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* :: $'key \Rightarrow 'val \Rightarrow ('key \times 'val)\ list \Rightarrow ('key \times 'val)\ list$
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'*: $map\text{-}of\ (update\ k\ v\ al) = (map\text{-}of\ al)(k \mapsto v)$
  ⟨*proof*⟩

corollary *update-conv*: $map\text{-}of\ (update\ k\ v\ al)\ k' = ((map\text{-}of\ al)(k \mapsto v))\ k'$
  ⟨*proof*⟩

lemma *dom-update*: $fst\ `\ set\ (update\ k\ v\ al) = \{k\} \cup fst\ `\ set\ al$
  ⟨*proof*⟩

lemma *update-keys*:
  $map\ fst\ (update\ k\ v\ al) =$
    $(if\ k \in set\ (map\ fst\ al)\ then\ map\ fst\ al\ else\ map\ fst\ al\ @\ [k])$
  ⟨*proof*⟩

lemma *distinct-update*:
  assumes $distinct\ (map\ fst\ al)$
  shows $distinct\ (map\ fst\ (update\ k\ v\ al))$
  ⟨*proof*⟩

lemma *update-filter*:
  $a \neq k \Longrightarrow update\ k\ v\ [q \leftarrow ps.\ fst\ q \neq a] = [q \leftarrow update\ k\ v\ ps.\ fst\ q \neq a]$
  ⟨*proof*⟩

lemma *update-triv*: $map\text{-}of\ al\ k = Some\ v \Longrightarrow update\ k\ v\ al = al$
  ⟨*proof*⟩

lemma *update-nonempty* [*simp*]: $update\ k\ v\ al \neq []$
  ⟨*proof*⟩

lemma *update-eqD*: *update k v al* = *update k v' al'* ⟹ *v* = *v'*
⟨*proof*⟩

lemma *update-last* [*simp*]: *update k v* (*update k v' al*) = *update k v al*
⟨*proof*⟩

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*))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *image-update* [*simp*]:
x ∉ *A* ⟹ *map-of* (*update x y al*) ‘ *A* = *map-of al* ‘ *A*
⟨*proof*⟩ **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*)
⟨*proof*⟩

lemma *updates-key-simp* [*simp*]:
updates (*k* # *ks*) *vs ps* =
(*case vs of* [] ⇒ *ps* | *v* # *vs* ⇒ *updates ks vs* (*update k v ps*))
⟨*proof*⟩

lemma *updates-conv'*: *map-of* (*updates ks vs al*) = (*map-of al*)(*ks*[↦]*vs*)
⟨*proof*⟩

lemma *updates-conv*: *map-of* (*updates ks vs al*) *k* = ((*map-of al*)(*ks*[↦]*vs*)) *k*
⟨*proof*⟩

lemma *distinct-updates*:
assumes *distinct* (*map fst al*)
shows *distinct* (*map fst* (*updates ks vs al*))
⟨*proof*⟩

lemma *updates-append1*[*simp*]: *size ks* < *size vs* ⟹
updates (*ks*@[*k*]) *vs al* = *update k* (*vs*!*size ks*) (*updates ks vs al*)
⟨*proof*⟩

lemma *updates-list-update-drop*[*simp*]:
size ks ≤ *i* ⟹ *i* < *size vs* ⟹
updates ks (*vs*[*i*:=*v*]) *al* = *updates ks vs al*
⟨*proof*⟩

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*)))
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *updates-apply-notin* [*simp*]:
k ∉ *set ks* ⟹ *map-of* (*updates ks vs al*) *k* = *map-of al k*
⟨*proof*⟩

lemma *updates-append-drop* [*simp*]:
size xs = *size ys* ⟹ *updates* (*xs* @ *zs*) *ys al* = *updates xs ys al*
⟨*proof*⟩

lemma *updates-append2-drop* [*simp*]:
size xs = *size ys* ⟹ *updates xs* (*ys* @ *zs*) *al* = *updates xs ys al*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *delete-conv'*: *map-of* (*delete k al*) = (*map-of al*)(*k* := *None*)
⟨*proof*⟩

corollary *delete-conv*: *map-of* (*delete k al*) *k'* = ((*map-of al*)(*k* := *None*)) *k'*
⟨*proof*⟩

lemma *delete-keys*: *map fst* (*delete k al*) = *removeAll k* (*map fst al*)
⟨*proof*⟩

lemma *distinct-delete*:
assumes *distinct* (*map fst al*)
shows *distinct* (*map fst* (*delete k al*))
⟨*proof*⟩

lemma *delete-id* [*simp*]: $k \notin fst\ `\ set\ al \Longrightarrow delete\ k\ al = al$
⟨*proof*⟩

lemma *delete-idem*: *delete k* (*delete k al*) = *delete k al*
⟨*proof*⟩

lemma *map-of-delete* [*simp*]: $k' \neq k \Longrightarrow map\text{-}of\ (delete\ k\ al)\ k' = map\text{-}of\ al\ k'$
⟨*proof*⟩

lemma *delete-notin-dom*: $k \notin fst\ `\ set\ (delete\ k\ al)$
⟨*proof*⟩

lemma *dom-delete-subset*: $fst\ `\ set\ (delete\ k\ al) \subseteq fst\ `\ set\ al$
⟨*proof*⟩

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

lemma *delete-update*: $k \neq l \Longrightarrow delete\ l\ (update\ k\ v\ al) = update\ k\ v\ (delete\ l\ al)$
⟨*proof*⟩

lemma *delete-twist*: *delete x* (*delete y al*) = *delete y* (*delete x al*)
⟨*proof*⟩

lemma *length-delete-le*: *length* (*delete k al*) ≤ *length al*
⟨*proof*⟩

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

qualified primrec *update-with-aux* :: *'val* ⇒ *'key* ⇒ (*'val* ⇒ *'val*) ⇒ (*'key* × *'val*) *list* ⇒ (*'key* × *'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* ⇒ (*'key* × *'val*) *list* ⇒ (*'key* × *'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* ↦ (*case map-of ps k of None* ⇒ *f v* | *Some v* ⇒ *f v*))) *k'*
⟨*proof*⟩

lemma *map-of-update-with-aux*:
map-of (*update-with-aux v k f ps*) = (*map-of ps*)(*k* ↦ (*case map-of ps k of None* ⇒ *f v* | *Some v* ⇒ *f v*))
⟨*proof*⟩

lemma *dom-update-with-aux*: *fst* ‘ *set* (*update-with-aux v k f ps*) = {*k*} ∪ *fst* ‘ *set ps*
⟨*proof*⟩

lemma *distinct-update-with-aux* [*simp*]:
distinct (*map fst* (*update-with-aux v k f ps*)) = *distinct* (*map fst ps*)
⟨*proof*⟩

lemma *set-update-with-aux*:
distinct (*map fst xs*)
⟹ *set* (*update-with-aux v k f xs*) = (*set xs* − {*k*} × *UNIV* ∪ {(*k*, *f* (*case map-of xs k of None* ⇒ *v* | *Some v* ⇒ *v*))})
⟨*proof*⟩

lemma *set-delete-aux*: *distinct* (*map fst xs*) ⟹ *set* (*delete-aux k xs*) = *set xs* − {*k*} × *UNIV*
⟨*proof*⟩

lemma *dom-delete-aux*: *distinct* (*map fst ps*) ⟹ *fst* ‘ *set* (*delete-aux k ps*) = *fst* ‘ *set ps* − {*k*}
⟨*proof*⟩

lemma *distinct-delete-aux* [*simp*]:
distinct (*map fst ps*) ⟹ *distinct* (*map fst* (*delete-aux k ps*))
⟨*proof*⟩

lemma *map-of-delete-aux'*:
distinct (*map fst xs*) ⟹ *map-of* (*delete-aux k xs*) = (*map-of xs*)(*k* := *None*)
⟨*proof*⟩

lemma *map-of-delete-aux*:
distinct (*map fst xs*) ⟹ *map-of* (*delete-aux k xs*) *k'* = ((*map-of xs*)(*k* := *None*)) *k'*
⟨*proof*⟩

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

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*)
⟨*proof*⟩

lemma *restr-conv'*: *map-of* (*restrict A al*) = ((*map-of al*)|‘ *A*)
⟨*proof*⟩

corollary *restr-conv*: *map-of* (*restrict A al*) *k* = ((*map-of al*)|‘ *A*) *k*
⟨*proof*⟩

lemma *distinct-restr*:
distinct (*map fst al*) ⟹ *distinct* (*map fst* (*restrict A al*))
⟨*proof*⟩

lemma *restr-empty* [*simp*]:
restrict {} *al* = []
restrict A [] = []
⟨*proof*⟩

lemma *restr-in* [*simp*]: *x* ∈ *A* ⟹ *map-of* (*restrict A al*) *x* = *map-of al x*
⟨*proof*⟩

lemma *restr-out* [*simp*]: *x* ∉ *A* ⟹ *map-of* (*restrict A al*) *x* = *None*
⟨*proof*⟩

lemma *dom-restr* [*simp*]: *fst* ‘ *set* (*restrict A al*) = *fst* ‘ *set al* ∩ *A*
⟨*proof*⟩

lemma *restr-upd-same* [*simp*]: *restrict* (−{*x*}) (*update x y al*) = *restrict* (−{*x*}) *al*
⟨*proof*⟩

lemma *restr-restr* [*simp*]: *restrict A* (*restrict B al*) = *restrict* (*A*∩*B*) *al*
⟨*proof*⟩

lemma *restr-update*[*simp*]:
map-of (*restrict D* (*update x y al*)) =
map-of ((*if x* ∈ *D then* (*update x y* (*restrict* (*D*−{*x*}) *al*)) *else restrict D al*))
⟨*proof*⟩

lemma *restr-delete* [*simp*]:
delete x (*restrict D al*) = (*if x* ∈ *D then restrict* (*D* − {*x*}) *al else restrict D al*)
⟨*proof*⟩

lemma *update-restr*:
map-of (*update x y* (*restrict D al*)) = *map-of* (*update x y* (*restrict* (*D* − {*x*}) *al*))
⟨*proof*⟩

lemma *update-restr-conv* [*simp*]:
x ∈ *D* ⟹
map-of (*update x y* (*restrict D al*)) = *map-of* (*update x y* (*restrict* (*D* − {*x*}) *al*))
⟨*proof*⟩

lemma *restr-updates* [*simp*]:
length xs = *length ys* ⟹ *set xs* ⊆ *D* ⟹
map-of (*restrict D* (*updates xs ys al*)) =
map-of (*updates xs ys* (*restrict* (*D* − *set xs*) *al*))
⟨*proof*⟩

lemma *restr-delete-twist*: (*restrict A* (*delete a ps*)) = *delete a* (*restrict A ps*)
⟨*proof*⟩

1.5 *clearjunk*

qualified function *clearjunk* :: ('*key* × '*val*) *list* ⇒ ('*key* × '*val*) *list*
where
clearjunk [] = []
| *clearjunk* (*p*#*ps*) = *p* # *clearjunk* (*delete* (*fst p*) *ps*)
⟨*proof*⟩
termination
⟨*proof*⟩

lemma *map-of-clearjunk*: *map-of* (*clearjunk al*) = *map-of al*
⟨*proof*⟩

lemma *clearjunk-keys-set*: *set* (*map fst* (*clearjunk al*)) = *set* (*map fst al*)
⟨*proof*⟩

lemma *dom-clearjunk*: *fst* ‘ *set* (*clearjunk al*) = *fst* ‘ *set al*
⟨*proof*⟩

lemma *distinct-clearjunk* [*simp*]: *distinct* (*map fst* (*clearjunk al*))
⟨*proof*⟩

lemma *ran-clearjunk*: *ran* (*map-of* (*clearjunk al*)) = *ran* (*map-of al*)
⟨*proof*⟩

lemma *ran-map-of*: *ran* (*map-of al*) = *snd* ‘ *set* (*clearjunk al*)
⟨*proof*⟩

lemma *clearjunk-update*: *clearjunk* (*update k v al*) = *update k v* (*clearjunk al*)

⟨*proof*⟩

lemma *clearjunk-updates*: *clearjunk* (*updates ks vs al*) = *updates ks vs* (*clearjunk al*)
⟨*proof*⟩

lemma *clearjunk-delete*: *clearjunk* (*delete x al*) = *delete x* (*clearjunk al*)
⟨*proof*⟩

lemma *clearjunk-restrict*: *clearjunk* (*restrict A al*) = *restrict A* (*clearjunk al*)
⟨*proof*⟩

lemma *distinct-clearjunk-id* [*simp*]: *distinct* (*map fst al*) ⟹ *clearjunk al* = *al*
⟨*proof*⟩

lemma *clearjunk-idem*: *clearjunk* (*clearjunk al*) = *clearjunk al*
⟨*proof*⟩

lemma *length-clearjunk*: *length* (*clearjunk al*) ≤ *length al*
⟨*proof*⟩

lemma *delete-map*:
assumes ⋀*kv*. *fst* (*f kv*) = *fst kv*
shows *delete k* (*map f ps*) = *map f* (*delete k ps*)
⟨*proof*⟩

lemma *clearjunk-map*:
assumes ⋀*kv*. *fst* (*f kv*) = *fst kv*
shows *clearjunk* (*map f ps*) = *map f* (*clearjunk ps*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *dom-map-ran*: *fst* ‘ *set* (*map-ran f al*) = *fst* ‘ *set al*
⟨*proof*⟩

lemma *map-ran-conv*: *map-of* (*map-ran f al*) *k* = *map-option* (*f k*) (*map-of al k*)
⟨*proof*⟩

lemma *distinct-map-ran*: *distinct* (*map fst al*) ⟹ *distinct* (*map fst* (*map-ran f*

al))
⟨*proof*⟩

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

lemma *clearjunk-map-ran*: *clearjunk* (*map-ran f al*) = *map-ran f* (*clearjunk al*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *merge-updates*: *merge qs ps* = *updates* (*rev* (*map fst ps*)) (*rev* (*map snd ps*)) *qs*
⟨*proof*⟩

lemma *dom-merge*: *fst* ‘ *set* (*merge xs ys*) = *fst* ‘ *set xs* ∪ *fst* ‘ *set ys*
⟨*proof*⟩

lemma *distinct-merge*:
assumes *distinct* (*map fst xs*)
shows *distinct* (*map fst* (*merge xs ys*))
⟨*proof*⟩

lemma *clearjunk-merge*: *clearjunk* (*merge xs ys*) = *merge* (*clearjunk xs*) *ys*
⟨*proof*⟩

lemma *merge-conv'*: *map-of* (*merge xs ys*) = *map-of xs* ++ *map-of ys*
⟨*proof*⟩

corollary *merge-conv*: *map-of* (*merge xs ys*) *k* = (*map-of xs* ++ *map-of ys*) *k*
⟨*proof*⟩

lemma *merge-empty*: *map-of* (*merge* [] *ys*) = *map-of ys*
⟨*proof*⟩

lemma *merge-assoc* [*simp*]: *map-of* (*merge m1* (*merge m2 m3*)) = *map-of* (*merge* (*merge m1 m2*) *m3*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *merge-None* [*iff*]:
(*map-of* (*merge m n*) *k* = *None*) = (*map-of n k* = *None* ∧ *map-of m k* = *None*)
⟨*proof*⟩

lemma *merge-upd* [*simp*]:
map-of (*merge m* (*update k v n*)) = *map-of* (*update k v* (*merge m n*))
⟨*proof*⟩

lemma *merge-updatess* [*simp*]:
map-of (*merge m* (*updates xs ys n*)) = *map-of* (*updates xs ys* (*merge m n*))
⟨*proof*⟩

lemma *merge-append*: *map-of* (*xs* @ *ys*) = *map-of* (*merge ys xs*)
⟨*proof*⟩

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*)
⟨*proof*⟩
termination
⟨*proof*⟩

lemma *compose-first-None* [*simp*]:
assumes *map-of xs k* = *None*
shows *map-of* (*compose xs ys*) *k* = *None*
⟨*proof*⟩

lemma *compose-conv*: *map-of* (*compose xs ys*) *k* = (*map-of ys* ∘$_m$ *map-of xs*) *k*
⟨*proof*⟩

lemma *compose-conv'*: *map-of* (*compose xs ys*) = (*map-of ys* ∘$_m$ *map-of xs*)
⟨*proof*⟩

lemma *compose-first-Some* [*simp*]:
assumes *map-of xs k = Some v*
shows *map-of* (*compose xs ys*) *k = map-of ys v*
⟨*proof*⟩

lemma *dom-compose*: *fst ' set* (*compose xs ys*) ⊆ *fst ' set xs*
⟨*proof*⟩

lemma *distinct-compose*:
assumes *distinct* (*map fst xs*)
shows *distinct* (*map fst* (*compose xs ys*))
⟨*proof*⟩

lemma *compose-delete-twist*: *compose* (*delete k xs*) *ys = delete k* (*compose xs ys*)
⟨*proof*⟩

lemma *compose-clearjunk*: *compose xs* (*clearjunk ys*) = *compose xs ys*
⟨*proof*⟩

lemma *clearjunk-compose*: *clearjunk* (*compose xs ys*) = *compose* (*clearjunk xs*) *ys*
⟨*proof*⟩

lemma *compose-empty* [*simp*]: *compose xs* [] = []
⟨*proof*⟩

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*)
⟨*proof*⟩

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*))
⟨*proof*⟩

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'*))
⟨*proof*⟩

lemma *dom-map-entry*: *fst ' set (map-entry k f xs) = fst ' set xs*
⟨*proof*⟩

lemma *distinct-map-entry*:
assumes *distinct (map fst xs)*
shows *distinct (map fst (map-entry k f xs))*
⟨*proof*⟩

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'))
⟨*proof*⟩

lemma *dom-map-default*: *fst ' set (map-default k v f xs) = insert k (fst ' set xs)*
⟨*proof*⟩

lemma *distinct-map-default*:
assumes *distinct (map fst xs)*
shows *distinct (map fst (map-default k v f xs))*
⟨*proof*⟩

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 = (λx. f x + g x)*

instance ⟨*proof*⟩

end

lemma *plus-fun-apply* [*simp*]:
(*f* + *g*) *x* = *f x* + *g x*
⟨*proof*⟩

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

definition *0* = (λ*x*. *0*)
instance ⟨*proof*⟩

end

lemma *zero-fun-apply* [*simp*]:
0 x = *0*
⟨*proof*⟩

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

definition *f* ∗ *g* = (λ*x*. *f x* ∗ *g x*)
instance ⟨*proof*⟩

end

lemma *times-fun-apply* [*simp*]:
(*f* ∗ *g*) *x* = *f x* ∗ *g x*
⟨*proof*⟩

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

definition *1* = (λ*x*. *1*)
instance ⟨*proof*⟩

end

lemma *one-fun-apply* [*simp*]:
1 x = *1*
⟨*proof*⟩

Additive structures

instance *fun* :: (*type*, *semigroup-add*) *semigroup-add*
⟨*proof*⟩

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

⟨*proof*⟩

instance *fun* :: (*type*, *ab-semigroup-add*) *ab-semigroup-add*
⟨*proof*⟩

instance *fun* :: (*type*, *cancel-ab-semigroup-add*) *cancel-ab-semigroup-add*
⟨*proof*⟩

instance *fun* :: (*type*, *monoid-add*) *monoid-add*
⟨*proof*⟩

instance *fun* :: (*type*, *comm-monoid-add*) *comm-monoid-add*
⟨*proof*⟩

instance *fun* :: (*type*, *cancel-comm-monoid-add*) *cancel-comm-monoid-add* ⟨*proof*⟩

instance *fun* :: (*type*, *group-add*) *group-add*
⟨*proof*⟩

instance *fun* :: (*type*, *ab-group-add*) *ab-group-add*
⟨*proof*⟩

Multiplicative structures

instance *fun* :: (*type*, *semigroup-mult*) *semigroup-mult*
⟨*proof*⟩

instance *fun* :: (*type*, *ab-semigroup-mult*) *ab-semigroup-mult*
⟨*proof*⟩

instance *fun* :: (*type*, *monoid-mult*) *monoid-mult*
⟨*proof*⟩

instance *fun* :: (*type*, *comm-monoid-mult*) *comm-monoid-mult*
⟨*proof*⟩

Misc

instance *fun* :: (*type*, *Rings.dvd*) *Rings.dvd* ⟨*proof*⟩

instance *fun* :: (*type*, *mult-zero*) *mult-zero*
⟨*proof*⟩

instance *fun* :: (*type*, *zero-neq-one*) *zero-neq-one*
⟨*proof*⟩

Ring structures

instance *fun* :: (*type*, *semiring*) *semiring*
⟨*proof*⟩

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

⟨*proof*⟩

instance *fun* :: (*type*, *semiring-0*) *semiring-0* ⟨*proof*⟩

instance *fun* :: (*type*, *comm-semiring-0*) *comm-semiring-0* ⟨*proof*⟩

instance *fun* :: (*type*, *semiring-0-cancel*) *semiring-0-cancel* ⟨*proof*⟩

instance *fun* :: (*type*, *comm-semiring-0-cancel*) *comm-semiring-0-cancel* ⟨*proof*⟩

instance *fun* :: (*type*, *semiring-1*) *semiring-1* ⟨*proof*⟩

lemma *of-nat-fun*: *of-nat n* = (λ*x*::′*a*. *of-nat n*)
⟨*proof*⟩

lemma *of-nat-fun-apply* [*simp*]:
of-nat n x = *of-nat n*
⟨*proof*⟩

instance *fun* :: (*type*, *comm-semiring-1*) *comm-semiring-1* ⟨*proof*⟩

instance *fun* :: (*type*, *semiring-1-cancel*) *semiring-1-cancel* ⟨*proof*⟩

instance *fun* :: (*type*, *comm-semiring-1-cancel*) *comm-semiring-1-cancel*
⟨*proof*⟩

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

instance *fun* :: (*type*, *ring*) *ring* ⟨*proof*⟩

instance *fun* :: (*type*, *comm-ring*) *comm-ring* ⟨*proof*⟩

instance *fun* :: (*type*, *ring-1*) *ring-1* ⟨*proof*⟩

instance *fun* :: (*type*, *comm-ring-1*) *comm-ring-1* ⟨*proof*⟩

instance *fun* :: (*type*, *ring-char-0*) *ring-char-0* ⟨*proof*⟩

Ordered structures

instance *fun* :: (*type*, *ordered-ab-semigroup-add*) *ordered-ab-semigroup-add*
⟨*proof*⟩

instance *fun* :: (*type*, *ordered-cancel-ab-semigroup-add*) *ordered-cancel-ab-semigroup-add*
⟨*proof*⟩

instance *fun* :: (*type*, *ordered-ab-semigroup-add-imp-le*) *ordered-ab-semigroup-add-imp-le*
⟨*proof*⟩

instance *fun* :: (*type*, *ordered-comm-monoid-add*) *ordered-comm-monoid-add* ⟨*proof*⟩

instance *fun* :: (*type*, *ordered-cancel-comm-monoid-add*) *ordered-cancel-comm-monoid-add* ⟨*proof*⟩

instance *fun* :: (*type*, *ordered-ab-group-add*) *ordered-ab-group-add* ⟨*proof*⟩

instance *fun* :: (*type*, *ordered-semiring*) *ordered-semiring* ⟨*proof*⟩

instance *fun* :: (*type*, *dioid*) *dioid* ⟨*proof*⟩

instance *fun* :: (*type*, *ordered-comm-semiring*) *ordered-comm-semiring* ⟨*proof*⟩

instance *fun* :: (*type*, *ordered-cancel-semiring*) *ordered-cancel-semiring* ⟨*proof*⟩

instance *fun* :: (*type*, *ordered-cancel-comm-semiring*) *ordered-cancel-comm-semiring* ⟨*proof*⟩

instance *fun* :: (*type*, *ordered-ring*) *ordered-ring* ⟨*proof*⟩

instance *fun* :: (*type*, *ordered-comm-ring*) *ordered-comm-ring* ⟨*proof*⟩

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 ⟨*proof*⟩

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 ⟨*proof*⟩

end

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

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

instance ⟨*proof*⟩

end

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

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

instance ⟨*proof*⟩

end

definition *elt-set-plus* :: *'a::plus* ⇒ *'a set* ⇒ *'a set* (**infixl** *+o 70*) **where**
$a\ +o\ B = \{c.\ \exists\, b{\in}B.\ c = a + b\}$

definition *elt-set-times* :: *'a::times* ⇒ *'a set* ⇒ *'a set* (**infixl** *∗o 80*) **where**
$a\ *o\ B = \{c.\ \exists\, b{\in}B.\ c = a * b\}$

abbreviation (*input*) *elt-set-eq* :: *'a* ⇒ *'a set* ⇒ *bool* (**infix** *=o 50*) **where**
$x =o A \equiv x \in A$

instance *set* :: (*semigroup-add*) *semigroup-add*
⟨*proof*⟩

instance *set* :: (*ab-semigroup-add*) *ab-semigroup-add*
⟨*proof*⟩

instance *set* :: (*monoid-add*) *monoid-add*

⟨*proof*⟩

instance *set* :: (*comm-monoid-add*) *comm-monoid-add*
⟨*proof*⟩

instance *set* :: (*semigroup-mult*) *semigroup-mult*
⟨*proof*⟩

instance *set* :: (*ab-semigroup-mult*) *ab-semigroup-mult*
⟨*proof*⟩

instance *set* :: (*monoid-mult*) *monoid-mult*
⟨*proof*⟩

instance *set* :: (*comm-monoid-mult*) *comm-monoid-mult*
⟨*proof*⟩

lemma *set-plus-intro* [*intro*]: $a \in C \Longrightarrow b \in D \Longrightarrow a + b \in C + D$
⟨*proof*⟩

lemma *set-plus-elim*:
assumes $x \in A + B$
obtains a b **where** $x = a + b$ **and** $a \in A$ **and** $b \in B$
⟨*proof*⟩

lemma *set-plus-intro2* [*intro*]: $b \in C \Longrightarrow a + b \in a +o C$
⟨*proof*⟩

lemma *set-plus-rearrange*:
$((a::'a::comm\text{-}monoid\text{-}add) +o C) + (b +o D) = (a + b) +o (C + D)$
⟨*proof*⟩

lemma *set-plus-rearrange2*: $(a::'a::semigroup\text{-}add) +o (b +o C) = (a + b) +o C$
⟨*proof*⟩

lemma *set-plus-rearrange3*: $((a::'a::semigroup\text{-}add) +o B) + C = a +o (B + C)$
⟨*proof*⟩

theorem *set-plus-rearrange4*: $C + ((a::'a::comm\text{-}monoid\text{-}add) +o D) = a +o (C + D)$
⟨*proof*⟩

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

lemma *set-plus-mono* [*intro*!]: $C \subseteq D \Longrightarrow a +o C \subseteq a +o D$
⟨*proof*⟩

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

$D + F$
⟨*proof*⟩

lemma *set-plus-mono3* [*intro*]: $a \in C \Longrightarrow a +o D \subseteq C + D$
⟨*proof*⟩

lemma *set-plus-mono4* [*intro*]: $(a::'a::comm\text{-}monoid\text{-}add) \in C \Longrightarrow a +o D \subseteq D + C$
⟨*proof*⟩

lemma *set-plus-mono5*: $a \in C \Longrightarrow B \subseteq D \Longrightarrow a +o B \subseteq C + D$
⟨*proof*⟩

lemma *set-plus-mono-b*: $C \subseteq D \Longrightarrow x \in a +o C \Longrightarrow x \in a +o D$
⟨*proof*⟩

lemma *set-plus-mono2-b*: $C \subseteq D \Longrightarrow E \subseteq F \Longrightarrow x \in C + E \Longrightarrow x \in D + F$
⟨*proof*⟩

lemma *set-plus-mono3-b*: $a \in C \Longrightarrow x \in a +o D \Longrightarrow x \in C + D$
⟨*proof*⟩

lemma *set-plus-mono4-b*: $(a::'a::comm\text{-}monoid\text{-}add) : C \Longrightarrow x \in a +o D \Longrightarrow x \in D + C$
⟨*proof*⟩

lemma *set-zero-plus* [*simp*]: $(0::'a::comm\text{-}monoid\text{-}add) +o C = C$
⟨*proof*⟩

lemma *set-zero-plus2*: $(0::'a::comm\text{-}monoid\text{-}add) \in A \Longrightarrow B \subseteq A + B$
⟨*proof*⟩

lemma *set-plus-imp-minus*: $(a::'a::ab\text{-}group\text{-}add) : b +o C \Longrightarrow (a - b) \in C$
⟨*proof*⟩

lemma *set-minus-imp-plus*: $(a::'a::ab\text{-}group\text{-}add) - b : C \Longrightarrow a \in b +o C$
⟨*proof*⟩

lemma *set-minus-plus*: $(a::'a::ab\text{-}group\text{-}add) - b \in C \longleftrightarrow a \in b +o C$
⟨*proof*⟩

lemma *set-times-intro* [*intro*]: $a \in C \Longrightarrow b \in D \Longrightarrow a * b \in C * D$
⟨*proof*⟩

lemma *set-times-elim*:
assumes $x \in A * B$
obtains $a\ b$ **where** $x = a * b$ **and** $a \in A$ **and** $b \in B$
⟨*proof*⟩

lemma *set-times-intro2* [*intro*!]: $b \in C \Longrightarrow a * b \in a *o C$
⟨*proof*⟩

lemma *set-times-rearrange*:
$((a::'a::comm\text{-}monoid\text{-}mult) *o C) * (b *o D) = (a * b) *o (C * D)$
⟨*proof*⟩

lemma *set-times-rearrange2*:
$(a::'a::semigroup\text{-}mult) *o (b *o C) = (a * b) *o C$
⟨*proof*⟩

lemma *set-times-rearrange3*:
$((a::'a::semigroup\text{-}mult) *o B) * C = a *o (B * C)$
⟨*proof*⟩

theorem *set-times-rearrange4*:
$C * ((a::'a::comm\text{-}monoid\text{-}mult) *o D) = a *o (C * D)$
⟨*proof*⟩

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

lemma *set-times-mono* [*intro*]: $C \subseteq D \Longrightarrow a *o C \subseteq a *o D$
⟨*proof*⟩

lemma *set-times-mono2* [*intro*]: $(C::'a::times\ set) \subseteq D \Longrightarrow E \subseteq F \Longrightarrow C * E \subseteq D * F$
⟨*proof*⟩

lemma *set-times-mono3* [*intro*]: $a \in C \Longrightarrow a *o D \subseteq C * D$
⟨*proof*⟩

lemma *set-times-mono4* [*intro*]: $(a::'a::comm\text{-}monoid\text{-}mult) : C \Longrightarrow a *o D \subseteq D * C$
⟨*proof*⟩

lemma *set-times-mono5*: $a \in C \Longrightarrow B \subseteq D \Longrightarrow a *o B \subseteq C * D$
⟨*proof*⟩

lemma *set-times-mono-b*: $C \subseteq D \Longrightarrow x \in a *o C \Longrightarrow x \in a *o D$
⟨*proof*⟩

lemma *set-times-mono2-b*: $C \subseteq D \Longrightarrow E \subseteq F \Longrightarrow x \in C * E \Longrightarrow x \in D * F$
⟨*proof*⟩

lemma *set-times-mono3-b*: $a \in C \Longrightarrow x \in a *o D \Longrightarrow x \in C * D$
⟨*proof*⟩

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

$x \in D * C$
⟨*proof*⟩

lemma *set-one-times* [*simp*]: $(1::'a::comm\text{-}monoid\text{-}mult) *o\ C = C$
⟨*proof*⟩

lemma *set-times-plus-distrib*:
$(a::'a::semiring) *o\ (b +o\ C) = (a * b) +o\ (a *o\ C)$
⟨*proof*⟩

lemma *set-times-plus-distrib2*:
$(a::'a::semiring) *o\ (B + C) = (a *o\ B) + (a *o\ C)$
⟨*proof*⟩

lemma *set-times-plus-distrib3*: $((a::'a::semiring) +o\ C) * D \subseteq a *o\ D + C * D$
⟨*proof*⟩

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

lemma *set-neg-intro*: $(a::'a::ring\text{-}1) \in (-\ 1) *o\ C \Longrightarrow -\ a \in C$
⟨*proof*⟩

lemma *set-neg-intro2*: $(a::'a::ring\text{-}1) \in C \Longrightarrow -\ a \in (-\ 1) *o\ C$
⟨*proof*⟩

lemma *set-plus-image*: $S + T = (\lambda(x, y).\ x + y)\ `\ (S \times T)$
⟨*proof*⟩

lemma *set-times-image*: $S * T = (\lambda(x, y).\ x * y)\ `\ (S \times T)$
⟨*proof*⟩

lemma *finite-set-plus*: *finite* $s \Longrightarrow$ *finite* $t \Longrightarrow$ *finite* $(s + t)$
⟨*proof*⟩

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

lemma *set-setsum-alt*:
assumes *fin*: *finite* I
shows *setsum* $S\ I = \{setsum\ s\ I\ |s.\ \forall i{\in}I.\ s\ i \in S\ i\}$
(**is** - = *?setsum* I)
⟨*proof*⟩

lemma *setsum-set-cond-linear*:
fixes f :: $'a::comm\text{-}monoid\text{-}add\ set \Rightarrow 'b::comm\text{-}monoid\text{-}add\ set$
assumes [*intro!*]: $\bigwedge A\ B.\ P\ A \Longrightarrow P\ B \Longrightarrow P\ (A + B)\ P\ \{0\}$
and f: $\bigwedge A\ B.\ P\ A \Longrightarrow P\ B \Longrightarrow f\ (A + B) = f\ A + f\ B\ f\ \{0\} = \{0\}$

assumes *all*: $\bigwedge i.\ i \in I \Longrightarrow P\ (S\ i)$
shows $f\ (setsum\ S\ I) = setsum\ (f \circ S)\ I$
⟨*proof*⟩

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 $f\ (setsum\ S\ I) = setsum\ (f \circ S)\ I$
⟨*proof*⟩

lemma *set-times-Un-distrib*:
$A * (B \cup C) = A * B \cup A * C$
$(A \cup B) * C = A * C \cup B * C$
⟨*proof*⟩

lemma *set-times-UNION-distrib*:
$A * UNION\ I\ M = (\bigcup i{\in}I.\ A * M\ i)$
$UNION\ I\ M * A = (\bigcup i{\in}I.\ M\ i * A)$
⟨*proof*⟩

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

lemma *bigo-pos-const*:
$(\exists\, c{::}'a{::}\mathit{linordered\text{-}idom}.\ \forall\, x.\ |h\ x| \leq c * |f\ x|) \longleftrightarrow$
$(\exists\, c.\ 0 < c \wedge (\forall\, x.\ |h\ x| \leq c * |f\ x|))$
⟨*proof*⟩

lemma *bigo-alt-def*: $O(f) = \{h.\ \exists\, c.\ 0 < c \wedge (\forall\, x.\ |h\ x| \leq c * |f\ x|)\}$
⟨*proof*⟩

lemma *bigo-elt-subset* [*intro*]: $f \in O(g) \Longrightarrow O(f) \leq O(g)$
⟨*proof*⟩

lemma *bigo-refl* [*intro*]: $f \in O(f)$
⟨*proof*⟩

lemma *bigo-zero*: $0 \in O(g)$
⟨*proof*⟩

lemma *bigo-zero2*: $O(\lambda x.\ 0) = \{\lambda x.\ 0\}$
⟨*proof*⟩

lemma *bigo-plus-self-subset* [*intro*]: $O(f) + O(f) \subseteq O(f)$
⟨*proof*⟩

lemma *bigo-plus-idemp* [*simp*]: $O(f) + O(f) = O(f)$
⟨*proof*⟩

lemma *bigo-plus-subset* [*intro*]: $O(f + g) \subseteq O(f) + O(g)$
⟨*proof*⟩

lemma *bigo-plus-subset2* [*intro*]: $A \subseteq O(f) \Longrightarrow B \subseteq O(f) \Longrightarrow A + B \subseteq O(f)$
⟨*proof*⟩

lemma *bigo-plus-eq*: $\forall\, x.\ 0 \leq f\ x \Longrightarrow \forall\, x.\ 0 \leq g\ x \Longrightarrow O(f + g) = O(f) + O(g)$
⟨*proof*⟩

lemma *bigo-bounded-alt*: $\forall\, x.\ 0 \leq f\ x \Longrightarrow \forall\, x.\ f\ x \leq c * g\ x \Longrightarrow f \in O(g)$
⟨*proof*⟩

lemma *bigo-bounded*: $\forall\, x.\ 0 \leq f\ x \Longrightarrow \forall\, x.\ f\ x \leq g\ x \Longrightarrow f \in O(g)$
⟨*proof*⟩

lemma *bigo-bounded2*: $\forall\, x.\ lb\ x \leq f\ x \Longrightarrow \forall\, x.\ f\ x \leq lb\ x + g\ x \Longrightarrow f \in lb +o O(g)$
⟨*proof*⟩

lemma *bigo-abs*: $(\lambda x.\ |f\ x|) =o\ O(f)$
⟨*proof*⟩

lemma *bigo-abs2*: $f =o\ O(\lambda x.\ |f\ x|)$
⟨*proof*⟩

lemma *bigo-abs3*: $O(f) = O(\lambda x.\ |f\ x|)$
⟨*proof*⟩

lemma *bigo-abs4*: $f =o\ g +o\ O(h) \Longrightarrow (\lambda x.\ |f\ x|) =o\ (\lambda x.\ |g\ x|) +o\ O(h)$
⟨*proof*⟩

lemma *bigo-abs5*: $f =o\ O(g) \Longrightarrow (\lambda x.\ |f\ x|) =o\ O(g)$
⟨*proof*⟩

lemma *bigo-elt-subset2* [*intro*]: $f \in g +o\ O(h) \Longrightarrow O(f) \subseteq O(g) + O(h)$
⟨*proof*⟩

lemma *bigo-mult* [*intro*]: $O(f)*O(g) \subseteq O(f * g)$
⟨*proof*⟩

lemma *bigo-mult2* [*intro*]: $f *o\ O(g) \subseteq O(f * g)$
⟨*proof*⟩

lemma *bigo-mult3*: $f \in O(h) \Longrightarrow g \in O(j) \Longrightarrow f * g \in O(h * j)$
⟨*proof*⟩

lemma *bigo-mult4* [*intro*]: $f \in k +o\ O(h) \Longrightarrow g * f \in (g * k) +o\ O(g * h)$
⟨*proof*⟩

lemma *bigo-mult5*:
fixes $f :: 'a \Rightarrow 'b$::*linordered-field*
assumes $\forall x.\ f\ x \neq 0$
shows $O(f * g) \subseteq f *o\ O(g)$
⟨*proof*⟩

lemma *bigo-mult6*:
fixes $f :: 'a \Rightarrow 'b$::*linordered-field*
shows $\forall x.\ f\ x \neq 0 \Longrightarrow O(f * g) = f *o\ O(g)$
⟨*proof*⟩

lemma *bigo-mult7*:
fixes $f :: 'a \Rightarrow 'b$::*linordered-field*
shows $\forall x.\ f\ x \neq 0 \Longrightarrow O(f * g) \subseteq O(f) * O(g)$
⟨*proof*⟩

lemma *bigo-mult8*:
fixes $f :: 'a \Rightarrow 'b$::*linordered-field*
shows $\forall x.\ f\ x \neq 0 \Longrightarrow O(f * g) = O(f) * O(g)$

⟨*proof*⟩

lemma *bigo-minus* [*intro*]: $f \in O(g) \Longrightarrow -f \in O(g)$
⟨*proof*⟩

lemma *bigo-minus2*: $f \in g +o O(h) \Longrightarrow -f \in -g +o O(h)$
⟨*proof*⟩

lemma *bigo-minus3*: $O(-f) = O(f)$
⟨*proof*⟩

lemma *bigo-plus-absorb-lemma1*: $f \in O(g) \Longrightarrow f +o O(g) \subseteq O(g)$
⟨*proof*⟩

lemma *bigo-plus-absorb-lemma2*: $f \in O(g) \Longrightarrow O(g) \subseteq f +o O(g)$
⟨*proof*⟩

lemma *bigo-plus-absorb* [*simp*]: $f \in O(g) \Longrightarrow f +o O(g) = O(g)$
⟨*proof*⟩

lemma *bigo-plus-absorb2* [*intro*]: $f \in O(g) \Longrightarrow A \subseteq O(g) \Longrightarrow f +o A \subseteq O(g)$
⟨*proof*⟩

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

lemma *bigo-add-commute*: $f \in g +o O(h) \longleftrightarrow g \in f +o O(h)$
⟨*proof*⟩

lemma *bigo-const1*: $(\lambda x.\ c) \in O(\lambda x.\ 1)$
⟨*proof*⟩

lemma *bigo-const2* [*intro*]: $O(\lambda x.\ c) \subseteq O(\lambda x.\ 1)$
⟨*proof*⟩

lemma *bigo-const3*:
fixes c :: *'a*::*linordered-field*
shows $c \neq 0 \Longrightarrow (\lambda x.\ 1) \in O(\lambda x.\ c)$
⟨*proof*⟩

lemma *bigo-const4*:
fixes c :: *'a*::*linordered-field*
shows $c \neq 0 \Longrightarrow O(\lambda x.\ 1) \subseteq O(\lambda x.\ c)$
⟨*proof*⟩

lemma *bigo-const* [*simp*]:
fixes c :: *'a*::*linordered-field*
shows $c \neq 0 \Longrightarrow O(\lambda x.\ c) = O(\lambda x.\ 1)$
⟨*proof*⟩

lemma *bigo-const-mult1*: $(\lambda x.\ c * f\ x) \in O(f)$
⟨*proof*⟩

lemma *bigo-const-mult2*: $O(\lambda x.\ c * f\ x) \subseteq O(f)$
⟨*proof*⟩

lemma *bigo-const-mult3*:
fixes $c :: 'a$::*linordered-field*
shows $c \neq 0 \Longrightarrow f \in O(\lambda x.\ c * f\ x)$
⟨*proof*⟩

lemma *bigo-const-mult4*:
fixes $c :: 'a$::*linordered-field*
shows $c \neq 0 \Longrightarrow O(f) \subseteq O(\lambda x.\ c * f\ x)$
⟨*proof*⟩

lemma *bigo-const-mult* [*simp*]:
fixes $c :: 'a$::*linordered-field*
shows $c \neq 0 \Longrightarrow O(\lambda x.\ c * f\ x) = O(f)$
⟨*proof*⟩

lemma *bigo-const-mult5* [*simp*]:
fixes $c :: 'a$::*linordered-field*
shows $c \neq 0 \Longrightarrow (\lambda x.\ c) *o\ O(f) = O(f)$
⟨*proof*⟩

lemma *bigo-const-mult6* [*intro*]: $(\lambda x.\ c) *o\ O(f) \subseteq O(f)$
⟨*proof*⟩

lemma *bigo-const-mult7* [*intro*]: $f =o\ O(g) \Longrightarrow (\lambda x.\ c * f\ x) =o\ O(g)$
⟨*proof*⟩

lemma *bigo-compose1*: $f =o\ O(g) \Longrightarrow (\lambda x.\ f\ (k\ x)) =o\ O(\lambda x.\ g\ (k\ x))$
⟨*proof*⟩

lemma *bigo-compose2*: $f =o\ g +o\ O(h) \Longrightarrow$
$(\lambda x.\ f\ (k\ x)) =o\ (\lambda x.\ g\ (k\ x)) +o\ O(\lambda x.\ h(k\ x))$
⟨*proof*⟩

4.2 Setsum

lemma *bigo-setsum-main*: $\forall x.\ \forall y \in A\ x.\ 0 \leq h\ x\ y \Longrightarrow$
$\exists c.\ \forall x.\ \forall y \in A\ x.\ |f\ x\ y| \leq c * h\ x\ y \Longrightarrow$
$(\lambda x.\ \sum y \in A\ x.\ f\ x\ y) =o\ O(\lambda x.\ \sum y \in A\ x.\ h\ x\ y)$
⟨*proof*⟩

lemma *bigo-setsum1*: $\forall x\ y.\ 0 \leq h\ x\ y \Longrightarrow$
$\exists c.\ \forall x\ y.\ |f\ x\ y| \leq c * h\ x\ y \Longrightarrow$

$(\lambda x.\ \sum y \in A\ x.\ f\ x\ y) =o\ O(\lambda x.\ \sum y \in A\ x.\ h\ x\ y)$
⟨*proof*⟩

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

lemma *bigo-setsum3*: $f =o\ O(h) \Longrightarrow$
$(\lambda x.\ \sum y \in A\ x.\ l\ x\ y * f\ (k\ x\ y)) =o\ O(\lambda x.\ \sum y \in A\ x.\ |l\ x\ y * h\ (k\ x\ y)|)$
⟨*proof*⟩

lemma *bigo-setsum4*: $f =o\ g +o\ O(h) \Longrightarrow$
$(\lambda x.\ \sum y \in A\ x.\ l\ x\ y * f\ (k\ x\ y)) =o$
$(\lambda x.\ \sum y \in A\ x.\ l\ x\ y * g\ (k\ x\ y)) +o$
$O(\lambda x.\ \sum y \in A\ x.\ |l\ x\ y * h\ (k\ x\ y)|)$
⟨*proof*⟩

lemma *bigo-setsum5*: $f =o\ O(h) \Longrightarrow \forall x\ y.\ 0 \leq l\ x\ y \Longrightarrow$
$\forall x.\ 0 \leq h\ x \Longrightarrow$
$(\lambda x.\ \sum y \in A\ x.\ l\ x\ y * f\ (k\ x\ y)) =o$
$O(\lambda x.\ \sum y \in A\ x.\ l\ x\ y * h\ (k\ x\ y))$
⟨*proof*⟩

lemma *bigo-setsum6*: $f =o\ g +o\ O(h) \Longrightarrow \forall x\ y.\ 0 \leq l\ x\ y \Longrightarrow$
$\forall x.\ 0 \leq h\ x \Longrightarrow$
$(\lambda x.\ \sum y \in A\ x.\ l\ x\ y * f\ (k\ x\ y)) =o$
$(\lambda x.\ \sum y \in A\ x.\ l\ x\ y * g\ (k\ x\ y)) +o$
$O(\lambda x.\ \sum y \in A\ x.\ l\ x\ y * h\ (k\ x\ y))$
⟨*proof*⟩

4.3 Misc useful stuff

lemma *bigo-useful-intro*: $A \subseteq O(f) \Longrightarrow B \subseteq O(f) \Longrightarrow A + B \subseteq O(f)$
⟨*proof*⟩

lemma *bigo-useful-add*: $f =o\ O(h) \Longrightarrow g =o\ O(h) \Longrightarrow f + g =o\ O(h)$
⟨*proof*⟩

lemma *bigo-useful-const-mult*:
fixes c :: *'a::linordered-field*
shows $c \neq 0 \Longrightarrow (\lambda x.\ c) * f =o\ O(h) \Longrightarrow f =o\ O(h)$
⟨*proof*⟩

lemma *bigo-fix*: $(\lambda x{::}nat.\ f\ (x + 1)) =o\ O(\lambda x.\ h\ (x + 1)) \Longrightarrow f\ 0 = 0 \Longrightarrow f =o\ O(h)$
⟨*proof*⟩

lemma *bigo-fix2*:

$(\lambda x.\ f\ ((x::nat) + 1)) =o\ (\lambda x.\ g(x + 1)) +o\ O(\lambda x.\ h(x + 1)) \Longrightarrow$
$f\ 0 = g\ 0 \Longrightarrow f =o\ g +o\ O(h)$
⟨*proof*⟩

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*: $f =o\ O(h) \Longrightarrow \forall x.\ |g\ x| \leq |f\ x| \Longrightarrow g =o\ O(h)$
⟨*proof*⟩

lemma *bigo-lesseq2*: $f =o\ O(h) \Longrightarrow \forall x.\ |g\ x| \leq f\ x \Longrightarrow g =o\ O(h)$
⟨*proof*⟩

lemma *bigo-lesseq3*: $f =o\ O(h) \Longrightarrow \forall x.\ 0 \leq g\ x \Longrightarrow \forall x.\ g\ x \leq f\ x \Longrightarrow g =o\ O(h)$
⟨*proof*⟩

lemma *bigo-lesseq4*: $f =o\ O(h) \Longrightarrow$
$\forall x.\ 0 \leq g\ x \Longrightarrow \forall x.\ g\ x \leq |f\ x| \Longrightarrow g =o\ O(h)$
⟨*proof*⟩

lemma *bigo-lesso1*: $\forall x.\ f\ x \leq g\ x \Longrightarrow f <o\ g =o\ O(h)$
⟨*proof*⟩

lemma *bigo-lesso2*: $f =o\ g +o\ O(h) \Longrightarrow$
$\forall x.\ 0 \leq k\ x \Longrightarrow \forall x.\ k\ x \leq f\ x \Longrightarrow k <o\ g =o\ O(h)$
⟨*proof*⟩

lemma *bigo-lesso3*: $f =o\ g +o\ O(h) \Longrightarrow$
$\forall x.\ 0 \leq k\ x \Longrightarrow \forall x.\ g\ x \leq k\ x \Longrightarrow f <o\ k =o\ O(h)$
⟨*proof*⟩

lemma *bigo-lesso4*:
fixes $k :: 'a \Rightarrow 'b::linordered\text{-}field$
shows $f <o\ g =o\ O(k) \Longrightarrow g =o\ h +o\ O(k) \Longrightarrow f <o\ h =o\ O(k)$
⟨*proof*⟩

lemma *bigo-lesso5*: $f <o\ g =o\ O(h) \Longrightarrow \exists C.\ \forall x.\ f\ x \leq g\ x + C * |h\ x|$
⟨*proof*⟩

lemma *lesso-add*: $f <o\ g =o\ O(h) \Longrightarrow k <o\ l =o\ O(h) \Longrightarrow (f + k) <o\ (g + l) =o\ O(h)$
⟨*proof*⟩

lemma *bigo-LIMSEQ1*: $f =o\ O(g) \Longrightarrow g \longrightarrow 0 \Longrightarrow f \longrightarrow (0::real)$
⟨*proof*⟩

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

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*
⟨*proof*⟩

instantiation *bit* :: {*zero*, *one*}
begin

definition *zero-bit-def*:
0 = *Bit False*

definition *one-bit-def*:
1 = *Bit True*

instance ⟨*proof*⟩

end

old-rep-datatype *0*::*bit* *1*::*bit*
⟨*proof*⟩

lemma *Bit-set-eq* [*simp*]:
Bit (*set b*) = *b*
⟨*proof*⟩

lemma *set-Bit-eq* [*simp*]:
set (*Bit P*) = *P*
⟨*proof*⟩

lemma *bit-eq-iff*:
x = *y* ⟷ (*set x* ⟷ *set y*)
⟨*proof*⟩

lemma *Bit-inject* [*simp*]:
Bit P = *Bit Q* ⟷ (*P* ⟷ *Q*)

⟨*proof*⟩

lemma *set* [*iff*]:
¬ *set 0*
set 1
⟨*proof*⟩

lemma [*code*]:
set 0 ⟷ *False*
set 1 ⟷ *True*
⟨*proof*⟩

lemma *set-iff*:
set b ⟷ *b = 1*
⟨*proof*⟩

lemma *bit-eq-iff-set*:
b = 0 ⟷ ¬ *set b*
b = 1 ⟷ *set b*
⟨*proof*⟩

lemma *Bit* [*simp*, *code*]:
Bit False = 0
Bit True = 1
⟨*proof*⟩

lemma *bit-not-0-iff* [*iff*]:
(*x*::*bit*) ≠ *0* ⟷ *x = 1*
⟨*proof*⟩

lemma *bit-not-1-iff* [*iff*]:
(*x*::*bit*) ≠ *1* ⟷ *x = 0*
⟨*proof*⟩

lemma [*code*]:
HOL.equal 0 b ⟷ ¬ *set b*
HOL.equal 1 b ⟷ *set b*
⟨*proof*⟩

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
⟨*proof*⟩

end

lemma *bit-add-self*: $x + x = (0 :: bit)$
⟨*proof*⟩

lemma *bit-mult-eq-1-iff* [*simp*]: $x * y = (1 :: bit) \longleftrightarrow x = 1 \wedge y = 1$
⟨*proof*⟩

Not sure whether the next two should be simp rules.

lemma *bit-add-eq-0-iff*: $x + y = (0 :: bit) \longleftrightarrow x = y$
⟨*proof*⟩

lemma *bit-add-eq-1-iff*: $x + y = (1 :: bit) \longleftrightarrow x \neq y$
⟨*proof*⟩

5.3 Numerals at type *bit*

All numerals reduce to either 0 or 1.

lemma *bit-minus1* [*simp*]: $- 1 = (1 :: bit)$
⟨*proof*⟩

lemma *bit-neg-numeral* [*simp*]: $(- numeral\ w :: bit) = numeral\ w$
⟨*proof*⟩

lemma *bit-numeral-even* [*simp*]: $numeral\ (Num.Bit0\ w) = (0 :: bit)$
⟨*proof*⟩

lemma *bit-numeral-odd* [*simp*]: $numeral\ (Num.Bit1\ w) = (1 :: bit)$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *of-bit-eq-iff*:
of-bit x = *of-bit y* ⟷ *x* = *y*
⟨*proof*⟩

end

context *semiring-1*
begin

lemma *of-nat-of-bit-eq*:
of-nat (*of-bit b*) = *of-bit b*
⟨*proof*⟩

end

context *ring-1*
begin

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

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*⟩

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*: $P \longrightarrow P \longrightarrow Q \Longrightarrow P \Longrightarrow Q$
⟨*proof*⟩

lemma *inject-refine*: $g\ (f\ x) = x \Longrightarrow g\ (f\ y) = y \Longrightarrow f\ x = f\ y \longleftrightarrow x = y$
⟨*proof*⟩

lemma *convol-apply*: *BNF-Def.convol f g x* $= (f\ x,\ g\ x)$
⟨*proof*⟩

lemma *Grp-UNIV-id*: *BNF-Def.Grp UNIV id* $= (op =)$
⟨*proof*⟩

lemma *sum-comp-cases*:
assumes *f o Inl* $= g$ *o Inl* **and** *f o Inr* $= g$ *o Inr*
shows $f = g$
⟨*proof*⟩

lemma *case-sum-Inl-Inr-L*: *case-sum* $(f$ *o Inl*$)\ (f$ *o Inr*$) = f$
⟨*proof*⟩

lemma *eq-o-InrI*: $[\![g$ *o Inl* $= h$; *case-sum h f* $= g]\!] \Longrightarrow f = g$ *o Inr*
⟨*proof*⟩

lemma *id-bnf-o*: *BNF-Composition.id-bnf* $\circ\ f = f$
⟨*proof*⟩

lemma *o-id-bnf*: $f \circ$ *BNF-Composition.id-bnf* $= f$
⟨*proof*⟩

lemma *if-True-False*:
(*if P then True else Q*) $\longleftrightarrow P \vee Q$
(*if P then False else Q*) $\longleftrightarrow \neg\ P \wedge Q$

(*if P then Q else True*) $\longleftrightarrow \neg\ P \vee Q$
(*if P then Q else False*) $\longleftrightarrow P \wedge Q$
⟨*proof*⟩

lemma *if-distrib-fun*: (*if c then f else g*) *x* = (*if c then f x else g x*)
⟨*proof*⟩

7.1 Coinduction

lemma *eq-comp-compI*: *a o b* = *f o x* $\Longrightarrow$ *x o c* = *id* $\Longrightarrow$ *f* = *a o* (*b o c*)
⟨*proof*⟩

lemma *self-bounded-weaken-left*: (*a* :: *'a* :: *semilattice-inf*) $\leq$ *inf a b* $\Longrightarrow$ *a* $\leq$ *b*
⟨*proof*⟩

lemma *self-bounded-weaken-right*: (*a* :: *'a* :: *semilattice-inf*) $\leq$ *inf b a* $\Longrightarrow$ *a* $\leq$ *b*
⟨*proof*⟩

lemma *symp-iff*: *symp R* $\longleftrightarrow$ *R* = *R^--1*
⟨*proof*⟩

lemma *equivp-inf*: ⟦*equivp R*; *equivp S*⟧ $\Longrightarrow$ *equivp* (*inf R S*)
⟨*proof*⟩

lemma *vimage2p-rel-prod*:
(λ*x 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*))
⟨*proof*⟩

lemma *predicate2I-obj*: (∀ *x y*. *P x y* $\longrightarrow$ *Q x y*) $\Longrightarrow$ *P* $\leq$ *Q*
⟨*proof*⟩

lemma *predicate2D-obj*: *P* $\leq$ *Q* $\Longrightarrow$ *P x y* $\longrightarrow$ *Q x y*
⟨*proof*⟩

locale *cong* =
fixes *rel* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ (*'b* ⇒ *'b* ⇒ *bool*)
and *eval* :: *'b* ⇒ *'a*
and *retr* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ (*'a* ⇒ *'a* ⇒ *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-fun* (*rel R*) *R*) *eval eval*; *rel* (*inf R* (*retr R*)) *x y*⟧
$\Longrightarrow$
retr R (*eval x*) (*eval y*)
begin

definition *cong* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *bool* **where**
cong R ≡ *equivp R* ∧ (*rel-fun* (*rel R*) *R*) *eval eval*

lemma *cong-retr*: *cong R* $\Longrightarrow$ *cong* (*inf R* (*retr R*))
⟨*proof*⟩

lemma *cong-equivp*: *cong R* $\Longrightarrow$ *equivp R*
⟨*proof*⟩

definition *gen-cong* :: ('*a* $\Rightarrow$ '*a* $\Rightarrow$ *bool*) $\Rightarrow$ '*a* $\Rightarrow$ '*a* $\Rightarrow$ *bool* **where**
gen-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-cong R x y*
⟨*proof*⟩

lemma *gen-cong-symp*[*intro*]: *gen-cong R x y* $\Longrightarrow$ *gen-cong R y x*
⟨*proof*⟩

lemma *gen-cong-transp*[*intro*]: *gen-cong R x y* $\Longrightarrow$ *gen-cong R y z* $\Longrightarrow$ *gen-cong R x z*
⟨*proof*⟩

lemma *equivp-gen-cong*: *equivp* (*gen-cong R*)
⟨*proof*⟩

lemma *leq-gen-cong*: *R* $\leq$ *gen-cong R*
⟨*proof*⟩

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

lemma *gen-cong-minimal*: ⟦*R* $\leq$ *R'*; *cong R'*⟧ $\Longrightarrow$ *gen-cong R* $\leq$ *R'*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *cong-gen-cong*: *cong* (*gen-cong R*)
⟨*proof*⟩

lemma *gen-cong-eval-rel-fun*:
(*rel-fun* (*rel* (*gen-cong R*)) (*gen-cong R*)) *eval eval*
⟨*proof*⟩

lemma *gen-cong-eval*:
rel (*gen-cong R*) *x y* $\Longrightarrow$ *gen-cong R* (*eval x*) (*eval y*)
⟨*proof*⟩

lemma *gen-cong-idem*: *gen-cong* (*gen-cong R*) = *gen-cong R*
⟨*proof*⟩

lemma *gen-cong-rho*:

ϱ = eval o f ⟹ rel (gen-cong R) (f x) (f y) ⟹ gen-cong R (ϱ x) (ϱ y)
⟨*proof*⟩
lemma *coinduction*:
assumes *coind*: ∀ *R*. *R* ≤ *retr R* ⟶ *R* ≤ *op* =
assumes *cih*: *R* ≤ *retr* (*gen-cong R*)
shows *R* ≤ *op* =
⟨*proof*⟩

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*
∧ *rel-fun S T g1 g2*)
⟨*proof*⟩

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* ⟨*proof*⟩
interpretation *step*: *cong rel′ eval′ retr* ⟨*proof*⟩

lemma *gen-cong-emb*: *base.gen-cong R* ≤ *step.gen-cong R*
⟨*proof*⟩

end

⟨*ML*⟩

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* ⊓ *y*) ⊓ *z* = *x* ⊓ (*y* ⊓ *z*)
assumes *disj-assoc*: (*x* ⊔ *y*) ⊔ *z* = *x* ⊔ (*y* ⊔ *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*
⟨*proof*⟩

sublocale *disj*: *abel-semigroup disj*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *compl-unique*: $[\![x \sqcap y = \mathbf{0};\ x \sqcup y = \mathbf{1}]\!] \Longrightarrow \sim x = y$
⟨*proof*⟩

lemma *double-compl* [*simp*]: $\sim (\sim x) = x$
⟨*proof*⟩

lemma *compl-eq-compl-iff* [*simp*]: $(\sim x = \sim y) = (x = y)$
⟨*proof*⟩

8.2 Conjunction

lemma *conj-absorb* [*simp*]: $x \sqcap x = x$
⟨*proof*⟩

lemma *conj-zero-right* [*simp*]: $x \sqcap \mathbf{0} = \mathbf{0}$
⟨*proof*⟩

lemma *compl-one* [*simp*]: $\sim \mathbf{1} = \mathbf{0}$
⟨*proof*⟩

lemma *conj-zero-left* [*simp*]: $\mathbf{0} \sqcap x = \mathbf{0}$
⟨*proof*⟩

lemma *conj-one-left* [*simp*]: $\mathbf{1} \sqcap x = x$
⟨*proof*⟩

lemma *conj-cancel-left* [*simp*]: $\sim x \sqcap x = \mathbf{0}$
⟨*proof*⟩

lemma *conj-left-absorb* [*simp*]: $x \sqcap (x \sqcap y) = x \sqcap y$
⟨*proof*⟩

lemma *conj-disj-distrib2*:
$(y \sqcup z) \sqcap x = (y \sqcap x) \sqcup (z \sqcap x)$
⟨*proof*⟩

