-Impl.keys t*) # *tl*(*RBT-Impl.keys t*)) *top*
    by (*simp add*: *hd-Cons-tl*[*symmetric*])
  { fix *x* :: - :: {*linorder*, *bounded-lattice-top*} **and** *xs*
    have *List.fold min* (*x*#*xs*) *top* = *List.fold min xs x*
    by (*simp add*: *inf-min*[*symmetric*])
  } note ∗∗ = *this*
  show *?thesis* **using** *assms*
    unfolding *fold-keys-def-alt fold1-keys-def-alt is-empty-empty*
    apply *transfer*
    apply (*case-tac t*)
    apply *simp*
    apply (*subst* ∗)
    apply *simp*
    apply (*subst* ∗∗)
    apply *simp*
  done
qed

lift-definition *r-max* :: (*'a* :: *linorder*, *unit*) *rbt* ⇒ *'a* **is** *rbt-max* **.**

lift-definition *r-max-opt* :: (*'a* :: *linorder*, *unit*) *rbt* ⇒ *'a* **is** *rbt-max-opt* **.**

lemma *r-max-alt-def*: *r-max t* = *fold1-keys max t*
by *transfer* (*simp add*: *rbt-max-def*)

lemma *r-max-eq-r-max-opt*:
  assumes ¬ (*RBT.is-empty t*)
  shows *r-max t* = *r-max-opt t*
using *assms* **unfolding** *is-empty-empty* **by** *transfer* (*auto intro*: *rbt-max-eq-rbt-max-opt*)

lemma *fold-keys-max-bot-eq*:
  fixes *t* :: (*'a* :: {*linorder*, *bounded-lattice-bot*}, *unit*) *rbt*

assumes ¬ (*RBT.is-empty t*)
shows *fold-keys max t bot* = *fold1-keys max t*
proof −
have ∗: ⋀*t*. *RBT-Impl.keys t* ≠ [] ⟹ *List.fold max* (*RBT-Impl.keys t*) *bot* =
List.fold max (*hd*(*RBT-Impl.keys t*) # *tl*(*RBT-Impl.keys t*)) *bot*
by (*simp add*: *hd-Cons-tl*[*symmetric*])
{ **fix** *x* :: - :: {*linorder*, *bounded-lattice-bot*} **and** *xs*
have *List.fold max* (*x*#*xs*) *bot* = *List.fold max xs x*
by (*simp add*: *sup-max*[*symmetric*])
} **note** ∗∗ = *this*
show *?thesis* **using** *assms*
unfolding *fold-keys-def-alt fold1-keys-def-alt is-empty-empty*
apply *transfer*
apply (*case-tac t*)
apply *simp*
apply (*subst* ∗)
apply *simp*
apply (*subst* ∗∗)
apply *simp*
done
qed

end

115 Code equations

code-datatype *Set Coset*

declare *list.set*[*code*]

lemma *empty-Set* [*code*]:
Set.empty = *Set RBT.empty*
by (*auto simp*: *Set-def*)

lemma *UNIV-Coset* [*code*]:
UNIV = *Coset RBT.empty*
by (*auto simp*: *Set-def*)

lemma *is-empty-Set* [*code*]:
Set.is-empty (*Set t*) = *RBT.is-empty t*
unfolding *Set.is-empty-def* **by** (*auto simp*: *fun-eq-iff Set-def intro*: *lookup-empty-empty*[*THEN iffD1*])

lemma *compl-code* [*code*]:
− *Set xs* = *Coset xs*
− *Coset xs* = *Set xs*
by (*simp-all add*: *Set-def*)

lemma *member-code* [*code*]:

x ∈ (*Set t*) = (*RBT.lookup t x* = *Some* ())
x ∈ (*Coset t*) = (*RBT.lookup t x* = *None*)
by (*simp-all add*: *Set-def*)

lemma *insert-code* [*code*]:
Set.insert x (*Set t*) = *Set* (*RBT.insert x* () *t*)
Set.insert x (*Coset t*) = *Coset* (*RBT.delete x t*)
by (*auto simp*: *Set-def*)

lemma *remove-code* [*code*]:
Set.remove x (*Set t*) = *Set* (*RBT.delete x t*)
Set.remove x (*Coset t*) = *Coset* (*RBT.insert x* () *t*)
by (*auto simp*: *Set-def*)

lemma *union-Set* [*code*]:
Set t ∪ *A* = *fold-keys Set.insert t A*
proof −
interpret *comp-fun-idem Set.insert*
by (*fact comp-fun-idem-insert*)
from *finite-fold-fold-keys*[*OF* ‹*comp-fun-commute Set.insert*›]
show *?thesis* **by** (*auto simp add*: *union-fold-insert*)
qed

lemma *inter-Set* [*code*]:
A ∩ *Set t* = *rbt-filter* (λ*k*. *k* ∈ *A*) *t*
by (*simp add*: *inter-Set-filter Set-filter-rbt-filter*)

lemma *minus-Set* [*code*]:
A − *Set t* = *fold-keys Set.remove t A*
proof −
interpret *comp-fun-idem Set.remove*
by (*fact comp-fun-idem-remove*)
from *finite-fold-fold-keys*[*OF* ‹*comp-fun-commute Set.remove*›]
show *?thesis* **by** (*auto simp add*: *minus-fold-remove*)
qed

lemma *union-Coset* [*code*]:
Coset t ∪ *A* = − *rbt-filter* (λ*k*. *k* ∉ *A*) *t*
proof −
have ∗: $\bigwedge A\ B.\ (-A \cup B) = -(-B \cap A)$ **by** *blast*
show *?thesis* **by** (*simp del*: *boolean-algebra-class.compl-inf add*: ∗ *inter-Set*)
qed

lemma *union-Set-Set* [*code*]:
Set t1 ∪ *Set t2* = *Set* (*RBT.union t1 t2*)
by (*auto simp add*: *lookup-union map-add-Some-iff Set-def*)

lemma *inter-Coset* [*code*]:
A ∩ *Coset t* = *fold-keys Set.remove t A*

by (*simp add*: *Diff-eq* [*symmetric*] *minus-Set*)

lemma *inter-Coset-Coset* [*code*]:
  Coset t1 ∩ *Coset t2* = *Coset* (*RBT.union t1 t2*)
by (*auto simp add*: *lookup-union map-add-Some-iff Set-def*)

lemma *minus-Coset* [*code*]:
  A − *Coset t* = *rbt-filter* (λ*k*. *k* ∈ *A*) *t*
by (*simp add*: *inter-Set*[*simplified Int-commute*])

lemma *filter-Set* [*code*]:
  Set.filter P (*Set t*) = (*rbt-filter P t*)
by (*auto simp add*: *Set-filter-rbt-filter*)

lemma *image-Set* [*code*]:
  image f (*Set t*) = *fold-keys* (λ*k A*. *Set.insert* (*f k*) *A*) *t* {}
proof −
  have *comp-fun-commute* (λ*k*. *Set.insert* (*f k*))
    by *standard auto*
  then show *?thesis*
    by (*auto simp add*: *image-fold-insert intro*!: *finite-fold-fold-keys*)
qed

lemma *Ball-Set* [*code*]:
  Ball (*Set t*) *P* ⟷ *RBT.foldi* (λ*s*. *s* = *True*) (λ*k v s*. *s* ∧ *P k*) *t True*
proof −
  have *comp-fun-commute* (λ*k s*. *s* ∧ *P k*)
    by *standard auto*
  then show *?thesis*
    by (*simp add*: *foldi-fold-conj*[*symmetric*] *Ball-fold finite-fold-fold-keys*)
qed

lemma *Bex-Set* [*code*]:
  Bex (*Set t*) *P* ⟷ *RBT.foldi* (λ*s*. *s* = *False*) (λ*k v s*. *s* ∨ *P k*) *t False*
proof −
  have *comp-fun-commute* (λ*k s*. *s* ∨ *P k*)
    by *standard auto*
  then show *?thesis*
    by (*simp add*: *foldi-fold-disj*[*symmetric*] *Bex-fold finite-fold-fold-keys*)
qed

lemma *subset-code* [*code*]:
  Set t ≤ *B* ⟷ (∀ *x*∈*Set t*. *x* ∈ *B*)
  A ≤ *Coset t* ⟷ (∀ *y*∈*Set t*. *y* ∉ *A*)
by *auto*

lemma *subset-Coset-empty-Set-empty* [*code*]:
  Coset t1 ≤ *Set t2* ⟷ (*case* (*RBT.impl-of t1*, *RBT.impl-of t2*) *of*
    (*rbt.Empty*, *rbt.Empty*) => *False* |

(-, -) => *Code.abort* (*STR* ''*non-empty-trees*'') (λ-. *Coset t1* $\leq$ *Set t2*))
proof −
have ∗: $\bigwedge$*t*. *RBT.impl-of t* = *rbt.Empty* $\Longrightarrow$ *t* = *RBT rbt.Empty*
by (*subst*(*asm*) *RBT-inverse*[*symmetric*]) (*auto simp*: *impl-of-inject*)
have ∗∗: *eq-onp is-rbt rbt.Empty rbt.Empty* **unfolding** *eq-onp-def* **by** *simp*
show *?thesis*
by (*auto simp*: *Set-def lookup.abs-eq*[*OF* ∗∗] *dest*!: ∗ *split*: *rbt.split*)
qed

A frequent case – avoid intermediate sets

lemma [*code-unfold*]:
Set t1 ⊆ *Set t2* ⟷ *RBT.foldi* (λ*s*. *s* = *True*) (λ*k v s*. *s* ∧ *k* ∈ *Set t2*) *t1 True*
by (*simp add*: *subset-code Ball-Set*)

lemma *card-Set* [*code*]:
card (*Set t*) = *fold-keys* (λ- *n*. *n* + *1*) *t 0*
by (*auto simp add*: *card.eq-fold intro*: *finite-fold-fold-keys comp-fun-commute-const*)

lemma *setsum-Set* [*code*]:
setsum f (*Set xs*) = *fold-keys* (*plus o f*) *xs 0*
proof −
have *comp-fun-commute* (λ*x*. *op* + (*f x*))
by *standard* (*auto simp*: *ac-simps*)
then show *?thesis*
by (*auto simp add*: *setsum.eq-fold finite-fold-fold-keys o-def*)
qed

lemma *the-elem-set* [*code*]:
fixes *t* :: ('*a* :: *linorder*, *unit*) *rbt*
shows *the-elem* (*Set t*) = (*case RBT.impl-of t of*
(*Branch RBT-Impl.B RBT-Impl.Empty x* () *RBT-Impl.Empty*) ⇒ *x*
| - ⇒ *Code.abort* (*STR* ''*not-a-singleton-tree*'') (λ-. *the-elem* (*Set t*)))
proof −
{
fix *x* :: '*a* :: *linorder*
let *?t* = *Branch RBT-Impl.B RBT-Impl.Empty x* () *RBT-Impl.Empty*
have ∗:*?t* ∈ {*t*. *is-rbt t*} **unfolding** *is-rbt-def* **by** *auto*
then have ∗∗:*eq-onp is-rbt ?t ?t* **unfolding** *eq-onp-def* **by** *auto*

have *RBT.impl-of t* = *?t* $\Longrightarrow$ *the-elem* (*Set t*) = *x*
by (*subst*(*asm*) *RBT-inverse*[*symmetric*, *OF* ∗])
(*auto simp*: *Set-def the-elem-def lookup.abs-eq*[*OF* ∗∗] *impl-of-inject*)
}
then show *?thesis*
by(*auto split*: *rbt.split unit.split color.split*)
qed

lemma *Pow-Set* [*code*]: *Pow* (*Set t*) = *fold-keys* (λ*x A*. *A* ∪ *Set.insert x* ‘ *A*) *t* {{}}

by (*simp add*: *Pow-fold finite-fold-fold-keys*[*OF comp-fun-commute-Pow-fold*])

lemma *product-Set* [*code*]:
Product-Type.product (*Set t1*) (*Set t2*) =
fold-keys (λ*x A*. *fold-keys* (λ*y*. *Set.insert* (*x*, *y*)) *t2 A*) *t1* {}
proof −
have ∗: *comp-fun-commute* (λ*y*. *Set.insert* (*x*, *y*)) **for** *x*
by *standard auto*
show *?thesis* **using** *finite-fold-fold-keys*[*OF comp-fun-commute-product-fold*, *of Set t2* {} *t1*]
by (*simp add*: *product-fold Product-Type.product-def finite-fold-fold-keys*[*OF* ∗])
qed

lemma *Id-on-Set* [*code*]: *Id-on* (*Set t*) = *fold-keys* (λ*x*. *Set.insert* (*x*, *x*)) *t* {}
proof −
have *comp-fun-commute* (λ*x*. *Set.insert* (*x*, *x*))
by *standard auto*
then show *?thesis*
by (*auto simp add*: *Id-on-fold intro*!: *finite-fold-fold-keys*)
qed

lemma *Image-Set* [*code*]:
(*Set t*) `` *S* = *fold-keys* (λ(*x*,*y*) *A*. *if x* ∈ *S then Set.insert y A else A*) *t* {}
by (*auto simp add*: *Image-fold finite-fold-fold-keys*[*OF comp-fun-commute-Image-fold*])

lemma *trancl-set-ntrancl* [*code*]:
trancl (*Set t*) = *ntrancl* (*card* (*Set t*) − *1*) (*Set t*)
by (*simp add*: *finite-trancl-ntranl*)

lemma *relcomp-Set*[*code*]:
(*Set t1*) *O* (*Set t2*) = *fold-keys*
(λ(*x*,*y*) *A*. *fold-keys* (λ(*w*,*z*) *A*′. *if y* = *w then Set.insert* (*x*,*z*) *A*′ *else A*′) *t2 A*) *t1* {}
proof −
interpret *comp-fun-idem Set.insert*
by (*fact comp-fun-idem-insert*)
have ∗: ⋀*x y*. *comp-fun-commute* (λ(*w*, *z*) *A*′. *if y* = *w then Set.insert* (*x*, *z*) *A*′ *else A*′)
by *standard* (*auto simp add*: *fun-eq-iff*)
show *?thesis*
using *finite-fold-fold-keys*[*OF comp-fun-commute-relcomp-fold*, *of Set t2* {} *t1*]
by (*simp add*: *relcomp-fold finite-fold-fold-keys*[*OF* ∗])
qed

lemma *wf-set* [*code*]:
wf (*Set t*) = *acyclic* (*Set t*)
by (*simp add*: *wf-iff-acyclic-if-finite*)

lemma *Min-fin-set-fold* [*code*]:

Min (*Set t*) =
(*if RBT.is-empty t*
then Code.abort (*STR ''not-non-empty-tree''*) (λ-. *Min* (*Set t*))
else r-min-opt t)
proof −
have ∗: *semilattice* (*min* :: *'a* ⇒ *'a* ⇒ *'a*) **..**
with *finite-fold1-fold1-keys* [*OF* ∗, *folded Min-def*]
show *?thesis*
by (*simp add*: *r-min-alt-def r-min-eq-r-min-opt* [*symmetric*])
qed

lemma *Inf-fin-set-fold* [*code*]:
Inf-fin (*Set t*) = *Min* (*Set t*)
by (*simp add*: *inf-min Inf-fin-def Min-def*)

lemma *Inf-Set-fold*:
fixes *t* :: (*'a* :: {*linorder*, *complete-lattice*}, *unit*) *rbt*
shows *Inf* (*Set t*) = (*if RBT.is-empty t then top else r-min-opt t*)
proof −
have *comp-fun-commute* (*min* :: *'a* ⇒ *'a* ⇒ *'a*)
by *standard* (*simp add*: *fun-eq-iff ac-simps*)
then have *t* ≠ *RBT.empty* ⟹ *Finite-Set.fold min top* (*Set t*) = *fold1-keys min t*
by (*simp add*: *finite-fold-fold-keys fold-keys-min-top-eq*)
then show *?thesis*
by (*auto simp add*: *Inf-fold-inf inf-min empty-Set*[*symmetric*]
r-min-eq-r-min-opt[*symmetric*] *r-min-alt-def*)
qed

definition *Inf'* :: *'a* :: {*linorder*, *complete-lattice*} *set* ⇒ *'a* **where** [*code del*]: *Inf' x* = *Inf x*
declare *Inf'-def*[*symmetric*, *code-unfold*]
declare *Inf-Set-fold*[*folded Inf'-def*, *code*]

lemma *INF-Set-fold* [*code*]:
fixes *f* :: - ⇒ *'a::complete-lattice*
shows *INFIMUM* (*Set t*) *f* = *fold-keys* (*inf* ∘ *f*) *t top*
proof −
have *comp-fun-commute* ((*inf* :: *'a* ⇒ *'a* ⇒ *'a*) ∘ *f*)
by *standard* (*auto simp add*: *fun-eq-iff ac-simps*)
then show *?thesis*
by (*auto simp*: *INF-fold-inf finite-fold-fold-keys*)
qed

lemma *Max-fin-set-fold* [*code*]:
Max (*Set t*) =
(*if RBT.is-empty t*
then Code.abort (*STR ''not-non-empty-tree''*) (λ-. *Max* (*Set t*))
else r-max-opt t)

proof −
  have ∗: *semilattice* $(max :: 'a \Rightarrow 'a \Rightarrow 'a)$ **..**
  with *finite-fold1-fold1-keys* [*OF* ∗, *folded Max-def*]
  show *?thesis*
    by (*simp add*: *r-max-alt-def r-max-eq-r-max-opt* [*symmetric*])
qed

lemma *Sup-fin-set-fold* [*code*]:
  Sup-fin (*Set t*) = *Max* (*Set t*)
by (*simp add*: *sup-max Sup-fin-def Max-def*)

lemma *Sup-Set-fold*:
  fixes *t* :: ('*a* :: {*linorder*, *complete-lattice*}, *unit*) *rbt*
  shows *Sup* (*Set t*) = (*if RBT.is-empty t then bot else r-max-opt t*)
proof −
  have *comp-fun-commute* $(max :: 'a \Rightarrow 'a \Rightarrow 'a)$
    by *standard* (*simp add*: *fun-eq-iff ac-simps*)
  then have $t \neq$ *RBT.empty* $\Longrightarrow$ *Finite-Set.fold max bot* (*Set t*) = *fold1-keys max t*
    by (*simp add*: *finite-fold-fold-keys fold-keys-max-bot-eq*)
  then show *?thesis*
    by (*auto simp add*: *Sup-fold-sup sup-max empty-Set*[*symmetric*]
      r-max-eq-r-max-opt[*symmetric*] *r-max-alt-def*)
qed

definition *Sup'* :: '*a* :: {*linorder*,*complete-lattice*} *set* ⇒ '*a*
  where [*code del*]: *Sup' x* = *Sup x*
declare *Sup'-def*[*symmetric*, *code-unfold*]
declare *Sup-Set-fold*[*folded Sup'-def*, *code*]

lemma *SUP-Set-fold* [*code*]:
  fixes *f* :: - ⇒ '*a*::*complete-lattice*
  shows *SUPREMUM* (*Set t*) *f* = *fold-keys* (*sup* ∘ *f*) *t bot*
proof −
  have *comp-fun-commute* $((sup :: 'a \Rightarrow 'a \Rightarrow 'a) \circ f)$
    by *standard* (*auto simp add*: *fun-eq-iff ac-simps*)
  then show *?thesis*
    by (*auto simp*: *SUP-fold-sup finite-fold-fold-keys*)
qed

lemma *sorted-list-set*[*code*]: *sorted-list-of-set* (*Set t*) = *RBT.keys t*
  by (*auto simp add*: *set-keys intro*: *sorted-distinct-set-unique*)

lemma *Bleast-code* [*code*]:
  Bleast (*Set t*) *P* =
    (*case filter P* (*RBT.keys t*) *of*
      x # *xs* ⇒ *x*
    | [] ⇒ *abort-Bleast* (*Set t*) *P*)
proof (*cases filter P* (*RBT.keys t*))

```
  case Nil
  thus ?thesis by (simp add: Bleast-def abort-Bleast-def)
next
  case (Cons x ys)
  have (LEAST x. x ∈ Set t ∧ P x) = x
  proof (rule Least-equality)
    show x ∈ Set t ∧ P x
      using Cons[symmetric]
      by (auto simp add: set-keys Cons-eq-filter-iff)
    next
      fix y
      assume y ∈ Set t ∧ P y
      then show x ≤ y
        using Cons[symmetric]
        by(auto simp add: set-keys Cons-eq-filter-iff)
          (metis sorted-Cons sorted-append sorted-keys)
  qed
  thus ?thesis using Cons by (simp add: Bleast-def)
qed

hide-const (open) RBT-Set.Set RBT-Set.Coset

end
```

116 Refute

```
theory Refute
imports Main
keywords refute :: diag and refute-params :: thy-decl
begin

ML-file refute.ML

refute-params
 [itself = 1,
  minsize = 1,
  maxsize = 8,
  maxvars = 10000,
  maxtime = 60,
  satsolver = auto,
  no-assms = false]

(* ------------------------------------------------------------------- *)
(* REFUTE                                                              *)
(*                                                                     *)
(* We use a SAT solver to search for a (finite) model that refutes a given  *)
(* HOL formula.                                                        *)
(* ------------------------------------------------------------------- *)
```

```
(* ------------------------------------------------------------------------- *)
(* NOTE                                                                      *)
(*                                                                           *)
(* I strongly recommend that you install a stand-alone SAT solver if you     *)
(* want to use 'refute'.  For details see 'HOL/Tools/sat_solver.ML'.  If you *)
(* have installed (a supported version of) zChaff, simply set 'ZCHAFF_HOME'  *)
(* in 'etc/settings'.                                                        *)
(* ------------------------------------------------------------------------- *)

(* ------------------------------------------------------------------------- *)
(* USAGE                                                                     *)
(*                                                                           *)
(* See the file 'HOL/ex/Refute_Examples.thy' for examples.  The supported    *)
(* parameters are explained below.                                           *)
(* ------------------------------------------------------------------------- *)

(* ------------------------------------------------------------------------- *)
(* CURRENT LIMITATIONS                                                       *)
(*                                                                           *)
(* 'refute' currently accepts formulas of higher-order predicate logic (with *)
(* equality), including free/bound/schematic variables, lambda abstractions, *)
(* sets and set membership, "arbitrary", "The", "Eps", records and           *)
(* inductively defined sets.  Constants are unfolded automatically, and sort *)
(* axioms are added as well.  Other, user-asserted axioms however are        *)
(* ignored.  Inductive datatypes and recursive functions are supported, but  *)
(* may lead to spurious countermodels.                                       *)
(*                                                                           *)
(* The (space) complexity of the algorithm is non-elementary.                *)
(*                                                                           *)
(* Schematic type variables are not supported.                               *)
(* ------------------------------------------------------------------------- *)

(* ------------------------------------------------------------------------- *)
(* PARAMETERS                                                                *)
(*                                                                           *)
(* The following global parameters are currently supported (and required,   *)
(* except for "expect"):                                                     *)
(*                                                                           *)
(* Name          Type    Description                                         *)
(*                                                                           *)
(* "minsize"     int     Only search for models with size at least           *)
(*                       'minsize'.                                          *)
(* "maxsize"     int     If >0, only search for models with size at most     *)
(*                       'maxsize'.                                          *)
(* "maxvars"     int     If >0, use at most 'maxvars' boolean variables      *)
(*                       when transforming the term into a propositional     *)
(*                       formula.                                            *)
(* "maxtime"     int     If >0, terminate after at most 'maxtime' seconds.   *)
(*                       This value is ignored under some ML compilers.      *)
```

```
(* "satsolver"   string  Name of the SAT solver to be used.                *)
(* "no_assms"    bool    If "true", assumptions in structured proofs are   *)
(*                       not considered.                                   *)
(* "expect"      string  Expected result ("genuine", "potential", "none", or *)
(*                       "unknown").                                       *)
(*                                                                         *)
(* The size of particular types can be specified in the form type=size    *)
(* (where 'type' is a string, and 'size' is an int).  Examples:           *)
(* "'a"=1                                                                  *)
(* "List.list"=2                                                           *)
(* ----------------------------------------------------------------------- *)

(* ----------------------------------------------------------------------- *)
(* FILES                                                                   *)
(*                                                                         *)
(* HOL/Tools/prop_logic.ML     Propositional logic                         *)
(* HOL/Tools/sat_solver.ML     SAT solvers                                 *)
(* HOL/Tools/refute.ML         Translation HOL -> propositional logic and  *)
(*                             Boolean assignment -> HOL model             *)
(* HOL/Refute.thy              This file: loads the ML files, basic setup, *)
(*                             documentation                               *)
(* HOL/SAT.thy                 Sets default parameters                     *)
(* HOL/ex/Refute_Examples.thy  Examples                                    *)
(* ----------------------------------------------------------------------- *)
```

end

117 TFL: recursive function definitions

theory *Old-Recdef*
imports *Main*
keywords
recdef :: *thy-decl* **and**
permissive congs hints
begin

117.1 Lemmas for TFL

lemma *tfl-wf-induct*: *ALL R. wf R −−>*
(ALL P. (ALL x. (ALL y. (y,x):R −−> P y) −−> P x) −−> (ALL x. P x))
apply *clarify*
apply (*rule-tac r* = *R* **and** *P* = *P* **and** *a* = *x* **in** *wf-induct*, *assumption*, *blast*)
done

lemma *tfl-cut-def*: *cut f r x* ≡ (λ*y. if* (*y*,*x*) ∈ *r then f y else undefined*)
unfolding *cut-def* **.**

lemma *tfl-cut-apply*: *ALL f R.* (*x*,*a*):*R −−>* (*cut f R a*)(*x*) = *f*(*x*)

apply *clarify*
apply (*rule cut-apply*, *assumption*)
done

lemma *tfl-wfrec*:
ALL M R f. (*f*=*wfrec R M*) −−> *wf R* −−> (*ALL x*. *f x* = *M* (*cut f R x*) *x*)
apply *clarify*
apply (*erule wfrec*)
done

lemma *tfl-eq-True*: (*x* = *True*) −−> *x*
by *blast*

lemma *tfl-rev-eq-mp*: (*x* = *y*) −−> *y* −−> *x*
by *blast*

lemma *tfl-simp-thm*: (*x* −−> *y*) −−> (*x* = *x*′) −−> (*x*′ −−> *y*)
by *blast*

lemma *tfl-P-imp-P-iff-True*: *P* ==> *P* = *True*
by *blast*

lemma *tfl-imp-trans*: (*A* −−> *B*) ==> (*B* −−> *C*) ==> (*A* −−> *C*)
by *blast*

lemma *tfl-disj-assoc*: (*a* ∨ *b*) ∨ *c* == *a* ∨ (*b* ∨ *c*)
by *simp*

lemma *tfl-disjE*: *P* ∨ *Q* ==> *P* −−> *R* ==> *Q* −−> *R* ==> *R*
by *blast*

lemma *tfl-exE*: ∃ *x*. *P x* ==> ∀ *x*. *P x* −−> *Q* ==> *Q*
by *blast*

ML-file *old-recdef.ML*

117.2 Rule setup

lemmas [*recdef-simp*] =
inv-image-def
measure-def
lex-prod-def
same-fst-def
less-Suc-eq [*THEN iffD2*]

lemmas [*recdef-cong*] =
if-cong let-cong image-cong INF-cong SUP-cong bex-cong ball-cong imp-cong
map-cong filter-cong takeWhile-cong dropWhile-cong foldl-cong foldr-cong

lemmas [*recdef-wf*] =
wf-trancl
wf-less-than
wf-lex-prod
wf-inv-image
wf-measure
wf-measures
wf-pred-nat
wf-same-fst
wf-empty

end

118 Syntactic classes for bitwise operations

theory *Bits*
imports *Main*
begin

class *bit* =
fixes *bitNOT* :: $'a \Rightarrow 'a$ (*NOT* - [*70*] *71*)
and *bitAND* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixr** *AND 64*)
and *bitOR* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixr** *OR 59*)
and *bitXOR* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixr** *XOR 59*)

We want the bitwise operations to bind slightly weaker than + and −, but ~~ to bind slightly stronger than ∗.

Testing and shifting operations.

class *bits* = *bit* +
fixes *test-bit* :: $'a \Rightarrow nat \Rightarrow bool$ (**infixl** !! *100*)
and *lsb* :: $'a \Rightarrow bool$
and *set-bit* :: $'a \Rightarrow nat \Rightarrow bool \Rightarrow 'a$
and *set-bits* :: $(nat \Rightarrow bool) \Rightarrow 'a$ (**binder** *BITS 10*)
and *shiftl* :: $'a \Rightarrow nat \Rightarrow 'a$ (**infixl** << *55*)
and *shiftr* :: $'a \Rightarrow nat \Rightarrow 'a$ (**infixl** >> *55*)

class *bitss* = *bits* +
fixes *msb* :: $'a \Rightarrow bool$

end

119 Bit operations in $\mathcal{Z}_{\in}$

theory *Bits-Bit*
imports *Bits* ~~/*src*/*HOL*/*Library*/*Bit*
begin

instantiation *bit* :: *bit*
begin

primrec *bitNOT-bit* **where**
  NOT 0 = (*1*::*bit*)
  | *NOT 1* = (*0*::*bit*)

primrec *bitAND-bit* **where**
  0 AND y = (*0*::*bit*)
  | *1 AND y* = (*y*::*bit*)

primrec *bitOR-bit* **where**
  0 OR y = (*y*::*bit*)
  | *1 OR y* = (*1*::*bit*)

primrec *bitXOR-bit* **where**
  0 XOR y = (*y*::*bit*)
  | *1 XOR y* = (*NOT y* :: *bit*)

instance ..

end

lemmas *bit-simps* =
  bitNOT-bit.simps bitAND-bit.simps bitOR-bit.simps bitXOR-bit.simps

lemma *bit-extra-simps* [*simp*]:
  x AND 0 = (*0*::*bit*)
  x AND 1 = (*x*::*bit*)
  x OR 1 = (*1*::*bit*)
  x OR 0 = (*x*::*bit*)
  x XOR 1 = *NOT* (*x*::*bit*)
  x XOR 0 = (*x*::*bit*)
  by (*cases x*, *auto*)+

lemma *bit-ops-comm*:
  (*x*::*bit*) *AND y* = *y AND x*
  (*x*::*bit*) *OR y* = *y OR x*
  (*x*::*bit*) *XOR y* = *y XOR x*
  by (*cases y*, *auto*)+

lemma *bit-ops-same* [*simp*]:
  (*x*::*bit*) *AND x* = *x*
  (*x*::*bit*) *OR x* = *x*
  (*x*::*bit*) *XOR x* = *0*
  by (*cases x*, *auto*)+

lemma *bit-not-not* [*simp*]: *NOT* (*NOT* (*x*::*bit*)) = *x*
  by (*cases x*) *auto*

lemma *bit-or-def*: $(b{::}bit)\ OR\ c = NOT\ (NOT\ b\ AND\ NOT\ c)$
by (*induct b*, *simp-all*)

lemma *bit-xor-def*: $(b{::}bit)\ XOR\ c = (b\ AND\ NOT\ c)\ OR\ (NOT\ b\ AND\ c)$
by (*induct b*, *simp-all*)

lemma *bit-NOT-eq-1-iff* [*simp*]: $NOT\ (b{::}bit) = 1 \longleftrightarrow b = 0$
by (*induct b*, *simp-all*)

lemma *bit-AND-eq-1-iff* [*simp*]: $(a{::}bit)\ AND\ b = 1 \longleftrightarrow a = 1 \wedge b = 1$
by (*induct a*, *simp-all*)

end

120 Useful Numerical Lemmas

theory *Misc-Numeric*
imports *Main*
begin

lemma *mod-2-neq-1-eq-eq-0*:
fixes $k :: int$
shows $k\ mod\ 2 \neq 1 \longleftrightarrow k\ mod\ 2 = 0$
by (*fact not-mod-2-eq-1-eq-0*)

lemma *z1pmod2*:
fixes $b :: int$
shows $(2 * b + 1)\ mod\ 2 = (1{::}int)$
by *arith*

lemma *diff-le-eq'*:
$a - b \leq c \longleftrightarrow a \leq b + (c{::}int)$
by *arith*

lemma *emep1*:
fixes $n\ d :: int$
shows $even\ n \Longrightarrow even\ d \Longrightarrow 0 \leq d \Longrightarrow (n + 1)\ mod\ d = (n\ mod\ d) + 1$
by (*auto simp add*: *pos-zmod-mult-2 add.commute dvd-def*)

lemma *int-mod-ge*:
$a < n \Longrightarrow 0 < (n :: int) \Longrightarrow a \leq a\ mod\ n$
by (*metis dual-order.trans le-cases mod-pos-pos-trivial pos-mod-conj*)

lemma *int-mod-ge'*:
$b < 0 \Longrightarrow 0 < (n :: int) \Longrightarrow b + n \leq b\ mod\ n$
by (*metis add-less-same-cancel2 int-mod-ge mod-add-self2*)

lemma *int-mod-le'*:

$(0::int) \leq b - n \Longrightarrow b\ mod\ n \leq b - n$
by (*metis minus-mod-self2 zmod-le-nonneg-dividend*)

lemma *zless2*:
$0 < (2 :: int)$
by (*fact zero-less-numeral*)

lemma *zless2p*:
$0 < (2\ \hat{}\ n :: int)$
by *arith*

lemma *zle2p*:
$0 \leq (2\ \hat{}\ n :: int)$
by *arith*

lemma *m1mod2k*:
$- 1\ mod\ 2\ \hat{}\ n = (2\ \hat{}\ n - 1 :: int)$
using *zless2p* **by** (*rule zmod-minus1*)

lemma *p1mod22k'*:
fixes *b* :: *int*
shows $(1 + 2 * b)\ mod\ (2 * 2\ \hat{}\ n) = 1 + 2 * (b\ mod\ 2\ \hat{}\ n)$
using *zle2p* **by** (*rule pos-zmod-mult-2*)

lemma *p1mod22k*:
fixes *b* :: *int*
shows $(2 * b + 1)\ mod\ (2 * 2\ \hat{}\ n) = 2 * (b\ mod\ 2\ \hat{}\ n) + 1$
by (*simp add*: *p1mod22k' add.commute*)

lemma *int-mod-lem*:
(*0* :: *int*) < *n* ==> (*0* <= *b* & *b* < *n*) = (*b mod n* = *b*)
apply *safe*
apply (*erule* (*1*) *mod-pos-pos-trivial*)
apply (*erule-tac* [!] *subst*)
apply *auto*
done

end

121 Integers as implict bit strings

theory *Bit-Representation*
imports *Misc-Numeric*
begin

121.1 Constructors and destructors for binary integers

definition *Bit* :: *int* ⇒ *bool* ⇒ *int* (**infixl** *BIT 90*)
where

k BIT b = (*if b then 1 else 0*) + *k* + *k*

lemma *Bit-B0*:
k BIT False = *k* + *k*
by (*unfold Bit-def*) *simp*

lemma *Bit-B1*:
k BIT True = *k* + *k* + *1*
by (*unfold Bit-def*) *simp*

lemma *Bit-B0-2t*: *k BIT False* = *2* ∗ *k*
by (*rule trans*, *rule Bit-B0*) *simp*

lemma *Bit-B1-2t*: *k BIT True* = *2* ∗ *k* + *1*
by (*rule trans*, *rule Bit-B1*) *simp*

definition *bin-last* :: *int* ⇒ *bool*
where
bin-last w ⟷ *w mod 2* = *1*

lemma *bin-last-odd*:
bin-last = *odd*
by (*rule ext*) (*simp add*: *bin-last-def even-iff-mod-2-eq-zero*)

definition *bin-rest* :: *int* ⇒ *int*
where
bin-rest w = *w div 2*

lemma *bin-rl-simp* [*simp*]:
bin-rest w BIT bin-last w = *w*
unfolding *bin-rest-def bin-last-def Bit-def*
using *mod-div-equality* [*of w 2*]
by (*cases w mod 2* = *0*, *simp-all*)

lemma *bin-rest-BIT* [*simp*]: *bin-rest* (*x BIT b*) = *x*
unfolding *bin-rest-def Bit-def*
by (*cases b*, *simp-all*)

lemma *bin-last-BIT* [*simp*]: *bin-last* (*x BIT b*) = *b*
unfolding *bin-last-def Bit-def*
by (*cases b*) *simp-all*

lemma *BIT-eq-iff* [*iff*]: *u BIT b* = *v BIT c* ⟷ *u* = *v* ∧ *b* = *c*
apply (*auto simp add*: *Bit-def*)
apply *arith*
apply *arith*
done

lemma *BIT-bin-simps* [*simp*]:

numeral k BIT False = *numeral* (*Num.Bit0 k*)
numeral k BIT True = *numeral* (*Num.Bit1 k*)
(− *numeral k*) *BIT False* = − *numeral* (*Num.Bit0 k*)
(− *numeral k*) *BIT True* = − *numeral* (*Num.BitM k*)
unfolding *numeral.simps numeral-BitM*
unfolding *Bit-def*
by (*simp-all del*: *arith-simps add-numeral-special diff-numeral-special*)

lemma *BIT-special-simps* [*simp*]:
shows *0 BIT False* = *0* **and** *0 BIT True* = *1*
and *1 BIT False* = *2* **and** *1 BIT True* = *3*
and (− *1*) *BIT False* = − *2* **and** (− *1*) *BIT True* = − *1*
unfolding *Bit-def* **by** *simp-all*

lemma *Bit-eq-0-iff*: *w BIT b* = *0* ⟷ *w* = *0* ∧ ¬ *b*
apply (*auto simp add*: *Bit-def*)
apply *arith*
done

lemma *Bit-eq-m1-iff*: *w BIT b* = −*1* ⟷ *w* = −*1* ∧ *b*
apply (*auto simp add*: *Bit-def*)
apply *arith*
done

lemma *BitM-inc*: *Num.BitM* (*Num.inc w*) = *Num.Bit1 w*
by (*induct w*, *simp-all*)

lemma *expand-BIT*:
numeral (*Num.Bit0 w*) = *numeral w BIT False*
numeral (*Num.Bit1 w*) = *numeral w BIT True*
− *numeral* (*Num.Bit0 w*) = (− *numeral w*) *BIT False*
− *numeral* (*Num.Bit1 w*) = (− *numeral* (*w* + *Num.One*)) *BIT True*
unfolding *add-One* **by** (*simp-all add*: *BitM-inc*)

lemma *bin-last-numeral-simps* [*simp*]:
¬ *bin-last 0*
bin-last 1
bin-last (− *1*)
bin-last Numeral1
¬ *bin-last* (*numeral* (*Num.Bit0 w*))
bin-last (*numeral* (*Num.Bit1 w*))
¬ *bin-last* (− *numeral* (*Num.Bit0 w*))
bin-last (− *numeral* (*Num.Bit1 w*))
by (*simp-all add*: *bin-last-def zmod-zminus1-eq-if*) (*auto simp add*: *divmod-def*)

lemma *bin-rest-numeral-simps* [*simp*]:
bin-rest 0 = *0*
bin-rest 1 = *0*
bin-rest (− *1*) = − *1*

bin-rest Numeral1 = *0*
bin-rest (*numeral* (*Num.Bit0 w*)) = *numeral w*
bin-rest (*numeral* (*Num.Bit1 w*)) = *numeral w*
bin-rest (− *numeral* (*Num.Bit0 w*)) = − *numeral w*
bin-rest (− *numeral* (*Num.Bit1 w*)) = − *numeral* (*w* + *Num.One*)
by (*simp-all add*: *bin-rest-def zdiv-zminus1-eq-if*) (*auto simp add*: *divmod-def*)

lemma *less-Bits*:
v BIT b < *w BIT c* ⟷ *v* < *w* ∨ *v* ≤ *w* ∧ ¬ *b* ∧ *c*
unfolding *Bit-def* **by** *auto*

lemma *le-Bits*:
v BIT b ≤ *w BIT c* ⟷ *v* < *w* ∨ *v* ≤ *w* ∧ (¬ *b* ∨ *c*)
unfolding *Bit-def* **by** *auto*

lemma *pred-BIT-simps* [*simp*]:
x BIT False − *1* = (*x* − *1*) *BIT True*
x BIT True − *1* = *x BIT False*
by (*simp-all add*: *Bit-B0-2t Bit-B1-2t*)

lemma *succ-BIT-simps* [*simp*]:
x BIT False + *1* = *x BIT True*
x BIT True + *1* = (*x* + *1*) *BIT False*
by (*simp-all add*: *Bit-B0-2t Bit-B1-2t*)

lemma *add-BIT-simps* [*simp*]:
x BIT False + *y BIT False* = (*x* + *y*) *BIT False*
x BIT False + *y BIT True* = (*x* + *y*) *BIT True*
x BIT True + *y BIT False* = (*x* + *y*) *BIT True*
x BIT True + *y BIT True* = (*x* + *y* + *1*) *BIT False*
by (*simp-all add*: *Bit-B0-2t Bit-B1-2t*)

lemma *mult-BIT-simps* [*simp*]:
x BIT False ∗ *y* = (*x* ∗ *y*) *BIT False*
x ∗ *y BIT False* = (*x* ∗ *y*) *BIT False*
x BIT True ∗ *y* = (*x* ∗ *y*) *BIT False* + *y*
by (*simp-all add*: *Bit-B0-2t Bit-B1-2t algebra-simps*)

lemma *B-mod-2'*:
X = *2* ==> (*w BIT True*) *mod X* = *1* & (*w BIT False*) *mod X* = *0*
apply (*simp* (*no-asm*) *only*: *Bit-B0 Bit-B1*)
apply *simp*
done

lemma *bin-ex-rl*: *EX w b*. *w BIT b* = *bin*
by (*metis bin-rl-simp*)

lemma *bin-exhaust*:
assumes *Q*: ⋀*x b*. *bin* = *x BIT b* ⟹ *Q*

```
  shows Q
  apply (insert bin-ex-rl [of bin])
  apply (erule exE)+
  apply (rule Q)
  apply force
  done

primrec bin-nth where
  Z: bin-nth w 0 ⟷ bin-last w
  | Suc: bin-nth w (Suc n) ⟷ bin-nth (bin-rest w) n

lemma bin-abs-lem:
  bin = (w BIT b) ==> bin ~= -1 --> bin ~= 0 -->
    nat |w| < nat |bin|
  apply clarsimp
  apply (unfold Bit-def)
  apply (cases b)
   apply (clarsimp, arith)
  apply (clarsimp, arith)
  done

lemma bin-induct:
  assumes PPls: P 0
    and PMin: P (- 1)
    and PBit: !!bin bit. P bin ==> P (bin BIT bit)
  shows P bin
  apply (rule-tac P=P and a=bin and f1=nat o abs
                in wf-measure [THEN wf-induct])
  apply (simp add: measure-def inv-image-def)
  apply (case-tac x rule: bin-exhaust)
  apply (frule bin-abs-lem)
  apply (auto simp add : PPls PMin PBit)
  done

lemma Bit-div2 [simp]: (w BIT b) div 2 = w
  unfolding bin-rest-def [symmetric] by (rule bin-rest-BIT)

lemma bin-nth-eq-iff:
  bin-nth x = bin-nth y ⟷ x = y
proof -
  have bin-nth-lem [rule-format]: ALL y. bin-nth x = bin-nth y --> x = y
    apply (induct x rule: bin-induct)
      apply safe
      apply (erule rev-mp)
      apply (induct-tac y rule: bin-induct)
        apply safe
        apply (drule-tac x=0 in fun-cong, force)
       apply (erule notE, rule ext,
            drule-tac x=Suc x in fun-cong, force)
```

```
     apply (drule-tac x=0 in fun-cong, force)
    apply (erule rev-mp)
    apply (induct-tac y rule: bin-induct)
      apply safe
      apply (drule-tac x=0 in fun-cong, force)
     apply (erule notE, rule ext,
          drule-tac x=Suc x in fun-cong, force)
     apply (metis Bit-eq-m1-iff Z bin-last-BIT)
   apply (case-tac y rule: bin-exhaust)
   apply clarify
   apply (erule allE)
   apply (erule impE)
    prefer 2
    apply (erule conjI)
    apply (drule-tac x=0 in fun-cong, force)
   apply (rule ext)
   apply (drule-tac x=Suc x for x in fun-cong, force)
   done
  show ?thesis
  by (auto elim: bin-nth-lem)
qed

lemmas bin-eqI = ext [THEN bin-nth-eq-iff [THEN iffD1]]

lemma bin-eq-iff:
  x = y ⟷ (∀ n. bin-nth x n = bin-nth y n)
  using bin-nth-eq-iff by auto

lemma bin-nth-zero [simp]: ¬ bin-nth 0 n
  by (induct n) auto

lemma bin-nth-1 [simp]: bin-nth 1 n ⟷ n = 0
  by (cases n) simp-all

lemma bin-nth-minus1 [simp]: bin-nth (− 1) n
  by (induct n) auto

lemma bin-nth-0-BIT: bin-nth (w BIT b) 0 ⟷ b
  by auto

lemma bin-nth-Suc-BIT: bin-nth (w BIT b) (Suc n) = bin-nth w n
  by auto

lemma bin-nth-minus [simp]: 0 < n ==> bin-nth (w BIT b) n = bin-nth w (n
− 1)
  by (cases n) auto

lemma bin-nth-numeral:
  bin-rest x = y ⟹ bin-nth x (numeral n) = bin-nth y (pred-numeral n)
```

by (*simp add*: *numeral-eq-Suc*)

lemmas *bin-nth-numeral-simps* [*simp*] =
bin-nth-numeral [*OF bin-rest-numeral-simps*(*2*)]
bin-nth-numeral [*OF bin-rest-numeral-simps*(*5*)]
bin-nth-numeral [*OF bin-rest-numeral-simps*(*6*)]
bin-nth-numeral [*OF bin-rest-numeral-simps*(*7*)]
bin-nth-numeral [*OF bin-rest-numeral-simps*(*8*)]

lemmas *bin-nth-simps* =
bin-nth.Z bin-nth.Suc bin-nth-zero bin-nth-minus1
bin-nth-numeral-simps

121.2 Truncating binary integers

definition *bin-sign* :: *int* ⇒ *int*
where
bin-sign-def: *bin-sign k* = (*if k* ≥ *0 then 0 else* − *1*)

lemma *bin-sign-simps* [*simp*]:
bin-sign 0 = *0*
bin-sign 1 = *0*
bin-sign (− *1*) = − *1*
bin-sign (*numeral k*) = *0*
bin-sign (− *numeral k*) = −*1*
bin-sign (*w BIT b*) = *bin-sign w*
unfolding *bin-sign-def Bit-def*
by *simp-all*

lemma *bin-sign-rest* [*simp*]:
bin-sign (*bin-rest w*) = *bin-sign w*
by (*cases w rule*: *bin-exhaust*) *auto*

primrec *bintrunc* :: *nat* ⇒ *int* ⇒ *int* **where**
Z : *bintrunc 0 bin* = *0*
| *Suc* : *bintrunc* (*Suc n*) *bin* = *bintrunc n* (*bin-rest bin*) *BIT* (*bin-last bin*)

primrec *sbintrunc* :: *nat* => *int* => *int* **where**
Z : *sbintrunc 0 bin* = (*if bin-last bin then* −*1 else 0*)
| *Suc* : *sbintrunc* (*Suc n*) *bin* = *sbintrunc n* (*bin-rest bin*) *BIT* (*bin-last bin*)

lemma *sign-bintr*: *bin-sign* (*bintrunc n w*) = *0*
by (*induct n arbitrary*: *w*) *auto*

lemma *bintrunc-mod2p*: *bintrunc n w* = (*w mod 2* ˆ *n*)
apply (*induct n arbitrary*: *w*, *clarsimp*)
apply (*simp add*: *bin-last-def bin-rest-def Bit-def zmod-zmult2-eq*)
done

lemma *sbintrunc-mod2p*: *sbintrunc n w = (w + 2 ^ n) mod 2 ^ (Suc n) − 2 ^ n*
  apply (*induct n arbitrary*: *w*)
   apply *simp*
   apply (*subst mod-add-left-eq*)
   apply (*simp add*: *bin-last-def*)
   apply *arith*
  apply (*simp add*: *bin-last-def bin-rest-def Bit-def*)
  apply (*clarsimp simp*: *mod-mult-mult1* [*symmetric*]
        zmod-zdiv-equality [*THEN diff-eq-eq* [*THEN iffD2* [*THEN sym*]]])
  apply (*rule trans* [*symmetric*, *OF - emep1*])
  apply *auto*
  done