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

8.3 Disjunction

lemma *disj-absorb* [*simp*]: $x \sqcup x = x$
⟨*proof*⟩

lemma *disj-one-right* [*simp*]: $x \sqcup \mathbf{1} = \mathbf{1}$
⟨*proof*⟩

lemma *compl-zero* [*simp*]: $\sim \mathbf{0} = \mathbf{1}$
⟨*proof*⟩

lemma *disj-zero-left* [*simp*]: $\mathbf{0} \sqcup x = x$
⟨*proof*⟩

lemma *disj-one-left* [*simp*]: $\mathbf{1} \sqcup x = \mathbf{1}$
⟨*proof*⟩

lemma *disj-cancel-left* [*simp*]: $\sim x \sqcup x = \mathbf{1}$
⟨*proof*⟩

lemma *disj-left-absorb* [*simp*]: $x \sqcup (x \sqcup y) = x \sqcup y$
⟨*proof*⟩

lemma *disj-conj-distrib2*:

$(y \sqcap z) \sqcup x = (y \sqcup x) \sqcap (z \sqcup x)$
⟨*proof*⟩

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*⟩

lemma *de-Morgan-disj* [*simp*]: $\sim (x \sqcup y) = \sim x \sqcap \sim y$
⟨*proof*⟩

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*⟩

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*:
$x \oplus y = (x \sqcup y) \sqcap (\sim x \sqcup \sim y)$
⟨*proof*⟩

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

lemma *xor-zero-left* [*simp*]: $\mathbf{0} \oplus x = x$
⟨*proof*⟩

lemma *xor-one-right* [*simp*]: $x \oplus \mathbf{1} = \sim x$
⟨*proof*⟩

lemma *xor-one-left* [*simp*]: $\mathbf{1} \oplus x = \sim x$
⟨*proof*⟩

lemma *xor-self* [*simp*]: $x \oplus x = \mathbf{0}$
⟨*proof*⟩

lemma *xor-left-self* [*simp*]: $x \oplus (x \oplus y) = y$
⟨*proof*⟩

lemma *xor-compl-left* [*simp*]: $\sim x \oplus y = \sim (x \oplus y)$
⟨*proof*⟩

lemma *xor-compl-right* [*simp*]: $x \oplus \sim y = \sim (x \oplus y)$
⟨*proof*⟩

lemma *xor-cancel-right*: $x \oplus \sim x = \mathbf{1}$
⟨*proof*⟩

lemma *xor-cancel-left*: $\sim x \oplus x = \mathbf{1}$
⟨*proof*⟩

lemma *conj-xor-distrib*: $x \sqcap (y \oplus z) = (x \sqcap y) \oplus (x \sqcap z)$
⟨*proof*⟩

lemma *conj-xor-distrib2*: $(y \oplus z) \sqcap x = (y \sqcap x) \oplus (z \sqcap x)$
⟨*proof*⟩

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?*]:
$(\bigwedge a\ b.\ [\![\ a \in Y;\ b \in Y\ ]\!] \Longrightarrow (a, b) \in r \vee (b, a) \in r) \Longrightarrow Y \in Chains\ r$
⟨*proof*⟩

lemma *in-Chains-subset*: $[\![\ M \in Chains\ r;\ M' \subseteq M\ ]\!] \Longrightarrow M' \in Chains\ r$
⟨*proof*⟩

lemma *FieldI1*: $(i, j) \in R \Longrightarrow i \in Field\ R$
⟨*proof*⟩

lemma *Chains-FieldD*: $[\![\ M \in Chains\ r;\ x \in M\ ]\!] \Longrightarrow x \in Field\ r$
⟨*proof*⟩

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

⟨*proof*⟩

lemma *partial-order-on-trans*:
⟦ *partial-order-on A r*; $(x, y) \in r$; $(y, z) \in r$ ⟧ ⟹ $(x, z) \in r$
⟨*proof*⟩

locale *bourbaki-witt-fixpoint* =
fixes *lub* :: $'a$ *set* ⇒ $'a$
and *leq* :: $('a \times 'a)$ *set*
and *f* :: $'a$ ⇒ $'a$
assumes *po*: *Partial-order leq*
and *lub-least*: ⟦ $M \in$ *Chains leq*; $M \neq \{\}$; $\bigwedge x.\ x \in M \Longrightarrow (x, z) \in$ *leq* ⟧ ⟹ $(lub\ M, z) \in$ *leq*
and *lub-upper*: ⟦ $M \in$ *Chains leq*; $x \in M$ ⟧ ⟹ $(x, lub\ M) \in$ *leq*
and *lub-in-Field*: ⟦ $M \in$ *Chains leq*; $M \neq \{\}$ ⟧ ⟹ *lub* $M \in$ *Field leq*
and *increasing*: $\bigwedge x.\ x \in$ *Field leq* ⟹ $(x, f\ x) \in$ *leq*
begin

lemma *leq-trans*: ⟦ $(x, y) \in$ *leq*; $(y, z) \in$ *leq* ⟧ ⟹ $(x, z) \in$ *leq*
⟨*proof*⟩

lemma *leq-refl*: $x \in$ *Field leq* ⟹ $(x, x) \in$ *leq*
⟨*proof*⟩

lemma *leq-antisym*: ⟦ $(x, y) \in$ *leq*; $(y, x) \in$ *leq* ⟧ ⟹ $x = y$
⟨*proof*⟩

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

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

lemma *fixp-above-outside*: $a \notin$ *Field leq* ⟹ *fixp-above a* = *a*
⟨*proof*⟩

lemma *fixp-above-inside*: $a \in$ *Field leq* ⟹ *fixp-above a* = *lub* (*iterates-above a*)
⟨*proof*⟩

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* ⟧ ⟹ $(x, f\ x) \in$ *leq*
⟨*proof*⟩

lemma *iterates-above-Field*: ⟦ $x \in$ *iterates-above* a; $a \in$ *Field leq* ⟧ ⟹ $x \in$ *Field leq*
⟨*proof*⟩

lemma *iterates-above-ge*:
assumes *y*: $y \in$ *iterates-above* a
and *a*: $a \in$ *Field leq*
shows $(a, y) \in$ *leq*
⟨*proof*⟩

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*⟩

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* $(f\ a)$
⟨*proof*⟩

lemma *iterates-above-Sup-aux*:
assumes *M*: $M \in$ *Chains leq* $M \neq \{\}$
and *M'*: $M' \in$ *Chains leq* $M' \neq \{\}$
and *comp*: $\bigwedge x.\ x \in M \Longrightarrow x \in$ *iterates-above* (*lub* M') $\vee$ *lub* $M' \in$ *iterates-above* x
shows (*lub* M, *lub* M') $\in$ *leq* $\vee$ *lub* $M \in$ *iterates-above* (*lub* M')
⟨*proof*⟩

lemma *iterates-above-triangle*:
assumes *x*: $x \in$ *iterates-above* a
and *y*: $y \in$ *iterates-above* a
and *a*: $a \in$ *Field leq*
shows $x \in$ *iterates-above* $y \vee y \in$ *iterates-above* x
⟨*proof*⟩

lemma *chain-iterates-above*:
assumes *a*: $a \in$ *Field leq*
shows *iterates-above* $a \in$ *Chains leq* (**is** *?C* $\in$ -)
⟨*proof*⟩

lemma *fixp-iterates-above*: *fixp-above a* ∈ *iterates-above a*
⟨*proof*⟩

lemma *fixp-above-Field*: *a* ∈ *Field leq* ⟹ *fixp-above a* ∈ *Field leq*
⟨*proof*⟩

lemma *fixp-above-unfold*:
assumes *a*: *a* ∈ *Field leq*
shows *fixp-above a* = *f* (*fixp-above a*) (**is** *?a* = *f ?a*)
⟨*proof*⟩

end

lemma *fixp-induct* [*case-names adm base step*]:
assumes *adm*: *ccpo.admissible lub* (λ*x y*. (*x*, *y*) ∈ *leq*) *P*
and *base*: *P a*
and *step*: ⋀*x*. *P x* ⟹ *P* (*f x*)
shows *P* (*fixp-above a*)
⟨*proof*⟩

end

end

10 Order on characters

theory *Char-ord*
imports *Main*
begin

instantiation *char* :: *linorder*
begin

definition *c1* ≤ *c2* ⟷ *nat-of-char c1* ≤ *nat-of-char c2*
definition *c1* < *c2* ⟷ *nat-of-char c1* < *nat-of-char c2*

instance
⟨*proof*⟩

end

lemma *less-eq-char-simps*:
(*0* :: *char*) ≤ *c*
Char k ≤ *0* ⟷ *numeral k mod 256* = (*0* :: *nat*)
Char k ≤ *Char l* ⟷ *numeral k mod 256* ≤ (*numeral l mod 256* :: *nat*)
⟨*proof*⟩

lemma *less-char-simps*:

$\neg\ c < (0 :: char)$
$0 < Char\ k \longleftrightarrow (0 :: nat) < numeral\ k\ mod\ 256$
$Char\ k < Char\ l \longleftrightarrow numeral\ k\ mod\ 256 < (numeral\ l\ mod\ 256 :: nat)$
⟨*proof*⟩

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

definition (*inf* :: *char* ⇒ -) = *min*
definition (*sup* :: *char* ⇒ -) = *max*

instance
⟨*proof*⟩

end

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

context includes *literal.lifting* **begin**
lift-definition *less-literal* :: *String.literal* ⇒ *String.literal* ⇒ *bool* **is** *ord.lexordp* *op* < ⟨*proof*⟩
lift-definition *less-eq-literal* :: *String.literal* ⇒ *String.literal* ⇒ *bool* **is** *ord.lexordp-eq* *op* < ⟨*proof*⟩

instance
⟨*proof*⟩

end
end

lemma *less-literal-code* [*code*]:
$op <= (\lambda xs\ ys.\ ord.lexordp\ op < (String.explode\ xs)\ (String.explode\ ys))$
⟨*proof*⟩

lemma *less-eq-literal-code* [*code*]:
$op \leq (\lambda xs\ ys.\ ord.lexordp\text{-}eq\ op < (String.explode\ xs)\ (String.explode\ ys))$
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

context begin
⟨*ML*⟩

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*
⟨*proof*⟩

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*)]
⟨*proof*⟩

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*)]
⟨*proof*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

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
⟨*proof*⟩

lemma *Push-inject1*: *Push i f = Push j g ==> i=j*
⟨*proof*⟩

lemma *Push-inject2*: *Push i f = Push j g ==> f=g*
⟨*proof*⟩

lemma *Push-inject*:
[| Push i f =Push j g; [| i=j; f=g |] ==> P |] ==> P
⟨*proof*⟩

lemma *Push-neq-K0*: *Push (Inr (Suc k)) f = (%z. Inr 0) ==> P*
⟨*proof*⟩

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

lemma *Node-K0-I*: (%*k*. *Inr 0*, *a*) : *Node*
⟨*proof*⟩

lemma *Node-Push-I*: *p*: *Node* ==> *apfst* (*Push i*) *p* : *Node*
⟨*proof*⟩

11.2 Freeness: Distinctness of Constructors

lemma *Scons-not-Atom* [*iff*]: *Scons M N* ≠ *Atom*(*a*)
⟨*proof*⟩

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

lemma *inj-Atom*: *inj*(*Atom*)
⟨*proof*⟩
lemmas *Atom-inject* = *inj-Atom* [*THEN injD*]

lemma *Atom-Atom-eq* [*iff*]: (*Atom*(*a*)=*Atom*(*b*)) = (*a*=*b*)
⟨*proof*⟩

lemma *inj-Leaf*: *inj*(*Leaf*)
⟨*proof*⟩

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

lemma *inj-Numb*: *inj*(*Numb*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *Scons-inject-lemma1*: *Scons M N <= Scons M' N' ==> M<=M'*
⟨*proof*⟩

lemma *Scons-inject-lemma2*: *Scons M N <= Scons M' N' ==> N<=N'*
⟨*proof*⟩

lemma *Scons-inject1*: *Scons M N = Scons M' N' ==> M=M'*
⟨*proof*⟩

lemma *Scons-inject2*: *Scons M N = Scons M' N' ==> N=N'*
⟨*proof*⟩

lemma *Scons-inject*:
[| *Scons M N = Scons M' N'*; [| *M=M'*; *N=N'* |] ==> *P* |] ==> *P*
⟨*proof*⟩

lemma *Scons-Scons-eq* [*iff*]: (*Scons M N = Scons M' N'*) = (*M=M'* & *N=N'*)
⟨*proof*⟩

lemma *Scons-not-Leaf* [*iff*]: *Scons M N* ≠ *Leaf*(*a*)
⟨*proof*⟩

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

lemma *Scons-not-Numb* [*iff*]: *Scons M N* ≠ *Numb*(*k*)
⟨*proof*⟩

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

lemma *Leaf-not-Numb* [*iff*]: *Leaf*(*a*) ≠ *Numb*(*k*)
⟨*proof*⟩

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

lemma *ndepth-K0*: *ndepth* (*Abs-Node*(%*k*. *Inr* 0, *x*)) = 0
⟨*proof*⟩

lemma *ndepth-Push-Node-aux*:
case-nat (*Inr* (*Suc i*)) *f k* = *Inr* 0 −−> *Suc*(*LEAST x*. *f x* = *Inr* 0) <= *k*
⟨*proof*⟩

lemma *ndepth-Push-Node*:
ndepth (*Push-Node* (*Inr* (*Suc i*)) *n*) = *Suc*(*ndepth*(*n*))
⟨*proof*⟩

lemma *ntrunc-0* [*simp*]: *ntrunc* 0 *M* = {}
⟨*proof*⟩

lemma *ntrunc-Atom* [*simp*]: *ntrunc* (*Suc k*) (*Atom a*) = *Atom*(*a*)
⟨*proof*⟩

lemma *ntrunc-Leaf* [*simp*]: *ntrunc* (*Suc k*) (*Leaf a*) = *Leaf*(*a*)
⟨*proof*⟩

lemma *ntrunc-Numb* [*simp*]: *ntrunc* (*Suc k*) (*Numb i*) = *Numb*(*i*)
⟨*proof*⟩

lemma *ntrunc-Scons* [*simp*]:
ntrunc (*Suc k*) (*Scons M N*) = *Scons* (*ntrunc k M*) (*ntrunc k N*)
⟨*proof*⟩

lemma *ntrunc-one-In0* [*simp*]: *ntrunc* (*Suc 0*) (*In0 M*) = {}
⟨*proof*⟩

lemma *ntrunc-In0* [*simp*]: *ntrunc* (*Suc*(*Suc k*)) (*In0 M*) = *In0* (*ntrunc* (*Suc k*) *M*)
⟨*proof*⟩

lemma *ntrunc-one-In1* [*simp*]: *ntrunc* (*Suc 0*) (*In1 M*) = {}
⟨*proof*⟩

lemma *ntrunc-In1* [*simp*]: *ntrunc* (*Suc*(*Suc k*)) (*In1 M*) = *In1* (*ntrunc* (*Suc k*) *M*)
⟨*proof*⟩

11.3 Set Constructions

lemma *uprodI* [*intro*!]: [| M:A; N:B |] ==> *Scons* M N : *uprod* A B
⟨*proof*⟩

lemma *uprodE* [*elim*!]:
[| c : *uprod* A B;
!!x y. [| x:A; y:B; c = *Scons* x y |] ==> P
|] ==> P
⟨*proof*⟩

lemma *uprodE2*: [| *Scons* M N : *uprod* A B; [| M:A; N:B |] ==> P |] ==> P
⟨*proof*⟩

lemma *usum-In0I* [*intro*]: M:A ==> *In0*(M) : *usum* A B
⟨*proof*⟩

lemma *usum-In1I* [*intro*]: N:B ==> *In1*(N) : *usum* A B
⟨*proof*⟩

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

lemma *In0-not-In1* [*iff*]: *In0*(M) ≠ *In1*(N)
⟨*proof*⟩

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

lemma *In0-inject*: *In0*(M) = *In0*(N) ==> M=N
⟨*proof*⟩

lemma *In1-inject*: *In1*(M) = *In1*(N) ==> M=N
⟨*proof*⟩

lemma *In0-eq* [*iff*]: (*In0* M = *In0* N) = (M=N)
⟨*proof*⟩

lemma *In1-eq* [*iff*]: $(In1\ M = In1\ N) = (M{=}N)$
⟨*proof*⟩

lemma *inj-In0*: *inj In0*
⟨*proof*⟩

lemma *inj-In1*: *inj In1*
⟨*proof*⟩

lemma *Lim-inject*: $Lim\ f = Lim\ g \Longrightarrow f = g$
⟨*proof*⟩

lemma *ntrunc-subsetI*: $ntrunc\ k\ M <= M$
⟨*proof*⟩

lemma *ntrunc-subsetD*: $(!!k.\ ntrunc\ k\ M <= N) \Longrightarrow M{<=}N$
⟨*proof*⟩

lemma *ntrunc-equality*: $(!!k.\ ntrunc\ k\ M = ntrunc\ k\ N) \Longrightarrow M{=}N$
⟨*proof*⟩

lemma *ntrunc-o-equality*:
$[|\ !!k.\ (ntrunc(k)\ o\ h1) = (ntrunc(k)\ o\ h2)\ |] \Longrightarrow h1{=}h2$
⟨*proof*⟩

lemma *uprod-mono*: $[|\ A{<=}A';\ \ B{<=}B'\ |] \Longrightarrow uprod\ A\ B <= uprod\ A'\ B'$
⟨*proof*⟩

lemma *usum-mono*: $[|\ A{<=}A';\ \ B{<=}B'\ |] \Longrightarrow usum\ A\ B <= usum\ A'\ B'$
⟨*proof*⟩

lemma *Scons-mono*: $[|\ M{<=}M';\ \ N{<=}N'\ |] \Longrightarrow Scons\ M\ N <= Scons\ M'\ N'$
⟨*proof*⟩

lemma *In0-mono*: $M{<=}N \Longrightarrow In0(M) <= In0(N)$
⟨*proof*⟩

lemma *In1-mono*: $M{<=}N \Longrightarrow In1(M) <= In1(N)$
⟨*proof*⟩

lemma *Split* [*simp*]: *Split c* (*Scons M N*) = *c M N*
⟨*proof*⟩

lemma *Case-In0* [*simp*]: *Case c d* (*In0 M*) = *c*(*M*)
⟨*proof*⟩

lemma *Case-In1* [*simp*]: *Case c d* (*In1 N*) = *d*(*N*)
⟨*proof*⟩

lemma *ntrunc-UN1*: *ntrunc k* (*UN x. f*(*x*)) = (*UN x. ntrunc k* (*f x*))
⟨*proof*⟩

lemma *Scons-UN1-x*: *Scons* (*UN x. f x*) *M* = (*UN x. Scons* (*f x*) *M*)
⟨*proof*⟩

lemma *Scons-UN1-y*: *Scons M* (*UN x. f x*) = (*UN x. Scons M* (*f x*))
⟨*proof*⟩

lemma *In0-UN1*: *In0*(*UN x. f*(*x*)) = (*UN x. In0*(*f*(*x*)))
⟨*proof*⟩

lemma *In1-UN1*: *In1*(*UN x. f*(*x*)) = (*UN x. In1*(*f*(*x*)))
⟨*proof*⟩

lemma *dprodI* [*intro*!]:
[| (*M*,*M'*):*r*; (*N*,*N'*):*s* |] ==> (*Scons M N*, *Scons M' N'*) : *dprod r s*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *dsum-In0I* [*intro*]: (*M*,*M'*):*r* ==> (*In0*(*M*), *In0*(*M'*)) : *dsum r s*
⟨*proof*⟩

lemma *dsum-In1I* [*intro*]: (*N*,*N'*):*s* ==> (*In1*(*N*), *In1*(*N'*)) : *dsum r s*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *dprod-mono*: [| *r*<=*r'*; *s*<=*s'* |] ==> *dprod r s* <= *dprod r' s'*
⟨*proof*⟩

lemma *dsum-mono*: [| *r*<=*r'*; *s*<=*s'* |] ==> *dsum r s* <= *dsum r' s'*
⟨*proof*⟩

lemma *dprod-Sigma*: (*dprod* (*A* × *B*) (*C* × *D*)) <= (*uprod A C*) × (*uprod B D*)
⟨*proof*⟩

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*)))
⟨*proof*⟩

lemma *dsum-Sigma*: (*dsum* (*A* × *B*) (*C* × *D*)) <= (*usum A C*) × (*usum B D*)
⟨*proof*⟩

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

lemma *Domain-dprod* [*simp*]: *Domain* (*dprod r s*) = *uprod* (*Domain r*) (*Domain s*)
⟨*proof*⟩

lemma *Domain-dsum* [*simp*]: *Domain* (*dsum r s*) = *usum* (*Domain r*) (*Domain s*)
⟨*proof*⟩

hides popular names

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

⟨*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*
⟨*proof*⟩

lemma *triangle-Suc* [*simp*]: *triangle* (*Suc n*) = *triangle n* + *Suc n*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *prod-decode-inverse* [*simp*]: *prod-encode* (*prod-decode n*) = *n*
⟨*proof*⟩

lemma *prod-decode-triangle-add*: *prod-decode* (*triangle k* + *m*) = *prod-decode-aux k m*
⟨*proof*⟩

lemma *prod-encode-inverse* [*simp*]: *prod-decode* (*prod-encode x*) = *x*
⟨*proof*⟩

lemma *inj-prod-encode*: *inj-on prod-encode A*
⟨*proof*⟩

lemma *inj-prod-decode*: *inj-on prod-decode A*
⟨*proof*⟩

lemma *surj-prod-encode*: *surj prod-encode*
⟨*proof*⟩

lemma *surj-prod-decode*: *surj prod-decode*
⟨*proof*⟩

lemma *bij-prod-encode*: *bij prod-encode*
⟨*proof*⟩

lemma *bij-prod-decode*: *bij prod-decode*
⟨*proof*⟩

lemma *prod-encode-eq*: *prod-encode x* = *prod-encode y* ⟷ *x* = *y*
⟨*proof*⟩

lemma *prod-decode-eq*: *prod-decode x* = *prod-decode y* ⟷ *x* = *y*
⟨*proof*⟩

Ordering properties

lemma *le-prod-encode-1*: *a* ≤ *prod-encode* (*a*, *b*)
⟨*proof*⟩

lemma *le-prod-encode-2*: *b* ≤ *prod-encode* (*a*, *b*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *sum-decode-inverse* [*simp*]: *sum-encode* (*sum-decode n*) = *n*
⟨*proof*⟩

lemma *inj-sum-encode*: *inj-on sum-encode A*
⟨*proof*⟩

lemma *inj-sum-decode*: *inj-on sum-decode A*
⟨*proof*⟩

lemma *surj-sum-encode*: *surj sum-encode*
⟨*proof*⟩

lemma *surj-sum-decode*: *surj sum-decode*
⟨*proof*⟩

lemma *bij-sum-encode*: *bij sum-encode*
⟨*proof*⟩

lemma *bij-sum-decode*: *bij sum-decode*
⟨*proof*⟩

lemma *sum-encode-eq*: *sum-encode x* = *sum-encode y* ⟷ *x* = *y*
⟨*proof*⟩

lemma *sum-decode-eq*: *sum-decode x* = *sum-decode y* ⟷ *x* = *y*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *int-decode-inverse* [*simp*]: *int-encode* (*int-decode n*) = *n*
⟨*proof*⟩

lemma *inj-int-encode*: *inj-on int-encode A*

⟨*proof*⟩

lemma *inj-int-decode*: *inj-on int-decode A*
⟨*proof*⟩

lemma *surj-int-encode*: *surj int-encode*
⟨*proof*⟩

lemma *surj-int-decode*: *surj int-decode*
⟨*proof*⟩

lemma *bij-int-encode*: *bij int-encode*
⟨*proof*⟩

lemma *bij-int-decode*: *bij int-decode*
⟨*proof*⟩

lemma *int-encode-eq*: *int-encode x* = *int-encode y* ⟷ *x* = *y*
⟨*proof*⟩

lemma *int-decode-eq*: *int-decode x* = *int-decode y* ⟷ *x* = *y*
⟨*proof*⟩

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*)
⟨*proof*⟩

termination *list-decode*
⟨*proof*⟩

lemma *list-encode-inverse* [*simp*]: *list-decode* (*list-encode x*) = *x*
⟨*proof*⟩

lemma *list-decode-inverse* [*simp*]: *list-encode* (*list-decode n*) = *n*
⟨*proof*⟩

lemma *inj-list-encode*: *inj-on list-encode A*
⟨*proof*⟩

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

⟨*proof*⟩

lemma *surj-list-encode*: *surj list-encode*
⟨*proof*⟩

lemma *surj-list-decode*: *surj list-decode*
⟨*proof*⟩

lemma *bij-list-encode*: *bij list-encode*
⟨*proof*⟩

lemma *bij-list-decode*: *bij list-decode*
⟨*proof*⟩

lemma *list-encode-eq*: *list-encode x = list-encode y ⟷ x = y*
⟨*proof*⟩

lemma *list-decode-eq*: *list-decode x = list-decode y ⟷ x = y*
⟨*proof*⟩

12.5 Finite sets of naturals

12.5.1 Preliminaries

lemma *finite-vimage-Suc-iff*: *finite (Suc −‘ F) ⟷ finite F*
⟨*proof*⟩

lemma *vimage-Suc-insert-0*: *Suc −‘ insert 0 A = Suc −‘ A*
⟨*proof*⟩

lemma *vimage-Suc-insert-Suc*:
Suc −‘ insert (Suc n) A = insert n (Suc −‘ A)
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *set-encode-inf*: ~ *finite A ⟹ set-encode A = 0*

⟨*proof*⟩

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

lemma *even-set-encode-iff*: *finite A* ⟹ *even* (*set-encode A*) ⟷ *0* ∉ *A*
⟨*proof*⟩

lemma *set-encode-vimage-Suc*: *set-encode* (*Suc* −‘ *A*) = *set-encode A div 2*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *set-decode-Suc* [*simp*]:
Suc n ∈ *set-decode x* ⟷ *n* ∈ *set-decode* (*x div 2*)
⟨*proof*⟩

lemma *set-decode-zero* [*simp*]: *set-decode 0* = {}
⟨*proof*⟩

lemma *set-decode-div-2*: *set-decode* (*x div 2*) = *Suc* −‘ *set-decode x*
⟨*proof*⟩

lemma *set-decode-plus-power-2*:
n ∉ *set-decode z* ⟹ *set-decode* (*2* ^ *n* + *z*) = *insert n* (*set-decode z*)
⟨*proof*⟩

lemma *finite-set-decode* [*simp*]: *finite* (*set-decode n*)
⟨*proof*⟩

12.5.4 Proof of isomorphism

lemma *set-decode-inverse* [*simp*]: *set-encode* (*set-decode n*) = *n*
⟨*proof*⟩

lemma *set-encode-inverse* [*simp*]: *finite A* ⟹ *set-decode* (*set-encode A*) = *A*
⟨*proof*⟩

lemma *inj-on-set-encode*: *inj-on set-encode* (*Collect finite*)
⟨*proof*⟩

lemma *set-encode-eq*:
  $[\![\mathit{finite}\ A; \mathit{finite}\ B]\!] \Longrightarrow \mathit{set\text{-}encode}\ A = \mathit{set\text{-}encode}\ B \longleftrightarrow A = B$
⟨*proof*⟩

lemma *subset-decode-imp-le*:
  assumes *set-decode m* ⊆ *set-decode n*
  shows $m \leq n$
⟨*proof*⟩

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*: $\exists\, \mathit{to\text{-}nat} :: 'a \Rightarrow \mathit{nat}.\ \mathit{inj}\ \mathit{to\text{-}nat}$

lemma *countable-classI*:
  fixes $f :: 'a \Rightarrow \mathit{nat}$
  assumes $\bigwedge x\ y.\ f\ x = f\ y \Longrightarrow x = y$
  shows *OFCLASS*('*a*, *countable-class*)
⟨*proof*⟩

13.2 Conversion functions

definition $\mathit{to\text{-}nat} :: 'a{::}\mathit{countable} \Rightarrow \mathit{nat}$ **where**
  $\mathit{to\text{-}nat} = (\mathit{SOME}\ f.\ \mathit{inj}\ f)$

definition $\mathit{from\text{-}nat} :: \mathit{nat} \Rightarrow 'a{::}\mathit{countable}$ **where**
  $\mathit{from\text{-}nat} = \mathit{inv}\ (\mathit{to\text{-}nat} :: 'a \Rightarrow \mathit{nat})$

lemma *inj-to-nat* [*simp*]: *inj to-nat*
  ⟨*proof*⟩

lemma *inj-on-to-nat*[*simp*, *intro*]: *inj-on to-nat S*
  ⟨*proof*⟩

lemma *surj-from-nat* [*simp*]: *surj from-nat*
  ⟨*proof*⟩

lemma *to-nat-split* [*simp*]: $\mathit{to\text{-}nat}\ x = \mathit{to\text{-}nat}\ y \longleftrightarrow x = y$
  ⟨*proof*⟩

lemma *from-nat-to-nat* [*simp*]:
from-nat (*to-nat x*) = *x*
⟨*proof*⟩

13.3 Finite types are countable

subclass (**in** *finite*) *countable*
⟨*proof*⟩

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*))))
⟨*proof*⟩

lemma *le-sum-encode-Inl*: *x* ≤ *y* ⟹ *x* ≤ *sum-encode* (*Inl y*)
⟨*proof*⟩

lemma *le-sum-encode-Inr*: *x* ≤ *y* ⟹ *x* ≤ *sum-encode* (*Inr y*)
⟨*proof*⟩ **termination**
⟨*proof*⟩

lemma *nth-item-covers*: *finite-item x* ⟹ ∃ *n*. *nth-item n* = *x*
⟨*proof*⟩

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*⟩

⟨*ML*⟩

end

13.5 Automatically proving countability of datatypes

⟨*ML*⟩

13.6 More Countable types

Naturals

instance *nat* :: *countable*
⟨*proof*⟩

Pairs

instance *prod* :: (*countable*, *countable*) *countable*
⟨*proof*⟩

Sums

instance *sum* :: (*countable*, *countable*) *countable*
⟨*proof*⟩

Integers

instance *int* :: *countable*
⟨*proof*⟩

Options

instance *option* :: (*countable*) *countable*
⟨*proof*⟩

Lists

instance *list* :: (*countable*) *countable*
⟨*proof*⟩

String literals

instance *String.literal* :: *countable*
⟨*proof*⟩

Functions

instance *fun* :: (*finite*, *countable*) *countable*
⟨*proof*⟩

Typereps

instance *typerep* :: *countable*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *Rats-eq-range-nat-to-rat-surj*: ℚ = *range nat-to-rat-surj*
⟨*proof*⟩

context *field-char-0*
begin

lemma *Rats-eq-range-of-rat-o-nat-to-rat-surj*:
ℚ = *range* (*of-rat* ∘ *nat-to-rat-surj*)
⟨*proof*⟩

lemma *surj-of-rat-nat-to-rat-surj*:
r ∈ ℚ ⟹ ∃ *n*. *r* = *of-rat* (*nat-to-rat-surj n*)
⟨*proof*⟩

end

instance *rat* :: *countable*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *infinite-nat-iff-unbounded*: *infinite* (*S*::*nat set*) ⟷ (∀ *m*. ∃ *n*>*m*. *n* ∈ *S*)
⟨*proof*⟩

lemma *finite-nat-iff-bounded*: *finite* (*S*::*nat set*) ⟷ (∃ *k*. *S* ⊆ {..<*k*})
⟨*proof*⟩

lemma *finite-nat-iff-bounded-le*: *finite* (*S*::*nat set*) ⟷ (∃ *k*. *S* ⊆ {.. *k*})
⟨*proof*⟩

lemma *finite-nat-bounded*: *finite* (*S*::*nat set*) ⟹ ∃ *k*. *S* ⊆ {..<*k*}
⟨*proof*⟩

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*: ∀ *m*>*k*. ∃ *n*>*m*. *n* ∈ *S* ⟹ *infinite* (*S*::*nat set*)
⟨*proof*⟩

lemma *nat-not-finite*: *finite* (*UNIV*::*nat set*) ⟹ *R*
⟨*proof*⟩

lemma *range-inj-infinite*:
inj (*f*::*nat* ⇒ *'a*) ⟹ *infinite* (*range f*)
⟨*proof*⟩

The set of integers is also infinite.

lemma *infinite-int-iff-infinite-nat-abs*: *infinite* (*S*::*int set*) ⟷ *infinite* ((*nat o abs*) ‘ *S*)
⟨*proof*⟩

proposition *infinite-int-iff-unbounded-le*: *infinite* (*S*::*int set*) ⟷ (∀ *m*. ∃ *n*. |*n*| ≥ *m* ∧ *n* ∈ *S*)
⟨*proof*⟩

proposition *infinite-int-iff-unbounded*: *infinite* (*S*::*int set*) ⟷ (∀ *m*. ∃ *n*. |*n*| > *m* ∧ *n* ∈ *S*)
⟨*proof*⟩

proposition *finite-int-iff-bounded*: *finite* (*S*::*int set*) ⟷ (∃ *k*. *abs* ‘ *S* ⊆ {..<*k*})
⟨*proof*⟩

proposition *finite-int-iff-bounded-le*: *finite* (*S*::*int set*) ⟷ (∃ *k*. *abs* ‘ *S* ⊆ {.. *k*})
⟨*proof*⟩

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*) ⟨*proof*⟩
lemma *not-MOST* [*simp*]: ¬ (*MOST x*. *P x*) ⟷ (*INFM x*. ¬ *P x*) ⟨*proof*⟩

lemma *INFM-const* [*simp*]: $(INFM\ x{::}'a.\ P) \longleftrightarrow P \wedge infinite\ (UNIV{::}'a\ set)$
⟨*proof*⟩

lemma *MOST-const* [*simp*]: $(MOST\ x{::}'a.\ P) \longleftrightarrow P \vee finite\ (UNIV{::}'a\ set)$
⟨*proof*⟩

lemma *INFM-imp-distrib*: $(INFM\ x.\ P\ x \longrightarrow Q\ x) \longleftrightarrow ((MOST\ x.\ P\ x) \longrightarrow (INFM\ x.\ Q\ x))$
⟨*proof*⟩

lemma *MOST-imp-iff*: $MOST\ x.\ P\ x \Longrightarrow (MOST\ x.\ P\ x \longrightarrow Q\ x) \longleftrightarrow (MOST\ x.\ Q\ x)$
⟨*proof*⟩

lemma *INFM-conjI*: $INFM\ x.\ P\ x \Longrightarrow MOST\ x.\ Q\ x \Longrightarrow INFM\ x.\ P\ x \wedge Q\ x$
⟨*proof*⟩

Properties of quantifiers with injective functions.

lemma *INFM-inj*: $INFM\ x.\ P\ (f\ x) \Longrightarrow inj\ f \Longrightarrow INFM\ x.\ P\ x$
⟨*proof*⟩

lemma *MOST-inj*: $MOST\ x.\ P\ x \Longrightarrow inj\ f \Longrightarrow MOST\ x.\ P\ (f\ x)$
⟨*proof*⟩

Properties of quantifiers with singletons.

lemma *not-INFM-eq* [*simp*]:
$\neg\ (INFM\ x.\ x = a)$
$\neg\ (INFM\ x.\ a = x)$
⟨*proof*⟩

lemma *MOST-neq* [*simp*]:
$MOST\ x.\ x \neq a$
$MOST\ x.\ a \neq x$
⟨*proof*⟩

lemma *INFM-neq* [*simp*]:
$(INFM\ x{::}'a.\ x \neq a) \longleftrightarrow infinite\ (UNIV{::}'a\ set)$
$(INFM\ x{::}'a.\ a \neq x) \longleftrightarrow infinite\ (UNIV{::}'a\ set)$
⟨*proof*⟩

lemma *MOST-eq* [*simp*]:
$(MOST\ x{::}'a.\ x = a) \longleftrightarrow finite\ (UNIV{::}'a\ set)$
$(MOST\ x{::}'a.\ a = x) \longleftrightarrow finite\ (UNIV{::}'a\ set)$
⟨*proof*⟩

lemma *MOST-eq-imp*:
$MOST\ x.\ x = a \longrightarrow P\ x$
$MOST\ x.\ a = x \longrightarrow P\ x$

⟨*proof*⟩

Properties of quantifiers over the naturals.

lemma *MOST-nat*: $(\forall_\infty n.\ P\ (n::nat)) \longleftrightarrow (\exists\, m.\ \forall\, n{>}m.\ P\ n)$
⟨*proof*⟩

lemma *MOST-nat-le*: $(\forall_\infty n.\ P\ (n::nat)) \longleftrightarrow (\exists\, m.\ \forall\, n{\geq}m.\ P\ n)$
⟨*proof*⟩

lemma *INFM-nat*: $(\exists_\infty n.\ P\ (n::nat)) \longleftrightarrow (\forall\, m.\ \exists\, n{>}m.\ P\ n)$
⟨*proof*⟩

lemma *INFM-nat-le*: $(\exists_\infty n.\ P\ (n::nat)) \longleftrightarrow (\forall\, m.\ \exists\, n{\geq}m.\ P\ n)$
⟨*proof*⟩

lemma *MOST-INFM*: *infinite* $(UNIV::'a\ set) \Longrightarrow MOST\ x::'a.\ P\ x \Longrightarrow INFM\ x::'a.\ P\ x$
⟨*proof*⟩

lemma *MOST-Suc-iff*: $(MOST\ n.\ P\ (Suc\ n)) \longleftrightarrow (MOST\ n.\ P\ n)$
⟨*proof*⟩

lemma
shows *MOST-SucI*: $MOST\ n.\ P\ n \Longrightarrow MOST\ n.\ P\ (Suc\ n)$
and *MOST-SucD*: $MOST\ n.\ P\ (Suc\ n) \Longrightarrow MOST\ n.\ P\ n$
⟨*proof*⟩

lemma *MOST-ge-nat*: $MOST\ n::nat.\ m \leq n$
⟨*proof*⟩

lemma *Inf-many-def*: *Inf-many* $P \longleftrightarrow$ *infinite* $\{x.\ P\ x\}$ ⟨*proof*⟩
lemma *Alm-all-def*: *Alm-all* $P \longleftrightarrow \neg\ (INFM\ x.\ \neg\ P\ x)$ ⟨*proof*⟩
lemma *INFM-iff-infinite*: $(INFM\ x.\ P\ x) \longleftrightarrow$ *infinite* $\{x.\ P\ x\}$ ⟨*proof*⟩
lemma *MOST-iff-cofinite*: $(MOST\ x.\ P\ x) \longleftrightarrow$ *finite* $\{x.\ \neg\ P\ x\}$ ⟨*proof*⟩
lemma *INFM-EX*: $(\exists_\infty x.\ P\ x) \Longrightarrow (\exists\, x.\ P\ x)$ ⟨*proof*⟩
lemma *ALL-MOST*: $\forall\, x.\ P\ x \Longrightarrow \forall_\infty x.\ P\ x$ ⟨*proof*⟩
lemma *INFM-mono*: $\exists_\infty x.\ P\ x \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow Q\ x) \Longrightarrow \exists_\infty x.\ Q\ x$ ⟨*proof*⟩
lemma *MOST-mono*: $\forall_\infty x.\ P\ x \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow Q\ x) \Longrightarrow \forall_\infty x.\ Q\ x$ ⟨*proof*⟩
lemma *INFM-disj-distrib*: $(\exists_\infty x.\ P\ x \vee Q\ x) \longleftrightarrow (\exists_\infty x.\ P\ x) \vee (\exists_\infty x.\ Q\ x)$ ⟨*proof*⟩
lemma *MOST-rev-mp*: $\forall_\infty x.\ P\ x \Longrightarrow \forall_\infty x.\ P\ x \longrightarrow Q\ x \Longrightarrow \forall_\infty x.\ Q\ x$ ⟨*proof*⟩
lemma *MOST-conj-distrib*: $(\forall_\infty x.\ P\ x \wedge Q\ x) \longleftrightarrow (\forall_\infty x.\ P\ x) \wedge (\forall_\infty x.\ Q\ x)$ ⟨*proof*⟩
lemma *MOST-conjI*: $MOST\ x.\ P\ x \Longrightarrow MOST\ x.\ Q\ x \Longrightarrow MOST\ x.\ P\ x \wedge Q\ x$ ⟨*proof*⟩
lemma *INFM-finite-Bex-distrib*: *finite* $A \Longrightarrow (INFM\ y.\ \exists\, x{\in}A.\ P\ x\ y) \longleftrightarrow (\exists\, x{\in}A.\ INFM\ y.\ P\ x\ y)$ ⟨*proof*⟩
lemma *MOST-finite-Ball-distrib*: *finite* $A \Longrightarrow (MOST\ y.\ \forall\, x{\in}A.\ P\ x\ y) \longleftrightarrow$

$(\forall x{\in}A.\ MOST\ y.\ P\ x\ y)$ ⟨*proof*⟩
lemma *INFM-E*: $INFM\ x.\ P\ x \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow thesis) \Longrightarrow thesis$ ⟨*proof*⟩
lemma *MOST-I*: $(\bigwedge x.\ P\ x) \Longrightarrow MOST\ x.\ P\ x$ ⟨*proof*⟩
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* ⇒ *nat* ⇒ *'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'*: $enumerate\ S\ (Suc\ n) = enumerate\ (S - \{enumerate\ S\ 0\})\ n$
  ⟨*proof*⟩

lemma *enumerate-in-set*: $infinite\ S \Longrightarrow enumerate\ S\ n \in S$
  ⟨*proof*⟩

declare *enumerate-0* [*simp del*] *enumerate-Suc* [*simp del*]

lemma *enumerate-step*: $infinite\ S \Longrightarrow enumerate\ S\ n < enumerate\ S\ (Suc\ n)$
  ⟨*proof*⟩

lemma *enumerate-mono*: $m < n \Longrightarrow infinite\ S \Longrightarrow enumerate\ S\ m < enumerate\ S\ n$
  ⟨*proof*⟩

lemma *le-enumerate*:
  assumes *S*: *infinite S*
  shows $n \leq enumerate\ S\ n$
  ⟨*proof*⟩

lemma *enumerate-Suc''*:
  fixes *S* :: *'a*::*wellorder set*
  assumes *infinite S*
  shows $enumerate\ S\ (Suc\ n) = (LEAST\ s.\ s \in S \wedge enumerate\ S\ n < s)$
  ⟨*proof*⟩

lemma *enumerate-Ex*:
  assumes *S*: *infinite* (*S*::*nat set*)
  shows $s \in S \Longrightarrow \exists n.\ enumerate\ S\ n = s$
⟨*proof*⟩

lemma *bij-enumerate*:
fixes S :: *nat set*
assumes S: *infinite S*
shows *bij-betw* (*enumerate S*) *UNIV S*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *countableI*: *inj-on* (f::$'a$ ⇒ *nat*) *S* ⟹ *countable S*
⟨*proof*⟩

lemma *countableI'*: *inj-on* (f::$'a$ ⇒ $'b$::*countable*) *S* ⟹ *countable S*
⟨*proof*⟩

lemma *countableE-bij*:
assumes S: *countable S* **obtains** f :: *nat* ⇒ $'a$ **and** C :: *nat set* **where** *bij-betw f C S*
⟨*proof*⟩

lemma *countableI-bij*: *bij-betw f* (*C*::*nat set*) *S* ⟹ *countable S*
⟨*proof*⟩

lemma *countable-finite*: *finite S* ⟹ *countable S*
⟨*proof*⟩

lemma *countableI-bij1*: *bij-betw f A B* ⟹ *countable A* ⟹ *countable B*
⟨*proof*⟩

lemma *countableI-bij2*: *bij-betw f B A* ⟹ *countable A* ⟹ *countable B*
⟨*proof*⟩

lemma *countable-iff-bij*[*simp*]: *bij-betw f A B* ⟹ *countable A* ⟷ *countable B*
⟨*proof*⟩

lemma *countable-subset*: $A \subseteq B \Longrightarrow$ *countable* $B \Longrightarrow$ *countable* A
⟨*proof*⟩

lemma *countableI-type*[*intro*, *simp*]: *countable* (A:: $'a$ :: *countable set*)
⟨*proof*⟩

15.2 Enumerate a countable set

lemma *countableE-infinite*:
assumes *countable S infinite S*
obtains e :: $'a \Rightarrow$ *nat* **where** *bij-betw e S UNIV*
⟨*proof*⟩

lemma *countable-enum-cases*:
assumes *countable S*
obtains (*finite*) f :: $'a \Rightarrow$ *nat* **where** *finite S bij-betw f S* {..<*card S*}
| (*infinite*) f :: $'a \Rightarrow$ *nat* **where** *infinite S bij-betw f S UNIV*
⟨*proof*⟩

definition *to-nat-on* :: $'a$ *set* $\Rightarrow$ $'a \Rightarrow$ *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* $\Rightarrow$ *nat* $\Rightarrow$ $'a$ **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*}
⟨*proof*⟩

lemma *to-nat-on-infinite*: *countable S* $\Longrightarrow$ *infinite S* $\Longrightarrow$ *bij-betw* (*to-nat-on S*) *S UNIV*
⟨*proof*⟩

lemma *bij-betw-from-nat-into-finite*: *finite S* $\Longrightarrow$ *bij-betw* (*from-nat-into S*) {..< *card S*} *S*
⟨*proof*⟩

lemma *bij-betw-from-nat-into*: *countable S* $\Longrightarrow$ *infinite S* $\Longrightarrow$ *bij-betw* (*from-nat-into S*) *UNIV S*
⟨*proof*⟩

lemma *inj-on-to-nat-on*[*intro*]: *countable A* $\Longrightarrow$ *inj-on* (*to-nat-on A*) *A*
⟨*proof*⟩

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* $\longleftrightarrow$ $a = b$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *subset-range-from-nat-into*: *countable A* $\Longrightarrow$ *A* $\subseteq$ *range* (*from-nat-into A*)
⟨*proof*⟩

lemma *from-nat-into*: *A* $\neq$ {} $\Longrightarrow$ *from-nat-into A n* $\in$ *A*
⟨*proof*⟩

lemma *range-from-nat-into-subset*: *A* $\neq$ {} $\Longrightarrow$ *range* (*from-nat-into A*) $\subseteq$ *A*
⟨*proof*⟩

lemma *range-from-nat-into*[*simp*]: *A* $\neq$ {} $\Longrightarrow$ *countable A* $\Longrightarrow$ *range* (*from-nat-into A*) = *A*
⟨*proof*⟩

lemma *image-to-nat-on*: *countable A* $\Longrightarrow$ *infinite A* $\Longrightarrow$ *to-nat-on A ' A* = *UNIV*
⟨*proof*⟩

lemma *to-nat-on-surj*: *countable A* $\Longrightarrow$ *infinite A* $\Longrightarrow$ $\exists$ *a*$\in$*A*. *to-nat-on A a* = *n*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *from-nat-into-surj*: *countable A* $\Longrightarrow$ *a* $\in$ *A* $\Longrightarrow$ $\exists$ *n*. *from-nat-into A n* = *a*

⟨*proof*⟩

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* $B \longleftrightarrow A = B$
⟨*proof*⟩

lemma *inj-on-from-nat-into*: *inj-on from-nat-into* $(\{A.\ A \neq \{\} \wedge$ *countable* $A\})$
⟨*proof*⟩

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* $(SIGMA\ i : I.\ A\ i)$
⟨*proof*⟩

lemma *countable-image*[*intro*, *simp*]:
assumes *countable* A
shows *countable* $(f`A)$
⟨*proof*⟩

lemma *countable-image-inj-on*: *countable* $(f\ `\ A) \Longrightarrow$ *inj-on* $f\ A \Longrightarrow$ *countable* A
⟨*proof*⟩

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*⟩

lemma *countable-Un*[*intro*]: *countable* $A \Longrightarrow$ *countable* $B \Longrightarrow$ *countable* $(A \cup B)$
⟨*proof*⟩

lemma *countable-Un-iff*[*simp*]: *countable* $(A \cup B) \longleftrightarrow$ *countable* $A \wedge$ *countable* B
⟨*proof*⟩

lemma *countable-Plus*[*intro*, *simp*]:
countable $A \Longrightarrow$ *countable* $B \Longrightarrow$ *countable* $(A <+> B)$
⟨*proof*⟩

lemma *countable-empty*[*intro*, *simp*]: *countable* $\{\}$
⟨*proof*⟩

lemma *countable-insert*[*intro*, *simp*]: *countable* $A \Longrightarrow$ *countable* $(insert\ a\ A)$
⟨*proof*⟩

lemma *countable-Int1*[*intro*, *simp*]: *countable A* $\Longrightarrow$ *countable* $(A \cap B)$
⟨*proof*⟩

lemma *countable-Int2*[*intro*, *simp*]: *countable B* $\Longrightarrow$ *countable* $(A \cap B)$
⟨*proof*⟩

lemma *countable-INT*[*intro*, *simp*]: $i \in I \Longrightarrow$ *countable* $(A\ i) \Longrightarrow$ *countable* $(\bigcap i{\in}I.\ A\ i)$
⟨*proof*⟩

lemma *countable-Diff*[*intro*, *simp*]: *countable A* $\Longrightarrow$ *countable* $(A - B)$
⟨*proof*⟩

lemma *countable-insert-eq* [*simp*]: *countable* $(insert\ x\ A) =$ *countable A*
⟨*proof*⟩

lemma *countable-vimage*: $B \subseteq range\ f \Longrightarrow$ *countable* $(f\ -`\ B) \Longrightarrow$ *countable B*
⟨*proof*⟩

lemma *surj-countable-vimage*: *surj f* $\Longrightarrow$ *countable* $(f\ -`\ B) \Longrightarrow$ *countable B*
⟨*proof*⟩

lemma *countable-Collect*[*simp*]: *countable A* $\Longrightarrow$ *countable* $\{a \in A.\ \varphi\ a\}$
⟨*proof*⟩

lemma *countable-Image*:
assumes $\bigwedge y.\ y \in Y \Longrightarrow$ *countable* $(X\ ``\ \{y\})$
assumes *countable Y*
shows *countable* $(X\ ``\ Y)$
⟨*proof*⟩

lemma *countable-relpow*:
fixes $X :: 'a\ rel$
assumes *Image-X*: $\bigwedge Y.$ *countable Y* $\Longrightarrow$ *countable* $(X\ ``\ Y)$
assumes *Y*: *countable Y*
shows *countable* $((X\ \hat{}\hat{}\ i)\ ``\ Y)$
⟨*proof*⟩

lemma *countable-funpow*:
fixes $f :: 'a\ set \Rightarrow 'a\ set$
assumes $\bigwedge A.$ *countable A* $\Longrightarrow$ *countable* $(f\ A)$
and *countable A*
shows *countable* $((f\ \hat{}\hat{}\ n)\ A)$
⟨*proof*⟩

lemma *countable-rtrancl*:
$(\bigwedge Y.$ *countable Y* $\Longrightarrow$ *countable* $(X\ ``\ Y)) \Longrightarrow$ *countable Y* $\Longrightarrow$ *countable* $(X\hat{}*\ ``\ Y)$
⟨*proof*⟩

lemma *countable-lists*[*intro*, *simp*]:
assumes *A*: *countable A* **shows** *countable* (*lists A*)
⟨*proof*⟩

lemma *Collect-finite-eq-lists*: *Collect finite* = *set* ‘ *lists UNIV*
⟨*proof*⟩

lemma *countable-Collect-finite*: *countable* (*Collect* (*finite*::′*a*::*countable set*⇒*bool*))
⟨*proof*⟩

lemma *countable-rat*: *countable* ℚ
⟨*proof*⟩

lemma *Collect-finite-subset-eq-lists*: {*A*. *finite A* ∧ *A* ⊆ *T*} = *set* ‘ *lists T*
⟨*proof*⟩

lemma *countable-Collect-finite-subset*:
countable T ⟹ *countable* {*A*. *finite A* ∧ *A* ⊆ *T*}
⟨*proof*⟩

lemma *countable-set-option* [*simp*]: *countable* (*set-option x*)
⟨*proof*⟩

15.4 Misc lemmas

lemma *infinite-countable-subset′*:
assumes *X*: *infinite X* **shows** ∃ *C*⊆*X*. *countable C* ∧ *infinite C*
⟨*proof*⟩

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*))
⟨*proof*⟩

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*⟩

lemma *transfer-countable*[*transfer-rule*]:
bi-unique R ⟹ *rel-fun* (*rel-set R*) *op* = *countable countable*
⟨*proof*⟩

15.5 Uncountable

abbreviation *uncountable* **where**
uncountable A ≡ ¬ *countable A*

lemma *uncountable-def*: *uncountable* $A \longleftrightarrow A \neq \{\} \wedge \neg\ (\exists f{::}(nat \Rightarrow 'a).\ range\ f = A)$
⟨*proof*⟩

lemma *uncountable-bij-betw*: *bij-betw* $f\ A\ B \Longrightarrow$ *uncountable* $B \Longrightarrow$ *uncountable* A
⟨*proof*⟩

lemma *uncountable-infinite*: *uncountable* $A \Longrightarrow$ *infinite* A
⟨*proof*⟩

lemma *uncountable-minus-countable*:
uncountable $A \Longrightarrow$ *countable* $B \Longrightarrow$ *uncountable* $(A - B)$
⟨*proof*⟩

lemma *countable-Diff-eq* [*simp*]: *countable* $(A - \{x\}) =$ *countable* A
⟨*proof*⟩

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*⟩

lemma *uncountable-UNIV-real*: *uncountable* $(UNIV{::}real\ set)$
⟨*proof*⟩

lemma *bij-betw-open-intervals*:
fixes *a b c d* :: *real*
assumes $a < b$ $c < d$
shows $\exists f.$ *bij-betw f* $\{a<..<b\}$ $\{c<..<d\}$
⟨*proof*⟩

lemma *bij-betw-tan*: *bij-betw tan* $\{-pi/2<..<pi/2\}$ *UNIV*
⟨*proof*⟩

lemma *uncountable-open-interval*:
fixes *a b* :: *real*
shows *uncountable* $\{a<..<b\} \longleftrightarrow a < b$
⟨*proof*⟩

lemma *uncountable-half-open-interval-1*:
fixes *a* :: *real* **shows** *uncountable* $\{a..<b\} \longleftrightarrow a<b$
⟨*proof*⟩

lemma *uncountable-half-open-interval-2*:
fixes *a* :: *real* **shows** *uncountable* $\{a<..b\} \longleftrightarrow a<b$
⟨*proof*⟩

lemma *real-interval-avoid-countable-set*:
fixes *a b* :: *real* **and** *A* :: *real set*
assumes $a < b$ **and** *countable A*
shows $\exists x \in \{a<..<b\}.\ x \notin A$
⟨*proof*⟩

lemma *open-minus-countable*:
fixes *S A* :: *real set* **assumes** *countable A S* $\neq \{\}$ *open S*
shows $\exists x \in S.\ x \notin A$
⟨*proof*⟩

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*.

⟨*ML*⟩

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* ≤ *inner x x*
  and *inner-eq-zero-iff* [*simp*]: *inner x x* = *0* ⟷ *x* = *0*
  and *norm-eq-sqrt-inner*: *norm x* = *sqrt* (*inner x x*)
begin

lemma *inner-zero-left* [*simp*]: *inner 0 x* = *0*
  ⟨*proof*⟩

lemma *inner-minus-left* [*simp*]: *inner* (− *x*) *y* = − *inner x y*
  ⟨*proof*⟩

lemma *inner-diff-left*: *inner* (*x* − *y*) *z* = *inner x z* − *inner y z*
  ⟨*proof*⟩

lemma *inner-setsum-left*: *inner* (∑ *x*∈*A*. *f x*) *y* = (∑ *x*∈*A*. *inner* (*f x*) *y*)
  ⟨*proof*⟩

Transfer distributivity rules to right argument.

lemma *inner-add-right*: *inner x* (*y* + *z*) = *inner x y* + *inner x z*
  ⟨*proof*⟩

lemma *inner-scaleR-right* [*simp*]: *inner x* (*scaleR r y*) = *r* ∗ (*inner x y*)
  ⟨*proof*⟩

lemma *inner-zero-right* [*simp*]: *inner x 0* = *0*
  ⟨*proof*⟩

lemma *inner-minus-right* [*simp*]: *inner x* (− *y*) = − *inner x y*
  ⟨*proof*⟩

lemma *inner-diff-right*: *inner x* (*y* − *z*) = *inner x y* − *inner x z*
  ⟨*proof*⟩

lemma *inner-setsum-right*: *inner x* (∑ *y*∈*A*. *f y*) = (∑ *y*∈*A*. *inner x* (*f y*))
  ⟨*proof*⟩

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*]: $0 < inner\ x\ x \longleftrightarrow x \neq 0$
⟨*proof*⟩

lemma *power2-norm-eq-inner*: $(norm\ x)^2 = inner\ x\ x$
⟨*proof*⟩

Identities involving real multiplication and division.

lemma *inner-mult-left*: $inner\ (of\text{-}real\ m * a)\ b = m * (inner\ a\ b)$
⟨*proof*⟩

lemma *inner-mult-right*: $inner\ a\ (of\text{-}real\ m * b) = m * (inner\ a\ b)$
⟨*proof*⟩

lemma *inner-mult-left'*: $inner\ (a * of\text{-}real\ m)\ b = m * (inner\ a\ b)$
⟨*proof*⟩

lemma *inner-mult-right'*: $inner\ a\ (b * of\text{-}real\ m) = (inner\ a\ b) * m$
⟨*proof*⟩

lemma *Cauchy-Schwarz-ineq*:
$(inner\ x\ y)^2 \leq inner\ x\ x * inner\ y\ y$
⟨*proof*⟩

lemma *Cauchy-Schwarz-ineq2*:
$|inner\ x\ y| \leq norm\ x * norm\ y$
⟨*proof*⟩

lemma *norm-cauchy-schwarz*: $inner\ x\ y \leq norm\ x * norm\ y$
⟨*proof*⟩

subclass *real-normed-vector*
⟨*proof*⟩

end

lemma *inner-divide-left*:
fixes $a :: 'a :: \{real\text{-}inner, real\text{-}div\text{-}algebra\}$
shows $inner\ (a\ /\ of\text{-}real\ m)\ b = (inner\ a\ b)\ /\ m$
⟨*proof*⟩

lemma *inner-divide-right*:
fixes $a :: 'a :: \{real\text{-}inner, real\text{-}div\text{-}algebra\}$
shows $inner\ a\ (b\ /\ of\text{-}real\ m) = (inner\ a\ b)\ /\ m$
⟨*proof*⟩

Re-enable constraints for *open*, *uniformity*, *dist*, and *norm*.