121.3 Simplifications for (s)bintrunc

lemma *bintrunc-n-0* [*simp*]: *bintrunc n 0 = 0*
  by (*induct n*) *auto*

lemma *sbintrunc-n-0* [*simp*]: *sbintrunc n 0 = 0*
  by (*induct n*) *auto*

lemma *sbintrunc-n-minus1* [*simp*]: *sbintrunc n* (− *1*) = −*1*
  by (*induct n*) *auto*

lemma *bintrunc-Suc-numeral*:
  bintrunc (*Suc n*) *1 = 1*
  bintrunc (*Suc n*) (− *1*) = *bintrunc n* (− *1*) *BIT True*
  bintrunc (*Suc n*) (*numeral* (*Num.Bit0 w*)) = *bintrunc n* (*numeral w*) *BIT False*
  bintrunc (*Suc n*) (*numeral* (*Num.Bit1 w*)) = *bintrunc n* (*numeral w*) *BIT True*
  bintrunc (*Suc n*) (− *numeral* (*Num.Bit0 w*)) =
   bintrunc n (− *numeral w*) *BIT False*
  bintrunc (*Suc n*) (− *numeral* (*Num.Bit1 w*)) =
   bintrunc n (− *numeral* (*w* + *Num.One*)) *BIT True*
  by *simp-all*

lemma *sbintrunc-0-numeral* [*simp*]:
  sbintrunc 0 1 = −*1*
  sbintrunc 0 (*numeral* (*Num.Bit0 w*)) = *0*
  sbintrunc 0 (*numeral* (*Num.Bit1 w*)) = −*1*
  sbintrunc 0 (− *numeral* (*Num.Bit0 w*)) = *0*
  sbintrunc 0 (− *numeral* (*Num.Bit1 w*)) = −*1*
  by *simp-all*

lemma *sbintrunc-Suc-numeral*:
  sbintrunc (*Suc n*) *1 = 1*
  sbintrunc (*Suc n*) (*numeral* (*Num.Bit0 w*)) =
   sbintrunc n (*numeral w*) *BIT False*
  sbintrunc (*Suc n*) (*numeral* (*Num.Bit1 w*)) =
   sbintrunc n (*numeral w*) *BIT True*

sbintrunc (*Suc n*) (− *numeral* (*Num.Bit0 w*)) =
sbintrunc n (− *numeral w*) *BIT False*
sbintrunc (*Suc n*) (− *numeral* (*Num.Bit1 w*)) =
sbintrunc n (− *numeral* (*w* + *Num.One*)) *BIT True*
by *simp-all*

lemma *bin-sign-lem*: (*bin-sign* (*sbintrunc n bin*) = −*1*) = *bin-nth bin n*
apply (*induct n arbitrary*: *bin*)
apply (*case-tac bin rule*: *bin-exhaust*, *case-tac b*, *auto*)
done

lemma *nth-bintr*: *bin-nth* (*bintrunc m w*) *n* = (*n* < *m* & *bin-nth w n*)
apply (*induct n arbitrary*: *w m*)
apply (*case-tac m*, *auto*)[*1*]
apply (*case-tac m*, *auto*)[*1*]
done

lemma *nth-sbintr*:
bin-nth (*sbintrunc m w*) *n* =
(*if n* < *m then bin-nth w n else bin-nth w m*)
apply (*induct n arbitrary*: *w m*)
apply (*case-tac m*)
apply *simp-all*
apply (*case-tac m*)
apply *simp-all*
done

lemma *bin-nth-Bit*:
bin-nth (*w BIT b*) *n* = (*n* = *0* & *b* | (*EX m*. *n* = *Suc m* & *bin-nth w m*))
by (*cases n*) *auto*

lemma *bin-nth-Bit0*:
bin-nth (*numeral* (*Num.Bit0 w*)) *n* ⟷
(∃ *m*. *n* = *Suc m* ∧ *bin-nth* (*numeral w*) *m*)
using *bin-nth-Bit* [**where** *w*=*numeral w* **and** *b*=*False*] **by** *simp*

lemma *bin-nth-Bit1*:
bin-nth (*numeral* (*Num.Bit1 w*)) *n* ⟷
n = *0* ∨ (∃ *m*. *n* = *Suc m* ∧ *bin-nth* (*numeral w*) *m*)
using *bin-nth-Bit* [**where** *w*=*numeral w* **and** *b*=*True*] **by** *simp*

lemma *bintrunc-bintrunc-l*:
n <= *m* ==> (*bintrunc m* (*bintrunc n w*) = *bintrunc n w*)
by (*rule bin-eqI*) (*auto simp add* : *nth-bintr*)

lemma *sbintrunc-sbintrunc-l*:
n <= *m* ==> (*sbintrunc m* (*sbintrunc n w*) = *sbintrunc n w*)
by (*rule bin-eqI*) (*auto simp*: *nth-sbintr*)

lemma *bintrunc-bintrunc-ge*:
n <= *m* ==> (*bintrunc n* (*bintrunc m w*) = *bintrunc n w*)
by (*rule bin-eqI*) (*auto simp*: *nth-bintr*)

lemma *bintrunc-bintrunc-min* [*simp*]:
bintrunc m (*bintrunc n w*) = *bintrunc* (*min m n*) *w*
apply (*rule bin-eqI*)
apply (*auto simp*: *nth-bintr*)
done

lemma *sbintrunc-sbintrunc-min* [*simp*]:
sbintrunc m (*sbintrunc n w*) = *sbintrunc* (*min m n*) *w*
apply (*rule bin-eqI*)
apply (*auto simp*: *nth-sbintr min.absorb1 min.absorb2*)
done

lemmas *bintrunc-Pls* =
bintrunc.Suc [**where** *bin=0*, *simplified bin-last-numeral-simps bin-rest-numeral-simps*]

lemmas *bintrunc-Min* [*simp*] =
bintrunc.Suc [**where** *bin=−1*, *simplified bin-last-numeral-simps bin-rest-numeral-simps*]

lemmas *bintrunc-BIT* [*simp*] =
bintrunc.Suc [**where** *bin=w BIT b*, *simplified bin-last-BIT bin-rest-BIT*] **for** *w b*

lemmas *bintrunc-Sucs* = *bintrunc-Pls bintrunc-Min bintrunc-BIT*
bintrunc-Suc-numeral

lemmas *sbintrunc-Suc-Pls* =
sbintrunc.Suc [**where** *bin=0*, *simplified bin-last-numeral-simps bin-rest-numeral-simps*]

lemmas *sbintrunc-Suc-Min* =
sbintrunc.Suc [**where** *bin=−1*, *simplified bin-last-numeral-simps bin-rest-numeral-simps*]

lemmas *sbintrunc-Suc-BIT* [*simp*] =
sbintrunc.Suc [**where** *bin=w BIT b*, *simplified bin-last-BIT bin-rest-BIT*] **for** *w*
b

lemmas *sbintrunc-Sucs* = *sbintrunc-Suc-Pls sbintrunc-Suc-Min sbintrunc-Suc-BIT*
sbintrunc-Suc-numeral

lemmas *sbintrunc-Pls* =
sbintrunc.Z [**where** *bin=0*,
simplified bin-last-numeral-simps bin-rest-numeral-simps]

lemmas *sbintrunc-Min* =
sbintrunc.Z [**where** *bin=−1*,
simplified bin-last-numeral-simps bin-rest-numeral-simps]

lemmas sbintrunc-0-BIT-B0 [simp] =
sbintrunc.Z [**where** bin=w BIT False,
simplified bin-last-numeral-simps bin-rest-numeral-simps] **for** w

lemmas sbintrunc-0-BIT-B1 [simp] =
sbintrunc.Z [**where** bin=w BIT True,
simplified bin-last-BIT bin-rest-numeral-simps] **for** w

lemmas sbintrunc-0-simps =
sbintrunc-Pls sbintrunc-Min sbintrunc-0-BIT-B0 sbintrunc-0-BIT-B1

lemmas bintrunc-simps = bintrunc.Z bintrunc-Sucs
lemmas sbintrunc-simps = sbintrunc-0-simps sbintrunc-Sucs

lemma bintrunc-minus:
0 < n ==> bintrunc (Suc (n − 1)) w = bintrunc n w
by auto

lemma sbintrunc-minus:
0 < n ==> sbintrunc (Suc (n − 1)) w = sbintrunc n w
by auto

lemmas bintrunc-minus-simps =
bintrunc-Sucs [THEN [2] bintrunc-minus [symmetric, THEN trans]]
lemmas sbintrunc-minus-simps =
sbintrunc-Sucs [THEN [2] sbintrunc-minus [symmetric, THEN trans]]

lemmas thobini1 = arg-cong [**where** f = %w. w BIT b] **for** b

lemmas bintrunc-BIT-I = trans [OF bintrunc-BIT thobini1]
lemmas bintrunc-Min-I = trans [OF bintrunc-Min thobini1]

lemmas bmsts = bintrunc-minus-simps(1−3) [THEN thobini1 [THEN [2] trans]]
lemmas bintrunc-Pls-minus-I = bmsts(1)
lemmas bintrunc-Min-minus-I = bmsts(2)
lemmas bintrunc-BIT-minus-I = bmsts(3)

lemma bintrunc-Suc-lem:
bintrunc (Suc n) x = y ==> m = Suc n ==> bintrunc m x = y
by auto

lemmas bintrunc-Suc-Ialts =
bintrunc-Min-I [THEN bintrunc-Suc-lem]
bintrunc-BIT-I [THEN bintrunc-Suc-lem]

lemmas sbintrunc-BIT-I = trans [OF sbintrunc-Suc-BIT thobini1]

lemmas sbintrunc-Suc-Is =
sbintrunc-Sucs(1−3) [THEN thobini1 [THEN [2] trans]]

lemmas *sbintrunc-Suc-minus-Is* =
sbintrunc-minus-simps(*1*−*3*) [*THEN thobini1* [*THEN* [*2*] *trans*]]

lemma *sbintrunc-Suc-lem*:
sbintrunc (*Suc n*) *x* = *y* ==> *m* = *Suc n* ==> *sbintrunc m x* = *y*
by *auto*

lemmas *sbintrunc-Suc-Ialts* =
sbintrunc-Suc-Is [*THEN sbintrunc-Suc-lem*]

lemma *sbintrunc-bintrunc-lt*:
m > *n* ==> *sbintrunc n* (*bintrunc m w*) = *sbintrunc n w*
by (*rule bin-eqI*) (*auto simp*: *nth-sbintr nth-bintr*)

lemma *bintrunc-sbintrunc-le*:
m <= *Suc n* ==> *bintrunc m* (*sbintrunc n w*) = *bintrunc m w*
apply (*rule bin-eqI*)
apply (*auto simp*: *nth-sbintr nth-bintr*)
apply (*subgoal-tac x*=*n*, *safe*, *arith*+)[*1*]
apply (*subgoal-tac x*=*n*, *safe*, *arith*+)[*1*]
done

lemmas *bintrunc-sbintrunc* [*simp*] = *order-refl* [*THEN bintrunc-sbintrunc-le*]
lemmas *sbintrunc-bintrunc* [*simp*] = *lessI* [*THEN sbintrunc-bintrunc-lt*]
lemmas *bintrunc-bintrunc* [*simp*] = *order-refl* [*THEN bintrunc-bintrunc-l*]
lemmas *sbintrunc-sbintrunc* [*simp*] = *order-refl* [*THEN sbintrunc-sbintrunc-l*]

lemma *bintrunc-sbintrunc′* [*simp*]:
0 < *n* ⟹ *bintrunc n* (*sbintrunc* (*n* − *1*) *w*) = *bintrunc n w*
by (*cases n*) (*auto simp del*: *bintrunc.Suc*)

lemma *sbintrunc-bintrunc′* [*simp*]:
0 < *n* ⟹ *sbintrunc* (*n* − *1*) (*bintrunc n w*) = *sbintrunc* (*n* − *1*) *w*
by (*cases n*) (*auto simp del*: *bintrunc.Suc*)

lemma *bin-sbin-eq-iff*:
bintrunc (*Suc n*) *x* = *bintrunc* (*Suc n*) *y* ⟷
sbintrunc n x = *sbintrunc n y*
apply (*rule iffI*)
apply (*rule box-equals* [*OF* - *sbintrunc-bintrunc sbintrunc-bintrunc*])
apply *simp*
apply (*rule box-equals* [*OF* - *bintrunc-sbintrunc bintrunc-sbintrunc*])
apply *simp*
done

lemma *bin-sbin-eq-iff′*:
0 < *n* ⟹ *bintrunc n x* = *bintrunc n y* ⟷
sbintrunc (*n* − *1*) *x* = *sbintrunc* (*n* − *1*) *y*

by (*cases n*) (*simp-all add*: *bin-sbin-eq-iff del*: *bintrunc.Suc*)

lemmas *bintrunc-sbintruncS0* [*simp*] = *bintrunc-sbintrunc′* [*unfolded One-nat-def*]
lemmas *sbintrunc-bintruncS0* [*simp*] = *sbintrunc-bintrunc′* [*unfolded One-nat-def*]

lemmas *bintrunc-bintrunc-l′* = *le-add1* [*THEN bintrunc-bintrunc-l*]
lemmas *sbintrunc-sbintrunc-l′* = *le-add1* [*THEN sbintrunc-sbintrunc-l*]

lemmas *nat-non0-gr* =
trans [*OF iszero-def* [*THEN Not-eq-iff* [*THEN iffD2*]] *refl*]

lemma *bintrunc-numeral*:
bintrunc (*numeral k*) *x* =
bintrunc (*pred-numeral k*) (*bin-rest x*) *BIT bin-last x*
by (*simp add*: *numeral-eq-Suc*)

lemma *sbintrunc-numeral*:
sbintrunc (*numeral k*) *x* =
sbintrunc (*pred-numeral k*) (*bin-rest x*) *BIT bin-last x*
by (*simp add*: *numeral-eq-Suc*)

lemma *bintrunc-numeral-simps* [*simp*]:
bintrunc (*numeral k*) (*numeral* (*Num.Bit0 w*)) =
bintrunc (*pred-numeral k*) (*numeral w*) *BIT False*
bintrunc (*numeral k*) (*numeral* (*Num.Bit1 w*)) =
bintrunc (*pred-numeral k*) (*numeral w*) *BIT True*
bintrunc (*numeral k*) (− *numeral* (*Num.Bit0 w*)) =
bintrunc (*pred-numeral k*) (− *numeral w*) *BIT False*
bintrunc (*numeral k*) (− *numeral* (*Num.Bit1 w*)) =
bintrunc (*pred-numeral k*) (− *numeral* (*w* + *Num.One*)) *BIT True*
bintrunc (*numeral k*) *1* = *1*
by (*simp-all add*: *bintrunc-numeral*)

lemma *sbintrunc-numeral-simps* [*simp*]:
sbintrunc (*numeral k*) (*numeral* (*Num.Bit0 w*)) =
sbintrunc (*pred-numeral k*) (*numeral w*) *BIT False*
sbintrunc (*numeral k*) (*numeral* (*Num.Bit1 w*)) =
sbintrunc (*pred-numeral k*) (*numeral w*) *BIT True*
sbintrunc (*numeral k*) (− *numeral* (*Num.Bit0 w*)) =
sbintrunc (*pred-numeral k*) (− *numeral w*) *BIT False*
sbintrunc (*numeral k*) (− *numeral* (*Num.Bit1 w*)) =
sbintrunc (*pred-numeral k*) (− *numeral* (*w* + *Num.One*)) *BIT True*
sbintrunc (*numeral k*) *1* = *1*
by (*simp-all add*: *sbintrunc-numeral*)

lemma *no-bintr-alt1*: *bintrunc n* = (λ*w*. *w mod 2* ^ *n* :: *int*)
by (*rule ext*) (*rule bintrunc-mod2p*)

lemma *range-bintrunc*: *range* (*bintrunc n*) = {*i*. *0* <= *i* & *i* < *2* ˆ *n*}
  apply (*unfold no-bintr-alt1*)
  apply (*auto simp add*: *image-iff*)
  apply (*rule exI*)
  apply (*auto intro*: *int-mod-lem* [*THEN iffD1*, *symmetric*])
  done

lemma *no-sbintr-alt2*:
  sbintrunc n = (%*w*. (*w* + *2* ˆ *n*) *mod 2* ˆ *Suc n* − *2* ˆ *n* :: *int*)
  by (*rule ext*) (*simp add* : *sbintrunc-mod2p*)

lemma *range-sbintrunc*:
  range (*sbintrunc n*) = {*i*. − (*2* ˆ *n*) <= *i* & *i* < *2* ˆ *n*}
  apply (*unfold no-sbintr-alt2*)
  apply (*auto simp add*: *image-iff eq-diff-eq*)
  apply (*rule exI*)
  apply (*auto intro*: *int-mod-lem* [*THEN iffD1*, *symmetric*])
  done

lemma *sb-inc-lem*:
  (*a*::*int*) + *2*ˆ*k* < *0* ⟹ *a* + *2*ˆ*k* + *2*ˆ(*Suc k*) <= (*a* + *2*ˆ*k*) *mod 2*ˆ(*Suc k*)
  apply (*erule int-mod-ge′* [**where** *n* = *2* ˆ (*Suc k*) **and** *b* = *a* + *2* ˆ *k*, *simplified zless2p*])
  apply (*rule TrueI*)
  done

lemma *sb-inc-lem′*:
  (*a*::*int*) < − (*2*ˆ*k*) ⟹ *a* + *2*ˆ*k* + *2*ˆ(*Suc k*) <= (*a* + *2*ˆ*k*) *mod 2*ˆ(*Suc k*)
  by (*rule sb-inc-lem*) *simp*

lemma *sbintrunc-inc*:
  x < − (*2*ˆ*n*) ==> *x* + *2*ˆ(*Suc n*) <= *sbintrunc n x*
  unfolding *no-sbintr-alt2* **by** (*drule sb-inc-lem′*) *simp*

lemma *sb-dec-lem*:
  (*0*::*int*) ≤ − (*2* ˆ *k*) + *a* ⟹ (*a* + *2* ˆ *k*) *mod* (*2* ∗ *2* ˆ *k*) ≤ − (*2* ˆ *k*) + *a*
  using *int-mod-le′*[**where** *n* = *2* ˆ (*Suc k*) **and** *b* = *a* + *2* ˆ *k*] **by** *simp*

lemma *sb-dec-lem′*:
  (*2*::*int*) ˆ *k* ≤ *a* ⟹ (*a* + *2* ˆ *k*) *mod* (*2* ∗ *2* ˆ *k*) ≤ − (*2* ˆ *k*) + *a*
  by (*rule sb-dec-lem*) *simp*

lemma *sbintrunc-dec*:
  x >= (*2* ˆ *n*) ==> *x* − *2* ˆ (*Suc n*) >= *sbintrunc n x*
  unfolding *no-sbintr-alt2* **by** (*drule sb-dec-lem′*) *simp*

lemmas *zmod-uminus′* = *zminus-zmod* [**where** *m*=*c*] **for** *c*
lemmas *zpower-zmod′* = *power-mod* [**where** *b*=*c* **and** *n*=*k*] **for** *c k*

lemmas *brdmod1s'* [*symmetric*] =
mod-add-left-eq mod-add-right-eq
mod-diff-left-eq mod-diff-right-eq
mod-mult-left-eq mod-mult-right-eq

lemmas *brdmods'* [*symmetric*] =
zpower-zmod' [*symmetric*]
trans [*OF mod-add-left-eq mod-add-right-eq*]
trans [*OF mod-diff-left-eq mod-diff-right-eq*]
trans [*OF mod-mult-right-eq mod-mult-left-eq*]
zmod-uminus' [*symmetric*]
mod-add-left-eq [**where** $b = 1::int$]
mod-diff-left-eq [**where** $b = 1::int$]

lemmas *bintr-arith1s* =
brdmod1s' [**where** $c=2\hat{}n::int$, *folded bintrunc-mod2p*] **for** n
lemmas *bintr-ariths* =
brdmods' [**where** $c=2\hat{}n::int$, *folded bintrunc-mod2p*] **for** n

lemmas *m2pths* = *pos-mod-sign pos-mod-bound* [*OF zless2p*]

lemma *bintr-ge0*: $0 \leq bintrunc\ n\ w$
by (*simp add*: *bintrunc-mod2p*)

lemma *bintr-lt2p*: $bintrunc\ n\ w < 2\ \hat{}\ n$
by (*simp add*: *bintrunc-mod2p*)

lemma *bintr-Min*: $bintrunc\ n\ (-\ 1) = 2\ \hat{}\ n - 1$
by (*simp add*: *bintrunc-mod2p m1mod2k*)

lemma *sbintr-ge*: $-\ (2\ \hat{}\ n) \leq sbintrunc\ n\ w$
by (*simp add*: *sbintrunc-mod2p*)

lemma *sbintr-lt*: $sbintrunc\ n\ w < 2\ \hat{}\ n$
by (*simp add*: *sbintrunc-mod2p*)

lemma *sign-Pls-ge-0*:
(*bin-sign bin* = *0*) = (*bin* >= (*0* :: *int*))
unfolding *bin-sign-def* **by** *simp*

lemma *sign-Min-lt-0*:
(*bin-sign bin* = −*1*) = (*bin* < (*0* :: *int*))
unfolding *bin-sign-def* **by** *simp*

lemma *bin-rest-trunc*:
(*bin-rest* (*bintrunc n bin*)) = *bintrunc* (*n* − *1*) (*bin-rest bin*)
by (*induct n arbitrary*: *bin*) *auto*

lemma *bin-rest-power-trunc*:
(*bin-rest* ^^ *k*) (*bintrunc n bin*) =
bintrunc (*n* − *k*) ((*bin-rest* ^^ *k*) *bin*)
by (*induct k*) (*auto simp*: *bin-rest-trunc*)

lemma *bin-rest-trunc-i*:
bintrunc n (*bin-rest bin*) = *bin-rest* (*bintrunc* (*Suc n*) *bin*)
by *auto*

lemma *bin-rest-strunc*:
bin-rest (*sbintrunc* (*Suc n*) *bin*) = *sbintrunc n* (*bin-rest bin*)
by (*induct n arbitrary*: *bin*) *auto*

lemma *bintrunc-rest* [*simp*]:
bintrunc n (*bin-rest* (*bintrunc n bin*)) = *bin-rest* (*bintrunc n bin*)
apply (*induct n arbitrary*: *bin*, *simp*)
apply (*case-tac bin rule*: *bin-exhaust*)
apply (*auto simp*: *bintrunc-bintrunc-l*)
done

lemma *sbintrunc-rest* [*simp*]:
sbintrunc n (*bin-rest* (*sbintrunc n bin*)) = *bin-rest* (*sbintrunc n bin*)
apply (*induct n arbitrary*: *bin*, *simp*)
apply (*case-tac bin rule*: *bin-exhaust*)
apply (*auto simp*: *bintrunc-bintrunc-l split*: *bool.splits*)
done

lemma *bintrunc-rest'*:
bintrunc n o bin-rest o bintrunc n = *bin-rest o bintrunc n*
by (*rule ext*) *auto*

lemma *sbintrunc-rest'* :
sbintrunc n o bin-rest o sbintrunc n = *bin-rest o sbintrunc n*
by (*rule ext*) *auto*

lemma *rco-lem*:
f o g o f = *g o f* ==> *f o* (*g o f*) ^^ *n* = *g* ^^ *n o f*
apply (*rule ext*)
apply (*induct-tac n*)
apply (*simp-all* (*no-asm*))
apply (*drule fun-cong*)
apply (*unfold o-def*)
apply (*erule trans*)
apply *simp*
done

lemmas *rco-bintr* = *bintrunc-rest'*
[*THEN rco-lem* [*THEN fun-cong*], *unfolded o-def*]
lemmas *rco-sbintr* = *sbintrunc-rest'*

[*THEN rco-lem* [*THEN fun-cong*], *unfolded o-def*]

121.4 Splitting and concatenation

primrec *bin-split* :: *nat* ⇒ *int* ⇒ *int* × *int* **where**
Z: *bin-split 0 w* = (*w*, *0*)
| *Suc*: *bin-split* (*Suc n*) *w* = (*let* (*w1*, *w2*) = *bin-split n* (*bin-rest w*)
in (*w1*, *w2 BIT bin-last w*))

lemma [*code*]:
bin-split (*Suc n*) *w* = (*let* (*w1*, *w2*) = *bin-split n* (*bin-rest w*) *in* (*w1*, *w2 BIT bin-last w*))
bin-split 0 w = (*w*, *0*)
by *simp-all*

primrec *bin-cat* :: *int* ⇒ *nat* ⇒ *int* ⇒ *int* **where**
Z: *bin-cat w 0 v* = *w*
| *Suc*: *bin-cat w* (*Suc n*) *v* = *bin-cat w n* (*bin-rest v*) *BIT bin-last v*

end

122 Bitwise Operations on Binary Integers

theory *Bits-Int*
imports *Bits Bit-Representation*
begin

122.1 Logical operations

bit-wise logical operations on the int type

instantiation *int* :: *bit*
begin

definition *int-not-def*:
bitNOT = (λ*x*::*int*. − *x* − *1*)

function *bitAND-int* **where**
bitAND-int x y =
(*if x* = *0 then 0 else if x* = −*1 then y else*
(*bin-rest x AND bin-rest y*) *BIT* (*bin-last x* ∧ *bin-last y*))
by *pat-completeness simp*

termination
by (*relation measure* (*nat o abs o fst*), *simp-all add*: *bin-rest-def*)

declare *bitAND-int.simps* [*simp del*]

definition *int-or-def*:

bitOR = (λ*x y*::*int*. *NOT* (*NOT x AND NOT y*))

definition *int-xor-def*:
bitXOR = (λ*x y*::*int*. (*x AND NOT y*) *OR* (*NOT x AND y*))

instance ..

end

122.1.1 Basic simplification rules

lemma *int-not-BIT* [*simp*]:
NOT (*w BIT b*) = (*NOT w*) *BIT* (¬ *b*)
unfolding *int-not-def Bit-def* **by** (*cases b*, *simp-all*)

lemma *int-not-simps* [*simp*]:
NOT (*0*::*int*) = −*1*
NOT (*1*::*int*) = −*2*
NOT (− *1*::*int*) = *0*
NOT (*numeral w*::*int*) = − *numeral* (*w* + *Num.One*)
NOT (− *numeral* (*Num.Bit0 w*)::*int*) = *numeral* (*Num.BitM w*)
NOT (− *numeral* (*Num.Bit1 w*)::*int*) = *numeral* (*Num.Bit0 w*)
unfolding *int-not-def* **by** *simp-all*

lemma *int-not-not* [*simp*]: *NOT* (*NOT* (*x*::*int*)) = *x*
unfolding *int-not-def* **by** *simp*

lemma *int-and-0* [*simp*]: (*0*::*int*) *AND x* = *0*
by (*simp add*: *bitAND-int.simps*)

lemma *int-and-m1* [*simp*]: (−*1*::*int*) *AND x* = *x*
by (*simp add*: *bitAND-int.simps*)

lemma *int-and-Bits* [*simp*]:
(*x BIT b*) *AND* (*y BIT c*) = (*x AND y*) *BIT* (*b* ∧ *c*)
by (*subst bitAND-int.simps*, *simp add*: *Bit-eq-0-iff Bit-eq-m1-iff*)

lemma *int-or-zero* [*simp*]: (*0*::*int*) *OR x* = *x*
unfolding *int-or-def* **by** *simp*

lemma *int-or-minus1* [*simp*]: (−*1*::*int*) *OR x* = −*1*
unfolding *int-or-def* **by** *simp*

lemma *int-or-Bits* [*simp*]:
(*x BIT b*) *OR* (*y BIT c*) = (*x OR y*) *BIT* (*b* ∨ *c*)
unfolding *int-or-def* **by** *simp*

lemma *int-xor-zero* [*simp*]: (*0*::*int*) *XOR x* = *x*
unfolding *int-xor-def* **by** *simp*

lemma *int-xor-Bits* [*simp*]:
(x *BIT* b) *XOR* (y *BIT* c) = (x *XOR* y) *BIT* (($b \vee c$) $\wedge \neg$ ($b \wedge c$))
unfolding *int-xor-def* **by** *auto*

122.1.2 Binary destructors

lemma *bin-rest-NOT* [*simp*]: *bin-rest* (*NOT* x) = *NOT* (*bin-rest* x)
by (*cases x rule*: *bin-exhaust*, *simp*)

lemma *bin-last-NOT* [*simp*]: *bin-last* (*NOT* x) $\longleftrightarrow \neg$ *bin-last* x
by (*cases x rule*: *bin-exhaust*, *simp*)

lemma *bin-rest-AND* [*simp*]: *bin-rest* (x *AND* y) = *bin-rest* x *AND* *bin-rest* y
by (*cases x rule*: *bin-exhaust*, *cases y rule*: *bin-exhaust*, *simp*)

lemma *bin-last-AND* [*simp*]: *bin-last* (x *AND* y) $\longleftrightarrow$ *bin-last* $x \wedge$ *bin-last* y
by (*cases x rule*: *bin-exhaust*, *cases y rule*: *bin-exhaust*, *simp*)

lemma *bin-rest-OR* [*simp*]: *bin-rest* (x *OR* y) = *bin-rest* x *OR* *bin-rest* y
by (*cases x rule*: *bin-exhaust*, *cases y rule*: *bin-exhaust*, *simp*)

lemma *bin-last-OR* [*simp*]: *bin-last* (x *OR* y) $\longleftrightarrow$ *bin-last* $x \vee$ *bin-last* y
by (*cases x rule*: *bin-exhaust*, *cases y rule*: *bin-exhaust*, *simp*)

lemma *bin-rest-XOR* [*simp*]: *bin-rest* (x *XOR* y) = *bin-rest* x *XOR* *bin-rest* y
by (*cases x rule*: *bin-exhaust*, *cases y rule*: *bin-exhaust*, *simp*)

lemma *bin-last-XOR* [*simp*]: *bin-last* (x *XOR* y) $\longleftrightarrow$ (*bin-last* $x \vee$ *bin-last* y) $\wedge$
$\neg$ (*bin-last* $x \wedge$ *bin-last* y)
by (*cases x rule*: *bin-exhaust*, *cases y rule*: *bin-exhaust*, *simp*)

lemma *bin-nth-ops*:
!!x y. *bin-nth* (x *AND* y) n = (*bin-nth* x n & *bin-nth* y n)
!!x y. *bin-nth* (x *OR* y) n = (*bin-nth* x n | *bin-nth* y n)
!!x y. *bin-nth* (x *XOR* y) n = (*bin-nth* x n ~= *bin-nth* y n)
!!x. *bin-nth* (*NOT* x) n = (~ *bin-nth* x n)
by (*induct n*) *auto*

122.1.3 Derived properties

lemma *int-xor-minus1* [*simp*]: (-1::*int*) *XOR* x = *NOT* x
by (*auto simp add*: *bin-eq-iff bin-nth-ops*)

lemma *int-xor-extra-simps* [*simp*]:
w *XOR* (*0*::*int*) = w
w *XOR* (-1::*int*) = *NOT* w
by (*auto simp add*: *bin-eq-iff bin-nth-ops*)

lemma *int-or-extra-simps* [*simp*]:

w OR (*0*::*int*) = *w*
w OR (−*1*::*int*) = −*1*
by (*auto simp add*: *bin-eq-iff bin-nth-ops*)

lemma *int-and-extra-simps* [*simp*]:
w AND (*0*::*int*) = *0*
w AND (−*1*::*int*) = *w*
by (*auto simp add*: *bin-eq-iff bin-nth-ops*)

lemma *bin-ops-comm*:
shows
int-and-comm: !!*y*::*int*. *x AND y* = *y AND x* **and**
int-or-comm: !!*y*::*int*. *x OR y* = *y OR x* **and**
int-xor-comm: !!*y*::*int*. *x XOR y* = *y XOR x*
by (*auto simp add*: *bin-eq-iff bin-nth-ops*)

lemma *bin-ops-same* [*simp*]:
(*x*::*int*) *AND x* = *x*
(*x*::*int*) *OR x* = *x*
(*x*::*int*) *XOR x* = *0*
by (*auto simp add*: *bin-eq-iff bin-nth-ops*)

lemmas *bin-log-esimps* =
int-and-extra-simps int-or-extra-simps int-xor-extra-simps
int-and-0 int-and-m1 int-or-zero int-or-minus1 int-xor-zero int-xor-minus1

lemma *bbw-ao-absorb*:
!!*y*::*int*. *x AND* (*y OR x*) = *x* & *x OR* (*y AND x*) = *x*
by (*auto simp add*: *bin-eq-iff bin-nth-ops*)

lemma *bbw-ao-absorbs-other*:
x AND (*x OR y*) = *x* ∧ (*y AND x*) *OR x* = (*x*::*int*)
(*y OR x*) *AND x* = *x* ∧ *x OR* (*x AND y*) = (*x*::*int*)
(*x OR y*) *AND x* = *x* ∧ (*x AND y*) *OR x* = (*x*::*int*)
by (*auto simp add*: *bin-eq-iff bin-nth-ops*)

lemmas *bbw-ao-absorbs* [*simp*] = *bbw-ao-absorb bbw-ao-absorbs-other*

lemma *int-xor-not*:
!!*y*::*int*. (*NOT x*) *XOR y* = *NOT* (*x XOR y*) &
x XOR (*NOT y*) = *NOT* (*x XOR y*)
by (*auto simp add*: *bin-eq-iff bin-nth-ops*)

lemma *int-and-assoc*:
(*x AND y*) *AND* (*z*::*int*) = *x AND* (*y AND z*)
by (*auto simp add*: *bin-eq-iff bin-nth-ops*)

lemma *int-or-assoc*:
  (*x OR y*) *OR* (*z*::*int*) = *x OR* (*y OR z*)
  by (*auto simp add*: *bin-eq-iff bin-nth-ops*)

lemma *int-xor-assoc*:
  (*x XOR y*) *XOR* (*z*::*int*) = *x XOR* (*y XOR z*)
  by (*auto simp add*: *bin-eq-iff bin-nth-ops*)

lemmas *bbw-assocs* = *int-and-assoc int-or-assoc int-xor-assoc*

lemma *bbw-lcs* [*simp*]:
  (*y*::*int*) *AND* (*x AND z*) = *x AND* (*y AND z*)
  (*y*::*int*) *OR* (*x OR z*) = *x OR* (*y OR z*)
  (*y*::*int*) *XOR* (*x XOR z*) = *x XOR* (*y XOR z*)
  by (*auto simp add*: *bin-eq-iff bin-nth-ops*)

lemma *bbw-not-dist*:
  !!*y*::*int*. *NOT* (*x OR y*) = (*NOT x*) *AND* (*NOT y*)
  !!*y*::*int*. *NOT* (*x AND y*) = (*NOT x*) *OR* (*NOT y*)
  by (*auto simp add*: *bin-eq-iff bin-nth-ops*)

lemma *bbw-oa-dist*:
  !!*y z*::*int*. (*x AND y*) *OR z* =
      (*x OR z*) *AND* (*y OR z*)
  by (*auto simp add*: *bin-eq-iff bin-nth-ops*)

lemma *bbw-ao-dist*:
  !!*y z*::*int*. (*x OR y*) *AND z* =
      (*x AND z*) *OR* (*y AND z*)
  by (*auto simp add*: *bin-eq-iff bin-nth-ops*)

122.1.4 Simplification with numerals

Cases for *0* and *−1* are already covered by other simp rules.

lemma *bin-rl-eqI*: ⟦*bin-rest x* = *bin-rest y*; *bin-last x* = *bin-last y*⟧ ⟹ *x* = *y*
  by (*metis* (*mono-tags*) *BIT-eq-iff bin-ex-rl bin-last-BIT bin-rest-BIT*)

lemma *bin-rest-neg-numeral-BitM* [*simp*]:
  bin-rest (− *numeral* (*Num.BitM w*)) = − *numeral w*
  by (*simp only*: *BIT-bin-simps* [*symmetric*] *bin-rest-BIT*)

lemma *bin-last-neg-numeral-BitM* [*simp*]:
  bin-last (− *numeral* (*Num.BitM w*))
  by (*simp only*: *BIT-bin-simps* [*symmetric*] *bin-last-BIT*)

FIXME: The rule sets below are very large (24 rules for each operator). Is there a simpler way to do this?

lemma *int-and-numerals* [*simp*]:
numeral (*Num.Bit0 x*) *AND numeral* (*Num.Bit0 y*) = (*numeral x AND numeral y*) *BIT False*
numeral (*Num.Bit0 x*) *AND numeral* (*Num.Bit1 y*) = (*numeral x AND numeral y*) *BIT False*
numeral (*Num.Bit1 x*) *AND numeral* (*Num.Bit0 y*) = (*numeral x AND numeral y*) *BIT False*
numeral (*Num.Bit1 x*) *AND numeral* (*Num.Bit1 y*) = (*numeral x AND numeral y*) *BIT True*
numeral (*Num.Bit0 x*) *AND* − *numeral* (*Num.Bit0 y*) = (*numeral x AND* − *numeral y*) *BIT False*
numeral (*Num.Bit0 x*) *AND* − *numeral* (*Num.Bit1 y*) = (*numeral x AND* − *numeral* (*y* + *Num.One*)) *BIT False*
numeral (*Num.Bit1 x*) *AND* − *numeral* (*Num.Bit0 y*) = (*numeral x AND* − *numeral y*) *BIT False*
numeral (*Num.Bit1 x*) *AND* − *numeral* (*Num.Bit1 y*) = (*numeral x AND* − *numeral* (*y* + *Num.One*)) *BIT True*
− *numeral* (*Num.Bit0 x*) *AND numeral* (*Num.Bit0 y*) = (− *numeral x AND numeral y*) *BIT False*
− *numeral* (*Num.Bit0 x*) *AND numeral* (*Num.Bit1 y*) = (− *numeral x AND numeral y*) *BIT False*
− *numeral* (*Num.Bit1 x*) *AND numeral* (*Num.Bit0 y*) = (− *numeral* (*x* + *Num.One*) *AND numeral y*) *BIT False*
− *numeral* (*Num.Bit1 x*) *AND numeral* (*Num.Bit1 y*) = (− *numeral* (*x* + *Num.One*) *AND numeral y*) *BIT True*
− *numeral* (*Num.Bit0 x*) *AND* − *numeral* (*Num.Bit0 y*) = (− *numeral x AND* − *numeral y*) *BIT False*
− *numeral* (*Num.Bit0 x*) *AND* − *numeral* (*Num.Bit1 y*) = (− *numeral x AND* − *numeral* (*y* + *Num.One*)) *BIT False*
− *numeral* (*Num.Bit1 x*) *AND* − *numeral* (*Num.Bit0 y*) = (− *numeral* (*x* + *Num.One*) *AND* − *numeral y*) *BIT False*
− *numeral* (*Num.Bit1 x*) *AND* − *numeral* (*Num.Bit1 y*) = (− *numeral* (*x* + *Num.One*) *AND* − *numeral* (*y* + *Num.One*)) *BIT True*
(*1*::*int*) *AND numeral* (*Num.Bit0 y*) = *0*
(*1*::*int*) *AND numeral* (*Num.Bit1 y*) = *1*
(*1*::*int*) *AND* − *numeral* (*Num.Bit0 y*) = *0*
(*1*::*int*) *AND* − *numeral* (*Num.Bit1 y*) = *1*
numeral (*Num.Bit0 x*) *AND* (*1*::*int*) = *0*
numeral (*Num.Bit1 x*) *AND* (*1*::*int*) = *1*
− *numeral* (*Num.Bit0 x*) *AND* (*1*::*int*) = *0*
− *numeral* (*Num.Bit1 x*) *AND* (*1*::*int*) = *1*
by (*rule bin-rl-eqI*, *simp*, *simp*)+

lemma *int-or-numerals* [*simp*]:
numeral (*Num.Bit0 x*) *OR numeral* (*Num.Bit0 y*) = (*numeral x OR numeral y*) *BIT False*
numeral (*Num.Bit0 x*) *OR numeral* (*Num.Bit1 y*) = (*numeral x OR numeral y*) *BIT True*
numeral (*Num.Bit1 x*) *OR numeral* (*Num.Bit0 y*) = (*numeral x OR numeral y*)

BIT True
numeral (*Num.Bit1 x*) *OR numeral* (*Num.Bit1 y*) = (*numeral x OR numeral y*) *BIT True*
numeral (*Num.Bit0 x*) *OR* − *numeral* (*Num.Bit0 y*) = (*numeral x OR* − *numeral y*) *BIT False*
numeral (*Num.Bit0 x*) *OR* − *numeral* (*Num.Bit1 y*) = (*numeral x OR* − *numeral* (*y* + *Num.One*)) *BIT True*
numeral (*Num.Bit1 x*) *OR* − *numeral* (*Num.Bit0 y*) = (*numeral x OR* − *numeral y*) *BIT True*
numeral (*Num.Bit1 x*) *OR* − *numeral* (*Num.Bit1 y*) = (*numeral x OR* − *numeral* (*y* + *Num.One*)) *BIT True*
− *numeral* (*Num.Bit0 x*) *OR numeral* (*Num.Bit0 y*) = (− *numeral x OR numeral y*) *BIT False*
− *numeral* (*Num.Bit0 x*) *OR numeral* (*Num.Bit1 y*) = (− *numeral x OR numeral y*) *BIT True*
− *numeral* (*Num.Bit1 x*) *OR numeral* (*Num.Bit0 y*) = (− *numeral* (*x* + *Num.One*) *OR numeral y*) *BIT True*
− *numeral* (*Num.Bit1 x*) *OR numeral* (*Num.Bit1 y*) = (− *numeral* (*x* + *Num.One*) *OR numeral y*) *BIT True*
− *numeral* (*Num.Bit0 x*) *OR* − *numeral* (*Num.Bit0 y*) = (− *numeral x OR* − *numeral y*) *BIT False*
− *numeral* (*Num.Bit0 x*) *OR* − *numeral* (*Num.Bit1 y*) = (− *numeral x OR* − *numeral* (*y* + *Num.One*)) *BIT True*
− *numeral* (*Num.Bit1 x*) *OR* − *numeral* (*Num.Bit0 y*) = (− *numeral* (*x* + *Num.One*) *OR* − *numeral y*) *BIT True*
− *numeral* (*Num.Bit1 x*) *OR* − *numeral* (*Num.Bit1 y*) = (− *numeral* (*x* + *Num.One*) *OR* − *numeral* (*y* + *Num.One*)) *BIT True*
(*1*::*int*) *OR numeral* (*Num.Bit0 y*) = *numeral* (*Num.Bit1 y*)
(*1*::*int*) *OR numeral* (*Num.Bit1 y*) = *numeral* (*Num.Bit1 y*)
(*1*::*int*) *OR* − *numeral* (*Num.Bit0 y*) = − *numeral* (*Num.BitM y*)
(*1*::*int*) *OR* − *numeral* (*Num.Bit1 y*) = − *numeral* (*Num.Bit1 y*)
numeral (*Num.Bit0 x*) *OR* (*1*::*int*) = *numeral* (*Num.Bit1 x*)
numeral (*Num.Bit1 x*) *OR* (*1*::*int*) = *numeral* (*Num.Bit1 x*)
− *numeral* (*Num.Bit0 x*) *OR* (*1*::*int*) = − *numeral* (*Num.BitM x*)
− *numeral* (*Num.Bit1 x*) *OR* (*1*::*int*) = − *numeral* (*Num.Bit1 x*)
by (*rule bin-rl-eqI*, *simp*, *simp*)+

lemma *int-xor-numerals* [*simp*]:
numeral (*Num.Bit0 x*) *XOR numeral* (*Num.Bit0 y*) = (*numeral x XOR numeral y*) *BIT False*
numeral (*Num.Bit0 x*) *XOR numeral* (*Num.Bit1 y*) = (*numeral x XOR numeral y*) *BIT True*
numeral (*Num.Bit1 x*) *XOR numeral* (*Num.Bit0 y*) = (*numeral x XOR numeral y*) *BIT True*
numeral (*Num.Bit1 x*) *XOR numeral* (*Num.Bit1 y*) = (*numeral x XOR numeral y*) *BIT False*
numeral (*Num.Bit0 x*) *XOR* − *numeral* (*Num.Bit0 y*) = (*numeral x XOR* − *numeral y*) *BIT False*
numeral (*Num.Bit0 x*) *XOR* − *numeral* (*Num.Bit1 y*) = (*numeral x XOR* −

numeral (*y* + *Num.One*)) *BIT True*
numeral (*Num.Bit1 x*) *XOR* − *numeral* (*Num.Bit0 y*) = (*numeral x XOR* − *numeral y*) *BIT True*
numeral (*Num.Bit1 x*) *XOR* − *numeral* (*Num.Bit1 y*) = (*numeral x XOR* − *numeral* (*y* + *Num.One*)) *BIT False*
− *numeral* (*Num.Bit0 x*) *XOR numeral* (*Num.Bit0 y*) = (− *numeral x XOR numeral y*) *BIT False*
− *numeral* (*Num.Bit0 x*) *XOR numeral* (*Num.Bit1 y*) = (− *numeral x XOR numeral y*) *BIT True*
− *numeral* (*Num.Bit1 x*) *XOR numeral* (*Num.Bit0 y*) = (− *numeral* (*x* + *Num.One*) *XOR numeral y*) *BIT True*
− *numeral* (*Num.Bit1 x*) *XOR numeral* (*Num.Bit1 y*) = (− *numeral* (*x* + *Num.One*) *XOR numeral y*) *BIT False*
− *numeral* (*Num.Bit0 x*) *XOR* − *numeral* (*Num.Bit0 y*) = (− *numeral x XOR* − *numeral y*) *BIT False*
− *numeral* (*Num.Bit0 x*) *XOR* − *numeral* (*Num.Bit1 y*) = (− *numeral x XOR* − *numeral* (*y* + *Num.One*)) *BIT True*
− *numeral* (*Num.Bit1 x*) *XOR* − *numeral* (*Num.Bit0 y*) = (− *numeral* (*x* + *Num.One*) *XOR* − *numeral y*) *BIT True*
− *numeral* (*Num.Bit1 x*) *XOR* − *numeral* (*Num.Bit1 y*) = (− *numeral* (*x* + *Num.One*) *XOR* − *numeral* (*y* + *Num.One*)) *BIT False*
(*1*::*int*) *XOR numeral* (*Num.Bit0 y*) = *numeral* (*Num.Bit1 y*)
(*1*::*int*) *XOR numeral* (*Num.Bit1 y*) = *numeral* (*Num.Bit0 y*)
(*1*::*int*) *XOR* − *numeral* (*Num.Bit0 y*) = − *numeral* (*Num.BitM y*)
(*1*::*int*) *XOR* − *numeral* (*Num.Bit1 y*) = − *numeral* (*Num.Bit0* (*y* + *Num.One*))
numeral (*Num.Bit0 x*) *XOR* (*1*::*int*) = *numeral* (*Num.Bit1 x*)
numeral (*Num.Bit1 x*) *XOR* (*1*::*int*) = *numeral* (*Num.Bit0 x*)
− *numeral* (*Num.Bit0 x*) *XOR* (*1*::*int*) = − *numeral* (*Num.BitM x*)
− *numeral* (*Num.Bit1 x*) *XOR* (*1*::*int*) = − *numeral* (*Num.Bit0* (*x* + *Num.One*))
by (*rule bin-rl-eqI*, *simp*, *simp*)+

122.1.5 Interactions with arithmetic

lemma *plus-and-or* [*rule-format*]:
ALL y::*int*. (*x AND y*) + (*x OR y*) = *x* + *y*
apply (*induct x rule*: *bin-induct*)
apply *clarsimp*
apply *clarsimp*
apply *clarsimp*
apply (*case-tac y rule*: *bin-exhaust*)
apply *clarsimp*
apply (*unfold Bit-def*)
apply *clarsimp*
apply (*erule-tac x* = *x* **in** *allE*)
apply *simp*
done

lemma *le-int-or*:
bin-sign (*y*::*int*) = *0* ==> *x* <= *x OR y*

```
  apply (induct y arbitrary: x rule: bin-induct)
    apply clarsimp
   apply clarsimp
  apply (case-tac x rule: bin-exhaust)
  apply (case-tac b)
   apply (case-tac [!] bit)
     apply (auto simp: le-Bits)
  done

lemmas int-and-le =
  xtrans(3) [OF bbw-ao-absorbs (2) [THEN conjunct2, symmetric] le-int-or]



lemma bin-add-not: x + NOT x = (−1::int)
  apply (induct x rule: bin-induct)
    apply clarsimp
   apply clarsimp
  apply (case-tac bit, auto)
  done
```

122.1.6 Truncating results of bit-wise operations

```
lemma bin-trunc-ao:
  !!x y. (bintrunc n x) AND (bintrunc n y) = bintrunc n (x AND y)
  !!x y. (bintrunc n x) OR (bintrunc n y) = bintrunc n (x OR y)
  by (auto simp add: bin-eq-iff bin-nth-ops nth-bintr)

lemma bin-trunc-xor:
  !!x y. bintrunc n (bintrunc n x XOR bintrunc n y) =
          bintrunc n (x XOR y)
  by (auto simp add: bin-eq-iff bin-nth-ops nth-bintr)

lemma bin-trunc-not:
  !!x. bintrunc n (NOT (bintrunc n x)) = bintrunc n (NOT x)
  by (auto simp add: bin-eq-iff bin-nth-ops nth-bintr)



lemma bintr-bintr-i:
  x = bintrunc n y ==> bintrunc n x = bintrunc n y
  by auto

lemmas bin-trunc-and = bin-trunc-ao(1) [THEN bintr-bintr-i]
lemmas bin-trunc-or = bin-trunc-ao(2) [THEN bintr-bintr-i]
```

122.2 Setting and clearing bits

```
primrec
  bin-sc :: nat => bool => int => int
where
```

Z: *bin-sc 0 b w* = *bin-rest w BIT b*
| *Suc*: *bin-sc* (*Suc n*) *b w* = *bin-sc n b* (*bin-rest w*) *BIT bin-last w*

lemma *bin-nth-sc* [*simp*]:
bin-nth (*bin-sc n b w*) *n* ⟷ *b*
by (*induct n arbitrary*: *w*) *auto*

lemma *bin-sc-sc-same* [*simp*]:
bin-sc n c (*bin-sc n b w*) = *bin-sc n c w*
by (*induct n arbitrary*: *w*) *auto*

lemma *bin-sc-sc-diff*:
m ~= *n* ==>
bin-sc m c (*bin-sc n b w*) = *bin-sc n b* (*bin-sc m c w*)
apply (*induct n arbitrary*: *w m*)
apply (*case-tac* [!] *m*)
apply *auto*
done

lemma *bin-nth-sc-gen*:
bin-nth (*bin-sc n b w*) *m* = (*if m* = *n then b else bin-nth w m*)
by (*induct n arbitrary*: *w m*) (*case-tac* [!] *m*, *auto*)

lemma *bin-sc-nth* [*simp*]:
(*bin-sc n* (*bin-nth w n*) *w*) = *w*
by (*induct n arbitrary*: *w*) *auto*

lemma *bin-sign-sc* [*simp*]:
bin-sign (*bin-sc n b w*) = *bin-sign w*
by (*induct n arbitrary*: *w*) *auto*

lemma *bin-sc-bintr* [*simp*]:
bintrunc m (*bin-sc n x* (*bintrunc m* (*w*))) = *bintrunc m* (*bin-sc n x w*)
apply (*induct n arbitrary*: *w m*)
apply (*case-tac* [!] *w rule*: *bin-exhaust*)
apply (*case-tac* [!] *m*, *auto*)
done

lemma *bin-clr-le*:
bin-sc n False w <= *w*
apply (*induct n arbitrary*: *w*)
apply (*case-tac* [!] *w rule*: *bin-exhaust*)
apply (*auto simp*: *le-Bits*)
done

lemma *bin-set-ge*:
bin-sc n True w >= *w*
apply (*induct n arbitrary*: *w*)
apply (*case-tac* [!] *w rule*: *bin-exhaust*)

```
  apply (auto simp: le-Bits)
 done

lemma bintr-bin-clr-le:
  bintrunc n (bin-sc m False w) <= bintrunc n w
  apply (induct n arbitrary: w m)
   apply simp
  apply (case-tac w rule: bin-exhaust)
  apply (case-tac m)
   apply (auto simp: le-Bits)
  done

lemma bintr-bin-set-ge:
  bintrunc n (bin-sc m True w) >= bintrunc n w
  apply (induct n arbitrary: w m)
   apply simp
  apply (case-tac w rule: bin-exhaust)
  apply (case-tac m)
   apply (auto simp: le-Bits)
  done

lemma bin-sc-FP [simp]: bin-sc n False 0 = 0
  by (induct n) auto

lemma bin-sc-TM [simp]: bin-sc n True (− 1) = − 1
  by (induct n) auto

lemmas bin-sc-simps = bin-sc.Z bin-sc.Suc bin-sc-TM bin-sc-FP

lemma bin-sc-minus:
  0 < n ==> bin-sc (Suc (n − 1)) b w = bin-sc n b w
  by auto

lemmas bin-sc-Suc-minus =
  trans [OF bin-sc-minus [symmetric] bin-sc.Suc]

lemma bin-sc-numeral [simp]:
  bin-sc (numeral k) b w =
    bin-sc (pred-numeral k) b (bin-rest w) BIT bin-last w
  by (simp add: numeral-eq-Suc)
```

122.3 Splitting and concatenation

```
definition bin-rcat :: nat ⇒ int list ⇒ int
where
  bin-rcat n = foldl (λu v. bin-cat u n v) 0

fun bin-rsplit-aux :: nat ⇒ nat ⇒ int ⇒ int list ⇒ int list
where
```

```
  bin-rsplit-aux n m c bs =
    (if m = 0 | n = 0 then bs else
      let (a, b) = bin-split n c
      in bin-rsplit-aux n (m - n) a (b # bs))

definition bin-rsplit :: nat ⇒ nat × int ⇒ int list
where
  bin-rsplit n w = bin-rsplit-aux n (fst w) (snd w) []

fun bin-rsplitl-aux :: nat ⇒ nat ⇒ int ⇒ int list ⇒ int list
where
  bin-rsplitl-aux n m c bs =
    (if m = 0 | n = 0 then bs else
      let (a, b) = bin-split (min m n) c
      in bin-rsplitl-aux n (m - n) a (b # bs))

definition bin-rsplitl :: nat ⇒ nat × int ⇒ int list
where
  bin-rsplitl n w = bin-rsplitl-aux n (fst w) (snd w) []

declare bin-rsplit-aux.simps [simp del]
declare bin-rsplitl-aux.simps [simp del]

lemma bin-sign-cat:
  bin-sign (bin-cat x n y) = bin-sign x
  by (induct n arbitrary: y) auto

lemma bin-cat-Suc-Bit:
  bin-cat w (Suc n) (v BIT b) = bin-cat w n v BIT b
  by auto

lemma bin-nth-cat:
  bin-nth (bin-cat x k y) n =
    (if n < k then bin-nth y n else bin-nth x (n - k))
  apply (induct k arbitrary: n y)
   apply clarsimp
  apply (case-tac n, auto)
  done

lemma bin-nth-split:
  bin-split n c = (a, b) ==>
    (ALL k. bin-nth a k = bin-nth c (n + k)) &
    (ALL k. bin-nth b k = (k < n & bin-nth c k))
  apply (induct n arbitrary: b c)
   apply clarsimp
  apply (clarsimp simp: Let-def split: prod.split-asm)
  apply (case-tac k)
  apply auto
  done
```

lemma *bin-cat-assoc*:
bin-cat (*bin-cat x m y*) *n z* = *bin-cat x* (*m* + *n*) (*bin-cat y n z*)
by (*induct n arbitrary*: *z*) *auto*

lemma *bin-cat-assoc-sym*:
bin-cat x m (*bin-cat y n z*) = *bin-cat* (*bin-cat x* (*m* − *n*) *y*) (*min m n*) *z*
apply (*induct n arbitrary*: *z m*, *clarsimp*)
apply (*case-tac m*, *auto*)
done

lemma *bin-cat-zero* [*simp*]: *bin-cat 0 n w* = *bintrunc n w*
by (*induct n arbitrary*: *w*) *auto*

lemma *bintr-cat1*:
bintrunc (*k* + *n*) (*bin-cat a n b*) = *bin-cat* (*bintrunc k a*) *n b*
by (*induct n arbitrary*: *b*) *auto*

lemma *bintr-cat*: *bintrunc m* (*bin-cat a n b*) =
bin-cat (*bintrunc* (*m* − *n*) *a*) *n* (*bintrunc* (*min m n*) *b*)
by (*rule bin-eqI*) (*auto simp*: *bin-nth-cat nth-bintr*)

lemma *bintr-cat-same* [*simp*]:
bintrunc n (*bin-cat a n b*) = *bintrunc n b*
by (*auto simp add* : *bintr-cat*)

lemma *cat-bintr* [*simp*]:
bin-cat a n (*bintrunc n b*) = *bin-cat a n b*
by (*induct n arbitrary*: *b*) *auto*

lemma *split-bintrunc*:
bin-split n c = (*a*, *b*) ==> *b* = *bintrunc n c*
by (*induct n arbitrary*: *b c*) (*auto simp*: *Let-def split*: *prod.split-asm*)

lemma *bin-cat-split*:
bin-split n w = (*u*, *v*) ==> *w* = *bin-cat u n v*
by (*induct n arbitrary*: *v w*) (*auto simp*: *Let-def split*: *prod.split-asm*)

lemma *bin-split-cat*:
bin-split n (*bin-cat v n w*) = (*v*, *bintrunc n w*)
by (*induct n arbitrary*: *w*) *auto*

lemma *bin-split-zero* [*simp*]: *bin-split n 0* = (*0*, *0*)
by (*induct n*) *auto*

lemma *bin-split-minus1* [*simp*]:
bin-split n (− *1*) = (− *1*, *bintrunc n* (− *1*))
by (*induct n*) *auto*

lemma *bin-split-trunc*:
bin-split (*min m n*) *c* = (*a*, *b*) ==>
bin-split n (*bintrunc m c*) = (*bintrunc* (*m* − *n*) *a*, *b*)
apply (*induct n arbitrary*: *m b c*, *clarsimp*)
apply (*simp add*: *bin-rest-trunc Let-def split*: *prod.split-asm*)
apply (*case-tac m*)
apply (*auto simp*: *Let-def split*: *prod.split-asm*)
done

lemma *bin-split-trunc1*:
bin-split n c = (*a*, *b*) ==>
bin-split n (*bintrunc m c*) = (*bintrunc* (*m* − *n*) *a*, *bintrunc m b*)
apply (*induct n arbitrary*: *m b c*, *clarsimp*)
apply (*simp add*: *bin-rest-trunc Let-def split*: *prod.split-asm*)
apply (*case-tac m*)
apply (*auto simp*: *Let-def split*: *prod.split-asm*)
done

lemma *bin-cat-num*:
bin-cat a n b = *a* ∗ *2* ˆ *n* + *bintrunc n b*
apply (*induct n arbitrary*: *b*, *clarsimp*)
apply (*simp add*: *Bit-def*)
done

lemma *bin-split-num*:
bin-split n b = (*b div 2* ˆ *n*, *b mod 2* ˆ *n*)
apply (*induct n arbitrary*: *b*, *simp*)
apply (*simp add*: *bin-rest-def zdiv-zmult2-eq*)
apply (*case-tac b rule*: *bin-exhaust*)
apply *simp*
apply (*simp add*: *Bit-def mod-mult-mult1 p1mod22k*)
done

122.4 Miscellaneous lemmas

lemma *nth-2p-bin*:
bin-nth (*2* ˆ *n*) *m* = (*m* = *n*)
apply (*induct n arbitrary*: *m*)
apply *clarsimp*
apply *safe*
apply (*case-tac m*)
apply (*auto simp*: *Bit-B0-2t* [*symmetric*])
done

lemma *ex-eq-or*:
(*EX m*. *n* = *Suc m* & (*m* = *k* | *P m*)) = (*n* = *Suc k* | (*EX m*. *n* = *Suc m* & *P m*))

by *auto*

lemma *power-BIT*: $2 \hat{}\ (Suc\ n) - 1 = (2 \hat{}\ n - 1)\ BIT\ True$
unfolding *Bit-B1*
by (*induct n*) *simp-all*

lemma *mod-BIT*:
$bin\ BIT\ bit\ mod\ 2 \hat{}\ Suc\ n = (bin\ mod\ 2 \hat{}\ n)\ BIT\ bit$
proof −
have $bin\ mod\ 2 \hat{}\ n < 2 \hat{}\ n$ **by** *simp*
then have $bin\ mod\ 2 \hat{}\ n \leq 2 \hat{}\ n - 1$ **by** *simp*
then have $2 * (bin\ mod\ 2 \hat{}\ n) \leq 2 * (2 \hat{}\ n - 1)$
by (*rule mult-left-mono*) *simp*
then have $2 * (bin\ mod\ 2 \hat{}\ n) + 1 < 2 * 2 \hat{}\ n$ **by** *simp*
then show *?thesis*
by (*auto simp add*: *Bit-def mod-mult-mult1 mod-add-left-eq* [*of* $2 * bin$]
mod-pos-pos-trivial)
qed

lemma *AND-mod*:
fixes x :: *int*
shows $x\ AND\ 2 \hat{}\ n - 1 = x\ mod\ 2 \hat{}\ n$
proof (*induct x arbitrary*: *n rule*: *bin-induct*)
case *1*
then show *?case*
by *simp*
next
case *2*
then show *?case*
by (*simp*, *simp add*: *m1mod2k*)
next
case (*3 bin bit*)
show *?case*
proof (*cases n*)
case *0*
then show *?thesis* **by** *simp*
next
case (*Suc m*)
with *3* **show** *?thesis*
by (*simp only*: *power-BIT mod-BIT int-and-Bits*) *simp*
qed
qed

end

123 Bool lists and integers

theory *Bool-List-Representation*
imports *Main Bits-Int*

begin

definition *map2* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'a\ list \Rightarrow 'b\ list \Rightarrow 'c\ list$
where
map2 f as bs = map (case-prod f) (zip as bs)

lemma *map2-Nil* [*simp*, *code*]:
map2 f [] ys = []
unfolding *map2-def* **by** *auto*

lemma *map2-Nil2* [*simp*, *code*]:
map2 f xs [] = []
unfolding *map2-def* **by** *auto*

lemma *map2-Cons* [*simp*, *code*]:
map2 f (x # xs) (y # ys) = f x y # map2 f xs ys
unfolding *map2-def* **by** *auto*

123.1 Operations on lists of booleans

primrec *bl-to-bin-aux* :: *bool list* ⇒ *int* ⇒ *int*
where
Nil: *bl-to-bin-aux [] w = w*
| *Cons*: *bl-to-bin-aux (b # bs) w =*
bl-to-bin-aux bs (w BIT b)

definition *bl-to-bin* :: *bool list* ⇒ *int*
where
bl-to-bin-def: *bl-to-bin bs = bl-to-bin-aux bs 0*

primrec *bin-to-bl-aux* :: *nat* ⇒ *int* ⇒ *bool list* ⇒ *bool list*
where
Z: *bin-to-bl-aux 0 w bl = bl*
| *Suc*: *bin-to-bl-aux (Suc n) w bl =*
bin-to-bl-aux n (bin-rest w) ((bin-last w) # bl)

definition *bin-to-bl* :: *nat* ⇒ *int* ⇒ *bool list*
where
bin-to-bl-def : *bin-to-bl n w = bin-to-bl-aux n w []*

primrec *bl-of-nth* :: *nat* ⇒ (*nat* ⇒ *bool*) ⇒ *bool list*
where
Suc: *bl-of-nth (Suc n) f = f n # bl-of-nth n f*
| *Z*: *bl-of-nth 0 f = []*

primrec *takefill* :: *'a* ⇒ *nat* ⇒ *'a list* ⇒ *'a list*
where
Z: *takefill fill 0 xs = []*
| *Suc*: *takefill fill (Suc n) xs = (*

case xs of [] => *fill* # *takefill fill n xs*
| *y* # *ys* => *y* # *takefill fill n ys*)

123.2 Arithmetic in terms of bool lists

Arithmetic operations in terms of the reversed bool list, assuming input list(s) the same length, and don't extend them.

primrec *rbl-succ* :: *bool list* => *bool list*
where
Nil: *rbl-succ Nil* = *Nil*
| *Cons*: *rbl-succ* (*x* # *xs*) = (*if x then False* # *rbl-succ xs else True* # *xs*)

primrec *rbl-pred* :: *bool list* => *bool list*
where
Nil: *rbl-pred Nil* = *Nil*
| *Cons*: *rbl-pred* (*x* # *xs*) = (*if x then False* # *xs else True* # *rbl-pred xs*)

primrec *rbl-add* :: *bool list* => *bool list* => *bool list*
where
— result is length of first arg, second arg may be longer
Nil: *rbl-add Nil x* = *Nil*
| *Cons*: *rbl-add* (*y* # *ys*) *x* = (*let ws* = *rbl-add ys* (*tl x*) *in*
(*y* ~= *hd x*) # (*if hd x* & *y then rbl-succ ws else ws*))

primrec *rbl-mult* :: *bool list* => *bool list* => *bool list*
where
— result is length of first arg, second arg may be longer
Nil: *rbl-mult Nil x* = *Nil*
| *Cons*: *rbl-mult* (*y* # *ys*) *x* = (*let ws* = *False* # *rbl-mult ys x in*
if y then rbl-add ws x else ws)

lemma *butlast-power*:
(*butlast* ^^ *n*) *bl* = *take* (*length bl* − *n*) *bl*
by (*induct n*) (*auto simp*: *butlast-take*)

lemma *bin-to-bl-aux-zero-minus-simp* [*simp*]:
0 < *n* $\Longrightarrow$ *bin-to-bl-aux n 0 bl* =
bin-to-bl-aux (*n* − *1*) *0* (*False* # *bl*)
by (*cases n*) *auto*

lemma *bin-to-bl-aux-minus1-minus-simp* [*simp*]:
0 < *n* ==> *bin-to-bl-aux n* (− *1*) *bl* =
bin-to-bl-aux (*n* − *1*) (− *1*) (*True* # *bl*)
by (*cases n*) *auto*

lemma *bin-to-bl-aux-one-minus-simp* [*simp*]:
0 < *n* $\Longrightarrow$ *bin-to-bl-aux n 1 bl* =
bin-to-bl-aux (*n* − *1*) *0* (*True* # *bl*)
by (*cases n*) *auto*

lemma *bin-to-bl-aux-Bit-minus-simp* [*simp*]:
0 < n ==> bin-to-bl-aux n (*w BIT b*) *bl* =
bin-to-bl-aux (*n* − *1*) *w* (*b* # *bl*)
by (*cases n*) *auto*

lemma *bin-to-bl-aux-Bit0-minus-simp* [*simp*]:
0 < n ==> bin-to-bl-aux n (*numeral* (*Num.Bit0 w*)) *bl* =
bin-to-bl-aux (*n* − *1*) (*numeral w*) (*False* # *bl*)
by (*cases n*) *auto*

lemma *bin-to-bl-aux-Bit1-minus-simp* [*simp*]:
0 < n ==> bin-to-bl-aux n (*numeral* (*Num.Bit1 w*)) *bl* =
bin-to-bl-aux (*n* − *1*) (*numeral w*) (*True* # *bl*)
by (*cases n*) *auto*

Link between bin and bool list.

lemma *bl-to-bin-aux-append*:
bl-to-bin-aux (*bs* @ *cs*) *w* = *bl-to-bin-aux cs* (*bl-to-bin-aux bs w*)
by (*induct bs arbitrary*: *w*) *auto*

lemma *bin-to-bl-aux-append*:
bin-to-bl-aux n w bs @ *cs* = *bin-to-bl-aux n w* (*bs* @ *cs*)
by (*induct n arbitrary*: *w bs*) *auto*

lemma *bl-to-bin-append*:
bl-to-bin (*bs* @ *cs*) = *bl-to-bin-aux cs* (*bl-to-bin bs*)
unfolding *bl-to-bin-def* **by** (*rule bl-to-bin-aux-append*)

lemma *bin-to-bl-aux-alt*:
bin-to-bl-aux n w bs = *bin-to-bl n w* @ *bs*
unfolding *bin-to-bl-def* **by** (*simp add* : *bin-to-bl-aux-append*)

lemma *bin-to-bl-0* [*simp*]: *bin-to-bl 0 bs* = []
unfolding *bin-to-bl-def* **by** *auto*

lemma *size-bin-to-bl-aux*:
size (*bin-to-bl-aux n w bs*) = *n* + *length bs*
by (*induct n arbitrary*: *w bs*) *auto*

lemma *size-bin-to-bl* [*simp*]: *size* (*bin-to-bl n w*) = *n*
unfolding *bin-to-bl-def* **by** (*simp add* : *size-bin-to-bl-aux*)

lemma *bin-bl-bin'*:
bl-to-bin (*bin-to-bl-aux n w bs*) =
bl-to-bin-aux bs (*bintrunc n w*)
by (*induct n arbitrary*: *w bs*) (*auto simp add* : *bl-to-bin-def*)

lemma *bin-bl-bin* [*simp*]: *bl-to-bin* (*bin-to-bl n w*) = *bintrunc n w*

unfolding *bin-to-bl-def bin-bl-bin′* **by** *auto*

lemma *bl-bin-bl′*:
bin-to-bl (*n* + *length bs*) (*bl-to-bin-aux bs w*) =
bin-to-bl-aux n w bs
apply (*induct bs arbitrary*: *w n*)
apply *auto*
apply (*simp-all only* : *add-Suc* [*symmetric*])
apply (*auto simp add* : *bin-to-bl-def*)
done

lemma *bl-bin-bl* [*simp*]: *bin-to-bl* (*length bs*) (*bl-to-bin bs*) = *bs*
unfolding *bl-to-bin-def*
apply (*rule box-equals*)
apply (*rule bl-bin-bl′*)
prefer *2*
apply (*rule bin-to-bl-aux.Z*)
apply *simp*
done

lemma *bl-to-bin-inj*:
bl-to-bin bs = *bl-to-bin cs* ==> *length bs* = *length cs* ==> *bs* = *cs*
apply (*rule-tac box-equals*)
defer
apply (*rule bl-bin-bl*)
apply (*rule bl-bin-bl*)
apply *simp*
done

lemma *bl-to-bin-False* [*simp*]: *bl-to-bin* (*False* # *bl*) = *bl-to-bin bl*
unfolding *bl-to-bin-def* **by** *auto*

lemma *bl-to-bin-Nil* [*simp*]: *bl-to-bin* [] = *0*
unfolding *bl-to-bin-def* **by** *auto*

lemma *bin-to-bl-zero-aux*:
bin-to-bl-aux n 0 bl = *replicate n False* @ *bl*
by (*induct n arbitrary*: *bl*) (*auto simp*: *replicate-app-Cons-same*)

lemma *bin-to-bl-zero*: *bin-to-bl n 0* = *replicate n False*
unfolding *bin-to-bl-def* **by** (*simp add*: *bin-to-bl-zero-aux*)

lemma *bin-to-bl-minus1-aux*:
bin-to-bl-aux n (− *1*) *bl* = *replicate n True* @ *bl*
by (*induct n arbitrary*: *bl*) (*auto simp*: *replicate-app-Cons-same*)

lemma *bin-to-bl-minus1*: *bin-to-bl n* (− *1*) = *replicate n True*
unfolding *bin-to-bl-def* **by** (*simp add*: *bin-to-bl-minus1-aux*)

lemma *bl-to-bin-rep-F*:
bl-to-bin (*replicate n False* @ *bl*) = *bl-to-bin bl*
apply (*simp add*: *bin-to-bl-zero-aux* [*symmetric*] *bin-bl-bin′*)
apply (*simp add*: *bl-to-bin-def*)
done

lemma *bin-to-bl-trunc* [*simp*]:
n <= *m* ==> *bin-to-bl n* (*bintrunc m w*) = *bin-to-bl n w*
by (*auto intro*: *bl-to-bin-inj*)

lemma *bin-to-bl-aux-bintr*:
bin-to-bl-aux n (*bintrunc m bin*) *bl* =
replicate (*n* − *m*) *False* @ *bin-to-bl-aux* (*min n m*) *bin bl*
apply (*induct n arbitrary*: *m bin bl*)
apply *clarsimp*
apply *clarsimp*
apply (*case-tac m*)
apply (*clarsimp simp*: *bin-to-bl-zero-aux*)
apply (*erule thin-rl*)
apply (*induct-tac n*)
apply *auto*
done

lemma *bin-to-bl-bintr*:
bin-to-bl n (*bintrunc m bin*) =
replicate (*n* − *m*) *False* @ *bin-to-bl* (*min n m*) *bin*
unfolding *bin-to-bl-def* **by** (*rule bin-to-bl-aux-bintr*)

lemma *bl-to-bin-rep-False*: *bl-to-bin* (*replicate n False*) = *0*
by (*induct n*) *auto*

lemma *len-bin-to-bl-aux*:
length (*bin-to-bl-aux n w bs*) = *n* + *length bs*
by (*fact size-bin-to-bl-aux*)

lemma *len-bin-to-bl* [*simp*]: *length* (*bin-to-bl n w*) = *n*
by (*fact size-bin-to-bl*)

lemma *sign-bl-bin′*:
bin-sign (*bl-to-bin-aux bs w*) = *bin-sign w*
by (*induct bs arbitrary*: *w*) *auto*

lemma *sign-bl-bin*: *bin-sign* (*bl-to-bin bs*) = *0*
unfolding *bl-to-bin-def* **by** (*simp add* : *sign-bl-bin′*)

lemma *bl-sbin-sign-aux*:
hd (*bin-to-bl-aux* (*Suc n*) *w bs*) =
(*bin-sign* (*sbintrunc n w*) = −*1*)
apply (*induct n arbitrary*: *w bs*)

```
  apply clarsimp
  apply (cases w rule: bin-exhaust)
  apply simp
 done

lemma bl-sbin-sign:
 hd (bin-to-bl (Suc n) w) = (bin-sign (sbintrunc n w) = −1)
 unfolding bin-to-bl-def by (rule bl-sbin-sign-aux)

lemma bin-nth-of-bl-aux:
 bin-nth (bl-to-bin-aux bl w) n =
   (n < size bl & rev bl ! n | n >= length bl & bin-nth w (n − size bl))
 apply (induct bl arbitrary: w)
  apply clarsimp
 apply clarsimp
 apply (cut-tac x=n and y=size bl in linorder-less-linear)
 apply (erule disjE, simp add: nth-append)+
 apply auto
 done

lemma bin-nth-of-bl: bin-nth (bl-to-bin bl) n = (n < length bl & rev bl ! n)
 unfolding bl-to-bin-def by (simp add : bin-nth-of-bl-aux)

lemma bin-nth-bl: n < m ⟹ bin-nth w n = nth (rev (bin-to-bl m w)) n
 apply (induct n arbitrary: m w)
  apply clarsimp
  apply (case-tac m, clarsimp)
  apply (clarsimp simp: bin-to-bl-def)
  apply (simp add: bin-to-bl-aux-alt)
 apply clarsimp
 apply (case-tac m, clarsimp)
 apply (clarsimp simp: bin-to-bl-def)
 apply (simp add: bin-to-bl-aux-alt)
 done

lemma nth-rev:
 n < length xs ⟹ rev xs ! n = xs ! (length xs − 1 − n)
 apply (induct xs)
  apply simp
 apply (clarsimp simp add : nth-append nth.simps split add : nat.split)
 apply (rule-tac f = λn. xs ! n in arg-cong)
 apply arith
 done

lemma nth-rev-alt: n < length ys ⟹ ys ! n = rev ys ! (length ys − Suc n)
 by (simp add: nth-rev)

lemma nth-bin-to-bl-aux:
 n < m + length bl ⟹ (bin-to-bl-aux m w bl) ! n =
```

```
  (if n < m then bin-nth w (m − 1 − n) else bl ! (n − m))
 apply (induct m arbitrary: w n bl)
  apply clarsimp
 apply clarsimp
 apply (case-tac w rule: bin-exhaust)
 apply simp
 done

lemma nth-bin-to-bl: n < m ==> (bin-to-bl m w) ! n = bin-nth w (m − Suc n)
 unfolding bin-to-bl-def by (simp add : nth-bin-to-bl-aux)

lemma bl-to-bin-lt2p-aux:
 bl-to-bin-aux bs w < (w + 1) * (2 ^ length bs)
 apply (induct bs arbitrary: w)
  apply clarsimp
 apply clarsimp
 apply (drule meta-spec, erule xtrans(8) [rotated], simp add: Bit-def)+
 done

lemma bl-to-bin-lt2p-drop:
 bl-to-bin bs < 2 ^ length (dropWhile Not bs)
proof (induct bs)
 case (Cons b bs) with bl-to-bin-lt2p-aux[where w=1]
 show ?case unfolding bl-to-bin-def by simp
qed simp

lemma bl-to-bin-lt2p: bl-to-bin bs < 2 ^ length bs
 by (metis bin-bl-bin bintr-lt2p bl-bin-bl)

lemma bl-to-bin-ge2p-aux:
 bl-to-bin-aux bs w >= w * (2 ^ length bs)
 apply (induct bs arbitrary: w)
  apply clarsimp
 apply clarsimp
  apply (drule meta-spec, erule order-trans [rotated],
        simp add: Bit-B0-2t Bit-B1-2t algebra-simps)+
  apply (simp add: Bit-def)
 done

lemma bl-to-bin-ge0: bl-to-bin bs >= 0
 apply (unfold bl-to-bin-def)
 apply (rule xtrans(4))
  apply (rule bl-to-bin-ge2p-aux)
 apply simp
 done

lemma butlast-rest-bin:
 butlast (bin-to-bl n w) = bin-to-bl (n − 1) (bin-rest w)
 apply (unfold bin-to-bl-def)
```

apply (*cases w rule*: *bin-exhaust*)
apply (*cases n*, *clarsimp*)
apply *clarsimp*
apply (*auto simp add*: *bin-to-bl-aux-alt*)
done

lemma *butlast-bin-rest*:
butlast bl = *bin-to-bl* (*length bl* − *Suc 0*) (*bin-rest* (*bl-to-bin bl*))
using *butlast-rest-bin* [**where** *w*=*bl-to-bin bl* **and** *n*=*length bl*] **by** *simp*

lemma *butlast-rest-bl2bin-aux*:
bl ~= [] ⟹
bl-to-bin-aux (*butlast bl*) *w* = *bin-rest* (*bl-to-bin-aux bl w*)
by (*induct bl arbitrary*: *w*) *auto*

lemma *butlast-rest-bl2bin*:
bl-to-bin (*butlast bl*) = *bin-rest* (*bl-to-bin bl*)
apply (*unfold bl-to-bin-def*)
apply (*cases bl*)
apply (*auto simp add*: *butlast-rest-bl2bin-aux*)
done

lemma *trunc-bl2bin-aux*:
bintrunc m (*bl-to-bin-aux bl w*) =
bl-to-bin-aux (*drop* (*length bl* − *m*) *bl*) (*bintrunc* (*m* − *length bl*) *w*)
proof (*induct bl arbitrary*: *w*)
case *Nil* **show** *?case* **by** *simp*
next
case (*Cons b bl*) **show** *?case*
proof (*cases m* − *length bl*)
case *0* **then have** *Suc* (*length bl*) − *m* = *Suc* (*length bl* − *m*) **by** *simp*
with *Cons* **show** *?thesis* **by** *simp*
next
case (*Suc n*) **then have** ∗: *m* − *Suc* (*length bl*) = *n* **by** *simp*
with *Suc Cons* **show** *?thesis* **by** *simp*
qed
qed

lemma *trunc-bl2bin*:
bintrunc m (*bl-to-bin bl*) = *bl-to-bin* (*drop* (*length bl* − *m*) *bl*)
unfolding *bl-to-bin-def* **by** (*simp add* : *trunc-bl2bin-aux*)

lemma *trunc-bl2bin-len* [*simp*]:
bintrunc (*length bl*) (*bl-to-bin bl*) = *bl-to-bin bl*
by (*simp add*: *trunc-bl2bin*)

lemma *bl2bin-drop*:
bl-to-bin (*drop k bl*) = *bintrunc* (*length bl* − *k*) (*bl-to-bin bl*)
apply (*rule trans*)

```
  prefer 2
  apply (rule trunc-bl2bin [symmetric])
 apply (cases k <= length bl)
  apply auto
 done

lemma nth-rest-power-bin:
 bin-nth ((bin-rest ^^ k) w) n = bin-nth w (n + k)
 apply (induct k arbitrary: n, clarsimp)
 apply clarsimp
 apply (simp only: bin-nth.Suc [symmetric] add-Suc)
 done

lemma take-rest-power-bin:
 m <= n ==> take m (bin-to-bl n w) = bin-to-bl m ((bin-rest ^^ (n - m)) w)
 apply (rule nth-equalityI)
  apply simp
 apply (clarsimp simp add: nth-bin-to-bl nth-rest-power-bin)
 done

lemma hd-butlast: size xs > 1 ==> hd (butlast xs) = hd xs
 by (cases xs) auto

lemma last-bin-last':
 size xs > 0 ⟹ last xs ⟷ bin-last (bl-to-bin-aux xs w)
 by (induct xs arbitrary: w) auto

lemma last-bin-last:
 size xs > 0 ==> last xs ⟷ bin-last (bl-to-bin xs)
 unfolding bl-to-bin-def by (erule last-bin-last')

lemma bin-last-last:
 bin-last w ⟷ last (bin-to-bl (Suc n) w)
 apply (unfold bin-to-bl-def)
 apply simp
 apply (auto simp add: bin-to-bl-aux-alt)
 done


lemma bl-xor-aux-bin:
 map2 (%x y. x ~= y) (bin-to-bl-aux n v bs) (bin-to-bl-aux n w cs) =
   bin-to-bl-aux n (v XOR w) (map2 (%x y. x ~= y) bs cs)
 apply (induct n arbitrary: v w bs cs)
  apply simp
 apply (case-tac v rule: bin-exhaust)
 apply (case-tac w rule: bin-exhaust)
 apply clarsimp
 apply (case-tac b)
```

apply *auto*
done

lemma *bl-or-aux-bin*:
map2 (*op* |) (*bin-to-bl-aux n v bs*) (*bin-to-bl-aux n w cs*) =
bin-to-bl-aux n (*v OR w*) (*map2* (*op* |) *bs cs*)
apply (*induct n arbitrary*: *v w bs cs*)
apply *simp*
apply (*case-tac v rule*: *bin-exhaust*)
apply (*case-tac w rule*: *bin-exhaust*)
apply *clarsimp*
done

lemma *bl-and-aux-bin*:
map2 (*op* &) (*bin-to-bl-aux n v bs*) (*bin-to-bl-aux n w cs*) =
bin-to-bl-aux n (*v AND w*) (*map2* (*op* &) *bs cs*)
apply (*induct n arbitrary*: *v w bs cs*)
apply *simp*
apply (*case-tac v rule*: *bin-exhaust*)
apply (*case-tac w rule*: *bin-exhaust*)
apply *clarsimp*
done

lemma *bl-not-aux-bin*:
map Not (*bin-to-bl-aux n w cs*) =
bin-to-bl-aux n (*NOT w*) (*map Not cs*)
apply (*induct n arbitrary*: *w cs*)
apply *clarsimp*
apply *clarsimp*
done

lemma *bl-not-bin*: *map Not* (*bin-to-bl n w*) = *bin-to-bl n* (*NOT w*)
unfolding *bin-to-bl-def* **by** (*simp add*: *bl-not-aux-bin*)

lemma *bl-and-bin*:
map2 (*op* $\wedge$) (*bin-to-bl n v*) (*bin-to-bl n w*) = *bin-to-bl n* (*v AND w*)
unfolding *bin-to-bl-def* **by** (*simp add*: *bl-and-aux-bin*)

lemma *bl-or-bin*:
map2 (*op* $\vee$) (*bin-to-bl n v*) (*bin-to-bl n w*) = *bin-to-bl n* (*v OR w*)
unfolding *bin-to-bl-def* **by** (*simp add*: *bl-or-aux-bin*)

lemma *bl-xor-bin*:
map2 ($\lambda x\ y.\ x \neq y$) (*bin-to-bl n v*) (*bin-to-bl n w*) = *bin-to-bl n* (*v XOR w*)
unfolding *bin-to-bl-def* **by** (*simp only*: *bl-xor-aux-bin map2-Nil*)

lemma *drop-bin2bl-aux*:
drop m (*bin-to-bl-aux n bin bs*) =
bin-to-bl-aux ($n - m$) *bin* (*drop* ($m - n$) *bs*)

apply (*induct n arbitrary*: *m bin bs*, *clarsimp*)
apply *clarsimp*
apply (*case-tac bin rule*: *bin-exhaust*)
apply (*case-tac m* <= *n*, *simp*)
apply (*case-tac m* − *n*, *simp*)
apply *simp*
apply (*rule-tac f* = %*nat*. *drop nat bs* **in** *arg-cong*)
apply *simp*
done

lemma *drop-bin2bl*: *drop m* (*bin-to-bl n bin*) = *bin-to-bl* (*n* − *m*) *bin*
unfolding *bin-to-bl-def* **by** (*simp add* : *drop-bin2bl-aux*)

lemma *take-bin2bl-lem1*:
take m (*bin-to-bl-aux m w bs*) = *bin-to-bl m w*
apply (*induct m arbitrary*: *w bs*, *clarsimp*)
apply *clarsimp*
apply (*simp add*: *bin-to-bl-aux-alt*)
apply (*simp add*: *bin-to-bl-def*)
apply (*simp add*: *bin-to-bl-aux-alt*)
done

lemma *take-bin2bl-lem*:
take m (*bin-to-bl-aux* (*m* + *n*) *w bs*) =
take m (*bin-to-bl* (*m* + *n*) *w*)
apply (*induct n arbitrary*: *w bs*)
apply (*simp-all* (*no-asm*) *add*: *bin-to-bl-def take-bin2bl-lem1*)
apply *simp*
done

lemma *bin-split-take*:
bin-split n c = (*a*, *b*) ⟹
bin-to-bl m a = *take m* (*bin-to-bl* (*m* + *n*) *c*)
apply (*induct n arbitrary*: *b c*)
apply *clarsimp*
apply (*clarsimp simp*: *Let-def split*: *prod.split-asm*)
apply (*simp add*: *bin-to-bl-def*)
apply (*simp add*: *take-bin2bl-lem*)
done

lemma *bin-split-take1*:
k = *m* + *n* ==> *bin-split n c* = (*a*, *b*) ==>
bin-to-bl m a = *take m* (*bin-to-bl k c*)
by (*auto elim*: *bin-split-take*)

lemma *nth-takefill*: *m* < *n* ⟹
takefill fill n l ! *m* = (*if m* < *length l then l* ! *m else fill*)
apply (*induct n arbitrary*: *m l*, *clarsimp*)
apply *clarsimp*

apply (*case-tac m*)
 apply (*simp split*: *list.split*)
apply (*simp split*: *list.split*)
done

lemma *takefill-alt*:
takefill fill n l = *take n l* @ *replicate* (*n* − *length l*) *fill*
by (*induct n arbitrary*: *l*) (*auto split*: *list.split*)

lemma *takefill-replicate* [*simp*]:
takefill fill n (*replicate m fill*) = *replicate n fill*
by (*simp add* : *takefill-alt replicate-add* [*symmetric*])

lemma *takefill-le′*:
n = *m* + *k* ⟹ *takefill x m* (*takefill x n l*) = *takefill x m l*
by (*induct m arbitrary*: *l n*) (*auto split*: *list.split*)

lemma *length-takefill* [*simp*]: *length* (*takefill fill n l*) = *n*
by (*simp add* : *takefill-alt*)

lemma *take-takefill′*:
!!*w n*. *n* = *k* + *m* ==> *take k* (*takefill fill n w*) = *takefill fill k w*
by (*induct k*) (*auto split add* : *list.split*)

lemma *drop-takefill*:
!!*w*. *drop k* (*takefill fill* (*m* + *k*) *w*) = *takefill fill m* (*drop k w*)
by (*induct k*) (*auto split add* : *list.split*)

lemma *takefill-le* [*simp*]:
m ≤ *n* ⟹ *takefill x m* (*takefill x n l*) = *takefill x m l*
by (*auto simp*: *le-iff-add takefill-le′*)

lemma *take-takefill* [*simp*]:
m ≤ *n* ⟹ *take m* (*takefill fill n w*) = *takefill fill m w*
by (*auto simp*: *le-iff-add take-takefill′*)

lemma *takefill-append*:
takefill fill (*m* + *length xs*) (*xs* @ *w*) = *xs* @ (*takefill fill m w*)
by (*induct xs*) *auto*

lemma *takefill-same′*:
l = *length xs* ==> *takefill fill l xs* = *xs*
by (*induct xs arbitrary*: *l*, *auto*)

lemmas *takefill-same* [*simp*] = *takefill-same′* [*OF refl*]

lemma *takefill-bintrunc*:
takefill False n bl = *rev* (*bin-to-bl n* (*bl-to-bin* (*rev bl*)))
apply (*rule nth-equalityI*)

```
  apply simp
 apply (clarsimp simp: nth-takefill nth-rev nth-bin-to-bl bin-nth-of-bl)
 done

lemma bl-bin-bl-rtf:
 bin-to-bl n (bl-to-bin bl) = rev (takefill False n (rev bl))
 by (simp add : takefill-bintrunc)

lemma bl-bin-bl-rep-drop:
 bin-to-bl n (bl-to-bin bl) =
   replicate (n - length bl) False @ drop (length bl - n) bl
 by (simp add: bl-bin-bl-rtf takefill-alt rev-take)

lemma tf-rev:
 n + k = m + length bl ==> takefill x m (rev (takefill y n bl)) =
   rev (takefill y m (rev (takefill x k (rev bl))))
 apply (rule nth-equalityI)
  apply (auto simp add: nth-takefill nth-rev)
 apply (rule-tac f = %n. bl ! n in arg-cong)
 apply arith
 done

lemma takefill-minus:
 0 < n ==> takefill fill (Suc (n - 1)) w = takefill fill n w
 by auto

lemmas takefill-Suc-cases =
 list.cases [THEN takefill.Suc [THEN trans]]

lemmas takefill-Suc-Nil = takefill-Suc-cases (1)
lemmas takefill-Suc-Cons = takefill-Suc-cases (2)

lemmas takefill-minus-simps = takefill-Suc-cases [THEN [2]
 takefill-minus [symmetric, THEN trans]]

lemma takefill-numeral-Nil [simp]:
 takefill fill (numeral k) [] = fill # takefill fill (pred-numeral k) []
 by (simp add: numeral-eq-Suc)

lemma takefill-numeral-Cons [simp]:
 takefill fill (numeral k) (x # xs) = x # takefill fill (pred-numeral k) xs
 by (simp add: numeral-eq-Suc)



lemma bl-to-bin-aux-cat:
 !!nv v. bl-to-bin-aux bs (bin-cat w nv v) =
   bin-cat w (nv + length bs) (bl-to-bin-aux bs v)
 apply (induct bs)
```

```
 apply simp
apply (simp add: bin-cat-Suc-Bit [symmetric] del: bin-cat.simps)
done

lemma bin-to-bl-aux-cat:
  !!w bs. bin-to-bl-aux (nv + nw) (bin-cat v nw w) bs =
    bin-to-bl-aux nv v (bin-to-bl-aux nw w bs)
  by (induct nw) auto

lemma bl-to-bin-aux-alt:
  bl-to-bin-aux bs w = bin-cat w (length bs) (bl-to-bin bs)
  using bl-to-bin-aux-cat [where nv = 0 and v = 0]
  unfolding bl-to-bin-def [symmetric] by simp

lemma bin-to-bl-cat:
  bin-to-bl (nv + nw) (bin-cat v nw w) =
    bin-to-bl-aux nv v (bin-to-bl nw w)
  unfolding bin-to-bl-def by (simp add: bin-to-bl-aux-cat)

lemmas bl-to-bin-aux-app-cat =
  trans [OF bl-to-bin-aux-append bl-to-bin-aux-alt]

lemmas bin-to-bl-aux-cat-app =
  trans [OF bin-to-bl-aux-cat bin-to-bl-aux-alt]

lemma bl-to-bin-app-cat:
  bl-to-bin (bsa @ bs) = bin-cat (bl-to-bin bsa) (length bs) (bl-to-bin bs)
  by (simp only: bl-to-bin-aux-app-cat bl-to-bin-def)

lemma bin-to-bl-cat-app:
  bin-to-bl (n + nw) (bin-cat w nw wa) = bin-to-bl n w @ bin-to-bl nw wa
  by (simp only: bin-to-bl-def bin-to-bl-aux-cat-app)


lemma bl-to-bin-app-cat-alt:
  bin-cat (bl-to-bin cs) n w = bl-to-bin (cs @ bin-to-bl n w)
  by (simp add : bl-to-bin-app-cat)

lemma mask-lem: (bl-to-bin (True # replicate n False)) =
    (bl-to-bin (replicate n True)) + 1
  apply (unfold bl-to-bin-def)
  apply (induct n)
   apply simp
  apply (simp only: Suc-eq-plus1 replicate-add
                  append-Cons [symmetric] bl-to-bin-aux-append)
  apply (simp add: Bit-B0-2t Bit-B1-2t)
  done
```

lemma *length-bl-of-nth* [*simp*]: *length* (*bl-of-nth n f*) = *n*
  by (*induct n*) *auto*

lemma *nth-bl-of-nth* [*simp*]:
  m < *n* ⟹ *rev* (*bl-of-nth n f*) ! *m* = *f m*
  apply (*induct n*)
   apply *simp*
  apply (*clarsimp simp add* : *nth-append*)
  apply (*rule-tac f* = *f* **in** *arg-cong*)
  apply *simp*
  done

lemma *bl-of-nth-inj*:
  (!!*k*. *k* < *n* ==> *f k* = *g k*) ==> *bl-of-nth n f* = *bl-of-nth n g*
  by (*induct n*) *auto*

lemma *bl-of-nth-nth-le*:
  n ≤ *length xs* ⟹ *bl-of-nth n* (*nth* (*rev xs*)) = *drop* (*length xs* − *n*) *xs*
  apply (*induct n arbitrary*: *xs*, *clarsimp*)
  apply *clarsimp*
  apply (*rule trans* [*OF* - *hd-Cons-tl*])
   apply (*frule Suc-le-lessD*)
   apply (*simp add*: *nth-rev trans* [*OF drop-Suc drop-tl*, *symmetric*])
   apply (*subst hd-drop-conv-nth*)
     apply *force*
    apply *simp-all*
  apply (*rule-tac f* = %*n*. *drop n xs* **in** *arg-cong*)
  apply *simp*
  done

lemma *bl-of-nth-nth* [*simp*]: *bl-of-nth* (*length xs*) (*op* ! (*rev xs*)) = *xs*
  by (*simp add*: *bl-of-nth-nth-le*)

lemma *size-rbl-pred*: *length* (*rbl-pred bl*) = *length bl*
  by (*induct bl*) *auto*

lemma *size-rbl-succ*: *length* (*rbl-succ bl*) = *length bl*
  by (*induct bl*) *auto*

lemma *size-rbl-add*:
  !!*cl*. *length* (*rbl-add bl cl*) = *length bl*
  by (*induct bl*) (*auto simp*: *Let-def size-rbl-succ*)

lemma *size-rbl-mult*:
  !!*cl*. *length* (*rbl-mult bl cl*) = *length bl*
  by (*induct bl*) (*auto simp add* : *Let-def size-rbl-add*)

lemmas *rbl-sizes* [*simp*] =
  size-rbl-pred size-rbl-succ size-rbl-add size-rbl-mult

lemmas *rbl-Nils* =
rbl-pred.Nil rbl-succ.Nil rbl-add.Nil rbl-mult.Nil

lemma *rbl-pred*:
rbl-pred (*rev* (*bin-to-bl n bin*)) = *rev* (*bin-to-bl n* (*bin* − *1*))
apply (*induct n arbitrary*: *bin*, *simp*)
apply (*unfold bin-to-bl-def*)
apply *clarsimp*
apply (*case-tac bin rule*: *bin-exhaust*)
apply (*case-tac b*)
apply (*clarsimp simp*: *bin-to-bl-aux-alt*)+
done

lemma *rbl-succ*:
rbl-succ (*rev* (*bin-to-bl n bin*)) = *rev* (*bin-to-bl n* (*bin* + *1*))
apply (*induct n arbitrary*: *bin*, *simp*)
apply (*unfold bin-to-bl-def*)
apply *clarsimp*
apply (*case-tac bin rule*: *bin-exhaust*)
apply (*case-tac b*)
apply (*clarsimp simp*: *bin-to-bl-aux-alt*)+
done

lemma *rbl-add*:
!!*bina binb*. *rbl-add* (*rev* (*bin-to-bl n bina*)) (*rev* (*bin-to-bl n binb*)) =
rev (*bin-to-bl n* (*bina* + *binb*))
apply (*induct n*, *simp*)
apply (*unfold bin-to-bl-def*)
apply *clarsimp*
apply (*case-tac bina rule*: *bin-exhaust*)
apply (*case-tac binb rule*: *bin-exhaust*)
apply (*case-tac b*)
apply (*case-tac* [!] *ba*)
apply (*auto simp*: *rbl-succ bin-to-bl-aux-alt Let-def ac-simps*)
done

lemma *rbl-add-app2*:
!!*blb*. *length blb* >= *length bla* ==>
rbl-add bla (*blb* @ *blc*) = *rbl-add bla blb*
apply (*induct bla*, *simp*)
apply *clarsimp*
apply (*case-tac blb*, *clarsimp*)
apply (*clarsimp simp*: *Let-def*)
done

lemma *rbl-add-take2*:
!!*blb*. *length blb* >= *length bla* ==>
rbl-add bla (*take* (*length bla*) *blb*) = *rbl-add bla blb*

```
  apply (induct bla, simp)
  apply clarsimp
  apply (case-tac blb, clarsimp)
  apply (clarsimp simp: Let-def)
  done

lemma rbl-add-long:
  m >= n ==> rbl-add (rev (bin-to-bl n bina)) (rev (bin-to-bl m binb)) =
    rev (bin-to-bl n (bina + binb))
  apply (rule box-equals [OF - rbl-add-take2 rbl-add])
   apply (rule-tac f = rbl-add (rev (bin-to-bl n bina)) in arg-cong)
   apply (rule rev-swap [THEN iffD1])
   apply (simp add: rev-take drop-bin2bl)
  apply simp
  done

lemma rbl-mult-app2:
  !!blb. length blb >= length bla ==>
    rbl-mult bla (blb @ blc) = rbl-mult bla blb
  apply (induct bla, simp)
  apply clarsimp
  apply (case-tac blb, clarsimp)
  apply (clarsimp simp: Let-def rbl-add-app2)
  done

lemma rbl-mult-take2:
  length blb >= length bla ==>
    rbl-mult bla (take (length bla) blb) = rbl-mult bla blb
  apply (rule trans)
   apply (rule rbl-mult-app2 [symmetric])
   apply simp
  apply (rule-tac f = rbl-mult bla in arg-cong)
  apply (rule append-take-drop-id)
  done

lemma rbl-mult-gt1:
  m >= length bl ==> rbl-mult bl (rev (bin-to-bl m binb)) =
    rbl-mult bl (rev (bin-to-bl (length bl) binb))
  apply (rule trans)
   apply (rule rbl-mult-take2 [symmetric])
   apply simp-all
  apply (rule-tac f = rbl-mult bl in arg-cong)
  apply (rule rev-swap [THEN iffD1])
  apply (simp add: rev-take drop-bin2bl)
  done

lemma rbl-mult-gt:
  m > n ==> rbl-mult (rev (bin-to-bl n bina)) (rev (bin-to-bl m binb)) =
    rbl-mult (rev (bin-to-bl n bina)) (rev (bin-to-bl n binb))
```

```
  by (auto intro: trans [OF rbl-mult-gt1])

lemmas rbl-mult-Suc = lessI [THEN rbl-mult-gt]

lemma rbbl-Cons:
  b # rev (bin-to-bl n x) = rev (bin-to-bl (Suc n) (x BIT b))
  apply (unfold bin-to-bl-def)
  apply simp
  apply (simp add: bin-to-bl-aux-alt)
  done

lemma rbl-mult: !!bina binb.
    rbl-mult (rev (bin-to-bl n bina)) (rev (bin-to-bl n binb)) =
    rev (bin-to-bl n (bina * binb))
  apply (induct n)
   apply simp
  apply (unfold bin-to-bl-def)
  apply clarsimp
  apply (case-tac bina rule: bin-exhaust)
  apply (case-tac binb rule: bin-exhaust)
  apply (case-tac b)
   apply (case-tac [!] ba)
     apply (auto simp: bin-to-bl-aux-alt Let-def)
     apply (auto simp: rbbl-Cons rbl-mult-Suc rbl-add)
  done

lemma rbl-add-split:
  P (rbl-add (y # ys) (x # xs)) =
    (ALL ws. length ws = length ys --> ws = rbl-add ys xs -->
   (y --> ((x --> P (False # rbl-succ ws)) & (~ x -->  P (True # ws)))) &
    (~ y --> P (x # ws)))
  apply (auto simp add: Let-def)
   apply (case-tac [!] y)
     apply auto
  done

lemma rbl-mult-split:
  P (rbl-mult (y # ys) xs) =
    (ALL ws. length ws = Suc (length ys) --> ws = False # rbl-mult ys xs -->

    (y --> P (rbl-add ws xs)) & (~ y -->  P ws))
  by (clarsimp simp add : Let-def)
```

123.3 Repeated splitting or concatenation

```
lemma sclem:
  size (concat (map (bin-to-bl n) xs)) = length xs * n
  by (induct xs) auto
```

lemma *bin-cat-foldl-lem*:
foldl (%*u*. *bin-cat u n*) *x xs* =
bin-cat x (*size xs* ∗ *n*) (*foldl* (%*u*. *bin-cat u n*) *y xs*)
apply (*induct xs arbitrary*: *x*)
apply *simp*
apply (*simp* (*no-asm*))
apply (*frule asm-rl*)
apply (*drule meta-spec*)
apply (*erule trans*)
apply (*drule-tac x* = *bin-cat y n a* **in** *meta-spec*)
apply (*simp add* : *bin-cat-assoc-sym min.absorb2*)
done

lemma *bin-rcat-bl*:
(*bin-rcat n wl*) = *bl-to-bin* (*concat* (*map* (*bin-to-bl n*) *wl*))
apply (*unfold bin-rcat-def*)
apply (*rule sym*)
apply (*induct wl*)
apply (*auto simp add* : *bl-to-bin-append*)
apply (*simp add* : *bl-to-bin-aux-alt sclem*)
apply (*simp add* : *bin-cat-foldl-lem* [*symmetric*])
done

lemmas *bin-rsplit-aux-simps* = *bin-rsplit-aux.simps bin-rsplitl-aux.simps*
lemmas *rsplit-aux-simps* = *bin-rsplit-aux-simps*

lemmas *th-if-simp1* = *if-split* [**where** *P* = *op* = *l*, *THEN iffD1*, *THEN conjunct1*, *THEN mp*] **for** *l*
lemmas *th-if-simp2* = *if-split* [**where** *P* = *op* = *l*, *THEN iffD1*, *THEN conjunct2*, *THEN mp*] **for** *l*

lemmas *rsplit-aux-simp1s* = *rsplit-aux-simps* [*THEN th-if-simp1*]

lemmas *rsplit-aux-simp2ls* = *rsplit-aux-simps* [*THEN th-if-simp2*]

lemmas *bin-rsplit-aux-simp2s* [*simp*] = *rsplit-aux-simp2ls* [*unfolded Let-def*]
lemmas *rbscl* = *bin-rsplit-aux-simp2s* (*2*)

lemmas *rsplit-aux-0-simps* [*simp*] =
rsplit-aux-simp1s [*OF disjI1*] *rsplit-aux-simp1s* [*OF disjI2*]

lemma *bin-rsplit-aux-append*:
bin-rsplit-aux n m c (*bs* @ *cs*) = *bin-rsplit-aux n m c bs* @ *cs*
apply (*induct n m c bs rule*: *bin-rsplit-aux.induct*)
apply (*subst bin-rsplit-aux.simps*)
apply (*subst bin-rsplit-aux.simps*)
apply (*clarsimp split*: *prod.split*)
done

lemma *bin-rsplitl-aux-append*:
bin-rsplitl-aux n m c (*bs* @ *cs*) = *bin-rsplitl-aux n m c bs* @ *cs*
apply (*induct n m c bs rule*: *bin-rsplitl-aux.induct*)
apply (*subst bin-rsplitl-aux.simps*)
apply (*subst bin-rsplitl-aux.simps*)
apply (*clarsimp split*: *prod.split*)
done

lemmas *rsplit-aux-apps* [**where** *bs* = []] =
bin-rsplit-aux-append bin-rsplitl-aux-append

lemmas *rsplit-def-auxs* = *bin-rsplit-def bin-rsplitl-def*

lemmas *rsplit-aux-alts* = *rsplit-aux-apps*
[*unfolded append-Nil rsplit-def-auxs* [*symmetric*]]

lemma *bin-split-minus*: *0* < *n* ==> *bin-split* (*Suc* (*n* − *1*)) *w* = *bin-split n w*
by *auto*

lemmas *bin-split-minus-simp* =
bin-split.Suc [*THEN* [*2*] *bin-split-minus* [*symmetric*, *THEN trans*]]

lemma *bin-split-pred-simp* [*simp*]:
(*0*::*nat*) < *numeral bin* ⟹
bin-split (*numeral bin*) *w* =
(*let* (*w1*, *w2*) = *bin-split* (*numeral bin* − *1*) (*bin-rest w*)
in (*w1*, *w2 BIT bin-last w*))
by (*simp only*: *bin-split-minus-simp*)

lemma *bin-rsplit-aux-simp-alt*:
bin-rsplit-aux n m c bs =
(*if m* = *0* ∨ *n* = *0*
then bs
else let (*a*, *b*) = *bin-split n c in bin-rsplit n* (*m* − *n*, *a*) @ *b* # *bs*)
unfolding *bin-rsplit-aux.simps* [*of n m c bs*]
apply *simp*
apply (*subst rsplit-aux-alts*)
apply (*simp add*: *bin-rsplit-def*)
done

lemmas *bin-rsplit-simp-alt* =
trans [*OF bin-rsplit-def bin-rsplit-aux-simp-alt*]

lemmas *bthrs* = *bin-rsplit-simp-alt* [*THEN* [*2*] *trans*]

lemma *bin-rsplit-size-sign'* [*rule-format*] :
⟦*n* > *0*; *rev sw* = *bin-rsplit n* (*nw*, *w*)⟧ ⟹
(*ALL v*: *set sw*. *bintrunc n v* = *v*)
apply (*induct sw arbitrary*: *nw w*)

```
  apply clarsimp
 apply clarsimp
 apply (drule bthrs)
 apply (simp (no-asm-use) add: Let-def split: prod.split-asm if-split-asm)
 apply clarify
 apply (drule split-bintrunc)
 apply simp
 done
```

lemmas *bin-rsplit-size-sign* = *bin-rsplit-size-sign'* [*OF asm-rl*
rev-rev-ident [*THEN trans*] *set-rev* [*THEN equalityD2* [*THEN subsetD*]]]

```
lemma bin-nth-rsplit [rule-format] :
 n > 0 ==> m < n ==> (ALL w k nw. rev sw = bin-rsplit n (nw, w) -->
     k < size sw --> bin-nth (sw ! k) m = bin-nth w (k * n + m))
 apply (induct sw)
  apply clarsimp
 apply clarsimp
 apply (drule bthrs)
 apply (simp (no-asm-use) add: Let-def split: prod.split-asm if-split-asm)
 apply clarify
 apply (erule allE, erule impE, erule exI)
 apply (case-tac k)
  apply clarsimp
  prefer 2
  apply clarsimp
  apply (erule allE)
  apply (erule (1) impE)
  apply (drule bin-nth-split, erule conjE, erule allE,
        erule trans, simp add : ac-simps)+
 done
```

```
lemma bin-rsplit-all:
 0 < nw ==> nw <= n ==> bin-rsplit n (nw, w) = [bintrunc n w]
 unfolding bin-rsplit-def
 by (clarsimp dest!: split-bintrunc simp: rsplit-aux-simp2ls split: prod.split)
```

```
lemma bin-rsplit-l [rule-format] :
 ALL bin. bin-rsplitl n (m, bin) = bin-rsplit n (m, bintrunc m bin)
 apply (rule-tac a = m in wf-less-than [THEN wf-induct])
 apply (simp (no-asm) add : bin-rsplitl-def bin-rsplit-def)
 apply (rule allI)
 apply (subst bin-rsplitl-aux.simps)
 apply (subst bin-rsplit-aux.simps)
 apply (clarsimp simp: Let-def split: prod.split)
 apply (drule bin-split-trunc)
 apply (drule sym [THEN trans], assumption)
 apply (subst rsplit-aux-alts(1))
 apply (subst rsplit-aux-alts(2))
```

```
  apply clarsimp
  unfolding bin-rsplit-def bin-rsplitl-def
  apply simp
  done

lemma bin-rsplit-rcat [rule-format] :
  n > 0 --> bin-rsplit n (n * size ws, bin-rcat n ws) = map (bintrunc n) ws
  apply (unfold bin-rsplit-def bin-rcat-def)
  apply (rule-tac xs = ws in rev-induct)
   apply clarsimp
  apply clarsimp
  apply (subst rsplit-aux-alts)
  unfolding bin-split-cat
  apply simp
  done

lemma bin-rsplit-aux-len-le [rule-format] :
  ∀ ws m. n ≠ 0 ⟶ ws = bin-rsplit-aux n nw w bs ⟶
    length ws ≤ m ⟷ nw + length bs * n ≤ m * n
proof −
  { fix i j j' k k' m :: nat and R
    assume d: (i::nat) ≤ j ∨ m < j'
    assume R1: i * k ≤ j * k ⟹ R
    assume R2: Suc m * k' ≤ j' * k' ⟹ R
    have R using d
      apply safe
       apply (rule R1, erule mult-le-mono1)
      apply (rule R2, erule Suc-le-eq [THEN iffD2 [THEN mult-le-mono1]])
      done
  } note A = this
  { fix sc m n lb :: nat
    have (0::nat) < sc ⟹ sc − n + (n + lb * n) ≤ m * n ⟷ sc + lb * n ≤
m * n
      apply safe
       apply arith
      apply (case-tac sc >= n)
       apply arith
      apply (insert linorder-le-less-linear [of m lb])
      apply (erule-tac k2=n and k'2=n in A)
       apply arith
      apply simp
      done
  } note B = this
  show ?thesis
    apply (induct n nw w bs rule: bin-rsplit-aux.induct)
    apply (subst bin-rsplit-aux.simps)
    apply (simp add: B Let-def split: prod.split)
    done
qed
```

lemma *bin-rsplit-len-le*:
  n ≠ *0* −−> *ws* = *bin-rsplit n* (*nw*, *w*) −−> (*length ws* <= *m*) = (*nw* <= *m* ∗ *n*)
  unfolding *bin-rsplit-def* **by** (*clarsimp simp add* : *bin-rsplit-aux-len-le*)

lemma *bin-rsplit-aux-len*:
  n ≠ *0* ⟹ *length* (*bin-rsplit-aux n nw w cs*) =
    (*nw* + *n* − *1*) *div n* + *length cs*
  apply (*induct n nw w cs rule*: *bin-rsplit-aux.induct*)
  apply (*subst bin-rsplit-aux.simps*)
  apply (*clarsimp simp*: *Let-def split*: *prod.split*)
  apply (*erule thin-rl*)
  apply (*case-tac m*)
  apply *simp*
  apply (*case-tac m* <= *n*)
  apply *auto*
  done

lemma *bin-rsplit-len*:
  n≠*0* ==> *length* (*bin-rsplit n* (*nw*, *w*)) = (*nw* + *n* − *1*) *div n*
  unfolding *bin-rsplit-def* **by** (*clarsimp simp add* : *bin-rsplit-aux-len*)

lemma *bin-rsplit-aux-len-indep*:
  n ≠ *0* ⟹ *length bs* = *length cs* ⟹
    length (*bin-rsplit-aux n nw v bs*) =
    length (*bin-rsplit-aux n nw w cs*)
proof (*induct n nw w cs arbitrary*: *v bs rule*: *bin-rsplit-aux.induct*)
  case (*1 n m w cs v bs*) **show** *?case*
  proof (*cases m* = *0*)
    case *True* **then show** *?thesis* **using** ‹*length bs* = *length cs*› **by** *simp*
  next
    case *False*
    from *1.hyps* ‹*m* ≠ *0*› ‹*n* ≠ *0*› **have** *hyp*: ⋀*v bs*. *length bs* = *Suc* (*length cs*) ⟹
      length (*bin-rsplit-aux n* (*m* − *n*) *v bs*) =
      length (*bin-rsplit-aux n* (*m* − *n*) (*fst* (*bin-split n w*)) (*snd* (*bin-split n w*) # *cs*))
    by *auto*
    show *?thesis* **using** ‹*length bs* = *length cs*› ‹*n* ≠ *0*›
      by (*auto simp add*: *bin-rsplit-aux-simp-alt Let-def bin-rsplit-len*
        split: *prod.split*)
  qed
qed

lemma *bin-rsplit-len-indep*:
  n≠*0* ==> *length* (*bin-rsplit n* (*nw*, *v*)) = *length* (*bin-rsplit n* (*nw*, *w*))
  apply (*unfold bin-rsplit-def*)
  apply (*simp* (*no-asm*))

apply (*erule bin-rsplit-aux-len-indep*)
apply (*rule refl*)
done

Even more bit operations

instantiation *int* :: *bitss*
begin

definition [*iff*]:
i !! $n \longleftrightarrow$ *bin-nth* i n

definition
lsb $i = (i :: int)$!! *0*

definition
set-bit i n b = *bin-sc* n b i

definition
set-bits f =
(*if* $\exists n. \forall n' \geq n. \neg f\ n'$ *then*
let n = *LEAST* $n. \forall n' \geq n. \neg f\ n'$
in *bl-to-bin* (*rev* (*map* f [*0*..<n]))
else if $\exists n. \forall n' \geq n. f\ n'$ *then*
let n = *LEAST* $n. \forall n' \geq n. f\ n'$
in *sbintrunc* n (*bl-to-bin* (*True* # *rev* (*map* f [*0*..<n])))
else *0* :: *int*)

definition
shiftl x $n = (x :: int) * 2$ ^ n

definition
shiftr x $n = (x :: int)$ *div* *2* ^ n

definition
msb $x \longleftrightarrow (x :: int) < 0$

instance ..