⟨*ML*⟩

lemma *bounded-bilinear-inner*:
bounded-bilinear (*inner*::'*a*::*real-inner* ⇒ '*a* ⇒ *real*)
⟨*proof*⟩

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*) ⟹ *g differentiable at x within s* ⟹ (λ*x*. *inner* (*f x*) (*g x*)) *differentiable at x within s*
⟨*proof*⟩

17.2 Class instances

instantiation *real* :: *real-inner*
begin

definition *inner-real-def* [*simp*]: *inner* = *op* ∗

instance
⟨*proof*⟩

end

instantiation *complex* :: *real-inner*
begin

definition *inner-complex-def*:
*inner x y = Re x * Re y + Im x * Im y*

instance
⟨*proof*⟩

end

lemma *complex-inner-1* [*simp*]: *inner 1 x = Re x*
⟨*proof*⟩

lemma *complex-inner-1-right* [*simp*]: *inner x 1 = Re x*
⟨*proof*⟩

lemma *complex-inner-ii-left* [*simp*]: *inner ii x = Im x*
⟨*proof*⟩

lemma *complex-inner-ii-right* [*simp*]: *inner x ii = Im x*
⟨*proof*⟩

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 :> (λh. inner h D)

lemma *gderiv-deriv* [*simp*]: *GDERIV f x :> D ⟷ DERIV f x :> D*
⟨*proof*⟩

lemma *GDERIV-DERIV-compose*:
⟦GDERIV f x :> df; DERIV g (f x) :> dg⟧
⟹ GDERIV (λx. g (f x)) x :> scaleR dg df
⟨*proof*⟩

lemma *has-derivative-subst*: *⟦FDERIV f x :> df; df = d⟧ ⟹ FDERIV f x :> d*
⟨*proof*⟩

lemma *GDERIV-subst*: *⟦GDERIV f x :> df; df = d⟧ ⟹ GDERIV f x :> d*
⟨*proof*⟩

lemma *GDERIV-const*: *GDERIV (λx. k) x :> 0*
⟨*proof*⟩

lemma *GDERIV-add*:
⟦GDERIV f x :> df; GDERIV g x :> dg⟧

$\Longrightarrow$ *GDERIV* $(\lambda x.\ f\ x + g\ x)\ x :> df + dg$
⟨*proof*⟩

lemma *GDERIV-minus*:
$GDERIV\ f\ x :> df \Longrightarrow GDERIV\ (\lambda x.\ -f\ x)\ x :> -df$
⟨*proof*⟩

lemma *GDERIV-diff*:
$[\![GDERIV\ f\ x :> df;\ GDERIV\ g\ x :> dg]\!]$
$\Longrightarrow GDERIV\ (\lambda x.\ f\ x - g\ x)\ x :> df - dg$
⟨*proof*⟩

lemma *GDERIV-scaleR*:
$[\![DERIV\ f\ x :> df;\ GDERIV\ g\ x :> dg]\!]$
$\Longrightarrow GDERIV\ (\lambda x.\ scaleR\ (f\ x)\ (g\ x))\ x$
$:> (scaleR\ (f\ x)\ dg + scaleR\ df\ (g\ x))$
⟨*proof*⟩

lemma *GDERIV-mult*:
$[\![GDERIV\ f\ x :> df;\ GDERIV\ g\ x :> dg]\!]$
$\Longrightarrow GDERIV\ (\lambda x.\ f\ x * g\ x)\ x :> scaleR\ (f\ x)\ dg + scaleR\ (g\ x)\ df$
⟨*proof*⟩

lemma *GDERIV-inverse*:
$[\![GDERIV\ f\ x :> df;\ f\ x \neq 0]\!]$
$\Longrightarrow GDERIV\ (\lambda x.\ inverse\ (f\ x))\ x :> -(inverse\ (f\ x))^2 *_R df$
⟨*proof*⟩

lemma *GDERIV-norm*:
assumes $x \neq 0$ **shows** $GDERIV\ (\lambda x.\ norm\ x)\ x :> sgn\ x$
⟨*proof*⟩

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 ⟨*proof*⟩
end

instantiation *prod* :: (*plus*, *plus*) *plus*
begin

definition *plus-prod-def*:
$x + y = (fst\ x + fst\ y,\ snd\ x + snd\ y)$

instance ⟨*proof*⟩
end

instantiation *prod* :: (*minus*, *minus*) *minus*
begin

definition *minus-prod-def*:
$x - y = (fst\ x - fst\ y,\ snd\ x - snd\ y)$

instance ⟨*proof*⟩
end

instantiation *prod* :: (*uminus*, *uminus*) *uminus*
begin

definition *uminus-prod-def*:
$- x = (-\ fst\ x,\ -\ snd\ x)$

instance ⟨*proof*⟩
end

lemma *fst-zero* [*simp*]: $fst\ 0 = 0$
⟨*proof*⟩

lemma *snd-zero* [*simp*]: $snd\ 0 = 0$
⟨*proof*⟩

lemma *fst-add* [*simp*]: $fst\ (x + y) = fst\ x + fst\ y$
⟨*proof*⟩

lemma *snd-add* [*simp*]: $snd\ (x + y) = snd\ x + snd\ y$
⟨*proof*⟩

lemma *fst-diff* [*simp*]: $fst\ (x - y) = fst\ x - fst\ y$
⟨*proof*⟩

lemma *snd-diff* [*simp*]: $snd\ (x - y) = snd\ x - snd\ y$
⟨*proof*⟩

lemma *fst-uminus* [*simp*]: $fst\ (-\ x) = -\ fst\ x$

⟨*proof*⟩

lemma *snd-uminus* [*simp*]: *snd* (− *x*) = − *snd x*
⟨*proof*⟩

lemma *add-Pair* [*simp*]: (*a*, *b*) + (*c*, *d*) = (*a* + *c*, *b* + *d*)
⟨*proof*⟩

lemma *diff-Pair* [*simp*]: (*a*, *b*) − (*c*, *d*) = (*a* − *c*, *b* − *d*)
⟨*proof*⟩

lemma *uminus-Pair* [*simp*, *code*]: − (*a*, *b*) = (− *a*, − *b*)
⟨*proof*⟩

18.2 Class instances

instance *prod* :: (*semigroup-add*, *semigroup-add*) *semigroup-add*
⟨*proof*⟩

instance *prod* :: (*ab-semigroup-add*, *ab-semigroup-add*) *ab-semigroup-add*
⟨*proof*⟩

instance *prod* :: (*monoid-add*, *monoid-add*) *monoid-add*
⟨*proof*⟩

instance *prod* :: (*comm-monoid-add*, *comm-monoid-add*) *comm-monoid-add*
⟨*proof*⟩

instance *prod* :: (*cancel-semigroup-add*, *cancel-semigroup-add*) *cancel-semigroup-add*
⟨*proof*⟩

instance *prod* :: (*cancel-ab-semigroup-add*, *cancel-ab-semigroup-add*) *cancel-ab-semigroup-add*
⟨*proof*⟩

instance *prod* :: (*cancel-comm-monoid-add*, *cancel-comm-monoid-add*) *cancel-comm-monoid-add*
⟨*proof*⟩

instance *prod* :: (*group-add*, *group-add*) *group-add*
⟨*proof*⟩

instance *prod* :: (*ab-group-add*, *ab-group-add*) *ab-group-add*
⟨*proof*⟩

lemma *fst-setsum*: *fst* (∑ *x*∈*A*. *f x*) = (∑ *x*∈*A*. *fst* (*f x*))
⟨*proof*⟩

lemma *snd-setsum*: *snd* (∑ *x*∈*A*. *f x*) = (∑ *x*∈*A*. *snd* (*f x*))
⟨*proof*⟩

lemma *setsum-prod*: $(\sum x{\in}A.\ (f\ x,\ g\ x)) = (\sum x{\in}A.\ f\ x,\ \sum x{\in}A.\ g\ x)$
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *snd-scaleR* [*simp*]: *snd* (*scaleR r A*) = *scaleR r* (*snd A*)
⟨*proof*⟩

lemma *scaleR-Pair* [*simp*]: *scaleR r* (*a*, *b*) = (*scaleR r a*, *scaleR r b*)
⟨*proof*⟩

instance
⟨*proof*⟩

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 ⟨*proof*⟩
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
⟨*proof*⟩
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)
⟨*proof*⟩

lemma *dist-fst-le*: *dist* (*fst* x) (*fst* y) ≤ *dist* x y
⟨*proof*⟩

lemma *dist-snd-le*: *dist* (*snd* x) (*snd* y) ≤ *dist* x y
⟨*proof*⟩

instance
⟨*proof*⟩

end

declare [[*code abort*: *dist*::($'a$::*metric-space*∗$'b$::*metric-space*)⇒($'a$∗$'b$) ⇒ *real*]]

lemma *Cauchy-fst*: *Cauchy* X ⟹ *Cauchy* (λn. *fst* (X n))
⟨*proof*⟩

lemma *Cauchy-snd*: *Cauchy* X ⟹ *Cauchy* (λn. *snd* (X n))
⟨*proof*⟩

lemma *Cauchy-Pair*:
assumes *Cauchy* X **and** *Cauchy* Y
shows *Cauchy* (λn. (X n, Y n))
⟨*proof*⟩

19.3 Product is a complete metric space

instance *prod* :: (*complete-space*, *complete-space*) *complete-space*
⟨*proof*⟩

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* × '*b*) = *scaleR* (*inverse* (*norm x*)) *x*

lemma *norm-Pair*: *norm* (*a*, *b*) = *sqrt* ((*norm a*)^2 + (*norm b*)^2)
⟨*proof*⟩

instance
⟨*proof*⟩

end

declare [[*code abort*: *norm*::('*a*::*real-normed-vector* * '*b*::*real-normed-vector*) ⇒ *real*]]

instance *prod* :: (*banach*, *banach*) *banach* ⟨*proof*⟩

19.4.1 Pair operations are linear

lemma *bounded-linear-fst*: *bounded-linear fst*
⟨*proof*⟩

lemma *bounded-linear-snd*: *bounded-linear snd*
⟨*proof*⟩

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* (λ*x*. (*f x*, *g x*))
⟨*proof*⟩

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 ((λ*x*. (*f x*, *g x*)) *has-derivative* (λ*h*. (*f' h*, *g' h*))) (*at x within s*)
⟨*proof*⟩

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*]:
((λ*p*. *f* (*fst p*) (*snd p*)) *has-derivative f'*) *F* ⟹ ((λ(*a*, *b*). *f a b*) *has-derivative f'*) *F*

⟨*proof*⟩

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*
⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *inner-Pair-0*: *inner x* (*0*, *b*) = *inner* (*snd x*) *b inner x* (*a*, *0*) = *inner* (*fst x*) *a*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *norm-commute*: *norm* (*x*,*y*) = *norm* (*y*,*x*)
⟨*proof*⟩

lemma *norm-fst-le*: *norm x* ≤ *norm* (*x*,*y*)
⟨*proof*⟩

lemma *norm-snd-le*: *norm y* ≤ *norm* (*x*,*y*)
⟨*proof*⟩

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* $\longleftrightarrow$ $(\forall x{\in}s.\ \forall y{\in}s.\ \forall u{\geq}0.\ \forall v{\geq}0.\ u + v = 1 \longrightarrow u *_R x + v *_R y \in s)$

lemma *convexI*:
assumes $\bigwedge x\ y\ u\ v.\ x \in s \Longrightarrow y \in s \Longrightarrow 0 \leq u \Longrightarrow 0 \leq v \Longrightarrow u + v = 1 \Longrightarrow u *_R x + v *_R y \in s$
shows *convex s*
⟨*proof*⟩

lemma *convexD*:
assumes *convex s* **and** $x \in s$ **and** $y \in s$ **and** $0 \leq u$ **and** $0 \leq v$ **and** $u + v = 1$
shows $u *_R x + v *_R y \in s$
⟨*proof*⟩

lemma *convex-alt*:
convex s $\longleftrightarrow$ $(\forall x{\in}s.\ \forall y{\in}s.\ \forall u.\ 0 \leq u \wedge u \leq 1 \longrightarrow ((1 - u) *_R x + u *_R y) \in s)$
(**is** - $\longleftrightarrow$ *?alt*)
⟨*proof*⟩

lemma *convexD-alt*:
assumes *convex s* $a \in s$ $b \in s$ $0 \leq u$ $u \leq 1$
shows $((1 - u) *_R a + u *_R b) \in s$
⟨*proof*⟩

lemma *mem-convex-alt*:
assumes *convex S* $x \in S$ $y \in S$ $u \geq 0$ $v \geq 0$ $u + v > 0$
shows $((u/(u+v)) *_R x + (v/(u+v)) *_R y) \in S$
⟨*proof*⟩

lemma *convex-empty*[*intro*,*simp*]: *convex* $\{\}$
⟨*proof*⟩

lemma *convex-singleton*[*intro*,*simp*]: *convex* $\{a\}$
⟨*proof*⟩

lemma *convex-UNIV*[*intro*,*simp*]: *convex UNIV*
⟨*proof*⟩

lemma *convex-Inter*: $(\forall s{\in}f.\ convex\ s) \Longrightarrow convex(\bigcap f)$
⟨*proof*⟩

lemma *convex-Int*: *convex s* $\Longrightarrow$ *convex t* $\Longrightarrow$ *convex* $(s \cap t)$
⟨*proof*⟩

lemma *convex-INT*: $\forall i{\in}A.\ convex\ (B\ i) \Longrightarrow convex\ (\bigcap i{\in}A.\ B\ i)$
⟨*proof*⟩

lemma *convex-Times*: *convex s* $\Longrightarrow$ *convex t* $\Longrightarrow$ *convex* $(s \times t)$

⟨*proof*⟩

lemma *convex-halfspace-le*: *convex* {*x*. *inner a x* ≤ *b*}
⟨*proof*⟩

lemma *convex-halfspace-ge*: *convex* {*x*. *inner a x* ≥ *b*}
⟨*proof*⟩

lemma *convex-hyperplane*: *convex* {*x*. *inner a x* = *b*}
⟨*proof*⟩

lemma *convex-halfspace-lt*: *convex* {*x*. *inner a x* < *b*}
⟨*proof*⟩

lemma *convex-halfspace-gt*: *convex* {*x*. *inner a x* > *b*}
⟨*proof*⟩

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*⟩

lemma *convex-Reals*: *convex* ℝ
⟨*proof*⟩

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 (∑ *i* ∈ *s*. *a i*) = *1*
assumes ⋀*i*. *i* ∈ *s* ⟹ *a i* ≥ *0*
and ⋀*i*. *i* ∈ *s* ⟹ *y i* ∈ *C*
shows (∑ *j* ∈ *s*. *a j* ∗*R* *y j*) ∈ *C*
⟨*proof*⟩

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* (λ*i*. *u i* ∗*R* *x i*) {*1*..*k*} ∈ *s*)
⟨*proof*⟩

lemma *convex-explicit*:

fixes s :: $'a$::*real-vector set*
shows *convex* $s \longleftrightarrow$
$(\forall\, t\ u.\ \mathit{finite}\ t \wedge t \subseteq s \wedge (\forall\, x{\in}t.\ 0 \leq u\ x) \wedge \mathit{setsum}\ u\ t = 1 \longrightarrow \mathit{setsum}\ (\lambda x.\ u\ x *_R x)\ t \in s)$
⟨*proof*⟩

lemma *convex-finite*:
assumes *finite* s
shows *convex* $s \longleftrightarrow (\forall\, u.\ (\forall\, x{\in}s.\ 0 \leq u\ x) \wedge \mathit{setsum}\ u\ s = 1 \longrightarrow \mathit{setsum}\ (\lambda x.\ u\ x *_R x)\ s \in s)$
⟨*proof*⟩

20.3 Functions that are convex on a set

definition *convex-on* :: $'a$::*real-vector set* ⇒ ($'a$ ⇒ *real*) ⇒ *bool*
where *convex-on* $s\ f \longleftrightarrow$
$(\forall\, x{\in}s.\ \forall\, y{\in}s.\ \forall\, u{\geq}0.\ \forall\, v{\geq}0.\ u + v = 1 \longrightarrow f\ (u *_R x + v *_R y) \leq u * f\ x + v * f\ y)$

lemma *convex-onI* [*intro?*]:
assumes $\bigwedge t\ x\ y.\ t > 0 \Longrightarrow t < 1 \Longrightarrow x \in A \Longrightarrow y \in A \Longrightarrow$
$f\ ((1 - t) *_R x + t *_R y) \leq (1 - t) * f\ x + t * f\ y$
shows *convex-on* $A\ f$
⟨*proof*⟩

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*⟩

lemma *convex-onD*:
assumes *convex-on* $A\ f$
shows $\bigwedge t\ x\ y.\ t \geq 0 \Longrightarrow t \leq 1 \Longrightarrow x \in A \Longrightarrow y \in A \Longrightarrow$
$f\ ((1 - t) *_R x + t *_R y) \leq (1 - t) * f\ x + t * f\ y$
⟨*proof*⟩

lemma *convex-onD-Icc*:
assumes *convex-on* $\{x..y\}\ f\ x \leq (y :: - :: \{\mathit{real\text{-}vector},\mathit{preorder}\})$
shows $\bigwedge t.\ t \geq 0 \Longrightarrow t \leq 1 \Longrightarrow$
$f\ ((1 - t) *_R x + t *_R y) \leq (1 - t) * f\ x + t * f\ y$
⟨*proof*⟩

lemma *convex-on-subset*: *convex-on* $t\ f \Longrightarrow s \subseteq t \Longrightarrow$ *convex-on* $s\ f$
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

lemma *convex-on-dist* [*intro*]:
fixes *s* :: *'a*::*real-normed-vector set*
shows *convex-on s* ($\lambda x.\ dist\ a\ x$)
⟨*proof*⟩

20.4 Arithmetic operations on sets preserve convexity

lemma *convex-linear-image*:
assumes *linear f*
and *convex s*
shows *convex* (f ‘ s)
⟨*proof*⟩

lemma *convex-linear-vimage*:
assumes *linear f*
and *convex s*
shows *convex* (f −‘ s)
⟨*proof*⟩

lemma *convex-scaling*:
assumes *convex s*
shows *convex* (($\lambda x.\ c *_R x$) ‘ s)
⟨*proof*⟩

lemma *convex-scaled*:
assumes *convex s*
shows *convex* (($\lambda x.\ x *_R c$) ‘ s)
⟨*proof*⟩

lemma *convex-negations*:
assumes *convex s*
shows *convex* $((\lambda x.\ -\ x)\ `\ s)$
⟨*proof*⟩

lemma *convex-sums*:
assumes *convex s*
and *convex t*
shows *convex* $\{x + y |\ x\ y.\ x \in s \wedge y \in t\}$
⟨*proof*⟩

lemma *convex-differences*:
assumes *convex s convex t*
shows *convex* $\{x - y |\ x\ y.\ x \in s \wedge y \in t\}$
⟨*proof*⟩

lemma *convex-translation*:
assumes *convex s*
shows *convex* $((\lambda x.\ a + x)\ `\ s)$
⟨*proof*⟩

lemma *convex-affinity*:
assumes *convex s*
shows *convex* $((\lambda x.\ a + c *_R x)\ `\ s)$
⟨*proof*⟩

lemma *pos-is-convex*: *convex* $\{0 :: real <..\}$
⟨*proof*⟩

lemma *convex-on-setsum*:
fixes $a :: 'a \Rightarrow real$
and $y :: 'a \Rightarrow 'b$::*real-vector*
and $f :: 'b \Rightarrow real$
assumes *finite s* $s \neq \{\}$
and *convex-on C f*
and *convex C*
and $(\sum\ i \in s.\ a\ i) = 1$
and $\bigwedge i.\ i \in s \Longrightarrow a\ i \geq 0$
and $\bigwedge i.\ i \in s \Longrightarrow y\ i \in C$
shows $f\ (\sum\ i \in s.\ a\ i *_R y\ i) \leq (\sum\ i \in s.\ a\ i * f\ (y\ i))$
⟨*proof*⟩

lemma *convex-on-alt*:
fixes $C :: 'a$::*real-vector set*
assumes *convex C*
shows *convex-on C f* $\longleftrightarrow$
$(\forall x \in C.\ \forall\ y \in C.\ \forall\ \mu :: real.\ \mu \geq 0 \wedge \mu \leq 1 \longrightarrow$
$f\ (\mu *_R x + (1 - \mu) *_R y) \leq \mu * f\ x + (1 - \mu) * f\ y)$

⟨*proof*⟩

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 $(f\ x - f\ t) / (x - t) \leq (f\ x - f\ y) / (x - y)$
and $(f\ x - f\ y) / (x - y) \leq (f\ t - f\ y) / (t - y)$
⟨*proof*⟩

lemma *pos-convex-function*:
fixes f :: *real* ⇒ *real*
assumes *convex* C
and *leq*: $\bigwedge x\ y.\ x \in C \Longrightarrow y \in C \Longrightarrow f'\ x * (y - x) \leq f\ y - f\ x$
shows *convex-on* C f
⟨*proof*⟩

lemma *atMostAtLeast-subset-convex*:
fixes C :: *real set*
assumes *convex* C
and $x \in C$ $y \in C$ $x < y$
shows $\{x\ ..\ y\} \subseteq C$
⟨*proof*⟩

lemma *f''-imp-f'*:
fixes f :: *real* ⇒ *real*
assumes *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*: $\bigwedge x.\ x \in C \Longrightarrow f''\ x \geq 0$
and $x \in C$ $y \in C$
shows $f'\ x * (y - x) \leq f\ y - f\ x$
⟨*proof*⟩

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
⟨*proof*⟩

lemma *minus-log-convex*:
fixes b :: *real*
assumes $b > 1$
shows *convex-on* $\{0 <..\}$ $(\lambda\ x.\ -\ log\ b\ x)$
⟨*proof*⟩

20.5 Convexity of real functions

lemma *convex-on-realI*:
assumes *connected A*
assumes $\bigwedge x.\ x \in A \Longrightarrow (f\ \textit{has-real-derivative}\ f'\ x)\ (\textit{at}\ x)$
assumes $\bigwedge x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow x \leq y \Longrightarrow f'\ x \leq f'\ y$
shows *convex-on A f*
⟨*proof*⟩

lemma *convex-on-inverse*:
assumes $A \subseteq \{0<..\}$
shows *convex-on A* (*inverse* :: *real* ⇒ *real*)
⟨*proof*⟩

lemma *convex-onD-Icc'*:
assumes *convex-on* $\{x..y\}$ *f c* ∈ $\{x..y\}$
defines $d \equiv y - x$
shows $f\ c \leq (f\ y - f\ x)\ /\ d * (c - x) + f\ x$
⟨*proof*⟩

lemma *convex-onD-Icc''*:
assumes *convex-on* $\{x..y\}$ *f c* ∈ $\{x..y\}$
defines $d \equiv y - x$
shows $f\ c \leq (f\ x - f\ y)\ /\ d * (y - c) + f\ y$
⟨*proof*⟩

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* (⨅-./ -) [*0*, *10*] *10*)
-INF :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b* (⨅-∈-./ -) [*0*, *0*, *10*] *10*)
-SUP1 :: *pttrns* ⇒ *'b* ⇒ *'b* (⨆-./ -) [*0*, *10*] *10*)
-SUP :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b* (⨆-∈-./ -) [*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* ⟨*proof*⟩

lemma *chain-transfer* [*transfer-rule*]:
((*A* ===> *A* ===> *op* =) ===> *rel-set A* ===> *op* =) *Complete-Partial-Order.chain Complete-Partial-Order.chain*
⟨*proof*⟩

end

lemma *linorder-chain* [*simp*, *intro*!]:
fixes *Y* :: - :: *linorder set*
shows *Complete-Partial-Order.chain op* ≤ *Y*
⟨*proof*⟩

lemma *fun-lub-apply*: ⋀*Sup*. *fun-lub Sup Y x* = *Sup* ((λ*f*. *f x*) ‘ *Y*)
⟨*proof*⟩

lemma *fun-lub-empty* [*simp*]: *fun-lub lub* {} = (λ-. *lub* {})
⟨*proof*⟩

lemma *chain-fun-ordD*:
assumes *Complete-Partial-Order.chain* (*fun-ord le*) *Y*
shows *Complete-Partial-Order.chain le* ((λ*f*. *f x*) ‘ *Y*)
⟨*proof*⟩

lemma *chain-Diff*:
Complete-Partial-Order.chain ord A
⟹ *Complete-Partial-Order.chain ord* (*A* − *B*)
⟨*proof*⟩

lemma *chain-rel-prodD1*:
Complete-Partial-Order.chain (*rel-prod orda ordb*) *Y*
⟹ *Complete-Partial-Order.chain orda* (*fst* ‘ *Y*)
⟨*proof*⟩

lemma *chain-rel-prodD2*:
Complete-Partial-Order.chain (*rel-prod orda ordb*) *Y*
⟹ *Complete-Partial-Order.chain ordb* (*snd* ‘ *Y*)

⟨*proof*⟩

context *ccpo* **begin**

lemma *ccpo-fun*: *class.ccpo* (*fun-lub Sup*) (*fun-ord op* ≤) (*mk-less* (*fun-ord op* ≤))
⟨*proof*⟩

lemma *ccpo-Sup-below-iff*: *Complete-Partial-Order.chain op* ≤ *Y* ⟹ *Sup Y* ≤ *x* ⟷ (∀ *y*∈*Y*. *y* ≤ *x*)
⟨*proof*⟩

lemma *Sup-minus-bot*:
assumes *chain*: *Complete-Partial-Order.chain op* ≤ *A*
shows ⨆(*A* − {⨆{}}) = ⨆*A*
⟨*proof*⟩

lemma *mono-lub*:
fixes *le-b* (**infix** ⊑ *60*)
assumes *chain*: *Complete-Partial-Order.chain* (*fun-ord op* ≤) *Y*
and *mono*: ⋀*f*. *f* ∈ *Y* ⟹ *monotone le-b op* ≤ *f*
shows *monotone op* ⊑ *op* ≤ (*fun-lub Sup Y*)
⟨*proof*⟩

context
fixes *le-b* (**infix** ⊑ *60*) **and** *Y f*
assumes *chain*: *Complete-Partial-Order.chain le-b Y*
and *mono1*: ⋀*y*. *y* ∈ *Y* ⟹ *monotone le-b op* ≤ (λ*x*. *f x y*)
and *mono2*: ⋀*x a b*. ⟦ *x* ∈ *Y*; *a* ⊑ *b*; *a* ∈ *Y*; *b* ∈ *Y* ⟧ ⟹ *f x a* ≤ *f x b*
begin

lemma *Sup-mono*:
assumes *le*: *x* ⊑ *y* **and** *x*: *x* ∈ *Y* **and** *y*: *y* ∈ *Y*
shows ⨆(*f x* ‘ *Y*) ≤ ⨆(*f y* ‘ *Y*) (**is** - ≤ *?rhs*)
⟨*proof*⟩

lemma *diag-Sup*: ⨆((λ*x*. ⨆(*f x* ‘ *Y*)) ‘ *Y*) = ⨆((λ*x*. *f x x*) ‘ *Y*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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*⟩

lemma *swap-Sup*:
fixes *le-b* (**infix** $\sqsubseteq$ *60*)
assumes *Y*: *Complete-Partial-Order.chain op* $\sqsubseteq$ *Y*
and *Z*: *Complete-Partial-Order.chain* (*fun-ord op* $\leq$) *Z*
and *mono*: $\bigwedge f.\ f \in Z \Longrightarrow$ *monotone op* $\sqsubseteq$ *op* $\leq f$
shows $\bigsqcup((\lambda x.\ \bigsqcup(x\ `\ Y))\ `\ Z) = \bigsqcup((\lambda x.\ \bigsqcup((\lambda f.\ f\ x)\ `\ Z))\ `\ Y)$
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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*
⟨*proof*⟩

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* $\leq f$
⟨*proof*⟩

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*⟩

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* $(\lambda x.\ f\ x\ (t\ x))$
⟨*proof*⟩

22.1 Continuity

definition *cont* :: $('a\ set \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('b\ set \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow bool$
where
cont luba orda lubb ordb f $\longleftrightarrow$

(∀ *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*⟩

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* (λ*x*. *x*)
⟨*proof*⟩

lemma *monotone-applyI*:
monotone orda ordb F ⟹ *monotone* (*fun-ord orda*) *ordb* (λ*f*. *F* (*f x*))
⟨*proof*⟩

lemma *monotone-if-fun* [*partial-function-mono*]:
⟦ *monotone* (*fun-ord orda*) (*fun-ord ordb*) *F*; *monotone* (*fun-ord orda*) (*fun-ord ordb*) *G* ⟧
⟹ *monotone* (*fun-ord orda*) (*fun-ord ordb*) (λ*f n*. *if c n then F f n else G f n*)
⟨*proof*⟩

lemma *monotone-fun-apply-fun* [*partial-function-mono*]:
monotone (*fun-ord* (*fun-ord ord*)) (*fun-ord ord*) (λ*f n*. *f t* (*g n*))
⟨*proof*⟩

lemma *monotone-fun-ord-apply*:
monotone orda (*fun-ord ordb*) *f* ⟷ (∀ *x*. *monotone orda ordb* (λ*y*. *f y x*))
⟨*proof*⟩

context *preorder* **begin**

lemma *transp-le* [*simp*, *cont-intro*]: *transp op* ≤
⟨*proof*⟩

lemma *monotone-const* [*simp*, *cont-intro*]: *monotone ord op* $\leq$ (λ-. *c*)
⟨*proof*⟩

end

lemma *transp-le* [*cont-intro*, *simp*]:
class.preorder ord (*mk-less ord*) $\Longrightarrow$ *transp ord*
⟨*proof*⟩

context *partial-function-definitions* **begin**

declare *const-mono* [*cont-intro*, *simp*]

lemma *transp-le* [*cont-intro*, *simp*]: *transp leq*
⟨*proof*⟩

lemma *preorder* [*cont-intro*, *simp*]: *class.preorder leq* (*mk-less leq*)
⟨*proof*⟩

declare *ccpo*[*cont-intro*, *simp*]

end

lemma *contI* [*intro?*]:
($\bigwedge$*Y*. ⟦ *Complete-Partial-Order.chain orda Y*; *Y* $\neq$ {} ⟧ $\Longrightarrow$ *f* (*luba Y*) = *lubb* (*f* ‘ *Y*))
$\Longrightarrow$ *cont luba orda lubb ordb f*
⟨*proof*⟩

lemma *contD*:
⟦ *cont luba orda lubb ordb f*; *Complete-Partial-Order.chain orda Y*; *Y* $\neq$ {} ⟧
$\Longrightarrow$ *f* (*luba Y*) = *lubb* (*f* ‘ *Y*)
⟨*proof*⟩

lemma *cont-id* [*simp*, *cont-intro*]: $\bigwedge$*Sup*. *cont Sup ord Sup ord id*
⟨*proof*⟩

lemma *cont-id'* [*simp*, *cont-intro*]: $\bigwedge$*Sup*. *cont Sup ord Sup ord* (λ*x*. *x*)
⟨*proof*⟩

lemma *cont-applyI* [*cont-intro*]:
assumes *cont*: *cont luba orda lubb ordb g*
shows *cont* (*fun-lub luba*) (*fun-ord orda*) *lubb ordb* (λ*f*. *g* (*f x*))
⟨*proof*⟩

lemma *call-cont*: *cont* (*fun-lub lub*) (*fun-ord ord*) *lub ord* (λ*f*. *f t*)
⟨*proof*⟩

lemma *cont-if* [*cont-intro*]:
⟦ *cont luba orda lubb ordb f*; *cont luba orda lubb ordb g* ⟧
⟹ *cont luba orda lubb ordb* (λ*x*. *if c then f x else g x*)
⟨*proof*⟩

lemma *mcontI* [*intro?*]:
⟦ *monotone orda ordb f*; *cont luba orda lubb ordb f* ⟧ ⟹ *mcont luba orda lubb ordb f*
⟨*proof*⟩

lemma *mcont-mono*: *mcont luba orda lubb ordb f* ⟹ *monotone orda ordb f*
⟨*proof*⟩

lemma *mcont-cont* [*simp*]: *mcont luba orda lubb ordb f* ⟹ *cont luba orda lubb ordb f*
⟨*proof*⟩

lemma *mcont-monoD*:
⟦ *mcont luba orda lubb ordb f*; *orda x y* ⟧ ⟹ *ordb* (*f x*) (*f y*)
⟨*proof*⟩

lemma *mcont-contD*:
⟦ *mcont luba orda lubb ordb f*; *Complete-Partial-Order.chain orda Y*; *Y* ≠ {} ⟧
⟹ *f* (*luba Y*) = *lubb* (*f* ‘ *Y*)
⟨*proof*⟩

lemma *mcont-call* [*cont-intro*, *simp*]:
mcont (*fun-lub lub*) (*fun-ord ord*) *lub ord* (λ*f*. *f t*)
⟨*proof*⟩

lemma *mcont-id′* [*cont-intro*, *simp*]: *mcont lub ord lub ord* (λ*x*. *x*)
⟨*proof*⟩

lemma *mcont-applyI*:
mcont luba orda lubb ordb (λ*x*. *F x*) ⟹ *mcont* (*fun-lub luba*) (*fun-ord orda*) *lubb ordb* (λ*f*. *F* (*f x*))
⟨*proof*⟩

lemma *mcont-if* [*cont-intro*, *simp*]:
⟦ *mcont luba orda lubb ordb* (λ*x*. *f x*); *mcont luba orda lubb ordb* (λ*x*. *g x*) ⟧
⟹ *mcont luba orda lubb ordb* (λ*x*. *if c then f x else g x*)
⟨*proof*⟩

lemma *cont-fun-lub-apply*:
cont luba orda (*fun-lub lubb*) (*fun-ord ordb*) *f* ⟷ (∀ *x*. *cont luba orda lubb ordb* (λ*y*. *f y x*))
⟨*proof*⟩

lemma *mcont-fun-lub-apply*:

mcont luba orda (*fun-lub lubb*) (*fun-ord ordb*) $f \longleftrightarrow (\forall x.\ mcont\ luba\ orda\ lubb\ ordb\ (\lambda y.\ f\ y\ x))$
⟨*proof*⟩

context *ccpo* **begin**

lemma *cont-const* [*simp*, *cont-intro*]: *cont luba orda Sup op* $\leq (\lambda x.\ c)$
⟨*proof*⟩

lemma *mcont-const* [*cont-intro*, *simp*]:
mcont luba orda Sup op $\leq (\lambda x.\ c)$
⟨*proof*⟩

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*⟩

lemma *mcont2mcont'*:
$[\![\bigwedge x.$ *mcont lub' ord' Sup op* $\leq (\lambda y.\ f\ x\ y);$
$\quad \bigwedge y.$ *mcont lub ord Sup op* $\leq (\lambda x.\ f\ x\ y);$
$\quad$ *mcont lub ord lub' ord'* $(\lambda y.\ t\ y)\ ]\!]$
$\Longrightarrow$ *mcont lub ord Sup op* $\leq (\lambda x.\ f\ x\ (t\ x))$
⟨*proof*⟩

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* $\leq (\lambda x.\ f\ (t\ x))$
⟨*proof*⟩

context
fixes *ord* :: *'b* ⇒ *'b* ⇒ *bool* (**infix** $\sqsubseteq$ *60*)
and *lub* :: *'b set* ⇒ *'b* ($\bigsqcup$- [*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*⟩

lemma *mcont-fun-lub-Sup*:
$[\![$ *Complete-Partial-Order.chain* (*fun-ord op* $\leq$) *M*;
$\quad \forall f{\in}M.$ *mcont lub ord Sup op* $\leq f\ ]\!]$

$\Longrightarrow$ *mcont lub op* $\sqsubseteq$ *Sup op* $\leq$ (*fun-lub Sup M*)
⟨*proof*⟩

lemma *iterates-mcont*:
assumes *f*: *f* $\in$ *ccpo.iterates* (*fun-lub Sup*) (*fun-ord op* $\leq$) *F*
and *mono*: $\bigwedge$*f*. *mcont lub op* $\sqsubseteq$ *Sup op* $\leq$ *f* $\Longrightarrow$ *mcont lub op* $\sqsubseteq$ *Sup op* $\leq$ (*F f*)
shows *mcont lub op* $\sqsubseteq$ *Sup op* $\leq$ *f*
⟨*proof*⟩

lemma *fixp-preserves-mcont*:
assumes *mono*: $\bigwedge$*x*. *monotone* (*fun-ord op* $\leq$) *op* $\leq$ (λ*f*. *F f x*)
and *mcont*: $\bigwedge$*f*. *mcont lub op* $\sqsubseteq$ *Sup op* $\leq$ *f* $\Longrightarrow$ *mcont lub op* $\sqsubseteq$ *Sup op* $\leq$ (*F f*)
shows *mcont lub op* $\sqsubseteq$ *Sup op* $\leq$ (*ccpo.fixp* (*fun-lub Sup*) (*fun-ord op* $\leq$) *F*)
(**is** *mcont - - - - ?fixp*)
⟨*proof*⟩

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* $\leq$ (λ*f*. *U* (*F* (*C f*)) *x*)
and *eq*: *f* $\equiv$ *C* (*ccpo.fixp* (*fun-lub Sup*) (*fun-ord op* $\leq$) (λ*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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

end

lemmas *fixp-preserves-mono1* = *fixp-preserves-mono-uc*[*of* λ*x*. *x* - λ*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* λ*f*. *case-prod* (*case-prod f*) - λ*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* $\lambda x.$ *x* - $\lambda x.$ *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* $\lambda f.$ *case-prod* (*case-prod f*) - $\lambda f.$ *curry* (*curry f*), *unfolded case-prod-curry curry-case-prod*, *OF* - - *refl*]
lemmas *fixp-preserves-mcont4* =
fixp-preserves-mcont-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*]

end

lemma (**in** *preorder*) *monotone-if-bot*:
fixes *bot*
assumes *mono*: $\bigwedge x\ y.\ [\![\ x \le y;\ \neg\ (x \le bound)\ ]\!] \Longrightarrow ord\ (f\ x)\ (f\ y)$
and *bot*: $\bigwedge x.\ \neg\ x \le bound \Longrightarrow ord\ bot\ (f\ x)\ ord\ bot\ bot$
shows *monotone op* $\le$ *ord* ($\lambda x.$ *if* $x \le$ *bound then bot else f x*)
⟨*proof*⟩

lemma (**in** *ccpo*) *mcont-if-bot*:
fixes *bot* **and** *lub* ($\bigvee$- [*900*] *900*) **and** *ord* (**infix** $\sqsubseteq$ *60*)
assumes *ccpo*: *class.ccpo lub op* $\sqsubseteq$ *lt*
and *mono*: $\bigwedge x\ y.\ [\![\ x \le y;\ \neg\ x \le bound\ ]\!] \Longrightarrow f\ x \sqsubseteq f\ y$
and *cont*: $\bigwedge Y.\ [\![$ *Complete-Partial-Order.chain op* $\le\ Y;\ Y \neq \{\};\ \bigwedge x.\ x \in Y \Longrightarrow \neg\ x \le bound\ ]\!] \Longrightarrow f\ (\bigsqcup Y) = \bigvee(f\ `\ Y)$
and *bot*: $\bigwedge x.\ \neg\ x \le bound \Longrightarrow bot \sqsubseteq f\ x$
shows *mcont Sup op* $\le$ *lub op* $\sqsubseteq$ ($\lambda x.$ *if* $x \le$ *bound then bot else f x*) (**is** *mcont* - - - - *?g*)
⟨*proof*⟩

context *partial-function-definitions* **begin**

lemma *mcont-const* [*cont-intro*, *simp*]:
mcont luba orda lub leq ($\lambda x.$ *c*)
⟨*proof*⟩

lemmas [*cont-intro*, *simp*] =
ccpo.cont-const[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]

lemma *mono2mono*:
assumes *monotone ordb leq* ($\lambda y.$ *f y*) *monotone orda ordb* ($\lambda x.$ *t x*)
shows *monotone orda leq* ($\lambda x.$ *f* (*t x*))
⟨*proof*⟩

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*: $\bigwedge x.\ g\ x = (\mathit{if}\ \mathit{leq}\ x\ \mathit{bound}\ \mathit{then}\ \mathit{bot}\ \mathit{else}\ f\ x)$
and *mono*: $\bigwedge x\ y.\ [\![\ \mathit{leq}\ x\ y;\ \neg\ \mathit{leq}\ x\ \mathit{bound}\ ]\!] \Longrightarrow \mathit{ord}\ (f\ x)\ (f\ y)$
and *bot*: $\bigwedge x.\ \neg\ \mathit{leq}\ x\ \mathit{bound} \Longrightarrow \mathit{ord}\ \mathit{bot}\ (f\ x)\ \mathit{ord}\ \mathit{bot}\ \mathit{bot}$
shows *monotone leq ord g*
⟨*proof*⟩

lemma *mcont-if-bot*:
fixes *bot*
assumes *ccpo*: *class.ccpo lub' ord* (*mk-less ord*)
and *bot*: $\bigwedge x.\ \neg\ \mathit{leq}\ x\ \mathit{bound} \Longrightarrow \mathit{ord}\ \mathit{bot}\ (f\ x)$
and *g*: $\bigwedge x.\ g\ x = (\mathit{if}\ \mathit{leq}\ x\ \mathit{bound}\ \mathit{then}\ \mathit{bot}\ \mathit{else}\ f\ x)$
and *mono*: $\bigwedge x\ y.\ [\![\ \mathit{leq}\ x\ y;\ \neg\ \mathit{leq}\ x\ \mathit{bound}\ ]\!] \Longrightarrow \mathit{ord}\ (f\ x)\ (f\ y)$
and *cont*: $\bigwedge Y.\ [\![\ \mathit{Complete\text{-}Partial\text{-}Order.chain}\ \mathit{leq}\ Y;\ Y \neq \{\};\ \bigwedge x.\ x \in Y \Longrightarrow \neg\ \mathit{leq}\ x\ \mathit{bound}\ ]\!] \Longrightarrow f\ (\mathit{lub}\ Y) = \mathit{lub'}\ (f\ `\ Y)$
shows *mcont lub leq lub' ord g*
⟨*proof*⟩

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* $(\lambda x.\ P\ (f\ x))$
⟨*proof*⟩

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* ⟧
⟹ *ccpo.admissible lub ord* (λ*x*. *P x* ∨ *Q x*)
⟨*proof*⟩

lemma *admissible-imp'* [*cont-intro*]:
⟦ *class.ccpo lub ord* (*mk-less ord*);
ccpo.admissible lub ord (λ*x*. ¬ *P x*);
ccpo.admissible lub ord (λ*x*. *Q x*) ⟧
⟹ *ccpo.admissible lub ord* (λ*x*. *P x* ⟶ *Q x*)
⟨*proof*⟩

lemma *admissible-imp* [*cont-intro*]:
(*Q* ⟹ *ccpo.admissible lub ord* (λ*x*. *P x*))
⟹ *ccpo.admissible lub ord* (λ*x*. *Q* ⟶ *P x*)
⟨*proof*⟩

lemma *admissible-not-mem'* [*THEN admissible-subst*, *cont-intro*, *simp*]:
shows *admissible-not-mem*: *ccpo.admissible Union op* ⊆ (λ*A*. *x* ∉ *A*)
⟨*proof*⟩

lemma *admissible-eqI*:
assumes *f*: *cont luba orda lub ord* (λ*x*. *f x*)
and *g*: *cont luba orda lub ord* (λ*x*. *g x*)
shows *ccpo.admissible luba orda* (λ*x*. *f x* = *g x*)
⟨*proof*⟩

corollary *admissible-eq-mcontI* [*cont-intro*]:
⟦ *mcont luba orda lub ord* (λ*x*. *f x*);
mcont luba orda lub ord (λ*x*. *g x*) ⟧
⟹ *ccpo.admissible luba orda* (λ*x*. *f x* = *g x*)
⟨*proof*⟩

lemma *admissible-iff* [*cont-intro*, *simp*]:
⟦ *ccpo.admissible lub ord* (λ*x*. *P x* ⟶ *Q x*); *ccpo.admissible lub ord* (λ*x*. *Q x* ⟶ *P x*) ⟧
⟹ *ccpo.admissible lub ord* (λ*x*. *P x* ⟷ *Q x*)
⟨*proof*⟩

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*⟩

end

lemma *admissible-leI*:
fixes *ord* (**infix** $\sqsubseteq$ *60*) **and** *lub* ($\bigsqcup$- [*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* $(\lambda x.\ f\ x \sqsubseteq g\ x)$
⟨*proof*⟩

declare *ccpo-class.admissible-leI*[*cont-intro*]

context *ccpo* **begin**

lemma *admissible-not-below*: *ccpo.admissible Sup op* $\leq$ $(\lambda x.\ \neg\ op \leq x\ y)$
⟨*proof*⟩

end

lemma (**in** *preorder*) *preorder* [*cont-intro*, *simp*]: *class.preorder op* $\leq$ (*mk-less op* $\leq$)
⟨*proof*⟩

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*:
$[\![$ *ccpo.admissible lub ord* $(\lambda y.\ \neg\ ord\ x\ y)$;
ccpo.admissible lub ord $(\lambda y.\ x \neq y)$ $]\!]$
$\Longrightarrow$ *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* $\leq x$
⟨*proof*⟩

lemma *compact-bot*:
assumes x = *Sup* {}
shows *compact Sup op* $\leq x$
⟨*proof*⟩

end

lemma *admissible-compact-neq'* [*THEN admissible-subst*, *cont-intro*, *simp*]:
shows *admissible-compact-neq*: *compact lub ord k* $\Longrightarrow$ *ccpo.admissible lub ord* $(\lambda x.\ k \neq x)$
⟨*proof*⟩

lemma *admissible-neq-compact'* [*THEN admissible-subst*, *cont-intro*, *simp*]:
shows *admissible-neq-compact*: *compact lub ord k* $\Longrightarrow$ *ccpo.admissible lub ord* $(\lambda x.\ x \neq k)$
⟨*proof*⟩

context *partial-function-definitions* **begin**

lemmas [*cont-intro*, *simp*] = *ccpo.compact-bot*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axiom*

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*⟩

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*: $P\ (\lambda\text{-}.\ lub\ \{\})$
and *step*: $\bigwedge f'.\ [\![\ P\ (U\ f');\ le\text{-}fun\ (U\ f')\ (U\ f)\ ]\!] \Longrightarrow P\ (U\ (F\ f'))$
shows $P\ (U\ f)$
⟨*proof*⟩

end

22.3 *op* = as order

definition *lub-singleton* :: $('a\ set \Rightarrow 'a) \Rightarrow bool$
where $lub\text{-}singleton\ lub \longleftrightarrow (\forall a.\ lub\ \{a\} = a)$

definition *the-Sup* :: $'a\ set \Rightarrow 'a$
where $the\text{-}Sup\ A = (THE\ a.\ a \in A)$

lemma *lub-singleton-the-Sup* [*cont-intro*, *simp*]: *lub-singleton the-Sup*
⟨*proof*⟩

lemma (**in** *ccpo*) *lub-singleton*: *lub-singleton Sup*
⟨*proof*⟩

lemma (**in** *partial-function-definitions*) *lub-singleton* [*cont-intro*, *simp*]: *lub-singleton lub*
⟨*proof*⟩

lemma *preorder-eq* [*cont-intro*, *simp*]:
class.preorder op = (*mk-less op* =)
⟨*proof*⟩

lemma *monotone-eqI* [*cont-intro*]:
assumes *class.preorder ord* (*mk-less ord*)
shows *monotone op* = *ord f*
⟨*proof*⟩

lemma *cont-eqI* [*cont-intro*]:
fixes $f :: 'a \Rightarrow 'b$
assumes *lub-singleton lub*
shows *cont the-Sup op* = *lub ord f*
⟨*proof*⟩

lemma *mcont-eqI* [*cont-intro*, *simp*]:
$[\![$ *class.preorder ord* (*mk-less ord*); *lub-singleton lub* $]\!]$
$\Longrightarrow$ *mcont the-Sup op* = *lub ord f*
⟨*proof*⟩

22.4 ccpo for products

definition *prod-lub* :: $('a\ set \Rightarrow 'a) \Rightarrow ('b\ set \Rightarrow 'b) \Rightarrow ('a \times 'b)\ set \Rightarrow 'a \times 'b$

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*)
⟨*proof*⟩

lemma *prod-lub-empty* [*simp*]: *prod-lub luba lubb* {} = (*luba* {}, *lubb* {})
⟨*proof*⟩

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*⟩

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*⟩

lemma *monotone-rel-prodI*:
assumes *mono2*: ⋀*a*. *monotone ordb ordc* (λ*b*. *f* (*a*, *b*))
and *mono1*: ⋀*b*. *monotone orda ordc* (λ*a*. *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*⟩

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* (λ*a*. *f* (*a*, *b*))
⟨*proof*⟩

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* (λ*b*. *f* (*a*, *b*))
⟨*proof*⟩

lemma *monotone-case-prodI*:
⟦ ⋀*a*. *monotone ordb ordc* (*f a*); ⋀*b*. *monotone orda ordc* (λ*a*. *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*)
⟨*proof*⟩

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* (λ*a*. *f a b*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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* (λ*b*. *f* (*a*, *b*))) ∧
(∀ *b*. *monotone orda ordc* (λ*a*. *f* (*a*, *b*)))
⟨*proof*⟩

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* (λ*a*. *f a b*))
⟨*proof*⟩

end

lemma *monotone-case-prod-apply-iff*:
monotone orda ordb (λ*x*. (*case-prod f x*) *y*) ⟷ *monotone orda ordb* (*case-prod* (λ*a b*. *f a b y*))
⟨*proof*⟩

lemma *monotone-case-prod-applyD*:
monotone orda ordb (λ*x*. (*case-prod f x*) *y*)
⟹ *monotone orda ordb* (*case-prod* (λ*a b*. *f a b y*))
⟨*proof*⟩

lemma *monotone-case-prod-applyI*:
monotone orda ordb (*case-prod* (λ*a b*. *f a b y*))
⟹ *monotone orda ordb* (λ*x*. (*case-prod f x*) *y*)
⟨*proof*⟩

lemma *cont-case-prod-apply-iff*:
cont luba orda lubb ordb (λ*x*. (*case-prod f x*) *y*) ⟷ *cont luba orda lubb ordb* (*case-prod* (λ*a b*. *f a b y*))
⟨*proof*⟩

lemma *cont-case-prod-applyI*:
cont luba orda lubb ordb (*case-prod* (λ*a b*. *f a b y*))
⟹ *cont luba orda lubb ordb* (λ*x*. (*case-prod f x*) *y*)
⟨*proof*⟩

lemma *cont-case-prod-applyD*:
cont luba orda lubb ordb (λ*x*. (*case-prod f x*) *y*)
⟹ *cont luba orda lubb ordb* (*case-prod* (λ*a b*. *f a b y*))
⟨*proof*⟩

lemma *mcont-case-prod-apply-iff* [*simp*]:
mcont luba orda lubb ordb (λ*x*. (*case-prod f x*) *y*) ⟷
mcont luba orda lubb ordb (*case-prod* (λ*a b*. *f a b y*))
⟨*proof*⟩

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* (λ*y*. *f* (*x*, *y*))
⟨*proof*⟩

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* (λ*x*. *f* (*x*, *y*))
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

lemma *cont-case-prodI*:
assumes *monotone* (*rel-prod orda ordb*) *op* ≤ (*case-prod f*)
and ⋀*x*. *cont lubb ordb Sup op* ≤ (λ*y*. *f x y*)
and ⋀*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* ≤ (*case-prod f*)
⟨*proof*⟩

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* ≤ (λ*y*. *f x y*)) ∧ (∀ *y*. *cont luba orda Sup op* ≤ (λ*x*. *f x y*))
⟨*proof*⟩

end

context *partial-function-definitions* **begin**

lemma *mono2mono2*:
assumes *f*: *monotone* (*rel-prod ordb ordc*) *leq* (λ(*x*, *y*). *f x y*)
and *t*: *monotone orda ordb* (λ*x*. *t x*)
and *t'*: *monotone orda ordc* (λ*x*. *t' x*)
shows *monotone orda leq* (λ*x*. *f* (*t x*) (*t' x*))
⟨*proof*⟩

lemma *cont-case-prodI* [*cont-intro*]:
⟦ *monotone* (*rel-prod orda ordb*) *leq* (*case-prod f*);
⋀*x*. *cont lubb ordb lub leq* (λ*y*. *f x y*);
⋀*y*. *cont luba orda lub leq* (λ*x*. *f x y*);
class.preorder orda (*mk-less orda*);
class.preorder ordb (*mk-less ordb*) ⟧
⟹ *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *lub leq* (*case-prod f*)
⟨*proof*⟩

lemma *cont-case-prod-iff*:

⟦ *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* ⟧
⟹ *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *lub leq* (*case-prod f*) ⟷
(∀ *x*. *cont lubb ordb lub leq* (λ*y*. *f x y*)) ∧ (∀ *y*. *cont luba orda lub leq* (λ*x*. *f x y*))
⟨*proof*⟩

lemma *mcont-case-prod-iff* [*simp*]:
⟦ *class.preorder orda* (*mk-less orda*); *lub-singleton luba*;
class.preorder ordb (*mk-less ordb*); *lub-singleton lubb* ⟧
⟹ *mcont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *lub leq* (*case-prod f*) ⟷
(∀ *x*. *mcont lubb ordb lub leq* (λ*y*. *f x y*)) ∧ (∀ *y*. *mcont luba orda lub leq* (λ*x*. *f x y*))
⟨*proof*⟩

end

lemma *mono2mono-case-prod* [*cont-intro*]:
assumes ⋀*x y*. *monotone orda ordb* (λ*f*. *pair f x y*)
shows *monotone orda ordb* (λ*f*. *case-prod* (*pair f*) *x*)
⟨*proof*⟩

22.5 Complete lattices as ccpo

context *complete-lattice* **begin**

lemma *complete-lattice-ccpo*: *class.ccpo Sup op* ≤ *op* <
⟨*proof*⟩

lemma *complete-lattice-ccpo'*: *class.ccpo Sup op* ≤ (*mk-less op* ≤)
⟨*proof*⟩

lemma *complete-lattice-partial-function-definitions*:
partial-function-definitions op ≤ *Sup*
⟨*proof*⟩

lemma *complete-lattice-partial-function-definitions-dual*:
partial-function-definitions op ≥ *Inf*
⟨*proof*⟩

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* ≤ (λ*x*. *f x*)
and *g*: *monotone ord op* ≤ (λ*x*. *g x*)
shows *monotone ord op* ≤ (λ*x*. *f x* ⊓ *g x*)
⟨*proof*⟩

lemma *mcont-const* [*simp*]: *mcont lub ord Sup op* $\leq$ (λ-. *c*)
⟨*proof*⟩

lemma *mono2mono-sup*:
assumes *f*: *monotone ord op* $\leq$ ($\lambda x.$ *f x*)
and *g*: *monotone ord op* $\leq$ ($\lambda x.$ *g x*)
shows *monotone ord op* $\leq$ ($\lambda x.$ *f x* $\sqcup$ *g x*)
⟨*proof*⟩

lemma *Sup-image-sup*:
assumes $Y \neq \{\}$
shows $\bigsqcup(op \sqcup x \mathbin{`} Y) = x \sqcup \bigsqcup Y$
⟨*proof*⟩

lemma *mcont-sup1*: *mcont Sup op* $\leq$ *Sup op* $\leq$ ($\lambda y.$ *x* $\sqcup$ *y*)
⟨*proof*⟩

lemma *mcont-sup2*: *mcont Sup op* $\leq$ *Sup op* $\leq$ ($\lambda x.$ *x* $\sqcup$ *y*)
⟨*proof*⟩

lemma *mcont2mcont-sup* [*cont-intro*, *simp*]:
⟦ *mcont lub ord Sup op* $\leq$ ($\lambda x.$ *f x*);
mcont lub ord Sup op $\leq$ ($\lambda x.$ *g x*) ⟧
$\Longrightarrow$ *mcont lub ord Sup op* $\leq$ ($\lambda x.$ *f x* $\sqcup$ *g x*)
⟨*proof*⟩

end

lemmas [*cont-intro*] = *admissible-leI*[*OF complete-lattice-ccpo'*]

context *complete-distrib-lattice* **begin**

lemma *mcont-inf1*: *mcont Sup op* $\leq$ *Sup op* $\leq$ ($\lambda y.$ *x* $\sqcap$ *y*)
⟨*proof*⟩

lemma *mcont-inf2*: *mcont Sup op* $\leq$ *Sup op* $\leq$ ($\lambda x.$ *x* $\sqcap$ *y*)
⟨*proof*⟩

lemma *mcont2mcont-inf* [*cont-intro*, *simp*]:
⟦ *mcont lub ord Sup op* $\leq$ ($\lambda x.$ *f x*);
mcont lub ord Sup op $\leq$ ($\lambda x.$ *g x*) ⟧
$\Longrightarrow$ *mcont lub ord Sup op* $\leq$ ($\lambda x.$ *f x* $\sqcap$ *g x*)
⟨*proof*⟩

end

interpretation *lfp*: *partial-function-definitions op* $\leq$ :: - :: *complete-lattice* $\Rightarrow$ - *Sup*

⟨*proof*⟩

⟨*ML*⟩

interpretation *gfp*: *partial-function-definitions op* ≥ :: - :: *complete-lattice* ⇒ - *Inf*
⟨*proof*⟩

⟨*ML*⟩

lemma *insert-mono* [*partial-function-mono*]:
monotone (*fun-ord op* ⊆) *op* ⊆ *A* ⟹ *monotone* (*fun-ord op* ⊆) *op* ⊆ (λ*y*. *insert x* (*A y*))
⟨*proof*⟩

lemma *mono2mono-insert* [*THEN lfp.mono2mono*, *cont-intro*, *simp*]:
shows *monotone-insert*: *monotone op* ⊆ *op* ⊆ (*insert x*)
⟨*proof*⟩

lemma *mcont2mcont-insert*[*THEN lfp.mcont2mcont*, *cont-intro*, *simp*]:
shows *mcont-insert*: *mcont Union op* ⊆ *Union op* ⊆ (*insert x*)
⟨*proof*⟩

lemma *mono2mono-image* [*THEN lfp.mono2mono*, *cont-intro*, *simp*]:
shows *monotone-image*: *monotone op* ⊆ *op* ⊆ (*op* ‘ *f*)
⟨*proof*⟩

lemma *cont-image*: *cont Union op* ⊆ *Union op* ⊆ (*op* ‘ *f*)
⟨*proof*⟩

lemma *mcont2mcont-image* [*THEN lfp.mcont2mcont*, *cont-intro*, *simp*]:
shows *mcont-image*: *mcont Union op* ⊆ *Union op* ⊆ (*op* ‘ *f*)
⟨*proof*⟩

context *complete-lattice* **begin**

lemma *monotone-Sup* [*cont-intro*, *simp*]:
monotone ord op ⊆ *f* ⟹ *monotone ord op* ≤ (λ*x*. ⨆ *f x*)
⟨*proof*⟩

lemma *cont-Sup*:
assumes *cont lub ord Union op* ⊆ *f*
shows *cont lub ord Sup op* ≤ (λ*x*. ⨆ *f x*)
⟨*proof*⟩

lemma *mcont-Sup*: *mcont lub ord Union op* ⊆ *f* ⟹ *mcont lub ord Sup op* ≤ (λ*x*. ⨆ *f x*)
⟨*proof*⟩

lemma *monotone-SUP*:
⟦ *monotone ord op* ⊆ *f*; ⋀*y*. *monotone ord op* ≤ (λ*x*. *g x y*) ⟧ ⟹ *monotone ord op* ≤ (λ*x*. ⨆ *y*∈*f x*. *g x y*)
⟨*proof*⟩

lemma *monotone-SUP2*:
(⋀*y*. *y* ∈ *A* ⟹ *monotone ord op* ≤ (λ*x*. *g x y*)) ⟹ *monotone ord op* ≤ (λ*x*. ⨆ *y*∈*A*. *g x y*)
⟨*proof*⟩

lemma *cont-SUP*:
assumes *f*: *mcont lub ord Union op* ⊆ *f*
and *g*: ⋀*y*. *mcont lub ord Sup op* ≤ (λ*x*. *g x y*)
shows *cont lub ord Sup op* ≤ (λ*x*. ⨆ *y*∈*f x*. *g x y*)
⟨*proof*⟩

lemma *mcont-SUP* [*cont-intro*, *simp*]:
⟦ *mcont lub ord Union op* ⊆ *f*; ⋀*y*. *mcont lub ord Sup op* ≤ (λ*x*. *g x y*) ⟧
⟹ *mcont lub ord Sup op* ≤ (λ*x*. ⨆ *y*∈*f x*. *g x y*)
⟨*proof*⟩

end

lemma *admissible-Ball* [*cont-intro*, *simp*]:
⟦ ⋀*x*. *ccpo.admissible lub ord* (λ*A*. *P A x*);
mcont lub ord Union op ⊆ *f*;
class.ccpo lub ord (*mk-less ord*) ⟧
⟹ *ccpo.admissible lub ord* (λ*A*. ∀ *x*∈*f A*. *P A x*)
⟨*proof*⟩

lemma *admissible-Bex'*[*THEN admissible-subst*, *cont-intro*, *simp*]:
shows *admissible-Bex*: *ccpo.admissible Union op* ⊆ (λ*A*. ∃ *x*∈*A*. *P x*)
⟨*proof*⟩

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* ⟨*proof*⟩
interpretation *b*: *ccpo lubb ordb mk-less ordb* ⟨*proof*⟩

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*⟩

interpretation *ab*: *ccpo prod-lub luba lubb rel-prod orda ordb mk-less* (*rel-prod orda ordb*)
⟨*proof*⟩

lemma *monotone-map-prod* [*simp*]:
monotone (*rel-prod orda ordb*) (*rel-prod ordc ordd*) (*map-prod f g*) ⟷
monotone orda ordc f ∧ *monotone ordb ordd g*
⟨*proof*⟩

lemma *parallel-fixp-induct*:
assumes *adm*: *ccpo.admissible* (*prod-lub luba lubb*) (*rel-prod orda ordb*) (λ*x*. *P* (*fst x*) (*snd x*))
and *f*: *monotone orda orda f*
and *g*: *monotone ordb ordb g*
and *bot*: *P* (*luba* {}) (*lubb* {})
and *step*: ⋀*x y*. *P x y* ⟹ *P* (*f x*) (*g y*)
shows *P* (*ccpo.fixp luba orda f*) (*ccpo.fixp lubb ordb g*)
⟨*proof*⟩

end

lemma *parallel-fixp-induct-uc*:
assumes *a*: *partial-function-definitions orda luba*
and *b*: *partial-function-definitions ordb lubb*
and *F*: ⋀*x*. *monotone* (*fun-ord orda*) *orda* (λ*f*. *U1* (*F* (*C1 f*)) *x*)
and *G*: ⋀*y*. *monotone* (*fun-ord ordb*) *ordb* (λ*g*. *U2* (*G* (*C2 g*)) *y*)
and *eq1*: *f* ≡ *C1* (*ccpo.fixp* (*fun-lub luba*) (*fun-ord orda*) (λ*f*. *U1* (*F* (*C1 f*))))
and *eq2*: *g* ≡ *C2* (*ccpo.fixp* (*fun-lub lubb*) (*fun-ord ordb*) (λ*g*. *U2* (*G* (*C2 g*))))
and *inverse*: ⋀*f*. *U1* (*C1 f*) = *f*
and *inverse2*: ⋀*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*)) (λ*x*. *P* (*fst x*) (*snd x*))
and *bot*: *P* (λ-. *luba* {}) (λ-. *lubb* {})
and *step*: ⋀*f g*. *P* (*U1 f*) (*U2 g*) ⟹ *P* (*U1* (*F f*)) (*U2* (*G g*))
shows *P* (*U1 f*) (*U2 g*)
⟨*proof*⟩

lemmas *parallel-fixp-induct-1-1* = *parallel-fixp-induct-uc*[
of - - - - λ*x*. *x* - λ*x*. *x* λ*x*. *x* - λ*x*. *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*
⟨*proof*⟩

lemma *mcont-fst*: *mcont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *luba orda fst*
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *monotone-snd*: *monotone* (*rel-prod orda ordb*) *ordb snd*
⟨*proof*⟩

lemma *mcont-snd*: *mcont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *lubb ordb snd*
⟨*proof*⟩

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*))
⟨*proof*⟩

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*)
⟨*proof*⟩

class *countable-complete-lattice* = *lattice* + *Inf* + *Sup* + *bot* + *top* +
assumes *ccInf-lower*: *countable A* ⟹ *x* ∈ *A* ⟹ *Inf A* ≤ *x*
assumes *ccInf-greatest*: *countable A* ⟹ (⋀*x*. *x* ∈ *A* ⟹ *z* ≤ *x*) ⟹ *z* ≤ *Inf A*
assumes *ccSup-upper*: *countable A* ⟹ *x* ∈ *A* ⟹ *x* ≤ *Sup A*
assumes *ccSup-least*: *countable A* ⟹ (⋀*x*. *x* ∈ *A* ⟹ *x* ≤ *z*) ⟹ *Sup A* ≤ *z*

assumes *ccInf-empty* [*simp*]: *Inf* {} = *top*
assumes *ccSup-empty* [*simp*]: *Sup* {} = *bot*
begin

subclass *bounded-lattice*
⟨*proof*⟩

lemma *ccINF-lower*: *countable A* ⟹ *i* ∈ *A* ⟹ (*INF i* :*A*. *f i*) ≤ *f i*
⟨*proof*⟩

lemma *ccINF-greatest*: *countable A* ⟹ (⋀*i*. *i* ∈ *A* ⟹ *u* ≤ *f i*) ⟹ *u* ≤ (*INF i* :*A*. *f i*)
⟨*proof*⟩

lemma *ccSUP-upper*: *countable A* ⟹ *i* ∈ *A* ⟹ *f i* ≤ (*SUP i* :*A*. *f i*)
⟨*proof*⟩

lemma *ccSUP-least*: *countable A* ⟹ (⋀*i*. *i* ∈ *A* ⟹ *f i* ≤ *u*) ⟹ (*SUP i* :*A*. *f i*) ≤ *u*
⟨*proof*⟩

lemma *ccInf-lower2*: *countable A* ⟹ *u* ∈ *A* ⟹ *u* ≤ *v* ⟹ *Inf A* ≤ *v*
⟨*proof*⟩

lemma *ccINF-lower2*: *countable A* ⟹ *i* ∈ *A* ⟹ *f i* ≤ *u* ⟹ (*INF i* :*A*. *f i*) ≤ *u*
⟨*proof*⟩

lemma *ccSup-upper2*: *countable A* ⟹ *u* ∈ *A* ⟹ *v* ≤ *u* ⟹ *v* ≤ *Sup A*
⟨*proof*⟩

lemma *ccSUP-upper2*: *countable A* ⟹ *i* ∈ *A* ⟹ *u* ≤ *f i* ⟹ *u* ≤ (*SUP i* :*A*. *f i*)
⟨*proof*⟩

lemma *le-ccInf-iff*: *countable A* ⟹ *b* ≤ *Inf A* ⟷ (∀ *a*∈*A*. *b* ≤ *a*)
⟨*proof*⟩

lemma *le-ccINF-iff*: *countable A* ⟹ *u* ≤ (*INF i* :*A*. *f i*) ⟷ (∀ *i*∈*A*. *u* ≤ *f i*)
⟨*proof*⟩

lemma *ccSup-le-iff*: *countable A* ⟹ *Sup A* ≤ *b* ⟷ (∀ *a*∈*A*. *a* ≤ *b*)
⟨*proof*⟩

lemma *ccSUP-le-iff*: *countable A* ⟹ (*SUP i* :*A*. *f i*) ≤ *u* ⟷ (∀ *i*∈*A*. *f i* ≤ *u*)
⟨*proof*⟩

lemma *ccInf-insert* [*simp*]: *countable A* ⟹ *Inf* (*insert a A*) = *inf a* (*Inf A*)
⟨*proof*⟩

lemma *ccINF-insert* [*simp*]: *countable A* $\Longrightarrow$ (*INF x*:*insert a A*. *f x*) = *inf* (*f a*) (*INFIMUM A f*)
⟨*proof*⟩

lemma *ccSup-insert* [*simp*]: *countable A* $\Longrightarrow$ *Sup* (*insert a A*) = *sup a* (*Sup A*)
⟨*proof*⟩

lemma *ccSUP-insert* [*simp*]: *countable A* $\Longrightarrow$ (*SUP x*:*insert a A*. *f x*) = *sup* (*f a*) (*SUPREMUM A f*)
⟨*proof*⟩

lemma *ccINF-empty* [*simp*]: (*INF x*:{}. *f x*) = *top*
⟨*proof*⟩

lemma *ccSUP-empty* [*simp*]: (*SUP x*:{}. *f x*) = *bot*
⟨*proof*⟩

lemma *ccInf-superset-mono*: *countable A* $\Longrightarrow$ *B* $\subseteq$ *A* $\Longrightarrow$ *Inf A* $\leq$ *Inf B*
⟨*proof*⟩

lemma *ccSup-subset-mono*: *countable B* $\Longrightarrow$ *A* $\subseteq$ *B* $\Longrightarrow$ *Sup A* $\leq$ *Sup B*
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *ccSUP-subset-mono*:
countable $B \Longrightarrow A \subseteq B \Longrightarrow (\bigwedge x.\ x \in A \Longrightarrow f\ x \leq g\ x) \Longrightarrow (SUP\ x{:}A.\ f\ x) \leq (SUP\ x{:}B.\ g\ x)$
⟨*proof*⟩

lemma *less-eq-ccInf-inter*: *countable* $A \Longrightarrow$ *countable* $B \Longrightarrow sup\ (Inf\ A)\ (Inf\ B) \leq Inf\ (A \cap B)$
⟨*proof*⟩

lemma *ccSup-inter-less-eq*: *countable* $A \Longrightarrow$ *countable* $B \Longrightarrow Sup\ (A \cap B) \leq inf\ (Sup\ A)\ (Sup\ B)$
⟨*proof*⟩

lemma *ccInf-union-distrib*: *countable* $A \Longrightarrow$ *countable* $B \Longrightarrow Inf\ (A \cup B) = inf\ (Inf\ A)\ (Inf\ B)$
⟨*proof*⟩

lemma *ccINF-union*:
countable $A \Longrightarrow$ *countable* $B \Longrightarrow (INF\ i{:}A \cup B.\ M\ i) = inf\ (INF\ i{:}A.\ M\ i)\ (INF\ i{:}B.\ M\ i)$
⟨*proof*⟩

lemma *ccSup-union-distrib*: *countable* $A \Longrightarrow$ *countable* $B \Longrightarrow Sup\ (A \cup B) = sup\ (Sup\ A)\ (Sup\ B)$
⟨*proof*⟩

lemma *ccSUP-union*:
countable $A \Longrightarrow$ *countable* $B \Longrightarrow (SUP\ i{:}A \cup B.\ M\ i) = sup\ (SUP\ i{:}A.\ M\ i)\ (SUP\ i{:}B.\ M\ i)$
⟨*proof*⟩

lemma *ccINF-inf-distrib*: *countable* $A \Longrightarrow inf\ (INF\ a{:}A.\ f\ a)\ (INF\ a{:}A.\ g\ a) = (INF\ a{:}A.\ inf\ (f\ a)\ (g\ a))$
⟨*proof*⟩

lemma *ccSUP-sup-distrib*: *countable* $A \Longrightarrow sup\ (SUP\ a{:}A.\ f\ a)\ (SUP\ a{:}A.\ g\ a) = (SUP\ a{:}A.\ sup\ (f\ a)\ (g\ a))$
⟨*proof*⟩

lemma *ccINF-const* [*simp*]: $A \neq \{\} \Longrightarrow (INF\ i\ {:}A.\ f) = f$
⟨*proof*⟩

lemma *ccSUP-const* [*simp*]: $A \neq \{\} \Longrightarrow (SUP\ i\ {:}A.\ f) = f$
⟨*proof*⟩

lemma *ccINF-top* [*simp*]: $(INF\ x{:}A.\ top) = top$
⟨*proof*⟩

lemma *ccSUP-bot* [*simp*]: $(SUP\ x{:}A.\ bot) = bot$
⟨*proof*⟩

lemma *ccINF-commute*: *countable* $A \Longrightarrow$ *countable* $B \Longrightarrow (INF\ i{:}A.\ INF\ j{:}B.\ f\ i\ j) = (INF\ j{:}B.\ INF\ i{:}A.\ f\ i\ j)$
⟨*proof*⟩

lemma *ccSUP-commute*: *countable* $A \Longrightarrow$ *countable* $B \Longrightarrow (SUP\ i{:}A.\ SUP\ j{:}B.\ f\ i\ j) = (SUP\ j{:}B.\ SUP\ i{:}A.\ f\ i\ j)$
⟨*proof*⟩

end

context
fixes $a :: 'a{::}\{$*countable-complete-lattice*, *linorder*$\}$
begin

lemma *less-ccSup-iff*: *countable* $S \Longrightarrow a < Sup\ S \longleftrightarrow (\exists\, x{\in}S.\ a < x)$
⟨*proof*⟩

lemma *less-ccSUP-iff*: *countable* $A \Longrightarrow a < (SUP\ i{:}A.\ f\ i) \longleftrightarrow (\exists\, x{\in}A.\ a < f\ x)$
⟨*proof*⟩

lemma *ccInf-less-iff*: *countable* $S \Longrightarrow Inf\ S < a \longleftrightarrow (\exists\, x{\in}S.\ x < a)$
⟨*proof*⟩

lemma *ccINF-less-iff*: *countable* $A \Longrightarrow (INF\ i{:}A.\ f\ i) < a \longleftrightarrow (\exists\, x{\in}A.\ f\ x < a)$
⟨*proof*⟩

end

class *countable-complete-distrib-lattice* = *countable-complete-lattice* +
assumes *sup-ccInf*: *countable* $B \Longrightarrow sup\ a\ (Inf\ B) = (INF\ b{:}B.\ sup\ a\ b)$
assumes *inf-ccSup*: *countable* $B \Longrightarrow inf\ a\ (Sup\ B) = (SUP\ b{:}B.\ inf\ a\ b)$
begin

lemma *sup-ccINF*:
countable $B \Longrightarrow sup\ a\ (INF\ b{:}B.\ f\ b) = (INF\ b{:}B.\ sup\ a\ (f\ b))$
⟨*proof*⟩

lemma *inf-ccSUP*:
countable $B \Longrightarrow inf\ a\ (SUP\ b{:}B.\ f\ b) = (SUP\ b{:}B.\ inf\ a\ (f\ b))$
⟨*proof*⟩

subclass *distrib-lattice*
⟨*proof*⟩

lemma *ccInf-sup*:

countable B $\Longrightarrow$ *sup* (*Inf B*) *a* = (*INF b*:*B*. *sup b a*)
⟨*proof*⟩

lemma *ccSup-inf*:
countable B $\Longrightarrow$ *inf* (*Sup B*) *a* = (*SUP b*:*B*. *inf b a*)
⟨*proof*⟩

lemma *ccINF-sup*:
countable B $\Longrightarrow$ *sup* (*INF b*:*B*. *f b*) *a* = (*INF b*:*B*. *sup* (*f b*) *a*)
⟨*proof*⟩

lemma *ccSUP-inf*:
countable B $\Longrightarrow$ *inf* (*SUP b*:*B*. *f b*) *a* = (*SUP b*:*B*. *inf* (*f b*) *a*)
⟨*proof*⟩

lemma *ccINF-sup-distrib2*:
countable A $\Longrightarrow$ *countable B* $\Longrightarrow$ *sup* (*INF a*:*A*. *f a*) (*INF b*:*B*. *g b*) = (*INF a*:*A*. *INF b*:*B*. *sup* (*f a*) (*g b*))
⟨*proof*⟩

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*))
⟨*proof*⟩

context
fixes *f* :: *'a* $\Rightarrow$ *'b*::*countable-complete-lattice*
assumes *mono f*
begin

lemma *mono-ccInf*:
countable A $\Longrightarrow$ *f* (*Inf A*) $\leq$ (*INF x*:*A*. *f x*)
⟨*proof*⟩

lemma *mono-ccSup*:
countable A $\Longrightarrow$ (*SUP x*:*A*. *f x*) $\leq$ *f* (*Sup A*)
⟨*proof*⟩

lemma *mono-ccINF*:
countable I $\Longrightarrow$ *f* (*INF i* : *I*. *A i*) $\leq$ (*INF x* : *I*. *f* (*A x*))
⟨*proof*⟩

lemma *mono-ccSUP*:
countable I $\Longrightarrow$ (*SUP x* : *I*. *f* (*A x*)) $\leq$ *f* (*SUP i* : *I*. *A i*)
⟨*proof*⟩

end

end

23.0.1 Instances of countable complete lattices

instance *fun* :: (*type*, *countable-complete-lattice*) *countable-complete-lattice*
⟨*proof*⟩

subclass (**in** *complete-lattice*) *countable-complete-lattice*
⟨*proof*⟩

subclass (**in** *complete-distrib-lattice*) *countable-complete-distrib-lattice*
⟨*proof*⟩

end

24 Cardinal Notations

theory *Cardinal-Notations*
imports *Main*
begin

notation
ordLeq2 (**infix** <=*o* *50*) **and**
ordLeq3 (**infix** ≤*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* ≡ *BNF-Cardinal-Arithmetic.cinfinite*
abbreviation *czero* ≡ *BNF-Cardinal-Arithmetic.czero*
abbreviation *cone* ≡ *BNF-Cardinal-Arithmetic.cone*
abbreviation *ctwo* ≡ *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* ⟷ |*A*| ≤*o* |*UNIV*::*nat set*|
⟨*proof*⟩

lemma *countable-card-le-natLeq*: *countable A* ⟷ |*A*| ≤*o* *natLeq*
⟨*proof*⟩

lemma *countable-or-card-of*:
assumes *countable A*
shows (*finite A* ∧ |*A*| <*o* |*UNIV*::*nat set*|) ∨
(*infinite A* ∧ |*A*| =*o* |*UNIV*::*nat set*|)
⟨*proof*⟩

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*|
⟨*proof*⟩

lemma *countable-or*:
countable A ⟹ (∃ *f*::'*a*⇒*nat*. *finite A* ∧ *inj-on f A*) ∨ (∃ *f*::'*a*⇒*nat*. *infinite A* ∧ *bij-betw f A UNIV*)
⟨*proof*⟩

lemma *countable-cases*[*elim*]:
assumes *countable A*
obtains (*Fin*) *f* :: '*a*⇒*nat* **where** *finite A inj-on f A*
| (*Inf*) *f* :: '*a*⇒*nat* **where** *infinite A bij-betw f A UNIV*
⟨*proof*⟩

lemma *countable-ordLeq*:
assumes |*A*| ≤*o* |*B*| **and** *countable B*
shows *countable A*
⟨*proof*⟩

lemma *countable-ordLess*:
assumes *AB*: |*A*| <*o* |*B*| **and** *B*: *countable B*
shows *countable A*
⟨*proof*⟩

25.2 The type of countable sets

typedef '*a cset* = {*A* :: '*a set*. *countable A*} **morphisms** *rcset acset*
⟨*proof*⟩

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* ⟨*proof*⟩

lift-definition *bot-cset* :: *'a cset* **is** {} **parametric** *empty-transfer* ⟨*proof*⟩

lift-definition *less-eq-cset* :: *'a cset* ⇒ *'a cset* ⇒ *bool*
is *subset-eq* **parametric** *subset-transfer* ⟨*proof*⟩

definition *less-cset* :: *'a cset* ⇒ *'a cset* ⇒ *bool*
where *xs* < *ys* ≡ *xs* ≤ *ys* ∧ *xs* ≠ (*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* <
⟨*proof*⟩

lift-definition *sup-cset* :: *'a cset* ⇒ *'a cset* ⇒ *'a cset*
is *union* **parametric** *union-transfer* ⟨*proof*⟩

lift-definition *inf-cset* :: *'a cset* ⇒ *'a cset* ⇒ *'a cset*
is *inter* **parametric** *inter-transfer* ⟨*proof*⟩

lift-definition *minus-cset* :: *'a cset* ⇒ *'a cset* ⇒ *'a cset*
is *minus* **parametric** *Diff-transfer* ⟨*proof*⟩

instance ⟨*proof*⟩

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*
⟨*proof*⟩
lift-definition *cinsert* :: *'a* ⇒ *'a cset* ⇒ *'a cset* **is** *insert* **parametric** *Lifting-Set.insert-transfer*
⟨*proof*⟩
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
⟨*proof*⟩
lift-definition *cBall* :: *'a cset* ⇒ (*'a* ⇒ *bool*) ⇒ *bool* **is** *Ball* **parametric** *Ball-transfer*
⟨*proof*⟩
lift-definition *cBex* :: *'a cset* ⇒ (*'a* ⇒ *bool*) ⇒ *bool* **is** *Bex* **parametric** *Bex-transfer*

⟨*proof*⟩
lift-definition *cUNION* :: *'a cset* ⇒ (*'a* ⇒ *'b cset*) ⇒ *'b cset*
  is *UNION* **parametric** *UNION-transfer* ⟨*proof*⟩
definition *cUnion* :: *'a cset cset* ⇒ *'a cset* **where** *cUnion A* = *cUNION A id*

lemma *Union-conv-UNION*: ⋃ *A* = *UNION A id*
⟨*proof*⟩

lemma *cUnion-transfer* [*transfer-rule*]:
  rel-fun (*pcr-cset* (*pcr-cset A*)) (*pcr-cset A*) *Union cUnion*
⟨*proof*⟩

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* ⟨*proof*⟩

25.3.2 *cinsert*

lemma *countable-insert-iff*: *countable* (*insert x A*) $\longleftrightarrow$ *countable A*
⟨*proof*⟩

lemma *set-cinsert*:
assumes *cin x A*
obtains *B* **where** *A* = *cinsert x B* **and** ¬ *cin x B*
⟨*proof*⟩

lemma *mk-disjoint-cinsert*: *cin a A* $\Longrightarrow$ ∃ *B*. *A* = *cinsert a B* ∧ ¬ *cin a B*
⟨*proof*⟩

25.3.3 *cimage*

lemma *subset-cimage-iff*: *csubset-eq B* (*cimage f A*) $\longleftrightarrow$ (∃ *AA*. *csubset-eq AA A* ∧ *B* = *cimage f AA*)
⟨*proof*⟩

25.3.4 bounded quantification

lemma *cBex-simps* [*simp*, *no-atp*]:
⋀*A P Q*. *cBex A* (λ*x*. *P x* ∧ *Q*) = (*cBex A P* ∧ *Q*)
⋀*A P Q*. *cBex A* (λ*x*. *P* ∧ *Q x*) = (*P* ∧ *cBex A Q*)
⋀*P*. *cBex cempty P* = *False*
⋀*a B P*. *cBex* (*cinsert a B*) *P* = (*P a* ∨ *cBex B P*)
⋀*A P f*. *cBex* (*cimage f A*) *P* = *cBex A* (λ*x*. *P* (*f x*))
⋀*A P*. (¬ *cBex A P*) = *cBall A* (λ*x*. ¬ *P x*)
⟨*proof*⟩

lemma *cBall-simps* [*simp*, *no-atp*]:
⋀*A P Q*. *cBall A* (λ*x*. *P x* ∨ *Q*) = (*cBall A P* ∨ *Q*)
⋀*A P Q*. *cBall A* (λ*x*. *P* ∨ *Q x*) = (*P* ∨ *cBall A Q*)
⋀*A P Q*. *cBall A* (λ*x*. *P* ⟶ *Q x*) = (*P* ⟶ *cBall A Q*)
⋀*A P Q*. *cBall A* (λ*x*. *P x* ⟶ *Q*) = (*cBex A P* ⟶ *Q*)
⋀*P*. *cBall cempty P* = *True*
⋀*a B P*. *cBall* (*cinsert a B*) *P* = (*P a* ∧ *cBall B P*)

$\bigwedge A\ P\ f.\ cBall\ (cimage\ f\ A)\ P = cBall\ A\ (\lambda x.\ P\ (f\ x))$
$\bigwedge A\ P.\ (\neg\ cBall\ A\ P) = cBex\ A\ (\lambda x.\ \neg\ P\ x)$
⟨*proof*⟩

lemma *atomize-cBall*:
$(\bigwedge x.\ cin\ x\ A \Longrightarrow P\ x) == Trueprop\ (cBall\ A\ (\lambda x.\ P\ x))$
⟨*proof*⟩

25.3.5 *cUnion*

lemma *cUNION-cimage*: $cUNION\ (cimage\ f\ A)\ g = cUNION\ A\ (g \circ f)$
including *cset.lifting* ⟨*proof*⟩

25.4 Setup for Lifting/Transfer

25.4.1 Relator and predicator properties

lift-definition *rel-cset* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a\ cset \Rightarrow 'b\ cset \Rightarrow bool$
is *rel-set* **parametric** *rel-set-transfer* ⟨*proof*⟩

lemma *rel-cset-alt-def*:
rel-cset R a b ⟷
$(\forall\, t \in rcset\ a.\ \exists\, u \in rcset\ b.\ R\ t\ u) \wedge$
$(\forall\, t \in rcset\ b.\ \exists\, u \in rcset\ a.\ R\ u\ t)$
⟨*proof*⟩

lemma *rel-cset-iff*:
rel-cset R a b ⟷
$(\forall\, t.\ cin\ t\ a \longrightarrow (\exists\, u.\ cin\ u\ b \wedge R\ t\ u)) \wedge$
$(\forall\, t.\ cin\ t\ b \longrightarrow (\exists\, u.\ cin\ u\ a \wedge R\ u\ t))$
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *rel-cset-csingle-iff* [*simp*]: *rel-cset R* (*csingle x*) (*csingle y*) ⟷ *R x y*
⟨*proof*⟩

25.4.2 Transfer rules for the Transfer package

Unconditional transfer rules

context begin interpretation *lifting-syntax* ⟨*proof*⟩

lemmas *cempty-parametric* [*transfer-rule*] = *empty-transfer*[*Transfer.transferred*]

lemma *cinsert-parametric* [*transfer-rule*]:
(*A* ===> *rel-cset A* ===> *rel-cset A*) *cinsert cinsert*
⟨*proof*⟩

lemma *cUn-parametric* [*transfer-rule*]:
(*rel-cset A* ===> *rel-cset A* ===> *rel-cset A*) *cUn cUn*
⟨*proof*⟩

lemma *cUnion-parametric* [*transfer-rule*]:
(*rel-cset* (*rel-cset A*) ===> *rel-cset A*) *cUnion cUnion*
⟨*proof*⟩

lemma *cimage-parametric* [*transfer-rule*]:
((*A* ===> *B*) ===> *rel-cset A* ===> *rel-cset B*) *cimage cimage*
⟨*proof*⟩

lemma *cBall-parametric* [*transfer-rule*]:
(*rel-cset A* ===> (*A* ===> *op* =) ===> *op* =) *cBall cBall*
⟨*proof*⟩

lemma *cBex-parametric* [*transfer-rule*]:
(*rel-cset A* ===> (*A* ===> *op* =) ===> *op* =) *cBex cBex*
⟨*proof*⟩

lemma *rel-cset-parametric* [*transfer-rule*]:
((*A* ===> *B* ===> *op* =) ===> *rel-cset A* ===> *rel-cset B* ===> *op* =)
rel-cset rel-cset
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *cInt-parametric* [*transfer-rule*]:
bi-unique A ⟹ (*rel-cset A* ===> *rel-cset A* ===> *rel-cset A*) *cInt cInt*
⟨*proof*⟩

lemma *cDiff-parametric* [*transfer-rule*]:
bi-unique A ⟹ (*rel-cset A* ===> *rel-cset A* ===> *rel-cset A*) *cDiff cDiff*
⟨*proof*⟩

lemma *csubset-parametric* [*transfer-rule*]:
bi-unique A ⟹ (*rel-cset A* ===> *rel-cset A* ===> *op* =) *csubset-eq csubset-eq*
⟨*proof*⟩

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 $|\{X.\ X \subseteq A \wedge countable\ X \wedge X \neq \{\}\}| \leq o\ |\{f{::}nat \Rightarrow 'a.\ range\ f \subseteq A\}|$
⟨*proof*⟩

lemma *card-of-countable-sets-Func*:
$|\{X.\ X \subseteq A \wedge countable\ X \wedge X \neq \{\}\}| \leq o\ |A|\ \hat{}c\ natLeq$
⟨*proof*⟩

lemma *ordLeq-countable-subsets*:
$|A| \leq o\ |\{X.\ X \subseteq A \wedge countable\ X\}|$
⟨*proof*⟩

lemma *finite-countable-subset*:
$finite\ \{X.\ X \subseteq A \wedge countable\ X\} \longleftrightarrow finite\ A$
⟨*proof*⟩

lemma *rcset-to-rcset*: $countable\ A \Longrightarrow rcset\ (the\text{-}inv\ rcset\ A) = A$
including *cset.lifting*
⟨*proof*⟩

lemma *Collect-Int-Times*: $\{(x,\ y).\ R\ x\ y\} \cap A \times B = \{(x,\ y).\ R\ x\ y \wedge x \in A \wedge y \in B\}$
⟨*proof*⟩

lemma *rel-cset-aux*:
$(\forall t \in rcset\ a.\ \exists u \in rcset\ b.\ R\ t\ u) \wedge (\forall t \in rcset\ b.\ \exists u \in rcset\ a.\ R\ u\ t) \longleftrightarrow$
$((BNF\text{-}Def.Grp\ \{x.\ rcset\ x \subseteq \{(a,\ b).\ R\ a\ b\}\}\ (cimage\ fst))^{-1-1}\ OO$
$BNF\text{-}Def.Grp\ \{x.\ rcset\ x \subseteq \{(a,\ b).\ R\ a\ b\}\}\ (cimage\ snd))\ a\ b\ (\mathbf{is}\ ?L = ?R)$
⟨*proof*⟩ **including** *cset.lifting*
⟨*proof*⟩

bnf $'a\ cset$
map: *cimage*
sets: *rcset*
bd: *natLeq*
wits: *cempty*
rel: *rel-cset*
⟨*proof*⟩ **including** *cset.lifting* ⟨*proof*⟩ **including** *cset.lifting* ⟨*proof*⟩
including *cset.lifting* ⟨*proof*⟩

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*)
⟨*proof*⟩ **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*)
⟨*proof*⟩ **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*]:
$subseq\ s \Longrightarrow subseq\ (reduce\ s\ n)$
⟨*proof*⟩

lemma *reduce-holds*:
$subseq\ s \Longrightarrow P\ n\ (s\ o\ reduce\ s\ n)$
⟨*proof*⟩

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*⟩

lemma *seqseq-holds*:
$P\ n\ (seqseq\ (Suc\ n))$
⟨*proof*⟩

definition *diagseq* **where** $diagseq\ i = seqseq\ i\ i$

lemma *subseq-mono*: $subseq\ f \Longrightarrow a \leq b \Longrightarrow f\ a \leq f\ b$
⟨*proof*⟩

lemma *subseq-strict-mono*: $subseq\ f \Longrightarrow a < b \Longrightarrow f\ a < f\ b$
⟨*proof*⟩

lemma *diagseq-mono*: $diagseq\ n < diagseq\ (Suc\ n)$
⟨*proof*⟩

lemma *subseq-diagseq*: *subseq diagseq*
⟨*proof*⟩

primrec *fold-reduce* **where**
$fold\text{-}reduce\ n\ 0 = id$
| $fold\text{-}reduce\ n\ (Suc\ k) = fold\text{-}reduce\ n\ k\ o\ reduce\ (seqseq\ (n + k))\ (n + k)$

lemma *subseq-fold-reduce*[*intro*, *simp*]: $subseq\ (fold\text{-}reduce\ n\ k)$
⟨*proof*⟩

lemma *ex-subseq-reduce-index*: $seqseq\ (n + k) = seqseq\ n\ o\ fold\text{-}reduce\ n\ k$
⟨*proof*⟩

lemma *seqseq-fold-reduce*: *seqseq n = fold-reduce 0 n*
⟨*proof*⟩

lemma *diagseq-fold-reduce*: *diagseq n = fold-reduce 0 n n*
⟨*proof*⟩

lemma *fold-reduce-add*: *fold-reduce 0 (m + n) = fold-reduce 0 m o fold-reduce m n*
⟨*proof*⟩

lemma *diagseq-add*: *diagseq (k + n) = (seqseq k o (fold-reduce k n)) (k + n)*
⟨*proof*⟩

lemma *diagseq-sub*:
assumes $m \leq n$ **shows** *diagseq n = (seqseq m o (fold-reduce m (n − m))) n*
⟨*proof*⟩

lemma *subseq-diagonal-rest*: *subseq (λx. fold-reduce k x (k + x))*
⟨*proof*⟩

lemma *diagseq-seqseq*: *diagseq o (op + k) = (seqseq k o (λx. fold-reduce k x (k + x)))*
⟨*proof*⟩

lemma *diagseq-holds*:
assumes *subseq-stable*: $\bigwedge r\ s\ n.$ *subseq r* $\Longrightarrow$ *P n s* $\Longrightarrow$ *P n (s o r)*
shows *P k (diagseq o (op + (Suc k)))*
⟨*proof*⟩

end

end

28 Handling Disjoint Sets

theory *Disjoint-Sets*
imports *Main*
begin

lemma *range-subsetD*: *range f* $\subseteq$ *B* $\Longrightarrow$ *f i* $\in$ *B*
⟨*proof*⟩

lemma *Int-Diff-disjoint*: $A \cap B \cap (A - B) = \{\}$
⟨*proof*⟩

lemma *Int-Diff-Un*: $A \cap B \cup (A - B) = A$
⟨*proof*⟩

lemma *mono-Un*: *mono A* $\Longrightarrow$ $(\bigcup i{\leq}n.\ A\ i) = A\ n$

⟨*proof*⟩

28.1 Set of Disjoint Sets

abbreviation *disjoint* :: *'a set set* ⇒ *bool* **where** *disjoint* ≡ *pairwise disjnt*

lemma *disjoint-def*: *disjoint A* ⟷ (∀ *a*∈*A*. ∀ *b*∈*A*. *a* ≠ *b* ⟶ *a* ∩ *b* = {})
⟨*proof*⟩

lemma *disjointI*:
(⋀*a b*. *a* ∈ *A* ⟹ *b* ∈ *A* ⟹ *a* ≠ *b* ⟹ *a* ∩ *b* = {}) ⟹ *disjoint A*
⟨*proof*⟩

lemma *disjointD*:
disjoint A ⟹ *a* ∈ *A* ⟹ *b* ∈ *A* ⟹ *a* ≠ *b* ⟹ *a* ∩ *b* = {}
⟨*proof*⟩

lemma *disjoint-INT*:
assumes ∗: ⋀*i*. *i* ∈ *I* ⟹ *disjoint* (*F i*)
shows *disjoint* {⋂ *i*∈*I*. *X i* | *X*. ∀ *i*∈*I*. *X i* ∈ *F i*}
⟨*proof*⟩

28.1.1 Family of Disjoint Sets

definition *disjoint-family-on* :: (*'i* ⇒ *'a set*) ⇒ *'i set* ⇒ *bool* **where**
disjoint-family-on A S ⟷ (∀ *m*∈*S*. ∀ *n*∈*S*. *m* ≠ *n* ⟶ *A m* ∩ *A n* = {})

abbreviation *disjoint-family A* ≡ *disjoint-family-on A UNIV*

lemma *disjoint-family-onD*:
disjoint-family-on A I ⟹ *i* ∈ *I* ⟹ *j* ∈ *I* ⟹ *i* ≠ *j* ⟹ *A i* ∩ *A j* = {}
⟨*proof*⟩

lemma *disjoint-family-subset*: *disjoint-family A* ⟹ (⋀*x*. *B x* ⊆ *A x*) ⟹ *disjoint-family B*
⟨*proof*⟩

lemma *disjoint-family-on-bisimulation*:
assumes *disjoint-family-on f S*
and ⋀*n m*. *n* ∈ *S* ⟹ *m* ∈ *S* ⟹ *n* ≠ *m* ⟹ *f n* ∩ *f m* = {} ⟹ *g n* ∩ *g m* = {}
shows *disjoint-family-on g S*
⟨*proof*⟩

lemma *disjoint-family-on-mono*:
A ⊆ *B* ⟹ *disjoint-family-on f B* ⟹ *disjoint-family-on f A*
⟨*proof*⟩

lemma *disjoint-family-Suc*:
(⋀*n*. *A n* ⊆ *A* (*Suc n*)) ⟹ *disjoint-family* (λ*i*. *A* (*Suc i*) − *A i*)

⟨*proof*⟩

lemma *disjoint-family-on-disjoint-image*:
disjoint-family-on A I ⟹ *disjoint* (*A* ‘ *I*)
⟨*proof*⟩

lemma *disjoint-family-on-vimageI*: *disjoint-family-on F I* ⟹ *disjoint-family-on* (λ*i*. *f* −‘ *F i*) *I*
⟨*proof*⟩

lemma *disjoint-image-disjoint-family-on*:
assumes *d*: *disjoint* (*A* ‘ *I*) **and** *i*: *inj-on A I*
shows *disjoint-family-on A I*
⟨*proof*⟩

lemma *disjoint-UN*:
assumes *F*: ⋀*i*. *i* ∈ *I* ⟹ *disjoint* (*F i*) **and** ∗: *disjoint-family-on* (λ*i*. ⋃ *F i*) *I*
shows *disjoint* (⋃ *i*∈*I*. *F i*)
⟨*proof*⟩

lemma *disjoint-union*: *disjoint C* ⟹ *disjoint B* ⟹ ⋃ *C* ∩ ⋃ *B* = {} ⟹ *disjoint* (*C* ∪ *B*)
⟨*proof*⟩

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*⟩

lemma *UN-disjointed-eq*: (⋃ *i*. *disjointed A i*) = (⋃ *i*. *A i*)
⟨*proof*⟩

lemma *less-disjoint-disjointed*: *m* < *n* ⟹ *disjointed A m* ∩ *disjointed A n* = {}
⟨*proof*⟩

lemma *disjoint-family-disjointed*: *disjoint-family* (*disjointed A*)
⟨*proof*⟩

lemma *disjointed-subset*: *disjointed A n* ⊆ *A n*
⟨*proof*⟩

lemma *disjointed-0*[*simp*]: *disjointed A 0* = *A 0*
⟨*proof*⟩

lemma *disjointed-mono*: *mono A* ⟹ *disjointed A* (*Suc n*) = *A* (*Suc n*) − *A n*

⟨*proof*⟩

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*⟩

setup-lifting *type-definition-dlist*

lemma *dlist-eq-iff*:
  dxs = *dys* ⟷ *list-of-dlist dxs* = *list-of-dlist dys*
  ⟨*proof*⟩

lemma *dlist-eqI*:
  list-of-dlist dxs = *list-of-dlist dys* ⟹ *dxs* = *dys*
  ⟨*proof*⟩

Formal, totalized constructor for *'a dlist*:

definition *Dlist* :: *'a list* ⇒ *'a dlist* **where**
  Dlist xs = *Abs-dlist* (*remdups xs*)

lemma *distinct-list-of-dlist* [*simp*, *intro*]:
  distinct (*list-of-dlist dxs*)
  ⟨*proof*⟩

lemma *list-of-dlist-Dlist* [*simp*]:
  list-of-dlist (*Dlist xs*) = *remdups xs*
  ⟨*proof*⟩

lemma *remdups-list-of-dlist* [*simp*]:
  remdups (*list-of-dlist dxs*) = *list-of-dlist dxs*
  ⟨*proof*⟩

lemma *Dlist-list-of-dlist* [*simp*, *code abstype*]:
  Dlist (*list-of-dlist dxs*) = *dxs*
  ⟨*proof*⟩

Fundamental operations:

context

begin

qualified definition *empty* :: *'a dlist* **where**
empty = *Dlist* []

qualified definition *insert* :: *'a* ⇒ *'a dlist* ⇒ *'a dlist* **where**
insert x dxs = *Dlist* (*List.insert x* (*list-of-dlist dxs*))

qualified definition *remove* :: *'a* ⇒ *'a dlist* ⇒ *'a dlist* **where**
remove x dxs = *Dlist* (*remove1 x* (*list-of-dlist dxs*))

qualified definition *map* :: (*'a* ⇒ *'b*) ⇒ *'a dlist* ⇒ *'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 = []
⟨*proof*⟩

lemma *list-of-dlist-insert* [*simp*, *code abstract*]:
list-of-dlist (*Dlist.insert x dxs*) = *List.insert x* (*list-of-dlist dxs*)
⟨*proof*⟩

lemma *list-of-dlist-remove* [*simp*, *code abstract*]:
list-of-dlist (*Dlist.remove x dxs*) = *remove1 x* (*list-of-dlist dxs*)
⟨*proof*⟩

lemma *list-of-dlist-map* [*simp*, *code abstract*]:
list-of-dlist (*Dlist.map f dxs*) = *remdups* (*List.map f* (*list-of-dlist dxs*))
⟨*proof*⟩

lemma *list-of-dlist-filter* [*simp*, *code abstract*]:
list-of-dlist (*Dlist.filter P dxs*) = *List.filter P* (*list-of-dlist dxs*)
⟨*proof*⟩

Explicit executable conversion

definition *dlist-of-list* [*simp*]:
dlist-of-list = *Dlist*

lemma [*code abstract*]:
list-of-dlist (*dlist-of-list xs*) = *remdups xs*
⟨*proof*⟩

Equality

instantiation *dlist* :: (*equal*) *equal*
begin

definition *HOL.equal dxs dys* ⟷ *HOL.equal* (*list-of-dlist dxs*) (*list-of-dlist dys*)

instance
⟨*proof*⟩

end

declare *equal-dlist-def* [*code*]

lemma [*code nbe*]: *HOL.equal* (*dxs* :: 'a::*equal dlist*) *dxs* ⟷ *True*
⟨*proof*⟩

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*⟩

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*⟩

29.4 Functorial structure

functor *map*: *map*
⟨*proof*⟩

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* = []
⟨*proof*⟩ **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*: ⋀*a b xs b' c ys*.
⟦ *b* ∈ *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*: ⋀*a b xs b' c ys*.
⟦ *b* ∉ *snd* ‘ *set xs*; *b'* ∈ *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*: ⋀*a b xs c ys*.
⟦ *b* ∉ *snd* ‘ *set xs*; *b* ∉ *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*
⟨*proof*⟩ **lemma** *set-wpull-subset*:
assumes *remdups* (*map snd xs*) = *remdups* (*map fst ys*)

shows *set* (*wpull xs ys*) ⊆ *set xs O set ys*
⟨*proof*⟩ **lemma** *set-fst-wpull*:
assumes *remdups* (*map snd xs*) = *remdups* (*map fst ys*)
shows *fst ' set* (*wpull xs ys*) = *fst ' set xs*
⟨*proof*⟩ **lemma** *set-snd-wpull*:
assumes *remdups* (*map snd xs*) = *remdups* (*map fst ys*)
shows *snd ' set* (*wpull xs ys*) = *snd ' set ys*
⟨*proof*⟩ **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*⟩ **lift-definition** *set* :: *'a dlist* ⇒ *'a set* **is** *List.set* ⟨*proof*⟩ **lemma** *map-transfer* [*transfer-rule*]:
(*rel-fun op* = (*rel-fun* (*pcr-dlist op* =) (*pcr-dlist op* =))) (λ*f x*. *remdups* (*List.map f x*)) *Dlist.map*
⟨*proof*⟩

bnf *'a dlist*
map: *Dlist.map*
sets: *set*
bd: *natLeq*
wits: *Dlist.empty*
⟨*proof*⟩

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*⟩

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
⟨*proof*⟩

end

instance *extended* :: (*linorder*)*linorder*
⟨*proof*⟩

lemma *Minf-le*[*simp*]: *Minf* ≤ *y*
⟨*proof*⟩
lemma *le-Pinf*[*simp*]: *x* ≤ *Pinf*
⟨*proof*⟩
lemma *le-Minf*[*simp*]: *x* ≤ *Minf* ⟷ *x* = *Minf*
⟨*proof*⟩
lemma *Pinf-le*[*simp*]: *Pinf* ≤ *x* ⟷ *x* = *Pinf*
⟨*proof*⟩

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*
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

instantiation *extended* :: (*zero*)*zero*
begin
definition *0* = *Fin*(*0*::'*a*)
instance ⟨*proof*⟩
end

declare *zero-extended-def*[*symmetric*, *code-post*]

instantiation *extended* :: (*one*)*one*
begin
definition *1* = *Fin*(*1*::'*a*)
instance ⟨*proof*⟩
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 ⟨*proof*⟩

end

instance *extended* :: (*ab-semigroup-add*)*ab-semigroup-add*
⟨*proof*⟩

instance *extended* :: (*ordered-ab-semigroup-add*)*ordered-ab-semigroup-add*
⟨*proof*⟩

instance *extended* :: (*comm-monoid-add*)*comm-monoid-add*
⟨*proof*⟩

instantiation *extended* :: (*uminus*)*uminus*
begin

fun *uminus-extended* **where**
− (*Fin x*) = *Fin* (− *x*) |
− *Pinf* = *Minf* |
− *Minf* = *Pinf*

instance ⟨*proof*⟩

end

instantiation *extended* :: (*ab-group-add*)*minus*
begin
definition *x* − *y* = *x* + −(*y*::′*a extended*)
instance ⟨*proof*⟩
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*
⟨*proof*⟩

Numerals:

instance *extended* :: ({*ab-semigroup-add*,*one*})*numeral* ⟨*proof*⟩

lemma *Fin-numeral*[*code-post*]: *Fin*(*numeral w*) = *numeral w*
⟨*proof*⟩

lemma *Fin-neg-numeral*[*code-post*]: *Fin* (− *numeral w*) = − *numeral w*
⟨*proof*⟩

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
⟨*proof*⟩
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*)
⟨*proof*⟩

lemma *INF-nat-binary*:
(*INF n*::*nat*. *if n = 0 then A else B*) = (*inf A B*::'*a*::*countable-complete-lattice*)
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *sup-continuous-mono*:
assumes [*simp*]: *sup-continuous F* **shows** *mono F*
⟨*proof*⟩

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* (λ*f*. *f x*)
and *sup-continuous-fun*: (⋀*s*. *sup-continuous* (λ*x*. *P x s*)) ⟹ *sup-continuous P*
and *sup-continuous-If*: *sup-continuous F* ⟹ *sup-continuous G* ⟹ *sup-continuous* (λ*f*. *if C then F f else G f*)
⟨*proof*⟩

lemma *sup-continuous-compose*:
assumes *f*: *sup-continuous f* **and** *g*: *sup-continuous g*
shows *sup-continuous* (λ*x*. *f* (*g x*))
⟨*proof*⟩

lemma *sup-continuous-sup*[*order-continuous-intros*]:
sup-continuous f ⟹ *sup-continuous g* ⟹ *sup-continuous* (λ*x*. *sup* (*f x*) (*g x*))
⟨*proof*⟩

lemma *sup-continuous-inf*[*order-continuous-intros*]:
fixes *P Q* :: '*a* :: *countable-complete-lattice* ⇒ '*b* :: *countable-complete-distrib-lattice*
assumes *P*: *sup-continuous P* **and** *Q*: *sup-continuous Q*
shows *sup-continuous* (λ*x*. *inf* (*P x*) (*Q x*))

⟨*proof*⟩

lemma *sup-continuous-and*[*order-continuous-intros*]:
sup-continuous P ⟹ *sup-continuous Q* ⟹ *sup-continuous* (λ*x*. *P x* ∧ *Q x*)
⟨*proof*⟩

lemma *sup-continuous-or*[*order-continuous-intros*]:
sup-continuous P ⟹ *sup-continuous Q* ⟹ *sup-continuous* (λ*x*. *P x* ∨ *Q x*)
⟨*proof*⟩

lemma *sup-continuous-lfp*:
assumes *sup-continuous F* **shows** *lfp F* = (*SUP i*. (*F* ^^ *i*) *bot*) (**is** *lfp F* = *?U*)
⟨*proof*⟩

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*]: ⋀*x*. *P x* ⟹ *x* ≤ *lfp f* ⟹ α (*f x*) = *g* (α *x*)
assumes *g-bound*: ⋀*x*. α *bot* ≤ *g x*
shows α (*lfp f*) = *lfp g*
⟨*proof*⟩

lemma *lfp-transfer*:
sup-continuous α ⟹ *sup-continuous f* ⟹ *sup-continuous g* ⟹
(⋀*x*. α *bot* ≤ *g x*) ⟹ (⋀*x*. *x* ≤ *lfp f* ⟹ α (*f x*) = *g* (α *x*)) ⟹ α (*lfp f*) = *lfp g*
⟨*proof*⟩

definition
inf-continuous :: ('*a*::*countable-complete-lattice* ⇒ '*b*::*countable-complete-lattice*) ⇒ *bool*
where
inf-continuous F ⟷ (∀ *M*::*nat* ⇒ '*a*. *antimono M* ⟶ *F* (*INF i*. *M i*) = (*INF i*. *F* (*M i*)))

lemma *inf-continuousD*: *inf-continuous F* ⟹ *antimono M* ⟹ *F* (*INF i*::*nat*. *M i*) = (*INF i*. *F* (*M i*))
⟨*proof*⟩

lemma *inf-continuous-mono*:
assumes [*simp*]: *inf-continuous F* **shows** *mono F*
⟨*proof*⟩

lemma [*order-continuous-intros*]:
shows *inf-continuous-const*: *inf-continuous* (λ*x*. *c*)
and *inf-continuous-id*: *inf-continuous* (λ*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* $G\ f$)
⟨*proof*⟩

lemma *inf-continuous-inf*[*order-continuous-intros*]:
inf-continuous $f \Longrightarrow$ *inf-continuous* $g \Longrightarrow$ *inf-continuous* ($\lambda x.$ *inf* ($f\ x$) ($g\ x$))
⟨*proof*⟩

lemma *inf-continuous-sup*[*order-continuous-intros*]:
fixes $P\ Q$ :: *'a* :: *countable-complete-lattice* ⇒ *'b* :: *countable-complete-distrib-lattice*
assumes P: *inf-continuous* P **and** Q: *inf-continuous* Q
shows *inf-continuous* ($\lambda x.$ *sup* ($P\ x$) ($Q\ x$))
⟨*proof*⟩

lemma *inf-continuous-and*[*order-continuous-intros*]:
inf-continuous $P \Longrightarrow$ *inf-continuous* $Q \Longrightarrow$ *inf-continuous* ($\lambda x.\ P\ x \wedge Q\ x$)
⟨*proof*⟩

lemma *inf-continuous-or*[*order-continuous-intros*]:
inf-continuous $P \Longrightarrow$ *inf-continuous* $Q \Longrightarrow$ *inf-continuous* ($\lambda x.\ P\ x \vee Q\ x$)
⟨*proof*⟩

lemma *inf-continuous-compose*:
assumes f: *inf-continuous* f **and** g: *inf-continuous* g
shows *inf-continuous* ($\lambda x.\ f\ (g\ x)$)
⟨*proof*⟩

lemma *inf-continuous-gfp*:
assumes *inf-continuous* F **shows** *gfp* F = (*INF* $i.$ (F ^^ i) *top*) (**is** *gfp* F = *?U*)
⟨*proof*⟩

lemma *gfp-transfer*:
assumes α: *inf-continuous* α **and** f: *inf-continuous* f **and** g: *inf-continuous* g
assumes [*simp*]: α *top* = *top* $\bigwedge x.\ \alpha\ (f\ x) = g\ (\alpha\ x)$
shows α (*gfp* f) = *gfp* g
⟨*proof*⟩

lemma *gfp-transfer-bounded*:
assumes P: P (f *top*) $\bigwedge x.\ P\ x \Longrightarrow P\ (f\ x)$ $\bigwedge M.$ *antimono* $M \Longrightarrow (\bigwedge i.\ P\ (M\ i)) \Longrightarrow P$ (*INF* i::*nat*. $M\ i$)
assumes α: $\bigwedge M.$ *antimono* $M \Longrightarrow (\bigwedge i$::*nat*. $P\ (M\ i)) \Longrightarrow \alpha$ (*INF* $i.\ M\ i$) = (*INF* $i.\ \alpha\ (M\ i)$)
assumes f: *inf-continuous* f **and** g: *inf-continuous* g
assumes [*simp*]: $\bigwedge x.\ P\ x \Longrightarrow \alpha\ (f\ x) = g\ (\alpha\ x)$
assumes *g-bound*: $\bigwedge x.\ g\ x \leq \alpha\ (f\ top)$
shows α (*gfp* f) = *gfp* g
⟨*proof*⟩

30.1.1 Least fixed points in countable complete lattices

definition (**in** *countable-complete-lattice*) *cclfp* :: $('a \Rightarrow 'a) \Rightarrow 'a$
where $cclfp\ f = (SUP\ i.\ (f\ \hat{}\hat{}\ i)\ bot)$

lemma *cclfp-unfold*:
assumes *sup-continuous F* **shows** $cclfp\ F = F\ (cclfp\ F)$
⟨*proof*⟩

lemma *cclfp-lowerbound*: **assumes** *f*: *mono f* **and** *A*: $f\ A \leq A$ **shows** $cclfp\ f \leq A$
⟨*proof*⟩

lemma *cclfp-transfer*:
assumes *sup-continuous* α *mono f*
assumes $\alpha\ bot = bot$ $\bigwedge x.\ \alpha\ (f\ x) = g\ (\alpha\ x)$
shows $\alpha\ (cclfp\ f) = cclfp\ g$
⟨*proof*⟩

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* $(SUP\ n.\ \sum i{<}n.\ f\ i)$
⟨*proof*⟩

lemma *suminf-eq-SUP*: $suminf\ f = (SUP\ n.\ \sum i{<}n.\ f\ i)$
⟨*proof*⟩

end

31.1 Type definition

We extend the standard natural numbers by a special value indicating infinity.

typedef *enat* = *UNIV* :: *nat option set* ⟨*proof*⟩

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 ⟨*proof*⟩

end

instance *enat* :: *countable*
⟨*proof*⟩

old-rep-datatype *enat* ∞ :: *enat*
⟨*proof*⟩

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* ≠ ∞) = (∃ *i*. *x* = *enat i*)
⟨*proof*⟩

lemma *not-enat-eq* [*iff*]: (∀ *y*. *x* ≠ *enat y*) = (*x* = ∞)
⟨*proof*⟩

lemma *enat-ex-split*: (∃ *c*::*enat*. *P c*) ⟷ *P* ∞ ∨ (∃ *c*::*nat*. *P c*)
⟨*proof*⟩

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*⟩

end

definition *eSuc* :: *enat* ⇒ *enat* **where**
eSuc i = (*case i of enat n* ⇒ *enat* (*Suc n*) | ∞ ⇒ ∞)

lemma *enat-0* [*code-post*]: *enat 0* = *0*
⟨*proof*⟩

lemma *enat-1* [*code-post*]: *enat 1* = *1*
⟨*proof*⟩

lemma *enat-0-iff*: *enat x* = *0* ⟷ *x* = *0* *0* = *enat x* ⟷ *x* = *0*
⟨*proof*⟩

lemma *enat-1-iff*: *enat x* = *1* ⟷ *x* = *1* *1* = *enat x* ⟷ *x* = *1*
⟨*proof*⟩

lemma *one-eSuc*: *1* = *eSuc 0*
⟨*proof*⟩

lemma *infinity-ne-i0* [*simp*]: (∞::*enat*) ≠ *0*
⟨*proof*⟩

lemma *i0-ne-infinity* [*simp*]: *0* ≠ (∞::*enat*)
⟨*proof*⟩

lemma *zero-one-enat-neq*:
¬ *0* = (*1*::*enat*)
¬ *1* = (*0*::*enat*)
⟨*proof*⟩

lemma *infinity-ne-i1* [*simp*]: (∞::*enat*) ≠ *1*
⟨*proof*⟩

lemma *i1-ne-infinity* [*simp*]: *1* ≠ (∞::*enat*)
⟨*proof*⟩

lemma *eSuc-enat*: *eSuc* (*enat n*) = *enat* (*Suc n*)
⟨*proof*⟩

lemma *eSuc-infinity* [*simp*]: *eSuc* ∞ = ∞
⟨*proof*⟩

lemma *eSuc-ne-0* [*simp*]: *eSuc n* ≠ *0*
⟨*proof*⟩

lemma *zero-ne-eSuc* [*simp*]: *0* ≠ *eSuc n*
⟨*proof*⟩

lemma *eSuc-inject* [*simp*]: *eSuc m* = *eSuc n* ⟷ *m* = *n*

⟨*proof*⟩

lemma *eSuc-enat-iff*: *eSuc x* = *enat y* ⟷ (∃ *n*. *y* = *Suc n* ∧ *x* = *enat n*)
⟨*proof*⟩

lemma *enat-eSuc-iff*: *enat y* = *eSuc x* ⟷ (∃ *n*. *y* = *Suc n* ∧ *enat n* = *x*)
⟨*proof*⟩

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* + ∞ = ∞
⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *eSuc-plus-1*:
eSuc n = *n* + *1*
⟨*proof*⟩

lemma *plus-1-eSuc*:
1 + *q* = *eSuc q*
q + *1* = *eSuc q*
⟨*proof*⟩

lemma *iadd-Suc*: *eSuc m* + *n* = *eSuc* (*m* + *n*)
⟨*proof*⟩

lemma *iadd-Suc-right*: *m* + *eSuc n* = *eSuc* (*m* + *n*)
⟨*proof*⟩

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*]:
$enat\ m * enat\ n = enat\ (m * n)$
$\infty * \infty = (\infty::enat)$
$\infty * enat\ n = (if\ n = 0\ then\ 0\ else\ \infty)$
$enat\ m * \infty = (if\ m = 0\ then\ 0\ else\ \infty)$
⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *mult-eSuc*: $eSuc\ m * n = n + m * n$
⟨*proof*⟩

lemma *mult-eSuc-right*: $m * eSuc\ n = m + m * n$
⟨*proof*⟩

lemma *of-nat-eq-enat*: $of\text{-}nat\ n = enat\ n$
⟨*proof*⟩

instance *enat* :: *semiring-char-0*
⟨*proof*⟩

lemma *imult-is-infinity*: $((a::enat) * b = \infty) = (a = \infty \wedge b \neq 0 \vee b = \infty \wedge a \neq 0)$
⟨*proof*⟩

31.5 Numerals

lemma *numeral-eq-enat*:
$numeral\ k = enat\ (numeral\ k)$
⟨*proof*⟩

lemma *enat-numeral* [*code-abbrev*]:
$enat\ (numeral\ k) = numeral\ k$
⟨*proof*⟩

lemma *infinity-ne-numeral* [*simp*]: $(\infty::enat) \neq numeral\ k$
⟨*proof*⟩

lemma *numeral-ne-infinity* [*simp*]: $numeral\ k \neq (\infty::enat)$
⟨*proof*⟩

lemma *eSuc-numeral* [*simp*]: $eSuc\ (numeral\ k) = numeral\ (k + Num.One)$
⟨*proof*⟩

31.6 Subtraction

instantiation *enat* :: *minus*
begin

definition *diff-enat-def*:
$a - b = (case\ a\ of\ (enat\ x) \Rightarrow (case\ b\ of\ (enat\ y) \Rightarrow enat\ (x - y) \mid \infty \Rightarrow 0)$
$\quad \mid \infty \Rightarrow \infty)$

instance ⟨*proof*⟩

end

lemma *idiff-enat-enat* [*simp*, *code*]: $enat\ a - enat\ b = enat\ (a - b)$
⟨*proof*⟩

lemma *idiff-infinity* [*simp*, *code*]: $\infty - n = (\infty::enat)$
⟨*proof*⟩

lemma *idiff-infinity-right* [*simp*, *code*]: $enat\ a - \infty = 0$
⟨*proof*⟩

lemma *idiff-0* [*simp*]: $(0::enat) - n = 0$
⟨*proof*⟩

lemmas *idiff-enat-0* [*simp*] = *idiff-0* [*unfolded zero-enat-def*]

lemma *idiff-0-right* [*simp*]: $(n::enat) - 0 = n$
⟨*proof*⟩

lemmas *idiff-enat-0-right* [*simp*] = *idiff-0-right* [*unfolded zero-enat-def*]

lemma *idiff-self* [*simp*]: $n \neq \infty \Longrightarrow (n::enat) - n = 0$
⟨*proof*⟩

lemma *eSuc-minus-eSuc* [*simp*]: $eSuc\ n - eSuc\ m = n - m$
⟨*proof*⟩

lemma *eSuc-minus-1* [*simp*]: $eSuc\ n - 1 = n$
⟨*proof*⟩

31.7 Ordering

instantiation *enat* :: *linordered-ab-semigroup-add*
begin

definition [*nitpick-simp*]:
$m \leq n = (case\ n\ of\ enat\ n1 \Rightarrow (case\ m\ of\ enat\ m1 \Rightarrow m1 \leq n1 \mid \infty \Rightarrow False)$
$\quad \mid \infty \Rightarrow True)$

definition [*nitpick-simp*]:
$m < n = (case\ m\ of\ enat\ m1 \Rightarrow (case\ n\ of\ enat\ n1 \Rightarrow m1 < n1 \mid \infty \Rightarrow True) \mid \infty \Rightarrow False)$

lemma *enat-ord-simps* [*simp*]:
$enat\ m \leq enat\ n \longleftrightarrow m \leq n$
$enat\ m < enat\ n \longleftrightarrow m < n$
$q \leq (\infty::enat)$
$q < (\infty::enat) \longleftrightarrow q \neq \infty$
$(\infty::enat) \leq q \longleftrightarrow q = \infty$
$(\infty::enat) < q \longleftrightarrow False$
⟨*proof*⟩

lemma *numeral-le-enat-iff* [*simp*]:
shows $numeral\ m \leq enat\ n \longleftrightarrow numeral\ m \leq n$
⟨*proof*⟩

lemma *numeral-less-enat-iff* [*simp*]:
shows $numeral\ m < enat\ n \longleftrightarrow numeral\ m < n$
⟨*proof*⟩

lemma *enat-ord-code* [*code*]:
$enat\ m \leq enat\ n \longleftrightarrow m \leq n$
$enat\ m < enat\ n \longleftrightarrow m < n$
$q \leq (\infty::enat) \longleftrightarrow True$
$enat\ m < \infty \longleftrightarrow True$
$\infty \leq enat\ n \longleftrightarrow False$
$(\infty::enat) < q \longleftrightarrow False$
⟨*proof*⟩

instance
⟨*proof*⟩

end

instance *enat* :: *dioid*
⟨*proof*⟩

instance *enat* :: {*linordered-nonzero-semiring*, *strict-ordered-comm-monoid-add*}
⟨*proof*⟩

lemma *enat-ord-number* [*simp*]:
$(numeral\ m :: enat) \leq numeral\ n \longleftrightarrow (numeral\ m :: nat) \leq numeral\ n$
$(numeral\ m :: enat) < numeral\ n \longleftrightarrow (numeral\ m :: nat) < numeral\ n$
⟨*proof*⟩

lemma *infinity-ileE* [*elim*!]: $\infty \leq enat\ m \Longrightarrow R$

⟨*proof*⟩

lemma *infinity-ilessE* [*elim*!]: $\infty < enat\ m \Longrightarrow R$
⟨*proof*⟩

lemma *eSuc-ile-mono* [*simp*]: $eSuc\ n \leq eSuc\ m \longleftrightarrow n \leq m$
⟨*proof*⟩

lemma *eSuc-mono* [*simp*]: $eSuc\ n < eSuc\ m \longleftrightarrow n < m$
⟨*proof*⟩

lemma *ile-eSuc* [*simp*]: $n \leq eSuc\ n$
⟨*proof*⟩

lemma *not-eSuc-ilei0* [*simp*]: $\neg\ eSuc\ n \leq 0$
⟨*proof*⟩

lemma *i0-iless-eSuc* [*simp*]: $0 < eSuc\ n$
⟨*proof*⟩

lemma *iless-eSuc0*[*simp*]: $(n < eSuc\ 0) = (n = 0)$
⟨*proof*⟩

lemma *ileI1*: $m < n \Longrightarrow eSuc\ m \leq n$
⟨*proof*⟩

lemma *Suc-ile-eq*: $enat\ (Suc\ m) \leq n \longleftrightarrow enat\ m < n$
⟨*proof*⟩

lemma *iless-Suc-eq* [*simp*]: $enat\ m < eSuc\ n \longleftrightarrow enat\ m \leq n$
⟨*proof*⟩

lemma *imult-infinity*: $(0{::}enat) < n \Longrightarrow \infty * n = \infty$
⟨*proof*⟩

lemma *imult-infinity-right*: $(0{::}enat) < n \Longrightarrow n * \infty = \infty$
⟨*proof*⟩

lemma *enat-0-less-mult-iff*: $(0 < (m{::}enat) * n) = (0 < m \wedge 0 < n)$
⟨*proof*⟩

lemma *mono-eSuc*: *mono eSuc*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *enat-ile*: *n* ≤ *enat m* ⟹ ∃ *k*. *n* = *enat k*
⟨*proof*⟩

lemma *enat-iless*: *n* < *enat m* ⟹ ∃ *k*. *n* = *enat k*
⟨*proof*⟩

lemma *iadd-le-enat-iff*:
x + *y* ≤ *enat n* ⟷ (∃ *y' x'*. *x* = *enat x'* ∧ *y* = *enat y'* ∧ *x'* + *y'* ≤ *n*)
⟨*proof*⟩

lemma *chain-incr*: ∀ *i*. ∃ *j*. *Y i* < *Y j* ⟹ ∃ *j*. *enat k* < *Y j*
⟨*proof*⟩

lemma *eSuc-max*: *eSuc* (*max x y*) = *max* (*eSuc x*) (*eSuc y*)
⟨*proof*⟩

lemma *eSuc-Max*:
assumes *finite A A* ≠ {}
shows *eSuc* (*Max A*) = *Max* (*eSuc* ‘ *A*)
⟨*proof*⟩

instantiation *enat* :: {*order-bot*, *order-top*}
begin

definition *bot-enat* :: *enat* **where** *bot-enat* = *0*
definition *top-enat* :: *enat* **where** *top-enat* = ∞

instance
⟨*proof*⟩

end

lemma *finite-enat-bounded*:
assumes *le-fin*: ⋀*y*. *y* ∈ *A* ⟹ *y* ≤ *enat n*
shows *finite A*
⟨*proof*⟩