end

end

124 Type Definition Theorems

theory *Misc-Typedef*
imports *Main*
begin

125 More lemmas about normal type definitions

lemma
tdD1: *type-definition Rep Abs A* $\Longrightarrow \forall x.$ *Rep* $x \in A$ **and**
tdD2: *type-definition Rep Abs A* $\Longrightarrow \forall x.$ *Abs* (*Rep* x) $= x$ **and**
tdD3: *type-definition Rep Abs A* $\Longrightarrow \forall y.\ y \in A \longrightarrow$ *Rep* (*Abs* y) $= y$
by (*auto simp*: *type-definition-def*)

lemma *td-nat-int*:
type-definition int nat (*Collect* (*op* <= *0*))
unfolding *type-definition-def* **by** *auto*

context *type-definition*
begin

declare *Rep* [*iff*] *Rep-inverse* [*simp*] *Rep-inject* [*simp*]

lemma *Abs-eqD*: *Abs* x = *Abs* y ==> $x \in A$ ==> $y \in A$ ==> $x = y$
by (*simp add*: *Abs-inject*)

lemma *Abs-inverse'*:
r : A ==> *Abs* $r = a$ ==> *Rep* $a = r$
by (*safe elim*!: *Abs-inverse*)

lemma *Rep-comp-inverse*:
Rep o $f = g$ ==> *Abs o* $g = f$
using *Rep-inverse* **by** *auto*

lemma *Rep-eqD* [*elim*!]: *Rep* x = *Rep* y ==> $x = y$
by *simp*

lemma *Rep-inverse'*: *Rep* $a = r$ ==> *Abs* $r = a$
by (*safe intro*!: *Rep-inverse*)

lemma *comp-Abs-inverse*:
f *o Abs* $= g$ ==> g *o Rep* $= f$
using *Rep-inverse* **by** *auto*

lemma *set-Rep*:
A = *range Rep*
proof (*rule set-eqI*)
fix x
show ($x \in A$) = ($x \in$ *range Rep*)
by (*auto dest*: *Abs-inverse* [*of* x, *symmetric*])
qed

lemma *set-Rep-Abs*: A = *range* (*Rep o Abs*)
proof (*rule set-eqI*)
fix x

```
  show (x ∈ A) = (x ∈ range (Rep o Abs))
    by (auto dest: Abs-inverse [of x, symmetric])
qed

lemma Abs-inj-on: inj-on Abs A
  unfolding inj-on-def
  by (auto dest: Abs-inject [THEN iffD1])

lemma image: Abs ‘ A = UNIV
  by (auto intro!: image-eqI)

lemmas td-thm = type-definition-axioms

lemma fns1:
  Rep o fa = fr o Rep | fa o Abs = Abs o fr ==> Abs o fr o Rep = fa
  by (auto dest: Rep-comp-inverse elim: comp-Abs-inverse simp: o-assoc)

lemmas fns1a = disjI1 [THEN fns1]
lemmas fns1b = disjI2 [THEN fns1]

lemma fns4:
  Rep o fa o Abs = fr ==>
   Rep o fa = fr o Rep & fa o Abs = Abs o fr
  by auto

end

interpretation nat-int: type-definition int nat Collect (op <= 0)
  by (rule td-nat-int)

declare
  nat-int.Rep-cases [cases del]
  nat-int.Abs-cases [cases del]
  nat-int.Rep-induct [induct del]
  nat-int.Abs-induct [induct del]
```

125.1 Extended form of type definition predicate

```
lemma td-conds:
  norm o norm = norm ==> (fr o norm = norm o fr) =
    (norm o fr o norm = fr o norm & norm o fr o norm = norm o fr)
  apply safe
    apply (simp-all add: comp-assoc)
   apply (simp-all add: o-assoc)
  done

lemma fn-comm-power:
  fa o tr = tr o fr ==> fa ^^ n o tr = tr o fr ^^ n
  apply (rule ext)
```

apply (*induct n*)
 apply (*auto dest*: *fun-cong*)
done

lemmas *fn-comm-power'* =
ext [*THEN fn-comm-power*, *THEN fun-cong*, *unfolded o-def*]

locale *td-ext* = *type-definition* +
fixes *norm*
assumes *eq-norm*: $\bigwedge x$. *Rep* (*Abs x*) = *norm x*
begin

lemma *Abs-norm* [*simp*]:
Abs (*norm x*) = *Abs x*
using *eq-norm* [*of x*] **by** (*auto elim*: *Rep-inverse'*)

lemma *td-th*:
g o Abs = *f* ==> *f* (*Rep x*) = *g x*
by (*drule comp-Abs-inverse* [*symmetric*]) *simp*

lemma *eq-norm'*: *Rep o Abs* = *norm*
by (*auto simp*: *eq-norm*)

lemma *norm-Rep* [*simp*]: *norm* (*Rep x*) = *Rep x*
by (*auto simp*: *eq-norm' intro*: *td-th*)

lemmas *td* = *td-thm*

lemma *set-iff-norm*: *w* : *A* $\longleftrightarrow$ *w* = *norm w*
by (*auto simp*: *set-Rep-Abs eq-norm' eq-norm* [*symmetric*])

lemma *inverse-norm*:
(*Abs n* = *w*) = (*Rep w* = *norm n*)
apply (*rule iffI*)
 apply (*clarsimp simp add*: *eq-norm*)
apply (*simp add*: *eq-norm'* [*symmetric*])
done

lemma *norm-eq-iff*:
(*norm x* = *norm y*) = (*Abs x* = *Abs y*)
by (*simp add*: *eq-norm'* [*symmetric*])

lemma *norm-comps*:
Abs o norm = *Abs*
norm o Rep = *Rep*
norm o norm = *norm*
by (*auto simp*: *eq-norm'* [*symmetric*] *o-def*)

lemmas *norm-norm* [*simp*] = *norm-comps*

lemma *fns5*:
Rep o fa o Abs = *fr* ==>
fr o norm = *fr* & *norm o fr* = *fr*
by (*fold eq-norm′*) *auto*

lemma *fns2*:
Abs o fr o Rep = *fa* ==>
(*norm o fr o norm* = *fr o norm*) = (*Rep o fa* = *fr o Rep*)
apply (*fold eq-norm′*)
apply *safe*
prefer *2*
apply (*simp add*: *o-assoc*)
apply (*rule ext*)
apply (*drule-tac x*=*Rep x* **in** *fun-cong*)
apply *auto*
done

lemma *fns3*:
Abs o fr o Rep = *fa* ==>
(*norm o fr o norm* = *norm o fr*) = (*fa o Abs* = *Abs o fr*)
apply (*fold eq-norm′*)
apply *safe*
prefer *2*
apply (*simp add*: *comp-assoc*)
apply (*rule ext*)
apply (*drule-tac f*=*a o b* **for** *a b* **in** *fun-cong*)
apply *simp*
done

lemma *fns*:
fr o norm = *norm o fr* ==>
(*fa o Abs* = *Abs o fr*) = (*Rep o fa* = *fr o Rep*)
apply *safe*
apply (*frule fns1b*)
prefer *2*
apply (*frule fns1a*)
apply (*rule fns3* [*THEN iffD1*])
prefer *3*
apply (*rule fns2* [*THEN iffD1*])
apply (*simp-all add*: *comp-assoc*)
apply (*simp-all add*: *o-assoc*)
done

lemma *range-norm*:
range (*Rep o Abs*) = *A*
by (*simp add*: *set-Rep-Abs*)

end

lemmas *td-ext-def'* =
td-ext-def [*unfolded type-definition-def td-ext-axioms-def*]

end

126 Miscellaneous lemmas, of at least doubtful value

theory *Word-Miscellaneous*
imports *Main ~~/src/HOL/Library/Bit Misc-Numeric*
begin

lemma *power-minus-simp*:
$0 < n \Longrightarrow a \hat{} n = a * a \hat{} (n - 1)$
by (*auto dest*: *gr0-implies-Suc*)

lemma *funpow-minus-simp*:
$0 < n \Longrightarrow f \hat{}\hat{} n = f \circ f \hat{}\hat{} (n - 1)$
by (*auto dest*: *gr0-implies-Suc*)

lemma *power-numeral*:
a ^ *numeral k* = *a* ∗ *a* ^ (*pred-numeral k*)
by (*simp add*: *numeral-eq-Suc*)

lemma *funpow-numeral* [*simp*]:
f ^^ *numeral k* = *f* ∘ *f* ^^ (*pred-numeral k*)
by (*simp add*: *numeral-eq-Suc*)

lemma *replicate-numeral* [*simp*]:
replicate (*numeral k*) *x* = *x* # *replicate* (*pred-numeral k*) *x*
by (*simp add*: *numeral-eq-Suc*)

lemma *rco-alt*: (*f o g*) ^^ *n o f* = *f o* (*g o f*) ^^ *n*
apply (*rule ext*)
apply (*induct n*)
apply (*simp-all add*: *o-def*)
done

lemma *list-exhaust-size-gt0*:
assumes *y*: ⋀*a list*. *y* = *a* # *list* ⟹ *P*
shows *0* < *length y* ⟹ *P*
apply (*cases y*, *simp*)
apply (*rule y*)
apply *fastforce*
done

lemma *list-exhaust-size-eq0*:

assumes *y*: *y* = [] ⟹ *P*
shows *length y* = *0* ⟹ *P*
apply (*cases y*)
apply (*rule y*, *simp*)
apply *simp*
done

lemma *size-Cons-lem-eq*:
y = *xa* # *list* ==> *size y* = *Suc k* ==> *size list* = *k*
by *auto*

lemmas *ls-splits* = *prod.split prod.split-asm if-split-asm*

lemma *not-B1-is-B0*: *y* ≠ (*1*::*bit*) ⟹ *y* = (*0*::*bit*)
by (*cases y*) *auto*

lemma *B1-ass-B0*:
assumes *y*: *y* = (*0*::*bit*) ⟹ *y* = (*1*::*bit*)
shows *y* = (*1*::*bit*)
apply (*rule classical*)
apply (*drule not-B1-is-B0*)
apply (*erule y*)
done

— simplifications for specific word lengths
lemmas *n2s-ths* [*THEN eq-reflection*] = *add-2-eq-Suc add-2-eq-Suc'*

lemmas *s2n-ths* = *n2s-ths* [*symmetric*]

lemma *and-len*: *xs* = *ys* ==> *xs* = *ys* & *length xs* = *length ys*
by *auto*

lemma *size-if*: *size* (*if p then xs else ys*) = (*if p then size xs else size ys*)
by *auto*

lemma *tl-if*: *tl* (*if p then xs else ys*) = (*if p then tl xs else tl ys*)
by *auto*

lemma *hd-if*: *hd* (*if p then xs else ys*) = (*if p then hd xs else hd ys*)
by *auto*

lemma *if-Not-x*: (*if p then* ~ *x else x*) = (*p* = (~ *x*))
by *auto*

lemma *if-x-Not*: (*if p then x else* ~ *x*) = (*p* = *x*)
by *auto*

lemma *if-same-and*: (*If p x y* & *If p u v*) = (*if p then x* & *u else y* & *v*)
by *auto*

lemma *if-same-eq*: (*If p x y* = (*If p u v*)) = (*if p then x* = (*u*) *else y* = (*v*))
by *auto*

lemma *if-same-eq-not*:
(*If p x y* = (~ *If p u v*)) = (*if p then x* = (~*u*) *else y* = (~*v*))
by *auto*

lemma *if-Cons*: (*if p then x # xs else y # ys*) = *If p x y # If p xs ys*
by *auto*

lemma *if-single*:
(*if xc then* [*xab*] *else* [*an*]) = [*if xc then xab else an*]
by *auto*

lemma *if-bool-simps*:
If p True y = (*p* | *y*) & *If p False y* = (~*p* & *y*) &
If p y True = (*p* −−> *y*) & *If p y False* = (*p* & *y*)
by *auto*

lemmas *if-simps* = *if-x-Not if-Not-x if-cancel if-True if-False if-bool-simps*

lemmas *seqr* = *eq-reflection* [**where** *x* = *size w*] **for** *w*

lemma *the-elemI*: *y* = {*x*} ==> *the-elem y* = *x*
by *simp*

lemma *nonemptyE*: *S* ~= {} ==> (!!*x*. *x* : *S* ==> *R*) ==> *R* **by** *auto*

lemma *gt-or-eq-0*: *0* < *y* ∨ *0* = (*y*::*nat*) **by** *arith*

lemmas *xtr1* = *xtrans*(*1*)
lemmas *xtr2* = *xtrans*(*2*)
lemmas *xtr3* = *xtrans*(*3*)
lemmas *xtr4* = *xtrans*(*4*)
lemmas *xtr5* = *xtrans*(*5*)
lemmas *xtr6* = *xtrans*(*6*)
lemmas *xtr7* = *xtrans*(*7*)
lemmas *xtr8* = *xtrans*(*8*)

lemmas *nat-simps* = *diff-add-inverse2 diff-add-inverse*
lemmas *nat-iffs* = *le-add1 le-add2*

lemma *sum-imp-diff*: *j* = *k* + *i* ==> *j* − *i* = (*k* :: *nat*) **by** *arith*

lemmas *pos-mod-sign2* = *zless2* [*THEN pos-mod-sign* [**where** *b* = *2*::*int*]]
lemmas *pos-mod-bound2* = *zless2* [*THEN pos-mod-bound* [**where** *b* = *2*::*int*]]

lemma *nmod2*: *n mod* (*2*::*int*) = *0* | *n mod 2* = *1*
by *arith*

lemmas *eme1p* = *emep1* [*simplified add.commute*]

lemma *le-diff-eq'*: (*a* ≤ *c* − *b*) = (*b* + *a* ≤ (*c*::*int*)) **by** *arith*

lemma *less-diff-eq'*: (*a* < *c* − *b*) = (*b* + *a* < (*c*::*int*)) **by** *arith*

lemma *diff-less-eq'*: (*a* − *b* < *c*) = (*a* < *b* + (*c*::*int*)) **by** *arith*

lemmas *m1mod22k* = *mult-pos-pos* [*OF zless2 zless2p*, *THEN zmod-minus1*]

lemma *z1pdiv2*:
(*2* ∗ *b* + *1*) *div 2* = (*b*::*int*) **by** *arith*

lemmas *zdiv-le-dividend* = *xtr3* [*OF div-by-1* [*symmetric*] *zdiv-mono2*,
simplified int-one-le-iff-zero-less, *simplified*]

lemma *axxbyy*:
a + *m* + *m* = *b* + *n* + *n* ==> (*a* = *0* | *a* = *1*) ==> (*b* = *0* | *b* = *1*) ==>
a = *b* & *m* = (*n* :: *int*) **by** *arith*

lemma *axxmod2*:
(*1* + *x* + *x*) *mod 2* = (*1* :: *int*) & (*0* + *x* + *x*) *mod 2* = (*0* :: *int*) **by** *arith*

lemma *axxdiv2*:
(*1* + *x* + *x*) *div 2* = (*x* :: *int*) & (*0* + *x* + *x*) *div 2* = (*x* :: *int*) **by** *arith*

lemmas *iszero-minus* = *trans* [*THEN trans*,
OF iszero-def neg-equal-0-iff-equal iszero-def [*symmetric*]]

lemmas *zadd-diff-inverse* = *trans* [*OF diff-add-cancel* [*symmetric*] *add.commute*]

lemmas *add-diff-cancel2* = *add.commute* [*THEN diff-eq-eq* [*THEN iffD2*]]

lemmas *rdmods* [*symmetric*] = *mod-minus-eq*
mod-diff-left-eq mod-diff-right-eq mod-add-left-eq
mod-add-right-eq mod-mult-right-eq mod-mult-left-eq

lemma *mod-plus-right*:
((*a* + *x*) *mod m* = (*b* + *x*) *mod m*) = (*a mod m* = *b mod* (*m* :: *nat*))
apply (*induct x*)
apply (*simp-all add*: *mod-Suc*)
apply *arith*
done

lemma *nat-minus-mod*: (*n* − *n mod m*) *mod m* = (*0* :: *nat*)
by (*induct n*) (*simp-all add* : *mod-Suc*)

lemmas *nat-minus-mod-plus-right* = *trans* [*OF nat-minus-mod mod-0* [*symmetric*], *THEN mod-plus-right* [*THEN iffD2*], *simplified*]

lemmas *push-mods'* = *mod-add-eq*
mod-mult-eq mod-diff-eq
mod-minus-eq

lemmas *push-mods* = *push-mods'* [*THEN eq-reflection*]
lemmas *pull-mods* = *push-mods* [*symmetric*] *rdmods* [*THEN eq-reflection*]
lemmas *mod-simps* =
mod-mult-self2-is-0 [*THEN eq-reflection*]
mod-mult-self1-is-0 [*THEN eq-reflection*]
mod-mod-trivial [*THEN eq-reflection*]

lemma *nat-mod-eq*:
!!*b*. *b* < *n* ==> *a mod n* = *b mod n* ==> *a mod n* = (*b* :: *nat*)
by (*induct a*) *auto*

lemmas *nat-mod-eq'* = *refl* [*THEN* [*2*] *nat-mod-eq*]

lemma *nat-mod-lem*:
(*0* :: *nat*) < *n* ==> *b* < *n* = (*b mod n* = *b*)
apply *safe*
apply (*erule nat-mod-eq'*)
apply (*erule subst*)
apply (*erule mod-less-divisor*)
done

lemma *mod-nat-add*:
(*x* :: *nat*) < *z* ==> *y* < *z* ==>
(*x* + *y*) *mod z* = (*if x* + *y* < *z then x* + *y else x* + *y* − *z*)
apply (*rule nat-mod-eq*)
apply *auto*
apply (*rule trans*)
apply (*rule le-mod-geq*)
apply *simp*
apply (*rule nat-mod-eq'*)
apply *arith*
done

lemma *mod-nat-sub*:
(*x* :: *nat*) < *z* ==> (*x* − *y*) *mod z* = *x* − *y*
by (*rule nat-mod-eq'*) *arith*

lemma *int-mod-eq*:
(*0* :: *int*) <= *b* ==> *b* < *n* ==> *a mod n* = *b mod n* ==> *a mod n* = *b*
by (*metis mod-pos-pos-trivial*)

lemmas *int-mod-eq′* = *mod-pos-pos-trivial*

lemma *int-mod-le*: (*0*::*int*) <= *a* ==> *a mod n* <= *a*
 by (*fact Divides.semiring-numeral-div-class.mod-less-eq-dividend*)

lemma *mod-add-if-z*:
 (*x* :: *int*) < *z* ==> *y* < *z* ==> *0* <= *y* ==> *0* <= *x* ==> *0* <= *z* ==>
 (*x* + *y*) *mod z* = (*if x* + *y* < *z then x* + *y else x* + *y* − *z*)
 by (*auto intro*: *int-mod-eq*)

lemma *mod-sub-if-z*:
 (*x* :: *int*) < *z* ==> *y* < *z* ==> *0* <= *y* ==> *0* <= *x* ==> *0* <= *z* ==>
 (*x* − *y*) *mod z* = (*if y* <= *x then x* − *y else x* − *y* + *z*)
 by (*auto intro*: *int-mod-eq*)

lemmas *zmde* = *zmod-zdiv-equality* [*THEN diff-eq-eq* [*THEN iffD2*], *symmetric*]
lemmas *mcl* = *mult-cancel-left* [*THEN iffD1*, *THEN make-pos-rule*]

lemma *zdiv-mult-self*: *m* ~= (*0* :: *int*) ==> (*a* + *m* ∗ *n*) *div m* = *a div m* + *n*
 apply (*rule mcl*)
  prefer *2*
  apply (*erule asm-rl*)
 apply (*simp add*: *zmde ring-distribs*)
 done

lemma *mod-power-lem*:
 a > *1* ==> *a* ˆ *n mod a* ˆ *m* = (*if m* <= *n then 0 else* (*a* :: *int*) ˆ *n*)
 apply *clarsimp*
 apply *safe*
  apply (*simp add*: *dvd-eq-mod-eq-0* [*symmetric*])
  apply (*drule le-iff-add* [*THEN iffD1*])
  apply (*force simp*: *power-add*)
 apply (*rule mod-pos-pos-trivial*)
  apply (*simp*)
 apply (*rule power-strict-increasing*)
  apply *auto*
 done

lemma *pl-pl-rels*:
 a + *b* = *c* + *d* ==>
  a >= *c* & *b* <= *d* | *a* <= *c* & *b* >= (*d* :: *nat*) **by** *arith*

lemmas *pl-pl-rels′* = *add.commute* [*THEN* [*2*] *trans*, *THEN pl-pl-rels*]

lemma *minus-eq*: (*m* − *k* = *m*) = (*k* = *0* | *m* = (*0* :: *nat*)) **by** *arith*

lemma *pl-pl-mm*: (*a* :: *nat*) + *b* = *c* + *d* ==> *a* − *c* = *d* − *b* **by** *arith*

lemmas *pl-pl-mm'* = *add.commute* [*THEN* [*2*] *trans*, *THEN pl-pl-mm*]

lemmas *dme* = *box-equals* [*OF div-mod-equality add-0-right add-0-right*]
lemmas *dtle* = *xtr3* [*OF dme* [*symmetric*] *le-add1*]
lemmas *th2* = *order-trans* [*OF order-refl* [*THEN* [*2*] *mult-le-mono*] *dtle*]

lemma *td-gal*:
  0 < *c* ==> (*a* >= *b* * *c*) = (*a div c* >= (*b* :: *nat*))
  apply *safe*
   apply (*erule* (*1*) *xtr4* [*OF div-le-mono div-mult-self-is-m*])
  apply (*erule th2*)
  done

lemmas *td-gal-lt* = *td-gal* [*simplified not-less* [*symmetric*], *simplified*]

lemma *div-mult-le*: (*a* :: *nat*) *div b* * *b* <= *a*
  by (*fact dtle*)

lemmas *sdl* = *split-div-lemma* [*THEN iffD1*, *symmetric*]

lemma *given-quot*: *f* > (*0* :: *nat*) ==> (*f* * *l* + (*f* − *1*)) *div f* = *l*
  by (*rule sdl*, *assumption*) (*simp* (*no-asm*))

lemma *given-quot-alt*: *f* > (*0* :: *nat*) ==> (*l* * *f* + *f* − *Suc 0*) *div f* = *l*
  apply (*frule given-quot*)
  apply (*rule trans*)
   prefer *2*
   apply (*erule asm-rl*)
  apply (*rule-tac f*=%*n*. *n div f* **in** *arg-cong*)
  apply (*simp add* : *ac-simps*)
  done

lemma *diff-mod-le*: (*a*::*nat*) < *d* ==> *b dvd d* ==> *a* − *a mod b* <= *d* − *b*
  apply (*unfold dvd-def*)
  apply *clarify*
  apply (*case-tac k*)
   apply *clarsimp*
  apply *clarify*
  apply (*cases b* > *0*)
   apply (*drule mult.commute* [*THEN xtr1*])
   apply (*frule* (*1*) *td-gal-lt* [*THEN iffD1*])
   apply (*clarsimp simp*: *le-simps*)
   apply (*rule mult-div-cancel* [*THEN* [*2*] *xtr4*])
   apply (*rule mult-mono*)
      apply *auto*
  done

lemma *less-le-mult'*:
  w * *c* < *b* * *c* ==> *0* ≤ *c* ==> (*w* + *1*) * *c* ≤ *b* * (*c*::*int*)

apply (*rule mult-right-mono*)
 apply (*rule zless-imp-add1-zle*)
 apply (*erule* (*1*) *mult-right-less-imp-less*)
apply *assumption*
done

lemma *less-le-mult*:
$w * c < b * c \Longrightarrow 0 \le c \Longrightarrow w * c + c \le b * (c{::}int)$
using *less-le-mult'* [*of w c b*] **by** (*simp add*: *algebra-simps*)

lemmas *less-le-mult-minus* = *iffD2* [*OF le-diff-eq less-le-mult*,
simplified left-diff-distrib]

lemma *gen-minus*: $0 < n ==> f\ n = f\ (Suc\ (n - 1))$
by *auto*

lemma *mpl-lem*: $j <= (i :: nat) ==> k < j ==> i - j + k < i$ **by** *arith*

lemma *nonneg-mod-div*:
$0 <= a ==> 0 <= b ==> 0 <= (a\ mod\ b :: int)\ \&\ 0 <= a\ div\ b$
apply (*cases b* = *0*, *clarsimp*)
apply (*auto intro*: *pos-imp-zdiv-nonneg-iff* [*THEN iffD2*])
done

declare *iszero-0* [*intro*]

lemma *min-pm* [*simp*]:
$min\ a\ b + (a - b) = (a :: nat)$
by *arith*

lemma *min-pm1* [*simp*]:
$a - b + min\ a\ b = (a :: nat)$
by *arith*

lemma *rev-min-pm* [*simp*]:
$min\ b\ a + (a - b) = (a :: nat)$
by *arith*

lemma *rev-min-pm1* [*simp*]:
$a - b + min\ b\ a = (a :: nat)$
by *arith*

lemma *min-minus* [*simp*]:
$min\ m\ (m - k) = (m - k :: nat)$
by *arith*

lemma *min-minus'* [*simp*]:
$min\ (m - k)\ m = (m - k :: nat)$
by *arith*

end

127 A type of finite bit strings

theory *Word*
imports
Type-Length
~~/src/HOL/Library/Boolean-Algebra
Bits-Bit
Bool-List-Representation
Misc-Typedef
Word-Miscellaneous
begin

See `Examples/WordExamples.thy` for examples.

127.1 Type definition

typedef (**overloaded**) *'a word* = {(*0*::*int*) ..< *2* ^ *len-of TYPE*(*'a*::*len0*)}
morphisms *uint Abs-word* **by** *auto*

lemma *uint-nonnegative*:
0 ≤ *uint w*
using *word.uint* [*of w*] **by** *simp*

lemma *uint-bounded*:
fixes *w* :: *'a*::*len0 word*
shows *uint w* < *2* ^ *len-of TYPE*(*'a*)
using *word.uint* [*of w*] **by** *simp*

lemma *uint-idem*:
fixes *w* :: *'a*::*len0 word*
shows *uint w mod 2* ^ *len-of TYPE*(*'a*) = *uint w*
using *uint-nonnegative uint-bounded* **by** (*rule mod-pos-pos-trivial*)

lemma *word-uint-eq-iff*:
a = *b* ⟷ *uint a* = *uint b*
by (*simp add*: *uint-inject*)

lemma *word-uint-eqI*:
uint a = *uint b* ⟹ *a* = *b*
by (*simp add*: *word-uint-eq-iff*)

definition *word-of-int* :: *int* ⇒ *'a*::*len0 word*
where
— representation of words using unsigned or signed bins, only difference in these is the type class
word-of-int k = *Abs-word* (*k mod 2* ^ *len-of TYPE*(*'a*))

lemma *uint-word-of-int*:
  uint (*word-of-int k* :: *'a*::*len0 word*) = *k mod 2 ^ len-of TYPE*(*'a*)
  by (*auto simp add*: *word-of-int-def intro*: *Abs-word-inverse*)

lemma *word-of-int-uint*:
  word-of-int (*uint w*) = *w*
  by (*simp add*: *word-of-int-def uint-idem uint-inverse*)

lemma *split-word-all*:
  ($\bigwedge$*x*::*'a*::*len0 word*. *PROP P x*) ≡ ($\bigwedge$*x*. *PROP P* (*word-of-int x*))
proof
  fix *x* :: *'a word*
  assume $\bigwedge$*x*. *PROP P* (*word-of-int x*)
  then have *PROP P* (*word-of-int* (*uint x*)) **.**
  then show *PROP P x* **by** (*simp add*: *word-of-int-uint*)
qed

127.2 Type conversions and casting

definition *sint* :: *'a*::*len word* ⇒ *int*
where
  — treats the most-significant-bit as a sign bit
  sint-uint: *sint w* = *sbintrunc* (*len-of TYPE* (*'a*) − *1*) (*uint w*)

definition *unat* :: *'a*::*len0 word* ⇒ *nat*
where
  unat w = *nat* (*uint w*)

definition *uints* :: *nat* ⇒ *int set*
where
  — the sets of integers representing the words
  uints n = *range* (*bintrunc n*)

definition *sints* :: *nat* ⇒ *int set*
where
  sints n = *range* (*sbintrunc* (*n* − *1*))

lemma *uints-num*:
  uints n = {*i*. *0* ≤ *i* ∧ *i* < *2* ^ *n*}
  by (*simp add*: *uints-def range-bintrunc*)

lemma *sints-num*:
  sints n = {*i*. − (*2* ^ (*n* − *1*)) ≤ *i* ∧ *i* < *2* ^ (*n* − *1*)}
  by (*simp add*: *sints-def range-sbintrunc*)

definition *unats* :: *nat* ⇒ *nat set*
where
  unats n = {*i*. *i* < *2* ^ *n*}

definition *norm-sint* :: *nat* ⇒ *int* ⇒ *int*
where
norm-sint n w = (*w* + *2* ^ (*n* − *1*)) *mod 2* ^ *n* − *2* ^ (*n* − *1*)

definition *scast* :: ′*a*::*len word* ⇒ ′*b*::*len word*
where
— cast a word to a different length
scast w = *word-of-int* (*sint w*)

definition *ucast* :: ′*a*::*len0 word* ⇒ ′*b*::*len0 word*
where
ucast w = *word-of-int* (*uint w*)

instantiation *word* :: (*len0*) *size*
begin

definition
word-size: *size* (*w* :: ′*a word*) = *len-of TYPE*(′*a*)

instance ..

end

lemma *word-size-gt-0* [*iff*]:
0 < *size* (*w*::′*a*::*len word*)
by (*simp add*: *word-size*)

lemmas *lens-gt-0* = *word-size-gt-0 len-gt-0*

lemma *lens-not-0* [*iff*]:
shows *size* (*w*::′*a*::*len word*) ≠ *0*
and *len-of TYPE*(′*a*::*len*) ≠ *0*
by *auto*

definition *source-size* :: (′*a*::*len0 word* ⇒ ′*b*) ⇒ *nat*
where
— whether a cast (or other) function is to a longer or shorter length
[*code del*]: *source-size c* = (*let arb* = *undefined*; *x* = *c arb in size arb*)

definition *target-size* :: (′*a* ⇒ ′*b*::*len0 word*) ⇒ *nat*
where
[*code del*]: *target-size c* = *size* (*c undefined*)

definition *is-up* :: (′*a*::*len0 word* ⇒ ′*b*::*len0 word*) ⇒ *bool*
where
is-up c ⟷ *source-size c* ≤ *target-size c*

definition *is-down* :: (′*a* :: *len0 word* ⇒ ′*b* :: *len0 word*) ⇒ *bool*

where
is-down c $\longleftrightarrow$ *target-size c* $\leq$ *source-size c*

definition *of-bl* :: *bool list* $\Rightarrow$ *'a::len0 word*
where
of-bl bl = *word-of-int* (*bl-to-bin bl*)

definition *to-bl* :: *'a::len0 word* $\Rightarrow$ *bool list*
where
to-bl w = *bin-to-bl* (*len-of TYPE* (*'a*)) (*uint w*)

definition *word-reverse* :: *'a::len0 word* $\Rightarrow$ *'a word*
where
word-reverse w = *of-bl* (*rev* (*to-bl w*))

definition *word-int-case* :: (*int* $\Rightarrow$ *'b*) $\Rightarrow$ *'a::len0 word* $\Rightarrow$ *'b*
where
word-int-case f w = *f* (*uint w*)

translations
case x of XCONST of-int y => b == CONST word-int-case (%*y. b*) *x*
case x of (*XCONST of-int* :: *'a*) *y => b => CONST word-int-case* (%*y. b*) *x*

127.3 Correspondence relation for theorem transfer

definition *cr-word* :: *int* $\Rightarrow$ *'a::len0 word* $\Rightarrow$ *bool*
where
cr-word = (λ*x y. word-of-int x* = *y*)

lemma *Quotient-word*:
Quotient (λ*x y. bintrunc* (*len-of TYPE*(*'a*)) *x* = *bintrunc* (*len-of TYPE*(*'a*)) *y*)
word-of-int uint (*cr-word* :: - $\Rightarrow$ *'a::len0 word* $\Rightarrow$ *bool*)
unfolding *Quotient-alt-def cr-word-def*
by (*simp add*: *no-bintr-alt1 word-of-int-uint*) (*simp add*: *word-of-int-def Abs-word-inject*)

lemma *reflp-word*:
reflp (λ*x y. bintrunc* (*len-of TYPE*(*'a::len0*)) *x* = *bintrunc* (*len-of TYPE*(*'a*))
y)
by (*simp add*: *reflp-def*)

setup-lifting *Quotient-word reflp-word*

TODO: The next lemma could be generated automatically.

lemma *uint-transfer* [*transfer-rule*]:
(*rel-fun pcr-word op* =) (*bintrunc* (*len-of TYPE*(*'a*)))
(*uint* :: *'a::len0 word* $\Rightarrow$ *int*)
unfolding *rel-fun-def word.pcr-cr-eq cr-word-def*
by (*simp add*: *no-bintr-alt1 uint-word-of-int*)

127.4 Basic code generation setup

definition *Word* :: *int* ⇒ *'a*::*len0 word*
where
[*code-post*]: *Word* = *word-of-int*

lemma [*code abstype*]:
Word (*uint w*) = *w*
by (*simp add*: *Word-def word-of-int-uint*)

declare *uint-word-of-int* [*code abstract*]

instantiation *word* :: (*len0*) *equal*
begin

definition *equal-word* :: *'a word* ⇒ *'a word* ⇒ *bool*
where
equal-word k l ⟷ *HOL.equal* (*uint k*) (*uint l*)

instance proof
qed (*simp add*: *equal equal-word-def word-uint-eq-iff*)

end

notation *fcomp* (**infixl** ∘> *60*)
notation *scomp* (**infixl** ∘→ *60*)

instantiation *word* :: ({*len0*, *typerep*}) *random*
begin

definition
random-word i = *Random.range i* ∘→ (λ*k*. *Pair* (
let j = *word-of-int* (*int-of-integer* (*integer-of-natural k*)) :: *'a word*
in (*j*, λ-::*unit*. *Code-Evaluation.term-of j*)))

instance ..

end

no-notation *fcomp* (**infixl** ∘> *60*)
no-notation *scomp* (**infixl** ∘→ *60*)

127.5 Type-definition locale instantiations

lemmas *uint-0* = *uint-nonnegative*
lemmas *uint-lt* = *uint-bounded*
lemmas *uint-mod-same* = *uint-idem*

lemma *td-ext-uint*:
td-ext (*uint* :: *'a word* ⇒ *int*) *word-of-int* (*uints* (*len-of TYPE*(*'a*::*len0*)))

(λw::*int*. *w mod 2 ^ len-of TYPE*(′*a*))
apply (*unfold td-ext-def* ′)
apply (*simp add*: *uints-num word-of-int-def bintrunc-mod2p*)
apply (*simp add*: *uint-mod-same uint-0 uint-lt*
word.uint-inverse word.Abs-word-inverse int-mod-lem)
done

interpretation *word-uint*:
td-ext uint::′*a*::*len0 word* ⇒ *int*
word-of-int
uints (*len-of TYPE*(′*a*::*len0*))
λ*w*. *w mod 2 ^ len-of TYPE*(′*a*::*len0*)
by (*fact td-ext-uint*)

lemmas *td-uint* = *word-uint.td-thm*
lemmas *int-word-uint* = *word-uint.eq-norm*

lemma *td-ext-ubin*:
td-ext (*uint* :: ′*a word* ⇒ *int*) *word-of-int* (*uints* (*len-of TYPE*(′*a*::*len0*)))
(*bintrunc* (*len-of TYPE*(′*a*)))
by (*unfold no-bintr-alt1*) (*fact td-ext-uint*)

interpretation *word-ubin*:
td-ext uint::′*a*::*len0 word* ⇒ *int*
word-of-int
uints (*len-of TYPE*(′*a*::*len0*))
bintrunc (*len-of TYPE*(′*a*::*len0*))
by (*fact td-ext-ubin*)

127.6 Arithmetic operations

lift-definition *word-succ* :: ′*a*::*len0 word* ⇒ ′*a word* **is** λ*x*. *x* + *1*
by (*metis bintr-ariths*(*6*))

lift-definition *word-pred* :: ′*a*::*len0 word* ⇒ ′*a word* **is** λ*x*. *x* − *1*
by (*metis bintr-ariths*(*7*))

instantiation *word* :: (*len0*) {*neg-numeral*, *Divides.div*, *comm-monoid-mult*, *comm-ring*}
begin

lift-definition *zero-word* :: ′*a word* **is** *0* **.**

lift-definition *one-word* :: ′*a word* **is** *1* **.**

lift-definition *plus-word* :: ′*a word* ⇒ ′*a word* ⇒ ′*a word* **is** *op* +
by (*metis bintr-ariths*(*2*))

lift-definition *minus-word* :: ′*a word* ⇒ ′*a word* ⇒ ′*a word* **is** *op* −
by (*metis bintr-ariths*(*3*))

lift-definition *uminus-word* :: *'a word* ⇒ *'a word* **is** *uminus*
by (*metis bintr-ariths*(*5*))

lift-definition *times-word* :: *'a word* ⇒ *'a word* ⇒ *'a word* **is** *op* ∗
by (*metis bintr-ariths*(*4*))

definition
word-div-def: *a div b* = *word-of-int* (*uint a div uint b*)

definition
word-mod-def: *a mod b* = *word-of-int* (*uint a mod uint b*)

instance
by *standard* (*transfer*, *simp add*: *algebra-simps*)+

end

Legacy theorems:

lemma *word-arith-wis* [*code*]: **shows**
word-add-def: *a* + *b* = *word-of-int* (*uint a* + *uint b*) **and**
word-sub-wi: *a* − *b* = *word-of-int* (*uint a* − *uint b*) **and**
word-mult-def: *a* ∗ *b* = *word-of-int* (*uint a* ∗ *uint b*) **and**
word-minus-def: − *a* = *word-of-int* (− *uint a*) **and**
word-succ-alt: *word-succ a* = *word-of-int* (*uint a* + *1*) **and**
word-pred-alt: *word-pred a* = *word-of-int* (*uint a* − *1*) **and**
word-0-wi: *0* = *word-of-int 0* **and**
word-1-wi: *1* = *word-of-int 1*
unfolding *plus-word-def minus-word-def times-word-def uminus-word-def*
unfolding *word-succ-def word-pred-def zero-word-def one-word-def*
by *simp-all*

lemmas *arths* =
bintr-ariths [*THEN word-ubin.norm-eq-iff* [*THEN iffD1*], *folded word-ubin.eq-norm*]

lemma *wi-homs*:
shows
wi-hom-add: *word-of-int a* + *word-of-int b* = *word-of-int* (*a* + *b*) **and**
wi-hom-sub: *word-of-int a* − *word-of-int b* = *word-of-int* (*a* − *b*) **and**
wi-hom-mult: *word-of-int a* ∗ *word-of-int b* = *word-of-int* (*a* ∗ *b*) **and**
wi-hom-neg: − *word-of-int a* = *word-of-int* (− *a*) **and**
wi-hom-succ: *word-succ* (*word-of-int a*) = *word-of-int* (*a* + *1*) **and**
wi-hom-pred: *word-pred* (*word-of-int a*) = *word-of-int* (*a* − *1*)
by (*transfer*, *simp*)+

lemmas *wi-hom-syms* = *wi-homs* [*symmetric*]

lemmas *word-of-int-homs* = *wi-homs word-0-wi word-1-wi*

lemmas *word-of-int-hom-syms* = *word-of-int-homs* [*symmetric*]

instance *word* :: (*len*) *comm-ring-1*
proof
  have *0* < *len-of TYPE*(*'a*) **by** (*rule len-gt-0*)
  then show (*0*::*'a word*) ≠ *1*
    by − (*transfer*, *auto simp add*: *gr0-conv-Suc*)
qed

lemma *word-of-nat*: *of-nat n* = *word-of-int* (*int n*)
  by (*induct n*) (*auto simp add* : *word-of-int-hom-syms*)

lemma *word-of-int*: *of-int* = *word-of-int*
  apply (*rule ext*)
  apply (*case-tac x rule*: *int-diff-cases*)
  apply (*simp add*: *word-of-nat wi-hom-sub*)
  done

definition *udvd* :: *'a*::*len word* => *'a*::*len word* => *bool* (**infixl** *udvd 50*)
where
  a udvd b = (*EX n*>=*0*. *uint b* = *n* ∗ *uint a*)

127.7 Ordering

instantiation *word* :: (*len0*) *linorder*
begin

definition
  word-le-def: *a* ≤ *b* ⟷ *uint a* ≤ *uint b*

definition
  word-less-def: *a* < *b* ⟷ *uint a* < *uint b*

instance
  by *standard* (*auto simp*: *word-less-def word-le-def*)

end

definition *word-sle* :: *'a* :: *len word* => *'a word* => *bool* ((-/ <=s -) [*50*, *51*] *50*)
where
  a <=s *b* = (*sint a* <= *sint b*)

definition *word-sless* :: *'a* :: *len word* => *'a word* => *bool* ((-/ <s -) [*50*, *51*] *50*)
where
  (*x* <s *y*) = (*x* <=s *y* & *x* ~= *y*)

127.8 Bit-wise operations

instantiation *word* :: (*len0*) *bits*
begin

lift-definition *bitNOT-word* :: *'a word* ⇒ *'a word* **is** *bitNOT*
by (*metis bin-trunc-not*)

lift-definition *bitAND-word* :: *'a word* ⇒ *'a word* ⇒ *'a word* **is** *bitAND*
by (*metis bin-trunc-and*)

lift-definition *bitOR-word* :: *'a word* ⇒ *'a word* ⇒ *'a word* **is** *bitOR*
by (*metis bin-trunc-or*)

lift-definition *bitXOR-word* :: *'a word* ⇒ *'a word* ⇒ *'a word* **is** *bitXOR*
by (*metis bin-trunc-xor*)

definition
word-test-bit-def: *test-bit a* = *bin-nth* (*uint a*)

definition
word-set-bit-def: *set-bit a n x* =
word-of-int (*bin-sc n x* (*uint a*))

definition
word-set-bits-def: (*BITS n. f n*) = *of-bl* (*bl-of-nth* (*len-of TYPE* (*'a*)) *f*)

definition
word-lsb-def: *lsb a* ⟷ *bin-last* (*uint a*)

definition *shiftl1* :: *'a word* ⇒ *'a word*
where
shiftl1 w = *word-of-int* (*uint w BIT False*)

definition *shiftr1* :: *'a word* ⇒ *'a word*
where
— shift right as unsigned or as signed, ie logical or arithmetic
shiftr1 w = *word-of-int* (*bin-rest* (*uint w*))

definition
shiftl-def: *w* << *n* = (*shiftl1* ^^ *n*) *w*

definition
shiftr-def: *w* >> *n* = (*shiftr1* ^^ *n*) *w*

instance ..

end

lemma [*code*]: **shows**
word-not-def: *NOT* (*a*::*'a*::*len0 word*) = *word-of-int* (*NOT* (*uint a*)) **and**
word-and-def: (*a*::*'a word*) *AND b* = *word-of-int* (*uint a AND uint b*) **and**
word-or-def: (*a*::*'a word*) *OR b* = *word-of-int* (*uint a OR uint b*) **and**

word-xor-def: $(a::'a\ word)\ XOR\ b = word\text{-}of\text{-}int\ (uint\ a\ XOR\ uint\ b)$
unfolding *bitNOT-word-def bitAND-word-def bitOR-word-def bitXOR-word-def*
by *simp-all*

instantiation *word* :: (*len*) *bitss*
begin

definition
word-msb-def:
msb a $\longleftrightarrow$ *bin-sign* (*sint a*) = −*1*

instance ..

end

definition *setBit* :: *'a* :: *len0 word* => *nat* => *'a word*
where
setBit w n = *set-bit w n True*

definition *clearBit* :: *'a* :: *len0 word* => *nat* => *'a word*
where
clearBit w n = *set-bit w n False*

127.9 Shift operations

definition *sshiftr1* :: *'a* :: *len word* => *'a word*
where
sshiftr1 w = *word-of-int* (*bin-rest* (*sint w*))

definition *bshiftr1* :: *bool* => *'a* :: *len word* => *'a word*
where
bshiftr1 b w = *of-bl* (*b* # *butlast* (*to-bl w*))

definition *sshiftr* :: *'a* :: *len word* => *nat* => *'a word* (**infixl** >>> *55*)
where
w >>> *n* = (*sshiftr1* ^^ *n*) *w*

definition *mask* :: *nat* => *'a*::*len word*
where
mask n = (*1* << *n*) − *1*

definition *revcast* :: *'a* :: *len0 word* => *'b* :: *len0 word*
where
revcast w = *of-bl* (*takefill False* (*len-of TYPE*(*'b*)) (*to-bl w*))

definition *slice1* :: *nat* => *'a* :: *len0 word* => *'b* :: *len0 word*
where
slice1 n w = *of-bl* (*takefill False n* (*to-bl w*))

definition *slice* :: *nat* => *'a* :: *len0 word* => *'b* :: *len0 word*
where
slice n w = *slice1* (*size w* − *n*) *w*

127.10 Rotation

definition *rotater1* :: *'a list* => *'a list*
where
rotater1 ys =
(*case ys of* [] => [] | *x* # *xs* => *last ys* # *butlast ys*)

definition *rotater* :: *nat* => *'a list* => *'a list*
where
rotater n = *rotater1* ^^ *n*

definition *word-rotr* :: *nat* => *'a* :: *len0 word* => *'a* :: *len0 word*
where
word-rotr n w = *of-bl* (*rotater n* (*to-bl w*))

definition *word-rotl* :: *nat* => *'a* :: *len0 word* => *'a* :: *len0 word*
where
word-rotl n w = *of-bl* (*rotate n* (*to-bl w*))

definition *word-roti* :: *int* => *'a* :: *len0 word* => *'a* :: *len0 word*
where
word-roti i w = (*if i* >= *0 then word-rotr* (*nat i*) *w*
else word-rotl (*nat* (− *i*)) *w*)