31.8 Cancellation simprocs

lemma *enat-add-left-cancel*: $a + b = a + c \longleftrightarrow a = (\infty::enat) \vee b = c$
⟨*proof*⟩

lemma *enat-add-left-cancel-le*: $a + b \leq a + c \longleftrightarrow a = (\infty::enat) \vee b \leq c$
⟨*proof*⟩

lemma *enat-add-left-cancel-less*: $a + b < a + c \longleftrightarrow a \neq (\infty::enat) \wedge b < c$
⟨*proof*⟩

⟨*ML*⟩

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
⟨*proof*⟩

lemma *less-infinityE*:
[| $n < \infty$; !!k. $n = enat\ k$ ==> P |] ==> P
⟨*proof*⟩

lemma *enat-less-induct*:
assumes *prem*: !!n. $\forall m::enat.\ m < n$ −−> $P\ m$ ==> $P\ n$ **shows** $P\ n$
⟨*proof*⟩

instance *enat* :: *wellorder*
⟨*proof*⟩

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* ∈ *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*⟩
end

instance *enat* :: *complete-linorder* ⟨*proof*⟩

lemma *eSuc-Sup*: $A \neq \{\} \Longrightarrow eSuc\ (Sup\ A) = Sup\ (eSuc\ `\ A)$
⟨*proof*⟩

lemma *sup-continuous-eSuc*: *sup-continuous f* $\Longrightarrow$ *sup-continuous* $(\lambda x.\ eSuc\ (f\ x))$
⟨*proof*⟩

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*: $(m + n = (0::enat)) = (m = 0 \wedge n = 0)$
⟨*proof*⟩

lemma *i0-lb* : $(0::enat) \leq n$
⟨*proof*⟩

lemma *ile0-eq*: $n \leq (0::enat) \longleftrightarrow n = 0$
⟨*proof*⟩

lemma *not-iless0*: $\neg\ n < (0::enat)$
⟨*proof*⟩

lemma *i0-less*[*simp*]: $(0::enat) < n \longleftrightarrow n \neq 0$
⟨*proof*⟩

lemma *imult-is-0*: $((m::enat) * n = 0) = (m = 0 \vee n = 0)$
⟨*proof*⟩

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*⟩

lemma (**in** *conditionally-complete-linorder*) *le-cSUP-iff*:
$A \neq \{\} \Longrightarrow$ *bdd-above* $(f`A) \Longrightarrow x \leq$ *SUPREMUM* $A\ f \longleftrightarrow (\forall\, y{<}x.\ \exists\, i{\in}A.\ y < f\ i)$
⟨*proof*⟩

lemma *le-cSup-iff-less*:
fixes $x :: 'a :: \{$*conditionally-complete-linorder*, *dense-linorder*$\}$
shows $A \neq \{\} \Longrightarrow$ *bdd-above* $(f`A) \Longrightarrow x \leq (SUP\ i{:}A.\ f\ i) \longleftrightarrow (\forall\, y{<}x.\ \exists\, i{\in}A.\ y \leq f\ i)$
⟨*proof*⟩

lemma *le-Sup-iff-less*:
fixes $x :: 'a :: \{$*complete-linorder*, *dense-linorder*$\}$
shows $x \leq (SUP\ i{:}A.\ f\ i) \longleftrightarrow (\forall\, y{<}x.\ \exists\, i{\in}A.\ y \leq f\ i)$ (**is** *?lhs = ?rhs*)
⟨*proof*⟩

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*⟩

lemma (**in** *conditionally-complete-linorder*) *cINF-le-iff*:
$A \neq \{\} \Longrightarrow$ *bdd-below* $(f`A) \Longrightarrow$ *INFIMUM* $A\ f \leq x \longleftrightarrow (\forall\, y{>}x.\ \exists\, i{\in}A.\ y > f\ i)$
⟨*proof*⟩

lemma *cInf-le-iff-less*:
fixes $x :: 'a :: \{$*conditionally-complete-linorder*, *dense-linorder*$\}$
shows $A \neq \{\} \Longrightarrow$ *bdd-below* $(f`A) \Longrightarrow (INF\ i{:}A.\ f\ i) \leq x \longleftrightarrow (\forall\, y{>}x.\ \exists\, i{\in}A.\ f\ i \leq y)$
⟨*proof*⟩

lemma *Inf-le-iff-less*:
fixes $x :: 'a :: \{$*complete-linorder*, *dense-linorder*$\}$
shows $(INF\ i{:}A.\ f\ i) \leq x \longleftrightarrow (\forall\, y{>}x.\ \exists\, i{\in}A.\ f\ i \leq y)$
⟨*proof*⟩

lemma *SUP-pair*:
fixes $f :: - \Rightarrow - \Rightarrow -$ :: *complete-lattice*
shows $(SUP\ i : A.\ SUP\ j : B.\ f\ i\ j) = (SUP\ p : A \times B.\ f\ (fst\ p)\ (snd\ p))$
⟨*proof*⟩

lemma *INF-pair*:
fixes $f :: - \Rightarrow - \Rightarrow -$ :: *complete-lattice*
shows $(INF\ i : A.\ INF\ j : B.\ f\ i\ j) = (INF\ p : A \times B.\ f\ (fst\ p)\ (snd\ p))$
⟨*proof*⟩

32.0.1 *Liminf* and *Limsup*

definition *Liminf* :: *'a filter* ⇒ (*'a* ⇒ *'b*) ⇒ *'b* :: *complete-lattice* **where**
Liminf F f = (*SUP P*:{*P*. *eventually P F*}. *INF x*:{*x*. *P x*}. *f x*)

definition *Limsup* :: *'a filter* ⇒ (*'a* ⇒ *'b*) ⇒ *'b* :: *complete-lattice* **where**
Limsup F f = (*INF P*:{*P*. *eventually P F*}. *SUP x*:{*x*. *P x*}. *f x*)

abbreviation *liminf* ≡ *Liminf sequentially*

abbreviation *limsup* ≡ *Limsup sequentially*

lemma *Liminf-eqI*:
(⋀*P*. *eventually P F* ⟹ *INFIMUM* (*Collect P*) *f* ≤ *x*) ⟹
(⋀*y*. (⋀*P*. *eventually P F* ⟹ *INFIMUM* (*Collect P*) *f* ≤ *y*) ⟹ *x* ≤ *y*) ⟹
Liminf F f = *x*
⟨*proof*⟩

lemma *Limsup-eqI*:
(⋀*P*. *eventually P F* ⟹ *x* ≤ *SUPREMUM* (*Collect P*) *f*) ⟹
(⋀*y*. (⋀*P*. *eventually P F* ⟹ *y* ≤ *SUPREMUM* (*Collect P*) *f*) ⟹ *y* ≤ *x*)
⟹ *Limsup F f* = *x*
⟨*proof*⟩

lemma *liminf-SUP-INF*: *liminf f* = (*SUP n*. *INF m*:{*n*..}. *f m*)
⟨*proof*⟩

lemma *limsup-INF-SUP*: *limsup f* = (*INF n*. *SUP m*:{*n*..}. *f m*)
⟨*proof*⟩

lemma *Limsup-const*:
assumes *ntriv*: ¬ *trivial-limit F*
shows *Limsup F* (λ*x*. *c*) = *c*
⟨*proof*⟩

lemma *Liminf-const*:
assumes *ntriv*: ¬ *trivial-limit F*
shows *Liminf F* (λ*x*. *c*) = *c*
⟨*proof*⟩

lemma *Liminf-mono*:
assumes *ev*: *eventually* (λ*x*. *f x* ≤ *g x*) *F*
shows *Liminf F f* ≤ *Liminf F g*
⟨*proof*⟩

lemma *Liminf-eq*:
assumes *eventually* (λ*x*. *f x* = *g x*) *F*
shows *Liminf F f* = *Liminf F g*
⟨*proof*⟩

lemma *Limsup-mono*:
assumes *ev*: *eventually* $(\lambda x.\ f\ x \leq g\ x)\ F$
shows *Limsup F f* $\leq$ *Limsup F g*
⟨*proof*⟩

lemma *Limsup-eq*:
assumes *eventually* $(\lambda x.\ f\ x = g\ x)$ *net*
shows *Limsup net f* = *Limsup net g*
⟨*proof*⟩

lemma *Liminf-le-Limsup*:
assumes *ntriv*: ¬ *trivial-limit F*
shows *Liminf F f* $\leq$ *Limsup F f*
⟨*proof*⟩

lemma *Liminf-bounded*:
assumes *ntriv*: ¬ *trivial-limit F*
assumes *le*: *eventually* $(\lambda n.\ C \leq X\ n)\ F$
shows $C \leq$ *Liminf F X*
⟨*proof*⟩

lemma *Limsup-bounded*:
assumes *ntriv*: ¬ *trivial-limit F*
assumes *le*: *eventually* $(\lambda n.\ X\ n \leq C)\ F$
shows *Limsup F X* $\leq C$
⟨*proof*⟩

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*⟩

lemma *le-Liminf-iff*:
fixes *X* :: - ⇒ - :: *complete-linorder*
shows $C \leq$ *Liminf F X* $\longleftrightarrow$ $(\forall\, y{<}C.$ *eventually* $(\lambda x.\ y < X\ x)\ F)$
⟨*proof*⟩

lemma *Limsup-le-iff*:
fixes *X* :: - ⇒ - :: *complete-linorder*
shows $C \geq$ *Limsup F X* $\longleftrightarrow$ $(\forall\, y{>}C.$ *eventually* $(\lambda x.\ y > X\ x)\ F)$
⟨*proof*⟩

lemma *less-LiminfD*:
$y <$ *Liminf F* $(f$ :: - ⇒ *'a* :: *complete-linorder*$) \Longrightarrow$ *eventually* $(\lambda x.\ f\ x > y)\ F$
⟨*proof*⟩

lemma *Limsup-lessD*:
$y >$ *Limsup F* $(f$ :: - ⇒ *'a* :: *complete-linorder*$) \Longrightarrow$ *eventually* $(\lambda x.\ f\ x < y)\ F$
⟨*proof*⟩

lemma *lim-imp-Liminf*:
fixes $f :: 'a \Rightarrow$ - :: {*complete-linorder*,*linorder-topology*}
assumes *ntriv*: $\neg$ *trivial-limit F*
assumes *lim*: $(f \longrightarrow f0)\ F$
shows *Liminf F f* $=$ *f0*
⟨*proof*⟩

lemma *lim-imp-Limsup*:
fixes $f :: 'a \Rightarrow$ - :: {*complete-linorder*,*linorder-topology*}
assumes *ntriv*: $\neg$ *trivial-limit F*
assumes *lim*: $(f \longrightarrow f0)\ F$
shows *Limsup F f* $=$ *f0*
⟨*proof*⟩

lemma *Liminf-eq-Limsup*:
fixes *f0* :: $'a$ :: {*complete-linorder*,*linorder-topology*}
assumes *ntriv*: $\neg$ *trivial-limit F*
and *lim*: *Liminf F f* $=$ *f0 Limsup F f* $=$ *f0*
shows $(f \longrightarrow f0)\ F$
⟨*proof*⟩

lemma *tendsto-iff-Liminf-eq-Limsup*:
fixes *f0* :: $'a$ :: {*complete-linorder*,*linorder-topology*}
shows $\neg$ *trivial-limit F* $\Longrightarrow$ $(f \longrightarrow f0)\ F \longleftrightarrow$ (*Liminf F f* $=$ *f0* $\wedge$ *Limsup F f* $=$ *f0*)
⟨*proof*⟩

lemma *liminf-subseq-mono*:
fixes *X* :: *nat* $\Rightarrow$ $'a$ :: *complete-linorder*
assumes *subseq r*
shows *liminf X* $\leq$ *liminf* $(X \circ r)$
⟨*proof*⟩

lemma *limsup-subseq-mono*:
fixes *X* :: *nat* $\Rightarrow$ $'a$ :: *complete-linorder*
assumes *subseq r*
shows *limsup* $(X \circ r) \leq$ *limsup X*
⟨*proof*⟩

lemma *continuous-on-imp-continuous-within*:
continuous-on s f $\Longrightarrow$ $t \subseteq s \Longrightarrow x \in s \Longrightarrow$ *continuous* (*at x within t*) *f*
⟨*proof*⟩

lemma *Liminf-compose-continuous-mono*:
fixes *f* :: $'a$::{*complete-linorder*, *linorder-topology*} $\Rightarrow$ $'b$::{*complete-linorder*, *linorder-topology*}
assumes *c*: *continuous-on UNIV f* **and** *am*: *mono f* **and** *F*: *F* $\neq$ *bot*
shows *Liminf F* $(\lambda n.\ f\ (g\ n)) = f\ (Liminf\ F\ g)$

⟨*proof*⟩

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*: $F \neq bot$
shows *Limsup F* (λ*n*. *f* (*g n*)) = *f* (*Limsup F g*)
⟨*proof*⟩

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*: $F \neq bot$
shows *Liminf F* (λ*n*. *f* (*g n*)) = *f* (*Limsup F g*)
⟨*proof*⟩

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*: $F \neq bot$
shows *Limsup F* (λ*n*. *f* (*g n*)) = *f* (*Liminf F g*)
⟨*proof*⟩

32.1 More Limits

lemma *convergent-limsup-cl*:
fixes *X* :: *nat* ⇒ $'a$::{*complete-linorder*,*linorder-topology*}
shows *convergent X* ⟹ *limsup X* = *lim X*
⟨*proof*⟩

lemma *convergent-liminf-cl*:
fixes *X* :: *nat* ⇒ $'a$::{*complete-linorder*,*linorder-topology*}
shows *convergent X* ⟹ *liminf X* = *lim X*
⟨*proof*⟩

lemma *lim-increasing-cl*:
assumes ⋀*n m*. *n* ≥ *m* ⟹ *f n* ≥ *f m*
obtains *l* **where** *f* ⟶ (*l*::$'a$::{*complete-linorder*,*linorder-topology*})
⟨*proof*⟩

lemma *lim-decreasing-cl*:
assumes ⋀*n m*. *n* ≥ *m* ⟹ *f n* ≤ *f m*
obtains *l* **where** *f* ⟶ (*l*::$'a$::{*complete-linorder*,*linorder-topology*})
⟨*proof*⟩

lemma *compact-complete-linorder*:
fixes *X* :: *nat* ⇒ $'a$::{*complete-linorder*,*linorder-topology*}
shows ∃ *l r*. *subseq r* ∧ (*X* ∘ *r*) ⟶ *l*

⟨*proof*⟩

lemma *tendsto-Limsup*:
fixes *f* :: - ⇒ *'a* :: {*complete-linorder*,*linorder-topology*}
shows *F* ≠ *bot* ⟹ *Limsup F f* = *Liminf F f* ⟹ (*f* ⟶ *Limsup F f*) *F*
⟨*proof*⟩

lemma *tendsto-Liminf*:
fixes *f* :: - ⇒ *'a* :: {*complete-linorder*,*linorder-topology*}
shows *F* ≠ *bot* ⟹ *Limsup F f* = *Liminf F f* ⟹ (*f* ⟶ *Liminf F f*) *F*
⟨*proof*⟩

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* ⇒ *nat* ⇒ *'a*::*ordered-comm-monoid-add*
shows (⋀*n*. *n* ∈ *A* ⟹ *mono* (*f n*)) ⟹ *mono* (λ*i*. ∑ *n*∈*A*. *f n i*)
⟨*proof*⟩

lemma *incseq-setsumI*:
fixes *f* :: *nat* ⇒ *'a*::*ordered-comm-monoid-add*
assumes ⋀*i*. *0* ≤ *f i*
shows *incseq* (λ*i*. *setsum f* {..< *i*})
⟨*proof*⟩

lemma *continuous-at-left-imp-sup-continuous*:
fixes *f* :: *'a*::{*complete-linorder*, *linorder-topology*} ⇒ *'b*::{*complete-linorder*, *linorder-topology*}
assumes *mono f* ⋀*x*. *continuous* (*at-left x*) *f*
shows *sup-continuous f*
⟨*proof*⟩

lemma *sup-continuous-at-left*:
fixes *f* :: *'a*::{*complete-linorder*, *linorder-topology*, *first-countable-topology*} ⇒ *'b*::{*complete-linorder*, *linorder-topology*}
assumes *f*: *sup-continuous f*
shows *continuous* (*at-left x*) *f*
⟨*proof*⟩

lemma *sup-continuous-iff-at-left*:

fixes f :: $'a$::{*complete-linorder*, *linorder-topology*, *first-countable-topology*} ⇒ $'b$::{*complete-linorder*, *linorder-topology*}
shows *sup-continuous* $f \longleftrightarrow (\forall x.\ \textit{continuous}\ (\textit{at-left}\ x)\ f) \wedge \textit{mono}\ f$
⟨*proof*⟩

lemma *continuous-at-right-imp-inf-continuous*:
fixes f :: $'a$::{*complete-linorder*, *linorder-topology*} ⇒ $'b$::{*complete-linorder*, *linorder-topology*}
assumes *mono* f $\bigwedge x.$ *continuous* (*at-right* x) f
shows *inf-continuous* f
⟨*proof*⟩

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*⟩

lemma *inf-continuous-iff-at-right*:
fixes f :: $'a$::{*complete-linorder*, *linorder-topology*, *first-countable-topology*} ⇒ $'b$::{*complete-linorder*, *linorder-topology*}
shows *inf-continuous* $f \longleftrightarrow (\forall x.\ \textit{continuous}\ (\textit{at-right}\ x)\ f) \wedge \textit{mono}\ f$
⟨*proof*⟩

instantiation *enat* :: *linorder-topology*
begin

definition *open-enat* :: *enat set* ⇒ *bool* **where**
open-enat = *generate-topology* (*range lessThan* ∪ *range greaterThan*)

instance
⟨*proof*⟩

end

lemma *open-enat*: *open* {*enat n*}
⟨*proof*⟩

lemma *open-enat-iff*:
fixes A :: *enat set*
shows *open* $A \longleftrightarrow (\infty \in A \longrightarrow (\exists n$::*nat*. $\{n <..\} \subseteq A))$
⟨*proof*⟩

lemma *nhds-enat*: *nhds* x = (*if* $x = \infty$ *then INF i. principal* {*enat i..*} *else principal* {x})
⟨*proof*⟩

instance *enat* :: *topological-comm-monoid-add*

⟨*proof*⟩

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 ⟨*proof*⟩

end

instantiation *ereal* :: *infinity*
begin

definition (∞::*ereal*) = *PInfty*
instance ⟨*proof*⟩

end

declare [[*coercion ereal* :: *real* ⇒ *ereal*]]

lemma *ereal-uminus-uminus*[*simp*]:
  fixes *a* :: *ereal*
  shows − (− *a*) = *a*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

declare

PInfty-eq-infinity[*code-post*]
MInfty-eq-minfinity[*code-post*]

lemma [*code-unfold*]:
$\infty = PInfty$
$- PInfty = MInfty$
⟨*proof*⟩

lemma *inj-ereal*[*simp*]: *inj-on ereal A*
⟨*proof*⟩

lemma *ereal-cases*[*cases type*: *ereal*]:
obtains (*real*) r **where** $x = ereal\ r$
| (*PInf*) $x = \infty$
| (*MInf*) $x = -\infty$
⟨*proof*⟩

lemmas *ereal2-cases* = *ereal-cases*[*case-product ereal-cases*]
lemmas *ereal3-cases* = *ereal2-cases*[*case-product ereal-cases*]

lemma *ereal-all-split*: $\bigwedge P.\ (\forall x{::}ereal.\ P\ x) \longleftrightarrow P\ \infty \wedge (\forall x.\ P\ (ereal\ x)) \wedge P\ (-\infty)$
⟨*proof*⟩

lemma *ereal-ex-split*: $\bigwedge P.\ (\exists x{::}ereal.\ P\ x) \longleftrightarrow P\ \infty \vee (\exists x.\ P\ (ereal\ x)) \vee P\ (-\infty)$
⟨*proof*⟩

lemma *ereal-uminus-eq-iff*[*simp*]:
fixes $a\ b :: ereal$
shows $-a = -b \longleftrightarrow a = b$
⟨*proof*⟩

function *real-of-ereal* :: *ereal* ⇒ *real* **where**
real-of-ereal (*ereal r*) = *r*
| *real-of-ereal* ∞ = *0*
| *real-of-ereal* $(-\infty)$ = *0*
⟨*proof*⟩
termination ⟨*proof*⟩

lemma *real-of-ereal*[*simp*]:
real-of-ereal $(-\ x :: ereal) = -$ (*real-of-ereal x*)
⟨*proof*⟩

lemma *range-ereal*[*simp*]: *range ereal* = *UNIV* $-$ $\{\infty, -\infty\}$
⟨*proof*⟩

lemma *ereal-range-uminus*[*simp*]: *range uminus* = (*UNIV*::*ereal set*)
⟨*proof*⟩

instantiation *ereal* :: *abs*
begin

function *abs-ereal* **where**
  $|ereal\ r| = ereal\ |r|$
| $|-\infty| = (\infty::ereal)$
| $|\infty| = (\infty::ereal)$
⟨*proof*⟩
termination ⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *abs-eq-infinity-cases*[*elim*!]:
  fixes x :: *ereal*
  assumes $|x| = \infty$
  obtains $x = \infty \mid x = -\infty$
  ⟨*proof*⟩

lemma *abs-neq-infinity-cases*[*elim*!]:
  fixes x :: *ereal*
  assumes $|x| \neq \infty$
  obtains r **where** $x = ereal\ r$
  ⟨*proof*⟩

lemma *abs-ereal-uminus*[*simp*]:
  fixes x :: *ereal*
  shows $|-x| = |x|$
  ⟨*proof*⟩

lemma *ereal-infinity-cases*:
  fixes a :: *ereal*
  shows $a \neq \infty \Longrightarrow a \neq -\infty \Longrightarrow |a| \neq \infty$
  ⟨*proof*⟩

33.1.1 Addition

instantiation *ereal* :: {*one*,*comm-monoid-add*,*zero-neq-one*}
begin

definition *0* = *ereal 0*
definition *1* = *ereal 1*

function *plus-ereal* **where**
  $ereal\ r + ereal\ p = ereal\ (r + p)$
| $\infty + a = (\infty::ereal)$
| $a + \infty = (\infty::ereal)$

| *ereal r* + −∞ = − ∞
| −∞ + *ereal p* = −(∞::*ereal*)
| −∞ + −∞ = −(∞::*ereal*)
⟨*proof*⟩
termination ⟨*proof*⟩

lemma *Infty-neq-0*[*simp*]:
(∞::*ereal*) ≠ *0* *0* ≠ (∞::*ereal*)
−(∞::*ereal*) ≠ *0* *0* ≠ −(∞::*ereal*)
⟨*proof*⟩

lemma *ereal-eq-0*[*simp*]:
ereal r = *0* ⟷ *r* = *0*
0 = *ereal r* ⟷ *r* = *0*
⟨*proof*⟩

lemma *ereal-eq-1*[*simp*]:
ereal r = *1* ⟷ *r* = *1*
1 = *ereal r* ⟷ *r* = *1*
⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *ereal-0-plus* [*simp*]: *ereal 0* + *x* = *x*
and *plus-ereal-0* [*simp*]: *x* + *ereal 0* = *x*
⟨*proof*⟩

instance *ereal* :: *numeral* ⟨*proof*⟩

lemma *real-of-ereal-0*[*simp*]: *real-of-ereal* (*0*::*ereal*) = *0*
⟨*proof*⟩

lemma *abs-ereal-zero*[*simp*]: |*0*| = (*0*::*ereal*)
⟨*proof*⟩

lemma *ereal-uminus-zero*[*simp*]: − *0* = (*0*::*ereal*)
⟨*proof*⟩

lemma *ereal-uminus-zero-iff*[*simp*]:
fixes *a* :: *ereal*
shows −*a* = *0* ⟷ *a* = *0*
⟨*proof*⟩

lemma *ereal-plus-eq-PInfty*[*simp*]:
fixes *a b* :: *ereal*
shows *a* + *b* = ∞ ⟷ *a* = ∞ ∨ *b* = ∞

⟨*proof*⟩

lemma *ereal-plus-eq-MInfty*[*simp*]:
fixes *a b* :: *ereal*
shows $a + b = -\infty \longleftrightarrow (a = -\infty \vee b = -\infty) \wedge a \neq \infty \wedge b \neq \infty$
⟨*proof*⟩

lemma *ereal-add-cancel-left*:
fixes *a b* :: *ereal*
assumes $a \neq -\infty$
shows $a + b = a + c \longleftrightarrow a = \infty \vee b = c$
⟨*proof*⟩

lemma *ereal-add-cancel-right*:
fixes *a b* :: *ereal*
assumes $a \neq -\infty$
shows $b + a = c + a \longleftrightarrow a = \infty \vee b = c$
⟨*proof*⟩

lemma *ereal-real*: *ereal* (*real-of-ereal x*) = (*if* $|x| = \infty$ *then 0 else x*)
⟨*proof*⟩

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*)
⟨*proof*⟩

33.1.2 Linear order on *ereal*

instantiation *ereal* :: *linorder*
begin

function *less-ereal*
where
ereal x < *ereal y* ⟷ *x* < *y*
| (∞::*ereal*) < *a* ⟷ *False*
| *a* < −(∞::*ereal*) ⟷ *False*
| *ereal x* < ∞ ⟷ *True*
| −∞ < *ereal r* ⟷ *True*
| −∞ < (∞::*ereal*) ⟷ *True*
⟨*proof*⟩
termination ⟨*proof*⟩

definition $x \leq (y::ereal) \longleftrightarrow x < y \vee x = y$

lemma *ereal-infty-less*[*simp*]:
fixes *x* :: *ereal*

shows $x < \infty \longleftrightarrow (x \neq \infty)$
$-\infty < x \longleftrightarrow (x \neq -\infty)$
⟨*proof*⟩

lemma *ereal-infty-less-eq*[*simp*]:
fixes x :: *ereal*
shows $\infty \leq x \longleftrightarrow x = \infty$
and $x \leq -\infty \longleftrightarrow x = -\infty$
⟨*proof*⟩

lemma *ereal-less*[*simp*]:
$ereal\ r < 0 \longleftrightarrow (r < 0)$
$0 < ereal\ r \longleftrightarrow (0 < r)$
$ereal\ r < 1 \longleftrightarrow (r < 1)$
$1 < ereal\ r \longleftrightarrow (1 < r)$
$0 < (\infty::ereal)$
$-(\infty::ereal) < 0$
⟨*proof*⟩

lemma *ereal-less-eq*[*simp*]:
$x \leq (\infty::ereal)$
$-(\infty::ereal) \leq x$
$ereal\ r \leq ereal\ p \longleftrightarrow r \leq p$
$ereal\ r \leq 0 \longleftrightarrow r \leq 0$
$0 \leq ereal\ r \longleftrightarrow 0 \leq r$
$ereal\ r \leq 1 \longleftrightarrow r \leq 1$
$1 \leq ereal\ r \longleftrightarrow 1 \leq r$
⟨*proof*⟩

lemma *ereal-infty-less-eq2*:
$a \leq b \Longrightarrow a = \infty \Longrightarrow b = (\infty::ereal)$
$a \leq b \Longrightarrow b = -\infty \Longrightarrow a = -(\infty::ereal)$
⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *ereal-dense2*: $x < y \Longrightarrow \exists z.\ x < ereal\ z \wedge ereal\ z < y$
⟨*proof*⟩

instance *ereal* :: *dense-linorder*
⟨*proof*⟩

instance *ereal* :: *ordered-comm-monoid-add*
⟨*proof*⟩

lemma *ereal-one-not-less-zero-ereal*[*simp*]: $\neg\ 1 < (0::ereal)$

⟨*proof*⟩

lemma *real-of-ereal-positive-mono*:
fixes $x\ y$ :: *ereal*
shows $0 \le x \Longrightarrow x \le y \Longrightarrow y \neq \infty \Longrightarrow$ *real-of-ereal* $x \le$ *real-of-ereal* y
⟨*proof*⟩

lemma *ereal-MInfty-lessI*[*intro*, *simp*]:
fixes a :: *ereal*
shows $a \neq -\infty \Longrightarrow -\infty < a$
⟨*proof*⟩

lemma *ereal-less-PInfty*[*intro*, *simp*]:
fixes a :: *ereal*
shows $a \neq \infty \Longrightarrow a < \infty$
⟨*proof*⟩

lemma *ereal-less-ereal-Ex*:
fixes $a\ b$ :: *ereal*
shows $x <$ *ereal* $r \longleftrightarrow x = -\infty \vee (\exists\, p.\ p < r \wedge x =$ *ereal* $p)$
⟨*proof*⟩

lemma *less-PInf-Ex-of-nat*: $x \neq \infty \longleftrightarrow (\exists\, n{::}nat.\ x <$ *ereal* $(real\ n))$
⟨*proof*⟩

lemma *ereal-add-mono*:
fixes $a\ b\ c\ d$ :: *ereal*
assumes $a \le b$
and $c \le d$
shows $a + c \le b + d$
⟨*proof*⟩

lemma *ereal-minus-le-minus*[*simp*]:
fixes $a\ b$ :: *ereal*
shows $-a \le -b \longleftrightarrow b \le a$
⟨*proof*⟩

lemma *ereal-minus-less-minus*[*simp*]:
fixes $a\ b$ :: *ereal*
shows $-a < -b \longleftrightarrow b < a$
⟨*proof*⟩

lemma *ereal-le-real-iff*:
$x \le$ *real-of-ereal* $y \longleftrightarrow (|y| \neq \infty \longrightarrow$ *ereal* $x \le y) \wedge (|y| = \infty \longrightarrow x \le 0)$
⟨*proof*⟩

lemma *real-le-ereal-iff*:
real-of-ereal $y \le x \longleftrightarrow (|y| \neq \infty \longrightarrow y \le$ *ereal* $x) \wedge (|y| = \infty \longrightarrow 0 \le x)$
⟨*proof*⟩

lemma *ereal-less-real-iff*:
$x < \textit{real-of-ereal}\ y \longleftrightarrow (|y| \neq \infty \longrightarrow \textit{ereal}\ x < y) \wedge (|y| = \infty \longrightarrow x < 0)$
⟨*proof*⟩

lemma *real-less-ereal-iff*:
$\textit{real-of-ereal}\ y < x \longleftrightarrow (|y| \neq \infty \longrightarrow y < \textit{ereal}\ x) \wedge (|y| = \infty \longrightarrow 0 < x)$
⟨*proof*⟩

lemma *real-of-ereal-pos*:
fixes $x :: \textit{ereal}$
shows $0 \leq x \Longrightarrow 0 \leq \textit{real-of-ereal}\ x$ ⟨*proof*⟩

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*]: $0 \leq x \Longrightarrow |x :: \textit{ereal}| = x$
⟨*proof*⟩

lemma *abs-ereal-less0*[*simp*]: $x < 0 \Longrightarrow |x :: \textit{ereal}| = -x$
⟨*proof*⟩

lemma *abs-ereal-pos*[*simp*]: $0 \leq |x :: \textit{ereal}|$
⟨*proof*⟩

lemma *ereal-abs-leI*:
fixes $x\ y :: \textit{ereal}$
shows $[\![\ x \leq y; -x \leq y\ ]\!] \Longrightarrow |x| \leq y$
⟨*proof*⟩

lemma *real-of-ereal-le-0*[*simp*]: $\textit{real-of-ereal}\ (x :: \textit{ereal}) \leq 0 \longleftrightarrow x \leq 0 \vee x = \infty$
⟨*proof*⟩

lemma *abs-real-of-ereal*[*simp*]: $|\textit{real-of-ereal}\ (x :: \textit{ereal})| = \textit{real-of-ereal}\ |x|$
⟨*proof*⟩

lemma *zero-less-real-of-ereal*:
fixes $x :: \textit{ereal}$
shows $0 < \textit{real-of-ereal}\ x \longleftrightarrow 0 < x \wedge x \neq \infty$
⟨*proof*⟩

lemma *ereal-0-le-uminus-iff*[*simp*]:
fixes $a :: \textit{ereal}$
shows $0 \leq -\ a \longleftrightarrow a \leq 0$
⟨*proof*⟩

lemma *ereal-uminus-le-0-iff*[*simp*]:
fixes $a :: \textit{ereal}$
shows $-\ a \leq 0 \longleftrightarrow 0 \leq a$

⟨*proof*⟩

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 $a + c < b + d$
⟨*proof*⟩

lemma *ereal-less-add*:
fixes $a\ b\ c$ :: *ereal*
shows $|a| \neq \infty \Longrightarrow c < b \Longrightarrow a + c < a + b$
⟨*proof*⟩

lemma *ereal-add-nonneg-eq-0-iff*:
fixes $a\ b$ :: *ereal*
shows $0 \leq a \Longrightarrow 0 \leq b \Longrightarrow a + b = 0 \longleftrightarrow a = 0 \wedge b = 0$
⟨*proof*⟩

lemma *ereal-uminus-eq-reorder*: $-\ a = b \longleftrightarrow a = (-b{::}ereal)$
⟨*proof*⟩

lemma *ereal-uminus-less-reorder*: $-\ a < b \longleftrightarrow -b < (a{::}ereal)$
⟨*proof*⟩

lemma *ereal-less-uminus-reorder*: $a < -\ b \longleftrightarrow b < -\ (a{::}ereal)$
⟨*proof*⟩

lemma *ereal-uminus-le-reorder*: $-\ a \leq b \longleftrightarrow -b \leq (a{::}ereal)$
⟨*proof*⟩

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*⟩

lemma *ereal-top*:
fixes x :: *ereal*
assumes $\bigwedge B.\ x \geq ereal\ B$
shows $x = \infty$
⟨*proof*⟩

lemma

shows *ereal-max*[*simp*]: $ereal\ (max\ x\ y) = max\ (ereal\ x)\ (ereal\ y)$
and *ereal-min*[*simp*]: $ereal\ (min\ x\ y) = min\ (ereal\ x)\ (ereal\ y)$
⟨*proof*⟩

lemma *ereal-max-0*: $max\ 0\ (ereal\ r) = ereal\ (max\ 0\ r)$
⟨*proof*⟩

lemma
fixes $f :: nat \Rightarrow ereal$
shows *ereal-incseq-uminus*[*simp*]: $incseq\ (\lambda x.\ -f\ x) \longleftrightarrow decseq\ f$
and *ereal-decseq-uminus*[*simp*]: $decseq\ (\lambda x.\ -f\ x) \longleftrightarrow incseq\ f$
⟨*proof*⟩

lemma *incseq-ereal*: $incseq\ f \Longrightarrow incseq\ (\lambda x.\ ereal\ (f\ x))$
⟨*proof*⟩

lemma *ereal-add-nonneg-nonneg*[*simp*]:
fixes $a\ b :: ereal$
shows $0 \le a \Longrightarrow 0 \le b \Longrightarrow 0 \le a + b$
⟨*proof*⟩

lemma *setsum-ereal*[*simp*]: $(\sum x{\in}A.\ ereal\ (f\ x)) = ereal\ (\sum x{\in}A.\ f\ x)$
⟨*proof*⟩

lemma *setsum-Pinfty*:
fixes $f :: 'a \Rightarrow ereal$
shows $(\sum x{\in}P.\ f\ x) = \infty \longleftrightarrow finite\ P \wedge (\exists\, i{\in}P.\ f\ i = \infty)$
⟨*proof*⟩

lemma *setsum-Inf*:
fixes $f :: 'a \Rightarrow ereal$
shows $|setsum\ f\ A| = \infty \longleftrightarrow finite\ A \wedge (\exists\, i{\in}A.\ |f\ i| = \infty)$
⟨*proof*⟩

lemma *setsum-real-of-ereal*:
fixes $f :: 'i \Rightarrow ereal$
assumes $\bigwedge x.\ x \in S \Longrightarrow |f\ x| \neq \infty$
shows $(\sum x{\in}S.\ real\text{-}of\text{-}ereal\ (f\ x)) = real\text{-}of\text{-}ereal\ (setsum\ f\ S)$
⟨*proof*⟩

lemma *setsum-ereal-0*:
fixes $f :: 'a \Rightarrow ereal$
assumes *finite A*
and $\bigwedge i.\ i \in A \Longrightarrow 0 \le f\ i$
shows $(\sum x{\in}A.\ f\ x) = 0 \longleftrightarrow (\forall\, i{\in}A.\ f\ i = 0)$
⟨*proof*⟩

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*
⟨*proof*⟩
termination ⟨*proof*⟩

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*⟩
termination ⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma [*simp*]:
  shows *ereal-1-times*: *ereal 1* ∗ *x* = *x*
  and *times-ereal-1*: *x* ∗ *ereal 1* = *x*
⟨*proof*⟩

lemma *one-not-le-zero-ereal*[*simp*]: ¬ (*1* ≤ (*0*::*ereal*))
  ⟨*proof*⟩

lemma *real-ereal-1*[*simp*]: *real-of-ereal* (*1*::*ereal*) = *1*
  ⟨*proof*⟩

lemma *real-of-ereal-le-1*:
  fixes *a* :: *ereal*
  shows *a* ≤ *1* ⟹ *real-of-ereal a* ≤ *1*
  ⟨*proof*⟩

lemma *abs-ereal-one*[*simp*]: |*1*| = (*1*::*ereal*)
  ⟨*proof*⟩

lemma *ereal-mult-zero*[*simp*]:

fixes a :: *ereal*
shows $a * 0 = 0$
⟨*proof*⟩

lemma *ereal-zero-mult*[*simp*]:
fixes a :: *ereal*
shows $0 * a = 0$
⟨*proof*⟩

lemma *ereal-m1-less-0*[*simp*]: $-(1::ereal) < 0$
⟨*proof*⟩

lemma *ereal-times*[*simp*]:
$1 \neq (\infty::ereal)$ $(\infty::ereal) \neq 1$
$1 \neq -(\infty::ereal)$ $-(\infty::ereal) \neq 1$
⟨*proof*⟩

lemma *ereal-plus-1*[*simp*]:
$1 + ereal\ r = ereal\ (r + 1)$
$ereal\ r + 1 = ereal\ (r + 1)$
$1 + -(\infty::ereal) = -\infty$
$-(\infty::ereal) + 1 = -\infty$
⟨*proof*⟩

lemma *ereal-zero-times*[*simp*]:
fixes $a\ b$ :: *ereal*
shows $a * b = 0 \longleftrightarrow a = 0 \vee b = 0$
⟨*proof*⟩

lemma *ereal-mult-eq-PInfty*[*simp*]:
$a * b = (\infty::ereal) \longleftrightarrow$
$(a = \infty \wedge b > 0) \vee (a > 0 \wedge b = \infty) \vee (a = -\infty \wedge b < 0) \vee (a < 0 \wedge b = -\infty)$
⟨*proof*⟩

lemma *ereal-mult-eq-MInfty*[*simp*]:
$a * b = -(\infty::ereal) \longleftrightarrow$
$(a = \infty \wedge b < 0) \vee (a < 0 \wedge b = \infty) \vee (a = -\infty \wedge b > 0) \vee (a > 0 \wedge b = -\infty)$
⟨*proof*⟩

lemma *ereal-abs-mult*: $|x * y :: ereal| = |x| * |y|$
⟨*proof*⟩

lemma *ereal-0-less-1*[*simp*]: $0 < (1::ereal)$
⟨*proof*⟩

lemma *ereal-mult-minus-left*[*simp*]:
fixes $a\ b$ :: *ereal*

shows $-a * b = -(a * b)$
⟨*proof*⟩

lemma *ereal-mult-minus-right*[*simp*]:
fixes $a\ b :: ereal$
shows $a * -b = -(a * b)$
⟨*proof*⟩

lemma *ereal-mult-infty*[*simp*]:
$a * (\infty::ereal) = (if\ a = 0\ then\ 0\ else\ if\ 0 < a\ then\ \infty\ else\ -\infty)$
⟨*proof*⟩

lemma *ereal-infty-mult*[*simp*]:
$(\infty::ereal) * a = (if\ a = 0\ then\ 0\ else\ if\ 0 < a\ then\ \infty\ else\ -\infty)$
⟨*proof*⟩

lemma *ereal-mult-strict-right-mono*:
assumes $a < b$
and $0 < c$
and $c < (\infty::ereal)$
shows $a * c < b * c$
⟨*proof*⟩

lemma *ereal-mult-strict-left-mono*:
$a < b \Longrightarrow 0 < c \Longrightarrow c < (\infty::ereal) \Longrightarrow c * a < c * b$
⟨*proof*⟩

lemma *ereal-mult-right-mono*:
fixes $a\ b\ c :: ereal$
shows $a \leq b \Longrightarrow 0 \leq c \Longrightarrow a * c \leq b * c$
⟨*proof*⟩

lemma *ereal-mult-left-mono*:
fixes $a\ b\ c :: ereal$
shows $a \leq b \Longrightarrow 0 \leq c \Longrightarrow c * a \leq c * b$
⟨*proof*⟩

lemma *zero-less-one-ereal*[*simp*]: $0 \leq (1::ereal)$
⟨*proof*⟩

lemma *ereal-0-le-mult*[*simp*]: $0 \leq a \Longrightarrow 0 \leq b \Longrightarrow 0 \leq a * (b :: ereal)$
⟨*proof*⟩

lemma *ereal-right-distrib*:
fixes $r\ a\ b :: ereal$
shows $0 \leq a \Longrightarrow 0 \leq b \Longrightarrow r * (a + b) = r * a + r * b$
⟨*proof*⟩

lemma *ereal-left-distrib*:

fixes r a b :: *ereal*
shows $0 \leq a \Longrightarrow 0 \leq b \Longrightarrow (a + b) * r = a * r + b * r$
⟨*proof*⟩

lemma *ereal-mult-le-0-iff*:
fixes a b :: *ereal*
shows $a * b \leq 0 \longleftrightarrow (0 \leq a \wedge b \leq 0) \vee (a \leq 0 \wedge 0 \leq b)$
⟨*proof*⟩

lemma *ereal-zero-le-0-iff*:
fixes a b :: *ereal*
shows $0 \leq a * b \longleftrightarrow (0 \leq a \wedge 0 \leq b) \vee (a \leq 0 \wedge b \leq 0)$
⟨*proof*⟩

lemma *ereal-mult-less-0-iff*:
fixes a b :: *ereal*
shows $a * b < 0 \longleftrightarrow (0 < a \wedge b < 0) \vee (a < 0 \wedge 0 < b)$
⟨*proof*⟩

lemma *ereal-zero-less-0-iff*:
fixes a b :: *ereal*
shows $0 < a * b \longleftrightarrow (0 < a \wedge 0 < b) \vee (a < 0 \wedge b < 0)$
⟨*proof*⟩

lemma *ereal-left-mult-cong*:
fixes a b c :: *ereal*
shows $c = d \Longrightarrow (d \neq 0 \Longrightarrow a = b) \Longrightarrow a * c = b * d$
⟨*proof*⟩

lemma *ereal-right-mult-cong*:
fixes a b c :: *ereal*
shows $c = d \Longrightarrow (d \neq 0 \Longrightarrow a = b) \Longrightarrow c * a = d * b$
⟨*proof*⟩

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 $(a + b) * c = a * c + b * c$
⟨*proof*⟩

lemma *numeral-eq-ereal* [*simp*]: *numeral* w = *ereal* (*numeral* w)
⟨*proof*⟩

lemma *distrib-left-ereal-nn*:
$c \geq 0 \Longrightarrow (x + y) * ereal\ c = x * ereal\ c + y * ereal\ c$
⟨*proof*⟩

lemma *setsum-ereal-right-distrib*:
fixes $f :: 'a \Rightarrow ereal$
shows $(\bigwedge i.\ i \in A \Longrightarrow 0 \le f\ i) \Longrightarrow r * setsum\ f\ A = (\sum n{\in}A.\ r * f\ n)$
⟨*proof*⟩

lemma *setsum-ereal-left-distrib*:
$(\bigwedge i.\ i \in A \Longrightarrow 0 \le f\ i) \Longrightarrow setsum\ f\ A * r = (\sum n{\in}A.\ f\ n * r :: ereal)$
⟨*proof*⟩

lemma *setsum-left-distrib-ereal*:
$c \ge 0 \Longrightarrow setsum\ f\ A * ereal\ c = (\sum x{\in}A.\ f\ x * c :: ereal)$
⟨*proof*⟩

lemma *ereal-le-epsilon*:
fixes $x\ y :: ereal$
assumes $\forall e.\ 0 < e \longrightarrow x \le y + e$
shows $x \le y$
⟨*proof*⟩

lemma *ereal-le-epsilon2*:
fixes $x\ y :: ereal$
assumes $\forall e.\ 0 < e \longrightarrow x \le y + ereal\ e$
shows $x \le y$
⟨*proof*⟩

lemma *ereal-le-real*:
fixes $x\ y :: ereal$
assumes $\forall z.\ x \le ereal\ z \longrightarrow y \le ereal\ z$
shows $y \le x$
⟨*proof*⟩

lemma *setprod-ereal-0*:
fixes $f :: 'a \Rightarrow ereal$
shows $(\prod i{\in}A.\ f\ i) = 0 \longleftrightarrow finite\ A \wedge (\exists i{\in}A.\ f\ i = 0)$
⟨*proof*⟩

lemma *setprod-ereal-pos*:
fixes $f :: 'a \Rightarrow ereal$
assumes *pos*: $\bigwedge i.\ i \in I \Longrightarrow 0 \le f\ i$
shows $0 \le (\prod i{\in}I.\ f\ i)$
⟨*proof*⟩

lemma *setprod-PInf*:
fixes $f :: 'a \Rightarrow ereal$
assumes $\bigwedge i.\ i \in I \Longrightarrow 0 \le f\ i$
shows $(\prod i{\in}I.\ f\ i) = \infty \longleftrightarrow finite\ I \wedge (\exists i{\in}I.\ f\ i = \infty) \wedge (\forall i{\in}I.\ f\ i \neq 0)$
⟨*proof*⟩

lemma *setprod-ereal*: $(\prod i{\in}A.\ ereal\ (f\ i)) = ereal\ (setprod\ f\ A)$

⟨*proof*⟩

33.1.4 Power

lemma *ereal-power*[*simp*]: (*ereal x*) ^ *n* = *ereal* (*x*^*n*)
⟨*proof*⟩

lemma *ereal-power-PInf*[*simp*]: (∞::*ereal*) ^ *n* = (*if n* = *0 then 1 else* ∞)
⟨*proof*⟩

lemma *ereal-power-uminus*[*simp*]:
fixes *x* :: *ereal*
shows (− *x*) ^ *n* = (*if even n then x* ^ *n else* − (*x*^*n*))
⟨*proof*⟩

lemma *ereal-power-numeral*[*simp*]:
(*numeral num* :: *ereal*) ^ *n* = *ereal* (*numeral num* ^ *n*)
⟨*proof*⟩

lemma *zero-le-power-ereal*[*simp*]:
fixes *a* :: *ereal*
assumes *0* ≤ *a*
shows *0* ≤ *a* ^ *n*
⟨*proof*⟩

33.1.5 Subtraction

lemma *ereal-minus-minus-image*[*simp*]:
fixes *S* :: *ereal set*
shows *uminus* ‘ *uminus* ‘ *S* = *S*
⟨*proof*⟩

lemma *ereal-uminus-lessThan*[*simp*]:
fixes *a* :: *ereal*
shows *uminus* ‘ {..<*a*} = {−*a*<..}
⟨*proof*⟩

lemma *ereal-uminus-greaterThan*[*simp*]: *uminus* ‘ {(*a*::*ereal*)<..} = {..<−*a*}
⟨*proof*⟩

instantiation *ereal* :: *minus*
begin

definition *x* − *y* = *x* + −(*y*::*ereal*)
instance ⟨*proof*⟩

end

lemma *ereal-minus*[*simp*]:
ereal r − *ereal p* = *ereal* (*r* − *p*)

$-\infty - \mathit{ereal}\ r = -\infty$
$\mathit{ereal}\ r - \infty = -\infty$
$(\infty::\mathit{ereal}) - x = \infty$
$-(\infty::\mathit{ereal}) - \infty = -\infty$
$x - -y = x + y$
$x - 0 = x$
$0 - x = -x$
⟨*proof*⟩

lemma *ereal-x-minus-x*[*simp*]: $x - x = (\mathit{if}\ |x| = \infty\ \mathit{then}\ \infty\ \mathit{else}\ 0::\mathit{ereal})$
⟨*proof*⟩

lemma *ereal-eq-minus-iff*:
fixes $x\ y\ z :: \mathit{ereal}$
shows $x = z - y \longleftrightarrow$
$(|y| \neq \infty \longrightarrow x + y = z) \wedge$
$(y = -\infty \longrightarrow x = \infty) \wedge$
$(y = \infty \longrightarrow z = \infty \longrightarrow x = \infty) \wedge$
$(y = \infty \longrightarrow z \neq \infty \longrightarrow x = -\infty)$
⟨*proof*⟩

lemma *ereal-eq-minus*:
fixes $x\ y\ z :: \mathit{ereal}$
shows $|y| \neq \infty \Longrightarrow x = z - y \longleftrightarrow x + y = z$
⟨*proof*⟩

lemma *ereal-less-minus-iff*:
fixes $x\ y\ z :: \mathit{ereal}$
shows $x < z - y \longleftrightarrow$
$(y = \infty \longrightarrow z = \infty \wedge x \neq \infty) \wedge$
$(y = -\infty \longrightarrow x \neq \infty) \wedge$
$(|y| \neq \infty \longrightarrow x + y < z)$
⟨*proof*⟩

lemma *ereal-less-minus*:
fixes $x\ y\ z :: \mathit{ereal}$
shows $|y| \neq \infty \Longrightarrow x < z - y \longleftrightarrow x + y < z$
⟨*proof*⟩

lemma *ereal-le-minus-iff*:
fixes $x\ y\ z :: \mathit{ereal}$
shows $x \leq z - y \longleftrightarrow (y = \infty \longrightarrow z \neq \infty \longrightarrow x = -\infty) \wedge (|y| \neq \infty \longrightarrow x + y \leq z)$
⟨*proof*⟩

lemma *ereal-le-minus*:
fixes $x\ y\ z :: \mathit{ereal}$
shows $|y| \neq \infty \Longrightarrow x \leq z - y \longleftrightarrow x + y \leq z$
⟨*proof*⟩

lemma *ereal-minus-less-iff*:
fixes $x\ y\ z$ :: *ereal*
shows $x - y < z \longleftrightarrow y \neq -\infty \wedge (y = \infty \longrightarrow x \neq \infty \wedge z \neq -\infty) \wedge (y \neq \infty \longrightarrow x < z + y)$
⟨*proof*⟩

lemma *ereal-minus-less*:
fixes $x\ y\ z$ :: *ereal*
shows $|y| \neq \infty \Longrightarrow x - y < z \longleftrightarrow x < z + y$
⟨*proof*⟩

lemma *ereal-minus-le-iff*:
fixes $x\ y\ z$ :: *ereal*
shows $x - y \leq z \longleftrightarrow$
$(y = -\infty \longrightarrow z = \infty) \wedge$
$(y = \infty \longrightarrow x = \infty \longrightarrow z = \infty) \wedge$
$(|y| \neq \infty \longrightarrow x \leq z + y)$
⟨*proof*⟩

lemma *ereal-minus-le*:
fixes $x\ y\ z$ :: *ereal*
shows $|y| \neq \infty \Longrightarrow x - y \leq z \longleftrightarrow x \leq z + y$
⟨*proof*⟩

lemma *ereal-minus-eq-minus-iff*:
fixes $a\ b\ c$ :: *ereal*
shows $a - b = a - c \longleftrightarrow$
$b = c \vee a = \infty \vee (a = -\infty \wedge b \neq -\infty \wedge c \neq -\infty)$
⟨*proof*⟩

lemma *ereal-add-le-add-iff*:
fixes $a\ b\ c$ :: *ereal*
shows $c + a \leq c + b \longleftrightarrow$
$a \leq b \vee c = \infty \vee (c = -\infty \wedge a \neq \infty \wedge b \neq \infty)$
⟨*proof*⟩

lemma *ereal-add-le-add-iff2*:
fixes $a\ b\ c$ :: *ereal*
shows $a + c \leq b + c \longleftrightarrow a \leq b \vee c = \infty \vee (c = -\infty \wedge a \neq \infty \wedge b \neq \infty)$
⟨*proof*⟩

lemma *ereal-mult-le-mult-iff*:
fixes $a\ b\ c$ :: *ereal*
shows $|c| \neq \infty \Longrightarrow c * a \leq c * b \longleftrightarrow (0 < c \longrightarrow a \leq b) \wedge (c < 0 \longrightarrow b \leq a)$
⟨*proof*⟩

lemma *ereal-minus-mono*:
fixes $A\ B\ C\ D$ :: *ereal* **assumes** $A \leq B\ D \leq C$

shows $A - C \leq B - D$
⟨*proof*⟩

lemma *ereal-mono-minus-cancel*:
fixes $a\ b\ c$ :: *ereal*
shows $c - a \leq c - b \Longrightarrow 0 \leq c \Longrightarrow c < \infty \Longrightarrow b \leq a$
⟨*proof*⟩

lemma *real-of-ereal-minus*:
fixes $a\ b$ :: *ereal*
shows *real-of-ereal* $(a - b) =$ (*if* $|a| = \infty \vee |b| = \infty$ *then 0 else real-of-ereal* a $-$ *real-of-ereal* b)
⟨*proof*⟩

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*$)$
⟨*proof*⟩

lemma *ereal-diff-positive*:
fixes $a\ b$ :: *ereal* **shows** $a \leq b \Longrightarrow 0 \leq b - a$
⟨*proof*⟩

lemma *ereal-between*:
fixes $x\ e$ :: *ereal*
assumes $|x| \neq \infty$
and $0 < e$
shows $x - e < x$
and $x < x + e$
⟨*proof*⟩

lemma *ereal-minus-eq-PInfty-iff*:
fixes $x\ y$ :: *ereal*
shows $x - y = \infty \longleftrightarrow y = -\infty \vee x = \infty$
⟨*proof*⟩

lemma *ereal-diff-add-eq-diff-diff-swap*:
fixes $x\ y\ z$ :: *ereal*
shows $|y| \neq \infty \Longrightarrow x - (y + z) = x - y - z$
⟨*proof*⟩

lemma *ereal-diff-add-assoc2*:
fixes $x\ y\ z$ :: *ereal*
shows $x + y - z = x - z + y$
⟨*proof*⟩

lemma *ereal-add-uminus-conv-diff*: **fixes** $x\ y\ z$ :: *ereal* **shows** $-x + y = y - x$
⟨*proof*⟩

lemma *ereal-minus-diff-eq*:

fixes $x\ y :: ereal$
shows $[\![x = \infty \longrightarrow y \neq \infty; x = -\infty \longrightarrow y \neq -\infty]\!] \Longrightarrow -(x - y) = y - x$
⟨*proof*⟩

lemma *ediff-le-self* [*simp*]: $x - y \leq (x :: enat)$
⟨*proof*⟩

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*
⟨*proof*⟩
termination ⟨*proof*⟩

definition *x div y* = *x* ∗ *inverse* (*y* :: *ereal*)

instance ⟨*proof*⟩

end

lemma *real-of-ereal-inverse*[*simp*]:
fixes *a* :: *ereal*
shows *real-of-ereal* (*inverse a*) = *1* / *real-of-ereal a*
⟨*proof*⟩

lemma *ereal-inverse*[*simp*]:
inverse (*0*::*ereal*) = ∞
inverse (*1*::*ereal*) = *1*
⟨*proof*⟩

lemma *ereal-divide*[*simp*]:
ereal r / *ereal p* = (*if p = 0 then ereal r* ∗ ∞ *else ereal* (*r* / *p*))
⟨*proof*⟩

lemma *ereal-divide-same*[*simp*]:
fixes *x* :: *ereal*
shows *x* / *x* = (*if* |*x*| = ∞ ∨ *x = 0 then 0 else 1*)
⟨*proof*⟩

lemma *ereal-inv-inv*[*simp*]:
fixes *x* :: *ereal*
shows *inverse* (*inverse x*) = (*if x* ≠ −∞ *then x else* ∞)
⟨*proof*⟩

lemma *ereal-inverse-minus*[*simp*]:
fixes x :: *ereal*
shows *inverse* $(-x) = (\textit{if } x = 0 \textit{ then } \infty \textit{ else } -\textit{inverse } x)$
⟨*proof*⟩

lemma *ereal-uminus-divide*[*simp*]:
fixes $x\ y$ :: *ereal*
shows $-x\ /\ y = -\ (x\ /\ y)$
⟨*proof*⟩

lemma *ereal-divide-Infty*[*simp*]:
fixes x :: *ereal*
shows $x\ /\ \infty = 0\ x\ /\ -\infty = 0$
⟨*proof*⟩

lemma *ereal-divide-one*[*simp*]: $x\ /\ 1 = (x::ereal)$
⟨*proof*⟩

lemma *ereal-divide-ereal*[*simp*]: $\infty\ /\ ereal\ r = (\textit{if } 0 \le r \textit{ then } \infty \textit{ else } -\infty)$
⟨*proof*⟩

lemma *ereal-inverse-nonneg-iff*: $0 \le \textit{inverse } (x :: ereal) \longleftrightarrow 0 \le x \vee x = -\infty$
⟨*proof*⟩

lemma *inverse-ereal-ge0I*: $0 \le (x :: ereal) \Longrightarrow 0 \le \textit{inverse } x$
⟨*proof*⟩

lemma *zero-le-divide-ereal*[*simp*]:
fixes a :: *ereal*
assumes $0 \le a$
and $0 \le b$
shows $0 \le a\ /\ b$
⟨*proof*⟩

lemma *ereal-le-divide-pos*:
fixes $x\ y\ z$ :: *ereal*
shows $x > 0 \Longrightarrow x \neq \infty \Longrightarrow y \le z\ /\ x \longleftrightarrow x * y \le z$
⟨*proof*⟩

lemma *ereal-divide-le-pos*:
fixes $x\ y\ z$ :: *ereal*
shows $x > 0 \Longrightarrow x \neq \infty \Longrightarrow z\ /\ x \le y \longleftrightarrow z \le x * y$
⟨*proof*⟩

lemma *ereal-le-divide-neg*:
fixes $x\ y\ z$ :: *ereal*
shows $x < 0 \Longrightarrow x \neq -\infty \Longrightarrow y \le z\ /\ x \longleftrightarrow z \le x * y$
⟨*proof*⟩

lemma *ereal-divide-le-neg*:
fixes $x\ y\ z$ :: *ereal*
shows $x < 0 \Longrightarrow x \neq -\infty \Longrightarrow z\ /\ x \leq y \longleftrightarrow x * y \leq z$
⟨*proof*⟩

lemma *ereal-inverse-antimono-strict*:
fixes $x\ y$ :: *ereal*
shows $0 \leq x \Longrightarrow x < y \Longrightarrow inverse\ y < inverse\ x$
⟨*proof*⟩

lemma *ereal-inverse-antimono*:
fixes $x\ y$ :: *ereal*
shows $0 \leq x \Longrightarrow x \leq y \Longrightarrow inverse\ y \leq inverse\ x$
⟨*proof*⟩

lemma *inverse-inverse-Pinfty-iff* [*simp*]:
fixes x :: *ereal*
shows $inverse\ x = \infty \longleftrightarrow x = 0$
⟨*proof*⟩

lemma *ereal-inverse-eq-0*:
fixes x :: *ereal*
shows $inverse\ x = 0 \longleftrightarrow x = \infty \vee x = -\infty$
⟨*proof*⟩

lemma *ereal-0-gt-inverse*:
fixes x :: *ereal*
shows $0 < inverse\ x \longleftrightarrow x \neq \infty \wedge 0 \leq x$
⟨*proof*⟩

lemma *ereal-inverse-le-0-iff*:
fixes x :: *ereal*
shows $inverse\ x \leq 0 \longleftrightarrow x < 0 \vee x = \infty$
⟨*proof*⟩

lemma *ereal-divide-eq-0-iff*: $x\ /\ y = 0 \longleftrightarrow x = 0 \vee |y :: ereal| = \infty$
⟨*proof*⟩

lemma *ereal-mult-less-right*:
fixes $a\ b\ c$ :: *ereal*
assumes $b * a < c * a$
and $0 < a$
and $a < \infty$
shows $b < c$
⟨*proof*⟩

lemma *ereal-mult-divide*: **fixes** $a\ b$ :: *ereal* **shows** $0 < b \Longrightarrow b < \infty \Longrightarrow b * (a\ /\ b) = a$
⟨*proof*⟩

lemma *ereal-power-divide*:
fixes $x\ y$ :: *ereal*
shows $y \neq 0 \Longrightarrow (x\ /\ y)\ \hat{}\ n = x\hat{}n\ /\ y\hat{}n$
⟨*proof*⟩