127.11 Split and cat operations

definition *word-cat* :: *'a* :: *len0 word* => *'b* :: *len0 word* => *'c* :: *len0 word*
where
word-cat a b = *word-of-int* (*bin-cat* (*uint a*) (*len-of TYPE* (*'b*)) (*uint b*))

definition *word-split* :: *'a* :: *len0 word* => (*'b* :: *len0 word*) ∗ (*'c* :: *len0 word*)
where
word-split a =
(*case bin-split* (*len-of TYPE* (*'c*)) (*uint a*) *of*
(*u*, *v*) => (*word-of-int u*, *word-of-int v*))

definition *word-rcat* :: *'a* :: *len0 word list* => *'b* :: *len0 word*
where
word-rcat ws =
word-of-int (*bin-rcat* (*len-of TYPE* (*'a*)) (*map uint ws*))

definition *word-rsplit* :: *'a* :: *len0 word* => *'b* :: *len word list*
where
word-rsplit w =
map word-of-int (*bin-rsplit* (*len-of TYPE* (*'b*)) (*len-of TYPE* (*'a*), *uint w*))

definition *max-word* :: *'a::len word* — Largest representable machine integer.
where
max-word = *word-of-int* (*2* ^ *len-of TYPE('a)* − *1*)

lemmas *of-nth-def* = *word-set-bits-def*

127.12 Theorems about typedefs

lemma *sint-sbintrunc'*:
sint (*word-of-int bin* :: *'a word*) =
(*sbintrunc* (*len-of TYPE* (*'a* :: *len*) − *1*) *bin*)
unfolding *sint-uint*
by (*auto simp*: *word-ubin.eq-norm sbintrunc-bintrunc-lt*)

lemma *uint-sint*:
uint w = *bintrunc* (*len-of TYPE*(*'a*)) (*sint* (*w* :: *'a* :: *len word*))
unfolding *sint-uint* **by** (*auto simp*: *bintrunc-sbintrunc-le*)

lemma *bintr-uint*:
fixes *w* :: *'a::len0 word*
shows *len-of TYPE*(*'a*) ≤ *n* ⟹ *bintrunc n* (*uint w*) = *uint w*
apply (*subst word-ubin.norm-Rep* [*symmetric*])
apply (*simp only*: *bintrunc-bintrunc-min word-size*)
apply (*simp add*: *min.absorb2*)
done

lemma *wi-bintr*:
len-of TYPE(*'a::len0*) ≤ *n* ⟹
word-of-int (*bintrunc n w*) = (*word-of-int w* :: *'a word*)
by (*clarsimp simp add*: *word-ubin.norm-eq-iff* [*symmetric*] *min.absorb1*)

lemma *td-ext-sbin*:
td-ext (*sint* :: *'a word* ⇒ *int*) *word-of-int* (*sints* (*len-of TYPE*(*'a::len*)))
(*sbintrunc* (*len-of TYPE*(*'a*) − *1*))
apply (*unfold td-ext-def' sint-uint*)
apply (*simp add* : *word-ubin.eq-norm*)
apply (*cases len-of TYPE*(*'a*))
apply (*auto simp add* : *sints-def*)
apply (*rule sym* [*THEN trans*])
apply (*rule word-ubin.Abs-norm*)
apply (*simp only*: *bintrunc-sbintrunc*)
apply (*drule sym*)
apply *simp*
done

lemma *td-ext-sint*:
td-ext (*sint* :: *'a word* ⇒ *int*) *word-of-int* (*sints* (*len-of TYPE*(*'a::len*)))
(λ*w*. (*w* + *2* ^ (*len-of TYPE*(*'a*) − *1*)) *mod 2* ^ *len-of TYPE*(*'a*) −
2 ^ (*len-of TYPE*(*'a*) − *1*))

using *td-ext-sbin* [**where** *?'a* = *'a*] **by** (*simp add*: *no-sbintr-alt2*)

interpretation *word-sint*:
td-ext sint ::*'a*::*len word* => *int*
word-of-int
sints (*len-of TYPE*(*'a*::*len*))
%w. (*w* + *2*ˆ(*len-of TYPE*(*'a*::*len*) − *1*)) *mod 2*ˆ*len-of TYPE*(*'a*::*len*) −
2 ˆ (*len-of TYPE*(*'a*::*len*) − *1*)
by (*rule td-ext-sint*)

interpretation *word-sbin*:
td-ext sint ::*'a*::*len word* => *int*
word-of-int
sints (*len-of TYPE*(*'a*::*len*))
sbintrunc (*len-of TYPE*(*'a*::*len*) − *1*)
by (*rule td-ext-sbin*)

lemmas *int-word-sint* = *td-ext-sint* [*THEN td-ext.eq-norm*]

lemmas *td-sint* = *word-sint.td*

lemma *to-bl-def'*:
(*to-bl* :: *'a* :: *len0 word* => *bool list*) =
bin-to-bl (*len-of TYPE*(*'a*)) *o uint*
by (*auto simp*: *to-bl-def*)

lemmas *word-reverse-no-def* [*simp*] = *word-reverse-def* [*of numeral w*] **for** *w*

lemma *uints-mod*: *uints n* = *range* (*λw*. *w mod 2* ˆ *n*)
by (*fact uints-def* [*unfolded no-bintr-alt1*])

lemma *word-numeral-alt*:
numeral b = *word-of-int* (*numeral b*)
by (*induct b*, *simp-all only*: *numeral.simps word-of-int-homs*)

declare *word-numeral-alt* [*symmetric*, *code-abbrev*]

lemma *word-neg-numeral-alt*:
− *numeral b* = *word-of-int* (− *numeral b*)
by (*simp only*: *word-numeral-alt wi-hom-neg*)

declare *word-neg-numeral-alt* [*symmetric*, *code-abbrev*]

lemma *word-numeral-transfer* [*transfer-rule*]:
(*rel-fun op* = *pcr-word*) *numeral numeral*
(*rel-fun op* = *pcr-word*) (− *numeral*) (− *numeral*)
apply (*simp-all add*: *rel-fun-def word.pcr-cr-eq cr-word-def*)
using *word-numeral-alt* [*symmetric*] *word-neg-numeral-alt* [*symmetric*] **by** *blast*+

lemma *uint-bintrunc* [*simp*]:
uint (*numeral bin* :: *'a word*) =
bintrunc (*len-of TYPE* (*'a* :: *len0*)) (*numeral bin*)
unfolding *word-numeral-alt* **by** (*rule word-ubin.eq-norm*)

lemma *uint-bintrunc-neg* [*simp*]: *uint* (− *numeral bin* :: *'a word*) =
bintrunc (*len-of TYPE* (*'a* :: *len0*)) (− *numeral bin*)
by (*simp only*: *word-neg-numeral-alt word-ubin.eq-norm*)

lemma *sint-sbintrunc* [*simp*]:
sint (*numeral bin* :: *'a word*) =
sbintrunc (*len-of TYPE* (*'a* :: *len*) − *1*) (*numeral bin*)
by (*simp only*: *word-numeral-alt word-sbin.eq-norm*)

lemma *sint-sbintrunc-neg* [*simp*]: *sint* (− *numeral bin* :: *'a word*) =
sbintrunc (*len-of TYPE* (*'a* :: *len*) − *1*) (− *numeral bin*)
by (*simp only*: *word-neg-numeral-alt word-sbin.eq-norm*)

lemma *unat-bintrunc* [*simp*]:
unat (*numeral bin* :: *'a* :: *len0 word*) =
nat (*bintrunc* (*len-of TYPE*(*'a*)) (*numeral bin*))
by (*simp only*: *unat-def uint-bintrunc*)

lemma *unat-bintrunc-neg* [*simp*]:
unat (− *numeral bin* :: *'a* :: *len0 word*) =
nat (*bintrunc* (*len-of TYPE*(*'a*)) (− *numeral bin*))
by (*simp only*: *unat-def uint-bintrunc-neg*)

lemma *size-0-eq*: *size* (*w* :: *'a* :: *len0 word*) = *0* ⟹ *v* = *w*
apply (*unfold word-size*)
apply (*rule word-uint.Rep-eqD*)
apply (*rule box-equals*)
defer
apply (*rule word-ubin.norm-Rep*)+
apply *simp*
done

lemma *uint-ge-0* [*iff*]: *0* ≤ *uint* (*x*::*'a*::*len0 word*)
using *word-uint.Rep* [*of x*] **by** (*simp add*: *uints-num*)

lemma *uint-lt2p* [*iff*]: *uint* (*x*::*'a*::*len0 word*) < *2* ^ *len-of TYPE*(*'a*)
using *word-uint.Rep* [*of x*] **by** (*simp add*: *uints-num*)

lemma *sint-ge*: − (*2* ^ (*len-of TYPE*(*'a*) − *1*)) ≤ *sint* (*x*::*'a*::*len word*)
using *word-sint.Rep* [*of x*] **by** (*simp add*: *sints-num*)

lemma *sint-lt*: *sint* (*x*::*'a*::*len word*) < *2* ^ (*len-of TYPE*(*'a*) − *1*)
using *word-sint.Rep* [*of x*] **by** (*simp add*: *sints-num*)

lemma *sign-uint-Pls* [*simp*]:
bin-sign (*uint x*) = *0*
by (*simp add*: *sign-Pls-ge-0*)

lemma *uint-m2p-neg*: *uint* (*x*::'*a*::*len0 word*) − *2* ^ *len-of TYPE*('*a*) < *0*
by (*simp only*: *diff-less-0-iff-less uint-lt2p*)

lemma *uint-m2p-not-non-neg*:
¬ *0* ≤ *uint* (*x*::'*a*::*len0 word*) − *2* ^ *len-of TYPE*('*a*)
by (*simp only*: *not-le uint-m2p-neg*)

lemma *lt2p-lem*:
len-of TYPE('*a*) ≤ *n* ⟹ *uint* (*w* :: '*a*::*len0 word*) < *2* ^ *n*
by (*metis bintr-uint bintrunc-mod2p int-mod-lem zless2p*)

lemma *uint-le-0-iff* [*simp*]: *uint x* ≤ *0* ⟷ *uint x* = *0*
by (*fact uint-ge-0* [*THEN leD*, *THEN linorder-antisym-conv1*])

lemma *uint-nat*: *uint w* = *int* (*unat w*)
unfolding *unat-def* **by** *auto*

lemma *uint-numeral*:
uint (*numeral b* :: '*a* :: *len0 word*) = *numeral b mod 2* ^ *len-of TYPE*('*a*)
unfolding *word-numeral-alt*
by (*simp only*: *int-word-uint*)

lemma *uint-neg-numeral*:
uint (− *numeral b* :: '*a* :: *len0 word*) = − *numeral b mod 2* ^ *len-of TYPE*('*a*)
unfolding *word-neg-numeral-alt*
by (*simp only*: *int-word-uint*)

lemma *unat-numeral*:
unat (*numeral b*::'*a*::*len0 word*) = *numeral b mod 2* ^ *len-of TYPE* ('*a*)
apply (*unfold unat-def*)
apply (*clarsimp simp only*: *uint-numeral*)
apply (*rule nat-mod-distrib* [*THEN trans*])
  apply (*rule zero-le-numeral*)
 apply (*simp-all add*: *nat-power-eq*)
done

lemma *sint-numeral*: *sint* (*numeral b* :: '*a* :: *len word*) = (*numeral b* +
  2 ^ (*len-of TYPE*('*a*) − *1*)) *mod 2* ^ *len-of TYPE*('*a*) −
  2 ^ (*len-of TYPE*('*a*) − *1*)
unfolding *word-numeral-alt* **by** (*rule int-word-sint*)

lemma *word-of-int-0* [*simp*, *code-post*]:
word-of-int 0 = *0*
unfolding *word-0-wi* **..**

lemma *word-of-int-1* [*simp*, *code-post*]:
word-of-int 1 = 1
unfolding *word-1-wi* **..**

lemma *word-of-int-neg-1* [*simp*]: *word-of-int* (− *1*) = − *1*
by (*simp add*: *wi-hom-syms*)

lemma *word-of-int-numeral* [*simp*] :
(*word-of-int* (*numeral bin*) :: ′*a* :: *len0 word*) = (*numeral bin*)
unfolding *word-numeral-alt* **..**

lemma *word-of-int-neg-numeral* [*simp*]:
(*word-of-int* (− *numeral bin*) :: ′*a* :: *len0 word*) = (− *numeral bin*)
unfolding *word-numeral-alt wi-hom-syms* **..**

lemma *word-int-case-wi*:
word-int-case f (*word-of-int i* :: ′*b word*) =
f (*i mod 2* ˆ *len-of TYPE*(′*b*::*len0*))
unfolding *word-int-case-def* **by** (*simp add*: *word-uint.eq-norm*)

lemma *word-int-split*:
P (*word-int-case f x*) =
(*ALL i*. *x* = (*word-of-int i* :: ′*b* :: *len0 word*) &
0 <= *i* & *i* < *2* ˆ *len-of TYPE*(′*b*) −−> *P* (*f i*))
unfolding *word-int-case-def*
by (*auto simp*: *word-uint.eq-norm mod-pos-pos-trivial*)

lemma *word-int-split-asm*:
P (*word-int-case f x*) =
(~ (*EX n*. *x* = (*word-of-int n* :: ′*b*::*len0 word*) &
0 <= *n* & *n* < *2* ˆ *len-of TYPE*(′*b*::*len0*) & ~ *P* (*f n*)))
unfolding *word-int-case-def*
by (*auto simp*: *word-uint.eq-norm mod-pos-pos-trivial*)

lemmas *uint-range*′ = *word-uint.Rep* [*unfolded uints-num mem-Collect-eq*]
lemmas *sint-range*′ = *word-sint.Rep* [*unfolded One-nat-def sints-num mem-Collect-eq*]

lemma *uint-range-size*: *0* <= *uint w* & *uint w* < *2* ˆ *size w*
unfolding *word-size* **by** (*rule uint-range*′)

lemma *sint-range-size*:
− (*2* ˆ (*size w* − *Suc 0*)) <= *sint w* & *sint w* < *2* ˆ (*size w* − *Suc 0*)
unfolding *word-size* **by** (*rule sint-range*′)

lemma *sint-above-size*: *2* ˆ (*size* (*w*::′*a*::*len word*) − *1*) ≤ *x* ⟹ *sint w* < *x*
unfolding *word-size* **by** (*rule less-le-trans* [*OF sint-lt*])

lemma *sint-below-size*:

$x \le - (2 \hat{}\ (size\ (w::'a::len\ word) - 1)) \Longrightarrow x \le sint\ w$
unfolding *word-size* **by** (*rule order-trans* [*OF - sint-ge*])

127.13 Testing bits

lemma *test-bit-eq-iff*: (*test-bit* (*u*::'*a*::*len0 word*) = *test-bit v*) = (*u* = *v*)
unfolding *word-test-bit-def* **by** (*simp add*: *bin-nth-eq-iff*)

lemma *test-bit-size* [*rule-format*] : (*w*::'*a*::*len0 word*) !! *n* −−> *n* < *size w*
apply (*unfold word-test-bit-def*)
apply (*subst word-ubin.norm-Rep* [*symmetric*])
apply (*simp only*: *nth-bintr word-size*)
apply *fast*
done

lemma *word-eq-iff*:
fixes *x y* :: '*a*::*len0 word*
shows *x* = *y* ⟷ (∀ *n*<*len-of TYPE*('*a*). *x* !! *n* = *y* !! *n*)
unfolding *uint-inject* [*symmetric*] *bin-eq-iff word-test-bit-def* [*symmetric*]
by (*metis test-bit-size* [*unfolded word-size*])

lemma *word-eqI* [*rule-format*]:
fixes *u* :: '*a*::*len0 word*
shows (*ALL n*. *n* < *size u* −−> *u* !! *n* = *v* !! *n*) ⟹ *u* = *v*
by (*simp add*: *word-size word-eq-iff*)

lemma *word-eqD*: (*u*::'*a*::*len0 word*) = *v* ⟹ *u* !! *x* = *v* !! *x*
by *simp*

lemma *test-bit-bin'*: *w* !! *n* = (*n* < *size w* & *bin-nth* (*uint w*) *n*)
unfolding *word-test-bit-def word-size*
by (*simp add*: *nth-bintr* [*symmetric*])

lemmas *test-bit-bin* = *test-bit-bin'* [*unfolded word-size*]

lemma *bin-nth-uint-imp*:
bin-nth (*uint* (*w*::'*a*::*len0 word*)) *n* ⟹ *n* < *len-of TYPE*('*a*)
apply (*rule nth-bintr* [*THEN iffD1*, *THEN conjunct1*])
apply (*subst word-ubin.norm-Rep*)
apply *assumption*
done

lemma *bin-nth-sint*:
fixes *w* :: '*a*::*len word*
shows *len-of TYPE*('*a*) ≤ *n* ⟹
bin-nth (*sint w*) *n* = *bin-nth* (*sint w*) (*len-of TYPE*('*a*) − *1*)
apply (*subst word-sbin.norm-Rep* [*symmetric*])
apply (*auto simp add*: *nth-sbintr*)
done

lemma *td-bl*:
type-definition (*to-bl* :: *'a*::*len0 word* => *bool list*)
of-bl
{*bl*. *length bl* = *len-of TYPE*(*'a*)}
apply (*unfold type-definition-def of-bl-def to-bl-def*)
apply (*simp add*: *word-ubin.eq-norm*)
apply *safe*
apply (*drule sym*)
apply *simp*
done

interpretation *word-bl*:
type-definition to-bl :: *'a*::*len0 word* => *bool list*
of-bl
{*bl*. *length bl* = *len-of TYPE*(*'a*::*len0*)}
by (*fact td-bl*)

lemmas *word-bl-Rep'* = *word-bl.Rep* [*unfolded mem-Collect-eq*, *iff*]

lemma *word-size-bl*: *size w* = *size* (*to-bl w*)
unfolding *word-size* **by** *auto*

lemma *to-bl-use-of-bl*:
(*to-bl w* = *bl*) = (*w* = *of-bl bl* ∧ *length bl* = *length* (*to-bl w*))
by (*fastforce elim*!: *word-bl.Abs-inverse* [*unfolded mem-Collect-eq*])

lemma *to-bl-word-rev*: *to-bl* (*word-reverse w*) = *rev* (*to-bl w*)
unfolding *word-reverse-def* **by** (*simp add*: *word-bl.Abs-inverse*)

lemma *word-rev-rev* [*simp*] : *word-reverse* (*word-reverse w*) = *w*
unfolding *word-reverse-def* **by** (*simp add* : *word-bl.Abs-inverse*)

lemma *word-rev-gal*: *word-reverse w* = *u* ⟹ *word-reverse u* = *w*
by (*metis word-rev-rev*)

lemma *word-rev-gal'*: *u* = *word-reverse w* ⟹ *w* = *word-reverse u*
by *simp*

lemma *length-bl-gt-0* [*iff*]: *0* < *length* (*to-bl* (*x*::*'a*::*len word*))
unfolding *word-bl-Rep'* **by** (*rule len-gt-0*)

lemma *bl-not-Nil* [*iff*]: *to-bl* (*x*::*'a*::*len word*) ≠ []
by (*fact length-bl-gt-0* [*unfolded length-greater-0-conv*])

lemma *length-bl-neq-0* [*iff*]: *length* (*to-bl* (*x*::*'a*::*len word*)) ≠ *0*
by (*fact length-bl-gt-0* [*THEN gr-implies-not0*])

lemma *hd-bl-sign-sint*: *hd* (*to-bl w*) = (*bin-sign* (*sint w*) = −*1*)
apply (*unfold to-bl-def sint-uint*)
apply (*rule trans* [*OF - bl-sbin-sign*])
apply *simp*
done

lemma *of-bl-drop′*:
lend = *length bl* − *len-of TYPE* (′*a* :: *len0*) ⟹
of-bl (*drop lend bl*) = (*of-bl bl* :: ′*a word*)
apply (*unfold of-bl-def*)
apply (*clarsimp simp add* : *trunc-bl2bin* [*symmetric*])
done

lemma *test-bit-of-bl*:
(*of-bl bl*::′*a*::*len0 word*) !! *n* = (*rev bl* ! *n* ∧ *n* < *len-of TYPE*(′*a*) ∧ *n* < *length bl*)
apply (*unfold of-bl-def word-test-bit-def*)
apply (*auto simp add*: *word-size word-ubin.eq-norm nth-bintr bin-nth-of-bl*)
done

lemma *no-of-bl*:
(*numeral bin* ::′*a*::*len0 word*) = *of-bl* (*bin-to-bl* (*len-of TYPE* (′*a*)) (*numeral bin*))
unfolding *of-bl-def* **by** *simp*

lemma *uint-bl*: *to-bl w* = *bin-to-bl* (*size w*) (*uint w*)
unfolding *word-size to-bl-def* **by** *auto*

lemma *to-bl-bin*: *bl-to-bin* (*to-bl w*) = *uint w*
unfolding *uint-bl* **by** (*simp add* : *word-size*)

lemma *to-bl-of-bin*:
to-bl (*word-of-int bin*::′*a*::*len0 word*) = *bin-to-bl* (*len-of TYPE*(′*a*)) *bin*
unfolding *uint-bl* **by** (*clarsimp simp add*: *word-ubin.eq-norm word-size*)

lemma *to-bl-numeral* [*simp*]:
to-bl (*numeral bin*::′*a*::*len0 word*) =
bin-to-bl (*len-of TYPE*(′*a*)) (*numeral bin*)
unfolding *word-numeral-alt* **by** (*rule to-bl-of-bin*)

lemma *to-bl-neg-numeral* [*simp*]:
to-bl (− *numeral bin*::′*a*::*len0 word*) =
bin-to-bl (*len-of TYPE*(′*a*)) (− *numeral bin*)
unfolding *word-neg-numeral-alt* **by** (*rule to-bl-of-bin*)

lemma *to-bl-to-bin* [*simp*] : *bl-to-bin* (*to-bl w*) = *uint w*
unfolding *uint-bl* **by** (*simp add* : *word-size*)

lemma *uint-bl-bin*:
fixes *x* :: ′*a*::*len0 word*

shows *bl-to-bin* (*bin-to-bl* (*len-of TYPE*('*a*)) (*uint x*)) = *uint x*
by (*rule trans* [*OF bin-bl-bin word-ubin.norm-Rep*])

lemma *uints-unats*: *uints n* = *int* ' *unats n*
apply (*unfold unats-def uints-num*)
apply *safe*
apply (*rule-tac image-eqI*)
apply (*erule-tac nat-0-le* [*symmetric*])
apply *auto*
apply (*erule-tac nat-less-iff* [*THEN iffD2*])
apply (*rule-tac* [*2*] *zless-nat-eq-int-zless* [*THEN iffD1*])
apply (*auto simp add* : *nat-power-eq of-nat-power*)
done

lemma *unats-uints*: *unats n* = *nat* ' *uints n*
by (*auto simp add* : *uints-unats image-iff*)

lemmas *bintr-num* = *word-ubin.norm-eq-iff*
[*of numeral a numeral b*, *symmetric*, *folded word-numeral-alt*] **for** *a b*
lemmas *sbintr-num* = *word-sbin.norm-eq-iff*
[*of numeral a numeral b*, *symmetric*, *folded word-numeral-alt*] **for** *a b*

lemma *num-of-bintr'*:
bintrunc (*len-of TYPE*('*a* :: *len0*)) (*numeral a*) = (*numeral b*) ⟹
numeral a = (*numeral b* :: '*a word*)
unfolding *bintr-num* **by** (*erule subst*, *simp*)

lemma *num-of-sbintr'*:
sbintrunc (*len-of TYPE*('*a* :: *len*) − *1*) (*numeral a*) = (*numeral b*) ⟹
numeral a = (*numeral b* :: '*a word*)
unfolding *sbintr-num* **by** (*erule subst*, *simp*)

lemma *num-abs-bintr*:
(*numeral x* :: '*a word*) =
word-of-int (*bintrunc* (*len-of TYPE*('*a*::*len0*)) (*numeral x*))
by (*simp only*: *word-ubin.Abs-norm word-numeral-alt*)

lemma *num-abs-sbintr*:
(*numeral x* :: '*a word*) =
word-of-int (*sbintrunc* (*len-of TYPE*('*a*::*len*) − *1*) (*numeral x*))
by (*simp only*: *word-sbin.Abs-norm word-numeral-alt*)

lemma *ucast-id*: *ucast w* = *w*
unfolding *ucast-def* **by** *auto*

lemma *scast-id*: *scast w* = *w*

unfolding *scast-def* **by** *auto*

lemma *ucast-bl*: *ucast w* = *of-bl* (*to-bl w*)
unfolding *ucast-def of-bl-def uint-bl*
by (*auto simp add* : *word-size*)

lemma *nth-ucast*:
(*ucast w*::'*a*::*len0 word*) !! *n* = (*w* !! *n* & *n* < *len-of TYPE*('*a*))
apply (*unfold ucast-def test-bit-bin*)
apply (*simp add*: *word-ubin.eq-norm nth-bintr word-size*)
apply (*fast elim*!: *bin-nth-uint-imp*)
done

lemma *ucast-bintr* [*simp*]:
ucast (*numeral w* ::'*a*::*len0 word*) =
word-of-int (*bintrunc* (*len-of TYPE*('*a*)) (*numeral w*))
unfolding *ucast-def* **by** *simp*

lemma *scast-sbintr* [*simp*]:
scast (*numeral w* ::'*a*::*len word*) =
word-of-int (*sbintrunc* (*len-of TYPE*('*a*) − *Suc 0*) (*numeral w*))
unfolding *scast-def* **by** *simp*

lemma *source-size*: *source-size* (*c*::'*a*::*len0 word* ⇒ -) = *len-of TYPE*('*a*)
unfolding *source-size-def word-size Let-def* **..**

lemma *target-size*: *target-size* (*c*::- ⇒ '*b*::*len0 word*) = *len-of TYPE*('*b*)
unfolding *target-size-def word-size Let-def* **..**

lemma *is-down*:
fixes *c* :: '*a*::*len0 word* ⇒ '*b*::*len0 word*
shows *is-down c* ⟷ *len-of TYPE*('*b*) ≤ *len-of TYPE*('*a*)
unfolding *is-down-def source-size target-size* **..**

lemma *is-up*:
fixes *c* :: '*a*::*len0 word* ⇒ '*b*::*len0 word*
shows *is-up c* ⟷ *len-of TYPE*('*a*) ≤ *len-of TYPE*('*b*)
unfolding *is-up-def source-size target-size* **..**

lemmas *is-up-down* = *trans* [*OF is-up is-down* [*symmetric*]]

lemma *down-cast-same* [*OF refl*]: *uc* = *ucast* ⟹ *is-down uc* ⟹ *uc* = *scast*
apply (*unfold is-down*)
apply *safe*
apply (*rule ext*)
apply (*unfold ucast-def scast-def uint-sint*)

apply (*rule word-ubin.norm-eq-iff* [*THEN iffD1*])
apply *simp*
done

lemma *word-rev-tf*:
to-bl (*of-bl bl*::′*a*::*len0 word*) =
rev (*takefill False* (*len-of TYPE*(′*a*)) (*rev bl*))
unfolding *of-bl-def uint-bl*
by (*clarsimp simp add*: *bl-bin-bl-rtf word-ubin.eq-norm word-size*)

lemma *word-rep-drop*:
to-bl (*of-bl bl*::′*a*::*len0 word*) =
replicate (*len-of TYPE*(′*a*) − *length bl*) *False* @
drop (*length bl* − *len-of TYPE*(′*a*)) *bl*
by (*simp add*: *word-rev-tf takefill-alt rev-take*)

lemma *to-bl-ucast*:
to-bl (*ucast* (*w*::′*b*::*len0 word*) ::′*a*::*len0 word*) =
replicate (*len-of TYPE*(′*a*) − *len-of TYPE*(′*b*)) *False* @
drop (*len-of TYPE*(′*b*) − *len-of TYPE*(′*a*)) (*to-bl w*)
apply (*unfold ucast-bl*)
apply (*rule trans*)
apply (*rule word-rep-drop*)
apply *simp*
done

lemma *ucast-up-app* [*OF refl*]:
uc = *ucast* ⟹ *source-size uc* + *n* = *target-size uc* ⟹
to-bl (*uc w*) = *replicate n False* @ (*to-bl w*)
by (*auto simp add* : *source-size target-size to-bl-ucast*)

lemma *ucast-down-drop* [*OF refl*]:
uc = *ucast* ⟹ *source-size uc* = *target-size uc* + *n* ⟹
to-bl (*uc w*) = *drop n* (*to-bl w*)
by (*auto simp add* : *source-size target-size to-bl-ucast*)

lemma *scast-down-drop* [*OF refl*]:
sc = *scast* ⟹ *source-size sc* = *target-size sc* + *n* ⟹
to-bl (*sc w*) = *drop n* (*to-bl w*)
apply (*subgoal-tac sc* = *ucast*)
apply *safe*
apply *simp*
apply (*erule ucast-down-drop*)
apply (*rule down-cast-same* [*symmetric*])
apply (*simp add* : *source-size target-size is-down*)
done

lemma *sint-up-scast* [*OF refl*]:
sc = *scast* ⟹ *is-up sc* ⟹ *sint* (*sc w*) = *sint w*

apply (*unfold is-up*)
apply *safe*
apply (*simp add*: *scast-def word-sbin.eq-norm*)
apply (*rule box-equals*)
prefer *3*
apply (*rule word-sbin.norm-Rep*)
apply (*rule sbintrunc-sbintrunc-l*)
defer
apply (*subst word-sbin.norm-Rep*)
apply (*rule refl*)
apply *simp*
done

lemma *uint-up-ucast* [*OF refl*]:
uc = *ucast* $\Longrightarrow$ *is-up uc* $\Longrightarrow$ *uint* (*uc w*) = *uint w*
apply (*unfold is-up*)
apply *safe*
apply (*rule bin-eqI*)
apply (*fold word-test-bit-def*)
apply (*auto simp add*: *nth-ucast*)
apply (*auto simp add*: *test-bit-bin*)
done

lemma *ucast-up-ucast* [*OF refl*]:
uc = *ucast* $\Longrightarrow$ *is-up uc* $\Longrightarrow$ *ucast* (*uc w*) = *ucast w*
apply (*simp* (*no-asm*) *add*: *ucast-def*)
apply (*clarsimp simp add*: *uint-up-ucast*)
done

lemma *scast-up-scast* [*OF refl*]:
sc = *scast* $\Longrightarrow$ *is-up sc* $\Longrightarrow$ *scast* (*sc w*) = *scast w*
apply (*simp* (*no-asm*) *add*: *scast-def*)
apply (*clarsimp simp add*: *sint-up-scast*)
done

lemma *ucast-of-bl-up* [*OF refl*]:
w = *of-bl bl* $\Longrightarrow$ *size bl* <= *size w* $\Longrightarrow$ *ucast w* = *of-bl bl*
by (*auto simp add* : *nth-ucast word-size test-bit-of-bl intro*!: *word-eqI*)

lemmas *ucast-up-ucast-id* = *trans* [*OF ucast-up-ucast ucast-id*]
lemmas *scast-up-scast-id* = *trans* [*OF scast-up-scast scast-id*]

lemmas *isduu* = *is-up-down* [**where** *c* = *ucast*, *THEN iffD2*]
lemmas *isdus* = *is-up-down* [**where** *c* = *scast*, *THEN iffD2*]
lemmas *ucast-down-ucast-id* = *isduu* [*THEN ucast-up-ucast-id*]
lemmas *scast-down-scast-id* = *isdus* [*THEN ucast-up-ucast-id*]

lemma *up-ucast-surj*:
is-up (*ucast* :: *'b*::*len0 word* => *'a*::*len0 word*) $\Longrightarrow$

surj (*ucast* :: *'a word* => *'b word*)
by (*rule surjI*, *erule ucast-up-ucast-id*)

lemma *up-scast-surj*:
is-up (*scast* :: *'b*::*len word* => *'a*::*len word*) ⟹
surj (*scast* :: *'a word* => *'b word*)
by (*rule surjI*, *erule scast-up-scast-id*)

lemma *down-scast-inj*:
is-down (*scast* :: *'b*::*len word* => *'a*::*len word*) ⟹
inj-on (*ucast* :: *'a word* => *'b word*) *A*
by (*rule inj-on-inverseI*, *erule scast-down-scast-id*)

lemma *down-ucast-inj*:
is-down (*ucast* :: *'b*::*len0 word* => *'a*::*len0 word*) ⟹
inj-on (*ucast* :: *'a word* => *'b word*) *A*
by (*rule inj-on-inverseI*, *erule ucast-down-ucast-id*)

lemma *of-bl-append-same*: *of-bl* (*X* @ *to-bl w*) = *w*
by (*rule word-bl.Rep-eqD*) (*simp add*: *word-rep-drop*)

lemma *ucast-down-wi* [*OF refl*]:
uc = *ucast* ⟹ *is-down uc* ⟹ *uc* (*word-of-int x*) = *word-of-int x*
apply (*unfold is-down*)
apply (*clarsimp simp add*: *ucast-def word-ubin.eq-norm*)
apply (*rule word-ubin.norm-eq-iff* [*THEN iffD1*])
apply (*erule bintrunc-bintrunc-ge*)
done

lemma *ucast-down-no* [*OF refl*]:
uc = *ucast* ⟹ *is-down uc* ⟹ *uc* (*numeral bin*) = *numeral bin*
unfolding *word-numeral-alt* **by** *clarify* (*rule ucast-down-wi*)

lemma *ucast-down-bl* [*OF refl*]:
uc = *ucast* ⟹ *is-down uc* ⟹ *uc* (*of-bl bl*) = *of-bl bl*
unfolding *of-bl-def* **by** *clarify* (*erule ucast-down-wi*)

lemmas *slice-def'* = *slice-def* [*unfolded word-size*]
lemmas *test-bit-def'* = *word-test-bit-def* [*THEN fun-cong*]

lemmas *word-log-defs* = *word-and-def word-or-def word-xor-def word-not-def*

127.14 Word Arithmetic

lemma *word-less-alt*: (*a* < *b*) = (*uint a* < *uint b*)
by (*fact word-less-def*)

lemma *signed-linorder*: *class.linorder word-sle word-sless*
by *standard* (*unfold word-sle-def word-sless-def*, *auto*)

interpretation *signed*: *linorder word-sle word-sless*
by (*rule signed-linorder*)

lemma *udvdI*:
$0 \leq n \Longrightarrow uint\ b = n * uint\ a \Longrightarrow a\ udvd\ b$
by (*auto simp*: *udvd-def*)

lemmas *word-div-no* [*simp*] = *word-div-def* [*of numeral a numeral b*] **for** *a b*

lemmas *word-mod-no* [*simp*] = *word-mod-def* [*of numeral a numeral b*] **for** *a b*

lemmas *word-less-no* [*simp*] = *word-less-def* [*of numeral a numeral b*] **for** *a b*

lemmas *word-le-no* [*simp*] = *word-le-def* [*of numeral a numeral b*] **for** *a b*

lemmas *word-sless-no* [*simp*] = *word-sless-def* [*of numeral a numeral b*] **for** *a b*

lemmas *word-sle-no* [*simp*] = *word-sle-def* [*of numeral a numeral b*] **for** *a b*

lemma *word-m1-wi*: $- 1 = word\text{-}of\text{-}int\ (- 1)$
using *word-neg-numeral-alt* [*of Num.One*] **by** *simp*

lemma *word-0-bl* [*simp*]: *of-bl* [] = *0*
unfolding *of-bl-def* **by** *simp*

lemma *word-1-bl*: *of-bl* [*True*] = *1*
unfolding *of-bl-def* **by** (*simp add*: *bl-to-bin-def*)

lemma *uint-eq-0* [*simp*]: *uint 0* = *0*
unfolding *word-0-wi word-ubin.eq-norm* **by** *simp*

lemma *of-bl-0* [*simp*]: *of-bl* (*replicate n False*) = *0*
by (*simp add*: *of-bl-def bl-to-bin-rep-False*)

lemma *to-bl-0* [*simp*]:
to-bl (*0*::*'a*::*len0 word*) = *replicate* (*len-of TYPE*(*'a*)) *False*
unfolding *uint-bl*
by (*simp add*: *word-size bin-to-bl-zero*)

lemma *uint-0-iff*:
$uint\ x = 0 \longleftrightarrow x = 0$
by (*simp add*: *word-uint-eq-iff*)

lemma *unat-0-iff*:
$unat\ x = 0 \longleftrightarrow x = 0$
unfolding *unat-def* **by** (*auto simp add* : *nat-eq-iff uint-0-iff*)

lemma *unat-0* [*simp*]:

unat 0 = 0
unfolding *unat-def* **by** *auto*

lemma *size-0-same′*:
size w = 0 ⟹ w = (v :: ′a :: len0 word)
apply (*unfold word-size*)
apply (*rule box-equals*)
defer
apply (*rule word-uint.Rep-inverse*)+
apply (*rule word-ubin.norm-eq-iff* [*THEN iffD1*])
apply *simp*
done

lemmas *size-0-same = size-0-same′* [*unfolded word-size*]

lemmas *unat-eq-0 = unat-0-iff*
lemmas *unat-eq-zero = unat-0-iff*

lemma *unat-gt-0*: (*0 < unat x*) = (*x ~= 0*)
by (*auto simp*: *unat-0-iff* [*symmetric*])

lemma *ucast-0* [*simp*]: *ucast 0 = 0*
unfolding *ucast-def* **by** *simp*

lemma *sint-0* [*simp*]: *sint 0 = 0*
unfolding *sint-uint* **by** *simp*

lemma *scast-0* [*simp*]: *scast 0 = 0*
unfolding *scast-def* **by** *simp*

lemma *sint-n1* [*simp*] : *sint* (*− 1*) = *− 1*
unfolding *word-m1-wi word-sbin.eq-norm* **by** *simp*

lemma *scast-n1* [*simp*]: *scast* (*− 1*) = *− 1*
unfolding *scast-def* **by** *simp*

lemma *uint-1* [*simp*]: *uint* (*1::′a::len word*) = *1*
by (*simp only*: *word-1-wi word-ubin.eq-norm*) (*simp add*: *bintrunc-minus-simps*(*4*))

lemma *unat-1* [*simp*]: *unat* (*1::′a::len word*) = *1*
unfolding *unat-def* **by** *simp*

lemma *ucast-1* [*simp*]: *ucast* (*1::′a::len word*) = *1*
unfolding *ucast-def* **by** *simp*

127.15 Transferring goals from words to ints

lemma *word-ths*:
shows

word-succ-p1: *word-succ a* = *a* + *1* **and**
word-pred-m1: *word-pred a* = *a* − *1* **and**
word-pred-succ: *word-pred* (*word-succ a*) = *a* **and**
word-succ-pred: *word-succ* (*word-pred a*) = *a* **and**
word-mult-succ: *word-succ a* ∗ *b* = *b* + *a* ∗ *b*
by (*transfer*, *simp add*: *algebra-simps*)+

lemma *uint-cong*: *x* = *y* ⟹ *uint x* = *uint y*
by *simp*

lemma *uint-word-ariths*:
fixes *a b* :: *'a::len0 word*
shows *uint* (*a* + *b*) = (*uint a* + *uint b*) *mod 2* ^ *len-of TYPE*(*'a::len0*)
and *uint* (*a* − *b*) = (*uint a* − *uint b*) *mod 2* ^ *len-of TYPE*(*'a*)
and *uint* (*a* ∗ *b*) = *uint a* ∗ *uint b mod 2* ^ *len-of TYPE*(*'a*)
and *uint* (− *a*) = − *uint a mod 2* ^ *len-of TYPE*(*'a*)
and *uint* (*word-succ a*) = (*uint a* + *1*) *mod 2* ^ *len-of TYPE*(*'a*)
and *uint* (*word-pred a*) = (*uint a* − *1*) *mod 2* ^ *len-of TYPE*(*'a*)
and *uint* (*0* :: *'a word*) = *0 mod 2* ^ *len-of TYPE*(*'a*)
and *uint* (*1* :: *'a word*) = *1 mod 2* ^ *len-of TYPE*(*'a*)
by (*simp-all add*: *word-arith-wis* [*THEN trans* [*OF uint-cong int-word-uint*]])

lemma *uint-word-arith-bintrs*:
fixes *a b* :: *'a::len0 word*
shows *uint* (*a* + *b*) = *bintrunc* (*len-of TYPE*(*'a*)) (*uint a* + *uint b*)
and *uint* (*a* − *b*) = *bintrunc* (*len-of TYPE*(*'a*)) (*uint a* − *uint b*)
and *uint* (*a* ∗ *b*) = *bintrunc* (*len-of TYPE*(*'a*)) (*uint a* ∗ *uint b*)
and *uint* (− *a*) = *bintrunc* (*len-of TYPE*(*'a*)) (− *uint a*)
and *uint* (*word-succ a*) = *bintrunc* (*len-of TYPE*(*'a*)) (*uint a* + *1*)
and *uint* (*word-pred a*) = *bintrunc* (*len-of TYPE*(*'a*)) (*uint a* − *1*)
and *uint* (*0* :: *'a word*) = *bintrunc* (*len-of TYPE*(*'a*)) *0*
and *uint* (*1* :: *'a word*) = *bintrunc* (*len-of TYPE*(*'a*)) *1*
by (*simp-all add*: *uint-word-ariths bintrunc-mod2p*)

lemma *sint-word-ariths*:
fixes *a b* :: *'a::len word*
shows *sint* (*a* + *b*) = *sbintrunc* (*len-of TYPE*(*'a*) − *1*) (*sint a* + *sint b*)
and *sint* (*a* − *b*) = *sbintrunc* (*len-of TYPE*(*'a*) − *1*) (*sint a* − *sint b*)
and *sint* (*a* ∗ *b*) = *sbintrunc* (*len-of TYPE*(*'a*) − *1*) (*sint a* ∗ *sint b*)
and *sint* (− *a*) = *sbintrunc* (*len-of TYPE*(*'a*) − *1*) (− *sint a*)
and *sint* (*word-succ a*) = *sbintrunc* (*len-of TYPE*(*'a*) − *1*) (*sint a* + *1*)
and *sint* (*word-pred a*) = *sbintrunc* (*len-of TYPE*(*'a*) − *1*) (*sint a* − *1*)
and *sint* (*0* :: *'a word*) = *sbintrunc* (*len-of TYPE*(*'a*) − *1*) *0*
and *sint* (*1* :: *'a word*) = *sbintrunc* (*len-of TYPE*(*'a*) − *1*) *1*
by (*simp-all add*: *uint-word-arith-bintrs*
[*THEN uint-sint* [*symmetric*, *THEN trans*],
unfolded uint-sint bintr-arith1s bintr-ariths
len-gt-0 [*THEN bin-sbin-eq-iff'*] *word-sbin.norm-Rep*])

lemmas *uint-div-alt* = *word-div-def* [*THEN trans* [*OF uint-cong int-word-uint*]]
lemmas *uint-mod-alt* = *word-mod-def* [*THEN trans* [*OF uint-cong int-word-uint*]]

lemma *word-pred-0-n1*: *word-pred 0* = *word-of-int* (− *1*)
unfolding *word-pred-m1* **by** *simp*

lemma *succ-pred-no* [*simp*]:
word-succ (*numeral w*) = *numeral w* + *1*
word-pred (*numeral w*) = *numeral w* − *1*
word-succ (− *numeral w*) = − *numeral w* + *1*
word-pred (− *numeral w*) = − *numeral w* − *1*
unfolding *word-succ-p1 word-pred-m1* **by** *simp-all*

lemma *word-sp-01* [*simp*] :
word-succ (− *1*) = *0* & *word-succ 0* = *1* & *word-pred 0* = − *1* & *word-pred 1* = *0*
unfolding *word-succ-p1 word-pred-m1* **by** *simp-all*

lemma *word-of-int-Ex*:
∃ *y*. *x* = *word-of-int y*
by (*rule-tac x*=*uint x* **in** *exI*) *simp*

127.16 Order on fixed-length words

lemma *word-zero-le* [*simp*] :
0 <= (*y* :: ′*a* :: *len0 word*)
unfolding *word-le-def* **by** *auto*

lemma *word-m1-ge* [*simp*] : *word-pred 0* >= *y*
unfolding *word-le-def*
by (*simp only* : *word-pred-0-n1 word-uint.eq-norm m1mod2k*) *auto*

lemma *word-n1-ge* [*simp*]: *y* ≤ (−*1*::′*a*::*len0 word*)
unfolding *word-le-def*
by (*simp only*: *word-m1-wi word-uint.eq-norm m1mod2k*) *auto*

lemmas *word-not-simps* [*simp*] =
word-zero-le [*THEN leD*] *word-m1-ge* [*THEN leD*] *word-n1-ge* [*THEN leD*]

lemma *word-gt-0*: *0* < *y* ⟷ *0* ≠ (*y* :: ′*a* :: *len0 word*)
by (*simp add*: *less-le*)

lemmas *word-gt-0-no* [*simp*] = *word-gt-0* [*of numeral y*] **for** *y*

lemma *word-sless-alt*: (*a* <*s b*) = (*sint a* < *sint b*)
unfolding *word-sle-def word-sless-def*
by (*auto simp add*: *less-le*)

lemma *word-le-nat-alt*: (*a* <= *b*) = (*unat a* <= *unat b*)
unfolding *unat-def word-le-def*
by (*rule nat-le-eq-zle* [*symmetric*]) *simp*

lemma *word-less-nat-alt*: (*a* < *b*) = (*unat a* < *unat b*)
unfolding *unat-def word-less-alt*
by (*rule nat-less-eq-zless* [*symmetric*]) *simp*

lemma *wi-less*:
(*word-of-int n* < (*word-of-int m* :: '*a* :: *len0 word*)) =
(*n mod 2* ˆ *len-of TYPE*('*a*) < *m mod 2* ˆ *len-of TYPE*('*a*))
unfolding *word-less-alt* **by** (*simp add*: *word-uint.eq-norm*)

lemma *wi-le*:
(*word-of-int n* <= (*word-of-int m* :: '*a* :: *len0 word*)) =
(*n mod 2* ˆ *len-of TYPE*('*a*) <= *m mod 2* ˆ *len-of TYPE*('*a*))
unfolding *word-le-def* **by** (*simp add*: *word-uint.eq-norm*)

lemma *udvd-nat-alt*: *a udvd b* = (*EX n*>=*0*. *unat b* = *n* ∗ *unat a*)
apply (*unfold udvd-def*)
apply *safe*
apply (*simp add*: *unat-def nat-mult-distrib*)
apply (*simp add*: *uint-nat of-nat-mult*)
apply (*rule exI*)
apply *safe*
prefer *2*
apply (*erule notE*)
apply (*rule refl*)
apply *force*
done

lemma *udvd-iff-dvd*: *x udvd y* ⟷ *unat x dvd unat y*
unfolding *dvd-def udvd-nat-alt* **by** *force*

lemmas *unat-mono* = *word-less-nat-alt* [*THEN iffD1*]

lemma *unat-minus-one*:
assumes *w* ≠ *0*
shows *unat* (*w* − *1*) = *unat w* − *1*
proof −
have *0* ≤ *uint w* **by** (*fact uint-nonnegative*)
moreover from *assms* **have** *0* ≠ *uint w* **by** (*simp add*: *uint-0-iff*)
ultimately have *1* ≤ *uint w* **by** *arith*
from *uint-lt2p* [*of w*] **have** *uint w* − *1* < *2* ˆ *len-of TYPE*('*a*) **by** *arith*
with ‹*1* ≤ *uint w*› **have** (*uint w* − *1*) *mod 2* ˆ *len-of TYPE*('*a*) = *uint w* − *1*
by (*auto intro*: *mod-pos-pos-trivial*)
with ‹*1* ≤ *uint w*› **have** *nat* ((*uint w* − *1*) *mod 2* ˆ *len-of TYPE*('*a*)) = *nat* (*uint w*) − *1*
by *auto*

then show *?thesis*
by (*simp only*: *unat-def int-word-uint word-arith-wis mod-diff-right-eq* [*symmetric*])
qed

lemma *measure-unat*: *p* ~= *0* ⟹ *unat* (*p* − *1*) < *unat p*
by (*simp add*: *unat-minus-one*) (*simp add*: *unat-0-iff* [*symmetric*])

lemmas *uint-add-ge0* [*simp*] = *add-nonneg-nonneg* [*OF uint-ge-0 uint-ge-0*]
lemmas *uint-mult-ge0* [*simp*] = *mult-nonneg-nonneg* [*OF uint-ge-0 uint-ge-0*]

lemma *uint-sub-lt2p* [*simp*]:
uint (*x* :: *'a* :: *len0 word*) − *uint* (*y* :: *'b* :: *len0 word*) <
2 ^ *len-of TYPE*(*'a*)
using *uint-ge-0* [*of y*] *uint-lt2p* [*of x*] **by** *arith*

127.17 Conditions for the addition (etc) of two words to overflow

lemma *uint-add-lem*:
(*uint x* + *uint y* < *2* ^ *len-of TYPE*(*'a*)) =
(*uint* (*x* + *y* :: *'a* :: *len0 word*) = *uint x* + *uint y*)
by (*unfold uint-word-ariths*) (*auto intro*!: *trans* [*OF - int-mod-lem*])

lemma *uint-mult-lem*:
(*uint x* ∗ *uint y* < *2* ^ *len-of TYPE*(*'a*)) =
(*uint* (*x* ∗ *y* :: *'a* :: *len0 word*) = *uint x* ∗ *uint y*)
by (*unfold uint-word-ariths*) (*auto intro*!: *trans* [*OF - int-mod-lem*])

lemma *uint-sub-lem*:
(*uint x* >= *uint y*) = (*uint* (*x* − *y*) = *uint x* − *uint y*)
by (*unfold uint-word-ariths*) (*auto intro*!: *trans* [*OF - int-mod-lem*])

lemma *uint-add-le*: *uint* (*x* + *y*) <= *uint x* + *uint y*
unfolding *uint-word-ariths* **by** (*metis uint-add-ge0 zmod-le-nonneg-dividend*)

lemma *uint-sub-ge*: *uint* (*x* − *y*) >= *uint x* − *uint y*
unfolding *uint-word-ariths* **by** (*metis int-mod-ge uint-sub-lt2p zless2p*)

lemma *mod-add-if-z*:
(*x* :: *int*) < *z* ==> *y* < *z* ==> *0* <= *y* ==> *0* <= *x* ==> *0* <= *z* ==>
(*x* + *y*) *mod z* = (*if x* + *y* < *z then x* + *y else x* + *y* − *z*)
by (*auto intro*: *int-mod-eq*)

lemma *uint-plus-if'*:
uint ((*a*::*'a word*) + *b*) =
(*if uint a* + *uint b* < *2* ^ *len-of TYPE*(*'a*::*len0*) *then uint a* + *uint b*
else uint a + *uint b* − *2* ^ *len-of TYPE*(*'a*))
using *mod-add-if-z* [*of uint a - uint b*] **by** (*simp add*: *uint-word-ariths*)

lemma *mod-sub-if-z*:
(x :: int) < z ==> y < z ==> 0 <= y ==> 0 <= x ==> 0 <= z ==>
(x − y) mod z = (if y <= x then x − y else x − y + z)
by (auto intro: int-mod-eq)

lemma *uint-sub-if'*:
uint ((a::'a word) − b) =
(if uint b ≤ uint a then uint a − uint b
else uint a − uint b + 2 ^ len-of TYPE('a::len0))
using mod-sub-if-z [of uint a - uint b] **by** (simp add: uint-word-ariths)

127.18 Definition of *uint-arith*

lemma *word-of-int-inverse*:
word-of-int r = a ⟹ 0 <= r ⟹ r < 2 ^ len-of TYPE('a) ⟹
uint (a::'a::len0 word) = r
apply (erule word-uint.Abs-inverse' [rotated])
apply (simp add: uints-num)
done

lemma *uint-split*:
fixes x::'a::len0 word
shows P (uint x) =
(ALL i. word-of-int i = x & 0 <= i & i < 2^len-of TYPE('a) --> P i)
apply (fold word-int-case-def)
apply (auto dest!: word-of-int-inverse simp: int-word-uint mod-pos-pos-trivial
split: word-int-split)
done

lemma *uint-split-asm*:
fixes x::'a::len0 word
shows P (uint x) =
(~(EX i. word-of-int i = x & 0 <= i & i < 2^len-of TYPE('a) & ~ P i))
by (auto dest!: word-of-int-inverse
simp: int-word-uint mod-pos-pos-trivial
split: uint-split)

lemmas *uint-splits* = uint-split uint-split-asm

lemmas *uint-arith-simps* =
word-le-def word-less-alt
word-uint.Rep-inject [symmetric]
uint-sub-if' uint-plus-if'

lemma *power-False-cong*: False ⟹ a ^ b = c ^ d
by auto

ML ‹
```
fun uint-arith-simpset ctxt =
  ctxt addsimps @{thms uint-arith-simps}
     delsimps @{thms word-uint.Rep-inject}
     |> fold Splitter.add-split @{thms if-split-asm}
     |> fold Simplifier.add-cong @{thms power-False-cong}

fun uint-arith-tacs ctxt =
  let
    fun arith-tac' n t =
      Arith-Data.arith-tac ctxt n t
        handle Cooper.COOPER - => Seq.empty;
  in
    [ clarify-tac ctxt 1,
      full-simp-tac (uint-arith-simpset ctxt) 1,
      ALLGOALS (full-simp-tac
        (put-simpset HOL-ss ctxt
          |> fold Splitter.add-split @{thms uint-splits}
          |> fold Simplifier.add-cong @{thms power-False-cong})),
      rewrite-goals-tac ctxt @{thms word-size},
      ALLGOALS (fn n => REPEAT (resolve-tac ctxt [allI, impI] n) THEN
                         REPEAT (eresolve-tac ctxt [conjE] n) THEN
                         REPEAT (dresolve-tac ctxt @{thms word-of-int-inverse} n
                                 THEN assume-tac ctxt n
                                 THEN assume-tac ctxt n)),
      TRYALL arith-tac' ]
  end

fun uint-arith-tac ctxt = SELECT-GOAL (EVERY (uint-arith-tacs ctxt))
```
›

method-setup *uint-arith* =
‹*Scan.succeed (SIMPLE-METHOD' o uint-arith-tac)*›
solving word arithmetic via integers and arith

127.19 More on overflows and monotonicity

lemma *no-plus-overflow-uint-size*:
((*x* :: '*a* :: *len0 word*) <= *x* + *y*) = (*uint x* + *uint y* < *2* ^ *size x*)
unfolding *word-size* **by** *uint-arith*

lemmas *no-olen-add* = *no-plus-overflow-uint-size* [*unfolded word-size*]

lemma *no-ulen-sub*: ((*x* :: '*a* :: *len0 word*) >= *x* − *y*) = (*uint y* <= *uint x*)
by *uint-arith*

lemma *no-olen-add'*:
fixes *x* :: '*a*::*len0 word*
shows ($x \le y + x$) = (*uint y* + *uint x* < *2* ^ *len-of TYPE*('*a*))

by (*simp add*: *ac-simps no-olen-add*)

lemmas *olen-add-eqv* = *trans* [*OF no-olen-add no-olen-add'* [*symmetric*]]

lemmas *uint-plus-simple-iff* = *trans* [*OF no-olen-add uint-add-lem*]
lemmas *uint-plus-simple* = *uint-plus-simple-iff* [*THEN iffD1*]
lemmas *uint-minus-simple-iff* = *trans* [*OF no-ulen-sub uint-sub-lem*]
lemmas *uint-minus-simple-alt* = *uint-sub-lem* [*folded word-le-def*]
lemmas *word-sub-le-iff* = *no-ulen-sub* [*folded word-le-def*]
lemmas *word-sub-le* = *word-sub-le-iff* [*THEN iffD2*]

lemma *word-less-sub1*:
$(x :: 'a :: len\ word)\ \tilde{}{=}\ 0 \Longrightarrow (1 < x) = (0 < x - 1)$
by *uint-arith*

lemma *word-le-sub1*:
$(x :: 'a :: len\ word)\ \tilde{}{=}\ 0 \Longrightarrow (1 <= x) = (0 <= x - 1)$
by *uint-arith*

lemma *sub-wrap-lt*:
$((x :: 'a :: len0\ word) < x - z) = (x < z)$
by *uint-arith*

lemma *sub-wrap*:
$((x :: 'a :: len0\ word) <= x - z) = (z = 0 \mid x < z)$
by *uint-arith*

lemma *plus-minus-not-NULL-ab*:
$(x :: 'a :: len0\ word) <= ab - c \Longrightarrow c <= ab \Longrightarrow c\ \tilde{}{=}\ 0 \Longrightarrow x + c\ \tilde{}{=}\ 0$
by *uint-arith*

lemma *plus-minus-no-overflow-ab*:
$(x :: 'a :: len0\ word) <= ab - c \Longrightarrow c <= ab \Longrightarrow x <= x + c$
by *uint-arith*

lemma *le-minus'*:
$(a :: 'a :: len0\ word) + c <= b \Longrightarrow a <= a + c \Longrightarrow c <= b - a$
by *uint-arith*

lemma *le-plus'*:
$(a :: 'a :: len0\ word) <= b \Longrightarrow c <= b - a \Longrightarrow a + c <= b$
by *uint-arith*

lemmas *le-plus* = *le-plus'* [*rotated*]

lemmas *le-minus* = *leD* [*THEN thin-rl*, *THEN le-minus'*]

lemma *word-plus-mono-right*:
$(y :: 'a :: len0\ word) <= z \Longrightarrow x <= x + z \Longrightarrow x + y <= x + z$

by *uint-arith*

lemma *word-less-minus-cancel*:
$y - x < z - x \Longrightarrow x <= z \Longrightarrow (y :: 'a :: len0\ word) < z$
by *uint-arith*

lemma *word-less-minus-mono-left*:
$(y :: 'a :: len0\ word) < z \Longrightarrow x <= y \Longrightarrow y - x < z - x$
by *uint-arith*

lemma *word-less-minus-mono*:
$a < c \Longrightarrow d < b \Longrightarrow a - b < a \Longrightarrow c - d < c$
$\Longrightarrow a - b < c - (d::'a::len\ word)$
by *uint-arith*

lemma *word-le-minus-cancel*:
$y - x <= z - x \Longrightarrow x <= z \Longrightarrow (y :: 'a :: len0\ word) <= z$
by *uint-arith*

lemma *word-le-minus-mono-left*:
$(y :: 'a :: len0\ word) <= z \Longrightarrow x <= y \Longrightarrow y - x <= z - x$
by *uint-arith*

lemma *word-le-minus-mono*:
$a <= c \Longrightarrow d <= b \Longrightarrow a - b <= a \Longrightarrow c - d <= c$
$\Longrightarrow a - b <= c - (d::'a::len\ word)$
by *uint-arith*

lemma *plus-le-left-cancel-wrap*:
$(x :: 'a :: len0\ word) + y' < x \Longrightarrow x + y < x \Longrightarrow (x + y' < x + y) = (y' < y)$
by *uint-arith*

lemma *plus-le-left-cancel-nowrap*:
$(x :: 'a :: len0\ word) <= x + y' \Longrightarrow x <= x + y \Longrightarrow$
$(x + y' < x + y) = (y' < y)$
by *uint-arith*

lemma *word-plus-mono-right2*:
$(a :: 'a :: len0\ word) <= a + b \Longrightarrow c <= b \Longrightarrow a <= a + c$
by *uint-arith*

lemma *word-less-add-right*:
$(x :: 'a :: len0\ word) < y - z \Longrightarrow z <= y \Longrightarrow x + z < y$
by *uint-arith*

lemma *word-less-sub-right*:
$(x :: 'a :: len0\ word) < y + z \Longrightarrow y <= x \Longrightarrow x - y < z$
by *uint-arith*

lemma *word-le-plus-either*:
  $(x :: 'a :: len0\ word) <= y \mid x <= z \Longrightarrow y <= y + z \Longrightarrow x <= y + z$
  by *uint-arith*

lemma *word-less-nowrapI*:
  $(x :: 'a :: len0\ word) < z - k \Longrightarrow k <= z \Longrightarrow 0 < k \Longrightarrow x < x + k$
  by *uint-arith*

lemma *inc-le*: $(i :: 'a :: len\ word) < m \Longrightarrow i + 1 <= m$
  by *uint-arith*

lemma *inc-i*:
  $(1 :: 'a :: len\ word) <= i \Longrightarrow i < m \Longrightarrow 1 <= (i + 1)\ \&\ i + 1 <= m$
  by *uint-arith*

lemma *udvd-incr-lem*:
  $up < uq \Longrightarrow up = ua + n * uint\ K \Longrightarrow$
    $uq = ua + n' * uint\ K \Longrightarrow up + uint\ K <= uq$
  apply *clarsimp*

  apply (*drule less-le-mult*)
  apply *safe*
  done

lemma *udvd-incr'*:
  $p < q \Longrightarrow uint\ p = ua + n * uint\ K \Longrightarrow$
    $uint\ q = ua + n' * uint\ K \Longrightarrow p + K <= q$
  apply (*unfold word-less-alt word-le-def*)
  apply (*drule* (*2*) *udvd-incr-lem*)
  apply (*erule uint-add-le* [*THEN order-trans*])
  done

lemma *udvd-decr'*:
  $p < q \Longrightarrow uint\ p = ua + n * uint\ K \Longrightarrow$
    $uint\ q = ua + n' * uint\ K \Longrightarrow p <= q - K$
  apply (*unfold word-less-alt word-le-def*)
  apply (*drule* (*2*) *udvd-incr-lem*)
  apply (*drule le-diff-eq* [*THEN iffD2*])
  apply (*erule order-trans*)
  apply (*rule uint-sub-ge*)
  done

lemmas *udvd-incr-lem0* = *udvd-incr-lem* [**where** *ua*=*0*, *unfolded add-0-left*]
lemmas *udvd-incr0* = *udvd-incr'* [**where** *ua*=*0*, *unfolded add-0-left*]
lemmas *udvd-decr0* = *udvd-decr'* [**where** *ua*=*0*, *unfolded add-0-left*]

lemma *udvd-minus-le'*:
  $xy < k \Longrightarrow z\ udvd\ xy \Longrightarrow z\ udvd\ k \Longrightarrow xy <= k - z$
  apply (*unfold udvd-def*)

apply *clarify*
apply (*erule* (*2*) *udvd-decr0*)
done

lemma *udvd-incr2-K*:
$p < a + s \Longrightarrow a <= a + s \Longrightarrow K\ udvd\ s \Longrightarrow K\ udvd\ p - a \Longrightarrow a <= p \Longrightarrow$
$0 < K \Longrightarrow p <= p + K\ \&\ p + K <= a + s$
using [[*simproc del*: *linordered-ring-less-cancel-factor*]]
apply (*unfold udvd-def*)
apply *clarify*
apply (*simp add*: *uint-arith-simps split*: *if-split-asm*)
prefer *2*
apply (*insert uint-range'* [*of s*])[*1*]
apply *arith*
apply (*drule add.commute* [*THEN xtr1*])
apply (*simp add*: *diff-less-eq* [*symmetric*])
apply (*drule less-le-mult*)
apply *arith*
apply *simp*
done

lemma *word-succ-rbl*:
to-bl w = *bl* $\Longrightarrow$ *to-bl* (*word-succ w*) = (*rev* (*rbl-succ* (*rev bl*)))
apply (*unfold word-succ-def*)
apply *clarify*
apply (*simp add*: *to-bl-of-bin*)
apply (*simp add*: *to-bl-def rbl-succ*)
done

lemma *word-pred-rbl*:
to-bl w = *bl* $\Longrightarrow$ *to-bl* (*word-pred w*) = (*rev* (*rbl-pred* (*rev bl*)))
apply (*unfold word-pred-def*)
apply *clarify*
apply (*simp add*: *to-bl-of-bin*)
apply (*simp add*: *to-bl-def rbl-pred*)
done

lemma *word-add-rbl*:
to-bl v = *vbl* $\Longrightarrow$ *to-bl w* = *wbl* $\Longrightarrow$
to-bl (*v* + *w*) = (*rev* (*rbl-add* (*rev vbl*) (*rev wbl*)))
apply (*unfold word-add-def*)
apply *clarify*
apply (*simp add*: *to-bl-of-bin*)
apply (*simp add*: *to-bl-def rbl-add*)
done

lemma *word-mult-rbl*:
to-bl v = *vbl* $\Longrightarrow$ *to-bl w* = *wbl* $\Longrightarrow$

to-bl (*v* ∗ *w*) = (*rev* (*rbl-mult* (*rev vbl*) (*rev wbl*)))
apply (*unfold word-mult-def*)
apply *clarify*
apply (*simp add*: *to-bl-of-bin*)
apply (*simp add*: *to-bl-def rbl-mult*)
done

lemma *rtb-rbl-ariths*:
rev (*to-bl w*) = *ys* ⟹ *rev* (*to-bl* (*word-succ w*)) = *rbl-succ ys*
rev (*to-bl w*) = *ys* ⟹ *rev* (*to-bl* (*word-pred w*)) = *rbl-pred ys*
rev (*to-bl v*) = *ys* ⟹ *rev* (*to-bl w*) = *xs* ⟹ *rev* (*to-bl* (*v* ∗ *w*)) = *rbl-mult ys xs*
rev (*to-bl v*) = *ys* ⟹ *rev* (*to-bl w*) = *xs* ⟹ *rev* (*to-bl* (*v* + *w*)) = *rbl-add ys xs*
by (*auto simp*: *rev-swap* [*symmetric*] *word-succ-rbl*
word-pred-rbl word-mult-rbl word-add-rbl)