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*⟩

lemma *ereal-divide-right-mono*[*simp*]:
fixes $x\ y\ z$ :: *ereal*
assumes $x \leq y$
and $0 < z$
shows $x\ /\ z \leq y\ /\ z$
⟨*proof*⟩

lemma *ereal-divide-left-mono*[*simp*]:
fixes $x\ y\ z$ :: *ereal*
assumes $y \leq x$
and $0 < z$
and $0 < x * y$
shows $z\ /\ x \leq z\ /\ y$
⟨*proof*⟩

lemma *ereal-divide-zero-left*[*simp*]:
fixes a :: *ereal*
shows $0\ /\ a = 0$
⟨*proof*⟩

lemma *ereal-times-divide-eq-left*[*simp*]:
fixes $a\ b\ c$ :: *ereal*
shows $b\ /\ c * a = b * a\ /\ c$
⟨*proof*⟩

lemma *ereal-times-divide-eq*: $a * (b\ /\ c :: ereal) = a * b\ /\ c$
⟨*proof*⟩

lemma *ereal-inverse-real*: $|z| \neq \infty \Longrightarrow z \neq 0 \Longrightarrow ereal\ (inverse\ (real\text{-}of\text{-}ereal\ z))$
$= inverse\ z$
⟨*proof*⟩

lemma *ereal-inverse-mult*:
$a \neq 0 \Longrightarrow b \neq 0 \Longrightarrow inverse\ (a * (b::ereal)) = inverse\ a * inverse\ b$
⟨*proof*⟩

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 ⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *ereal-complete-Inf*:
∃ *x*. (∀ *y*∈*S*::*ereal set*. *x* ≤ *y*) ∧ (∀ *z*. (∀ *y*∈*S*. *z* ≤ *y*) ⟶ *z* ≤ *x*)
⟨*proof*⟩

instance
⟨*proof*⟩

end

instance *ereal* :: *complete-linorder* ⟨*proof*⟩

instance *ereal* :: *linear-continuum*
⟨*proof*⟩

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
⟨*proof*⟩

end

lemma *continuous-on-ereal*[*continuous-intros*]:
assumes *f*: *continuous-on s f* **shows** *continuous-on s* (λ*x*. *ereal* (*f x*))
⟨*proof*⟩

lemma *tendsto-ereal*[*tendsto-intros*, *simp*, *intro*]: (*f* ⟶ *x*) *F* ⟹ ((λ*x*. *ereal* (*f x*)) ⟶ *ereal x*) *F*
⟨*proof*⟩

lemma *tendsto-uminus-ereal*[*tendsto-intros*, *simp*, *intro*]: (*f* ⟶ *x*) *F* ⟹ ((λ*x*. − *f x*::*ereal*) ⟶ − *x*) *F*
⟨*proof*⟩

lemma *at-infty-ereal-eq-at-top*: *at* ∞ = *filtermap ereal at-top*
⟨*proof*⟩

lemma *ereal-Lim-uminus*: (*f* ⟶ *f0*) *net* ⟷ ((λ*x*. − *f x*::*ereal*) ⟶ − *f0*) *net*
⟨*proof*⟩

lemma *ereal-divide-less-iff*: *0* < (*c*::*ereal*) ⟹ *c* < ∞ ⟹ *a* / *c* < *b* ⟷ *a* < *b* ∗ *c*
⟨*proof*⟩

lemma *ereal-less-divide-iff*: *0* < (*c*::*ereal*) ⟹ *c* < ∞ ⟹ *a* < *b* / *c* ⟷ *a* ∗ *c* < *b*
⟨*proof*⟩

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*⟩

lemma *tendsto-cmult-ereal-not-0*[*tendsto-intros*, *simp*, *intro*]:
assumes *x* ≠ *0* **and** *f*: (*f* ⟶ *x*) *F* **shows** ((λ*x*. *c* ∗ *f x*::*ereal*) ⟶ *c* ∗ *x*) *F*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *continuous-at-ereal*[*continuous-intros*]: *continuous F f* $\Longrightarrow$ *continuous F* $(\lambda x.\ ereal\ (f\,x))$
⟨*proof*⟩

lemma *ereal-Sup*:
assumes $*$: $|SUP\ a{:}A.\ ereal\ a| \neq \infty$
shows *ereal* $(Sup\ A) = (SUP\ a{:}A.\ ereal\ a)$
⟨*proof*⟩

lemma *ereal-SUP*: $|SUP\ a{:}A.\ ereal\ (f\,a)| \neq \infty \Longrightarrow ereal\ (SUP\ a{:}A.\ f\,a) = (SUP\ a{:}A.\ ereal\ (f\,a))$
⟨*proof*⟩

lemma *ereal-Inf*:
assumes $*$: $|INF\ a{:}A.\ ereal\ a| \neq \infty$
shows *ereal* $(Inf\ A) = (INF\ a{:}A.\ ereal\ a)$
⟨*proof*⟩

lemma *ereal-Inf'*:
assumes $*$: *bdd-below A* $A \neq \{\}$
shows *ereal* $(Inf\ A) = (INF\ a{:}A.\ ereal\ a)$
⟨*proof*⟩

lemma *ereal-INF*: $|INF\ a{:}A.\ ereal\ (f\,a)| \neq \infty \Longrightarrow ereal\ (INF\ a{:}A.\ f\,a) = (INF\ a{:}A.\ ereal\ (f\,a))$
⟨*proof*⟩

lemma *ereal-Sup-uminus-image-eq*: *Sup* $(uminus\ `\ S{::}ereal\ set) = -\ Inf\ S$
⟨*proof*⟩

lemma *ereal-SUP-uminus-eq*:
fixes $f :: 'a \Rightarrow ereal$
shows $(SUP\ x{:}S.\ uminus\ (f\,x)) = -\ (INF\ x{:}S.\ f\,x)$
⟨*proof*⟩

lemma *ereal-inj-on-uminus*[*intro*, *simp*]: *inj-on uminus* $(A :: ereal\ set)$
⟨*proof*⟩

lemma *ereal-Inf-uminus-image-eq*: *Inf* $(uminus\ `\ S{::}ereal\ set) = -\ Sup\ S$
⟨*proof*⟩

lemma *ereal-INF-uminus-eq*:
fixes $f :: 'a \Rightarrow ereal$
shows $(INF\ x{:}S.\ -\ f\ x) = -\ (SUP\ x{:}S.\ f\ x)$
⟨*proof*⟩

lemma *ereal-SUP-uminus*:
fixes $f :: 'a \Rightarrow ereal$
shows $(SUP\ i : R.\ -\ f\ i) = -\ (INF\ i : R.\ f\ i)$
⟨*proof*⟩

lemma *ereal-SUP-not-infty*:
fixes $f :: - \Rightarrow ereal$
shows $A \neq \{\} \Longrightarrow l \neq -\infty \Longrightarrow u \neq \infty \Longrightarrow \forall a{\in}A.\ l \leq f\ a \wedge f\ a \leq u \Longrightarrow |SUPREMUM\ A\ f| \neq \infty$
⟨*proof*⟩

lemma *ereal-INF-not-infty*:
fixes $f :: - \Rightarrow ereal$
shows $A \neq \{\} \Longrightarrow l \neq -\infty \Longrightarrow u \neq \infty \Longrightarrow \forall a{\in}A.\ l \leq f\ a \wedge f\ a \leq u \Longrightarrow |INFIMUM\ A\ f| \neq \infty$
⟨*proof*⟩

lemma *ereal-image-uminus-shift*:
fixes $X\ Y :: ereal\ set$
shows $uminus\ `\ X = Y \longleftrightarrow X = uminus\ `\ Y$
⟨*proof*⟩

lemma *Sup-eq-MInfty*:
fixes $S :: ereal\ set$
shows $Sup\ S = -\infty \longleftrightarrow S = \{\} \vee S = \{-\infty\}$
⟨*proof*⟩

lemma *Inf-eq-PInfty*:
fixes $S :: ereal\ set$
shows $Inf\ S = \infty \longleftrightarrow S = \{\} \vee S = \{\infty\}$
⟨*proof*⟩

lemma *Inf-eq-MInfty*:
fixes $S :: ereal\ set$
shows $-\infty \in S \Longrightarrow Inf\ S = -\infty$
⟨*proof*⟩

lemma *Sup-eq-PInfty*:
fixes $S :: ereal\ set$
shows $\infty \in S \Longrightarrow Sup\ S = \infty$
⟨*proof*⟩

lemma *not-MInfty-nonneg*[*simp*]: $0 \leq (x{::}ereal) \Longrightarrow x \neq -\infty$

⟨*proof*⟩

lemma *Sup-ereal-close*:
fixes $e :: ereal$
assumes $0 < e$
and S: $|Sup\ S| \neq \infty\ S \neq \{\}$
shows $\exists x{\in}S.\ Sup\ S - e < x$
⟨*proof*⟩

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*⟩

lemma *SUP-PInfty*:
$(\bigwedge n{::}nat.\ \exists i{\in}A.\ ereal\ (real\ n) \leq f\ i) \Longrightarrow (SUP\ i{:}A.\ f\ i :: ereal) = \infty$
⟨*proof*⟩

lemma *SUP-nat-Infty*: $(SUP\ i{::}nat.\ ereal\ (real\ i)) = \infty$
⟨*proof*⟩

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*⟩

lemma *SUP-ereal-add-right*:
fixes $c :: ereal$
shows $I \neq \{\} \Longrightarrow c \neq -\infty \Longrightarrow (SUP\ i{:}I.\ c + f\ i) = c + (SUP\ i{:}I.\ f\ i)$
⟨*proof*⟩

lemma *SUP-ereal-minus-right*:
assumes $I \neq \{\}\ c \neq -\infty$
shows $(SUP\ i{:}I.\ c - f\ i :: ereal) = c - (INF\ i{:}I.\ f\ i)$
⟨*proof*⟩

lemma *SUP-ereal-minus-left*:
assumes $I \neq \{\}\ c \neq \infty$
shows $(SUP\ i{:}I.\ f\ i - c{::}\ ereal) = (SUP\ i{:}I.\ f\ i) - c$
⟨*proof*⟩

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*⟩

lemma *SUP-ereal-le-addI*:

fixes $f :: 'i \Rightarrow ereal$
assumes $\bigwedge i.\ f\ i + y \leq z$ **and** $y \neq -\infty$
shows $SUPREMUM\ UNIV\ f + y \leq z$
⟨*proof*⟩

lemma *SUP-combine*:
fixes $f :: 'a{::}semilattice\text{-}sup \Rightarrow 'a{::}semilattice\text{-}sup \Rightarrow 'b{::}complete\text{-}lattice$
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*⟩

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 $(SUP\ i.\ f\ i + g\ i) = SUPREMUM\ UNIV\ f + SUPREMUM\ UNIV\ g$
⟨*proof*⟩

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*⟩

lemma *SUP-ereal-add-pos*:
fixes $f\ g :: nat \Rightarrow ereal$
assumes *inc*: *incseq f incseq g*
and *pos*: $\bigwedge i.\ 0 \leq f\ i\ \bigwedge i.\ 0 \leq g\ i$
shows $(SUP\ i.\ f\ i + g\ i) = SUPREMUM\ UNIV\ f + SUPREMUM\ UNIV\ g$
⟨*proof*⟩

lemma *SUP-ereal-setsum*:
fixes $f\ g :: 'a \Rightarrow nat \Rightarrow ereal$
assumes $\bigwedge n.\ n \in A \Longrightarrow incseq\ (f\ n)$
and *pos*: $\bigwedge n\ i.\ n \in A \Longrightarrow 0 \leq f\ n\ i$
shows $(SUP\ i.\ \sum n{\in}A.\ f\ n\ i) = (\sum n{\in}A.\ SUPREMUM\ UNIV\ (f\ n))$
⟨*proof*⟩

lemma *SUP-ereal-mult-left*:
fixes $f :: 'a \Rightarrow ereal$
assumes $I \neq \{\}$
assumes f: $\bigwedge i.\ i \in I \Longrightarrow 0 \leq f\ i$ **and** c: $0 \leq c$
shows $(SUP\ i{:}I.\ c * f\ i) = c * (SUP\ i{:}I.\ f\ i)$
⟨*proof*⟩

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*⟩

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*⟩

lemma *SUP-countable-SUP*:
$A \neq \{\} \Longrightarrow \exists f{::}nat \Rightarrow ereal.\ range\ f \subseteq g`A \wedge SUPREMUM\ A\ g = SUPREMUM\ UNIV\ f$
⟨*proof*⟩

33.3 Relation to *enat*

definition $ereal\text{-}of\text{-}enat\ n = (case\ n\ of\ enat\ n \Rightarrow ereal\ (real\ n) \mid \infty \Rightarrow \infty)$

declare [[*coercion ereal-of-enat* :: *enat* ⇒ *ereal*]]
declare [[*coercion* (λ*n*. *ereal* (*real n*)) :: *nat* ⇒ *ereal*]]

lemma *ereal-of-enat-simps*[*simp*]:
$ereal\text{-}of\text{-}enat\ (enat\ n) = ereal\ n$
$ereal\text{-}of\text{-}enat\ \infty = \infty$
⟨*proof*⟩

lemma *ereal-of-enat-le-iff*[*simp*]: $ereal\text{-}of\text{-}enat\ m \leq ereal\text{-}of\text{-}enat\ n \longleftrightarrow m \leq n$
⟨*proof*⟩

lemma *ereal-of-enat-less-iff*[*simp*]: $ereal\text{-}of\text{-}enat\ m < ereal\text{-}of\text{-}enat\ n \longleftrightarrow m < n$
⟨*proof*⟩

lemma *numeral-le-ereal-of-enat-iff*[*simp*]: $numeral\ m \leq ereal\text{-}of\text{-}enat\ n \longleftrightarrow numeral\ m \leq n$
⟨*proof*⟩

lemma *numeral-less-ereal-of-enat-iff*[*simp*]: $numeral\ m < ereal\text{-}of\text{-}enat\ n \longleftrightarrow numeral\ m < n$
⟨*proof*⟩

lemma *ereal-of-enat-ge-zero-cancel-iff*[*simp*]: $0 \leq ereal\text{-}of\text{-}enat\ n \longleftrightarrow 0 \leq n$
⟨*proof*⟩

lemma *ereal-of-enat-gt-zero-cancel-iff*[*simp*]: $0 < ereal\text{-}of\text{-}enat\ n \longleftrightarrow 0 < n$
⟨*proof*⟩

lemma *ereal-of-enat-zero*[*simp*]: $ereal\text{-}of\text{-}enat\ 0 = 0$
⟨*proof*⟩

lemma *ereal-of-enat-inf*[*simp*]: $ereal\text{-}of\text{-}enat\ n = \infty \longleftrightarrow n = \infty$

⟨*proof*⟩

lemma *ereal-of-enat-add*: *ereal-of-enat* $(m + n) =$ *ereal-of-enat* $m +$ *ereal-of-enat* n
⟨*proof*⟩

lemma *ereal-of-enat-sub*:
assumes $n \leq m$
shows *ereal-of-enat* $(m - n) =$ *ereal-of-enat* $m -$ *ereal-of-enat* n
⟨*proof*⟩

lemma *ereal-of-enat-mult*:
ereal-of-enat $(m * n) =$ *ereal-of-enat* $m *$ *ereal-of-enat* n
⟨*proof*⟩

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* $n \geq 0$
⟨*proof*⟩

lemma *ereal-of-enat-Sup*:
assumes $A \neq \{\}$ **shows** *ereal-of-enat* $(Sup\ A) = (SUP\ a : A.$ *ereal-of-enat* $a)$
⟨*proof*⟩

lemma *ereal-of-enat-SUP*:
$A \neq \{\} \Longrightarrow$ *ereal-of-enat* $(SUP\ a{:}A.\ f\ a) = (SUP\ a : A.$ *ereal-of-enat* $(f\ a))$
⟨*proof*⟩

33.4 Limits on *ereal*

lemma *open-PInfty*: *open* $A \Longrightarrow \infty \in A \Longrightarrow (\exists\, x.\ \{$*ereal* $x{<}..\} \subseteq A)$
⟨*proof*⟩

lemma *open-MInfty*: *open* $A \Longrightarrow -\infty \in A \Longrightarrow (\exists\, x.\ \{..{<}$*ereal* $x\} \subseteq A)$
⟨*proof*⟩

lemma *open-ereal-vimage*: *open* $S \Longrightarrow$ *open* $($*ereal* $-`\ S)$
⟨*proof*⟩

lemma *open-ereal*: *open* $S \Longrightarrow$ *open* $($*ereal* $`\ S)$
⟨*proof*⟩

lemma *eventually-finite*:
fixes x :: *ereal*
assumes $|x| \neq \infty$ $(f \longrightarrow x)\ F$
shows *eventually* $(\lambda x.\ |f\ x| \neq \infty)\ F$
⟨*proof*⟩

lemma *open-ereal-def*:
open A $\longleftrightarrow$ *open* (*ereal* $-`$ *A*) $\wedge$ ($\infty \in A \longrightarrow (\exists x.\ \{ereal\ x <..\} \subseteq A)$) $\wedge$ ($-\infty \in A \longrightarrow (\exists x.\ \{..<ereal\ x\} \subseteq A)$)
(**is** *open A* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *open-PInfty2*:
assumes *open A*
and $\infty \in A$
obtains *x* **where** $\{ereal\ x<..\} \subseteq A$
⟨*proof*⟩

lemma *open-MInfty2*:
assumes *open A*
and $-\infty \in A$
obtains *x* **where** $\{..<ereal\ x\} \subseteq A$
⟨*proof*⟩

lemma *ereal-openE*:
assumes *open A*
obtains *x y* **where** *open* (*ereal* $-`$ *A*)
and $\infty \in A \Longrightarrow \{ereal\ x<..\} \subseteq A$
and $-\infty \in A \Longrightarrow \{..<ereal\ y\} \subseteq A$
⟨*proof*⟩

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*⟩

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*⟩

33.4.1 Convergent sequences

lemma *lim-real-of-ereal*[*simp*]:
assumes *lim*: $(f \longrightarrow ereal\ x)\ net$
shows $((\lambda x.\ real\text{-}of\text{-}ereal\ (f\ x)) \longrightarrow x)\ net$
⟨*proof*⟩

lemma *lim-ereal*[*simp*]: $((\lambda n.\ ereal\ (f\ n)) \longrightarrow ereal\ x)\ net \longleftrightarrow (f \longrightarrow x)\ net$
⟨*proof*⟩

lemma *convergent-real-imp-convergent-ereal*:
assumes *convergent a*
shows *convergent* $(\lambda n.\ ereal\ (a\ n))$ **and** $lim\ (\lambda n.\ ereal\ (a\ n)) = ereal\ (lim\ a)$
⟨*proof*⟩

lemma *tendsto-PInfty*: $(f \longrightarrow \infty)\ F \longleftrightarrow (\forall r.\ eventually\ (\lambda x.\ ereal\ r < f\ x)\ F)$
⟨*proof*⟩

lemma *tendsto-PInfty'*: $(f \longrightarrow \infty)\ F = (\forall r{>}c.\ eventually\ (\lambda x.\ ereal\ r < f\ x)\ F)$
⟨*proof*⟩

lemma *tendsto-PInfty-eq-at-top*:
$((\lambda z.\ ereal\ (f\ z)) \longrightarrow \infty)\ F \longleftrightarrow (LIM\ z\ F.\ f\ z :> at\text{-}top)$
⟨*proof*⟩

lemma *tendsto-MInfty*: $(f \longrightarrow -\infty)\ F \longleftrightarrow (\forall r.\ eventually\ (\lambda x.\ f\ x < ereal\ r)\ F)$
⟨*proof*⟩

lemma *tendsto-MInfty'*: $(f \longrightarrow -\infty)\ F = (\forall r{<}c.\ eventually\ (\lambda x.\ ereal\ r > f\ x)\ F)$
⟨*proof*⟩

lemma *Lim-PInfty*: $f \longrightarrow \infty \longleftrightarrow (\forall B.\ \exists N.\ \forall n{\geq}N.\ f\ n \geq ereal\ B)$
⟨*proof*⟩

lemma *Lim-MInfty*: $f \longrightarrow -\infty \longleftrightarrow (\forall B.\ \exists N.\ \forall n{\geq}N.\ ereal\ B \geq f\ n)$
⟨*proof*⟩

lemma *Lim-bounded-PInfty*: $f \longrightarrow l \Longrightarrow (\bigwedge n.\ f\ n \leq ereal\ B) \Longrightarrow l \neq \infty$
⟨*proof*⟩

lemma *Lim-bounded-MInfty*: $f \longrightarrow l \Longrightarrow (\bigwedge n.\ ereal\ B \leq f\ n) \Longrightarrow l \neq -\infty$
⟨*proof*⟩

lemma *tendsto-zero-erealI*:

assumes $\bigwedge e.\ e > 0 \Longrightarrow$ *eventually* $(\lambda x.\ |f\ x| < ereal\ e)\ F$
shows $(f \longrightarrow 0)\ F$
⟨*proof*⟩

lemma *tendsto-explicit*:
$f \longrightarrow f0 \longleftrightarrow (\forall S.\ open\ S \longrightarrow f0 \in S \longrightarrow (\exists N.\ \forall n{\geq}N.\ f\ n \in S))$
⟨*proof*⟩

lemma *Lim-bounded-PInfty2*: $f \longrightarrow l \Longrightarrow \forall n{\geq}N.\ f\ n \leq ereal\ B \Longrightarrow l \neq \infty$
⟨*proof*⟩

lemma *Lim-bounded-ereal*: $f \longrightarrow (l :: 'a{::}linorder\text{-}topology) \Longrightarrow \forall n{\geq}M.\ f\ n \leq C \Longrightarrow l \leq C$
⟨*proof*⟩

lemma *Lim-bounded2-ereal*:
assumes *lim*:$f \longrightarrow (l :: 'a{::}linorder\text{-}topology)$
and *ge*: $\forall n{\geq}N.\ f\ n \geq C$
shows $l \geq C$
⟨*proof*⟩

lemma *real-of-ereal-mult*[*simp*]:
fixes $a\ b :: ereal$
shows *real-of-ereal* $(a * b) =$ *real-of-ereal* $a *$ *real-of-ereal* b
⟨*proof*⟩

lemma *real-of-ereal-eq-0*:
fixes $x :: ereal$
shows *real-of-ereal* $x = 0 \longleftrightarrow x = \infty \vee x = -\infty \vee x = 0$
⟨*proof*⟩

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*⟩

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*⟩

lemma *ereal-mult-cancel-left*:
fixes $a\ b\ c :: ereal$
shows $a * b = a * c \longleftrightarrow (|a| = \infty \wedge 0 < b * c) \vee a = 0 \vee b = c$
⟨*proof*⟩

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*⟩

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*⟩

lemma *ereal-inj-affinity*:
fixes $m\ t$ :: *ereal*
assumes $|m| \neq \infty$
and $m \neq 0$
and $|t| \neq \infty$
shows *inj-on* $(\lambda x.\ m * x + t)\ A$
⟨*proof*⟩

lemma *ereal-PInfty-eq-plus*[*simp*]:
fixes $a\ b$ :: *ereal*
shows $\infty = a + b \longleftrightarrow a = \infty \vee b = \infty$
⟨*proof*⟩

lemma *ereal-MInfty-eq-plus*[*simp*]:
fixes $a\ b$ :: *ereal*
shows $-\infty = a + b \longleftrightarrow (a = -\infty \wedge b \neq \infty) \vee (b = -\infty \wedge a \neq \infty)$
⟨*proof*⟩

lemma *ereal-less-divide-pos*:
fixes $x\ y$ :: *ereal*
shows $x > 0 \Longrightarrow x \neq \infty \Longrightarrow y < z\ /\ x \longleftrightarrow x * y < z$
⟨*proof*⟩

lemma *ereal-divide-less-pos*:
fixes $x\ y\ z$ :: *ereal*
shows $x > 0 \Longrightarrow x \neq \infty \Longrightarrow y\ /\ x < z \longleftrightarrow y < x * z$
⟨*proof*⟩

lemma *ereal-divide-eq*:
fixes $a\ b\ c$ :: *ereal*
shows $b \neq 0 \Longrightarrow |b| \neq \infty \Longrightarrow a\ /\ b = c \longleftrightarrow a = b * c$
⟨*proof*⟩

lemma *ereal-inverse-not-MInfty*[*simp*]: *inverse* $(a$::*ereal*$) \neq -\infty$
⟨*proof*⟩

lemma *ereal-mult-m1*[*simp*]: $x * ereal\ (-1) = -x$
⟨*proof*⟩

lemma *ereal-real'*:
assumes $|x| \neq \infty$
shows $ereal\ (real\text{-}of\text{-}ereal\ x) = x$
⟨*proof*⟩

lemma *real-ereal-id*: $real\text{-}of\text{-}ereal \circ ereal = id$
⟨*proof*⟩

lemma *open-image-ereal*: $open(UNIV - \{ \infty , (-\infty :: ereal)\})$
⟨*proof*⟩

lemma *ereal-le-distrib*:
fixes $a\ b\ c :: ereal$
shows $c * (a + b) \leq c * a + c * b$
⟨*proof*⟩

lemma *ereal-pos-distrib*:
fixes $a\ b\ c :: ereal$
assumes $0 \leq c$
and $c \neq \infty$
shows $c * (a + b) = c * a + c * b$
⟨*proof*⟩

lemma *ereal-max-mono*: $(a::ereal) \leq b \Longrightarrow c \leq d \Longrightarrow max\ a\ c \leq max\ b\ d$
⟨*proof*⟩

lemma *ereal-max-least*: $(a::ereal) \leq x \Longrightarrow c \leq x \Longrightarrow max\ a\ c \leq x$
⟨*proof*⟩

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 $u \longrightarrow x$
⟨*proof*⟩

lemma *tendsto-obtains-N*:
assumes $f \longrightarrow f0$
assumes $open\ S$
and $f0 \in S$
obtains N **where** $\forall n \geq N.\ f\ n \in S$
⟨*proof*⟩

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* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *ereal-Limsup-uminus*:
fixes $f :: 'a \Rightarrow ereal$
shows *Limsup net* $(\lambda x.\ -\ (f\ x)) = -$ *Liminf net f*
⟨*proof*⟩

lemma *liminf-bounded-iff*:
fixes $x :: nat \Rightarrow ereal$
shows $C \leq liminf\ x \longleftrightarrow (\forall B{<}C.\ \exists N.\ \forall n \geq N.\ B < x\ n)$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *Liminf-add-le*:
fixes $f\ g :: - \Rightarrow ereal$
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* $(\lambda x.\ f\ x + g\ x)$
⟨*proof*⟩

lemma *Sup-ereal-mult-right'*:
assumes *nonempty*: $Y \neq \{\}$
and x: $x \geq 0$
shows $(SUP\ i{:}Y.\ f\ i) * ereal\ x = (SUP\ i{:}Y.\ f\ i * ereal\ x)$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *Sup-ereal-mult-left'*:
$[\![\ Y \neq \{\};\ x \geq 0\ ]\!] \Longrightarrow ereal\ x * (SUP\ i{:}Y.\ f\ i) = (SUP\ i{:}Y.\ ereal\ x * f\ i)$
⟨*proof*⟩

lemma *sup-continuous-add*[*order-continuous-intros*]:
fixes $f\ g :: 'a{::}complete\text{-}lattice \Rightarrow ereal$
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* $(\lambda x.\ f\ x + g\ x)$
⟨*proof*⟩

lemma *sup-continuous-mult-right*[*order-continuous-intros*]:
$0 \leq c \Longrightarrow c < \infty \Longrightarrow$ *sup-continuous f* $\Longrightarrow$ *sup-continuous* $(\lambda x.\ f\ x * c :: ereal)$
⟨*proof*⟩

lemma *sup-continuous-mult-left*[*order-continuous-intros*]:
$0 \leq c \Longrightarrow c < \infty \Longrightarrow$ *sup-continuous f* $\Longrightarrow$ *sup-continuous* $(\lambda x.\ c * f\ x :: ereal)$
⟨*proof*⟩

lemma *sup-continuous-ereal-of-enat*[*order-continuous-intros*]:

assumes f: *sup-continuous* f **shows** *sup-continuous* $(\lambda x.\ ereal\text{-}of\text{-}enat\ (f\ x))$
⟨*proof*⟩

33.4.2 Sums

lemma *sums-ereal-positive*:
fixes $f :: nat \Rightarrow ereal$
assumes $\bigwedge i.\ 0 \le f\ i$
shows $f\ sums\ (SUP\ n.\ \sum i{<}n.\ f\ i)$
⟨*proof*⟩

lemma *summable-ereal-pos*:
fixes $f :: nat \Rightarrow ereal$
assumes $\bigwedge i.\ 0 \le f\ i$
shows *summable* f
⟨*proof*⟩

lemma *sums-ereal*: $(\lambda x.\ ereal\ (f\ x))\ sums\ ereal\ x \longleftrightarrow f\ sums\ x$
⟨*proof*⟩

lemma *suminf-ereal-eq-SUP*:
fixes $f :: nat \Rightarrow ereal$
assumes $\bigwedge i.\ 0 \le f\ i$
shows $(\sum x.\ f\ x) = (SUP\ n.\ \sum i{<}n.\ f\ i)$
⟨*proof*⟩

lemma *suminf-bound*:
fixes $f :: nat \Rightarrow ereal$
assumes $\forall N.\ (\sum n{<}N.\ f\ n) \le x$
and *pos*: $\bigwedge n.\ 0 \le f\ n$
shows $suminf\ f \le x$
⟨*proof*⟩

lemma *suminf-bound-add*:
fixes $f :: nat \Rightarrow ereal$
assumes $\forall N.\ (\sum n{<}N.\ f\ n) + y \le x$
and *pos*: $\bigwedge n.\ 0 \le f\ n$
and $y \neq -\infty$
shows $suminf\ f + y \le x$
⟨*proof*⟩

lemma *suminf-upper*:
fixes $f :: nat \Rightarrow ereal$
assumes $\bigwedge n.\ 0 \le f\ n$
shows $(\sum n{<}N.\ f\ n) \le (\sum n.\ f\ n)$
⟨*proof*⟩

lemma *suminf-0-le*:
fixes $f :: nat \Rightarrow ereal$

assumes $\bigwedge n.\ 0 \leq f\ n$
shows $0 \leq (\sum n.\ f\ n)$
⟨*proof*⟩

lemma *suminf-le-pos*:
fixes $f\ g :: nat \Rightarrow ereal$
assumes $\bigwedge N.\ f\ N \leq g\ N$
and $\bigwedge N.\ 0 \leq f\ N$
shows $suminf\ f \leq suminf\ g$
⟨*proof*⟩

lemma *suminf-half-series-ereal*: $(\sum n.\ (1/2 :: ereal)\ \hat{}\ Suc\ n) = 1$
⟨*proof*⟩

lemma *suminf-add-ereal*:
fixes $f\ g :: nat \Rightarrow ereal$
assumes $\bigwedge i.\ 0 \leq f\ i$
and $\bigwedge i.\ 0 \leq g\ i$
shows $(\sum i.\ f\ i + g\ i) = suminf\ f + suminf\ g$
⟨*proof*⟩

lemma *suminf-cmult-ereal*:
fixes $f\ g :: nat \Rightarrow ereal$
assumes $\bigwedge i.\ 0 \leq f\ i$
and $0 \leq a$
shows $(\sum i.\ a * f\ i) = a * suminf\ f$
⟨*proof*⟩

lemma *suminf-PInfty*:
fixes $f :: nat \Rightarrow ereal$
assumes $\bigwedge i.\ 0 \leq f\ i$
and $suminf\ f \neq \infty$
shows $f\ i \neq \infty$
⟨*proof*⟩

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*⟩

lemma *summable-ereal*:
assumes $\bigwedge i.\ 0 \leq f\ i$
and $(\sum i.\ ereal\ (f\ i)) \neq \infty$
shows $summable\ f$
⟨*proof*⟩

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*⟩

lemma *suminf-ereal-minus*:
fixes $f\ g :: nat \Rightarrow ereal$
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*⟩

lemma *suminf-ereal-PInf* [*simp*]: $(\sum x.\ \infty::ereal) = \infty$
⟨*proof*⟩

lemma *summable-real-of-ereal*:
fixes $f :: nat \Rightarrow ereal$
assumes *f*: $\bigwedge i.\ 0 \leq f\ i$
and *fin*: $(\sum i.\ f\ i) \neq \infty$
shows *summable* $(\lambda i.\ real\text{-}of\text{-}ereal\ (f\ i))$
⟨*proof*⟩

lemma *suminf-SUP-eq*:
fixes $f :: nat \Rightarrow nat \Rightarrow ereal$
assumes $\bigwedge i.\ incseq\ (\lambda n.\ f\ n\ i)$
and $\bigwedge n\ i.\ 0 \leq f\ n\ i$
shows $(\sum i.\ SUP\ n.\ f\ n\ i) = (SUP\ n.\ \sum i.\ f\ n\ i)$
⟨*proof*⟩

lemma *suminf-setsum-ereal*:
fixes $f :: - \Rightarrow - \Rightarrow ereal$
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*⟩

lemma *suminf-ereal-eq-0*:
fixes $f :: nat \Rightarrow ereal$
assumes *nneg*: $\bigwedge i.\ 0 \leq f\ i$
shows $(\sum i.\ f\ i) = 0 \longleftrightarrow (\forall i.\ f\ i = 0)$
⟨*proof*⟩

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*⟩

lemma *sums-suminf-ereal*: $f\ sums\ x \Longrightarrow (\sum i.\ ereal\ (f\ i)) = ereal\ x$
⟨*proof*⟩

lemma *suminf-ereal'*: *summable f* $\Longrightarrow$ $(\sum i.\ ereal\ (f\ i)) = ereal\ (\sum i.\ f\ i)$
⟨*proof*⟩

lemma *suminf-ereal-finite*: *summable f* $\Longrightarrow$ $(\sum i.\ ereal\ (f\ i)) \neq \infty$
⟨*proof*⟩

lemma *suminf-ereal-finite-neg*:
assumes *summable f*
shows $(\sum x.\ ereal\ (f\ x)) \neq -\infty$
⟨*proof*⟩

lemma *SUP-ereal-add-directed*:
fixes *f g* :: *'a* ⇒ *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*⟩

lemma *SUP-ereal-setsum-directed*:
fixes *f g* :: *'a* ⇒ *'b* ⇒ *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*⟩

lemma *suminf-SUP-eq-directed*:
fixes *f* :: *- ⇒ nat ⇒ ereal*
assumes $I \neq \{\}$
assumes *directed*: $\bigwedge N\ i\ j.\ i \in I \Longrightarrow j \in I \Longrightarrow finite\ N \Longrightarrow \exists k{\in}I.\ \forall n{\in}N.\ f\ i\ n \leq f\ k\ n \wedge f\ j\ n \leq f\ k\ n$
assumes *nonneg*: $\bigwedge n\ i.\ 0 \leq f\ n\ i$
shows $(\sum i.\ SUP\ n{:}I.\ f\ n\ i) = (SUP\ n{:}I.\ \sum i.\ f\ n\ i)$
⟨*proof*⟩

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*⟩

lemma *continuous-within-ereal*[*intro*, *simp*]: $x \in A \Longrightarrow continuous\ (at\ x\ within\ A)\ ereal$
⟨*proof*⟩

lemma *ereal-open-uminus*:
fixes *S* :: *ereal set*
assumes *open S*
shows *open* (*uminus* ‘ *S*)

⟨*proof*⟩

lemma *ereal-uminus-complement*:
fixes S :: *ereal set*
shows *uminus* ‘ $(-S) = -$ *uminus* ‘ S
⟨*proof*⟩

lemma *ereal-closed-uminus*:
fixes S :: *ereal set*
assumes *closed* S
shows *closed* (*uminus* ‘ S)
⟨*proof*⟩

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*⟩

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*⟩

lemma *open-uminus-iff*:
fixes S :: *ereal set*
shows *open* (*uminus* ‘ S) $\longleftrightarrow$ *open* S
⟨*proof*⟩

lemma *ereal-Liminf-uminus*:
fixes f :: $'a \Rightarrow$ *ereal*
shows *Liminf net* $(\lambda x. -(f\,x)) = -$ *Limsup net* f
⟨*proof*⟩

lemma *Liminf-PInfty*:
fixes f :: $'a \Rightarrow$ *ereal*
assumes ¬ *trivial-limit net*
shows $(f \longrightarrow \infty)$ *net* $\longleftrightarrow$ *Liminf net* $f = \infty$
⟨*proof*⟩

lemma *Limsup-MInfty*:
fixes f :: $'a \Rightarrow$ *ereal*
assumes ¬ *trivial-limit net*
shows $(f \longrightarrow -\infty)$ *net* $\longleftrightarrow$ *Limsup net* $f = -\infty$

⟨*proof*⟩

lemma *convergent-ereal*: — RENAME
fixes $X :: nat \Rightarrow 'a :: \{complete\text{-}linorder, linorder\text{-}topology\}$
shows $convergent\ X \longleftrightarrow limsup\ X = liminf\ X$
⟨*proof*⟩

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*⟩

lemma *liminf-PInfty*:
fixes $X :: nat \Rightarrow ereal$
shows $X \longrightarrow \infty \longleftrightarrow liminf\ X = \infty$
⟨*proof*⟩

lemma *limsup-MInfty*:
fixes $X :: nat \Rightarrow ereal$
shows $X \longrightarrow -\infty \longleftrightarrow limsup\ X = -\infty$
⟨*proof*⟩

lemma *ereal-lim-mono*:
fixes $X\ Y :: nat \Rightarrow 'a :: linorder\text{-}topology$
assumes $\bigwedge n.\ N \leq n \Longrightarrow X\ n \leq Y\ n$
and $X \longrightarrow x$
and $Y \longrightarrow y$
shows $x \leq y$
⟨*proof*⟩

lemma *incseq-le-ereal*:
fixes $X :: nat \Rightarrow 'a :: linorder\text{-}topology$
assumes *inc*: *incseq X*
and *lim*: $X \longrightarrow L$
shows $X\ N \leq L$
⟨*proof*⟩

lemma *decseq-ge-ereal*:
assumes *dec*: *decseq X*
and *lim*: $X \longrightarrow (L :: 'a :: linorder\text{-}topology)$
shows $X\ N \geq L$
⟨*proof*⟩

lemma *bounded-abs*:
fixes $a :: real$
assumes $a \leq x$
and $x \leq b$
shows $|x| \leq max\ |a|\ |b|$

⟨*proof*⟩

lemma *ereal-Sup-lim*:
fixes $a :: 'a::\{complete\text{-}linorder, linorder\text{-}topology\}$
assumes $\bigwedge n.\ b\ n \in s$
and $b \longrightarrow a$
shows $a \leq Sup\ s$
⟨*proof*⟩

lemma *ereal-Inf-lim*:
fixes $a :: 'a::\{complete\text{-}linorder, linorder\text{-}topology\}$
assumes $\bigwedge n.\ b\ n \in s$
and $b \longrightarrow a$
shows $Inf\ s \leq a$
⟨*proof*⟩

lemma *SUP-Lim-ereal*:
fixes $X :: nat \Rightarrow 'a::\{complete\text{-}linorder, linorder\text{-}topology\}$
assumes *inc*: *incseq X*
and *l*: $X \longrightarrow l$
shows $(SUP\ n.\ X\ n) = l$
⟨*proof*⟩

lemma *INF-Lim-ereal*:
fixes $X :: nat \Rightarrow 'a::\{complete\text{-}linorder, linorder\text{-}topology\}$
assumes *dec*: *decseq X*
and *l*: $X \longrightarrow l$
shows $(INF\ n.\ X\ n) = l$
⟨*proof*⟩

lemma *SUP-eq-LIMSEQ*:
assumes *mono f*
shows $(SUP\ n.\ ereal\ (f\ n)) = ereal\ x \longleftrightarrow f \longrightarrow x$
⟨*proof*⟩

lemma *liminf-ereal-cminus*:
fixes $f :: nat \Rightarrow ereal$
assumes $c \neq -\infty$
shows $liminf\ (\lambda x.\ c - f\ x) = c - limsup\ f$
⟨*proof*⟩

33.4.3 Continuity

lemma *continuous-at-of-ereal*:
$|x0 :: ereal| \neq \infty \Longrightarrow continuous\ (at\ x0)\ real\text{-}of\text{-}ereal$
⟨*proof*⟩

lemma *nhds-ereal*: $nhds\ (ereal\ r) = filtermap\ ereal\ (nhds\ r)$
⟨*proof*⟩

lemma *at-ereal*: $at\ (ereal\ r) = filtermap\ ereal\ (at\ r)$
⟨*proof*⟩

lemma *at-left-ereal*: $at\text{-}left\ (ereal\ r) = filtermap\ ereal\ (at\text{-}left\ r)$
⟨*proof*⟩

lemma *at-right-ereal*: $at\text{-}right\ (ereal\ r) = filtermap\ ereal\ (at\text{-}right\ r)$
⟨*proof*⟩

lemma
shows *at-left-PInf*: $at\text{-}left\ \infty = filtermap\ ereal\ at\text{-}top$
and *at-right-MInf*: $at\text{-}right\ (-\infty) = filtermap\ ereal\ at\text{-}bot$
⟨*proof*⟩

lemma *ereal-tendsto-simps1*:
$((f \circ real\text{-}of\text{-}ereal) \longrightarrow y)\ (at\text{-}left\ (ereal\ x)) \longleftrightarrow (f \longrightarrow y)\ (at\text{-}left\ x)$
$((f \circ real\text{-}of\text{-}ereal) \longrightarrow y)\ (at\text{-}right\ (ereal\ x)) \longleftrightarrow (f \longrightarrow y)\ (at\text{-}right\ x)$
$((f \circ real\text{-}of\text{-}ereal) \longrightarrow y)\ (at\text{-}left\ (\infty{::}ereal)) \longleftrightarrow (f \longrightarrow y)\ at\text{-}top$
$((f \circ real\text{-}of\text{-}ereal) \longrightarrow y)\ (at\text{-}right\ (-\infty{::}ereal)) \longleftrightarrow (f \longrightarrow y)\ at\text{-}bot$
⟨*proof*⟩

lemma *ereal-tendsto-simps2*:
$((ereal \circ f) \longrightarrow ereal\ a)\ F \longleftrightarrow (f \longrightarrow a)\ F$
$((ereal \circ f) \longrightarrow \infty)\ F \longleftrightarrow (LIM\ x\ F.\ f\ x :> at\text{-}top)$
$((ereal \circ f) \longrightarrow -\infty)\ F \longleftrightarrow (LIM\ x\ F.\ f\ x :> at\text{-}bot)$
⟨*proof*⟩

lemma *inverse-infty-ereal-tendsto-0*: $inverse\ -\infty\rightarrow (0{::}ereal)$
⟨*proof*⟩

lemma *inverse-ereal-tendsto-pos*:
fixes $x :: ereal$ **assumes** $0 < x$
shows $inverse\ -x\rightarrow inverse\ x$
⟨*proof*⟩

lemma *inverse-ereal-tendsto-at-right-0*: $(inverse \longrightarrow \infty)\ (at\text{-}right\ (0{::}ereal))$
⟨*proof*⟩

lemmas *ereal-tendsto-simps* = *ereal-tendsto-simps1 ereal-tendsto-simps2*

lemma *continuous-at-iff-ereal*:
fixes $f :: 'a{::}t2\text{-}space \Rightarrow real$
shows $continuous\ (at\ x0\ within\ s)\ f \longleftrightarrow continuous\ (at\ x0\ within\ s)\ (ereal \circ f)$
⟨*proof*⟩

lemma *continuous-on-iff-ereal*:
fixes $f :: 'a{::}t2\text{-}space => real$
assumes *open A*

shows *continuous-on A f* $\longleftrightarrow$ *continuous-on A* (*ereal* $\circ$ *f*)
⟨*proof*⟩

lemma *continuous-on-real*: *continuous-on* (*UNIV* − {∞, −∞::*ereal*}) *real-of-ereal*
⟨*proof*⟩

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* $\circ$ *f*)
⟨*proof*⟩

lemma *continuous-uminus-ereal* [*continuous-intros*]: *continuous-on* (*A* :: *ereal set*) *uminus*
⟨*proof*⟩

lemma *ereal-uminus-atMost* [*simp*]: *uminus* ' {..(*a*::*ereal*)} = {−*a*..}
⟨*proof*⟩

lemma *continuous-on-inverse-ereal* [*continuous-intros*]:
continuous-on {*0*::*ereal* ..} *inverse*
⟨*proof*⟩

lemma *continuous-inverse-ereal-nonpos*: *continuous-on* ({..<*0*} :: *ereal set*) *inverse*
⟨*proof*⟩

lemma *tendsto-inverse-ereal*:
assumes (*f* $\longrightarrow$ (*c* :: *ereal*)) *F*
assumes *eventually* (λ*x*. *f x* ≥ *0*) *F*
shows ((λ*x*. *inverse* (*f x*)) $\longrightarrow$ *inverse c*) *F*
⟨*proof*⟩

33.4.4 liminf and limsup

lemma *Limsup-ereal-mult-right*:
assumes *F* ≠ *bot* (*c*::*real*) ≥ *0*
shows *Limsup F* (λ*n*. *f n* ∗ *ereal c*) = *Limsup F f* ∗ *ereal c*
⟨*proof*⟩

lemma *Liminf-ereal-mult-right*:
assumes *F* ≠ *bot* (*c*::*real*) ≥ *0*
shows *Liminf F* (λ*n*. *f n* ∗ *ereal c*) = *Liminf F f* ∗ *ereal c*
⟨*proof*⟩

lemma *Limsup-ereal-mult-left*:
assumes *F* ≠ *bot* (*c*::*real*) ≥ *0*
shows *Limsup F* (λ*n*. *ereal c* ∗ *f n*) = *ereal c* ∗ *Limsup F f*
⟨*proof*⟩

lemma *limsup-ereal-mult-right*:
(*c*::*real*) ≥ *0* ⟹ *limsup* (λ*n*. *f n* ∗ *ereal c*) = *limsup f* ∗ *ereal c*
⟨*proof*⟩

lemma *limsup-ereal-mult-left*:
(*c*::*real*) ≥ *0* ⟹ *limsup* (λ*n*. *ereal c* ∗ *f n*) = *ereal c* ∗ *limsup f*
⟨*proof*⟩

lemma *Limsup-add-ereal-right*:
F ≠ *bot* ⟹ *abs c* ≠ ∞ ⟹ *Limsup F* (λ*n*. *g n* + (*c* :: *ereal*)) = *Limsup F g* + *c*
⟨*proof*⟩

lemma *Limsup-add-ereal-left*:
F ≠ *bot* ⟹ *abs c* ≠ ∞ ⟹ *Limsup F* (λ*n*. (*c* :: *ereal*) + *g n*) = *c* + *Limsup F g*
⟨*proof*⟩

lemma *Liminf-add-ereal-right*:
F ≠ *bot* ⟹ *abs c* ≠ ∞ ⟹ *Liminf F* (λ*n*. *g n* + (*c* :: *ereal*)) = *Liminf F g* + *c*
⟨*proof*⟩

lemma *Liminf-add-ereal-left*:
F ≠ *bot* ⟹ *abs c* ≠ ∞ ⟹ *Liminf F* (λ*n*. (*c* :: *ereal*) + *g n*) = *c* + *Liminf F g*
⟨*proof*⟩

lemma
assumes *F* ≠ *bot*
assumes *nonneg*: *eventually* (λ*x*. *f x* ≥ (*0*::*ereal*)) *F*
shows *Liminf-inverse-ereal*: *Liminf F* (λ*x*. *inverse* (*f x*)) = *inverse* (*Limsup F f*)
and *Limsup-inverse-ereal*: *Limsup F* (λ*x*. *inverse* (*f x*)) = *inverse* (*Liminf F f*)
⟨*proof*⟩

33.4.5 Tests for code generator

value − ∞ :: *ereal*
value |−∞| :: *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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *indicator-abs-le-1*: |*indicator S x*| ≤ (*1*::'*a*::*linordered-idom*)
⟨*proof*⟩

lemma *indicator-eq-0-iff*: *indicator A x* = (*0*::-::*zero-neq-one*) ⟷ *x* ∉ *A*
⟨*proof*⟩

lemma *indicator-eq-1-iff*: *indicator A x* = (*1*::-::*zero-neq-one*) ⟷ *x* ∈ *A*
⟨*proof*⟩

lemma *split-indicator*: *P* (*indicator S x*) ⟷ ((*x* ∈ *S* ⟶ *P 1*) ∧ (*x* ∉ *S* ⟶ *P 0*))
⟨*proof*⟩

lemma *split-indicator-asm*: *P* (*indicator S x*) ⟷ (¬ (*x* ∈ *S* ∧ ¬ *P 1* ∨ *x* ∉ *S* ∧ ¬ *P 0*))
⟨*proof*⟩

lemma *indicator-inter-arith*: *indicator* (*A* ∩ *B*) *x* = *indicator A x* ∗ (*indicator B x*::'*a*::*semiring-1*)
⟨*proof*⟩

lemma *indicator-union-arith*: *indicator* (*A* ∪ *B*) *x* = *indicator A x* + *indicator B x* − *indicator A x* ∗ (*indicator B x*::'*a*::*ring-1*)
⟨*proof*⟩

lemma *indicator-inter-min*: *indicator* (*A* ∩ *B*) *x* = *min* (*indicator A x*) (*indicator B x*::'*a*::*linordered-semidom*)
and *indicator-union-max*: *indicator* (*A* ∪ *B*) *x* = *max* (*indicator A x*) (*indicator B x*::'*a*::*linordered-semidom*)
⟨*proof*⟩

lemma *indicator-disj-union*: *A* ∩ *B* = {} ⟹ *indicator* (*A* ∪ *B*) *x* = (*indicator A x* + *indicator B x*::'*a*::*linordered-semidom*)
⟨*proof*⟩

lemma *indicator-compl*: *indicator* (− *A*) *x* = *1* − (*indicator A x*::'*a*::*ring-1*)

and *indicator-diff*: *indicator* $(A - B)\ x = indicator\ A\ x * (1 - indicator\ B\ x::'a::ring\text{-}1)$
⟨*proof*⟩

lemma *indicator-times*: *indicator* $(A \times B)\ x = indicator\ A\ (fst\ x) * (indicator\ B\ (snd\ x)::'a::semiring\text{-}1)$
⟨*proof*⟩

lemma *indicator-sum*: *indicator* $(A <+> B)\ x = (case\ x\ of\ Inl\ x \Rightarrow indicator\ A\ x \mid Inr\ x \Rightarrow indicator\ B\ x)$
⟨*proof*⟩

lemma *indicator-image*: $inj\ f \Longrightarrow indicator\ (f\ `\ X)\ (f\ x) = (indicator\ X\ x::-::zero\text{-}neq\text{-}one)$
⟨*proof*⟩

lemma *indicator-vimage*: *indicator* $(f\ -`\ A)\ x = indicator\ A\ (f\ x)$
⟨*proof*⟩

lemma
fixes $f :: 'a \Rightarrow 'b::semiring\text{-}1$ **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*]: $(\sum x \in A.\ indicator\ B\ x * f\ x) = (\sum x \in A \cap B.\ f\ x)$
⟨*proof*⟩

lemma *setsum-indicator-eq-card*:
assumes *finite A*
shows $(\sum x \in A.\ indicator\ B\ x) = card\ (A\ Int\ B)$
⟨*proof*⟩

lemma *setsum-indicator-scaleR*[*simp*]:
finite A $\Longrightarrow$
$(\sum x \in A.\ indicator\ (B\ x)\ (g\ x)\ *_R\ f\ x) = (\sum x \in \{x \in A.\ g\ x \in B\ x\}.\ f\ x::'a::real\text{-}vector)$
⟨*proof*⟩

lemma *LIMSEQ-indicator-incseq*:
assumes *incseq A*
shows $(\lambda i.\ indicator\ (A\ i)\ x :: 'a :: \{topological\text{-}space, one, zero\}) \longrightarrow indicator\ (\bigcup i.\ A\ i)\ x$
⟨*proof*⟩

lemma *LIMSEQ-indicator-UN*:
$(\lambda k.\ indicator\ (\bigcup i<k.\ A\ i)\ x :: 'a :: \{topological\text{-}space, one, zero\}) \longrightarrow indicator\ (\bigcup i.\ A\ i)\ x$
⟨*proof*⟩

lemma *LIMSEQ-indicator-decseq*:

assumes *decseq A*
shows $(\lambda i.\ \mathit{indicator}\ (A\ i)\ x :: 'a :: \{\mathit{topological\text{-}space}, \mathit{one}, \mathit{zero}\}) \longrightarrow \mathit{indicator}\ (\bigcap i.\ A\ i)\ x$
⟨*proof*⟩

lemma *LIMSEQ-indicator-INT*:
$(\lambda k.\ \mathit{indicator}\ (\bigcap i{<}k.\ A\ i)\ x :: 'a :: \{\mathit{topological\text{-}space}, \mathit{one}, \mathit{zero}\}) \longrightarrow \mathit{indicator}\ (\bigcap i.\ A\ i)\ x$
⟨*proof*⟩

lemma *indicator-add*:
$A \cap B = \{\} \Longrightarrow (\mathit{indicator}\ A\ x ::\text{-}::\mathit{monoid\text{-}add}) + \mathit{indicator}\ B\ x = \mathit{indicator}\ (A \cup B)\ x$
⟨*proof*⟩

lemma *of-real-indicator*: $\mathit{of\text{-}real}\ (\mathit{indicator}\ A\ x) = \mathit{indicator}\ A\ x$
⟨*proof*⟩

lemma *real-of-nat-indicator*: $\mathit{real}\ (\mathit{indicator}\ A\ x :: \mathit{nat}) = \mathit{indicator}\ A\ x$
⟨*proof*⟩

lemma *abs-indicator*: $|\mathit{indicator}\ A\ x :: 'a ::\mathit{linordered\text{-}idom}| = \mathit{indicator}\ A\ x$
⟨*proof*⟩

lemma *mult-indicator-subset*:
$A \subseteq B \Longrightarrow \mathit{indicator}\ A\ x * \mathit{indicator}\ B\ x = (\mathit{indicator}\ A\ x :: 'a ::\{\mathit{comm\text{-}semiring\text{-}1}\})$
⟨*proof*⟩

lemma *indicator-sums*:
assumes $\bigwedge i\ j.\ i \neq j \Longrightarrow A\ i \cap A\ j = \{\}$
shows $(\lambda i.\ \mathit{indicator}\ (A\ i)\ x ::\mathit{real})\ \mathit{sums}\ \mathit{indicator}\ (\bigcup i.\ A\ i)\ x$
⟨*proof*⟩

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 $b \neq (-\infty ::\mathit{ereal})\ a \geq b$
shows $a = b + (a - b)$
⟨*proof*⟩

lemma *Limsup-const-add*:
fixes $c :: 'a ::\{\mathit{complete\text{-}linorder}, \mathit{linorder\text{-}topology}, \mathit{topological\text{-}monoid\text{-}add}, \mathit{ordered\text{-}ab\text{-}semigroup\text{-}add}\}$
shows $F \neq \mathit{bot} \Longrightarrow \mathit{Limsup}\ F\ (\lambda x.\ c + f\ x) = c + \mathit{Limsup}\ F\ f$

⟨*proof*⟩

lemma *Liminf-const-add*:
fixes $c :: 'a$::{*complete-linorder*, *linorder-topology*, *topological-monoid-add*, *ordered-ab-semigroup-add*}
shows $F \neq bot \Longrightarrow Liminf\ F\ (\lambda x.\ c + f\ x) = c + Liminf\ F\ f$
⟨*proof*⟩

lemma *Liminf-add-const*:
fixes $c :: 'a$::{*complete-linorder*, *linorder-topology*, *topological-monoid-add*, *ordered-ab-semigroup-add*}
shows $F \neq bot \Longrightarrow Liminf\ F\ (\lambda x.\ f\ x + c) = Liminf\ F\ f + c$
⟨*proof*⟩

lemma *sums-offset*:
fixes $f\ g :: nat \Rightarrow 'a$ :: {*t2-space*, *topological-comm-monoid-add*}
assumes $(\lambda n.\ f\ (n + i))$ *sums* l **shows** f *sums* $(l + (\sum j{<}i.\ f\ j))$
⟨*proof*⟩

lemma *suminf-offset*:
fixes $f\ g :: nat \Rightarrow 'a$ :: {*t2-space*, *topological-comm-monoid-add*}
shows *summable* $(\lambda j.\ f\ (j + i)) \Longrightarrow suminf\ f = (\sum j.\ f\ (j + i)) + (\sum j{<}i.\ f\ j)$
⟨*proof*⟩

lemma *eventually-at-left-1*: $(\bigwedge z$::*real*. $0 < z \Longrightarrow z < 1 \Longrightarrow P\ z) \Longrightarrow$ *eventually* P (*at-left 1*)
⟨*proof*⟩

lemma *mult-eq-1*:
fixes $a\ b :: 'a$ :: {*ordered-semiring*, *comm-monoid-mult*}
shows $0 \leq a \Longrightarrow a \leq 1 \Longrightarrow b \leq 1 \Longrightarrow a * b = 1 \longleftrightarrow (a = 1 \wedge b = 1)$
⟨*proof*⟩

lemma *ereal-add-diff-cancel*:
fixes $a\ b ::$ *ereal*
shows $|b| \neq \infty \Longrightarrow (a + b) - b = a$
⟨*proof*⟩

lemma *add-top*:
fixes $x :: 'a$::{*order-top*, *ordered-comm-monoid-add*}
shows $0 \leq x \Longrightarrow x + top = top$
⟨*proof*⟩

lemma *top-add*:
fixes $x :: 'a$::{*order-top*, *ordered-comm-monoid-add*}
shows $0 \leq x \Longrightarrow top + x = top$
⟨*proof*⟩

lemma *le-lfp*: *mono* $f \Longrightarrow x \leq lfp\ f \Longrightarrow f\ x \leq lfp\ f$
⟨*proof*⟩

lemma *lfp-transfer*:
assumes α: *sup-continuous* α **and** f: *sup-continuous* f **and** *mg*: *mono* g
assumes *bot*: α *bot* $\leq$ *lfp* g **and** *eq*: $\bigwedge x.\ x \leq$ *lfp* $f \Longrightarrow \alpha\ (f\ x) = g\ (\alpha\ x)$
shows α (*lfp* f) = *lfp* g
⟨*proof*⟩

lemma *sup-continuous-applyD*: *sup-continuous* $f \Longrightarrow$ *sup-continuous* $(\lambda x.\ f\ x\ h)$
⟨*proof*⟩

lemma *sup-continuous-SUP*[*order-continuous-intros*]:
fixes M :: - ⇒ - ⇒ 'a::*complete-lattice*
assumes M: $\bigwedge i.\ i \in I \Longrightarrow$ *sup-continuous* $(M\ i)$
shows *sup-continuous* $(SUP\ i{:}I.\ M\ i)$
⟨*proof*⟩

lemma *sup-continuous-apply-SUP*[*order-continuous-intros*]:
fixes M :: - ⇒ - ⇒ 'a::*complete-lattice*
shows $(\bigwedge i.\ i \in I \Longrightarrow$ *sup-continuous* $(M\ i)) \Longrightarrow$ *sup-continuous* $(\lambda x.\ SUP\ i{:}I.\ M\ i\ x)$
⟨*proof*⟩

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*⟩

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* $(\lambda s.\ f\ s\ (g\ s))$
shows *sup-continuous* $(\lambda x.$ *lfp* $(f\ x))$
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *continuous-on-max*:
fixes $f\ g$ :: 'a::*topological-space* ⇒ 'b::*linorder-topology*
shows *continuous-on* $A\ f \Longrightarrow$ *continuous-on* $A\ g \Longrightarrow$ *continuous-on* A $(\lambda x.$ *max* $(f\ x)\ (g\ x))$
⟨*proof*⟩

lemma *continuous-on-cmult-ereal*:
$|c{::}ereal| \neq \infty \Longrightarrow$ *continuous-on* $A\ f \Longrightarrow$ *continuous-on* A $(\lambda x.\ c * f\ x)$
⟨*proof*⟩

context *linordered-nonzero-semiring*

begin

lemma *of-nat-nonneg* [*simp*]: $0 \leq$ *of-nat n*
⟨*proof*⟩

lemma *of-nat-mono*[*simp*]: $i \leq j \Longrightarrow$ *of-nat i* $\leq$ *of-nat j*
⟨*proof*⟩

end

lemma *real-of-nat-Sup*:
assumes $A \neq \{\}$ *bdd-above A*
shows *of-nat* (*Sup A*) = (*SUP a*:*A*. *of-nat a* :: *real*)
⟨*proof*⟩

lemma *of-nat-less*[*simp*]:
$i < j \Longrightarrow$ *of-nat i* < (*of-nat j*::'*a*::{*linordered-nonzero-semiring*, *semiring-char-0*})
⟨*proof*⟩

lemma *of-nat-le-iff*[*simp*]:
of-nat i $\leq$ (*of-nat j*::'*a*::{*linordered-nonzero-semiring*, *semiring-char-0*}) $\longleftrightarrow$ *i* $\leq$ *j*
⟨*proof*⟩

lemma (**in** *complete-lattice*) *SUP-sup-const1*:
$I \neq \{\} \Longrightarrow$ (*SUP i*:*I*. *sup c* (*f i*)) = *sup c* (*SUP i*:*I*. *f i*)
⟨*proof*⟩

lemma (**in** *complete-lattice*) *SUP-sup-const2*:
$I \neq \{\} \Longrightarrow$ (*SUP i*:*I*. *sup* (*f i*) *c*) = *sup* (*SUP i*:*I*. *f i*) *c*
⟨*proof*⟩

lemma *one-less-of-natD*:
(1::'*a*::*linordered-semidom*) < *of-nat n* $\Longrightarrow$ 1 < *n*
⟨*proof*⟩

lemma *setsum-le-suminf*:
fixes *f* :: *nat* $\Rightarrow$ '*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*
⟨*proof*⟩