127.20 Arithmetic type class instantiations

lemmas *word-le-0-iff* [*simp*] =
word-zero-le [*THEN leD*, *THEN linorder-antisym-conv1*]

lemma *word-of-int-nat*:
0 <= *x* ⟹ *word-of-int x* = *of-nat* (*nat x*)
by (*simp add*: *of-nat-nat word-of-int*)

lemma *iszero-word-no* [*simp*]:
iszero (*numeral bin* :: ′*a* :: *len word*) =
iszero (*bintrunc* (*len-of TYPE*(′*a*)) (*numeral bin*))
using *word-ubin.norm-eq-iff* [**where** ′*a*=′*a*, *of numeral bin 0*]
by (*simp add*: *iszero-def* [*symmetric*])

Use *iszero* to simplify equalities between word numerals.

lemmas *word-eq-numeral-iff-iszero* [*simp*] =
eq-numeral-iff-iszero [**where** ′*a*=′*a*::*len word*]

127.21 Word and nat

lemma *td-ext-unat* [*OF refl*]:
n = *len-of TYPE* (′*a* :: *len*) ⟹
td-ext (*unat* :: ′*a word* => *nat*) *of-nat*
(*unats n*) (%*i*. *i mod 2* ˆ *n*)
apply (*unfold td-ext-def* ′ *unat-def word-of-nat unats-uints*)
apply (*auto intro*!: *imageI simp add* : *word-of-int-hom-syms*)
apply (*erule word-uint.Abs-inverse* [*THEN arg-cong*])
apply (*simp add*: *int-word-uint nat-mod-distrib nat-power-eq*)
done

lemmas *unat-of-nat* = *td-ext-unat* [*THEN td-ext.eq-norm*]

interpretation *word-unat*:
td-ext unat::'a::len word => nat
of-nat
unats (len-of TYPE('a::len))
%i. i mod 2 ^ len-of TYPE('a::len)
by (*rule td-ext-unat*)

lemmas *td-unat = word-unat.td-thm*

lemmas *unat-lt2p* [*iff*] = *word-unat.Rep* [*unfolded unats-def mem-Collect-eq*]

lemma *unat-le*: *y <= unat (z :: 'a :: len word)* $\Longrightarrow$ *y : unats (len-of TYPE ('a))*
apply (*unfold unats-def*)
apply *clarsimp*
apply (*rule xtrans*, *rule unat-lt2p*, *assumption*)
done

lemma *word-nchotomy*:
ALL w. EX n. (w :: 'a :: len word) = of-nat n & n < 2 ^ len-of TYPE ('a)
apply (*rule allI*)
apply (*rule word-unat.Abs-cases*)
apply (*unfold unats-def*)
apply *auto*
done

lemma *of-nat-eq*:
fixes *w :: 'a::len word*
shows (*of-nat n = w*) = ($\exists$ *q. n = unat w + q * 2 ^ len-of TYPE('a)*)
apply (*rule trans*)
apply (*rule word-unat.inverse-norm*)
apply (*rule iffI*)
apply (*rule mod-eqD*)
apply *simp*
apply *clarsimp*
done

lemma *of-nat-eq-size*:
(*of-nat n = w*) = (*EX q. n = unat w + q * 2 ^ size w*)
unfolding *word-size* **by** (*rule of-nat-eq*)

lemma *of-nat-0*:
(*of-nat m = (0::'a::len word)*) = ($\exists$ *q. m = q * 2 ^ len-of TYPE('a)*)
by (*simp add*: *of-nat-eq*)

lemma *of-nat-2p* [*simp*]:
of-nat (2 ^ len-of TYPE('a)) = (0::'a::len word)
by (*fact mult-1* [*symmetric*, *THEN iffD2* [*OF of-nat-0 exI*]])

lemma *of-nat-gt-0*: *of-nat k ~= 0* $\Longrightarrow$ *0 < k*

by (*cases k*) *auto*

lemma *of-nat-neq-0*:
0 < *k* ⟹ *k* < *2* ^ *len-of TYPE* (′*a* :: *len*) ⟹ *of-nat k* ~= (*0* :: ′*a word*)
by (*clarsimp simp add* : *of-nat-0*)

lemma *Abs-fnat-hom-add*:
of-nat a + *of-nat b* = *of-nat* (*a* + *b*)
by *simp*

lemma *Abs-fnat-hom-mult*:
of-nat a ∗ *of-nat b* = (*of-nat* (*a* ∗ *b*) :: ′*a* :: *len word*)
by (*simp add*: *word-of-nat wi-hom-mult*)

lemma *Abs-fnat-hom-Suc*:
word-succ (*of-nat a*) = *of-nat* (*Suc a*)
by (*simp add*: *word-of-nat wi-hom-succ ac-simps*)

lemma *Abs-fnat-hom-0*: (*0*::′*a*::*len word*) = *of-nat 0*
by *simp*

lemma *Abs-fnat-hom-1*: (*1*::′*a*::*len word*) = *of-nat* (*Suc 0*)
by *simp*

lemmas *Abs-fnat-homs* =
Abs-fnat-hom-add Abs-fnat-hom-mult Abs-fnat-hom-Suc
Abs-fnat-hom-0 Abs-fnat-hom-1

lemma *word-arith-nat-add*:
a + *b* = *of-nat* (*unat a* + *unat b*)
by *simp*

lemma *word-arith-nat-mult*:
a ∗ *b* = *of-nat* (*unat a* ∗ *unat b*)
by (*simp add*: *of-nat-mult*)

lemma *word-arith-nat-Suc*:
word-succ a = *of-nat* (*Suc* (*unat a*))
by (*subst Abs-fnat-hom-Suc* [*symmetric*]) *simp*

lemma *word-arith-nat-div*:
a div b = *of-nat* (*unat a div unat b*)
by (*simp add*: *word-div-def word-of-nat zdiv-int uint-nat*)

lemma *word-arith-nat-mod*:
a mod b = *of-nat* (*unat a mod unat b*)
by (*simp add*: *word-mod-def word-of-nat zmod-int uint-nat*)

lemmas *word-arith-nat-defs* =

```
  word-arith-nat-add word-arith-nat-mult
  word-arith-nat-Suc Abs-fnat-hom-0
  Abs-fnat-hom-1 word-arith-nat-div
  word-arith-nat-mod

lemma unat-cong: x = y ⟹ unat x = unat y
  by simp

lemmas unat-word-ariths = word-arith-nat-defs
  [THEN trans [OF unat-cong unat-of-nat]]

lemmas word-sub-less-iff = word-sub-le-iff
  [unfolded linorder-not-less [symmetric] Not-eq-iff]

lemma unat-add-lem:
  (unat x + unat y < 2 ^ len-of TYPE('a)) =
    (unat (x + y :: 'a :: len word) = unat x + unat y)
  unfolding unat-word-ariths
  by (auto intro!: trans [OF - nat-mod-lem])

lemma unat-mult-lem:
  (unat x * unat y < 2 ^ len-of TYPE('a)) =
    (unat (x * y :: 'a :: len word) = unat x * unat y)
  unfolding unat-word-ariths
  by (auto intro!: trans [OF - nat-mod-lem])

lemmas unat-plus-if' = trans [OF unat-word-ariths(1) mod-nat-add, simplified]

lemma le-no-overflow:
  x <= b ⟹ a <= a + b ⟹ x <= a + (b :: 'a :: len0 word)
  apply (erule order-trans)
  apply (erule olen-add-eqv [THEN iffD1])
  done

lemmas un-ui-le = trans [OF word-le-nat-alt [symmetric] word-le-def]

lemma unat-sub-if-size:
  unat (x − y) = (if unat y <= unat x
   then unat x − unat y
   else unat x + 2 ^ size x − unat y)
  apply (unfold word-size)
  apply (simp add: un-ui-le)
  apply (auto simp add: unat-def uint-sub-if')
   apply (rule nat-diff-distrib)
    prefer 3
    apply (simp add: algebra-simps)
    apply (rule nat-diff-distrib [THEN trans])
      prefer 3
      apply (subst nat-add-distrib)
```

prefer *3*
apply (*simp add*: *nat-power-eq*)
apply *auto*
apply *uint-arith*
done

lemmas *unat-sub-if'* = *unat-sub-if-size* [*unfolded word-size*]

lemma *unat-div*: *unat* ((*x* :: *'a* :: *len word*) *div y*) = *unat x div unat y*
apply (*simp add* : *unat-word-ariths*)
apply (*rule unat-lt2p* [*THEN xtr7*, *THEN nat-mod-eq'*])
apply (*rule div-le-dividend*)
done

lemma *unat-mod*: *unat* ((*x* :: *'a* :: *len word*) *mod y*) = *unat x mod unat y*
apply (*clarsimp simp add* : *unat-word-ariths*)
apply (*cases unat y*)
prefer *2*
apply (*rule unat-lt2p* [*THEN xtr7*, *THEN nat-mod-eq'*])
apply (*rule mod-le-divisor*)
apply *auto*
done

lemma *uint-div*: *uint* ((*x* :: *'a* :: *len word*) *div y*) = *uint x div uint y*
unfolding *uint-nat* **by** (*simp add* : *unat-div zdiv-int*)

lemma *uint-mod*: *uint* ((*x* :: *'a* :: *len word*) *mod y*) = *uint x mod uint y*
unfolding *uint-nat* **by** (*simp add* : *unat-mod zmod-int*)

127.22 Definition of *unat-arith* tactic

lemma *unat-split*:
fixes *x*::*'a*::*len word*
shows *P* (*unat x*) =
(*ALL n*. *of-nat n* = *x* & *n* < *2^len-of TYPE*(*'a*) --> *P n*)
by (*auto simp*: *unat-of-nat*)

lemma *unat-split-asm*:
fixes *x*::*'a*::*len word*
shows *P* (*unat x*) =
(~(*EX n*. *of-nat n* = *x* & *n* < *2^len-of TYPE*(*'a*) & ~ *P n*))
by (*auto simp*: *unat-of-nat*)

lemmas *of-nat-inverse* =
word-unat.Abs-inverse' [*rotated*, *unfolded unats-def*, *simplified*]

lemmas *unat-splits* = *unat-split unat-split-asm*

lemmas *unat-arith-simps* =

word-le-nat-alt word-less-nat-alt
word-unat.Rep-inject [*symmetric*]
unat-sub-if′ unat-plus-if′ unat-div unat-mod

ML ‹
```
fun unat-arith-simpset ctxt =
  ctxt addsimps @{thms unat-arith-simps}
     delsimps @{thms word-unat.Rep-inject}
     |> fold Splitter.add-split @{thms if-split-asm}
     |> fold Simplifier.add-cong @{thms power-False-cong}

fun unat-arith-tacs ctxt =
  let
    fun arith-tac′ n t =
      Arith-Data.arith-tac ctxt n t
        handle Cooper.COOPER - => Seq.empty;
  in
    [ clarify-tac ctxt 1,
      full-simp-tac (unat-arith-simpset ctxt) 1,
      ALLGOALS (full-simp-tac
        (put-simpset HOL-ss ctxt
          |> fold Splitter.add-split @{thms unat-splits}
          |> fold Simplifier.add-cong @{thms power-False-cong})),
      rewrite-goals-tac ctxt @{thms word-size},
      ALLGOALS (fn n => REPEAT (resolve-tac ctxt [allI, impI] n) THEN
                        REPEAT (eresolve-tac ctxt [conjE] n) THEN
                        REPEAT (dresolve-tac ctxt @{thms of-nat-inverse} n THEN
assume-tac ctxt n)),
      TRYALL arith-tac′ ]
  end

fun unat-arith-tac ctxt = SELECT-GOAL (EVERY (unat-arith-tacs ctxt))
```
›

method-setup *unat-arith* =
‹*Scan.succeed* (*SIMPLE-METHOD′ o unat-arith-tac*)›
solving word arithmetic via natural numbers and arith

lemma *no-plus-overflow-unat-size*:
$((x :: 'a :: len\ word) <= x + y) = (unat\ x + unat\ y < 2\ \hat{}\ size\ x)$
unfolding *word-size* **by** *unat-arith*

lemmas *no-olen-add-nat* = *no-plus-overflow-unat-size* [*unfolded word-size*]

lemmas *unat-plus-simple* = *trans* [*OF no-olen-add-nat unat-add-lem*]

lemma *word-div-mult*:
$(0 :: 'a :: len\ word) < y \Longrightarrow unat\ x * unat\ y < 2\ \hat{}\ len\text{-}of\ TYPE('a) \Longrightarrow$

x ∗ *y div y* = *x*
apply *unat-arith*
apply *clarsimp*
apply (*subst unat-mult-lem* [*THEN iffD1*])
apply *auto*
done

lemma *div-lt'*: (*i* :: *'a* :: *len word*) <= *k div x* ⟹
unat i ∗ *unat x* < *2* ^ *len-of TYPE*(*'a*)
apply *unat-arith*
apply *clarsimp*
apply (*drule mult-le-mono1*)
apply (*erule order-le-less-trans*)
apply (*rule xtr7* [*OF unat-lt2p div-mult-le*])
done

lemmas *div-lt''* = *order-less-imp-le* [*THEN div-lt'*]

lemma *div-lt-mult*: (*i* :: *'a* :: *len word*) < *k div x* ⟹ *0* < *x* ⟹ *i* ∗ *x* < *k*
apply (*frule div-lt''* [*THEN unat-mult-lem* [*THEN iffD1*]])
apply (*simp add*: *unat-arith-simps*)
apply (*drule* (*1*) *mult-less-mono1*)
apply (*erule order-less-le-trans*)
apply (*rule div-mult-le*)
done

lemma *div-le-mult*:
(*i* :: *'a* :: *len word*) <= *k div x* ⟹ *0* < *x* ⟹ *i* ∗ *x* <= *k*
apply (*frule div-lt'* [*THEN unat-mult-lem* [*THEN iffD1*]])
apply (*simp add*: *unat-arith-simps*)
apply (*drule mult-le-mono1*)
apply (*erule order-trans*)
apply (*rule div-mult-le*)
done

lemma *div-lt-uint'*:
(*i* :: *'a* :: *len word*) <= *k div x* ⟹ *uint i* ∗ *uint x* < *2* ^ *len-of TYPE*(*'a*)
apply (*unfold uint-nat*)
apply (*drule div-lt'*)
by (*metis of-nat-less-iff of-nat-mult of-nat-numeral of-nat-power*)

lemmas *div-lt-uint''* = *order-less-imp-le* [*THEN div-lt-uint'*]

lemma *word-le-exists'*:
(*x* :: *'a* :: *len0 word*) <= *y* ⟹
(*EX z*. *y* = *x* + *z* & *uint x* + *uint z* < *2* ^ *len-of TYPE*(*'a*))
apply (*rule exI*)
apply (*rule conjI*)
apply (*rule zadd-diff-inverse*)

apply *uint-arith*
done

lemmas *plus-minus-not-NULL* = *order-less-imp-le* [*THEN plus-minus-not-NULL-ab*]

lemmas *plus-minus-no-overflow* =
order-less-imp-le [*THEN plus-minus-no-overflow-ab*]

lemmas *mcs* = *word-less-minus-cancel word-less-minus-mono-left*
word-le-minus-cancel word-le-minus-mono-left

lemmas *word-l-diffs* = *mcs* [**where** $y = w + x$, *unfolded add-diff-cancel*] **for** w x
lemmas *word-diff-ls* = *mcs* [**where** $z = w + x$, *unfolded add-diff-cancel*] **for** w x
lemmas *word-plus-mcs* = *word-diff-ls* [**where** $y = v + x$, *unfolded add-diff-cancel*]
for v x

lemmas *le-unat-uoi* = *unat-le* [*THEN word-unat.Abs-inverse*]

lemmas *thd* = *refl* [*THEN* [*2*] *split-div-lemma* [*THEN iffD2*], *THEN conjunct1*]

lemmas *uno-simps* [*THEN le-unat-uoi*] = *mod-le-divisor div-le-dividend dtle*

lemma *word-mod-div-equality*:
(*n div b*) ∗ *b* + (*n mod b*) = (*n* :: ′*a* :: *len word*)
apply (*unfold word-less-nat-alt word-arith-nat-defs*)
apply (*cut-tac y=unat b* **in** *gt-or-eq-0*)
apply (*erule disjE*)
apply (*simp only*: *mod-div-equality uno-simps Word.word-unat.Rep-inverse*)
apply *simp*
done

lemma *word-div-mult-le*: *a div b* ∗ *b* <= (*a*::′*a*::*len word*)
apply (*unfold word-le-nat-alt word-arith-nat-defs*)
apply (*cut-tac y=unat b* **in** *gt-or-eq-0*)
apply (*erule disjE*)
apply (*simp only*: *div-mult-le uno-simps Word.word-unat.Rep-inverse*)
apply *simp*
done

lemma *word-mod-less-divisor*: *0* < *n* ⟹ *m mod n* < (*n* :: ′*a* :: *len word*)
apply (*simp only*: *word-less-nat-alt word-arith-nat-defs*)
apply (*clarsimp simp add* : *uno-simps*)
done

lemma *word-of-int-power-hom*:
word-of-int a ^ *n* = (*word-of-int* (*a* ^ *n*) :: ′*a* :: *len word*)
by (*induct n*) (*simp-all add*: *wi-hom-mult* [*symmetric*])

lemma *word-arith-power-alt*:

a ˆ n = (word-of-int (uint a ˆ n) :: 'a :: len word)
by *(simp add : word-of-int-power-hom [symmetric])*

lemma *of-bl-length-less*:
length x = k ⟹ k < len-of TYPE('a) ⟹ (of-bl x :: 'a :: len word) < 2 ˆ k
apply *(unfold of-bl-def word-less-alt word-numeral-alt)*
apply *safe*
apply *(simp (no-asm) add: word-of-int-power-hom word-uint.eq-norm*
del: word-of-int-numeral)
apply *(simp add: mod-pos-pos-trivial)*
apply *(subst mod-pos-pos-trivial)*
apply *(rule bl-to-bin-ge0)*
apply *(rule order-less-trans)*
apply *(rule bl-to-bin-lt2p)*
apply *simp*
apply *(rule bl-to-bin-lt2p)*
done

127.23 Cardinality, finiteness of set of words

instance *word :: (len0) finite*
by *standard (simp add: type-definition.univ [OF type-definition-word])*

lemma *card-word*: *CARD('a::len0 word) = 2 ˆ len-of TYPE('a)*
by *(simp add: type-definition.card [OF type-definition-word] nat-power-eq)*

lemma *card-word-size*:
card (UNIV :: 'a :: len0 word set) = (2 ˆ size (x :: 'a word))
unfolding *word-size* **by** *(rule card-word)*

127.24 Bitwise Operations on Words

lemmas *bin-log-bintrs = bin-trunc-not bin-trunc-xor bin-trunc-and bin-trunc-or*

lemmas *wils1 = bin-log-bintrs [THEN word-ubin.norm-eq-iff [THEN iffD1],*
folded word-ubin.eq-norm, THEN eq-reflection]

lemmas *word-log-binary-defs =*
word-and-def word-or-def word-xor-def

lemma *word-wi-log-defs*:
NOT word-of-int a = word-of-int (NOT a)
word-of-int a AND word-of-int b = word-of-int (a AND b)
word-of-int a OR word-of-int b = word-of-int (a OR b)
word-of-int a XOR word-of-int b = word-of-int (a XOR b)

by (*transfer*, *rule refl*)+

lemma *word-no-log-defs* [*simp*]:
NOT (*numeral a*) = *word-of-int* (*NOT* (*numeral a*))
NOT (− *numeral a*) = *word-of-int* (*NOT* (− *numeral a*))
numeral a AND numeral b = *word-of-int* (*numeral a AND numeral b*)
numeral a AND − *numeral b* = *word-of-int* (*numeral a AND* − *numeral b*)
− *numeral a AND numeral b* = *word-of-int* (− *numeral a AND numeral b*)
− *numeral a AND* − *numeral b* = *word-of-int* (− *numeral a AND* − *numeral b*)
numeral a OR numeral b = *word-of-int* (*numeral a OR numeral b*)
numeral a OR − *numeral b* = *word-of-int* (*numeral a OR* − *numeral b*)
− *numeral a OR numeral b* = *word-of-int* (− *numeral a OR numeral b*)
− *numeral a OR* − *numeral b* = *word-of-int* (− *numeral a OR* − *numeral b*)
numeral a XOR numeral b = *word-of-int* (*numeral a XOR numeral b*)
numeral a XOR − *numeral b* = *word-of-int* (*numeral a XOR* − *numeral b*)
− *numeral a XOR numeral b* = *word-of-int* (− *numeral a XOR numeral b*)
− *numeral a XOR* − *numeral b* = *word-of-int* (− *numeral a XOR* − *numeral b*)
by (*transfer*, *rule refl*)+

Special cases for when one of the arguments equals 1.

lemma *word-bitwise-1-simps* [*simp*]:
NOT (*1*::′*a*::*len0 word*) = −*2*
1 AND numeral b = *word-of-int* (*1 AND numeral b*)
1 AND − *numeral b* = *word-of-int* (*1 AND* − *numeral b*)
numeral a AND 1 = *word-of-int* (*numeral a AND 1*)
− *numeral a AND 1* = *word-of-int* (− *numeral a AND 1*)
1 OR numeral b = *word-of-int* (*1 OR numeral b*)
1 OR − *numeral b* = *word-of-int* (*1 OR* − *numeral b*)
numeral a OR 1 = *word-of-int* (*numeral a OR 1*)
− *numeral a OR 1* = *word-of-int* (− *numeral a OR 1*)
1 XOR numeral b = *word-of-int* (*1 XOR numeral b*)
1 XOR − *numeral b* = *word-of-int* (*1 XOR* − *numeral b*)
numeral a XOR 1 = *word-of-int* (*numeral a XOR 1*)
− *numeral a XOR 1* = *word-of-int* (− *numeral a XOR 1*)
by (*transfer*, *simp*)+

Special cases for when one of the arguments equals -1.

lemma *word-bitwise-m1-simps* [*simp*]:
NOT (−*1*::′*a*::*len0 word*) = *0*
(−*1*::′*a*::*len0 word*) *AND x* = *x*
x AND (−*1*::′*a*::*len0 word*) = *x*
(−*1*::′*a*::*len0 word*) *OR x* = −*1*
x OR (−*1*::′*a*::*len0 word*) = −*1*
(−*1*::′*a*::*len0 word*) *XOR x* = *NOT x*
x XOR (−*1*::′*a*::*len0 word*) = *NOT x*
by (*transfer*, *simp*)+

lemma *uint-or*: *uint* (*x OR y*) = (*uint x*) *OR* (*uint y*)
by (*transfer*, *simp add*: *bin-trunc-ao*)

lemma *uint-and*: *uint* (*x AND y*) = (*uint x*) *AND* (*uint y*)
by (*transfer*, *simp add*: *bin-trunc-ao*)

lemma *test-bit-wi* [*simp*]:
(*word-of-int x*::′*a*::*len0 word*) !! *n* ⟷ *n* < *len-of TYPE*(′*a*) ∧ *bin-nth x n*
unfolding *word-test-bit-def*
by (*simp add*: *word-ubin.eq-norm nth-bintr*)

lemma *word-test-bit-transfer* [*transfer-rule*]:
(*rel-fun pcr-word* (*rel-fun op* = *op* =))
(λ*x n*. *n* < *len-of TYPE*(′*a*) ∧ *bin-nth x n*) (*test-bit* :: ′*a*::*len0 word* ⇒ -)
unfolding *rel-fun-def word.pcr-cr-eq cr-word-def* **by** *simp*

lemma *word-ops-nth-size*:
n < *size* (*x*::′*a*::*len0 word*) ⟹
(*x OR y*) !! *n* = (*x* !! *n* | *y* !! *n*) &
(*x AND y*) !! *n* = (*x* !! *n* & *y* !! *n*) &
(*x XOR y*) !! *n* = (*x* !! *n* ~= *y* !! *n*) &
(*NOT x*) !! *n* = (~ *x* !! *n*)
unfolding *word-size* **by** *transfer* (*simp add*: *bin-nth-ops*)

lemma *word-ao-nth*:
fixes *x* :: ′*a*::*len0 word*
shows (*x OR y*) !! *n* = (*x* !! *n* | *y* !! *n*) &
(*x AND y*) !! *n* = (*x* !! *n* & *y* !! *n*)
by *transfer* (*auto simp add*: *bin-nth-ops*)

lemma *test-bit-numeral* [*simp*]:
(*numeral w* :: ′*a*::*len0 word*) !! *n* ⟷
n < *len-of TYPE*(′*a*) ∧ *bin-nth* (*numeral w*) *n*
by *transfer* (*rule refl*)

lemma *test-bit-neg-numeral* [*simp*]:
(− *numeral w* :: ′*a*::*len0 word*) !! *n* ⟷
n < *len-of TYPE*(′*a*) ∧ *bin-nth* (− *numeral w*) *n*
by *transfer* (*rule refl*)

lemma *test-bit-1* [*simp*]: (*1*::′*a*::*len word*) !! *n* ⟷ *n* = *0*
by *transfer auto*

lemma *nth-0* [*simp*]: ~ (*0*::′*a*::*len0 word*) !! *n*
by *transfer simp*

lemma *nth-minus1* [*simp*]: (−*1*::′*a*::*len0 word*) !! *n* ⟷ *n* < *len-of TYPE*(′*a*)
by *transfer simp*

lemmas *bwsimps* =
wi-hom-add
word-wi-log-defs

lemma *word-bw-assocs*:
fixes *x* :: *'a::len0 word*
shows
(*x AND y*) *AND z* = *x AND y AND z*
(*x OR y*) *OR z* = *x OR y OR z*
(*x XOR y*) *XOR z* = *x XOR y XOR z*
by (*auto simp*: *word-eq-iff word-ops-nth-size* [*unfolded word-size*])

lemma *word-bw-comms*:
fixes *x* :: *'a::len0 word*
shows
x AND y = *y AND x*
x OR y = *y OR x*
x XOR y = *y XOR x*
by (*auto simp*: *word-eq-iff word-ops-nth-size* [*unfolded word-size*])

lemma *word-bw-lcs*:
fixes *x* :: *'a::len0 word*
shows
y AND x AND z = *x AND y AND z*
y OR x OR z = *x OR y OR z*
y XOR x XOR z = *x XOR y XOR z*
by (*auto simp*: *word-eq-iff word-ops-nth-size* [*unfolded word-size*])

lemma *word-log-esimps* [*simp*]:
fixes *x* :: *'a::len0 word*
shows
x AND 0 = *0*
x AND −*1* = *x*
x OR 0 = *x*
x OR −*1* = −*1*
x XOR 0 = *x*
x XOR −*1* = *NOT x*
0 AND x = *0*
−*1 AND x* = *x*
0 OR x = *x*
−*1 OR x* = −*1*
0 XOR x = *x*
−*1 XOR x* = *NOT x*
by (*auto simp*: *word-eq-iff word-ops-nth-size* [*unfolded word-size*])

lemma *word-not-dist*:
fixes *x* :: *'a::len0 word*
shows
NOT (*x OR y*) = *NOT x AND NOT y*

NOT (*x AND y*) = *NOT x OR NOT y*
by (*auto simp*: *word-eq-iff word-ops-nth-size* [*unfolded word-size*])

lemma *word-bw-same*:
fixes *x* :: *'a*::*len0 word*
shows
x AND x = *x*
x OR x = *x*
x XOR x = *0*
by (*auto simp*: *word-eq-iff word-ops-nth-size* [*unfolded word-size*])

lemma *word-ao-absorbs* [*simp*]:
fixes *x* :: *'a*::*len0 word*
shows
x AND (*y OR x*) = *x*
x OR y AND x = *x*
x AND (*x OR y*) = *x*
y AND x OR x = *x*
(*y OR x*) *AND x* = *x*
x OR x AND y = *x*
(*x OR y*) *AND x* = *x*
x AND y OR x = *x*
by (*auto simp*: *word-eq-iff word-ops-nth-size* [*unfolded word-size*])

lemma *word-not-not* [*simp*]:
NOT NOT (*x*::*'a*::*len0 word*) = *x*
by (*auto simp*: *word-eq-iff word-ops-nth-size* [*unfolded word-size*])

lemma *word-ao-dist*:
fixes *x* :: *'a*::*len0 word*
shows (*x OR y*) *AND z* = *x AND z OR y AND z*
by (*auto simp*: *word-eq-iff word-ops-nth-size* [*unfolded word-size*])

lemma *word-oa-dist*:
fixes *x* :: *'a*::*len0 word*
shows *x AND y OR z* = (*x OR z*) *AND* (*y OR z*)
by (*auto simp*: *word-eq-iff word-ops-nth-size* [*unfolded word-size*])

lemma *word-add-not* [*simp*]:
fixes *x* :: *'a*::*len0 word*
shows *x* + *NOT x* = −*1*
by *transfer* (*simp add*: *bin-add-not*)

lemma *word-plus-and-or* [*simp*]:
fixes *x* :: *'a*::*len0 word*
shows (*x AND y*) + (*x OR y*) = *x* + *y*
by *transfer* (*simp add*: *plus-and-or*)

lemma *leoa*:

fixes *x* :: *'a*::*len0 word*
shows (*w* = (*x OR y*)) ⟹ (*y* = (*w AND y*)) **by** *auto*
lemma *leao*:
fixes *x'* :: *'a*::*len0 word*
shows (*w'* = (*x' AND y'*)) ⟹ (*x'* = (*x' OR w'*)) **by** *auto*

lemma *word-ao-equiv*:
fixes *w w'* :: *'a*::*len0 word*
shows (*w* = *w OR w'*) = (*w'* = *w AND w'*)
by (*auto intro*: *leoa leao*)

lemma *le-word-or2*: *x* <= *x OR* (*y*::*'a*::*len0 word*)
unfolding *word-le-def uint-or*
by (*auto intro*: *le-int-or*)

lemmas *le-word-or1* = *xtr3* [*OF word-bw-comms* (*2*) *le-word-or2*]
lemmas *word-and-le1* = *xtr3* [*OF word-ao-absorbs* (*4*) [*symmetric*] *le-word-or2*]
lemmas *word-and-le2* = *xtr3* [*OF word-ao-absorbs* (*8*) [*symmetric*] *le-word-or2*]

lemma *bl-word-not*: *to-bl* (*NOT w*) = *map Not* (*to-bl w*)
unfolding *to-bl-def word-log-defs bl-not-bin*
by (*simp add*: *word-ubin.eq-norm*)

lemma *bl-word-xor*: *to-bl* (*v XOR w*) = *map2 op* ~= (*to-bl v*) (*to-bl w*)
unfolding *to-bl-def word-log-defs bl-xor-bin*
by (*simp add*: *word-ubin.eq-norm*)

lemma *bl-word-or*: *to-bl* (*v OR w*) = *map2 op* | (*to-bl v*) (*to-bl w*)
unfolding *to-bl-def word-log-defs bl-or-bin*
by (*simp add*: *word-ubin.eq-norm*)

lemma *bl-word-and*: *to-bl* (*v AND w*) = *map2 op* & (*to-bl v*) (*to-bl w*)
unfolding *to-bl-def word-log-defs bl-and-bin*
by (*simp add*: *word-ubin.eq-norm*)

lemma *word-lsb-alt*: *lsb* (*w*::*'a*::*len0 word*) = *test-bit w 0*
by (*auto simp*: *word-test-bit-def word-lsb-def*)

lemma *word-lsb-1-0* [*simp*]: *lsb* (*1*::*'a*::*len word*) & ~ *lsb* (*0*::*'b*::*len0 word*)
unfolding *word-lsb-def uint-eq-0 uint-1* **by** *simp*

lemma *word-lsb-last*: *lsb* (*w*::*'a*::*len word*) = *last* (*to-bl w*)
apply (*unfold word-lsb-def uint-bl bin-to-bl-def*)
apply (*rule-tac bin*=*uint w* **in** *bin-exhaust*)
apply (*cases size w*)
apply *auto*
apply (*auto simp add*: *bin-to-bl-aux-alt*)
done

lemma *word-lsb-int*: *lsb w* = (*uint w mod 2* = *1*)
unfolding *word-lsb-def bin-last-def* **by** *auto*

lemma *word-msb-sint*: *msb w* = (*sint w* < *0*)
unfolding *word-msb-def sign-Min-lt-0* **..**

lemma *msb-word-of-int*:
msb (*word-of-int x*::*'a*::*len word*) = *bin-nth x* (*len-of TYPE*(*'a*) − *1*)
unfolding *word-msb-def* **by** (*simp add*: *word-sbin.eq-norm bin-sign-lem*)

lemma *word-msb-numeral* [*simp*]:
msb (*numeral w*::*'a*::*len word*) = *bin-nth* (*numeral w*) (*len-of TYPE*(*'a*) − *1*)
unfolding *word-numeral-alt* **by** (*rule msb-word-of-int*)

lemma *word-msb-neg-numeral* [*simp*]:
msb (− *numeral w*::*'a*::*len word*) = *bin-nth* (− *numeral w*) (*len-of TYPE*(*'a*) − *1*)
unfolding *word-neg-numeral-alt* **by** (*rule msb-word-of-int*)

lemma *word-msb-0* [*simp*]: ¬ *msb* (*0*::*'a*::*len word*)
unfolding *word-msb-def* **by** *simp*

lemma *word-msb-1* [*simp*]: *msb* (*1*::*'a*::*len word*) ⟷ *len-of TYPE*(*'a*) = *1*
unfolding *word-1-wi msb-word-of-int eq-iff* [**where** *'a*=*nat*]
by (*simp add*: *Suc-le-eq*)

lemma *word-msb-nth*:
msb (*w*::*'a*::*len word*) = *bin-nth* (*uint w*) (*len-of TYPE*(*'a*) − *1*)
unfolding *word-msb-def sint-uint* **by** (*simp add*: *bin-sign-lem*)

lemma *word-msb-alt*: *msb* (*w*::*'a*::*len word*) = *hd* (*to-bl w*)
apply (*unfold word-msb-nth uint-bl*)
apply (*subst hd-conv-nth*)
apply (*rule length-greater-0-conv* [*THEN iffD1*])
apply *simp*
apply (*simp add* : *nth-bin-to-bl word-size*)
done

lemma *word-set-nth* [*simp*]:
set-bit w n (*test-bit w n*) = (*w*::*'a*::*len0 word*)
unfolding *word-test-bit-def word-set-bit-def* **by** *auto*

lemma *bin-nth-uint'*:
bin-nth (*uint w*) *n* = (*rev* (*bin-to-bl* (*size w*) (*uint w*)) ! *n* & *n* < *size w*)
apply (*unfold word-size*)
apply (*safe elim*!: *bin-nth-uint-imp*)
apply (*frule bin-nth-uint-imp*)
apply (*fast dest*!: *bin-nth-bl*)+
done

lemmas *bin-nth-uint* = *bin-nth-uint′* [*unfolded word-size*]

lemma *test-bit-bl*: *w* !! *n* = (*rev* (*to-bl w*) ! *n* & *n* < *size w*)
  unfolding *to-bl-def word-test-bit-def word-size*
  by (*rule bin-nth-uint*)

lemma *to-bl-nth*: *n* < *size w* ⟹ *to-bl w* ! *n* = *w* !! (*size w* − *Suc n*)
  apply (*unfold test-bit-bl*)
  apply *clarsimp*
  apply (*rule trans*)
   apply (*rule nth-rev-alt*)
   apply (*auto simp add*: *word-size*)
  done

lemma *test-bit-set*:
  fixes *w* :: ′*a*::*len0 word*
  shows (*set-bit w n x*) !! *n* = (*n* < *size w* & *x*)
  unfolding *word-size word-test-bit-def word-set-bit-def*
  by (*clarsimp simp add* : *word-ubin.eq-norm nth-bintr*)

lemma *test-bit-set-gen*:
  fixes *w* :: ′*a*::*len0 word*
  shows *test-bit* (*set-bit w n x*) *m* =
        (*if m* = *n then n* < *size w* & *x else test-bit w m*)
  apply (*unfold word-size word-test-bit-def word-set-bit-def*)
  apply (*clarsimp simp add*: *word-ubin.eq-norm nth-bintr bin-nth-sc-gen*)
  apply (*auto elim*!: *test-bit-size* [*unfolded word-size*]
              simp add: *word-test-bit-def* [*symmetric*])
  done

lemma *of-bl-rep-False*: *of-bl* (*replicate n False* @ *bs*) = *of-bl bs*
  unfolding *of-bl-def bl-to-bin-rep-F* **by** *auto*

lemma *msb-nth*:
  fixes *w* :: ′*a*::*len word*
  shows *msb w* = *w* !! (*len-of TYPE*(′*a*) − *1*)
  unfolding *word-msb-nth word-test-bit-def* **by** *simp*

lemmas *msb0* = *len-gt-0* [*THEN diff-Suc-less*, *THEN word-ops-nth-size* [*unfolded word-size*]]
lemmas *msb1* = *msb0* [**where** *i* = *0*]
lemmas *word-ops-msb* = *msb1* [*unfolded msb-nth* [*symmetric*, *unfolded One-nat-def*]]

lemmas *lsb0* = *len-gt-0* [*THEN word-ops-nth-size* [*unfolded word-size*]]
lemmas *word-ops-lsb* = *lsb0* [*unfolded word-lsb-alt*]

lemma *td-ext-nth* [*OF refl refl refl*, *unfolded word-size*]:
  n = *size* (*w*::′*a*::*len0 word*) ⟹ *ofn* = *set-bits* ⟹ [*w*, *ofn g*] = *l* ⟹

td-ext test-bit ofn {*f*. *ALL i*. *f i* −−> *i* < *n*} (%*h i*. *h i* & *i* < *n*)
apply (*unfold word-size td-ext-def'*)
apply *safe*
apply (*rule-tac* [*3*] *ext*)
apply (*rule-tac* [*4*] *ext*)
apply (*unfold word-size of-nth-def test-bit-bl*)
apply *safe*
defer
apply (*clarsimp simp*: *word-bl.Abs-inverse*)+
apply (*rule word-bl.Rep-inverse'*)
apply (*rule sym* [*THEN trans*])
apply (*rule bl-of-nth-nth*)
apply *simp*
apply (*rule bl-of-nth-inj*)
apply (*clarsimp simp add* : *test-bit-bl word-size*)
done

interpretation *test-bit*:
td-ext op !! :: *'a*::*len0 word* => *nat* => *bool*
set-bits
{*f*. ∀ *i*. *f i* ⟶ *i* < *len-of TYPE*(*'a*::*len0*)}
(λ*h i*. *h i* ∧ *i* < *len-of TYPE*(*'a*::*len0*))
by (*rule td-ext-nth*)

lemmas *td-nth* = *test-bit.td-thm*

lemma *word-set-set-same* [*simp*]:
fixes *w* :: *'a*::*len0 word*
shows *set-bit* (*set-bit w n x*) *n y* = *set-bit w n y*
by (*rule word-eqI*) (*simp add* : *test-bit-set-gen word-size*)

lemma *word-set-set-diff*:
fixes *w* :: *'a*::*len0 word*
assumes *m* ~= *n*
shows *set-bit* (*set-bit w m x*) *n y* = *set-bit* (*set-bit w n y*) *m x*
by (*rule word-eqI*) (*clarsimp simp add*: *test-bit-set-gen word-size assms*)

lemma *nth-sint*:
fixes *w* :: *'a*::*len word*
defines *l* ≡ *len-of TYPE* (*'a*)
shows *bin-nth* (*sint w*) *n* = (*if n* < *l* − *1 then w* !! *n else w* !! (*l* − *1*))
unfolding *sint-uint l-def*
by (*clarsimp simp add*: *nth-sbintr word-test-bit-def* [*symmetric*])

lemma *word-lsb-numeral* [*simp*]:
lsb (*numeral bin* :: *'a* :: *len word*) ⟷ *bin-last* (*numeral bin*)
unfolding *word-lsb-alt test-bit-numeral* **by** *simp*

lemma *word-lsb-neg-numeral* [*simp*]:

lsb (− *numeral bin* :: ′*a* :: *len word*) ⟷ *bin-last* (− *numeral bin*)
unfolding *word-lsb-alt test-bit-neg-numeral* **by** *simp*

lemma *set-bit-word-of-int*:
set-bit (*word-of-int x*) *n b* = *word-of-int* (*bin-sc n b x*)
unfolding *word-set-bit-def*
apply (*rule word-eqI*)
apply (*simp add*: *word-size bin-nth-sc-gen word-ubin.eq-norm nth-bintr*)
done

lemma *word-set-numeral* [*simp*]:
set-bit (*numeral bin*::′*a*::*len0 word*) *n b* =
word-of-int (*bin-sc n b* (*numeral bin*))
unfolding *word-numeral-alt* **by** (*rule set-bit-word-of-int*)

lemma *word-set-neg-numeral* [*simp*]:
set-bit (− *numeral bin*::′*a*::*len0 word*) *n b* =
word-of-int (*bin-sc n b* (− *numeral bin*))
unfolding *word-neg-numeral-alt* **by** (*rule set-bit-word-of-int*)

lemma *word-set-bit-0* [*simp*]:
set-bit 0 n b = *word-of-int* (*bin-sc n b 0*)
unfolding *word-0-wi* **by** (*rule set-bit-word-of-int*)

lemma *word-set-bit-1* [*simp*]:
set-bit 1 n b = *word-of-int* (*bin-sc n b 1*)
unfolding *word-1-wi* **by** (*rule set-bit-word-of-int*)

lemma *setBit-no* [*simp*]:
setBit (*numeral bin*) *n* = *word-of-int* (*bin-sc n True* (*numeral bin*))
by (*simp add*: *setBit-def*)

lemma *clearBit-no* [*simp*]:
clearBit (*numeral bin*) *n* = *word-of-int* (*bin-sc n False* (*numeral bin*))
by (*simp add*: *clearBit-def*)

lemma *to-bl-n1*:
to-bl (−*1*::′*a*::*len0 word*) = *replicate* (*len-of TYPE* (′*a*)) *True*
apply (*rule word-bl.Abs-inverse*′)
apply *simp*
apply (*rule word-eqI*)
apply (*clarsimp simp add*: *word-size*)
apply (*auto simp add*: *word-bl.Abs-inverse test-bit-bl word-size*)
done

lemma *word-msb-n1* [*simp*]: *msb* (−*1*::′*a*::*len word*)
unfolding *word-msb-alt to-bl-n1* **by** *simp*

lemma *word-set-nth-iff*:

(*set-bit w n b = w*) = (*w* !! *n* = *b* | *n* >= *size* (*w*::′*a*::*len0 word*))
apply (*rule iffI*)
apply (*rule disjCI*)
apply (*drule word-eqD*)
apply (*erule sym* [*THEN trans*])
apply (*simp add*: *test-bit-set*)
apply (*erule disjE*)
apply *clarsimp*
apply (*rule word-eqI*)
apply (*clarsimp simp add* : *test-bit-set-gen*)
apply (*drule test-bit-size*)
apply *force*
done

lemma *test-bit-2p*:
(*word-of-int* (*2* ˆ *n*)::′*a*::*len word*) !! *m* ⟷ *m* = *n* ∧ *m* < *len-of TYPE*(′*a*)
unfolding *word-test-bit-def*
by (*auto simp add*: *word-ubin.eq-norm nth-bintr nth-2p-bin*)

lemma *nth-w2p*:
((*2*::′*a*::*len word*) ˆ *n*) !! *m* ⟷ *m* = *n* ∧ *m* < *len-of TYPE*(′*a*::*len*)
unfolding *test-bit-2p* [*symmetric*] *word-of-int* [*symmetric*]
by (*simp add*: *of-int-power*)

lemma *uint-2p*:
(*0*::′*a*::*len word*) < *2* ˆ *n* ⟹ *uint* (*2* ˆ *n*::′*a*::*len word*) = *2* ˆ *n*
apply (*unfold word-arith-power-alt*)
apply (*case-tac len-of TYPE* (′*a*))
apply *clarsimp*
apply (*case-tac nat*)
apply *clarsimp*
apply (*case-tac n*)
apply *clarsimp*
apply *clarsimp*
apply (*drule word-gt-0* [*THEN iffD1*])
apply (*safe intro*!: *word-eqI*)
apply (*auto simp add*: *nth-2p-bin*)
apply (*erule notE*)
apply (*simp* (*no-asm-use*) *add*: *uint-word-of-int word-size*)
apply (*subst mod-pos-pos-trivial*)
apply *simp*
apply (*rule power-strict-increasing*)
apply *simp-all*
done

lemma *word-of-int-2p*: (*word-of-int* (*2* ˆ *n*) :: ′*a* :: *len word*) = *2* ˆ *n*
apply (*unfold word-arith-power-alt*)
apply (*case-tac len-of TYPE* (′*a*))
apply *clarsimp*

apply (*case-tac nat*)
apply (*rule word-ubin.norm-eq-iff* [*THEN iffD1*])
apply (*rule box-equals*)
apply (*rule-tac* [*2*] *bintr-ariths* (*1*))+
apply *simp*
apply *simp*
done

lemma *bang-is-le*: x !! $m \Longrightarrow 2$ ˆ m <= (x :: $'a$ :: *len word*)
apply (*rule xtr3*)
apply (*rule-tac* [*2*] $y = x$ **in** *le-word-or2*)
apply (*rule word-eqI*)
apply (*auto simp add*: *word-ao-nth nth-w2p word-size*)
done

lemma *word-clr-le*:
fixes *w* :: *'a*::*len0 word*
shows *w* >= *set-bit w n False*
apply (*unfold word-set-bit-def word-le-def word-ubin.eq-norm*)
apply (*rule order-trans*)
apply (*rule bintr-bin-clr-le*)
apply *simp*
done

lemma *word-set-ge*:
fixes *w* :: *'a*::*len word*
shows *w* <= *set-bit w n True*
apply (*unfold word-set-bit-def word-le-def word-ubin.eq-norm*)
apply (*rule order-trans* [*OF - bintr-bin-set-ge*])
apply *simp*
done

127.25 Shifting, Rotating, and Splitting Words

lemma *shiftl1-wi* [*simp*]: *shiftl1* (*word-of-int w*) = *word-of-int* (*w BIT False*)
unfolding *shiftl1-def*
apply (*simp add*: *word-ubin.norm-eq-iff* [*symmetric*] *word-ubin.eq-norm*)
apply (*subst refl* [*THEN bintrunc-BIT-I*, *symmetric*])
apply (*subst bintrunc-bintrunc-min*)
apply *simp*
done

lemma *shiftl1-numeral* [*simp*]:
shiftl1 (*numeral w*) = *numeral* (*Num.Bit0 w*)
unfolding *word-numeral-alt shiftl1-wi* **by** *simp*

lemma *shiftl1-neg-numeral* [*simp*]:
shiftl1 (− *numeral w*) = − *numeral* (*Num.Bit0 w*)
unfolding *word-neg-numeral-alt shiftl1-wi* **by** *simp*

lemma *shiftl1-0* [*simp*] : *shiftl1 0* = *0*
unfolding *shiftl1-def* **by** *simp*

lemma *shiftl1-def-u*: *shiftl1 w* = *word-of-int* (*uint w BIT False*)
by (*simp only*: *shiftl1-def*)

lemma *shiftl1-def-s*: *shiftl1 w* = *word-of-int* (*sint w BIT False*)
unfolding *shiftl1-def Bit-B0 wi-hom-syms* **by** *simp*

lemma *shiftr1-0* [*simp*]: *shiftr1 0* = *0*
unfolding *shiftr1-def* **by** *simp*

lemma *sshiftr1-0* [*simp*]: *sshiftr1 0* = *0*
unfolding *sshiftr1-def* **by** *simp*

lemma *sshiftr1-n1* [*simp*] : *sshiftr1* (− *1*) = − *1*
unfolding *sshiftr1-def* **by** *simp*

lemma *shiftl-0* [*simp*] : (*0*::'*a*::*len0 word*) << *n* = *0*
unfolding *shiftl-def* **by** (*induct n*) *auto*

lemma *shiftr-0* [*simp*] : (*0*::'*a*::*len0 word*) >> *n* = *0*
unfolding *shiftr-def* **by** (*induct n*) *auto*

lemma *sshiftr-0* [*simp*] : *0* >>> *n* = *0*
unfolding *sshiftr-def* **by** (*induct n*) *auto*

lemma *sshiftr-n1* [*simp*] : −*1* >>> *n* = −*1*
unfolding *sshiftr-def* **by** (*induct n*) *auto*

lemma *nth-shiftl1*: *shiftl1 w* !! *n* = (*n* < *size w* & *n* > *0* & *w* !! (*n* − *1*))
apply (*unfold shiftl1-def word-test-bit-def*)
apply (*simp add*: *nth-bintr word-ubin.eq-norm word-size*)
apply (*cases n*)
apply *auto*
done

lemma *nth-shiftl'* [*rule-format*]:
ALL n. ((*w*::'*a*::*len0 word*) << *m*) !! *n* = (*n* < *size w* & *n* >= *m* & *w* !! (*n* − *m*))
apply (*unfold shiftl-def*)
apply (*induct m*)
apply (*force elim*!: *test-bit-size*)
apply (*clarsimp simp add* : *nth-shiftl1 word-size*)
apply *arith*
done

lemmas *nth-shiftl* = *nth-shiftl'* [*unfolded word-size*]

```
lemma nth-shiftr1: shiftr1 w !! n = w !! Suc n
  apply (unfold shiftr1-def word-test-bit-def)
  apply (simp add: nth-bintr word-ubin.eq-norm)
  apply safe
  apply (drule bin-nth.Suc [THEN iffD2, THEN bin-nth-uint-imp])
  apply simp
  done

lemma nth-shiftr:
  ⋀n. ((w::'a::len0 word) >> m) !! n = w !! (n + m)
  apply (unfold shiftr-def)
  apply (induct m)
   apply (auto simp add : nth-shiftr1)
  done

lemma uint-shiftr1: uint (shiftr1 w) = bin-rest (uint w)
  apply (unfold shiftr1-def word-ubin.eq-norm bin-rest-trunc-i)
  apply (subst bintr-uint [symmetric, OF order-refl])
  apply (simp only : bintrunc-bintrunc-l)
  apply simp
  done

lemma nth-sshiftr1:
  sshiftr1 w !! n = (if n = size w − 1 then w !! n else w !! Suc n)
  apply (unfold sshiftr1-def word-test-bit-def)
  apply (simp add: nth-bintr word-ubin.eq-norm
                   bin-nth.Suc [symmetric] word-size
              del: bin-nth.simps)
  apply (simp add: nth-bintr uint-sint del : bin-nth.simps)
  apply (auto simp add: bin-nth-sint)
  done

lemma nth-sshiftr [rule-format] :
  ALL n. sshiftr w m !! n = (n < size w &
    (if n + m >= size w then w !! (size w − 1) else w !! (n + m)))
  apply (unfold sshiftr-def)
  apply (induct-tac m)
   apply (simp add: test-bit-bl)
  apply (clarsimp simp add: nth-sshiftr1 word-size)
  apply safe
       apply arith
      apply arith
     apply (erule thin-rl)
     apply (case-tac n)
      apply safe
      apply simp
```

```
    apply simp
   apply (erule thin-rl)
   apply (case-tac n)
    apply safe
    apply simp
   apply simp
  apply arith+
 done

lemma shiftr1-div-2: uint (shiftr1 w) = uint w div 2
 apply (unfold shiftr1-def bin-rest-def)
 apply (rule word-uint.Abs-inverse)
 apply (simp add: uints-num pos-imp-zdiv-nonneg-iff)
 apply (rule xtr7)
  prefer 2
  apply (rule zdiv-le-dividend)
   apply auto
 done

lemma sshiftr1-div-2: sint (sshiftr1 w) = sint w div 2
 apply (unfold sshiftr1-def bin-rest-def [symmetric])
 apply (simp add: word-sbin.eq-norm)
 apply (rule trans)
  defer
  apply (subst word-sbin.norm-Rep [symmetric])
  apply (rule refl)
 apply (subst word-sbin.norm-Rep [symmetric])
 apply (unfold One-nat-def)
 apply (rule sbintrunc-rest)
 done

lemma shiftr-div-2n: uint (shiftr w n) = uint w div 2 ^ n
 apply (unfold shiftr-def)
 apply (induct n)
  apply simp
 apply (simp add: shiftr1-div-2 mult.commute
                  zdiv-zmult2-eq [symmetric])
 done

lemma sshiftr-div-2n: sint (sshiftr w n) = sint w div 2 ^ n
 apply (unfold sshiftr-def)
 apply (induct n)
  apply simp
 apply (simp add: sshiftr1-div-2 mult.commute
                  zdiv-zmult2-eq [symmetric])
 done
```

127.25.1 shift functions in terms of lists of bools

lemmas *bshiftr1-numeral* [*simp*] =
bshiftr1-def [**where** *w=numeral w*, *unfolded to-bl-numeral*] **for** *w*

lemma *bshiftr1-bl*: *to-bl* (*bshiftr1 b w*) = *b* # *butlast* (*to-bl w*)
unfolding *bshiftr1-def* **by** (*rule word-bl.Abs-inverse*) *simp*

lemma *shiftl1-of-bl*: *shiftl1* (*of-bl bl*) = *of-bl* (*bl* @ [*False*])
by (*simp add*: *of-bl-def bl-to-bin-append*)

lemma *shiftl1-bl*: *shiftl1* (*w*::'*a*::*len0 word*) = *of-bl* (*to-bl w* @ [*False*])
proof −
have *shiftl1 w* = *shiftl1* (*of-bl* (*to-bl w*)) **by** *simp*
also have ... = *of-bl* (*to-bl w* @ [*False*]) **by** (*rule shiftl1-of-bl*)
finally show *?thesis* **.**
qed

lemma *bl-shiftl1*:
to-bl (*shiftl1* (*w* :: '*a* :: *len word*)) = *tl* (*to-bl w*) @ [*False*]
apply (*simp add*: *shiftl1-bl word-rep-drop drop-Suc drop-Cons'*)
apply (*fast intro*!: *Suc-leI*)
done

lemma *bl-shiftl1'*:
to-bl (*shiftl1 w*) = *tl* (*to-bl w* @ [*False*])
unfolding *shiftl1-bl*
by (*simp add*: *word-rep-drop drop-Suc del*: *drop-append*)

lemma *shiftr1-bl*: *shiftr1 w* = *of-bl* (*butlast* (*to-bl w*))
apply (*unfold shiftr1-def uint-bl of-bl-def*)
apply (*simp add*: *butlast-rest-bin word-size*)
apply (*simp add*: *bin-rest-trunc* [*symmetric*, *unfolded One-nat-def*])
done

lemma *bl-shiftr1*:
to-bl (*shiftr1* (*w* :: '*a* :: *len word*)) = *False* # *butlast* (*to-bl w*)
unfolding *shiftr1-bl*
by (*simp add* : *word-rep-drop len-gt-0* [*THEN Suc-leI*])

lemma *bl-shiftr1'*:
to-bl (*shiftr1 w*) = *butlast* (*False* # *to-bl w*)
apply (*rule word-bl.Abs-inverse'*)
apply (*simp del*: *butlast.simps*)
apply (*simp add*: *shiftr1-bl of-bl-def*)
done

lemma *shiftl1-rev*:

shiftl1 w = word-reverse (shiftr1 (word-reverse w))
apply (*unfold word-reverse-def*)
apply (*rule word-bl.Rep-inverse′ [symmetric]*)
apply (*simp add: bl-shiftl1′ bl-shiftr1′ word-bl.Abs-inverse*)
apply (*cases to-bl w*)
apply *auto*
done

lemma *shiftl-rev*:
shiftl w n = word-reverse (shiftr (word-reverse w) n)
apply (*unfold shiftl-def shiftr-def*)
apply (*induct n*)
apply (*auto simp add : shiftl1-rev*)
done

lemma *rev-shiftl*: *word-reverse w << n = word-reverse (w >> n)*
by (*simp add: shiftl-rev*)

lemma *shiftr-rev*: *w >> n = word-reverse (word-reverse w << n)*
by (*simp add: rev-shiftl*)

lemma *rev-shiftr*: *word-reverse w >> n = word-reverse (w << n)*
by (*simp add: shiftr-rev*)

lemma *bl-sshiftr1*:
to-bl (sshiftr1 (w :: ′a :: len word)) = hd (to-bl w) # butlast (to-bl w)
apply (*unfold sshiftr1-def uint-bl word-size*)
apply (*simp add: butlast-rest-bin word-ubin.eq-norm*)
apply (*simp add: sint-uint*)
apply (*rule nth-equalityI*)
apply *clarsimp*
apply *clarsimp*
apply (*case-tac i*)
apply (*simp-all add: hd-conv-nth length-0-conv [symmetric]*
nth-bin-to-bl bin-nth.Suc [symmetric]
nth-sbintr
del: bin-nth.Suc)
apply *force*
apply (*rule impI*)
apply (*rule-tac f = bin-nth (uint w)* **in** *arg-cong*)
apply *simp*
done

lemma *drop-shiftr*:
drop n (to-bl ((w :: ′a :: len word) >> n)) = take (size w − n) (to-bl w)
apply (*unfold shiftr-def*)
apply (*induct n*)
prefer *2*
apply (*simp add: drop-Suc bl-shiftr1 butlast-drop [symmetric]*)

apply (*rule butlast-take* [*THEN trans*])
apply (*auto simp*: *word-size*)
done

lemma *drop-sshiftr*:
drop n (*to-bl* ((*w* :: *'a* :: *len word*) >>> *n*)) = *take* (*size w* − *n*) (*to-bl w*)
apply (*unfold sshiftr-def*)
apply (*induct n*)
prefer *2*
apply (*simp add*: *drop-Suc bl-sshiftr1 butlast-drop* [*symmetric*])
apply (*rule butlast-take* [*THEN trans*])
apply (*auto simp*: *word-size*)
done

lemma *take-shiftr*:
$n \leq$ *size w* ⟹ *take n* (*to-bl* (*w* >> *n*)) = *replicate n False*
apply (*unfold shiftr-def*)
apply (*induct n*)
prefer *2*
apply (*simp add*: *bl-shiftr1' length-0-conv* [*symmetric*] *word-size*)
apply (*rule take-butlast* [*THEN trans*])
apply (*auto simp*: *word-size*)
done

lemma *take-sshiftr'* [*rule-format*] :
n <= *size* (*w* :: *'a* :: *len word*) −−> *hd* (*to-bl* (*w* >>> *n*)) = *hd* (*to-bl w*) &
take n (*to-bl* (*w* >>> *n*)) = *replicate n* (*hd* (*to-bl w*))
apply (*unfold sshiftr-def*)
apply (*induct n*)
prefer *2*
apply (*simp add*: *bl-sshiftr1*)
apply (*rule impI*)
apply (*rule take-butlast* [*THEN trans*])
apply (*auto simp*: *word-size*)
done

lemmas *hd-sshiftr* = *take-sshiftr'* [*THEN conjunct1*]
lemmas *take-sshiftr* = *take-sshiftr'* [*THEN conjunct2*]

lemma *atd-lem*: *take n xs* = *t* ⟹ *drop n xs* = *d* ⟹ *xs* = *t* @ *d*
by (*auto intro*: *append-take-drop-id* [*symmetric*])

lemmas *bl-shiftr* = *atd-lem* [*OF take-shiftr drop-shiftr*]
lemmas *bl-sshiftr* = *atd-lem* [*OF take-sshiftr drop-sshiftr*]

lemma *shiftl-of-bl*: *of-bl bl* << *n* = *of-bl* (*bl* @ *replicate n False*)
unfolding *shiftl-def*
by (*induct n*) (*auto simp*: *shiftl1-of-bl replicate-app-Cons-same*)

lemma *shiftl-bl*:
  (*w*::′*a*::*len0 word*) << (*n*::*nat*) = *of-bl* (*to-bl w* @ *replicate n False*)
proof −
  have *w* << *n* = *of-bl* (*to-bl w*) << *n* **by** *simp*
  also have ... = *of-bl* (*to-bl w* @ *replicate n False*) **by** (*rule shiftl-of-bl*)
  finally show *?thesis* **.**
qed

lemmas *shiftl-numeral* [*simp*] = *shiftl-def* [**where** *w*=*numeral w*] **for** *w*

lemma *bl-shiftl*:
  to-bl (*w* << *n*) = *drop n* (*to-bl w*) @ *replicate* (*min* (*size w*) *n*) *False*
  by (*simp add*: *shiftl-bl word-rep-drop word-size*)

lemma *shiftl-zero-size*:
  fixes *x* :: ′*a*::*len0 word*
  shows *size x* <= *n* ⟹ *x* << *n* = *0*
  apply (*unfold word-size*)
  apply (*rule word-eqI*)
  apply (*clarsimp simp add*: *shiftl-bl word-size test-bit-of-bl nth-append*)
  done

lemma *shiftl1-2t*: *shiftl1* (*w* :: ′*a* :: *len word*) = *2* ∗ *w*
  by (*simp add*: *shiftl1-def Bit-def wi-hom-mult* [*symmetric*])

lemma *shiftl1-p*: *shiftl1* (*w* :: ′*a* :: *len word*) = *w* + *w*
  by (*simp add*: *shiftl1-2t*)

lemma *shiftl-t2n*: *shiftl* (*w* :: ′*a* :: *len word*) *n* = *2* ˆ *n* ∗ *w*
  unfolding *shiftl-def*
  by (*induct n*) (*auto simp*: *shiftl1-2t*)

lemma *shiftr1-bintr* [*simp*]:
  (*shiftr1* (*numeral w*) :: ′*a* :: *len0 word*) =
    word-of-int (*bin-rest* (*bintrunc* (*len-of TYPE* (′*a*)) (*numeral w*)))
  unfolding *shiftr1-def word-numeral-alt*
  by (*simp add*: *word-ubin.eq-norm*)

lemma *sshiftr1-sbintr* [*simp*]:
  (*sshiftr1* (*numeral w*) :: ′*a* :: *len word*) =
    word-of-int (*bin-rest* (*sbintrunc* (*len-of TYPE* (′*a*) − *1*) (*numeral w*)))
  unfolding *sshiftr1-def word-numeral-alt*
  by (*simp add*: *word-sbin.eq-norm*)

lemma *shiftr-no* [*simp*]:

  (*numeral w*::′*a*::*len0 word*) >> *n* = *word-of-int*

((*bin-rest* ^^ *n*) (*bintrunc* (*len-of TYPE*('*a*)) (*numeral w*)))
apply (*rule word-eqI*)
apply (*auto simp*: *nth-shiftr nth-rest-power-bin nth-bintr word-size*)
done

lemma *sshiftr-no* [*simp*]:

(*numeral w*::'*a*::*len word*) >>> *n* = *word-of-int*
((*bin-rest* ^^ *n*) (*sbintrunc* (*len-of TYPE*('*a*) − *1*) (*numeral w*)))
apply (*rule word-eqI*)
apply (*auto simp*: *nth-sshiftr nth-rest-power-bin nth-sbintr word-size*)
apply (*subgoal-tac na* + *n* = *len-of TYPE*('*a*) − *Suc 0*, *simp*, *simp*)+
done

lemma *shiftr1-bl-of*:
length bl ≤ *len-of TYPE*('*a*) ⟹
shiftr1 (*of-bl bl*::'*a*::*len0 word*) = *of-bl* (*butlast bl*)
by (*clarsimp simp*: *shiftr1-def of-bl-def butlast-rest-bl2bin*
word-ubin.eq-norm trunc-bl2bin)

lemma *shiftr-bl-of*:
length bl ≤ *len-of TYPE*('*a*) ⟹
(*of-bl bl*::'*a*::*len0 word*) >> *n* = *of-bl* (*take* (*length bl* − *n*) *bl*)
apply (*unfold shiftr-def*)
apply (*induct n*)
apply *clarsimp*
apply *clarsimp*
apply (*subst shiftr1-bl-of*)
apply *simp*
apply (*simp add*: *butlast-take*)
done

lemma *shiftr-bl*:
(*x*::'*a*::*len0 word*) >> *n* ≡ *of-bl* (*take* (*len-of TYPE*('*a*) − *n*) (*to-bl x*))
using *shiftr-bl-of* [**where** '*a*='*a*, *of to-bl x*] **by** *simp*

lemma *msb-shift*:
msb (*w*::'*a*::*len word*) ⟷ (*w* >> (*len-of TYPE*('*a*) − *1*)) ≠ *0*
apply (*unfold shiftr-bl word-msb-alt*)
apply (*simp add*: *word-size Suc-le-eq take-Suc*)
apply (*cases hd* (*to-bl w*))
apply (*auto simp*: *word-1-bl*
of-bl-rep-False [**where** *n*=*1* **and** *bs*=[], *simplified*])
done

lemma *zip-replicate*:
n ≥ *length ys* ⟹ *zip* (*replicate n x*) *ys* = *map* (λ*y*. (*x*, *y*)) *ys*
apply (*induct ys arbitrary*: *n*, *simp-all*)
apply (*case-tac n*, *simp-all*)

done

lemma *align-lem-or* [*rule-format*] :
ALL x m. length x = n + m --> length y = n + m -->
drop m x = replicate n False --> take m y = replicate m False -->
map2 op | x y = take m x @ drop m y
apply (*induct-tac y*)
apply *force*
apply *clarsimp*
apply (*case-tac x*, *force*)
apply (*case-tac m*, *auto*)
apply (*drule-tac t=length xs* **for** *xs* **in** *sym*)
apply (*clarsimp simp*: *map2-def zip-replicate o-def*)
done

lemma *align-lem-and* [*rule-format*] :
ALL x m. length x = n + m --> length y = n + m -->
drop m x = replicate n False --> take m y = replicate m False -->
map2 op & x y = replicate (n + m) False
apply (*induct-tac y*)
apply *force*
apply *clarsimp*
apply (*case-tac x*, *force*)
apply (*case-tac m*, *auto*)
apply (*drule-tac t=length xs* **for** *xs* **in** *sym*)
apply (*clarsimp simp*: *map2-def zip-replicate o-def map-replicate-const*)
done

lemma *aligned-bl-add-size* [*OF refl*]:
size x − n = m ⟹ n <= size x ⟹ drop m (to-bl x) = replicate n False ⟹
take m (to-bl y) = replicate m False ⟹
to-bl (x + y) = take m (to-bl x) @ drop m (to-bl y)
apply (*subgoal-tac x AND y = 0*)
prefer *2*
apply (*rule word-bl.Rep-eqD*)
apply (*simp add*: *bl-word-and*)
apply (*rule align-lem-and* [*THEN trans*])
apply (*simp-all add*: *word-size*)[*5*]
apply *simp*
apply (*subst word-plus-and-or* [*symmetric*])
apply (*simp add* : *bl-word-or*)
apply (*rule align-lem-or*)
apply (*simp-all add*: *word-size*)
done

127.25.2 Mask

lemma *nth-mask* [*OF refl*, *simp*]:
m = mask n ⟹ test-bit m i = (i < n & i < size m)

apply (*unfold mask-def test-bit-bl*)
apply (*simp only*: *word-1-bl* [*symmetric*] *shiftl-of-bl*)
apply (*clarsimp simp add*: *word-size*)
apply (*simp only*: *of-bl-def mask-lem word-of-int-hom-syms add-diff-cancel2*)
apply (*fold of-bl-def*)
apply (*simp add*: *word-1-bl*)
apply (*rule test-bit-of-bl* [*THEN trans*, *unfolded test-bit-bl word-size*])
apply *auto*
done

lemma *mask-bl*: *mask n* = *of-bl* (*replicate n True*)
by (*auto simp add* : *test-bit-of-bl word-size intro*: *word-eqI*)

lemma *mask-bin*: *mask n* = *word-of-int* (*bintrunc n* (− *1*))
by (*auto simp add*: *nth-bintr word-size intro*: *word-eqI*)

lemma *and-mask-bintr*: *w AND mask n* = *word-of-int* (*bintrunc n* (*uint w*))
apply (*rule word-eqI*)
apply (*simp add*: *nth-bintr word-size word-ops-nth-size*)
apply (*auto simp add*: *test-bit-bin*)
done

lemma *and-mask-wi*: *word-of-int i AND mask n* = *word-of-int* (*bintrunc n i*)
by (*auto simp add*: *nth-bintr word-size word-ops-nth-size word-eq-iff*)

lemma *and-mask-no*: *numeral i AND mask n* = *word-of-int* (*bintrunc n* (*numeral i*))
unfolding *word-numeral-alt* **by** (*rule and-mask-wi*)

lemma *bl-and-mask′*:
to-bl (*w AND mask n* :: ′*a* :: *len word*) =
replicate (*len-of TYPE*(′*a*) − *n*) *False* @
drop (*len-of TYPE*(′*a*) − *n*) (*to-bl w*)
apply (*rule nth-equalityI*)
apply *simp*
apply (*clarsimp simp add*: *to-bl-nth word-size*)
apply (*simp add*: *word-size word-ops-nth-size*)
apply (*auto simp add*: *word-size test-bit-bl nth-append nth-rev*)
done

lemma *and-mask-mod-2p*: *w AND mask n* = *word-of-int* (*uint w mod 2* ˆ *n*)
by (*simp only*: *and-mask-bintr bintrunc-mod2p*)

lemma *and-mask-lt-2p*: *uint* (*w AND mask n*) < *2* ˆ *n*
apply (*simp add*: *and-mask-bintr word-ubin.eq-norm*)
apply (*simp add*: *bintrunc-mod2p*)
apply (*rule xtr8*)
prefer *2*
apply (*rule pos-mod-bound*)

apply *auto*
done

lemma *eq-mod-iff*: $0 < (n::int) \Longrightarrow b = b \bmod n \longleftrightarrow 0 \leq b \wedge b < n$
by (*simp add*: *int-mod-lem eq-sym-conv*)

lemma *mask-eq-iff*: $(w\ AND\ mask\ n) = w \longleftrightarrow uint\ w < 2\ \hat{}\ n$
apply (*simp add*: *and-mask-bintr*)
apply (*simp add*: *word-ubin.inverse-norm*)
apply (*simp add*: *eq-mod-iff bintrunc-mod2p min-def*)
apply (*fast intro*!: *lt2p-lem*)
done

lemma *and-mask-dvd*: $2\ \hat{}\ n\ dvd\ uint\ w = (w\ AND\ mask\ n = 0)$
apply (*simp add*: *dvd-eq-mod-eq-0 and-mask-mod-2p*)
apply (*simp add*: *word-uint.norm-eq-iff* [*symmetric*] *word-of-int-homs*
del: *word-of-int-0*)
apply (*subst word-uint.norm-Rep* [*symmetric*])
apply (*simp only*: *bintrunc-bintrunc-min bintrunc-mod2p* [*symmetric*] *min-def*)
apply *auto*
done

lemma *and-mask-dvd-nat*: $2\ \hat{}\ n\ dvd\ unat\ w = (w\ AND\ mask\ n = 0)$
apply (*unfold unat-def*)
apply (*rule trans* [*OF - and-mask-dvd*])
apply (*unfold dvd-def*)
apply *auto*
apply (*drule uint-ge-0* [*THEN nat-int.Abs-inverse′* [*simplified*], *symmetric*])
apply (*simp add* : *of-nat-mult of-nat-power*)
apply (*simp add* : *nat-mult-distrib nat-power-eq*)
done

lemma *word-2p-lem*:
$n < size\ w \Longrightarrow w < 2\ \hat{}\ n = (uint\ (w :: 'a :: len\ word) < 2\ \hat{}\ n)$
apply (*unfold word-size word-less-alt word-numeral-alt*)
apply (*clarsimp simp add*: *word-of-int-power-hom word-uint.eq-norm*
mod-pos-pos-trivial
simp del: *word-of-int-numeral*)
done

lemma *less-mask-eq*: $x < 2\ \hat{}\ n \Longrightarrow x\ AND\ mask\ n = (x :: 'a :: len\ word)$
apply (*unfold word-less-alt word-numeral-alt*)
apply (*clarsimp simp add*: *and-mask-mod-2p word-of-int-power-hom*
word-uint.eq-norm
simp del: *word-of-int-numeral*)
apply (*drule xtr8* [*rotated*])
apply (*rule int-mod-le*)
apply (*auto simp add* : *mod-pos-pos-trivial*)
done

lemmas *mask-eq-iff-w2p* = *trans* [*OF mask-eq-iff word-2p-lem* [*symmetric*]]

lemmas *and-mask-less'* = *iffD2* [*OF word-2p-lem and-mask-lt-2p*, *simplified word-size*]

lemma *and-mask-less-size*: $n < size\ x \Longrightarrow x\ AND\ mask\ n < 2\hat{}n$
  unfolding *word-size* **by** (*erule and-mask-less'*)

lemma *word-mod-2p-is-mask* [*OF refl*]:
  $c = 2\ \hat{}\ n \Longrightarrow c > 0 \Longrightarrow x\ mod\ c = (x :: 'a :: len\ word)\ AND\ mask\ n$
  by (*clarsimp simp add*: *word-mod-def uint-2p and-mask-mod-2p*)

lemma *mask-eqs*:
  $(a\ AND\ mask\ n) + b\ AND\ mask\ n = a + b\ AND\ mask\ n$
  $a + (b\ AND\ mask\ n)\ AND\ mask\ n = a + b\ AND\ mask\ n$
  $(a\ AND\ mask\ n) - b\ AND\ mask\ n = a - b\ AND\ mask\ n$
  $a - (b\ AND\ mask\ n)\ AND\ mask\ n = a - b\ AND\ mask\ n$
  $a * (b\ AND\ mask\ n)\ AND\ mask\ n = a * b\ AND\ mask\ n$
  $(b\ AND\ mask\ n) * a\ AND\ mask\ n = b * a\ AND\ mask\ n$
  $(a\ AND\ mask\ n) + (b\ AND\ mask\ n)\ AND\ mask\ n = a + b\ AND\ mask\ n$
  $(a\ AND\ mask\ n) - (b\ AND\ mask\ n)\ AND\ mask\ n = a - b\ AND\ mask\ n$
  $(a\ AND\ mask\ n) * (b\ AND\ mask\ n)\ AND\ mask\ n = a * b\ AND\ mask\ n$
  $-\ (a\ AND\ mask\ n)\ AND\ mask\ n = -\ a\ AND\ mask\ n$
  $word\text{-}succ\ (a\ AND\ mask\ n)\ AND\ mask\ n = word\text{-}succ\ a\ AND\ mask\ n$
  $word\text{-}pred\ (a\ AND\ mask\ n)\ AND\ mask\ n = word\text{-}pred\ a\ AND\ mask\ n$
  using *word-of-int-Ex* [**where** *x*=*a*] *word-of-int-Ex* [**where** *x*=*b*]
  by (*auto simp*: *and-mask-wi bintr-ariths bintr-arith1s word-of-int-homs*)

lemma *mask-power-eq*:
  $(x\ AND\ mask\ n)\ \hat{}\ k\ AND\ mask\ n = x\ \hat{}\ k\ AND\ mask\ n$
  using *word-of-int-Ex* [**where** *x*=*x*]
  by (*clarsimp simp*: *and-mask-wi word-of-int-power-hom bintr-ariths*)

127.25.3 Revcast

lemmas *revcast-def'* = *revcast-def* [*simplified*]
lemmas *revcast-def''* = *revcast-def'* [*simplified word-size*]
lemmas *revcast-no-def* [*simp*] = *revcast-def'* [**where** *w*=*numeral w*, *unfolded word-size*]
for *w*

lemma *to-bl-revcast*:
  $to\text{-}bl\ (revcast\ w :: 'a :: len0\ word) =$
   $takefill\ False\ (len\text{-}of\ TYPE\ ('a))\ (to\text{-}bl\ w)$
  apply (*unfold revcast-def' word-size*)
  apply (*rule word-bl.Abs-inverse*)
  apply *simp*
  done

lemma *revcast-rev-ucast* [*OF refl refl refl*]:

cs = [*rc*, *uc*] ⟹ *rc* = *revcast* (*word-reverse w*) ⟹ *uc* = *ucast w* ⟹
rc = *word-reverse uc*
apply (*unfold ucast-def revcast-def' Let-def word-reverse-def*)
apply (*clarsimp simp add* : *to-bl-of-bin takefill-bintrunc*)
apply (*simp add* : *word-bl.Abs-inverse word-size*)
done

lemma *revcast-ucast*: *revcast w* = *word-reverse* (*ucast* (*word-reverse w*))
using *revcast-rev-ucast* [*of word-reverse w*] **by** *simp*

lemma *ucast-revcast*: *ucast w* = *word-reverse* (*revcast* (*word-reverse w*))
by (*fact revcast-rev-ucast* [*THEN word-rev-gal'*])

lemma *ucast-rev-revcast*: *ucast* (*word-reverse w*) = *word-reverse* (*revcast w*)
by (*fact revcast-ucast* [*THEN word-rev-gal'*])

— linking revcast and cast via shift

lemmas *wsst-TYs* = *source-size target-size word-size*

lemma *revcast-down-uu* [*OF refl*]:
rc = *revcast* ⟹ *source-size rc* = *target-size rc* + *n* ⟹
rc (*w* :: *'a* :: *len word*) = *ucast* (*w* >> *n*)
apply (*simp add*: *revcast-def'*)
apply (*rule word-bl.Rep-inverse'*)
apply (*rule trans*, *rule ucast-down-drop*)
prefer *2*
apply (*rule trans*, *rule drop-shiftr*)
apply (*auto simp*: *takefill-alt wsst-TYs*)
done

lemma *revcast-down-us* [*OF refl*]:
rc = *revcast* ⟹ *source-size rc* = *target-size rc* + *n* ⟹
rc (*w* :: *'a* :: *len word*) = *ucast* (*w* >>> *n*)
apply (*simp add*: *revcast-def'*)
apply (*rule word-bl.Rep-inverse'*)
apply (*rule trans*, *rule ucast-down-drop*)
prefer *2*
apply (*rule trans*, *rule drop-sshiftr*)
apply (*auto simp*: *takefill-alt wsst-TYs*)
done

lemma *revcast-down-su* [*OF refl*]:
rc = *revcast* ⟹ *source-size rc* = *target-size rc* + *n* ⟹
rc (*w* :: *'a* :: *len word*) = *scast* (*w* >> *n*)
apply (*simp add*: *revcast-def'*)
apply (*rule word-bl.Rep-inverse'*)
apply (*rule trans*, *rule scast-down-drop*)

prefer *2*
apply (*rule trans*, *rule drop-shiftr*)
apply (*auto simp*: *takefill-alt wsst-TYs*)
done

lemma *revcast-down-ss* [*OF refl*]:
rc = *revcast* ⟹ *source-size rc* = *target-size rc* + *n* ⟹
rc (*w* :: *'a* :: *len word*) = *scast* (*w* >>> *n*)
apply (*simp add*: *revcast-def'*)
apply (*rule word-bl.Rep-inverse'*)
apply (*rule trans*, *rule scast-down-drop*)
prefer *2*
apply (*rule trans*, *rule drop-sshiftr*)
apply (*auto simp*: *takefill-alt wsst-TYs*)
done

lemma *cast-down-rev*:
uc = *ucast* ⟹ *source-size uc* = *target-size uc* + *n* ⟹
uc w = *revcast* ((*w* :: *'a* :: *len word*) << *n*)
apply (*unfold shiftl-rev*)
apply *clarify*
apply (*simp add*: *revcast-rev-ucast*)
apply (*rule word-rev-gal'*)
apply (*rule trans* [*OF - revcast-rev-ucast*])
apply (*rule revcast-down-uu* [*symmetric*])
apply (*auto simp add*: *wsst-TYs*)
done

lemma *revcast-up* [*OF refl*]:
rc = *revcast* ⟹ *source-size rc* + *n* = *target-size rc* ⟹
rc w = (*ucast w* :: *'a* :: *len word*) << *n*
apply (*simp add*: *revcast-def'*)
apply (*rule word-bl.Rep-inverse'*)
apply (*simp add*: *takefill-alt*)
apply (*rule bl-shiftl* [*THEN trans*])
apply (*subst ucast-up-app*)
apply (*auto simp add*: *wsst-TYs*)
done

lemmas *rc1* = *revcast-up* [*THEN*
revcast-rev-ucast [*symmetric*, *THEN trans*, *THEN word-rev-gal*, *symmetric*]]
lemmas *rc2* = *revcast-down-uu* [*THEN*
revcast-rev-ucast [*symmetric*, *THEN trans*, *THEN word-rev-gal*, *symmetric*]]

lemmas *ucast-up* =
rc1 [*simplified rev-shiftr* [*symmetric*] *revcast-ucast* [*symmetric*]]
lemmas *ucast-down* =
rc2 [*simplified rev-shiftr revcast-ucast* [*symmetric*]]

127.25.4 Slices

lemma *slice1-no-bin* [*simp*]:
slice1 n (*numeral w* :: *'b word*) = *of-bl* (*takefill False n* (*bin-to-bl* (*len-of TYPE*(*'b* :: *len0*)) (*numeral w*)))
by (*simp add*: *slice1-def*)

lemma *slice-no-bin* [*simp*]:
slice n (*numeral w* :: *'b word*) = *of-bl* (*takefill False* (*len-of TYPE*(*'b* :: *len0*) − *n*)
(*bin-to-bl* (*len-of TYPE*(*'b* :: *len0*)) (*numeral w*)))
by (*simp add*: *slice-def word-size*)

lemma *slice1-0* [*simp*] : *slice1 n 0* = *0*
unfolding *slice1-def* **by** *simp*

lemma *slice-0* [*simp*] : *slice n 0* = *0*
unfolding *slice-def* **by** *auto*

lemma *slice-take'*: *slice n w* = *of-bl* (*take* (*size w* − *n*) (*to-bl w*))
unfolding *slice-def' slice1-def*
by (*simp add* : *takefill-alt word-size*)

lemmas *slice-take* = *slice-take'* [*unfolded word-size*]

— shiftr to a word of the same size is just slice, slice is just shiftr then ucast
lemmas *shiftr-slice* = *trans* [*OF shiftr-bl* [*THEN meta-eq-to-obj-eq*] *slice-take* [*symmetric*]]

lemma *slice-shiftr*: *slice n w* = *ucast* (*w* >> *n*)
apply (*unfold slice-take shiftr-bl*)
apply (*rule ucast-of-bl-up* [*symmetric*])
apply (*simp add*: *word-size*)
done

lemma *nth-slice*:
(*slice n w* :: *'a* :: *len0 word*) !! *m* =
(*w* !! (*m* + *n*) & *m* < *len-of TYPE* (*'a*))
unfolding *slice-shiftr*
by (*simp add* : *nth-ucast nth-shiftr*)

lemma *slice1-down-alt'*:
sl = *slice1 n w* ⟹ *fs* = *size sl* ⟹ *fs* + *k* = *n* ⟹
to-bl sl = *takefill False fs* (*drop k* (*to-bl w*))
unfolding *slice1-def word-size of-bl-def uint-bl*
by (*clarsimp simp*: *word-ubin.eq-norm bl-bin-bl-rep-drop drop-takefill*)

lemma *slice1-up-alt'*:
sl = *slice1 n w* ⟹ *fs* = *size sl* ⟹ *fs* = *n* + *k* ⟹
to-bl sl = *takefill False fs* (*replicate k False* @ (*to-bl w*))
apply (*unfold slice1-def word-size of-bl-def uint-bl*)

```
  apply (clarsimp simp: word-ubin.eq-norm bl-bin-bl-rep-drop
                   takefill-append [symmetric])
  apply (rule-tac f = %k. takefill False (len-of TYPE('a))
    (replicate k False @ bin-to-bl (len-of TYPE('b)) (uint w)) in arg-cong)
  apply arith
  done

lemmas sd1 = slice1-down-alt' [OF refl refl, unfolded word-size]
lemmas su1 = slice1-up-alt' [OF refl refl, unfolded word-size]
lemmas slice1-down-alt = le-add-diff-inverse [THEN sd1]
lemmas slice1-up-alts =
  le-add-diff-inverse [symmetric, THEN su1]
  le-add-diff-inverse2 [symmetric, THEN su1]

lemma ucast-slice1: ucast w = slice1 (size w) w
  unfolding slice1-def ucast-bl
  by (simp add : takefill-same' word-size)

lemma ucast-slice: ucast w = slice 0 w
  unfolding slice-def by (simp add : ucast-slice1)

lemma slice-id: slice 0 t = t
  by (simp only: ucast-slice [symmetric] ucast-id)

lemma revcast-slice1 [OF refl]:
  rc = revcast w ==> slice1 (size rc) w = rc
  unfolding slice1-def revcast-def' by (simp add : word-size)

lemma slice1-tf-tf':
  to-bl (slice1 n w :: 'a :: len0 word) =
   rev (takefill False (len-of TYPE('a)) (rev (takefill False n (to-bl w))))
  unfolding slice1-def by (rule word-rev-tf)

lemmas slice1-tf-tf = slice1-tf-tf' [THEN word-bl.Rep-inverse', symmetric]

lemma rev-slice1:
  n + k = len-of TYPE('a) + len-of TYPE('b) ==>
  slice1 n (word-reverse w :: 'b :: len0 word) =
  word-reverse (slice1 k w :: 'a :: len0 word)
  apply (unfold word-reverse-def slice1-tf-tf)
  apply (rule word-bl.Rep-inverse')
  apply (rule rev-swap [THEN iffD1])
  apply (rule trans [symmetric])
  apply (rule tf-rev)
   apply (simp add: word-bl.Abs-inverse)
  apply (simp add: word-bl.Abs-inverse)
  done

lemma rev-slice:
```

$n + k + len\text{-}of\ TYPE('a::len0) = len\text{-}of\ TYPE('b::len0) \Longrightarrow$
  $slice\ n\ (word\text{-}reverse\ (w::'b\ word)) = word\text{-}reverse\ (slice\ k\ w::'a\ word)$
  apply (*unfold slice-def word-size*)
  apply (*rule rev-slice1*)
  apply *arith*
  done

lemmas *sym-notr* =
  not-iff [*THEN iffD2*, *THEN not-sym*, *THEN not-iff* [*THEN iffD1*]]

— problem posed by TPHOLs referee: criterion for overflow of addition of signed integers

lemma *sofl-test*:
  $(sint\ (x :: 'a :: len\ word) + sint\ y = sint\ (x + y)) =$
    $((((x+y)\ XOR\ x)\ AND\ ((x+y)\ XOR\ y)) >> (size\ x - 1) = 0)$
  apply (*unfold word-size*)
  apply (*cases len-of TYPE('a)*, *simp*)
  apply (*subst msb-shift* [*THEN sym-notr*])
  apply (*simp add*: *word-ops-msb*)
  apply (*simp add*: *word-msb-sint*)
  apply *safe*
      apply *simp-all*
    apply (*unfold sint-word-ariths*)
    apply (*unfold word-sbin.set-iff-norm* [*symmetric*] *sints-num*)
    apply *safe*
      apply (*insert sint-range'* [**where** *x=x*])
      apply (*insert sint-range'* [**where** *x=y*])
      defer
      apply (*simp* (*no-asm*), *arith*)
     apply (*simp* (*no-asm*), *arith*)
    defer
    defer
    apply (*simp* (*no-asm*), *arith*)
   apply (*simp* (*no-asm*), *arith*)
  apply (*rule notI* [*THEN notnotD*],
      drule leI not-le-imp-less,
      drule sbintrunc-inc sbintrunc-dec,
      simp)+
  done

127.26 Split and cat

lemmas *word-split-bin'* = *word-split-def*
lemmas *word-cat-bin'* = *word-cat-def*

lemma *word-rsplit-no*:
  (*word-rsplit* (*numeral bin* :: *'b* :: *len0 word*) :: *'a word list*) =
    map word-of-int (*bin-rsplit* (*len-of TYPE*(*'a* :: *len*))

(*len-of TYPE('b)*, *bintrunc* (*len-of TYPE('b)*) (*numeral bin*)))
unfolding *word-rsplit-def* **by** (*simp add*: *word-ubin.eq-norm*)

lemmas *word-rsplit-no-cl* [*simp*] = *word-rsplit-no*
[*unfolded bin-rsplitl-def bin-rsplit-l* [*symmetric*]]

lemma *test-bit-cat*:
wc = *word-cat a b* ⟹ *wc* !! *n* = (*n* < *size wc* &
(*if n* < *size b then b* !! *n else a* !! (*n* − *size b*)))
apply (*unfold word-cat-bin' test-bit-bin*)
apply (*auto simp add* : *word-ubin.eq-norm nth-bintr bin-nth-cat word-size*)
apply (*erule bin-nth-uint-imp*)
done

lemma *word-cat-bl*: *word-cat a b* = *of-bl* (*to-bl a* @ *to-bl b*)
apply (*unfold of-bl-def to-bl-def word-cat-bin'*)
apply (*simp add*: *bl-to-bin-app-cat*)
done

lemma *of-bl-append*:
(*of-bl* (*xs* @ *ys*) :: *'a* :: *len word*) = *of-bl xs* ∗ *2*^(*length ys*) + *of-bl ys*
apply (*unfold of-bl-def*)
apply (*simp add*: *bl-to-bin-app-cat bin-cat-num*)
apply (*simp add*: *word-of-int-power-hom* [*symmetric*] *word-of-int-hom-syms*)
done

lemma *of-bl-False* [*simp*]:
of-bl (*False#xs*) = *of-bl xs*
by (*rule word-eqI*)
(*auto simp add*: *test-bit-of-bl nth-append*)

lemma *of-bl-True* [*simp*]:
(*of-bl* (*True#xs*)::*'a*::*len word*) = *2*^*length xs* + *of-bl xs*
by (*subst of-bl-append* [**where** *xs*=[*True*], *simplified*])
(*simp add*: *word-1-bl*)

lemma *of-bl-Cons*:
of-bl (*x#xs*) = *of-bool x* ∗ *2*^*length xs* + *of-bl xs*
by (*cases x*) *simp-all*

lemma *split-uint-lem*: *bin-split n* (*uint* (*w* :: *'a* :: *len0 word*)) = (*a*, *b*) ⟹
a = *bintrunc* (*len-of TYPE('a)* − *n*) *a* & *b* = *bintrunc* (*len-of TYPE('a)*) *b*
apply (*frule word-ubin.norm-Rep* [*THEN ssubst*])
apply (*drule bin-split-trunc1*)
apply (*drule sym* [*THEN trans*])
apply *assumption*
apply *safe*
done

lemma *word-split-bl'*:
std = *size c* − *size b* ⟹ (*word-split c* = (*a*, *b*)) ⟹
(*a* = *of-bl* (*take std* (*to-bl c*)) & *b* = *of-bl* (*drop std* (*to-bl c*)))
apply (*unfold word-split-bin'*)
apply *safe*
defer
apply (*clarsimp split*: *prod.splits*)
apply *hypsubst-thin*
apply (*drule word-ubin.norm-Rep* [*THEN ssubst*])
apply (*drule split-bintrunc*)
apply (*simp add* : *of-bl-def bl2bin-drop word-size*
word-ubin.norm-eq-iff [*symmetric*] *min-def del* : *word-ubin.norm-Rep*)
apply (*clarsimp split*: *prod.splits*)
apply (*frule split-uint-lem* [*THEN conjunct1*])
apply (*unfold word-size*)
apply (*cases len-of TYPE*('*a*) >= *len-of TYPE*('*b*))
defer
apply *simp*
apply (*simp add* : *of-bl-def to-bl-def*)
apply (*subst bin-split-take1* [*symmetric*])
prefer *2*
apply *assumption*
apply *simp*
apply (*erule thin-rl*)
apply (*erule arg-cong* [*THEN trans*])
apply (*simp add* : *word-ubin.norm-eq-iff* [*symmetric*])
done

lemma *word-split-bl*: *std* = *size c* − *size b* ⟹
(*a* = *of-bl* (*take std* (*to-bl c*)) & *b* = *of-bl* (*drop std* (*to-bl c*))) ⟷
word-split c = (*a*, *b*)
apply (*rule iffI*)
defer
apply (*erule* (*1*) *word-split-bl'*)
apply (*case-tac word-split c*)
apply (*auto simp add* : *word-size*)
apply (*frule word-split-bl'* [*rotated*])
apply (*auto simp add* : *word-size*)
done

lemma *word-split-bl-eq*:
(*word-split* (*c*::'*a*::*len word*) :: ('*c* :: *len0 word* ∗ '*d* :: *len0 word*)) =
(*of-bl* (*take* (*len-of TYPE*('*a*::*len*) − *len-of TYPE*('*d*::*len0*)) (*to-bl c*)),
of-bl (*drop* (*len-of TYPE*('*a*) − *len-of TYPE*('*d*)) (*to-bl c*)))
apply (*rule word-split-bl* [*THEN iffD1*])
apply (*unfold word-size*)
apply (*rule refl conjI*)+
done

— keep quantifiers for use in simplification
lemma *test-bit-split'*:
word-split c = (*a*, *b*) −−> (*ALL n m*. *b* !! *n* = (*n* < *size b* & *c* !! *n*) &
a !! *m* = (*m* < *size a* & *c* !! (*m* + *size b*)))
apply (*unfold word-split-bin' test-bit-bin*)
apply (*clarify*)
apply (*clarsimp simp*: *word-ubin.eq-norm nth-bintr word-size split*: *prod.splits*)
apply (*drule bin-nth-split*)
apply *safe*
apply (*simp-all add*: *add.commute*)
apply (*erule bin-nth-uint-imp*)+
done

lemma *test-bit-split*:
word-split c = (*a*, *b*) $\Longrightarrow$
($\forall$ *n*::*nat*. *b* !! *n* $\longleftrightarrow$ *n* < *size b* $\wedge$ *c* !! *n*) $\wedge$ ($\forall$ *m*::*nat*. *a* !! *m* $\longleftrightarrow$ *m* < *size a* $\wedge$ *c* !! (*m* + *size b*))
by (*simp add*: *test-bit-split'*)

lemma *test-bit-split-eq*: *word-split c* = (*a*, *b*) $\longleftrightarrow$
((*ALL n*::*nat*. *b* !! *n* = (*n* < *size b* & *c* !! *n*)) &
(*ALL m*::*nat*. *a* !! *m* = (*m* < *size a* & *c* !! (*m* + *size b*))))
apply (*rule-tac iffI*)
apply (*rule-tac conjI*)
apply (*erule test-bit-split* [*THEN conjunct1*])
apply (*erule test-bit-split* [*THEN conjunct2*])
apply (*case-tac word-split c*)
apply (*frule test-bit-split*)
apply (*erule trans*)
apply (*fastforce intro* ! : *word-eqI simp add* : *word-size*)
done

— this odd result is analogous to *ucast-id*, result to the length given by the result type

lemma *word-cat-id*: *word-cat a b* = *b*
unfolding *word-cat-bin'* **by** (*simp add*: *word-ubin.inverse-norm*)

— limited hom result
lemma *word-cat-hom*:
len-of TYPE('*a*::*len0*) <= *len-of TYPE*('*b*::*len0*) + *len-of TYPE* ('*c*::*len0*)
$\Longrightarrow$
(*word-cat* (*word-of-int w* :: '*b word*) (*b* :: '*c word*) :: '*a word*) =
word-of-int (*bin-cat w* (*size b*) (*uint b*))
apply (*unfold word-cat-def word-size*)
apply (*clarsimp simp add*: *word-ubin.norm-eq-iff* [*symmetric*]
word-ubin.eq-norm bintr-cat min.absorb1)
done

lemma *word-cat-split-alt*:
size w <= size u + size v ⟹ word-split w = (u, v) ⟹ word-cat u v = w
apply (*rule word-eqI*)
apply (*drule test-bit-split*)
apply (*clarsimp simp add* : *test-bit-cat word-size*)
apply *safe*
apply *arith*
done

lemmas *word-cat-split-size* = *sym* [*THEN* [*2*] *word-cat-split-alt* [*symmetric*]]

127.26.1 Split and slice

lemma *split-slices*:
word-split w = (u, v) ⟹ u = slice (size v) w & v = slice 0 w
apply (*drule test-bit-split*)
apply (*rule conjI*)
apply (*rule word-eqI*, *clarsimp simp*: *nth-slice word-size*)+
done

lemma *slice-cat1* [*OF refl*]:
wc = word-cat a b ⟹ size wc >= size a + size b ⟹ slice (size b) wc = a
apply *safe*
apply (*rule word-eqI*)
apply (*simp add*: *nth-slice test-bit-cat word-size*)
done

lemmas *slice-cat2* = *trans* [*OF slice-id word-cat-id*]

lemma *cat-slices*:
a = slice n c ⟹ b = slice 0 c ⟹ n = size b ⟹
size a + size b >= size c ⟹ word-cat a b = c
apply *safe*
apply (*rule word-eqI*)
apply (*simp add*: *nth-slice test-bit-cat word-size*)
apply *safe*
apply *arith*
done

lemma *word-split-cat-alt*:
w = word-cat u v ⟹ size u + size v <= size w ⟹ word-split w = (u, v)
apply (*case-tac word-split w*)
apply (*rule trans*, *assumption*)
apply (*drule test-bit-split*)
apply *safe*
apply (*rule word-eqI*, *clarsimp simp*: *test-bit-cat word-size*)+
done

lemmas *word-cat-bl-no-bin* [*simp*] =

word-cat-bl [**where** *a*=*numeral a* **and** *b*=*numeral b*,
unfolded to-bl-numeral]
for *a b*

lemmas *word-split-bl-no-bin* [*simp*] =
word-split-bl-eq [**where** *c*=*numeral c*, *unfolded to-bl-numeral*] **for** *c*

this odd result arises from the fact that the statement of the result implies that the decoded words are of the same type, and therefore of the same length, as the original word

lemma *word-rsplit-same*: *word-rsplit w* = [*w*]
unfolding *word-rsplit-def* **by** (*simp add* : *bin-rsplit-all*)

lemma *word-rsplit-empty-iff-size*:
(*word-rsplit w* = []) = (*size w* = *0*)
unfolding *word-rsplit-def bin-rsplit-def word-size*
by (*simp add*: *bin-rsplit-aux-simp-alt Let-def split*: *prod.split*)

lemma *test-bit-rsplit*:
sw = *word-rsplit w* ⟹ *m* < *size* (*hd sw* :: *'a* :: *len word*) ⟹
k < *length sw* ⟹ (*rev sw* ! *k*) !! *m* = (*w* !! (*k* ∗ *size* (*hd sw*) + *m*))
apply (*unfold word-rsplit-def word-test-bit-def*)
apply (*rule trans*)
apply (*rule-tac f* = %*x*. *bin-nth x m* **in** *arg-cong*)
apply (*rule nth-map* [*symmetric*])
apply *simp*
apply (*rule bin-nth-rsplit*)
apply *simp-all*
apply (*simp add* : *word-size rev-map*)
apply (*rule trans*)
defer
apply (*rule map-ident* [*THEN fun-cong*])
apply (*rule refl* [*THEN map-cong*])
apply (*simp add* : *word-ubin.eq-norm*)
apply (*erule bin-rsplit-size-sign* [*OF len-gt-0 refl*])
done

lemma *word-rcat-bl*: *word-rcat wl* = *of-bl* (*concat* (*map to-bl wl*))
unfolding *word-rcat-def to-bl-def' of-bl-def*
by (*clarsimp simp add* : *bin-rcat-bl*)

lemma *size-rcat-lem'*:
size (*concat* (*map to-bl wl*)) = *length wl* ∗ *size* (*hd wl*)
unfolding *word-size* **by** (*induct wl*) *auto*

lemmas *size-rcat-lem* = *size-rcat-lem'* [*unfolded word-size*]

lemmas *td-gal-lt-len* = *len-gt-0* [*THEN td-gal-lt*]

lemma *nth-rcat-lem*:
*n < length (wl::'a word list) * len-of TYPE('a::len) ⟹*
rev (concat (map to-bl wl)) ! n =
rev (to-bl (rev wl ! (n div len-of TYPE('a)))) ! (n mod len-of TYPE('a))
apply (*induct wl*)
apply *clarsimp*
apply (*clarsimp simp add : nth-append size-rcat-lem*)
apply (*simp (no-asm-use) only: mult-Suc [symmetric]*
td-gal-lt-len less-Suc-eq-le mod-div-equality')
apply *clarsimp*
done

lemma *test-bit-rcat*:
sw = size (hd wl :: 'a :: len word) ⟹ rc = word-rcat wl ⟹ rc !! n =
(n < size rc & n div sw < size wl & (rev wl) ! (n div sw) !! (n mod sw))
apply (*unfold word-rcat-bl word-size*)
apply (*clarsimp simp add :*
test-bit-of-bl size-rcat-lem word-size td-gal-lt-len)
apply *safe*
apply (*auto simp add :*
test-bit-bl word-size td-gal-lt-len [THEN iffD2, THEN nth-rcat-lem])
done

lemma *foldl-eq-foldr*:
foldl op + x xs = foldr op + (x # xs) (0 :: 'a :: comm-monoid-add)
by (*induct xs arbitrary: x*) (*auto simp add : add.assoc*)

lemmas *test-bit-cong = arg-cong* [**where** *f = test-bit, THEN fun-cong*]

lemmas *test-bit-rsplit-alt =*
trans [OF nth-rev-alt [THEN test-bit-cong]
test-bit-rsplit [OF refl asm-rl diff-Suc-less]]

— lazy way of expressing that u and v, and su and sv, have same types
lemma *word-rsplit-len-indep [OF refl refl refl refl]*:
[u,v] = p ⟹ [su,sv] = q ⟹ word-rsplit u = su ⟹
word-rsplit v = sv ⟹ length su = length sv
apply (*unfold word-rsplit-def*)
apply (*auto simp add : bin-rsplit-len-indep*)
done

lemma *length-word-rsplit-size*:
n = len-of TYPE ('a :: len) ⟹
*(length (word-rsplit w :: 'a word list) <= m) = (size w <= m * n)*
apply (*unfold word-rsplit-def word-size*)
apply (*clarsimp simp add : bin-rsplit-len-le*)
done

lemmas *length-word-rsplit-lt-size =*

length-word-rsplit-size [*unfolded Not-eq-iff linorder-not-less* [*symmetric*]]

lemma *length-word-rsplit-exp-size*:
n = *len-of TYPE* ('*a* :: *len*) ⟹
length (*word-rsplit w* :: '*a word list*) = (*size w* + *n* − *1*) *div n*
unfolding *word-rsplit-def* **by** (*clarsimp simp add* : *word-size bin-rsplit-len*)

lemma *length-word-rsplit-even-size*:
n = *len-of TYPE* ('*a* :: *len*) ⟹ *size w* = *m* ∗ *n* ⟹
length (*word-rsplit w* :: '*a word list*) = *m*
by (*clarsimp simp add* : *length-word-rsplit-exp-size given-quot-alt*)

lemmas *length-word-rsplit-exp-size'* = *refl* [*THEN length-word-rsplit-exp-size*]

lemmas *tdle* = *iffD2* [*OF split-div-lemma refl*, *THEN conjunct1*]
lemmas *dtle* = *xtr4* [*OF tdle mult.commute*]

lemma *word-rcat-rsplit*: *word-rcat* (*word-rsplit w*) = *w*
apply (*rule word-eqI*)
apply (*clarsimp simp add* : *test-bit-rcat word-size*)
apply (*subst refl* [*THEN test-bit-rsplit*])
apply (*simp-all add*: *word-size*
refl [*THEN length-word-rsplit-size* [*simplified not-less* [*symmetric*], *simplified*]])
apply *safe*
apply (*erule xtr7*, *rule len-gt-0* [*THEN dtle*])+
done

lemma *size-word-rsplit-rcat-size*:
⟦*word-rcat* (*ws*::'*a*::*len word list*) = (*frcw*::'*b*::*len0 word*);
size frcw = *length ws* ∗ *len-of TYPE*('*a*)⟧
⟹ *length* (*word-rsplit frcw*::'*a word list*) = *length ws*
apply (*clarsimp simp add* : *word-size length-word-rsplit-exp-size'*)
apply (*fast intro*: *given-quot-alt*)
done

lemma *msrevs*:
fixes *n*::*nat*
shows *0* < *n* ⟹ (*k* ∗ *n* + *m*) *div n* = *m div n* + *k*
and (*k* ∗ *n* + *m*) *mod n* = *m mod n*
by (*auto simp*: *add.commute*)

lemma *word-rsplit-rcat-size* [*OF refl*]:
word-rcat (*ws* :: '*a* :: *len word list*) = *frcw* ⟹
size frcw = *length ws* ∗ *len-of TYPE* ('*a*) ⟹ *word-rsplit frcw* = *ws*
apply (*frule size-word-rsplit-rcat-size*, *assumption*)
apply (*clarsimp simp add* : *word-size*)
apply (*rule nth-equalityI*, *assumption*)
apply *clarsimp*

apply (*rule word-eqI* [*rule-format*])
apply (*rule trans*)
apply (*rule test-bit-rsplit-alt*)
apply (*clarsimp simp*: *word-size*)+
apply (*rule trans*)
apply (*rule test-bit-rcat* [*OF refl refl*])
apply (*simp add*: *word-size*)
apply (*subst nth-rev*)
apply *arith*
apply (*simp add*: *le0* [*THEN* [*2*] *xtr7*, *THEN diff-Suc-less*])
apply *safe*
apply (*simp add*: *diff-mult-distrib*)
apply (*rule mpl-lem*)
apply (*cases size ws*)
apply *simp-all*
done