34.2 Defining the extended non-negative reals

Basic definitions and type class setup

typedef *ennreal* = {*x* :: *ereal*. 0 $\leq$ *x*}
morphisms *enn2ereal e2ennreal'*
⟨*proof*⟩

definition *e2ennreal x = e2ennreal′ (max 0 x)*

lemma *enn2ereal-range*: *e2ennreal ‘ {0..} = UNIV*
⟨*proof*⟩

lemma *type-definition-ennreal′*: *type-definition enn2ereal e2ennreal {x. 0 ≤ x}*
⟨*proof*⟩

setup-lifting *type-definition-ennreal′*

declare [[*coercion e2ennreal*]]

instantiation *ennreal* :: *complete-linorder*
begin

lift-definition *top-ennreal* :: *ennreal* **is** *top* ⟨*proof*⟩
lift-definition *bot-ennreal* :: *ennreal* **is** *0* ⟨*proof*⟩
lift-definition *sup-ennreal* :: *ennreal ⇒ ennreal ⇒ ennreal* **is** *sup* ⟨*proof*⟩
lift-definition *inf-ennreal* :: *ennreal ⇒ ennreal ⇒ ennreal* **is** *inf* ⟨*proof*⟩

lift-definition *Inf-ennreal* :: *ennreal set ⇒ ennreal* **is** *Inf*
⟨*proof*⟩

lift-definition *Sup-ennreal* :: *ennreal set ⇒ ennreal* **is** *sup 0 ∘ Sup*
⟨*proof*⟩

lift-definition *less-eq-ennreal* :: *ennreal ⇒ ennreal ⇒ bool* **is** *op ≤* ⟨*proof*⟩
lift-definition *less-ennreal* :: *ennreal ⇒ ennreal ⇒ bool* **is** *op <* ⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *pcr-ennreal-enn2ereal*[*simp*]: *pcr-ennreal (enn2ereal x) x*
⟨*proof*⟩

lemma *rel-fun-eq-pcr-ennreal*: *rel-fun op = pcr-ennreal f g ⟷ f = enn2ereal ∘ g*
⟨*proof*⟩

instantiation *ennreal* :: *infinity*
begin

definition *infinity-ennreal* :: *ennreal*
where
[*simp*]: *∞ = (top*::*ennreal)*

instance ⟨*proof*⟩

end

instantiation *ennreal* :: {*semiring-1-no-zero-divisors*, *comm-semiring-1*}
begin

lift-definition *one-ennreal* :: *ennreal* **is** *1* ⟨*proof*⟩
lift-definition *zero-ennreal* :: *ennreal* **is** *0* ⟨*proof*⟩
lift-definition *plus-ennreal* :: *ennreal* ⇒ *ennreal* ⇒ *ennreal* **is** *op +* ⟨*proof*⟩
lift-definition *times-ennreal* :: *ennreal* ⇒ *ennreal* ⇒ *ennreal* **is** *op ∗* ⟨*proof*⟩

instance
⟨*proof*⟩

end

instantiation *ennreal* :: *minus*
begin

lift-definition *minus-ennreal* :: *ennreal* ⇒ *ennreal* ⇒ *ennreal* **is** *λa b. max 0* (*a − b*)
⟨*proof*⟩

instance ⟨*proof*⟩

end

instance *ennreal* :: *numeral* ⟨*proof*⟩

instantiation *ennreal* :: *inverse*
begin

lift-definition *inverse-ennreal* :: *ennreal* ⇒ *ennreal* **is** *inverse*
⟨*proof*⟩

definition *divide-ennreal* :: *ennreal* ⇒ *ennreal* ⇒ *ennreal*
where *x div y = x ∗ inverse* (*y* :: *ennreal*)

instance ⟨*proof*⟩

end

lemma *ennreal-zero-less-one*: *0 <* (*1*::*ennreal*) — TODO: remove
⟨*proof*⟩

instance *ennreal* :: *dioid*
⟨*proof*⟩

instance *ennreal* :: *ordered-comm-semiring*
⟨*proof*⟩

instance *ennreal* :: *linordered-nonzero-semiring*
⟨*proof*⟩

instance *ennreal* :: *strict-ordered-ab-semigroup-add*
⟨*proof*⟩

declare [[*coercion of-nat* :: *nat* ⇒ *ennreal*]]

lemma *e2ennreal-neg*: $x \leq 0 \Longrightarrow e2ennreal\ x = 0$
⟨*proof*⟩

lemma *e2ennreal-mono*: $x \leq y \Longrightarrow e2ennreal\ x \leq e2ennreal\ y$
⟨*proof*⟩

lemma *enn2ereal-nonneg*[*simp*]: $0 \leq enn2ereal\ x$
⟨*proof*⟩

lemma *ereal-ennreal-cases*:
obtains b **where** $0 \leq a\ a = enn2ereal\ b \mid a < 0$
⟨*proof*⟩

lemma *rel-fun-liminf*[*transfer-rule*]: *rel-fun* (*rel-fun op* = *pcr-ennreal*) *pcr-ennreal liminf liminf*
⟨*proof*⟩

lemma *rel-fun-limsup*[*transfer-rule*]: *rel-fun* (*rel-fun op* = *pcr-ennreal*) *pcr-ennreal limsup limsup*
⟨*proof*⟩

lemma *setsum-enn2ereal*[*simp*]: $(\bigwedge i.\ i \in I \Longrightarrow 0 \leq f\ i) \Longrightarrow (\sum i{\in}I.\ enn2ereal\ (f\ i)) = enn2ereal\ (setsum\ f\ I)$
⟨*proof*⟩

lemma *transfer-e2ennreal-setsum* [*transfer-rule*]:
rel-fun (*rel-fun op* = *pcr-ennreal*) (*rel-fun op* = *pcr-ennreal*) *setsum setsum*
⟨*proof*⟩

lemma *enn2ereal-of-nat*[*simp*]: *enn2ereal* (*of-nat n*) = *ereal n*
⟨*proof*⟩

lemma *enn2ereal-numeral*[*simp*]: *enn2ereal* (*numeral a*) = *numeral a*
⟨*proof*⟩

lemma *transfer-numeral*[*transfer-rule*]: *pcr-ennreal* (*numeral a*) (*numeral a*)
⟨*proof*⟩

34.3 Cancellation simprocs

lemma *ennreal-add-left-cancel*: $a + b = a + c \longleftrightarrow a = (\infty::ennreal) \vee b = c$
⟨*proof*⟩

lemma *ennreal-add-left-cancel-le*: $a + b \leq a + c \longleftrightarrow a = (\infty::ennreal) \vee b \leq c$
⟨*proof*⟩

lemma *ereal-add-left-cancel-less*:
fixes $a\ b\ c :: ereal$
shows $0 \leq a \Longrightarrow 0 \leq b \Longrightarrow a + b < a + c \longleftrightarrow a \neq \infty \wedge b < c$
⟨*proof*⟩

lemma *ennreal-add-left-cancel-less*: $a + b < a + c \longleftrightarrow a \neq (\infty::ennreal) \wedge b < c$
⟨*proof*⟩

⟨*ML*⟩

34.4 Order with top

lemma *ennreal-zero-less-top*[*simp*]: $0 < (top::ennreal)$
⟨*proof*⟩

lemma *ennreal-one-less-top*[*simp*]: $1 < (top::ennreal)$
⟨*proof*⟩

lemma *ennreal-zero-neq-top*[*simp*]: $0 \neq (top::ennreal)$
⟨*proof*⟩

lemma *ennreal-top-neq-zero*[*simp*]: $(top::ennreal) \neq 0$
⟨*proof*⟩

lemma *ennreal-top-neq-one*[*simp*]: $top \neq (1::ennreal)$
⟨*proof*⟩

lemma *ennreal-one-neq-top*[*simp*]: $1 \neq (top::ennreal)$
⟨*proof*⟩

lemma *ennreal-add-less-top*[*simp*]:
fixes $a\ b :: ennreal$
shows $a + b < top \longleftrightarrow a < top \wedge b < top$
⟨*proof*⟩

lemma *ennreal-add-eq-top*[*simp*]:
fixes $a\ b :: ennreal$
shows $a + b = top \longleftrightarrow a = top \vee b = top$
⟨*proof*⟩

lemma *ennreal-setsum-less-top*[*simp*]:

fixes $f :: 'a \Rightarrow ennreal$
shows $finite\ I \Longrightarrow (\sum i{\in}I.\ f\ i) < top \longleftrightarrow (\forall\, i{\in}I.\ f\ i < top)$
⟨*proof*⟩

lemma *ennreal-setsum-eq-top*[*simp*]:
fixes $f :: 'a \Rightarrow ennreal$
shows $finite\ I \Longrightarrow (\sum i{\in}I.\ f\ i) = top \longleftrightarrow (\exists\, i{\in}I.\ f\ i = top)$
⟨*proof*⟩

lemma *ennreal-mult-eq-top-iff*:
fixes $a\ b :: ennreal$
shows $a * b = top \longleftrightarrow (a = top \wedge b \neq 0) \vee (b = top \wedge a \neq 0)$
⟨*proof*⟩

lemma *ennreal-top-eq-mult-iff*:
fixes $a\ b :: ennreal$
shows $top = a * b \longleftrightarrow (a = top \wedge b \neq 0) \vee (b = top \wedge a \neq 0)$
⟨*proof*⟩

lemma *ennreal-mult-less-top*:
fixes $a\ b :: ennreal$
shows $a * b < top \longleftrightarrow (a = 0 \vee b = 0 \vee (a < top \wedge b < top))$
⟨*proof*⟩

lemma *top-power-ennreal*: $top\ \hat{}\ n = (if\ n = 0\ then\ 1\ else\ top :: ennreal)$
⟨*proof*⟩

lemma *ennreal-setprod-eq-0*[*simp*]:
fixes $f :: 'a \Rightarrow ennreal$
shows $(setprod\ f\ A = 0) = (finite\ A \wedge (\exists\, i{\in}A.\ f\ i = 0))$
⟨*proof*⟩

lemma *ennreal-setprod-eq-top*:
fixes $f :: 'a \Rightarrow ennreal$
shows $(\prod i{\in}I.\ f\ i) = top \longleftrightarrow (finite\ I \wedge ((\forall\, i{\in}I.\ f\ i \neq 0) \wedge (\exists\, i{\in}I.\ f\ i = top)))$
⟨*proof*⟩

lemma *ennreal-top-mult*: $top * a = (if\ a = 0\ then\ 0\ else\ top :: ennreal)$
⟨*proof*⟩

lemma *ennreal-mult-top*: $a * top = (if\ a = 0\ then\ 0\ else\ top :: ennreal)$
⟨*proof*⟩

lemma *enn2ereal-eq-top-iff*[*simp*]: $enn2ereal\ x = \infty \longleftrightarrow x = top$
⟨*proof*⟩

lemma *enn2ereal-top*: $enn2ereal\ top = \infty$
⟨*proof*⟩

lemma *e2ennreal-infty*: $e2ennreal\ \infty = top$
⟨*proof*⟩

lemma *ennreal-top-minus*[*simp*]: $top - x = (top::ennreal)$
⟨*proof*⟩

lemma *minus-top-ennreal*: $x - top = (if\ x = top\ then\ top\ else\ 0::\ ennreal)$
⟨*proof*⟩

lemma *bot-ennreal*: $bot = (0::ennreal)$
⟨*proof*⟩

lemma *ennreal-of-nat-neq-top*[*simp*]: $of\text{-}nat\ i \neq (top::ennreal)$
⟨*proof*⟩

lemma *numeral-eq-of-nat*: $(numeral\ a::ennreal) = of\text{-}nat\ (numeral\ a)$
⟨*proof*⟩

lemma *of-nat-less-top*: $of\text{-}nat\ i < (top::ennreal)$
⟨*proof*⟩

lemma *top-neq-numeral*[*simp*]: $top \neq (numeral\ i::ennreal)$
⟨*proof*⟩

lemma *ennreal-numeral-less-top*[*simp*]: $numeral\ i < (top::ennreal)$
⟨*proof*⟩

lemma *ennreal-add-bot*[*simp*]: $bot + x = (x::ennreal)$
⟨*proof*⟩

instance *ennreal* :: *semiring-char-0*
⟨*proof*⟩

34.5 Arithmetic

lemma *ennreal-minus-zero*[*simp*]: $a - (0::ennreal) = a$
⟨*proof*⟩

lemma *ennreal-add-diff-cancel-right*[*simp*]:
fixes $x\ y\ z$ :: *ennreal* **shows** $y \neq top \Longrightarrow (x + y) - y = x$
⟨*proof*⟩

lemma *ennreal-add-diff-cancel-left*[*simp*]:
fixes $x\ y\ z$ :: *ennreal* **shows** $y \neq top \Longrightarrow (y + x) - y = x$
⟨*proof*⟩

lemma
fixes $a\ b$ :: *ennreal*
shows $a - b = 0 \Longrightarrow a \leq b$

⟨*proof*⟩

lemma *ennreal-minus-cancel*:
fixes $a\ b\ c$ :: *ennreal*
shows $c \neq top \Longrightarrow a \leq c \Longrightarrow b \leq c \Longrightarrow c - a = c - b \Longrightarrow a = b$
⟨*proof*⟩

lemma *sup-const-add-ennreal*:
fixes $a\ b\ c$ :: *ennreal*
shows $sup\ (c + a)\ (c + b) = c + sup\ a\ b$
⟨*proof*⟩

lemma *ennreal-diff-add-assoc*:
fixes $a\ b\ c$ :: *ennreal*
shows $a \leq b \Longrightarrow c + b - a = c + (b - a)$
⟨*proof*⟩

lemma *mult-divide-eq-ennreal*:
fixes $a\ b$ :: *ennreal*
shows $b \neq 0 \Longrightarrow b \neq top \Longrightarrow (a * b)\ /\ b = a$
⟨*proof*⟩

lemma *divide-mult-eq*: $a \neq 0 \Longrightarrow a \neq \infty \Longrightarrow x * a\ /\ (b * a) = x\ /\ (b{::}ennreal)$
⟨*proof*⟩

lemma *ennreal-mult-divide-eq*:
fixes $a\ b$ :: *ennreal*
shows $b \neq 0 \Longrightarrow b \neq top \Longrightarrow (a * b)\ /\ b = a$
⟨*proof*⟩

lemma *ennreal-add-diff-cancel*:
fixes $a\ b$ :: *ennreal*
shows $b \neq \infty \Longrightarrow (a + b) - b = a$
⟨*proof*⟩

lemma *ennreal-minus-eq-0*:
$a - b = 0 \Longrightarrow a \leq (b{::}ennreal)$
⟨*proof*⟩

lemma *ennreal-mono-minus-cancel*:
fixes $a\ b\ c$ :: *ennreal*
shows $a - b \leq a - c \Longrightarrow a < top \Longrightarrow b \leq a \Longrightarrow c \leq a \Longrightarrow c \leq b$
⟨*proof*⟩

lemma *ennreal-mono-minus*:
fixes $a\ b\ c$ :: *ennreal*
shows $c \leq b \Longrightarrow a - b \leq a - c$
⟨*proof*⟩

lemma *ennreal-minus-pos-iff*:
fixes *a b* :: *ennreal*
shows $a < top \vee b < top \Longrightarrow 0 < a - b \Longrightarrow b < a$
⟨*proof*⟩

lemma *ennreal-inverse-top*[*simp*]: *inverse top* = (*0*::*ennreal*)
⟨*proof*⟩

lemma *ennreal-inverse-zero*[*simp*]: *inverse 0* = (*top*::*ennreal*)
⟨*proof*⟩

lemma *ennreal-top-divide*: *top* / (*x*::*ennreal*) = (*if x* = *top then 0 else top*)
⟨*proof*⟩

lemma *ennreal-zero-divide*[*simp*]: *0* / (*x*::*ennreal*) = *0*
⟨*proof*⟩

lemma *ennreal-divide-zero*[*simp*]: *x* / (*0*::*ennreal*) = (*if x* = *0 then 0 else top*)
⟨*proof*⟩

lemma *ennreal-divide-top*[*simp*]: *x* / (*top*::*ennreal*) = *0*
⟨*proof*⟩

lemma *ennreal-times-divide*: *a* ∗ (*b* / *c*) = *a* ∗ *b* / (*c*::*ennreal*)
⟨*proof*⟩

lemma *ennreal-zero-less-divide*: *0* < *a* / *b* ⟷ (*0* < *a* ∧ *b* < (*top*::*ennreal*))
⟨*proof*⟩

lemma *divide-right-mono-ennreal*:
fixes *a b c* :: *ennreal*
shows $a \leq b \Longrightarrow a \,/\, c \leq b \,/\, c$
⟨*proof*⟩

lemma *ennreal-mult-strict-right-mono*: (*a*::*ennreal*) < *c* ⟹ *0* < *b* ⟹ *b* < *top*
⟹ *a* ∗ *b* < *c* ∗ *b*
⟨*proof*⟩

lemma *ennreal-indicator-less*[*simp*]:
indicator A x ≤ (*indicator B x*::*ennreal*) ⟷ (*x* ∈ *A* ⟶ *x* ∈ *B*)
⟨*proof*⟩

lemma *ennreal-inverse-positive*: *0* < *inverse x* ⟷ (*x*::*ennreal*) ≠ *top*
⟨*proof*⟩

lemma *ennreal-inverse-mult'*: ((*0* < *b* ∨ *a* < *top*) ∧ (*0* < *a* ∨ *b* < *top*)) ⟹
inverse (*a* ∗ *b*::*ennreal*) = *inverse a* ∗ *inverse b*
⟨*proof*⟩

lemma *ennreal-inverse-mult*: $a < top \Longrightarrow b < top \Longrightarrow inverse\ (a * b::ennreal) = inverse\ a * inverse\ b$
⟨*proof*⟩

lemma *ennreal-inverse-1*[*simp*]: $inverse\ (1::ennreal) = 1$
⟨*proof*⟩

lemma *ennreal-inverse-eq-0-iff*[*simp*]: $inverse\ (a::ennreal) = 0 \longleftrightarrow a = top$
⟨*proof*⟩

lemma *ennreal-inverse-eq-top-iff*[*simp*]: $inverse\ (a::ennreal) = top \longleftrightarrow a = 0$
⟨*proof*⟩

lemma *ennreal-divide-eq-0-iff*[*simp*]: $(a::ennreal)\ /\ b = 0 \longleftrightarrow (a = 0 \vee b = top)$
⟨*proof*⟩

lemma *ennreal-divide-eq-top-iff*: $(a::ennreal)\ /\ b = top \longleftrightarrow ((a \neq 0 \wedge b = 0) \vee (a = top \wedge b \neq top))$
⟨*proof*⟩

lemma *one-divide-one-divide-ennreal*[*simp*]: $1\ /\ (1\ /\ c) = (c::ennreal)$
including *ennreal.lifting*
⟨*proof*⟩

lemma *ennreal-mult-left-cong*:
$((a::ennreal) \neq 0 \Longrightarrow b = c) \Longrightarrow a * b = a * c$
⟨*proof*⟩

lemma *ennreal-mult-right-cong*:
$((a::ennreal) \neq 0 \Longrightarrow b = c) \Longrightarrow b * a = c * a$
⟨*proof*⟩

lemma *ennreal-zero-less-mult-iff*: $0 < a * b \longleftrightarrow 0 < a \wedge 0 < (b::ennreal)$
⟨*proof*⟩

lemma *less-diff-eq-ennreal*:
fixes $a\ b\ c :: ennreal$
shows $b < top \vee c < top \Longrightarrow a < b - c \longleftrightarrow a + c < b$
⟨*proof*⟩

lemma *diff-add-cancel-ennreal*:
fixes $a\ b :: ennreal$ **shows** $a \leq b \Longrightarrow b - a + a = b$
⟨*proof*⟩

lemma *ennreal-diff-self*[*simp*]: $a \neq top \Longrightarrow a - a = (0::ennreal)$
⟨*proof*⟩

lemma *ennreal-minus-mono*:
fixes $a\ b\ c :: ennreal$

shows $a \leq c \Longrightarrow d \leq b \Longrightarrow a - b \leq c - d$
⟨*proof*⟩

lemma *ennreal-minus-eq-top*[*simp*]: $a - (b::ennreal) = top \longleftrightarrow a = top$
⟨*proof*⟩

lemma *ennreal-divide-self*[*simp*]: $a \neq 0 \Longrightarrow a < top \Longrightarrow a \mathbin{/} a = (1::ennreal)$
⟨*proof*⟩

34.6 Coercion from *real* to *ennreal*

lift-definition *ennreal* :: *real* ⇒ *ennreal* **is** *sup 0* ∘ *ereal*
⟨*proof*⟩

declare [[*coercion ennreal*]]

lemma *ennreal-cases*[*cases type*: *ennreal*]:
fixes x :: *ennreal*
obtains (*real*) r :: *real* **where** $0 \leq r$ $x = ennreal\ r$ | (*top*) $x = top$
⟨*proof*⟩

lemmas *ennreal2-cases* = *ennreal-cases*[*case-product ennreal-cases*]
lemmas *ennreal3-cases* = *ennreal-cases*[*case-product ennreal2-cases*]

lemma *ennreal-neq-top*[*simp*]: $ennreal\ r \neq top$
⟨*proof*⟩

lemma *top-neq-ennreal*[*simp*]: $top \neq ennreal\ r$
⟨*proof*⟩

lemma *ennreal-less-top*[*simp*]: $ennreal\ x < top$
⟨*proof*⟩

lemma *ennreal-neg*: $x \leq 0 \Longrightarrow ennreal\ x = 0$
⟨*proof*⟩

lemma *ennreal-inj*[*simp*]:
$0 \leq a \Longrightarrow 0 \leq b \Longrightarrow ennreal\ a = ennreal\ b \longleftrightarrow a = b$
⟨*proof*⟩

lemma *ennreal-le-iff*[*simp*]: $0 \leq y \Longrightarrow ennreal\ x \leq ennreal\ y \longleftrightarrow x \leq y$
⟨*proof*⟩

lemma *le-ennreal-iff*: $0 \leq r \Longrightarrow x \leq ennreal\ r \longleftrightarrow (\exists\, q{\geq}0.\ x = ennreal\ q \wedge q \leq r)$
⟨*proof*⟩

lemma *ennreal-less-iff*: $0 \leq r \Longrightarrow ennreal\ r < ennreal\ q \longleftrightarrow r < q$
⟨*proof*⟩

lemma *ennreal-eq-zero-iff* [*simp*]: $0 \leq x \Longrightarrow ennreal\ x = 0 \longleftrightarrow x = 0$
⟨*proof*⟩

lemma *ennreal-less-zero-iff* [*simp*]: $0 < ennreal\ x \longleftrightarrow 0 < x$
⟨*proof*⟩

lemma *ennreal-lessI*: $0 < q \Longrightarrow r < q \Longrightarrow ennreal\ r < ennreal\ q$
⟨*proof*⟩

lemma *ennreal-leI*: $x \leq y \Longrightarrow ennreal\ x \leq ennreal\ y$
⟨*proof*⟩

lemma *enn2ereal-ennreal*[*simp*]: $0 \leq x \Longrightarrow enn2ereal\ (ennreal\ x) = x$
⟨*proof*⟩

lemma *e2ennreal-enn2ereal*[*simp*]: $e2ennreal\ (enn2ereal\ x) = x$
⟨*proof*⟩

lemma *ennreal-0* [*simp*]: $ennreal\ 0 = 0$
⟨*proof*⟩

lemma *ennreal-1* [*simp*]: $ennreal\ 1 = 1$
⟨*proof*⟩

lemma *ennreal-eq-0-iff*: $ennreal\ x = 0 \longleftrightarrow x \leq 0$
⟨*proof*⟩

lemma *ennreal-le-iff2*: $ennreal\ x \leq ennreal\ y \longleftrightarrow ((0 \leq y \wedge x \leq y) \vee (x \leq 0 \wedge y \leq 0))$
⟨*proof*⟩

lemma *ennreal-eq-1* [*simp*]: $ennreal\ x = 1 \longleftrightarrow x = 1$
⟨*proof*⟩

lemma *ennreal-le-1* [*simp*]: $ennreal\ x \leq 1 \longleftrightarrow x \leq 1$
⟨*proof*⟩

lemma *ennreal-ge-1* [*simp*]: $ennreal\ x \geq 1 \longleftrightarrow x \geq 1$
⟨*proof*⟩

lemma *ennreal-plus*[*simp*]:
$0 \leq a \Longrightarrow 0 \leq b \Longrightarrow ennreal\ (a + b) = ennreal\ a + ennreal\ b$
⟨*proof*⟩

lemma *setsum-ennreal*[*simp*]: $(\bigwedge i.\ i \in I \Longrightarrow 0 \leq f\ i) \Longrightarrow (\sum i \in I.\ ennreal\ (f\ i)) = ennreal\ (setsum\ f\ I)$
⟨*proof*⟩

lemma *ennreal-of-nat-eq-real-of-nat*: *of-nat i = ennreal (of-nat i)*
⟨*proof*⟩

lemma *of-nat-le-ennreal-iff*[*simp*]: *0 ≤ r ⟹ of-nat i ≤ ennreal r ⟷ of-nat i ≤ r*
⟨*proof*⟩

lemma *ennreal-le-of-nat-iff*[*simp*]: *ennreal r ≤ of-nat i ⟷ r ≤ of-nat i*
⟨*proof*⟩

lemma *ennreal-indicator*: *ennreal (indicator A x) = indicator A x*
⟨*proof*⟩

lemma *ennreal-numeral*[*simp*]: *ennreal (numeral n) = numeral n*
⟨*proof*⟩

lemma *min-ennreal*: *0 ≤ x ⟹ 0 ≤ y ⟹ min (ennreal x) (ennreal y) = ennreal (min x y)*
⟨*proof*⟩

lemma *ennreal-half*[*simp*]: *ennreal (1/2) = inverse 2*
⟨*proof*⟩

lemma *ennreal-minus*: *0 ≤ q ⟹ ennreal r − ennreal q = ennreal (r − q)*
⟨*proof*⟩

lemma *ennreal-minus-top*[*simp*]: *ennreal a − top = 0*
⟨*proof*⟩

lemma *ennreal-mult*: *0 ≤ a ⟹ 0 ≤ b ⟹ ennreal (a * b) = ennreal a * ennreal b*
⟨*proof*⟩

lemma *ennreal-mult'*: *0 ≤ a ⟹ ennreal (a * b) = ennreal a * ennreal b*
⟨*proof*⟩

lemma *indicator-mult-ennreal*: *indicator A x * ennreal r = ennreal (indicator A x * r)*
⟨*proof*⟩

lemma *ennreal-mult''*: *0 ≤ b ⟹ ennreal (a * b) = ennreal a * ennreal b*
⟨*proof*⟩

lemma *numeral-mult-ennreal*: *0 ≤ x ⟹ numeral b * ennreal x = ennreal (numeral b * x)*
⟨*proof*⟩

lemma *ennreal-power*: *0 ≤ r ⟹ ennreal r ^ n = ennreal (r ^ n)*
⟨*proof*⟩

lemma *power-eq-top-ennreal*: $x \hat{} n = top \longleftrightarrow (n \neq 0 \wedge (x::ennreal) = top)$
⟨*proof*⟩

lemma *inverse-ennreal*: $0 < r \Longrightarrow inverse\ (ennreal\ r) = ennreal\ (inverse\ r)$
⟨*proof*⟩

lemma *divide-ennreal*: $0 \leq r \Longrightarrow 0 < q \Longrightarrow ennreal\ r\ /\ ennreal\ q = ennreal\ (r\ /\ q)$
⟨*proof*⟩

lemma *ennreal-inverse-power*: $inverse\ (x \hat{} n :: ennreal) = inverse\ x \hat{} n$
⟨*proof*⟩

lemma *ennreal-divide-numeral*: $0 \leq x \Longrightarrow ennreal\ x\ /\ numeral\ b = ennreal\ (x\ /\ numeral\ b)$
⟨*proof*⟩

lemma *setprod-ennreal*: $(\bigwedge i.\ i \in A \Longrightarrow 0 \leq f\ i) \Longrightarrow (\prod i{\in}A.\ ennreal\ (f\ i)) = ennreal\ (setprod\ f\ A)$
⟨*proof*⟩

lemma *mult-right-ennreal-cancel*: $a * ennreal\ c = b * ennreal\ c \longleftrightarrow (a = b \vee c \leq 0)$
⟨*proof*⟩

lemma *ennreal-le-epsilon*:
$(\bigwedge e::real.\ y < top \Longrightarrow 0 < e \Longrightarrow x \leq y + ennreal\ e) \Longrightarrow x \leq y$
⟨*proof*⟩

lemma *ennreal-rat-dense*:
fixes $x\ y :: ennreal$
shows $x < y \Longrightarrow \exists r::rat.\ x < real\text{-}of\text{-}rat\ r \wedge real\text{-}of\text{-}rat\ r < y$
⟨*proof*⟩

lemma *ennreal-Ex-less-of-nat*: $(x::ennreal) < top \Longrightarrow \exists n.\ x < of\text{-}nat\ n$
⟨*proof*⟩

34.7 Coercion from *ennreal* to *real*

definition $enn2real\ x = real\text{-}of\text{-}ereal\ (enn2ereal\ x)$

lemma *enn2real-nonneg*[*simp*]: $0 \leq enn2real\ x$
⟨*proof*⟩

lemma *enn2real-mono*: $a \leq b \Longrightarrow b < top \Longrightarrow enn2real\ a \leq enn2real\ b$
⟨*proof*⟩

lemma *enn2real-of-nat*[*simp*]: $enn2real\ (of\text{-}nat\ n) = n$

⟨*proof*⟩

lemma *enn2real-ennreal*[*simp*]: $0 \leq r \Longrightarrow$ *enn2real* (*ennreal* r) $= r$
⟨*proof*⟩

lemma *ennreal-enn2real*[*simp*]: $r < \mathit{top} \Longrightarrow$ *ennreal* (*enn2real* r) $= r$
⟨*proof*⟩

lemma *real-of-ereal-enn2ereal*[*simp*]: *real-of-ereal* (*enn2ereal* x) $=$ *enn2real* x
⟨*proof*⟩

lemma *enn2real-top*[*simp*]: *enn2real top* $= 0$
⟨*proof*⟩

lemma *enn2real-0*[*simp*]: *enn2real* $0 = 0$
⟨*proof*⟩

lemma *enn2real-1*[*simp*]: *enn2real* $1 = 1$
⟨*proof*⟩

lemma *enn2real-numeral*[*simp*]: *enn2real* (*numeral* n) $=$ (*numeral* n)
⟨*proof*⟩

lemma *enn2real-mult*: *enn2real* $(a * b) =$ *enn2real* $a *$ *enn2real* b
⟨*proof*⟩

lemma *enn2real-leI*: $0 \leq B \Longrightarrow x \leq$ *ennreal* $B \Longrightarrow$ *enn2real* $x \leq B$
⟨*proof*⟩

lemma *enn2real-positive-iff*: $0 <$ *enn2real* $x \longleftrightarrow (0 < x \wedge x < \mathit{top})$
⟨*proof*⟩

34.8 Coercion from *enat* to *ennreal*

definition *ennreal-of-enat* :: *enat* ⇒ *ennreal*
where
ennreal-of-enat $n =$ (*case* n *of* $\infty \Rightarrow$ *top* | *enat* $n \Rightarrow$ *of-nat* n)

declare [[*coercion ennreal-of-enat*]]
declare [[*coercion of-nat* :: *nat* ⇒ *ennreal*]]

lemma *ennreal-of-enat-infty*[*simp*]: *ennreal-of-enat* $\infty = \infty$
⟨*proof*⟩

lemma *ennreal-of-enat-enat*[*simp*]: *ennreal-of-enat* (*enat* n) $=$ *of-nat* n
⟨*proof*⟩

lemma *ennreal-of-enat-0*[*simp*]: *ennreal-of-enat* $0 = 0$
⟨*proof*⟩

lemma *ennreal-of-enat-1*[*simp*]: *ennreal-of-enat 1 = 1*
⟨*proof*⟩

lemma *ennreal-top-neq-of-nat*[*simp*]: (*top*::*ennreal*) ≠ *of-nat i*
⟨*proof*⟩

lemma *ennreal-of-enat-inj*[*simp*]: *ennreal-of-enat i = ennreal-of-enat j* ⟷ *i = j*
⟨*proof*⟩

lemma *ennreal-of-enat-le-iff*[*simp*]: *ennreal-of-enat m* ≤ *ennreal-of-enat n* ⟷ *m* ≤ *n*
⟨*proof*⟩

lemma *of-nat-less-ennreal-of-nat*[*simp*]: *of-nat n* ≤ *ennreal-of-enat x* ⟷ *of-nat n* ≤ *x*
⟨*proof*⟩

lemma *ennreal-of-enat-Sup*: *ennreal-of-enat* (*Sup X*) = (*SUP x*:*X*. *ennreal-of-enat x*)
⟨*proof*⟩

lemma *ennreal-of-enat-eSuc*[*simp*]: *ennreal-of-enat* (*eSuc x*) = *1* + *ennreal-of-enat x*
⟨*proof*⟩

34.9 Topology on *ennreal*

lemma *enn2ereal-Iio*: *enn2ereal* −‘ {..<*a*} = (*if 0* ≤ *a then* {..< *e2ennreal a*} *else* {})
⟨*proof*⟩

lemma *enn2ereal-Ioi*: *enn2ereal* −‘ {*a* <..} = (*if 0* ≤ *a then* {*e2ennreal a* <..} *else UNIV*)
⟨*proof*⟩

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*⟩

end

lemma *continuous-on-e2ennreal*: *continuous-on A e2ennreal*

⟨*proof*⟩

lemma *continuous-at-e2ennreal*: *continuous* (*at x within A*) *e2ennreal*
⟨*proof*⟩

lemma *continuous-on-enn2ereal*: *continuous-on UNIV enn2ereal*
⟨*proof*⟩

lemma *continuous-at-enn2ereal*: *continuous* (*at x within A*) *enn2ereal*
⟨*proof*⟩

lemma *sup-continuous-e2ennreal*[*order-continuous-intros*]:
assumes *f*: *sup-continuous f* **shows** *sup-continuous* (λ*x*. *e2ennreal* (*f x*))
⟨*proof*⟩

lemma *sup-continuous-enn2ereal*[*order-continuous-intros*]:
assumes *f*: *sup-continuous f* **shows** *sup-continuous* (λ*x*. *enn2ereal* (*f x*))
⟨*proof*⟩

lemma *sup-continuous-mult-left-ennreal'*:
fixes *c* :: *ennreal*
shows *sup-continuous* (λ*x*. *c* ∗ *x*)
⟨*proof*⟩

lemma *sup-continuous-mult-left-ennreal*[*order-continuous-intros*]:
sup-continuous f ⟹ *sup-continuous* (λ*x*. *c* ∗ *f x* :: *ennreal*)
⟨*proof*⟩

lemma *sup-continuous-mult-right-ennreal*[*order-continuous-intros*]:
sup-continuous f ⟹ *sup-continuous* (λ*x*. *f x* ∗ *c* :: *ennreal*)
⟨*proof*⟩

lemma *sup-continuous-divide-ennreal*[*order-continuous-intros*]:
fixes *f g* :: *'a*::*complete-lattice* ⇒ *ennreal*
shows *sup-continuous f* ⟹ *sup-continuous* (λ*x*. *f x* / *c*)
⟨*proof*⟩

lemma *transfer-enn2ereal-continuous-on* [*transfer-rule*]:
rel-fun (*op* =) (*rel-fun* (*rel-fun op* = *pcr-ennreal*) *op* =) *continuous-on continuous-on*
⟨*proof*⟩

lemma *transfer-sup-continuous*[*transfer-rule*]:
(*rel-fun* (*rel-fun* (*op* =) *pcr-ennreal*) *op* =) *sup-continuous sup-continuous*
⟨*proof*⟩

lemma *continuous-on-ennreal*[*tendsto-intros*]:
continuous-on A f ⟹ *continuous-on A* (λ*x*. *ennreal* (*f x*))
⟨*proof*⟩

lemma *tendsto-ennrealD*:
assumes *lim*: $((\lambda x.\ ennreal\ (f\ x)) \longrightarrow ennreal\ x)\ F$
assumes $*$: $\forall_F\ x\ in\ F.\ 0 \le f\ x$ **and** x: $0 \le x$
shows $(f \longrightarrow x)\ F$
⟨*proof*⟩

lemma *tendsto-ennreal-iff*[*simp*]:
$\forall_F\ x\ in\ F.\ 0 \le f\ x \Longrightarrow 0 \le x \Longrightarrow ((\lambda x.\ ennreal\ (f\ x)) \longrightarrow ennreal\ x)\ F \longleftrightarrow (f \longrightarrow x)\ F$
⟨*proof*⟩

lemma *tendsto-enn2ereal-iff*[*simp*]: $((\lambda i.\ enn2ereal\ (f\ i)) \longrightarrow enn2ereal\ x)\ F \longleftrightarrow (f \longrightarrow x)\ F$
⟨*proof*⟩

lemma *continuous-on-add-ennreal*:
fixes $f\ g :: 'a{::}topological\text{-}space \Rightarrow ennreal$
shows *continuous-on A f* $\Longrightarrow$ *continuous-on A g* $\Longrightarrow$ *continuous-on A* $(\lambda x.\ f\ x + g\ x)$
⟨*proof*⟩

lemma *continuous-on-inverse-ennreal*[*continuous-intros*]:
fixes $f :: 'a{::}topological\text{-}space \Rightarrow ennreal$
shows *continuous-on A f* $\Longrightarrow$ *continuous-on A* $(\lambda x.\ inverse\ (f\ x))$
⟨*proof*⟩

instance *ennreal* :: *topological-comm-monoid-add*
⟨*proof*⟩

lemma *sup-continuous-add-ennreal*[*order-continuous-intros*]:
fixes $f\ g :: 'a{::}complete\text{-}lattice \Rightarrow ennreal$
shows *sup-continuous f* $\Longrightarrow$ *sup-continuous g* $\Longrightarrow$ *sup-continuous* $(\lambda x.\ f\ x + g\ x)$
⟨*proof*⟩

lemma *ennreal-suminf-lessD*: $(\sum i.\ f\ i :: ennreal) < x \Longrightarrow f\ i < x$
⟨*proof*⟩

lemma *sums-ennreal*[*simp*]: $(\bigwedge i.\ 0 \le f\ i) \Longrightarrow 0 \le x \Longrightarrow (\lambda i.\ ennreal\ (f\ i))\ sums\ ennreal\ x \longleftrightarrow f\ sums\ x$
⟨*proof*⟩

lemma *summable-suminf-not-top*: $(\bigwedge i.\ 0 \le f\ i) \Longrightarrow (\sum i.\ ennreal\ (f\ i)) \neq top \Longrightarrow summable\ f$
⟨*proof*⟩

lemma *suminf-ennreal*[*simp*]:
$(\bigwedge i.\ 0 \le f\ i) \Longrightarrow (\sum i.\ ennreal\ (f\ i)) \neq top \Longrightarrow (\sum i.\ ennreal\ (f\ i)) = ennreal\ (\sum i.\ f\ i)$

⟨*proof*⟩

lemma *sums-enn2ereal*[*simp*]: $(\lambda i.\ enn2ereal\ (f\ i))\ sums\ enn2ereal\ x \longleftrightarrow f\ sums\ x$
⟨*proof*⟩

lemma *suminf-enn2ereal*[*simp*]: $(\sum i.\ enn2ereal\ (f\ i)) = enn2ereal\ (suminf\ f)$
⟨*proof*⟩

lemma *transfer-e2ennreal-suminf* [*transfer-rule*]: *rel-fun* (*rel-fun op* = *pcr-ennreal*) *pcr-ennreal suminf suminf*
⟨*proof*⟩

lemma *ennreal-suminf-cmult*[*simp*]: $(\sum i.\ r * f\ i) = r * (\sum i.\ f\ i{::}ennreal)$
⟨*proof*⟩

lemma *ennreal-suminf-multc*[*simp*]: $(\sum i.\ f\ i * r) = (\sum i.\ f\ i{::}ennreal) * r$
⟨*proof*⟩

lemma *ennreal-suminf-divide*[*simp*]: $(\sum i.\ f\ i\ /\ r) = (\sum i.\ f\ i{::}ennreal)\ /\ r$
⟨*proof*⟩

lemma *ennreal-suminf-neq-top*: *summable f* $\Longrightarrow$ $(\bigwedge i.\ 0 \le f\ i) \Longrightarrow (\sum i.\ ennreal\ (f\ i)) \neq top$
⟨*proof*⟩

lemma *suminf-ennreal-eq*:
$(\bigwedge i.\ 0 \le f\ i) \Longrightarrow f\ sums\ x \Longrightarrow (\sum i.\ ennreal\ (f\ i)) = ennreal\ x$
⟨*proof*⟩

lemma *ennreal-suminf-bound-add*:
fixes $f :: nat \Rightarrow ennreal$
shows $(\bigwedge N.\ (\sum n{<}N.\ f\ n) + y \le x) \Longrightarrow suminf\ f + y \le x$
⟨*proof*⟩

lemma *ennreal-suminf-SUP-eq-directed*:
fixes $f :: 'a \Rightarrow nat \Rightarrow ennreal$
assumes $*$: $\bigwedge N\ i\ j.\ i \in I \Longrightarrow j \in I \Longrightarrow finite\ N \Longrightarrow \exists k{\in}I.\ \forall n{\in}N.\ f\ i\ n \le f\ k\ n \wedge f\ j\ n \le f\ k\ n$
shows $(\sum n.\ SUP\ i{:}I.\ f\ i\ n) = (SUP\ i{:}I.\ \sum n.\ f\ i\ n)$
⟨*proof*⟩

lemma *INF-ennreal-add-const*:
fixes $f\ g :: nat \Rightarrow ennreal$
shows $(INF\ i.\ f\ i + c) = (INF\ i.\ f\ i) + c$
⟨*proof*⟩

lemma *INF-ennreal-const-add*:
fixes $f\ g :: nat \Rightarrow ennreal$

shows $(INF\ i.\ c + f\ i) = c + (INF\ i.\ f\ i)$
⟨*proof*⟩

lemma *SUP-mult-left-ennreal*: $c * (SUP\ i{:}I.\ f\ i) = (SUP\ i{:}I.\ c * f\ i :: ennreal)$
⟨*proof*⟩

lemma *SUP-mult-right-ennreal*: $(SUP\ i{:}I.\ f\ i) * c = (SUP\ i{:}I.\ f\ i * c :: ennreal)$
⟨*proof*⟩

lemma *SUP-divide-ennreal*: $(SUP\ i{:}I.\ f\ i)\ /\ c = (SUP\ i{:}I.\ f\ i\ /\ c :: ennreal)$
⟨*proof*⟩

lemma *ennreal-SUP-of-nat-eq-top*: $(SUP\ x.\ of\text{-}nat\ x :: ennreal) = top$
⟨*proof*⟩

lemma *ennreal-SUP-eq-top*:
fixes $f :: 'a \Rightarrow ennreal$
assumes $\bigwedge n.\ \exists\, i{\in}I.\ of\text{-}nat\ n \leq f\ i$
shows $(SUP\ i : I.\ f\ i) = top$
⟨*proof*⟩

lemma *ennreal-INF-const-minus*:
fixes $f :: 'a \Rightarrow ennreal$
shows $I \neq \{\} \Longrightarrow (SUP\ x{:}I.\ c - f\ x) = c - (INF\ x{:}I.\ f\ x)$
⟨*proof*⟩

lemma *of-nat-Sup-ennreal*:
assumes $A \neq \{\}$ *bdd-above A*
shows $of\text{-}nat\ (Sup\ A) = (SUP\ a{:}A.\ of\text{-}nat\ a :: ennreal)$
⟨*proof*⟩

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*⟩

lemma *ennreal-SUP-add*:
fixes $f\ g :: nat \Rightarrow ennreal$
shows $incseq\ f \Longrightarrow incseq\ g \Longrightarrow (SUP\ i.\ f\ i + g\ i) = SUPREMUM\ UNIV\ f + SUPREMUM\ UNIV\ g$
⟨*proof*⟩

lemma *ennreal-SUP-setsum*:
fixes $f :: 'a \Rightarrow nat \Rightarrow ennreal$
shows $(\bigwedge i.\ i \in I \Longrightarrow incseq\ (f\ i)) \Longrightarrow (SUP\ n.\ \sum i{\in}I.\ f\ i\ n) = (\sum i{\in}I.\ SUP\ n.\ f\ i\ n)$
⟨*proof*⟩

lemma *ennreal-liminf-minus*:
fixes $f :: nat \Rightarrow ennreal$
shows $(\bigwedge n.\ f\ n \leq c) \Longrightarrow liminf\ (\lambda n.\ c - f\ n) = c - limsup\ f$
⟨*proof*⟩

lemma *ennreal-continuous-on-cmult*:
$(c::ennreal) < top \Longrightarrow continuous\text{-}on\ A\ f \Longrightarrow continuous\text{-}on\ A\ (\lambda x.\ c * f\ x)$
⟨*proof*⟩

lemma *ennreal-tendsto-cmult*:
$(c::ennreal) < top \Longrightarrow (f \longrightarrow x)\ F \Longrightarrow ((\lambda x.\ c * f\ x) \longrightarrow c * x)\ F$
⟨*proof*⟩

lemma *tendsto-ennrealI*[*intro*, *simp*]:
$(f \longrightarrow x)\ F \Longrightarrow ((\lambda x.\ ennreal\ (f\ x)) \longrightarrow ennreal\ x)\ F$
⟨*proof*⟩

lemma *ennreal-suminf-minus*:
fixes $f\ g :: nat \Rightarrow ennreal$
shows $(\bigwedge i.\ g\ i \leq f\ i) \Longrightarrow suminf\ f \neq top \Longrightarrow suminf\ g \neq top \Longrightarrow (\sum i.\ f\ i - g\ i) = suminf\ f - suminf\ g$
⟨*proof*⟩

lemma *ennreal-Sup-countable-SUP*:
$A \neq \{\} \Longrightarrow \exists f::nat \Rightarrow ennreal.\ incseq\ f \wedge range\ f \subseteq A \wedge Sup\ A = (SUP\ i.\ f\ i)$
⟨*proof*⟩

lemma *ennreal-SUP-countable-SUP*:
$A \neq \{\} \Longrightarrow \exists f::nat \Rightarrow ennreal.\ range\ f \subseteq g`A \wedge SUPREMUM\ A\ g = SUPREMUM\ UNIV\ f$
⟨*proof*⟩

lemma *of-nat-tendsto-top-ennreal*: $(\lambda n::nat.\ of\text{-}nat\ n :: ennreal) \longrightarrow top$
⟨*proof*⟩

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*⟩

lemma *ennreal-suminf-SUP-eq*:
fixes $f :: nat \Rightarrow nat \Rightarrow ennreal$
shows $(\bigwedge i.\ incseq\ (\lambda n.\ f\ n\ i)) \Longrightarrow (\sum i.\ SUP\ n.\ f\ n\ i) = (SUP\ n.\ \sum i.\ f\ n\ i)$
⟨*proof*⟩

lemma *ennreal-SUP-add-left*:
fixes $c :: ennreal$

shows $I \neq \{\} \Longrightarrow (SUP\ i{:}I.\ f\ i + c) = (SUP\ i{:}I.\ f\ i) + c$
⟨*proof*⟩

lemma *ennreal-SUP-const-minus*:
fixes $f :: 'a \Rightarrow ennreal$
shows $I \neq \{\} \Longrightarrow c < top \Longrightarrow (INF\ x{:}I.\ c - f\ x) = c - (SUP\ x{:}I.\ f\ x)$
⟨*proof*⟩

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 $x \neq \infty$
shows $\exists\, i \in A.\ f\ i < x + e$
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

lemma *ennreal-approx-unit*:
$(\bigwedge a{::}ennreal.\ 0 < a \Longrightarrow a < 1 \Longrightarrow a * z \leq y) \Longrightarrow z \leq y$
⟨*proof*⟩

lemma *suminf-ennreal2*:
$(\bigwedge i.\ 0 \leq f\ i) \Longrightarrow summable\ f \Longrightarrow (\sum i.\ ennreal\ (f\ i)) = ennreal\ (\sum i.\ f\ i)$
⟨*proof*⟩

lemma *less-top-ennreal*: $x < top \longleftrightarrow (\exists\, r{\geq}0.\ x = ennreal\ r)$

⟨*proof*⟩

lemma *tendsto-top-iff-ennreal*:
fixes $f :: 'a \Rightarrow ennreal$
shows $(f \longrightarrow top)\ F \longleftrightarrow (\forall l{\geq}0.\ eventually\ (\lambda x.\ ennreal\ l < f\ x)\ F)$
⟨*proof*⟩

lemma *ennreal-tendsto-top-eq-at-top*:
$((\lambda z.\ ennreal\ (f\ z)) \longrightarrow top)\ F \longleftrightarrow (LIM\ z\ F.\ f\ z :> at\text{-}top)$
⟨*proof*⟩

lemma *tendsto-0-if-Limsup-eq-0-ennreal*:
fixes $f :: - \Rightarrow ennreal$
shows $Limsup\ F\ f = 0 \Longrightarrow (f \longrightarrow 0)\ F$
⟨*proof*⟩

lemma *diff-le-self-ennreal*[*simp*]: $a - b \leq (a{::}ennreal)$
⟨*proof*⟩

lemma *ennreal-ineq-diff-add*: $b \leq a \Longrightarrow a = b + (a - b{::}ennreal)$
⟨*proof*⟩

lemma *ennreal-mult-strict-left-mono*: $(a{::}ennreal) < c \Longrightarrow 0 < b \Longrightarrow b < top \Longrightarrow b * a < b * c$
⟨*proof*⟩

lemma *ennreal-between*: $0 < e \Longrightarrow 0 < x \Longrightarrow x < top \Longrightarrow x - e < (x{::}ennreal)$
⟨*proof*⟩

lemma *minus-less-iff-ennreal*: $b < top \Longrightarrow b \leq a \Longrightarrow a - b < c \longleftrightarrow a < c + (b{::}ennreal)$
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *phantom-comp-of-phantom* [*simp*]: *phantom* ∘ *of-phantom* = *id*
and *of-phantom-comp-phantom* [*simp*]: *of-phantom* ∘ *phantom* = *id*
⟨*proof*⟩

syntax -*Phantom* :: *type* ⇒ *logic* ((*1Phantom*/(*1* $'$(-$'$))))
translations
Phantom($'t$) => *CONST phantom* :: - ⇒ ($'t$, -) *phantom*

⟨*ML*⟩

lemma *of-phantom-inject* [*simp*]:
of-phantom x = *of-phantom y* ⟷ *x* = *y*
⟨*proof*⟩

end

36 Cardinality of types

theory *Cardinality*
imports *Phantom-Type*
begin

36.1 Preliminary lemmas

lemma (**in** *type-definition*) *univ*:
UNIV = *Abs* ‘ *A*
⟨*proof*⟩

lemma (**in** *type-definition*) *card*: *card* (*UNIV* :: $'b$ *set*) = *card A*
⟨*proof*⟩

lemma *finite-range-Some*: *finite* (*range* (*Some* :: $'a$ ⇒ $'a$ *option*)) = *finite* (*UNIV* :: $'a$ *set*)
⟨*proof*⟩

lemma *infinite-literal*: ¬ *finite* (*UNIV* :: *String.literal set*)
⟨*proof*⟩

36.2 Cardinalities of types

syntax -*type-card* :: *type* => *nat* ((*1CARD*/(*1* $'$(-$'$))))

translations *CARD*($'t$) => *CONST card* (*CONST UNIV* :: $'t$ *set*)

⟨*ML*⟩

lemma *card-prod* [*simp*]: $CARD('a \times 'b) = CARD('a) * CARD('b)$
⟨*proof*⟩

lemma *card-UNIV-sum*: $CARD('a + 'b) = (\textit{if } CARD('a) \neq 0 \wedge CARD('b) \neq 0 \textit{ then } CARD('a) + CARD('b) \textit{ else } 0)$
⟨*proof*⟩

lemma *card-sum* [*simp*]: $CARD('a + 'b) = CARD('a::\textit{finite}) + CARD('b::\textit{finite})$
⟨*proof*⟩

lemma *card-UNIV-option*: $CARD('a\ option) = (\textit{if } CARD('a) = 0 \textit{ then } 0 \textit{ else } CARD('a) + 1)$
⟨*proof*⟩

lemma *card-option* [*simp*]: $CARD('a\ option) = Suc\ CARD('a::\textit{finite})$
⟨*proof*⟩

lemma *card-UNIV-set*: $CARD('a\ set) = (\textit{if } CARD('a) = 0 \textit{ then } 0 \textit{ else } 2 \hat{} CARD('a))$
⟨*proof*⟩

lemma *card-set* [*simp*]: $CARD('a\ set) = 2 \hat{} CARD('a::\textit{finite})$
⟨*proof*⟩

lemma *card-nat* [*simp*]: $CARD(nat) = 0$
⟨*proof*⟩

lemma *card-fun*: $CARD('a \Rightarrow 'b) = (\textit{if } CARD('a) \neq 0 \wedge CARD('b) \neq 0 \vee CARD('b) = 1 \textit{ then } CARD('b) \hat{} CARD('a) \textit{ else } 0)$
⟨*proof*⟩

corollary *finite-UNIV-fun*:
$\textit{finite}\ (UNIV :: ('a \Rightarrow 'b)\ set) \longleftrightarrow$
$\textit{finite}\ (UNIV :: 'a\ set) \wedge \textit{finite}\ (UNIV :: 'b\ set) \vee CARD('b) = 1$
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *card-literal*: $CARD(String.literal) = 0$
⟨*proof*⟩

36.3 Classes with at least 1 and 2

Class finite already captures "at least 1"

lemma *zero-less-card-finite* [*simp*]: $0 < CARD('a::\textit{finite})$
⟨*proof*⟩

lemma *one-le-card-finite* [*simp*]: $Suc\ 0 \leq CARD('a::\textit{finite})$
⟨*proof*⟩

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*)
  ⟨*proof*⟩

lemma *one-less-int-card*: *1* < *int CARD*('*a*::*card2*)
  ⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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 ⟨*proof*⟩
end

instantiation *int* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*int*) *False*
definition *card-UNIV* = *Phantom*(*int*) *0*

instance ⟨*proof*⟩
end

instantiation *natural* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*natural*) *False*
definition *card-UNIV* = *Phantom*(*natural*) *0*
instance
⟨*proof*⟩
end

instantiation *integer* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*integer*) *False*
definition *card-UNIV* = *Phantom*(*integer*) *0*
instance
⟨*proof*⟩
end

instantiation *list* :: (*type*) *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*('*a list*) *False*
definition *card-UNIV* = *Phantom*('*a list*) *0*
instance ⟨*proof*⟩
end

instantiation *unit* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*unit*) *True*
definition *card-UNIV* = *Phantom*(*unit*) *1*
instance ⟨*proof*⟩
end

instantiation *bool* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*bool*) *True*
definition *card-UNIV* = *Phantom*(*bool*) *2*
instance ⟨*proof*⟩
end

instantiation *char* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*char*) *True*
definition *card-UNIV* = *Phantom*(*char*) *256*
instance ⟨*proof*⟩
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 ⟨*proof*⟩
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 ⟨*proof*⟩
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
⟨*proof*⟩
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 ⟨*proof*⟩
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
⟨*proof*⟩
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 ⟨*proof*⟩
end

instantiation *option* :: (*finite-UNIV*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*('*a option*) (*of-phantom* (*finite-UNIV* :: '*a finite-UNIV*))
instance ⟨*proof*⟩
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 ⟨*proof*⟩
end

instantiation *String.literal* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*String.literal*) *False*
definition *card-UNIV* = *Phantom*(*String.literal*) *0*
instance
⟨*proof*⟩
end

instantiation *set* :: (*finite-UNIV*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*('*a set*) (*of-phantom* (*finite-UNIV* :: '*a finite-UNIV*))
instance ⟨*proof*⟩
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 ⟨*proof*⟩
end

lemma *UNIV-finite-1*: *UNIV* = *set* [*finite-1*.a_1]
⟨*proof*⟩

lemma *UNIV-finite-2*: *UNIV* = *set* [*finite-2*.a_1, *finite-2*.a_2]
⟨*proof*⟩

lemma *UNIV-finite-3*: *UNIV* = *set* [*finite-3*.a_1, *finite-3*.a_2, *finite-3*.a_3]
⟨*proof*⟩

lemma *UNIV-finite-4*: *UNIV* = *set* [*finite-4*.a_1, *finite-4*.a_2, *finite-4*.a_3, *finite-4*.a_4]
⟨*proof*⟩

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]
⟨*proof*⟩

instantiation *Enum.finite-1* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*Enum.finite-1*) *True*
definition *card-UNIV* = *Phantom*(*Enum.finite-1*) *1*
instance
⟨*proof*⟩
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
⟨*proof*⟩
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
⟨*proof*⟩
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
⟨*proof*⟩
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
⟨*proof*⟩
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*)
⟨*proof*⟩

lemma *card-UNIV'-code* [*code*]:
card-UNIV' = *card-UNIV*
⟨*proof*⟩

end

lemma *card-Compl*:
finite A ⟹ *card* (− *A*) = *card* (*UNIV* :: '*a set*) − *card* (*A* :: '*a set*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩ **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*)
⟨*proof*⟩ **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*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

notepad begin — test code setup
⟨*proof*⟩
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 ⇒ ('a ⇀ 'b) ⇒ 'a ⇒ '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*)

⟨*proof*⟩

lemma *map-default-insert*:
  map-default b $(f(a \mapsto b')) = (\textit{map-default}\ b\ f)(a := b')$
⟨*proof*⟩

lemma *map-default-empty* [*simp*]: *map-default b empty* $= (\lambda a.\ b)$
⟨*proof*⟩

lemma *map-default-inject*:
  fixes $g\ g' :: 'a \rightharpoonup 'b$
  assumes *infin-eq*: $\neg$ *finite* (*UNIV* :: *'a set*) $\vee$ $b = b'$
  and *fin*: *finite* (*dom g*) **and** *b*: $b \notin \textit{ran}\ g$
  and *fin'*: *finite* (*dom g'*) **and** *b'*: $b' \notin \textit{ran}\ g'$
  and *eq'*: *map-default b g* = *map-default b' g'*
  shows $b = b'\ g = g'$
⟨*proof*⟩

37.2 The finfun type

definition *finfun* = $\{f :: 'a \Rightarrow 'b.\ \exists\, b.\ \textit{finite}\ \{a.\ f\ a \neq b\}\}$

typedef ('*a*,'*b*) *finfun* ((- ⇒*f* /-) [*22*, *21*] *21*) = *finfun* :: (*'a* => *'b*) *set*
  morphisms *finfun-apply Abs-finfun*
⟨*proof*⟩

type-notation *finfun* ((- ⇒*f* /-) [*22*, *21*] *21*)

setup-lifting *type-definition-finfun*

lemma *fun-upd-finfun*: $y(a := b) \in \textit{finfun} \longleftrightarrow y \in \textit{finfun}$
⟨*proof*⟩

lemma *const-finfun*: $(\lambda x.\ a) \in \textit{finfun}$
⟨*proof*⟩

lemma *finfun-left-compose*:
  assumes $y \in \textit{finfun}$
  shows $g \circ y \in \textit{finfun}$
⟨*proof*⟩

lemma assumes $y \in \textit{finfun}$
  shows *fst-finfun*: $\textit{fst} \circ y \in \textit{finfun}$
  and *snd-finfun*: $\textit{snd} \circ y \in \textit{finfun}$
⟨*proof*⟩

lemma *map-of-finfun*: *map-of xs* $\in$ *finfun*
⟨*proof*⟩

lemma *Diag-finfun*: $(\lambda x.\ (f\ x,\ g\ x)) \in \mathit{finfun} \longleftrightarrow f \in \mathit{finfun} \wedge g \in \mathit{finfun}$
⟨*proof*⟩

lemma *finfun-right-compose*:
assumes *g*: $g \in \mathit{finfun}$ **and** *inj*: *inj f*
shows $g\ o\ f \in \mathit{finfun}$
⟨*proof*⟩

lemma *finfun-curry*:
assumes *fin*: $f \in \mathit{finfun}$
shows *curry f* $\in$ *finfun curry f a* $\in$ *finfun*
⟨*proof*⟩

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 $x\ y :: 'a \Rightarrow 'b$
assumes *fin*: *finite* (*UNIV* :: *'a set*)
shows *Abs-finfun x* = *Abs-finfun y* $\longleftrightarrow$ $x = y$
⟨*proof*⟩

lemma *Abs-finfun-inject-finite-class*:
fixes $x\ y :: ('a :: \mathit{finite}) \Rightarrow 'b$
shows *Abs-finfun x* = *Abs-finfun y* $\longleftrightarrow$ $x = y$
⟨*proof*⟩

lemma *Abs-finfun-inj-finite*:
assumes *fin*: *finite* (*UNIV* :: *'a set*)
shows *inj* (*Abs-finfun* :: (*'a* $\Rightarrow$ *'b*) $\Rightarrow$ *'a* $\Rightarrow$*f* *'b*)
⟨*proof*⟩

lemma *Abs-finfun-inverse-finite*:
fixes $x :: 'a \Rightarrow 'b$
assumes *fin*: *finite* (*UNIV* :: *'a set*)
shows *finfun-apply* (*Abs-finfun x*) = x
including *finfun*
⟨*proof*⟩

lemma *Abs-finfun-inverse-finite-class*:
fixes $x :: ('a :: \mathit{finite}) \Rightarrow 'b$
shows *finfun-apply* (*Abs-finfun x*) = x
⟨*proof*⟩

lemma *finfun-eq-finite-UNIV*: *finite* (*UNIV* :: *'a set*) $\Longrightarrow$ (*finfun* :: (*'a* $\Rightarrow$ *'b*) *set*)