127.27 Rotation

lemmas *rotater-0′* [*simp*] = *rotater-def* [**where** *n* = *0*, *simplified*]

lemmas *word-rot-defs* = *word-roti-def word-rotr-def word-rotl-def*

lemma *rotate-eq-mod*:
m mod length xs = *n mod length xs* ⟹ *rotate m xs* = *rotate n xs*
apply (*rule box-equals*)
defer
apply (*rule rotate-conv-mod* [*symmetric*])+
apply *simp*
done

lemmas *rotate-eqs* =
trans [*OF rotate0* [*THEN fun-cong*] *id-apply*]
rotate-rotate [*symmetric*]
rotate-id
rotate-conv-mod
rotate-eq-mod

127.27.1 Rotation of list to right

lemma *rotate1-rl′*: *rotater1* (*l* @ [*a*]) = *a* # *l*
unfolding *rotater1-def* **by** (*cases l*) *auto*

lemma *rotate1-rl* [*simp*] : *rotater1* (*rotate1 l*) = *l*
apply (*unfold rotater1-def*)
apply (*cases l*)
apply (*case-tac* [*2*] *list*)
apply *auto*
done

lemma *rotate1-lr* [*simp*] : *rotate1* (*rotater1 l*) = *l*
unfolding *rotater1-def* **by** (*cases l*) *auto*

lemma *rotater1-rev'*: *rotater1* (*rev xs*) = *rev* (*rotate1 xs*)
apply (*cases xs*)
apply (*simp add* : *rotater1-def*)
apply (*simp add* : *rotate1-rl'*)
done

lemma *rotater-rev'*: *rotater n* (*rev xs*) = *rev* (*rotate n xs*)
unfolding *rotater-def* **by** (*induct n*) (*auto intro*: *rotater1-rev'*)

lemma *rotater-rev*: *rotater n ys* = *rev* (*rotate n* (*rev ys*))
using *rotater-rev'* [**where** *xs* = *rev ys*] **by** *simp*

lemma *rotater-drop-take*:
rotater n xs =
drop (*length xs* − *n mod length xs*) *xs* @
take (*length xs* − *n mod length xs*) *xs*
by (*clarsimp simp add* : *rotater-rev rotate-drop-take rev-take rev-drop*)

lemma *rotater-Suc* [*simp*] :
rotater (*Suc n*) *xs* = *rotater1* (*rotater n xs*)
unfolding *rotater-def* **by** *auto*

lemma *rotate-inv-plus* [*rule-format*] :
ALL k. *k* = *m* + *n* −−> *rotater k* (*rotate n xs*) = *rotater m xs* &
rotate k (*rotater n xs*) = *rotate m xs* &
rotater n (*rotate k xs*) = *rotate m xs* &
rotate n (*rotater k xs*) = *rotater m xs*
unfolding *rotater-def rotate-def*
by (*induct n*) (*auto intro*: *funpow-swap1* [*THEN trans*])

lemmas *rotate-inv-rel* = *le-add-diff-inverse2* [*symmetric*, *THEN rotate-inv-plus*]

lemmas *rotate-inv-eq* = *order-refl* [*THEN rotate-inv-rel*, *simplified*]

lemmas *rotate-lr* [*simp*] = *rotate-inv-eq* [*THEN conjunct1*]
lemmas *rotate-rl* [*simp*] = *rotate-inv-eq* [*THEN conjunct2*, *THEN conjunct1*]

lemma *rotate-gal*: (*rotater n xs* = *ys*) = (*rotate n ys* = *xs*)
by *auto*

lemma *rotate-gal'*: (*ys* = *rotater n xs*) = (*xs* = *rotate n ys*)
by *auto*

lemma *length-rotater* [*simp*]:
length (*rotater n xs*) = *length xs*
by (*simp add* : *rotater-rev*)

lemma *restrict-to-left*:
  assumes $x = y$
  shows $(x = z) = (y = z)$
  using *assms* **by** *simp*

lemmas *rrs0* = *rotate-eqs* [*THEN restrict-to-left*,
  simplified rotate-gal [*symmetric*] *rotate-gal'* [*symmetric*]]
lemmas *rrs1* = *rrs0* [*THEN refl* [*THEN rev-iffD1*]]
lemmas *rotater-eqs* = *rrs1* [*simplified length-rotater*]
lemmas *rotater-0* = *rotater-eqs* (*1*)
lemmas *rotater-add* = *rotater-eqs* (*2*)

127.27.2 map, map2, commuting with rotate(r)

lemma *butlast-map*:
  xs ~= [] ⟹ *butlast* (*map f xs*) = *map f* (*butlast xs*)
  by (*induct xs*) *auto*

lemma *rotater1-map*: *rotater1* (*map f xs*) = *map f* (*rotater1 xs*)
  unfolding *rotater1-def*
  by (*cases xs*) (*auto simp add*: *last-map butlast-map*)

lemma *rotater-map*:
  rotater n (*map f xs*) = *map f* (*rotater n xs*)
  unfolding *rotater-def*
  by (*induct n*) (*auto simp add* : *rotater1-map*)

lemma *but-last-zip* [*rule-format*] :
  ALL ys. *length xs* = *length ys* --> *xs* ~= [] -->
    last (*zip xs ys*) = (*last xs*, *last ys*) &
    butlast (*zip xs ys*) = *zip* (*butlast xs*) (*butlast ys*)
  apply (*induct xs*)
  apply *auto*
    apply ((*case-tac ys*, *auto simp*: *neq-Nil-conv*)[*1*])+
  done

lemma *but-last-map2* [*rule-format*] :
  ALL ys. *length xs* = *length ys* --> *xs* ~= [] -->
    last (*map2 f xs ys*) = *f* (*last xs*) (*last ys*) &
    butlast (*map2 f xs ys*) = *map2 f* (*butlast xs*) (*butlast ys*)
  apply (*induct xs*)
  apply *auto*
    apply (*unfold map2-def*)
    apply ((*case-tac ys*, *auto simp*: *neq-Nil-conv*)[*1*])+
  done

lemma *rotater1-zip*:
  length xs = *length ys* ⟹

rotater1 (*zip xs ys*) = *zip* (*rotater1 xs*) (*rotater1 ys*)
apply (*unfold rotater1-def*)
apply (*cases xs*)
apply *auto*
apply ((*case-tac ys*, *auto simp*: *neq-Nil-conv but-last-zip*)[*1*])+
done

lemma *rotater1-map2*:
length xs = *length ys* $\Longrightarrow$
rotater1 (*map2 f xs ys*) = *map2 f* (*rotater1 xs*) (*rotater1 ys*)
unfolding *map2-def* **by** (*simp add*: *rotater1-map rotater1-zip*)

lemmas *lrth* =
box-equals [*OF asm-rl length-rotater* [*symmetric*]
length-rotater [*symmetric*],
THEN rotater1-map2]

lemma *rotater-map2*:
length xs = *length ys* $\Longrightarrow$
rotater n (*map2 f xs ys*) = *map2 f* (*rotater n xs*) (*rotater n ys*)
by (*induct n*) (*auto intro*!: *lrth*)

lemma *rotate1-map2*:
length xs = *length ys* $\Longrightarrow$
rotate1 (*map2 f xs ys*) = *map2 f* (*rotate1 xs*) (*rotate1 ys*)
apply (*unfold map2-def*)
apply (*cases xs*)
apply (*cases ys*, *auto*)+
done

lemmas *lth* = *box-equals* [*OF asm-rl length-rotate* [*symmetric*]
length-rotate [*symmetric*], *THEN rotate1-map2*]

lemma *rotate-map2*:
length xs = *length ys* $\Longrightarrow$
rotate n (*map2 f xs ys*) = *map2 f* (*rotate n xs*) (*rotate n ys*)
by (*induct n*) (*auto intro*!: *lth*)

— corresponding equalities for word rotation

lemma *to-bl-rotl*:
to-bl (*word-rotl n w*) = *rotate n* (*to-bl w*)
by (*simp add*: *word-bl.Abs-inverse′ word-rotl-def*)

lemmas *blrs0* = *rotate-eqs* [*THEN to-bl-rotl* [*THEN trans*]]

lemmas *word-rotl-eqs* =
blrs0 [*simplified word-bl-Rep′ word-bl.Rep-inject to-bl-rotl* [*symmetric*]]

lemma *to-bl-rotr*:
to-bl (*word-rotr n w*) = *rotater n* (*to-bl w*)
by (*simp add*: *word-bl.Abs-inverse′ word-rotr-def*)

lemmas *brrs0* = *rotater-eqs* [*THEN to-bl-rotr* [*THEN trans*]]

lemmas *word-rotr-eqs* =
brrs0 [*simplified word-bl-Rep′ word-bl.Rep-inject to-bl-rotr* [*symmetric*]]

declare *word-rotr-eqs* (*1*) [*simp*]
declare *word-rotl-eqs* (*1*) [*simp*]

lemma
word-rot-rl [*simp*]:
word-rotl k (*word-rotr k v*) = *v* **and**
word-rot-lr [*simp*]:
word-rotr k (*word-rotl k v*) = *v*
by (*auto simp add*: *to-bl-rotr to-bl-rotl word-bl.Rep-inject* [*symmetric*])

lemma
word-rot-gal:
(*word-rotr n v* = *w*) = (*word-rotl n w* = *v*) **and**
word-rot-gal′:
(*w* = *word-rotr n v*) = (*v* = *word-rotl n w*)
by (*auto simp*: *to-bl-rotr to-bl-rotl word-bl.Rep-inject* [*symmetric*]
dest: *sym*)

lemma *word-rotr-rev*:
word-rotr n w = *word-reverse* (*word-rotl n* (*word-reverse w*))
by (*simp only*: *word-bl.Rep-inject* [*symmetric*] *to-bl-word-rev*
to-bl-rotr to-bl-rotl rotater-rev)

lemma *word-roti-0* [*simp*]: *word-roti 0 w* = *w*
by (*unfold word-rot-defs*) *auto*

lemmas *abl-cong* = *arg-cong* [**where** *f* = *of-bl*]

lemma *word-roti-add*:
word-roti (*m* + *n*) *w* = *word-roti m* (*word-roti n w*)
proof −
have *rotater-eq-lem*:
$\bigwedge m\ n\ xs.\ m = n \Longrightarrow rotater\ m\ xs = rotater\ n\ xs$
by *auto*

have *rotate-eq-lem*:
$\bigwedge m\ n\ xs.\ m = n \Longrightarrow rotate\ m\ xs = rotate\ n\ xs$
by *auto*

note *rpts* [*symmetric*] =
rotate-inv-plus [*THEN conjunct1*]
rotate-inv-plus [*THEN conjunct2*, *THEN conjunct1*]
rotate-inv-plus [*THEN conjunct2*, *THEN conjunct2*, *THEN conjunct1*]
rotate-inv-plus [*THEN conjunct2*, *THEN conjunct2*, *THEN conjunct2*]

note *rrp* = *trans* [*symmetric*, *OF rotate-rotate rotate-eq-lem*]
note *rrrp* = *trans* [*symmetric*, *OF rotater-add* [*symmetric*] *rotater-eq-lem*]

show *?thesis*
apply (*unfold word-rot-defs*)
apply (*simp only*: *split*: *if-split*)
apply (*safe intro*!: *abl-cong*)
apply (*simp-all only*: *to-bl-rotl* [*THEN word-bl.Rep-inverse′*]
to-bl-rotl
to-bl-rotr [*THEN word-bl.Rep-inverse′*]
to-bl-rotr)
apply (*rule rrp rrrp rpts*,
simp add: *nat-add-distrib* [*symmetric*]
nat-diff-distrib [*symmetric*])+
done
qed

lemma *word-roti-conv-mod′*: *word-roti n w* = *word-roti* (*n mod int* (*size w*)) *w*
apply (*unfold word-rot-defs*)
apply (*cut-tac y*=*size w* **in** *gt-or-eq-0*)
apply (*erule disjE*)
apply *simp-all*
apply (*safe intro*!: *abl-cong*)
apply (*rule rotater-eqs*)
apply (*simp add*: *word-size nat-mod-distrib*)
apply (*simp add*: *rotater-add* [*symmetric*] *rotate-gal* [*symmetric*])
apply (*rule rotater-eqs*)
apply (*simp add*: *word-size nat-mod-distrib*)
apply (*rule of-nat-eq-0-iff* [*THEN iffD1*])
apply (*auto simp add*: *not-le mod-eq-0-iff-dvd zdvd-int nat-add-distrib* [*symmetric*])
using *mod-mod-trivial zmod-eq-dvd-iff*
apply *blast*
done

lemmas *word-roti-conv-mod* = *word-roti-conv-mod′* [*unfolded word-size*]

127.27.3 "Word rotation commutes with bit-wise operations

locale *word-rotate*
begin

lemmas *word-rot-defs′* = *to-bl-rotl to-bl-rotr*

```
lemmas blwl-syms [symmetric] = bl-word-not bl-word-and bl-word-or bl-word-xor

lemmas lbl-lbl = trans [OF word-bl-Rep' word-bl-Rep' [symmetric]]

lemmas ths-map2 [OF lbl-lbl] = rotate-map2 rotater-map2

lemmas ths-map [where xs = to-bl v] = rotate-map rotater-map for v

lemmas th1s [simplified word-rot-defs' [symmetric]] = ths-map2 ths-map

lemma word-rot-logs:
  word-rotl n (NOT v) = NOT word-rotl n v
  word-rotr n (NOT v) = NOT word-rotr n v
  word-rotl n (x AND y) = word-rotl n x AND word-rotl n y
  word-rotr n (x AND y) = word-rotr n x AND word-rotr n y
  word-rotl n (x OR y) = word-rotl n x OR word-rotl n y
  word-rotr n (x OR y) = word-rotr n x OR word-rotr n y
  word-rotl n (x XOR y) = word-rotl n x XOR word-rotl n y
  word-rotr n (x XOR y) = word-rotr n x XOR word-rotr n y
  by (rule word-bl.Rep-eqD,
      rule word-rot-defs' [THEN trans],
      simp only: blwl-syms [symmetric],
      rule th1s [THEN trans],
      rule refl)+
end

lemmas word-rot-logs = word-rotate.word-rot-logs

lemmas bl-word-rotl-dt = trans [OF to-bl-rotl rotate-drop-take,
  simplified word-bl-Rep']

lemmas bl-word-rotr-dt = trans [OF to-bl-rotr rotater-drop-take,
  simplified word-bl-Rep']

lemma bl-word-roti-dt':
  n = nat ((- i) mod int (size (w :: 'a :: len word))) ⟹
    to-bl (word-roti i w) = drop n (to-bl w) @ take n (to-bl w)
  apply (unfold word-roti-def)
  apply (simp add: bl-word-rotl-dt bl-word-rotr-dt word-size)
  apply safe
   apply (simp add: zmod-zminus1-eq-if)
   apply safe
    apply (simp add: nat-mult-distrib)
   apply (simp add: nat-diff-distrib [OF pos-mod-sign pos-mod-conj
                                   [THEN conjunct2, THEN order-less-imp-le]]
                 nat-mod-distrib)
  apply (simp add: nat-mod-distrib)
  done
```

lemmas *bl-word-roti-dt* = *bl-word-roti-dt′* [*unfolded word-size*]

lemmas *word-rotl-dt* = *bl-word-rotl-dt* [*THEN word-bl.Rep-inverse′* [*symmetric*]]
lemmas *word-rotr-dt* = *bl-word-rotr-dt* [*THEN word-bl.Rep-inverse′* [*symmetric*]]
lemmas *word-roti-dt* = *bl-word-roti-dt* [*THEN word-bl.Rep-inverse′* [*symmetric*]]

lemma *word-rotx-0* [*simp*] : *word-rotr i 0* = *0* & *word-rotl i 0* = *0*
by (*simp add* : *word-rotr-dt word-rotl-dt replicate-add* [*symmetric*])

lemma *word-roti-0′* [*simp*] : *word-roti n 0* = *0*
unfolding *word-roti-def* **by** *auto*

lemmas *word-rotr-dt-no-bin′* [*simp*] =
word-rotr-dt [**where** *w*=*numeral w*, *unfolded to-bl-numeral*] **for** *w*

lemmas *word-rotl-dt-no-bin′* [*simp*] =
word-rotl-dt [**where** *w*=*numeral w*, *unfolded to-bl-numeral*] **for** *w*

declare *word-roti-def* [*simp*]

127.28 Maximum machine word

lemma *word-int-cases*:
obtains *n* **where** (*x* ::′*a*::*len0 word*) = *word-of-int n* **and** *0* ≤ *n* **and** *n* < *2*ˆ*len-of TYPE*(′*a*)
by (*cases x rule*: *word-uint.Abs-cases*) (*simp add*: *uints-num*)

lemma *word-nat-cases* [*cases type*: *word*]:
obtains *n* **where** (*x* ::′*a*::*len word*) = *of-nat n* **and** *n* < *2*ˆ*len-of TYPE*(′*a*)
by (*cases x rule*: *word-unat.Abs-cases*) (*simp add*: *unats-def*)

lemma *max-word-eq*: (*max-word*::′*a*::*len word*) = *2*ˆ*len-of TYPE*(′*a*) − *1*
by (*simp add*: *max-word-def word-of-int-hom-syms word-of-int-2p*)

lemma *max-word-max* [*simp*,*intro*!]: *n* ≤ *max-word*
by (*cases n rule*: *word-int-cases*)
(*simp add*: *max-word-def word-le-def int-word-uint mod-pos-pos-trivial del*: *minus-mod-self1*)

lemma *word-of-int-2p-len*: *word-of-int* (*2* ˆ *len-of TYPE*(′*a*)) = (*0*::′*a*::*len0 word*)
by (*subst word-uint.Abs-norm* [*symmetric*]) *simp*

lemma *word-pow-0*:
(*2*::′*a*::*len word*) ˆ *len-of TYPE*(′*a*) = *0*
proof −
have *word-of-int* (*2* ˆ *len-of TYPE*(′*a*)) = (*0*::′*a word*)
by (*rule word-of-int-2p-len*)

thus *?thesis* **by** (*simp add*: *word-of-int-2p*)
qed

lemma *max-word-wrap*: $x + 1 = 0 \Longrightarrow x = max\text{-}word$
apply (*simp add*: *max-word-eq*)
apply *uint-arith*
apply *auto*
apply (*simp add*: *word-pow-0*)
done

lemma *max-word-minus*:
max-word = (−*1*::′*a*::*len word*)
proof −
have −*1* + *1* = (*0*::′*a word*) **by** *simp*
thus *?thesis* **by** (*rule max-word-wrap* [*symmetric*])
qed

lemma *max-word-bl* [*simp*]:
to-bl (*max-word*::′*a*::*len word*) = *replicate* (*len-of TYPE*(′*a*)) *True*
by (*subst max-word-minus to-bl-n1*)+ *simp*

lemma *max-test-bit* [*simp*]:
(*max-word*::′*a*::*len word*) !! *n* = (*n* < *len-of TYPE*(′*a*))
by (*auto simp add*: *test-bit-bl word-size*)

lemma *word-and-max* [*simp*]:
x AND max-word = *x*
by (*rule word-eqI*) (*simp add*: *word-ops-nth-size word-size*)

lemma *word-or-max* [*simp*]:
x OR max-word = *max-word*
by (*rule word-eqI*) (*simp add*: *word-ops-nth-size word-size*)

lemma *word-ao-dist2*:
x AND (*y OR z*) = *x AND y OR x AND* (*z*::′*a*::*len0 word*)
by (*rule word-eqI*) (*auto simp add*: *word-ops-nth-size word-size*)

lemma *word-oa-dist2*:
x OR y AND z = (*x OR y*) *AND* (*x OR* (*z*::′*a*::*len0 word*))
by (*rule word-eqI*) (*auto simp add*: *word-ops-nth-size word-size*)

lemma *word-and-not* [*simp*]:
x AND NOT x = (*0*::′*a*::*len0 word*)
by (*rule word-eqI*) (*auto simp add*: *word-ops-nth-size word-size*)

lemma *word-or-not* [*simp*]:
x OR NOT x = *max-word*
by (*rule word-eqI*) (*auto simp add*: *word-ops-nth-size word-size*)

lemma *word-boolean*:
boolean (*op AND*) (*op OR*) *bitNOT 0 max-word*
apply (*rule boolean.intro*)
apply (*rule word-bw-assocs*)
apply (*rule word-bw-assocs*)
apply (*rule word-bw-comms*)
apply (*rule word-bw-comms*)
apply (*rule word-ao-dist2*)
apply (*rule word-oa-dist2*)
apply (*rule word-and-max*)
apply (*rule word-log-esimps*)
apply (*rule word-and-not*)
apply (*rule word-or-not*)
done

interpretation *word-bool-alg*:
boolean op AND op OR bitNOT 0 max-word
by (*rule word-boolean*)

lemma *word-xor-and-or*:
x XOR y = *x AND NOT y OR NOT x AND* (*y*::*'a*::*len0 word*)
by (*rule word-eqI*) (*auto simp add*: *word-ops-nth-size word-size*)

interpretation *word-bool-alg*:
boolean-xor op AND op OR bitNOT 0 max-word op XOR
apply (*rule boolean-xor.intro*)
apply (*rule word-boolean*)
apply (*rule boolean-xor-axioms.intro*)
apply (*rule word-xor-and-or*)
done

lemma *shiftr-x-0* [*iff*]:
(*x*::*'a*::*len0 word*) >> *0* = *x*
by (*simp add*: *shiftr-bl*)

lemma *shiftl-x-0* [*simp*]:
(*x* :: *'a* :: *len word*) << *0* = *x*
by (*simp add*: *shiftl-t2n*)

lemma *shiftl-1* [*simp*]:
(*1*::*'a*::*len word*) << *n* = *2*^*n*
by (*simp add*: *shiftl-t2n*)

lemma *uint-lt-0* [*simp*]:
uint x < *0* = *False*
by (*simp add*: *linorder-not-less*)

lemma *shiftr1-1* [*simp*]:
shiftr1 (*1*::*'a*::*len word*) = *0*

unfolding *shiftr1-def* **by** *simp*

lemma *shiftr-1*[*simp*]:
(1::$'a$::*len word*) >> n = (*if* $n = 0$ *then* 1 *else* 0)
by (*induct n*) (*auto simp*: *shiftr-def*)

lemma *word-less-1* [*simp*]:
((x::$'a$::*len word*) < 1) = ($x = 0$)
by (*simp add*: *word-less-nat-alt unat-0-iff*)

lemma *to-bl-mask*:
to-bl (*mask n* :: $'a$::*len word*) =
replicate (*len-of TYPE*($'a$) − n) *False* @
replicate (*min* (*len-of TYPE*($'a$)) n) *True*
by (*simp add*: *mask-bl word-rep-drop min-def*)

lemma *map-replicate-True*:
n = *length xs* $\Longrightarrow$
map ($\lambda(x,y)$. x & y) (*zip xs* (*replicate n True*)) = *xs*
by (*induct xs arbitrary*: n) *auto*

lemma *map-replicate-False*:
n = *length xs* $\Longrightarrow$ *map* ($\lambda(x,y)$. x & y)
(*zip xs* (*replicate n False*)) = *replicate n False*
by (*induct xs arbitrary*: n) *auto*

lemma *bl-and-mask*:
fixes w :: $'a$::*len word*
fixes n
defines $n' \equiv$ *len-of TYPE*($'a$) − n
shows *to-bl* (w *AND mask* n) = *replicate* n' *False* @ *drop* n' (*to-bl* w)
proof −
note [*simp*] = *map-replicate-True map-replicate-False*
have *to-bl* (w *AND mask* n) =
map2 op & (*to-bl* w) (*to-bl* (*mask* n::$'a$::*len word*))
by (*simp add*: *bl-word-and*)
also
have *to-bl* w = *take* n' (*to-bl* w) @ *drop* n' (*to-bl* w) **by** *simp*
also
have *map2 op* & ... (*to-bl* (*mask* n::$'a$::*len word*)) =
replicate n' *False* @ *drop* n' (*to-bl* w)
unfolding *to-bl-mask* n'*-def map2-def*
by (*subst zip-append*) *auto*
finally
show *?thesis* **.**
qed

lemma *drop-rev-takefill*:
length xs $\le n \Longrightarrow$

drop (*n* − *length xs*) (*rev* (*takefill False n* (*rev xs*))) = *xs*
by (*simp add*: *takefill-alt rev-take*)

lemma *map-nth-0* [*simp*]:
map (*op* !! (*0*::′*a*::*len0 word*)) *xs* = *replicate* (*length xs*) *False*
by (*induct xs*) *auto*

lemma *uint-plus-if-size*:
uint (*x* + *y*) =
(*if uint x* + *uint y* < *2*ˆ*size x then*
uint x + *uint y*
else
uint x + *uint y* − *2*ˆ*size x*)
by (*simp add*: *word-arith-wis int-word-uint mod-add-if-z*
word-size)

lemma *unat-plus-if-size*:
unat (*x* + (*y*::′*a*::*len word*)) =
(*if unat x* + *unat y* < *2*ˆ*size x then*
unat x + *unat y*
else
unat x + *unat y* − *2*ˆ*size x*)
apply (*subst word-arith-nat-defs*)
apply (*subst unat-of-nat*)
apply (*simp add*: *mod-nat-add word-size*)
done

lemma *word-neq-0-conv*:
fixes *w* :: ′*a* :: *len word*
shows (*w* ≠ *0*) = (*0* < *w*)
unfolding *word-gt-0* **by** *simp*

lemma *max-lt*:
unat (*max a b div c*) = *unat* (*max a b*) *div unat* (*c*:: ′*a* :: *len word*)
by (*fact unat-div*)

lemma *uint-sub-if-size*:
uint (*x* − *y*) =
(*if uint y* ≤ *uint x then*
uint x − *uint y*
else
uint x − *uint y* + *2*ˆ*size x*)
by (*simp add*: *word-arith-wis int-word-uint mod-sub-if-z*
word-size)

lemma *unat-sub*:
b <= *a* ⟹ *unat* (*a* − *b*) = *unat a* − *unat b*
by (*simp add*: *unat-def uint-sub-if-size word-le-def nat-diff-distrib*)

lemmas *word-less-sub1-numberof* [*simp*] = *word-less-sub1* [*of numeral w*] **for** *w*
lemmas *word-le-sub1-numberof* [*simp*] = *word-le-sub1* [*of numeral w*] **for** *w*

lemma *word-of-int-minus*:
  word-of-int (*2^len-of TYPE('a)* − *i*) = (*word-of-int* (−*i*)::'*a*::*len word*)
proof −
  have *x*: *2^len-of TYPE('a)* − *i* = −*i* + *2^len-of TYPE('a)* **by** *simp*
  show *?thesis*
    apply (*subst x*)
    apply (*subst word-uint.Abs-norm* [*symmetric*], *subst mod-add-self2*)
    apply *simp*
    done
qed

lemmas *word-of-int-inj* =
  word-uint.Abs-inject [*unfolded uints-num*, *simplified*]

lemma *word-le-less-eq*:
  (*x* ::'*z*::*len word*) ≤ *y* = (*x* = *y* ∨ *x* < *y*)
  by (*auto simp add*: *order-class.le-less*)

lemma *mod-plus-cong*:
  assumes *1*: (*b*::*int*) = *b'*
    and *2*: *x mod b'* = *x' mod b'*
    and *3*: *y mod b'* = *y' mod b'*
    and *4*: *x'* + *y'* = *z'*
  shows (*x* + *y*) *mod b* = *z' mod b'*
proof −
  from *1 2*[*symmetric*] *3*[*symmetric*] **have** (*x* + *y*) *mod b* = (*x' mod b'* + *y' mod b'*) *mod b'*
    by (*simp add*: *mod-add-eq*[*symmetric*])
  also have ... = (*x'* + *y'*) *mod b'*
    by (*simp add*: *mod-add-eq*[*symmetric*])
  finally show *?thesis* **by** (*simp add*: *4*)
qed

lemma *mod-minus-cong*:
  assumes *1*: (*b*::*int*) = *b'*
    and *2*: *x mod b'* = *x' mod b'*
    and *3*: *y mod b'* = *y' mod b'*
    and *4*: *x'* − *y'* = *z'*
  shows (*x* − *y*) *mod b* = *z' mod b'*
  using *assms*
  apply (*subst mod-diff-left-eq*)
  apply (*subst mod-diff-right-eq*)
  apply (*simp add*: *mod-diff-left-eq* [*symmetric*] *mod-diff-right-eq* [*symmetric*])
  done

lemma *word-induct-less*:

$[\![P\ (0::'a::len\ word);\ \bigwedge n.\ [\![n < m;\ P\ n]\!] \Longrightarrow P\ (1 + n)]\!] \Longrightarrow P\ m$
apply (*cases m*)
apply *atomize*
apply (*erule rev-mp*)+
apply (*rule-tac x=m* **in** *spec*)
apply (*induct-tac n*)
 apply *simp*
apply *clarsimp*
apply (*erule impE*)
 apply *clarsimp*
 apply (*erule-tac x=n* **in** *allE*)
 apply (*erule impE*)
  apply (*simp add*: *unat-arith-simps*)
  apply (*clarsimp simp*: *unat-of-nat*)
 apply *simp*
apply (*erule-tac x=of-nat na* **in** *allE*)
apply (*erule impE*)
 apply (*simp add*: *unat-arith-simps*)
 apply (*clarsimp simp*: *unat-of-nat*)
apply *simp*
done

lemma *word-induct*:
$[\![P\ (0::'a::len\ word);\ \bigwedge n.\ P\ n \Longrightarrow P\ (1 + n)]\!] \Longrightarrow P\ m$
by (*erule word-induct-less*, *simp*)

lemma *word-induct2* [*induct type*]:
$[\![P\ 0;\ \bigwedge n.\ [\![1 + n \neq 0;\ P\ n]\!] \Longrightarrow P\ (1 + n)]\!] \Longrightarrow P\ (n::'b::len\ word)$
apply (*rule word-induct*, *simp*)
apply (*case-tac 1+n = 0*, *auto*)
done

127.29 Recursion combinator for words

definition *word-rec* :: $'a \Rightarrow ('b::len\ word \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'b\ word \Rightarrow 'a$
where
word-rec forZero forSuc n = *rec-nat forZero* (*forSuc* ∘ *of-nat*) (*unat n*)

lemma *word-rec-0*: *word-rec z s 0* = *z*
by (*simp add*: *word-rec-def*)

lemma *word-rec-Suc*:
$1 + n \neq (0::'a::len\ word) \Longrightarrow$ *word-rec z s* (*1* + *n*) = *s n* (*word-rec z s n*)
apply (*simp add*: *word-rec-def unat-word-ariths*)
apply (*subst nat-mod-eq'*)
apply (*metis Suc-eq-plus1-left Suc-lessI of-nat-2p unat-1 unat-lt2p word-arith-nat-add*)
apply *simp*
done

lemma *word-rec-Pred*:
n ≠ *0* ⟹ *word-rec z s n* = *s* (*n* − *1*) (*word-rec z s* (*n* − *1*))
apply (*rule subst*[**where** *t*=*n* **and** *s*=*1* + (*n* − *1*)])
apply *simp*
apply (*subst word-rec-Suc*)
apply *simp*
apply *simp*
done

lemma *word-rec-in*:
f (*word-rec z* (λ-. *f*) *n*) = *word-rec* (*f z*) (λ-. *f*) *n*
by (*induct n*) (*simp-all add*: *word-rec-0 word-rec-Suc*)

lemma *word-rec-in2*:
f n (*word-rec z f n*) = *word-rec* (*f 0 z*) (*f* ∘ *op* + *1*) *n*
by (*induct n*) (*simp-all add*: *word-rec-0 word-rec-Suc*)

lemma *word-rec-twice*:
m ≤ *n* ⟹ *word-rec z f n* = *word-rec* (*word-rec z f* (*n* − *m*)) (*f* ∘ *op* + (*n* − *m*)) *m*
apply (*erule rev-mp*)
apply (*rule-tac x*=*z* **in** *spec*)
apply (*rule-tac x*=*f* **in** *spec*)
apply (*induct n*)
apply (*simp add*: *word-rec-0*)
apply *clarsimp*
apply (*rule-tac t*=*1* + *n* − *m* **and** *s*=*1* + (*n* − *m*) **in** *subst*)
apply *simp*
apply (*case-tac 1* + (*n* − *m*) = *0*)
apply (*simp add*: *word-rec-0*)
apply (*rule-tac f* = *word-rec a b* **for** *a b* **in** *arg-cong*)
apply (*rule-tac t*=*m* **and** *s*=*m* + (*1* + (*n* − *m*)) **in** *subst*)
apply *simp*
apply (*simp* (*no-asm-use*))
apply (*simp add*: *word-rec-Suc word-rec-in2*)
apply (*erule impE*)
apply *uint-arith*
apply (*drule-tac x*=*x* ∘ *op* + *1* **in** *spec*)
apply (*drule-tac x*=*x 0 xa* **in** *spec*)
apply *simp*
apply (*rule-tac t*=λ*a*. *x* (*1* + (*n* − *m* + *a*)) **and** *s*=λ*a*. *x* (*1* + (*n* − *m*) + *a*) **in** *subst*)
apply (*clarsimp simp add*: *fun-eq-iff*)
apply (*rule-tac t*=(*1* + (*n* − *m* + *xb*)) **and** *s*=*1* + (*n* − *m*) + *xb* **in** *subst*)
apply *simp*
apply (*rule refl*)
apply (*rule refl*)
done

lemma *word-rec-id*: *word-rec z* (λ-. *id*) *n* = *z*
  by (*induct n*) (*auto simp add*: *word-rec-0 word-rec-Suc*)

lemma *word-rec-id-eq*: ∀ *m* < *n*. *f m* = *id* ⟹ *word-rec z f n* = *z*
apply (*erule rev-mp*)
apply (*induct n*)
 apply (*auto simp add*: *word-rec-0 word-rec-Suc*)
 apply (*drule spec*, *erule mp*)
 apply *uint-arith*
apply (*drule-tac x*=*n* **in** *spec*, *erule impE*)
 apply *uint-arith*
apply *simp*
done

lemma *word-rec-max*:
  ∀ *m*≥*n*. *m* ≠ − *1* ⟶ *f m* = *id* ⟹ *word-rec z f* (− *1*) = *word-rec z f n*
apply (*subst word-rec-twice*[**where** *n*=−*1* **and** *m*=−*1* − *n*])
 apply *simp*
apply *simp*
apply (*rule word-rec-id-eq*)
apply *clarsimp*
apply (*drule spec*, *rule mp*, *erule mp*)
 apply (*rule word-plus-mono-right2*[*OF - order-less-imp-le*])
  prefer *2*
  apply *assumption*
 apply *simp*
apply (*erule contrapos-pn*)
apply *simp*
apply (*drule arg-cong*[**where** *f*=λ*x*. *x* − *n*])
apply *simp*
done

lemma *unatSuc*:
  1 + *n* ≠ (*0*::′*a*::*len word*) ⟹ *unat* (*1* + *n*) = *Suc* (*unat n*)
  by *unat-arith*

declare *bin-to-bl-def* [*simp*]

ML-file *Tools/word-lib.ML*
ML-file *Tools/smt-word.ML*

hide-const (**open**) *Word*

end

128 Old Version of Bindings to Satisfiability Modulo Theories (SMT) solvers

theory *Old-SMT*
imports *../Real ../Word/Word*
keywords *old-smt-status* :: *diag*
begin

ML-file *Old-SMT/old-smt-utils.ML*
ML-file *Old-SMT/old-smt-failure.ML*
ML-file *Old-SMT/old-smt-config.ML*

128.1 Triggers for quantifier instantiation

Some SMT solvers support patterns as a quantifier instantiation heuristics. Patterns may either be positive terms (tagged by "pat") triggering quantifier instantiations – when the solver finds a term matching a positive pattern, it instantiates the corresponding quantifier accordingly – or negative terms (tagged by "nopat") inhibiting quantifier instantiations. A list of patterns of the same kind is called a multipattern, and all patterns in a multipattern are considered conjunctively for quantifier instantiation. A list of multipatterns is called a trigger, and their multipatterns act disjunctively during quantifier instantiation. Each multipattern should mention at least all quantified variables of the preceding quantifier block.

typedecl *pattern*

consts
  pat :: $'a \Rightarrow$ *pattern*
  nopat :: $'a \Rightarrow$ *pattern*

definition *trigger* :: *pattern list list* $\Rightarrow$ *bool* $\Rightarrow$ *bool* **where** *trigger* - $P = P$

128.2 Quantifier weights

Weight annotations to quantifiers influence the priority of quantifier instantiations. They should be handled with care for solvers, which support them, because incorrect choices of weights might render a problem unsolvable.

definition *weight* :: *int* $\Rightarrow$ *bool* $\Rightarrow$ *bool* **where** *weight* - $P = P$

Weights must be non-negative. The value *0* is equivalent to providing no weight at all.

Weights should only be used at quantifiers and only inside triggers (if the quantifier has triggers). Valid usages of weights are as follows:

- $\forall x.$ *trigger* [[*pat* (P x)]] (*weight 2* (P x))
- $\forall x.$ *weight 3* (P x)

128.3 Higher-order encoding

Application is made explicit for constants occurring with varying numbers of arguments. This is achieved by the introduction of the following constant.

definition *fun-app* **where** *fun-app f = f*

Some solvers support a theory of arrays which can be used to encode higher-order functions. The following set of lemmas specifies the properties of such (extensional) arrays.

lemmas *array-rules = ext fun-upd-apply fun-upd-same fun-upd-other fun-upd-upd fun-app-def*

128.4 First-order logic

Some SMT solvers only accept problems in first-order logic, i.e., where formulas and terms are syntactically separated. When translating higher-order into first-order problems, all uninterpreted constants (those not built-in in the target solver) are treated as function symbols in the first-order sense. Their occurrences as head symbols in atoms (i.e., as predicate symbols) are turned into terms by logically equating such atoms with *True*. For technical reasons, *True* and *False* occurring inside terms are replaced by the following constants.

definition *term-true* **where** *term-true = True*
definition *term-false* **where** *term-false = False*

128.5 Integer division and modulo for Z3

definition *z3div* :: *int* ⇒ *int* ⇒ *int* **where**
z3div k l = (*if* 0 ≤ *l then k div l else* −(*k div* (−*l*)))

definition *z3mod* :: *int* ⇒ *int* ⇒ *int* **where**
z3mod k l = (*if* 0 ≤ *l then k mod l else k mod* (−*l*))

128.6 Setup

ML-file *Old-SMT/old-smt-builtin.ML*
ML-file *Old-SMT/old-smt-datatypes.ML*
ML-file *Old-SMT/old-smt-normalize.ML*
ML-file *Old-SMT/old-smt-translate.ML*
ML-file *Old-SMT/old-smt-solver.ML*
ML-file *Old-SMT/old-smtlib-interface.ML*
ML-file *Old-SMT/old-z3-interface.ML*
ML-file *Old-SMT/old-z3-proof-parser.ML*
ML-file *Old-SMT/old-z3-proof-tools.ML*
ML-file *Old-SMT/old-z3-proof-literals.ML*
ML-file *Old-SMT/old-z3-proof-methods.ML*
named-theorems *old-z3-simp simplification rules for Z3 proof reconstruction*
ML-file *Old-SMT/old-z3-proof-reconstruction.ML*

ML-file *Old-SMT/old-z3-model.ML*
ML-file *Old-SMT/old-smt-setup-solvers.ML*

setup ‹
  Old-SMT-Config.setup #>
  Old-SMT-Normalize.setup #>
  Old-SMTLIB-Interface.setup #>
  Old-Z3-Interface.setup #>
  Old-SMT-Setup-Solvers.setup
›

method-setup *old-smt* = ‹
  Scan.optional Attrib.thms [] >>
    (*fn thms* => *fn ctxt* =>
      METHOD (*fn facts* => *HEADGOAL* (*Old-SMT-Solver.smt-tac ctxt* (*thms* @ *facts*))))
› *apply an SMT solver to the current goal*

128.7 Configuration

The current configuration can be printed by the command *old-smt-status*, which shows the values of most options.

128.8 General configuration options

The option *old-smt-solver* can be used to change the target SMT solver. The possible values can be obtained from the *old-smt-status* command.

Due to licensing restrictions, Yices and Z3 are not installed/enabled by default. Z3 is free for non-commercial applications and can be enabled by setting the *OLD-Z3-NON-COMMERCIAL* environment variable to *yes*.

declare [[*old-smt-solver* = *z3*]]

Since SMT solvers are potentially non-terminating, there is a timeout (given in seconds) to restrict their runtime. A value greater than 120 (seconds) is in most cases not advisable.

declare [[*old-smt-timeout* = *20*]]

SMT solvers apply randomized heuristics. In case a problem is not solvable by an SMT solver, changing the following option might help.

declare [[*old-smt-random-seed* = *1*]]

In general, the binding to SMT solvers runs as an oracle, i.e, the SMT solvers are fully trusted without additional checks. The following option can cause the SMT solver to run in proof-producing mode, giving a checkable certificate. This is currently only implemented for Z3.

declare [[*old-smt-oracle* = *false*]]

Each SMT solver provides several commandline options to tweak its behaviour. They can be passed to the solver by setting the following options.

declare [[*old-cvc3-options* =]]
declare [[*old-yices-options* =]]
declare [[*old-z3-options* =]]

Enable the following option to use built-in support for datatypes and records. Currently, this is only implemented for Z3 running in oracle mode.

declare [[*old-smt-datatypes* = *false*]]

The SMT method provides an inference mechanism to detect simple triggers in quantified formulas, which might increase the number of problems solvable by SMT solvers (note: triggers guide quantifier instantiations in the SMT solver). To turn it on, set the following option.

declare [[*old-smt-infer-triggers* = *false*]]

The SMT method monomorphizes the given facts, that is, it tries to instantiate all schematic type variables with fixed types occurring in the problem. This is a (possibly nonterminating) fixed-point construction whose cycles are limited by the following option.

declare [[*monomorph-max-rounds* = *5*]]

In addition, the number of generated monomorphic instances is limited by the following option.

declare [[*monomorph-max-new-instances* = *500*]]

128.9 Certificates

By setting the option *old-smt-certificates* to the name of a file, all following applications of an SMT solver a cached in that file. Any further application of the same SMT solver (using the very same configuration) re-uses the cached certificate instead of invoking the solver. An empty string disables caching certificates.

The filename should be given as an explicit path. It is good practice to use the name of the current theory (with ending *.certs* instead of *.thy*) as the certificates file. Certificate files should be used at most once in a certain theory context, to avoid race conditions with other concurrent accesses.

declare [[*old-smt-certificates* =]]

The option *old-smt-read-only-certificates* controls whether only stored certificates are should be used or invocation of an SMT solver is allowed. When set to *true*, no SMT solver will ever be invoked and only the existing certificates found in the configured cache are used; when set to *false* and there is no cached certificate for some proposition, then the configured SMT solver is invoked.

declare [[*old-smt-read-only-certificates* = *false*]]

128.10 Tracing

The SMT method, when applied, traces important information. To make it entirely silent, set the following option to *false*.

declare [[*old-smt-verbose* = *true*]]

For tracing the generated problem file given to the SMT solver as well as the returned result of the solver, the option *old-smt-trace* should be set to *true*.

declare [[*old-smt-trace* = *false*]]

From the set of assumptions given to the SMT solver, those assumptions used in the proof are traced when the following option is set to *true*. This only works for Z3 when it runs in non-oracle mode (see options *old-smt-solver* and *old-smt-oracle* above).

declare [[*old-smt-trace-used-facts* = *false*]]

128.11 Schematic rules for Z3 proof reconstruction

Several prof rules of Z3 are not very well documented. There are two lemma groups which can turn failing Z3 proof reconstruction attempts into succeeding ones: the facts in *z3-rule* are tried prior to any implemented reconstruction procedure for all uncertain Z3 proof rules; the facts in *z3-simp* are only fed to invocations of the simplifier when reconstructing theory-specific proof steps.

lemmas [*old-z3-rule*] =
refl eq-commute conj-commute disj-commute simp-thms nnf-simps
ring-distribs field-simps times-divide-eq-right times-divide-eq-left
if-True if-False not-not

lemma [*old-z3-rule*]:
$(P \wedge Q) = (\neg(\neg P \vee \neg Q))$
$(P \wedge Q) = (\neg(\neg Q \vee \neg P))$
$(\neg P \wedge Q) = (\neg(P \vee \neg Q))$
$(\neg P \wedge Q) = (\neg(\neg Q \vee P))$
$(P \wedge \neg Q) = (\neg(\neg P \vee Q))$
$(P \wedge \neg Q) = (\neg(Q \vee \neg P))$
$(\neg P \wedge \neg Q) = (\neg(P \vee Q))$
$(\neg P \wedge \neg Q) = (\neg(Q \vee P))$
by *auto*

lemma [*old-z3-rule*]:
$(P \longrightarrow Q) = (Q \vee \neg P)$
$(\neg P \longrightarrow Q) = (P \vee Q)$
$(\neg P \longrightarrow Q) = (Q \vee P)$
$(\mathit{True} \longrightarrow P) = P$
$(P \longrightarrow \mathit{True}) = \mathit{True}$

$(False \longrightarrow P) = True$
$(P \longrightarrow P) = True$
by *auto*

lemma [*old-z3-rule*]:
$((P = Q) \longrightarrow R) = (R \mid (Q = (\neg P)))$
by *auto*

lemma [*old-z3-rule*]:
$(\neg True) = False$
$(\neg False) = True$
$(x = x) = True$
$(P = True) = P$
$(True = P) = P$
$(P = False) = (\neg P)$
$(False = P) = (\neg P)$
$((\neg P) = P) = False$
$(P = (\neg P)) = False$
$((\neg P) = (\neg Q)) = (P = Q)$
$\neg(P = (\neg Q)) = (P = Q)$
$\neg((\neg P) = Q) = (P = Q)$
$(P \neq Q) = (Q = (\neg P))$
$(P = Q) = ((\neg P \lor Q) \land (P \lor \neg Q))$
$(P \neq Q) = ((\neg P \lor \neg Q) \land (P \lor Q))$
by *auto*

lemma [*old-z3-rule*]:
$(\textit{if } P \textit{ then } P \textit{ else } \neg P) = True$
$(\textit{if } \neg P \textit{ then } \neg P \textit{ else } P) = True$
$(\textit{if } P \textit{ then } True \textit{ else } False) = P$
$(\textit{if } P \textit{ then } False \textit{ else } True) = (\neg P)$
$(\textit{if } P \textit{ then } Q \textit{ else } True) = ((\neg P) \lor Q)$
$(\textit{if } P \textit{ then } Q \textit{ else } True) = (Q \lor (\neg P))$
$(\textit{if } P \textit{ then } Q \textit{ else } \neg Q) = (P = Q)$
$(\textit{if } P \textit{ then } Q \textit{ else } \neg Q) = (Q = P)$
$(\textit{if } P \textit{ then } \neg Q \textit{ else } Q) = (P = (\neg Q))$
$(\textit{if } P \textit{ then } \neg Q \textit{ else } Q) = ((\neg Q) = P)$
$(\textit{if } \neg P \textit{ then } x \textit{ else } y) = (\textit{if } P \textit{ then } y \textit{ else } x)$
$(\textit{if } P \textit{ then } (\textit{if } Q \textit{ then } x \textit{ else } y) \textit{ else } x) = (\textit{if } P \land (\neg Q) \textit{ then } y \textit{ else } x)$
$(\textit{if } P \textit{ then } (\textit{if } Q \textit{ then } x \textit{ else } y) \textit{ else } x) = (\textit{if } (\neg Q) \land P \textit{ then } y \textit{ else } x)$
$(\textit{if } P \textit{ then } (\textit{if } Q \textit{ then } x \textit{ else } y) \textit{ else } y) = (\textit{if } P \land Q \textit{ then } x \textit{ else } y)$
$(\textit{if } P \textit{ then } (\textit{if } Q \textit{ then } x \textit{ else } y) \textit{ else } y) = (\textit{if } Q \land P \textit{ then } x \textit{ else } y)$
$(\textit{if } P \textit{ then } x \textit{ else if } P \textit{ then } y \textit{ else } z) = (\textit{if } P \textit{ then } x \textit{ else } z)$
$(\textit{if } P \textit{ then } x \textit{ else if } Q \textit{ then } x \textit{ else } y) = (\textit{if } P \lor Q \textit{ then } x \textit{ else } y)$
$(\textit{if } P \textit{ then } x \textit{ else if } Q \textit{ then } x \textit{ else } y) = (\textit{if } Q \lor P \textit{ then } x \textit{ else } y)$
$(\textit{if } P \textit{ then } x = y \textit{ else } x = z) = (x = (\textit{if } P \textit{ then } y \textit{ else } z))$
$(\textit{if } P \textit{ then } x = y \textit{ else } y = z) = (y = (\textit{if } P \textit{ then } x \textit{ else } z))$
$(\textit{if } P \textit{ then } x = y \textit{ else } z = y) = (y = (\textit{if } P \textit{ then } x \textit{ else } z))$
by *auto*

lemma [*old-z3-rule*]:
$0 + (x::int) = x$
$x + 0 = x$
$x + x = 2 * x$
$0 * x = 0$
$1 * x = x$
$x + y = y + x$
by *auto*

lemma [*old-z3-rule*]:
$P = Q \vee P \vee Q$
$P = Q \vee \neg P \vee \neg Q$
$(\neg P) = Q \vee \neg P \vee Q$
$(\neg P) = Q \vee P \vee \neg Q$
$P = (\neg Q) \vee \neg P \vee Q$
$P = (\neg Q) \vee P \vee \neg Q$
$P \neq Q \vee P \vee \neg Q$
$P \neq Q \vee \neg P \vee Q$
$P \neq (\neg Q) \vee P \vee Q$
$(\neg P) \neq Q \vee P \vee Q$
$P \vee Q \vee P \neq (\neg Q)$
$P \vee Q \vee (\neg P) \neq Q$
$P \vee \neg Q \vee P \neq Q$
$\neg P \vee Q \vee P \neq Q$
$P \vee y = (\mathit{if}\ P\ \mathit{then}\ x\ \mathit{else}\ y)$
$P \vee (\mathit{if}\ P\ \mathit{then}\ x\ \mathit{else}\ y) = y$
$\neg P \vee x = (\mathit{if}\ P\ \mathit{then}\ x\ \mathit{else}\ y)$
$\neg P \vee (\mathit{if}\ P\ \mathit{then}\ x\ \mathit{else}\ y) = x$
$P \vee R \vee \neg(\mathit{if}\ P\ \mathit{then}\ Q\ \mathit{else}\ R)$
$\neg P \vee Q \vee \neg(\mathit{if}\ P\ \mathit{then}\ Q\ \mathit{else}\ R)$
$\neg(\mathit{if}\ P\ \mathit{then}\ Q\ \mathit{else}\ R) \vee \neg P \vee Q$
$\neg(\mathit{if}\ P\ \mathit{then}\ Q\ \mathit{else}\ R) \vee P \vee R$
$(\mathit{if}\ P\ \mathit{then}\ Q\ \mathit{else}\ R) \vee \neg P \vee \neg Q$
$(\mathit{if}\ P\ \mathit{then}\ Q\ \mathit{else}\ R) \vee P \vee \neg R$
$(\mathit{if}\ P\ \mathit{then}\ \neg Q\ \mathit{else}\ R) \vee \neg P \vee Q$
$(\mathit{if}\ P\ \mathit{then}\ Q\ \mathit{else}\ \neg R) \vee P \vee R$
by *auto*

ML-file *Old-SMT/old-smt-real.ML*
ML-file *Old-SMT/old-smt-word.ML*

hide-type (**open**) *pattern*
hide-const *fun-app term-true term-false z3div z3mod*
hide-const (**open**) *trigger pat nopat weight*

end

References

[1] J. Avigad and K. Donnelly. Formalizing O notation in Isabelle/HOL. In D. Basin and M. Rusiowitch, editors, *Automated Reasoning: second international conference, IJCAR 2004*, pages 357–371. Springer, 2004.

[2] A. Podelski and A. Rybalchenko. Transition invariants. In *19th Annual IEEE Symposium on Logic in Computer Science (LICS'04)*, pages 32–41, 2004.