$= UNIV$
⟨*proof*⟩

lemma *finfun-finite-UNIV-class*: $finfun = (UNIV :: ('a :: finite \Rightarrow 'b)\ set)$
⟨*proof*⟩

lemma *map-default-in-finfun*:
assumes *fin*: $finite\ (dom\ f)$
shows $map\text{-}default\ b\ f \in finfun$
⟨*proof*⟩

lemma *finfun-cases-map-default*:
obtains $b\ g$ **where** $f = Abs\text{-}finfun\ (map\text{-}default\ b\ g)\ finite\ (dom\ g)\ b \notin ran\ g$
⟨*proof*⟩

37.3 Kernel functions for type $'a \Rightarrow f\ 'b$

lift-definition *finfun-const* :: $'b \Rightarrow 'a \Rightarrow f\ 'b\ (K\$/\ -\ [0]\ 1)$
is $\lambda\ b\ x.\ b$ ⟨*proof*⟩

lift-definition *finfun-update* :: $'a \Rightarrow f\ 'b \Rightarrow 'a \Rightarrow 'b \Rightarrow 'a \Rightarrow f\ 'b\ (-'(-\ \$:=\ -')\ [1000,0,0]\ 1000)$ **is** *fun-upd*
⟨*proof*⟩

lemma *finfun-update-twist*: $a \neq a' \Longrightarrow f(a\ \$:=\ b)(a'\ \$:=\ b') = f(a'\ \$:=\ b')(a\ \$:=\ b)$
⟨*proof*⟩

lemma *finfun-update-twice* [*simp*]:
$f(a\ \$:=\ b)(a\ \$:=\ b') = f(a\ \$:=\ b')$
⟨*proof*⟩

lemma *finfun-update-const-same*: $(K\$\ b)(a\ \$:=\ b) = (K\$\ b)$
⟨*proof*⟩

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*]: $finfun\text{-}update\text{-}code = finfun\text{-}update$

code-datatype *finfun-const finfun-update-code*

lemma *finfun-update-const-code* [*code*]:
$(K\$\ b)(a\ \$:=\ b') = (if\ b = b'\ then\ (K\$\ b)\ else\ finfun\text{-}update\text{-}code\ (K\$\ b)\ a\ b')$
⟨*proof*⟩

lemma *finfun-update-update-code* [*code*]:
$(finfun\text{-}update\text{-}code\ f\ a\ b)(a'\ \$:=\ b') = (if\ a = a'\ then\ f(a\ \$:=\ b')\ else\ finfun\text{-}update\text{-}code\ (f(a'\ \$:=\ b'))\ a\ b)$
⟨*proof*⟩

37.5 Setup for quickcheck

quickcheck-generator *finfun constructors*: *finfun-update-code*, *finfun-const* :: *'b* ⇒ *'a* ⇒*f* *'b*

37.6 *finfun-update* as instance of *comp-fun-commute*

interpretation *finfun-update*: *comp-fun-commute* λ*a f*. *f*(*a* :: *'a* $:= *b'*)
  including *finfun*
⟨*proof*⟩

lemma *fold-finfun-update-finite-univ*:
  assumes *fin*: *finite* (*UNIV* :: *'a set*)
  shows *Finite-Set.fold* (λ*a f*. *f*(*a* $:= *b'*)) (*K*$ *b*) (*UNIV* :: *'a set*) = (*K*$ *b'*)
⟨*proof*⟩

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*⟩

lemma *finite-finfun-default-aux*:
  fixes *f* :: *'a* ⇒ *'b*
  assumes *fin*: *f* ∈ *finfun*
  shows *finite* {*a*. *f a* ≠ *finfun-default-aux f*}
⟨*proof*⟩

lemma *finfun-default-aux-update-const*:
  fixes *f* :: *'a* ⇒ *'b*
  assumes *fin*: *f* ∈ *finfun*
  shows *finfun-default-aux* (*f*(*a* := *b*)) = *finfun-default-aux f*
⟨*proof*⟩

lift-definition *finfun-default* :: *'a* ⇒*f* *'b* ⇒ *'b*
is *finfun-default-aux* ⟨*proof*⟩

lemma *finite-finfun-default*: *finite* {*a*. *finfun-apply f a* ≠ *finfun-default f*}
⟨*proof*⟩

lemma *finfun-default-const*: *finfun-default* ((*K*$ *b*) :: *'a* ⇒*f* *'b*) = (*if finite* (*UNIV* :: *'a set*) *then undefined else b*)

⟨*proof*⟩

lemma *finfun-default-update-const*:
finfun-default (*f*(*a* $:= *b*)) = *finfun-default f*
⟨*proof*⟩

lemma *finfun-default-const-code* [*code*]:
finfun-default ((*K*$ *c*) :: 'a :: *card-UNIV* ⇒*f* 'b) = (*if CARD*('a) = *0 then c else undefined*)
⟨*proof*⟩

lemma *finfun-default-update-code* [*code*]:
finfun-default (*finfun-update-code f a b*) = *finfun-default f*
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *upd-upd-twice*: *upd a b''* (*upd a b'* (*cnst b*)) = *upd a b''* (*cnst b*)
⟨*proof*⟩

lemma *map-default-update-const*:
assumes *fin*: *finite* (*dom f*)
and *anf*: *a* ∉ *dom f*
and *fg*: *f* ⊆*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*⟩

lemma *map-default-update-twice*:

assumes *fin*: *finite* (*dom f*)
and *anf*: $a \notin dom\ f$
and *fg*: $f \subseteq_m g$
shows *upd a d''* (*upd a d'* (*Finite-Set.fold* ($\lambda a.$ *upd a* (*map-default d g a*)) (*cnst d*) (*dom f*))) =
upd a d'' (*Finite-Set.fold* ($\lambda a.$ *upd a* (*map-default d g a*)) (*cnst d*) (*dom f*))
⟨*proof*⟩

lemma *map-default-eq-id* [*simp*]: *map-default d* (($\lambda a.$ *Some* (*f a*)) |‘ {$a.$ *f a* ≠ *d*}) = *f*
⟨*proof*⟩

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*⟩

lemma *finfun-rec-upd* [*simp*]:
finfun-rec cnst upd (*f*(*a'* $:= *b'*)) = *upd a' b'* (*finfun-rec cnst upd f*)
including *finfun*
⟨*proof*⟩

end

locale *finfun-rec-wf* = *finfun-rec-wf-aux* +
assumes *const-update-all*:
finite (*UNIV* :: *'a set*) ⟹ *Finite-Set.fold* ($\lambda a.$ *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*⟩

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*⟩

lemma *finfun-exhaust-disj*: ($\exists b.\ x$ = *finfun-const b*) ∨ ($\exists f\ a\ b.\ x$ = *finfun-update f a b*)

⟨*proof*⟩

lemma *finfun-exhaust*:
obtains b **where** $x = (K\$\ b)$
| $f\ a\ b$ **where** $x = f(a\ \$:=\ b)$
⟨*proof*⟩

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*⟩

37.10 Function application

notation *finfun-apply* (**infixl** $ *999*)

interpretation *finfun-apply-aux*: *finfun-rec-wf-aux* $\lambda b.\ b\ \lambda a'\ b\ c.\ if\ (a = a')\ then\ b\ else\ c$
⟨*proof*⟩

interpretation *finfun-apply*: *finfun-rec-wf* $\lambda b.\ b\ \lambda a'\ b\ c.\ if\ (a = a')\ then\ b\ else\ c$
⟨*proof*⟩

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*⟩

lemma *finfun-upd-apply*: $f(a\ \$:=\ b)\ \$\ a' = (if\ a = a'\ then\ b\ else\ f\ \$\ a')$
and *finfun-upd-apply-code* [*code*]: $(finfun\text{-}update\text{-}code\ f\ a\ b)\ \$\ a' = (if\ a = a'\ then\ b\ else\ f\ \$\ a')$
⟨*proof*⟩

lemma *finfun-const-apply* [*simp*, *code*]: $(K\$\ b)\ \$\ a = b$
⟨*proof*⟩

lemma *finfun-upd-apply-same* [*simp*]:
$f(a\ \$:=\ b)\ \$\ a = b$
⟨*proof*⟩

lemma *finfun-upd-apply-other* [*simp*]:
$a \neq a' \Longrightarrow f(a\ \$:=\ b)\ \$\ a' = f\ \$\ a'$
⟨*proof*⟩

lemma *finfun-ext*: $(\bigwedge a.\ f\ \$\ a = g\ \$\ a) \Longrightarrow f = g$
⟨*proof*⟩

lemma *expand-finfun-eq*: $(f = g) = (op\ \$\ f = op\ \$\ g)$
⟨*proof*⟩

lemma *finfun-upd-triv* [*simp*]: $f(x\ \$:=\ f\ \$\ x) = f$
⟨*proof*⟩

lemma *finfun-const-inject* [*simp*]: $(K\$\ b) = (K\$\ b') \equiv b = b'$
⟨*proof*⟩

lemma *finfun-const-eq-update*:
$((K\$\ b) = f(a\ \$:=\ b')) = (b = b' \wedge (\forall a'.\ a \neq a' \longrightarrow f\ \$\ a' = b))$
⟨*proof*⟩

37.11 Function composition

definition *finfun-comp* :: $('a \Rightarrow 'b) \Rightarrow 'c \Rightarrow_f 'a \Rightarrow 'c \Rightarrow_f 'b$ (**infixr** $\circ\$$ *55*)
where [*code del*]: $g \circ\$ f = finfun\text{-}rec\ (\lambda b.\ (K\$\ g\ b))\ (\lambda a\ b\ c.\ c(a\ \$:=\ g\ b))\ f$

notation (*ASCII*)
finfun-comp (**infixr** *o$* *55*)

interpretation *finfun-comp-aux*: *finfun-rec-wf-aux* $(\lambda b.\ (K\$\ g\ b))\ (\lambda a\ b\ c.\ c(a\ \$:=\ g\ b))$
⟨*proof*⟩

interpretation *finfun-comp*: *finfun-rec-wf* $(\lambda b.\ (K\$\ g\ b))\ (\lambda a\ b\ c.\ c(a\ \$:=\ g\ b))$
⟨*proof*⟩

lemma *finfun-comp-const* [*simp*, *code*]:
$g \circ\$\ (K\$\ c) = (K\$\ g\ c)$
⟨*proof*⟩

lemma *finfun-comp-update* [*simp*]: $g \circ\$\ (f(a\ \$:=\ b)) = (g \circ\$\ f)(a\ \$:=\ g\ b)$
and *finfun-comp-update-code* [*code*]:
$g \circ\$\ (finfun\text{-}update\text{-}code\ f\ a\ b) = finfun\text{-}update\text{-}code\ (g \circ\$\ f)\ a\ (g\ b)$
⟨*proof*⟩

lemma *finfun-comp-apply* [*simp*]:
$op\ \$\ (g \circ\$\ f) = g \circ op\ \$\ f$
⟨*proof*⟩

lemma *finfun-comp-comp-collapse* [*simp*]: $f \circ\$\ g \circ\$\ h = (f \circ g) \circ\$\ h$
⟨*proof*⟩

lemma *finfun-comp-const1* [*simp*]: $(\lambda x.\ c) \circ\$\ f = (K\$\ c)$
⟨*proof*⟩

lemma *finfun-comp-id1* [*simp*]: $(\lambda x.\ x) \circ\$\ f = f\ id \circ\$\ f = f$

⟨*proof*⟩

lemma *finfun-comp-conv-comp*: $g \circ\$ f = Abs\text{-}finfun\ (g \circ op\ \$\ f)$
including *finfun*
⟨*proof*⟩

definition *finfun-comp2* :: $'b \Rightarrow f\ 'c \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow f\ 'c$ (**infixr** $\$\circ$ *55*)
where [*code del*]: $g\ \$\circ\ f = Abs\text{-}finfun\ (op\ \$\ g \circ f)$

notation (*ASCII*)
finfun-comp2 (**infixr** $\$o$ *55*)

lemma *finfun-comp2-const* [*code*, *simp*]: *finfun-comp2* $(K\$\ c)\ f = (K\$\ c)$
including *finfun*
⟨*proof*⟩

lemma *finfun-comp2-update*:
includes *finfun*
assumes *inj*: *inj f*
shows *finfun-comp2* $(g(b\ \$:=\ c))\ f = (if\ b \in range\ f\ then\ (finfun\text{-}comp2\ g\ f)(inv\ f\ b\ \$:=\ c)\ else\ finfun\text{-}comp2\ g\ f)$
⟨*proof*⟩

37.12 Universal quantification

definition *finfun-All-except* :: $'a\ list \Rightarrow 'a \Rightarrow f\ bool \Rightarrow bool$
where [*code del*]: $finfun\text{-}All\text{-}except\ A\ P \equiv \forall a.\ a \in set\ A \vee P\ \$\ a$

lemma *finfun-All-except-const*: $finfun\text{-}All\text{-}except\ A\ (K\$\ b) \longleftrightarrow b \vee set\ A = UNIV$
⟨*proof*⟩

lemma *finfun-All-except-const-finfun-UNIV-code* [*code*]:
$finfun\text{-}All\text{-}except\ A\ (K\$\ b) = (b \vee is\text{-}list\text{-}UNIV\ A)$
⟨*proof*⟩

lemma *finfun-All-except-update*:
$finfun\text{-}All\text{-}except\ A\ f(a\ \$:=\ b) = ((a \in set\ A \vee b) \wedge finfun\text{-}All\text{-}except\ (a\ \#\ A)\ f)$
⟨*proof*⟩

lemma *finfun-All-except-update-code* [*code*]:
fixes $a :: 'a :: card\text{-}UNIV$
shows $finfun\text{-}All\text{-}except\ A\ (finfun\text{-}update\text{-}code\ f\ a\ b) = ((a \in set\ A \vee b) \wedge finfun\text{-}All\text{-}except\ (a\ \#\ A)\ f)$
⟨*proof*⟩

definition *finfun-All* :: $'a \Rightarrow f\ bool \Rightarrow bool$
where $finfun\text{-}All = finfun\text{-}All\text{-}except\ []$

lemma *finfun-All-const* [*simp*]: *finfun-All* (*K*$ *b*) = *b*
⟨*proof*⟩

lemma *finfun-All-update*: *finfun-All f*(*a* $:= *b*) = (*b* ∧ *finfun-All-except* [*a*] *f*)
⟨*proof*⟩

lemma *finfun-All-All*: *finfun-All P* = *All* (*op* $ *P*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *finfun-Ex-const* [*simp*]: *finfun-Ex* (*K*$ *b*) = *b*
⟨*proof*⟩

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*))
⟨*proof*⟩

interpretation *finfun-Diag*: *finfun-rec-wf* λ*b*. *Pair b* ∘$ *g* λ*a b c*. *c*(*a* $:= (*b*, *g* $
a))
⟨*proof*⟩

lemma *finfun-Diag-const1*: ($*K*$ *b*, *g*$) = *Pair b* ∘$ *g*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *finfun-Diag-const2*: (f, K c) = ($\lambda b.\ (b,\ c)$) $\circ$$ f
⟨*proof*⟩

lemma *finfun-Diag-update2*: ($f, g($a $:= c)$) = (f, g)($a $:= ($f$ $ $a,\ c$))
⟨*proof*⟩

lemma *finfun-Diag-const-const* [*simp*]: (K b, K c) = (K $(b,\ c)$)
⟨*proof*⟩

lemma *finfun-Diag-const-update*:
(K $b, g($a $:= c)$) = (K b, g)($a $:= $(b,\ c)$)
⟨*proof*⟩

lemma *finfun-Diag-update-const*:
($f($a $:= b), K$ c) = (f, K c)($a $:= $(b,\ c)$)
⟨*proof*⟩

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)))
⟨*proof*⟩

lemma *finfun-Diag-apply* [*simp*]: *op* $ (f, g) = ($\lambda x.\ (f$ $ $x,\ g$ $ x))
⟨*proof*⟩

lemma *finfun-Diag-conv-Abs-finfun*:
(f, g) = *Abs-finfun* (($\lambda x.\ (f$ $ $x,\ g$ $ x)))
including *finfun*
⟨*proof*⟩

lemma *finfun-Diag-eq*: (f, g) = (f', g') $\longleftrightarrow$ $f = f' \wedge g = g'$
⟨*proof*⟩

definition *finfun-fst* :: $'a \Rightarrow f$ $('b \times 'c) \Rightarrow 'a \Rightarrow f$ $'b$
where [*code*]: *finfun-fst* f = *fst* $\circ$$ f

lemma *finfun-fst-const*: *finfun-fst* ($K$$ *bc*) = ($K$$ *fst bc*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *finfun-fst-comp-conv*: *finfun-fst* (f $\circ$$ g) = (*fst* $\circ$ f) $\circ$$ g
⟨*proof*⟩

lemma *finfun-fst-conv* [*simp*]: *finfun-fst* (f, g) = f

⟨*proof*⟩

lemma *finfun-fst-conv-Abs-finfun*: *finfun-fst* = (λ*f*. *Abs-finfun* (*fst* ∘ *op* $ *f*))
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *finfun-snd-comp-conv*: *finfun-snd* (*f* ∘$ *g*) = (*snd* ∘ *f*) ∘$ *g*
⟨*proof*⟩

lemma *finfun-snd-conv* [*simp*]: *finfun-snd* ($*f*, *g*$) = *g*
⟨*proof*⟩

lemma *finfun-snd-conv-Abs-finfun*: *finfun-snd* = (λ*f*. *Abs-finfun* (*snd* ∘ *op* $ *f*))
⟨*proof*⟩

lemma *finfun-Diag-collapse* [*simp*]: ($*finfun-fst f*, *finfun-snd f*$) = *f*
⟨*proof*⟩

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*))
⟨*proof*⟩

interpretation *finfun-curry*: *finfun-rec-wf finfun-const* ∘ *finfun-const* λ(*a*, *b*) *c f*. *f*(*a* $:= (*f* $ *a*)(*b* $:= *c*))
⟨*proof*⟩

lemma *finfun-curry-const* [*simp*, *code*]: *finfun-curry* (*K*$ *c*) = (*K*$ *K*$ *c*)
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *finfun-Abs-finfun-curry*: **assumes** *fin*: *f* ∈ *finfun*
shows (λ*a*. *Abs-finfun* (*curry f a*)) ∈ *finfun*
including *finfun*
⟨*proof*⟩

lemma *finfun-curry-conv-curry*:
fixes *f* :: ('*a* × '*b*) ⇒*f* '*c*
shows *finfun-curry f* = *Abs-finfun* (λ*a*. *Abs-finfun* (*curry* (*finfun-apply f*) *a*))
including *finfun*
⟨*proof*⟩

37.15 Executable equality for FinFuns

lemma *eq-finfun-All-ext*: (*f* = *g*) ⟷ *finfun-All* ((λ(*x*, *y*). *x* = *y*) ∘$ ($*f*, *g*$))
⟨*proof*⟩

instantiation *finfun* :: ({*card-UNIV*,*equal*},*equal*) *equal* **begin**
definition *eq-finfun-def* [*code*]: *HOL.equal f g* ⟷ *finfun-All* ((λ(*x*, *y*). *x* = *y*) ∘$ ($*f*, *g*$))
instance ⟨*proof*⟩
end

lemma [*code nbe*]:
HOL.equal (*f* :: - ⇒*f* -) *f* ⟷ *True*
⟨*proof*⟩

37.16 An operator that explicitly removes all redundant updates in the generated representations

definition *finfun-clearjunk* :: '*a* ⇒*f* '*b* ⇒ '*a* ⇒*f* '*b*
where [*simp*, *code del*]: *finfun-clearjunk* = *id*

lemma *finfun-clearjunk-const* [*code*]: *finfun-clearjunk* (*K*$ *b*) = (*K*$ *b*)
⟨*proof*⟩

lemma *finfun-clearjunk-update* [*code*]:
finfun-clearjunk (*finfun-update-code f a b*) = *f*(*a* $:= *b*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)))
⟨*proof*⟩

lemma *finfun-dom-finfunI*: (λ*a*. *f* $ *a* ≠ *finfun-default f*) ∈ *finfun*
⟨*proof*⟩

lemma *finfun-dom-update* [*simp*]:
finfun-dom (*f*(*a* $:= *b*)) = (*finfun-dom f*)(*a* $:= (*b* ≠ *finfun-default f*))
including *finfun* ⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *finite-finfun-dom*: *finite* {*x*. *finfun-dom f* $ *x*}
⟨*proof*⟩

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*⟩

lemma *finfun-const-False-conv-bot*: *op* $ (*K*$ *False*) = *bot*
⟨*proof*⟩

lemma *finfun-const-True-conv-top*: *op* $ (*K*$ *True*) = *top*
⟨*proof*⟩

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*

$\wedge$ *sorted xs* $\wedge$ *distinct xs*)
⟨*proof*⟩

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*))))
⟨*proof*⟩

lemma *remove1-insort-insert-same*:
x ∉ *set xs* ⟹ *remove1 x* (*insort-insert x xs*) = *xs*
⟨*proof*⟩

lemma *finfun-dom-conv*:
finfun-dom f $ *x* ⟷ *f* $ *x* ≠ *finfun-default f*
⟨*proof*⟩

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*⟩

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*))
⟨*proof*⟩

More type class instantiations

lemma *card-eq-1-iff*: *card A* = *1* ⟷ *A* ≠ {} $\wedge$ (∀ *x*∈*A*. ∀ *y*∈*A*. *x* = *y*)
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *card-UNIV-finfun*:
defines *F* == *finfun* :: (*'a* ⇒ *'b*) *set*
shows *CARD*(*'a* ⇒*f* *'b*) = (*if CARD*(*'a*) ≠ *0* $\wedge$ *CARD*(*'b*) ≠ *0* $\vee$ *CARD*(*'b*) = *1 then CARD*(*'b*) ^ *CARD*(*'a*) *else 0*)
⟨*proof*⟩

lemma *finite-UNIV-finfun*:
finite (*UNIV* :: (*'a* ⇒*f* *'b*) *set*) ⟷
(*finite* (*UNIV* :: *'a set*) $\wedge$ *finite* (*UNIV* :: *'b set*) $\vee$ *CARD*(*'b*) = *1*)
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

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
⟨*proof*⟩
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 ⟨*proof*⟩
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*)
⟨*proof*⟩

lemma *add-inf-distrib-right*: *inf a b* + *c* = *inf* (*a* + *c*) (*b* + *c*)
⟨*proof*⟩

end

class *semilattice-sup-ab-group-add* = *ordered-ab-group-add* + *semilattice-sup*
begin

lemma *add-sup-distrib-left*: $a + sup\ b\ c = sup\ (a + b)\ (a + c)$
⟨*proof*⟩

lemma *add-sup-distrib-right*: $sup\ a\ b + c = sup\ (a + c)\ (b + c)$
⟨*proof*⟩

end

class *lattice-ab-group-add* = *ordered-ab-group-add* + *lattice*
begin

subclass *semilattice-inf-ab-group-add* ⟨*proof*⟩
subclass *semilattice-sup-ab-group-add* ⟨*proof*⟩

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*⟩

lemma *sup-eq-neg-inf*: $sup\ a\ b = -\ inf\ (-\ a)\ (-\ b)$
⟨*proof*⟩

lemma *neg-inf-eq-sup*: $-\ inf\ a\ b = sup\ (-\ a)\ (-\ b)$
⟨*proof*⟩

lemma *diff-inf-eq-sup*: $a - inf\ b\ c = a + sup\ (-\ b)\ (-\ c)$
⟨*proof*⟩

lemma *neg-sup-eq-inf*: $-\ sup\ a\ b = inf\ (-\ a)\ (-\ b)$
⟨*proof*⟩

lemma *diff-sup-eq-inf*: $a - sup\ b\ c = a + inf\ (-\ b)\ (-\ c)$
⟨*proof*⟩

lemma *add-eq-inf-sup*: $a + b = sup\ a\ b + inf\ a\ b$
⟨*proof*⟩

38.1 Positive Part, Negative Part, Absolute Value

definition *nprt* :: $'a \Rightarrow 'a$
where $nprt\ x = inf\ x\ 0$

definition *pprt* :: $'a \Rightarrow 'a$

where $pprt\ x = sup\ x\ 0$

lemma *pprt-neg*: $pprt\ (-\ x) = -\ nprt\ x$
⟨*proof*⟩

lemma *nprt-neg*: $nprt\ (-\ x) = -\ pprt\ x$
⟨*proof*⟩

lemma *prts*: $a = pprt\ a + nprt\ a$
⟨*proof*⟩

lemma *zero-le-pprt*[*simp*]: $0 \leq pprt\ a$
⟨*proof*⟩

lemma *nprt-le-zero*[*simp*]: $nprt\ a \leq 0$
⟨*proof*⟩

lemma *le-eq-neg*: $a \leq -\ b \longleftrightarrow a + b \leq 0$
(**is** *?l = ?r*)
⟨*proof*⟩

lemma *pprt-0*[*simp*]: $pprt\ 0 = 0$ ⟨*proof*⟩
lemma *nprt-0*[*simp*]: $nprt\ 0 = 0$ ⟨*proof*⟩

lemma *pprt-eq-id* [*simp*, *no-atp*]: $0 \leq x \Longrightarrow pprt\ x = x$
⟨*proof*⟩

lemma *nprt-eq-id* [*simp*, *no-atp*]: $x \leq 0 \Longrightarrow nprt\ x = x$
⟨*proof*⟩

lemma *pprt-eq-0* [*simp*, *no-atp*]: $x \leq 0 \Longrightarrow pprt\ x = 0$
⟨*proof*⟩

lemma *nprt-eq-0* [*simp*, *no-atp*]: $0 \leq x \Longrightarrow nprt\ x = 0$
⟨*proof*⟩

lemma *sup-0-imp-0*:
assumes $sup\ a\ (-\ a) = 0$
shows $a = 0$
⟨*proof*⟩

lemma *inf-0-imp-0*: $inf\ a\ (-\ a) = 0 \Longrightarrow a = 0$
⟨*proof*⟩

lemma *inf-0-eq-0* [*simp*, *no-atp*]: $inf\ a\ (-\ a) = 0 \longleftrightarrow a = 0$
⟨*proof*⟩

lemma *sup-0-eq-0* [*simp*, *no-atp*]: $sup\ a\ (-\ a) = 0 \longleftrightarrow a = 0$
⟨*proof*⟩

lemma *zero-le-double-add-iff-zero-le-single-add* [*simp*]: $0 \leq a + a \longleftrightarrow 0 \leq a$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *double-zero* [*simp*]: $a + a = 0 \longleftrightarrow a = 0$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *zero-less-double-add-iff-zero-less-single-add* [*simp*]: $0 < a + a \longleftrightarrow 0 < a$
⟨*proof*⟩

lemma *double-add-le-zero-iff-single-add-le-zero* [*simp*]: $a + a \leq 0 \longleftrightarrow a \leq 0$
⟨*proof*⟩

lemma *double-add-less-zero-iff-single-less-zero* [*simp*]: $a + a < 0 \longleftrightarrow a < 0$
⟨*proof*⟩

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*⟩

lemma *minus-le-self-iff*: $- a \leq a \longleftrightarrow 0 \leq a$
⟨*proof*⟩

lemma *zero-le-iff-zero-nprt*: $0 \leq a \longleftrightarrow nprt\ a = 0$
⟨*proof*⟩

lemma *le-zero-iff-zero-pprt*: $a \leq 0 \longleftrightarrow pprt\ a = 0$
⟨*proof*⟩

lemma *le-zero-iff-pprt-id*: $0 \leq a \longleftrightarrow pprt\ a = a$
⟨*proof*⟩

lemma *zero-le-iff-nprt-id*: $a \leq 0 \longleftrightarrow nprt\ a = a$
⟨*proof*⟩

lemma *pprt-mono* [*simp*, *no-atp*]: $a \leq b \Longrightarrow pprt\ a \leq pprt\ b$
⟨*proof*⟩

lemma *nprt-mono* [*simp*, *no-atp*]: $a \leq b \Longrightarrow nprt\ a \leq nprt\ b$
⟨*proof*⟩

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*⟩

subclass *ordered-ab-group-add-abs*
⟨*proof*⟩

end

lemma *sup-eq-if*:
  fixes *a* :: 'a::{*lattice-ab-group-add*,*linorder*}
  shows $sup\ a\ (-\ a) = (if\ a < 0\ then\ -\ a\ else\ a)$
  ⟨*proof*⟩

lemma *abs-if-lattice*:
  fixes *a* :: 'a::{*lattice-ab-group-add-abs*,*linorder*}
  shows $|a| = (if\ a < 0\ then\ -\ a\ else\ a)$
  ⟨*proof*⟩

lemma *estimate-by-abs*:
  fixes *a b c* :: 'a::*lattice-ab-group-add-abs*
  assumes $a + b \leq c$
  shows $a \leq c + |b|$
⟨*proof*⟩

class *lattice-ring* = *ordered-ring* + *lattice-ab-group-add-abs*
begin

subclass *semilattice-inf-ab-group-add* ⟨*proof*⟩
subclass *semilattice-sup-ab-group-add* ⟨*proof*⟩

end

lemma *abs-le-mult*:
  fixes *a b* :: 'a::*lattice-ring*
  shows $|a * b| \leq |a| * |b|$
⟨*proof*⟩

instance *lattice-ring* ⊆ *ordered-ring-abs*
⟨*proof*⟩

lemma *mult-le-prts*:
  fixes *a b* :: 'a::*lattice-ring*
  assumes $a1 \leq a$

and $a \leq a2$
and $b1 \leq b$
and $b \leq b2$
shows $a * b \leq$
$pprt\ a2 * pprt\ b2 + pprt\ a1 * nprt\ b2 + nprt\ a2 * pprt\ b1 + nprt\ a1 * nprt\ b1$
⟨*proof*⟩

lemma *mult-ge-prts*:
fixes $a\ b :: 'a$::*lattice-ring*
assumes $a1 \leq a$
and $a \leq a2$
and $b1 \leq b$
and $b \leq b2$
shows $a * b \geq$
$nprt\ a1 * pprt\ b2 + nprt\ a2 * nprt\ b2 + pprt\ a1 * pprt\ b1 + pprt\ a2 * nprt\ b1$
⟨*proof*⟩

instance *int* :: *lattice-ring*
⟨*proof*⟩

instance *real* :: *lattice-ring*
⟨*proof*⟩

end

39 Floating-Point Numbers

theory *Float*
imports *Complex-Main Lattice-Algebras*
begin

definition $float = \{m * 2\ powr\ e \mid (m :: int)\ (e :: int).\ True\}$

typedef *float* = *float*
morphisms *real-of-float float-of*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *zero-float*[*simp*]: *0* ∈ *float*
⟨*proof*⟩
lemma *one-float*[*simp*]: *1* ∈ *float*
⟨*proof*⟩
lemma *numeral-float*[*simp*]: *numeral i* ∈ *float*
⟨*proof*⟩
lemma *neg-numeral-float*[*simp*]: − *numeral i* ∈ *float*
⟨*proof*⟩
lemma *real-of-int-float*[*simp*]: *real-of-int* (*x* :: *int*) ∈ *float*
⟨*proof*⟩
lemma *real-of-nat-float*[*simp*]: *real* (*x* :: *nat*) ∈ *float*
⟨*proof*⟩
lemma *two-powr-int-float*[*simp*]: *2 powr* (*real-of-int* (*i*::*int*)) ∈ *float*
⟨*proof*⟩
lemma *two-powr-nat-float*[*simp*]: *2 powr* (*real* (*i*::*nat*)) ∈ *float*
⟨*proof*⟩
lemma *two-powr-minus-int-float*[*simp*]: *2 powr* − (*real-of-int* (*i*::*int*)) ∈ *float*
⟨*proof*⟩
lemma *two-powr-minus-nat-float*[*simp*]: *2 powr* − (*real* (*i*::*nat*)) ∈ *float*
⟨*proof*⟩
lemma *two-powr-numeral-float*[*simp*]: *2 powr numeral i* ∈ *float*
⟨*proof*⟩
lemma *two-powr-neg-numeral-float*[*simp*]: *2 powr* − *numeral i* ∈ *float*
⟨*proof*⟩
lemma *two-pow-float*[*simp*]: *2* ^ *n* ∈ *float*
⟨*proof*⟩

lemma *plus-float*[*simp*]: *r* ∈ *float* ⟹ *p* ∈ *float* ⟹ *r* + *p* ∈ *float*
⟨*proof*⟩

lemma *uminus-float*[*simp*]: *x* ∈ *float* ⟹ −*x* ∈ *float*
⟨*proof*⟩

lemma *times-float*[*simp*]: *x* ∈ *float* ⟹ *y* ∈ *float* ⟹ *x* ∗ *y* ∈ *float*
⟨*proof*⟩

lemma *minus-float*[*simp*]: $x \in \mathit{float} \Longrightarrow y \in \mathit{float} \Longrightarrow x - y \in \mathit{float}$
⟨*proof*⟩

lemma *abs-float*[*simp*]: $x \in \mathit{float} \Longrightarrow |x| \in \mathit{float}$
⟨*proof*⟩

lemma *sgn-of-float*[*simp*]: $x \in \mathit{float} \Longrightarrow \mathit{sgn}\ x \in \mathit{float}$
⟨*proof*⟩

lemma *div-power-2-float*[*simp*]: $x \in \mathit{float} \Longrightarrow x\ /\ 2\hat{}d \in \mathit{float}$
⟨*proof*⟩

lemma *div-power-2-int-float*[*simp*]: $x \in \mathit{float} \Longrightarrow x\ /\ (2::\mathit{int})\hat{}d \in \mathit{float}$
⟨*proof*⟩

lemma *div-numeral-Bit0-float*[*simp*]:
assumes *x*: $x\ /\ \mathit{numeral}\ n \in \mathit{float}$
shows $x\ /\ (\mathit{numeral}\ (\mathit{Num.Bit0}\ n)) \in \mathit{float}$
⟨*proof*⟩

lemma *div-neg-numeral-Bit0-float*[*simp*]:
assumes *x*: $x\ /\ \mathit{numeral}\ n \in \mathit{float}$
shows $x\ /\ (-\ \mathit{numeral}\ (\mathit{Num.Bit0}\ n)) \in \mathit{float}$
⟨*proof*⟩

lemma *power-float*[*simp*]:
assumes $a \in \mathit{float}$
shows $a\ \hat{}\ b \in \mathit{float}$
⟨*proof*⟩

lift-definition *Float* :: *int* ⇒ *int* ⇒ *float* **is** $\lambda(m::\mathit{int})\ (e::\mathit{int}).\ m * 2\ \mathit{powr}\ e$
⟨*proof*⟩
declare *Float.rep-eq*[*simp*]

code-datatype *Float*

lemma *compute-real-of-float*[*code*]:
$\mathit{real\text{-}of\text{-}float}\ (\mathit{Float}\ m\ e) = (\mathit{if}\ e \geq 0\ \mathit{then}\ m * 2\ \hat{}\ \mathit{nat}\ e\ \mathit{else}\ m\ /\ 2\ \hat{}\ (\mathit{nat}\ (-e)))$
⟨*proof*⟩

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* ⟨*proof*⟩
declare *zero-float.rep-eq*[*simp*]

lift-definition *one-float* :: *float* **is** *1* ⟨*proof*⟩
declare *one-float.rep-eq*[*simp*]
lift-definition *plus-float* :: *float* ⇒ *float* ⇒ *float* **is** *op +* ⟨*proof*⟩
declare *plus-float.rep-eq*[*simp*]
lift-definition *times-float* :: *float* ⇒ *float* ⇒ *float* **is** *op ∗* ⟨*proof*⟩
declare *times-float.rep-eq*[*simp*]
lift-definition *minus-float* :: *float* ⇒ *float* ⇒ *float* **is** *op −* ⟨*proof*⟩
declare *minus-float.rep-eq*[*simp*]
lift-definition *uminus-float* :: *float* ⇒ *float* **is** *uminus* ⟨*proof*⟩
declare *uminus-float.rep-eq*[*simp*]

lift-definition *abs-float* :: *float* ⇒ *float* **is** *abs* ⟨*proof*⟩
declare *abs-float.rep-eq*[*simp*]
lift-definition *sgn-float* :: *float* ⇒ *float* **is** *sgn* ⟨*proof*⟩
declare *sgn-float.rep-eq*[*simp*]

lift-definition *equal-float* :: *float* ⇒ *float* ⇒ *bool* **is** *op =* :: *real* ⇒ *real* ⇒ *bool*
⟨*proof*⟩

lift-definition *less-eq-float* :: *float* ⇒ *float* ⇒ *bool* **is** *op ≤* ⟨*proof*⟩
declare *less-eq-float.rep-eq*[*simp*]
lift-definition *less-float* :: *float* ⇒ *float* ⇒ *bool* **is** *op <* ⟨*proof*⟩
declare *less-float.rep-eq*[*simp*]

instance
⟨*proof*⟩

end

lemma *real-of-float* [*simp*]: *real-of-float* (*of-nat n*) = *of-nat n*
⟨*proof*⟩

lemma *real-of-float-of-int-eq* [*simp*]: *real-of-float* (*of-int z*) = *of-int z*
⟨*proof*⟩

lemma *Float-0-eq-0*[*simp*]: *Float 0 e* = *0*
⟨*proof*⟩

lemma *real-of-float-power*[*simp*]:
fixes *f* :: *float*
shows *real-of-float* (*f^n*) = *real-of-float f^n*
⟨*proof*⟩

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*)

⟨*proof*⟩

instance *float* :: *unbounded-dense-linorder*
⟨*proof*⟩

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
⟨*proof*⟩

end

lemma *float-numeral*[*simp*]: *real-of-float* (*numeral x* :: *float*) = *numeral x*
⟨*proof*⟩

lemma *transfer-numeral* [*transfer-rule*]:
rel-fun (*op* =) *pcr-float* (*numeral* :: - ⇒ *real*) (*numeral* :: - ⇒ *float*)
⟨*proof*⟩

lemma *float-neg-numeral*[*simp*]: *real-of-float* (− *numeral x* :: *float*) = − *numeral x*
⟨*proof*⟩

lemma *transfer-neg-numeral* [*transfer-rule*]:
rel-fun (*op* =) *pcr-float* (− *numeral* :: - ⇒ *real*) (− *numeral* :: - ⇒ *float*)
⟨*proof*⟩

lemma
shows *float-of-numeral*[*simp*]: *numeral k* = *float-of* (*numeral k*)
and *float-of-neg-numeral*[*simp*]: − *numeral k* = *float-of* (− *numeral k*)
⟨*proof*⟩

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 ⟨*proof*⟩

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 ⟨*proof*⟩

end

instantiation *float* :: *random*
begin

definition *Quickcheck-Random.random i* =
scomp (*Quickcheck-Random.random* (*2* ^ *nat-of-natural i*))
(λ*man*. *scomp* (*Quickcheck-Random.random i*) (λ*exp*. *Pair* (*valtermify-float man exp*)))

instance ⟨*proof*⟩

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*⟩

lemma *int-cancel-factors*:
fixes *n* :: *int*
assumes *1* < *r*
shows *n* = *0* ∨ (∃ *k i*. *n* = *k* ∗ *r* ^ *i* ∧ ¬ *r dvd k*)
⟨*proof*⟩

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*⟩

lemma *mult-powr-eq-mult-powr-iff*:
fixes *m1 m2 e1 e2* :: *int*
shows *¬ 2 dvd m1 ⟹ ¬ 2 dvd m2 ⟹ m1 * 2 powr e1 = m2 * 2 powr e2 ⟷ m1 = m2 ∧ e1 = e2*
⟨*proof*⟩

lemma *floatE-normed*:
assumes *x*: *x ∈ float*
obtains (*zero*) *x = 0*
| (*powr*) *m e* :: *int* **where** *x = m * 2 powr e ¬ 2 dvd m x ≠ 0*
⟨*proof*⟩

lemma *float-normed-cases*:
fixes *f* :: *float*
obtains (*zero*) *f = 0*
| (*powr*) *m e* :: *int* **where** *real-of-float f = m * 2 powr e ¬ 2 dvd m f ≠ 0*
⟨*proof*⟩

definition *mantissa* :: *float ⇒ int* **where**
*mantissa f = fst (SOME 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))*

definition *exponent* :: *float ⇒ int* **where**
*exponent f = snd (SOME 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))*

lemma
shows *exponent-0*[*simp*]: *exponent (float-of 0) = 0* (**is** *?E*)
and *mantissa-0*[*simp*]: *mantissa (float-of 0) = 0* (**is** *?M*)
⟨*proof*⟩

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*⟩

lemma *mantissa-noteq-0*: *f ≠ float-of 0 ⟹ mantissa f ≠ 0*
⟨*proof*⟩

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*⟩

39.5 Compute arithmetic operations

lemma *Float-mantissa-exponent*: *Float* (*mantissa f*) (*exponent f*) = *f*
⟨*proof*⟩

lemma *Float-cases* [*cases type*: *float*]:
fixes *f* :: *float*
obtains (*Float*) *m e* :: *int* **where** *f = Float m e*
⟨*proof*⟩

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*⟩

context
begin

qualified lemma *compute-float-zero*[*code-unfold*, *code*]: *0 = Float 0 0*
⟨*proof*⟩ **lemma** *compute-float-one*[*code-unfold*, *code*]: *1 = Float 1 0*
⟨*proof*⟩

lift-definition *normfloat* :: *float ⇒ float* **is** *λx. x* ⟨*proof*⟩
lemma *normloat-id*[*simp*]: *normfloat x = x* ⟨*proof*⟩ **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)
⟨*proof*⟩ **lemma** *compute-float-numeral*[*code-abbrev*]: *Float* (*numeral k*) *0 = numeral k*
⟨*proof*⟩ **lemma** *compute-float-neg-numeral*[*code-abbrev*]: *Float* (*− numeral k*) *0 = − numeral k*
⟨*proof*⟩ **lemma** *compute-float-uminus*[*code*]: *− Float m1 e1 = Float* (*− m1*) *e1*
⟨*proof*⟩ **lemma** *compute-float-times*[*code*]: *Float m1 e1 * Float m2 e2 = Float* (*m1 * m2*) (*e1 + e2*)
⟨*proof*⟩ **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^nat* (*e2 − e1*)) *e1*
else Float (*m2 + m1 * 2^nat* (*e1 − e2*)) *e2*)
⟨*proof*⟩ **lemma** *compute-float-minus*[*code*]: **fixes** *f g*::*float* **shows** *f − g = f +* (*−g*)
⟨*proof*⟩ **lemma** *compute-float-sgn*[*code*]: *sgn* (*Float m1 e1*) = (*if 0 < m1 then 1 else if m1 < 0 then −1 else 0*)
⟨*proof*⟩

lift-definition *is-float-pos* :: *float* ⇒ *bool* **is** *op* < *0* :: *real* ⇒ *bool* ⟨*proof*⟩ **lemma** *compute-is-float-pos*[*code*]: *is-float-pos* (*Float m e*) ⟷ *0* < *m*
⟨*proof*⟩

lift-definition *is-float-nonneg* :: *float* ⇒ *bool* **is** *op* ≤ *0* :: *real* ⇒ *bool* ⟨*proof*⟩ **lemma** *compute-is-float-nonneg*[*code*]: *is-float-nonneg* (*Float m e*) ⟷ *0* ≤ *m*
⟨*proof*⟩

lift-definition *is-float-zero* :: *float* ⇒ *bool* **is** *op* = *0* :: *real* ⇒ *bool* ⟨*proof*⟩ **lemma** *compute-is-float-zero*[*code*]: *is-float-zero* (*Float m e*) ⟷ *0* = *m*
⟨*proof*⟩ **lemma** *compute-float-abs*[*code*]: |*Float m e*| = *Float* |*m*| *e*
⟨*proof*⟩ **lemma** *compute-float-eq*[*code*]: *equal-class.equal f g* = *is-float-zero* (*f* − *g*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *round-up-float*[*simp*]: *round-up prec x* ∈ *float*
⟨*proof*⟩

lemma *round-up*: *x* ≤ *round-up prec x*
⟨*proof*⟩

lemma *round-down*: *round-down prec x* ≤ *x*
⟨*proof*⟩

lemma *round-up-0*[*simp*]: *round-up p 0 = 0*
⟨*proof*⟩

lemma *round-down-0*[*simp*]: *round-down p 0 = 0*
⟨*proof*⟩

lemma *round-up-diff-round-down*:
round-up prec x − round-down prec x ≤ 2 powr −prec
⟨*proof*⟩

lemma *round-down-shift*: *round-down p (x * 2 powr k) = 2 powr k * round-down (p + k) x*
⟨*proof*⟩

lemma *round-up-shift*: *round-up p (x * 2 powr k) = 2 powr k * round-up (p + k) x*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *round-up-mono*: *x ≤ y ⟹ round-up p x ≤ round-up p y*
⟨*proof*⟩

lemma *round-up-le1*:
assumes *x ≤ 1 prec ≥ 0*
shows *round-up prec x ≤ 1*
⟨*proof*⟩

lemma *round-up-less1*:
assumes *x < 1 / 2 p > 0*
shows *round-up p x < 1*
⟨*proof*⟩

lemma *round-down-ge1*:
assumes *x*: *x ≥ 1*
assumes *prec*: *p ≥ − log 2 x*
shows *1 ≤ round-down p x*
⟨*proof*⟩

lemma *round-up-le0*: *x ≤ 0 ⟹ round-up p x ≤ 0*
⟨*proof*⟩

39.8 Rounding Floats

definition *div-twopow* :: *int ⇒ nat ⇒ int*
where [*simp*]: *div-twopow x n = x div (2 ^ n)*

definition *mod-twopow* :: *int* ⇒ *nat* ⇒ *int*
where [*simp*]: *mod-twopow x n* = *x mod* (*2* ^ *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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

lift-definition *float-up* :: *int* ⇒ *float* ⇒ *float* **is** *round-up* ⟨*proof*⟩
declare *float-up.rep-eq*[*simp*]

lemma *round-up-correct*: *round-up e f* − *f* ∈ {*0..2 powr* −*e*}
⟨*proof*⟩

lemma *float-up-correct*: *real-of-float* (*float-up e f*) − *real-of-float f* ∈ {*0..2 powr* −*e*}
⟨*proof*⟩

lift-definition *float-down* :: *int* ⇒ *float* ⇒ *float* **is** *round-down* ⟨*proof*⟩
declare *float-down.rep-eq*[*simp*]

lemma *round-down-correct*: *f* − (*round-down e f*) ∈ {*0..2 powr* −*e*}
⟨*proof*⟩

lemma *float-down-correct*: *real-of-float f* − *real-of-float* (*float-down e f*) ∈ {*0..2 powr* −*e*}
⟨*proof*⟩

context
begin

qualified lemma *compute-float-down*[*code*]:
float-down p (*Float m e*) =
(*if p* + *e* < *0 then Float* (*div-twopow m* (*nat* (−(*p* + *e*)))) (−*p*) *else Float m e*)
⟨*proof*⟩

lemma *abs-round-down-le*: |*f* − (*round-down e f*)| ≤ *2 powr* −*e*
⟨*proof*⟩

lemma *abs-round-up-le*: |*f* − (*round-up e f*)| ≤ *2 powr* −*e*
⟨*proof*⟩

lemma *round-down-nonneg*: *0* ≤ *s* ⟹ *0* ≤ *round-down p s*

⟨*proof*⟩

lemma *ceil-divide-floor-conv*:
assumes $b \neq 0$
shows $\lceil \textit{real-of-int } a \mathbin{/} \textit{real-of-int } b \rceil = (\textit{if } b \textit{ dvd } a \textit{ then } a \textit{ div } b \textit{ else } \lfloor \textit{real-of-int } a \mathbin{/} \textit{real-of-int } b \rfloor + 1)$
⟨*proof*⟩ **lemma** *compute-float-up*[*code*]: $\textit{float-up } p\ x = -\textit{ float-down } p\ (-x)$
⟨*proof*⟩

end

39.9 Compute bitlen of integers

definition *bitlen* :: $\textit{int} \Rightarrow \textit{int}$
where $\textit{bitlen } a = (\textit{if } a > 0 \textit{ then } \lfloor \textit{log } 2\ a \rfloor + 1 \textit{ else } 0)$

lemma *bitlen-nonneg*: $0 \leq \textit{bitlen } x$
⟨*proof*⟩

lemma *bitlen-bounds*:
assumes $x > 0$
shows $2 \mathbin{\hat{}} \textit{nat } (\textit{bitlen } x - 1) \leq x \wedge x < 2 \mathbin{\hat{}} \textit{nat } (\textit{bitlen } x)$
⟨*proof*⟩

lemma *bitlen-pow2*[*simp*]:
assumes $b > 0$
shows $\textit{bitlen } (b * 2 \mathbin{\hat{}} c) = \textit{bitlen } b + c$
⟨*proof*⟩

lemma *bitlen-Float*:
fixes $m\ e$
defines $f \equiv \textit{Float } m\ e$
shows $\textit{bitlen } (|\textit{mantissa } f|) + \textit{exponent } f = (\textit{if } m = 0 \textit{ then } 0 \textit{ else bitlen } |m| + e)$
⟨*proof*⟩

context
begin

qualified lemma *compute-bitlen*[*code*]: $\textit{bitlen } x = (\textit{if } x > 0 \textit{ then bitlen } (x \textit{ div } 2) + 1 \textit{ else } 0)$
⟨*proof*⟩

end

lemma *float-gt1-scale*: **assumes** $1 \leq \textit{Float } m\ e$
shows $0 \leq e + (\textit{bitlen } m - 1)$
⟨*proof*⟩

lemma *bitlen-div*:
assumes $0 < m$
shows $1 \leq$ *real-of-int* m / $2\hat{}$*nat* (*bitlen* $m - 1$)
and *real-of-int* m / $2\hat{}$*nat* (*bitlen* $m - 1$) < 2
⟨*proof*⟩

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* $x \leq x$
⟨*proof*⟩

lemma *truncate-down-le*: $x \leq y \Longrightarrow$ *truncate-down prec* $x \leq y$
⟨*proof*⟩

lemma *truncate-down-zero*[*simp*]: *truncate-down prec* 0 = 0
⟨*proof*⟩

lemma *truncate-down-float*[*simp*]: *truncate-down p* $x \in$ *float*
⟨*proof*⟩

definition *truncate-up*::*nat* ⇒ *real* ⇒ *real*
where *truncate-up prec* x = *round-up* (*prec* − ⌊*log* 2 ⌊|x|⌋⌋) x

lemma *truncate-up*: $x \leq$ *truncate-up prec* x
⟨*proof*⟩

lemma *truncate-up-le*: $x \leq y \Longrightarrow x \leq$ *truncate-up prec* y
⟨*proof*⟩

lemma *truncate-up-zero*[*simp*]: *truncate-up prec* 0 = 0
⟨*proof*⟩

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
⟨*proof*⟩

lemma *truncate-up-float*[*simp*]: *truncate-up p* $x \in$ *float*
⟨*proof*⟩

lemma *mult-powr-eq*: $0 < b \Longrightarrow b \neq 1 \Longrightarrow 0 < x \Longrightarrow x * b$ *powr* $y = b$ *powr* (y + *log* b x)
⟨*proof*⟩

lemma *truncate-down-pos*:
assumes $x > 0$
shows *truncate-down p* $x > 0$

⟨*proof*⟩

lemma *truncate-down-nonneg*: $0 \leq y \Longrightarrow 0 \leq$ *truncate-down prec y*
⟨*proof*⟩

lemma *truncate-down-ge1*: $1 \leq x \Longrightarrow 1 \leq$ *truncate-down p x*
⟨*proof*⟩

lemma *truncate-up-nonpos*: $x \leq 0 \Longrightarrow$ *truncate-up prec x* ≤ 0
⟨*proof*⟩

lemma *truncate-up-le1*:
assumes $x \leq 1$
shows *truncate-up p x* ≤ 1
⟨*proof*⟩

lemma *truncate-down-shift-int*: *truncate-down p* $(x *$ *2 powr real-of-int k*$)$ = *truncate-down p x* $*$ *2 powr k*
⟨*proof*⟩

lemma *truncate-down-shift-nat*: *truncate-down p* $(x *$ *2 powr real k*$)$ = *truncate-down p x* $*$ *2 powr k*
⟨*proof*⟩

lemma *truncate-up-shift-int*: *truncate-up p* $(x *$ *2 powr real-of-int k*$)$ = *truncate-up p x* $*$ *2 powr k*
⟨*proof*⟩

lemma *truncate-up-shift-nat*: *truncate-up p* $(x *$ *2 powr real k*$)$ = *truncate-up p x* $*$ *2 powr k*
⟨*proof*⟩

39.11 Truncating Floats

lift-definition *float-round-up* :: *nat* ⇒ *float* ⇒ *float* **is** *truncate-up*
⟨*proof*⟩

lemma *float-round-up*: *real-of-float x* $\leq$ *real-of-float* (*float-round-up prec x*)
⟨*proof*⟩

lemma *float-round-up-zero*[*simp*]: *float-round-up prec* $0 = 0$
⟨*proof*⟩

lift-definition *float-round-down* :: *nat* ⇒ *float* ⇒ *float* **is** *truncate-down*
⟨*proof*⟩

lemma *float-round-down*: *real-of-float* (*float-round-down prec x*) $\leq$ *real-of-float x*
⟨*proof*⟩

lemma *float-round-down-zero*[*simp*]: *float-round-down prec 0 = 0*
⟨*proof*⟩

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)*
⟨*proof*⟩

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)
⟨*proof*⟩ **lemma** *compute-float-round-up*[*code*]:
float-round-up prec x = − float-round-down prec (−x)
⟨*proof*⟩

end

39.12 Approximation of positive rationals

lemma *div-mult-twopow-eq*:
fixes *a b* :: *nat*
shows *a div ((2*::*nat) ^ n) div b = a div (b * 2 ^ n)*
⟨*proof*⟩

lemma *real-div-nat-eq-floor-of-divide*:
fixes *a b* :: *nat*
shows *a div b = real-of-int ⌊a / b⌋*
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

lift-definition *lapprox-posrat* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *float*
is λ*prec* (*x*::*nat*) (*y*::*nat*). *truncate-down prec* (*x / y*)
⟨*proof*⟩

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^*nat l div y else x div* 2^*nat* (− *l*) *div y*
in normfloat (*Float d* (− *l*)))
⟨*proof*⟩

end

lift-definition *rapprox-posrat* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *float*
is λ*prec* (*x*::*nat*) (*y*::*nat*). *truncate-up prec* (*x / y*)
⟨*proof*⟩

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*⟩

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*⟩

lemma *rapprox-posrat-less1*:
*0 ≤ x ⟹ 0 < y ⟹ 2 * x < y ⟹ real-of-float (rapprox-posrat n x y) < 1*
⟨*proof*⟩

lift-definition *lapprox-rat* :: *nat ⇒ int ⇒ int ⇒ float* **is**
λprec (x::int) (y::int). truncate-down prec (x / y)
⟨*proof*⟩

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))))
⟨*proof*⟩

lift-definition *rapprox-rat* :: *nat ⇒ int ⇒ int ⇒ float* **is**
λprec (x::int) (y::int). truncate-up prec (x / y)
⟨*proof*⟩

lemma *rapprox-rat = rapprox-posrat*
⟨*proof*⟩

lemma *lapprox-rat = lapprox-posrat*
⟨*proof*⟩ **lemma** *compute-rapprox-rat*[*code*]:
rapprox-rat prec x y = − lapprox-rat prec (−x) y
⟨*proof*⟩ **lemma** *compute-truncate-down*[*code*]: *truncate-down p (Ratreal r) = (let (a, b) = quotient-of r in lapprox-rat p a b)*
⟨*proof*⟩ **lemma** *compute-truncate-up*[*code*]: *truncate-up p (Ratreal r) = (let (a, b) = quotient-of r in rapprox-rat p a b)*
⟨*proof*⟩

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*

⟨*proof*⟩

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*)
⟨*proof*⟩

lift-definition *float-divr* :: *nat* ⇒ *float* ⇒ *float* ⇒ *float* **is** *real-divr*
⟨*proof*⟩ **lemma** *compute-float-divr*[*code*]:
float-divr prec x y = − *float-divl prec* (−*x*) *y*
⟨*proof*⟩

end

39.14 Approximate Power

lemma *div2-less-self*[*termination-simp*]:
fixes *n* :: *nat*
shows *odd n* ⟹ *n div 2* < *n*
⟨*proof*⟩

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*
⟨*proof*⟩

lift-definition *power-down-fl* :: *nat* ⇒ *float* ⇒ *nat* ⇒ *float* **is** *power-down*
⟨*proof*⟩

lemma *power-float-transfer*[*transfer-rule*]:
(*rel-fun pcr-float* (*rel-fun op* = *pcr-float*)) *op* ^ *op* ^
⟨*proof*⟩

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))*
⟨*proof*⟩

lemma *power-down-pos*: *0 < x ⟹ 0 < power-down p x n*
⟨*proof*⟩

lemma *power-down-nonneg*: *0 ≤ x ⟹ 0 ≤ power-down p x n*
⟨*proof*⟩

lemma *power-down*: *0 ≤ x ⟹ power-down p x n ≤ x ^ n*
⟨*proof*⟩

lemma *power-up*: *0 ≤ x ⟹ x ^ n ≤ power-up p x n*
⟨*proof*⟩

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 ≤ x ⟹ power-down-fl p x n ≤ x ^ n*
⟨*proof*⟩

lemma *power-up-fl*: *0 ≤ x ⟹ x ^ n ≤ power-up-fl p x n*
⟨*proof*⟩

lemma *real-power-up-fl*: *real-of-float (power-up-fl p x n) = power-up p x n*
⟨*proof*⟩

lemma *real-power-down-fl*: *real-of-float (power-down-fl p x n) = power-down p x n*
⟨*proof*⟩

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 ∈ float ⟹ y ∈ float ⟹ plus-down p x y ∈ float*
⟨*proof*⟩

lemma *float-plus-up-float*[*intro*, *simp*]: $x \in \mathit{float} \Longrightarrow y \in \mathit{float} \Longrightarrow \mathit{plus\text{-}up}\ p\ x\ y \in \mathit{float}$
⟨*proof*⟩

lift-definition *float-plus-down*::*nat* ⇒ *float* ⇒ *float* ⇒ *float* **is** *plus-down* ⟨*proof*⟩

lift-definition *float-plus-up*::*nat* ⇒ *float* ⇒ *float* ⇒ *float* **is** *plus-up* ⟨*proof*⟩

lemma *plus-down*: $\mathit{plus\text{-}down}\ \mathit{prec}\ x\ y \leq x + y$
and *plus-up*: $x + y \leq \mathit{plus\text{-}up}\ \mathit{prec}\ x\ y$
⟨*proof*⟩

lemma *float-plus-down*: $\mathit{real\text{-}of\text{-}float}\ (\mathit{float\text{-}plus\text{-}down}\ \mathit{prec}\ x\ y) \leq x + y$
and *float-plus-up*: $x + y \leq \mathit{real\text{-}of\text{-}float}\ (\mathit{float\text{-}plus\text{-}up}\ \mathit{prec}\ x\ y)$
⟨*proof*⟩

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*]: $\mathit{plus\text{-}up}\ p\ x\ y = -\ \mathit{plus\text{-}down}\ p\ (-x)\ (-y)$
⟨*proof*⟩

lemma *truncate-down-log2-eqI*:
assumes $\lfloor \mathit{log}\ 2\ |x| \rfloor = \lfloor \mathit{log}\ 2\ |y| \rfloor$
assumes $\lfloor x * 2\ \mathit{powr}\ (p - \lfloor \mathit{log}\ 2\ |x| \rfloor) \rfloor = \lfloor y * 2\ \mathit{powr}\ (p - \lfloor \mathit{log}\ 2\ |x| \rfloor) \rfloor$
shows $\mathit{truncate\text{-}down}\ p\ x = \mathit{truncate\text{-}down}\ p\ y$
⟨*proof*⟩

lemma *bitlen-eq-zero-iff*: $\mathit{bitlen}\ x = 0 \longleftrightarrow x \leq 0$
⟨*proof*⟩

lemma *sum-neq-zeroI*:
fixes *a k* :: *real*
shows $|a| \geq k \Longrightarrow |b| < k \Longrightarrow a + b \neq 0$
and $|a| > k \Longrightarrow |b| \leq k \Longrightarrow a + b \neq 0$
⟨*proof*⟩

lemma *abs-real-le-2-powr-bitlen*[*simp*]: $|\mathit{real\text{-}of\text{-}int}\ m2| < 2\ \mathit{powr}\ \mathit{real\text{-}of\text{-}int}\ (\mathit{bitlen}\ |m2|)$
⟨*proof*⟩

lemma *floor-sum-times-2-powr-sgn-eq*:
fixes *ai p q* :: *int*
and *a b* :: *real*
assumes $a * 2\ \mathit{powr}\ p = \mathit{ai}$
and *b-le-1*: $|b * 2\ \mathit{powr}\ (p + 1)| \leq 1$

and *leqp*: $q \leq p$
shows $\lfloor (a + b) * 2\ powr\ q \rfloor = \lfloor (2 * ai + sgn\ b) * 2\ powr\ (q - p - 1) \rfloor$
⟨*proof*⟩

lemma *log2-abs-int-add-less-half-sgn-eq*:
fixes *ai* :: *int*
and *b* :: *real*
assumes $|b| \leq 1/2$
and $ai \neq 0$
shows $\lfloor log\ 2\ |real\text{-}of\text{-}int\ ai + b| \rfloor = \lfloor log\ 2\ |ai + sgn\ b\ /\ 2| \rfloor$
⟨*proof*⟩

context
begin

qualified lemma *compute-far-float-plus-down*:
fixes *m1 e1 m2 e2* :: *int*
and *p* :: *nat*
defines $k1 \equiv Suc\ p - nat\ (bitlen\ |m1|)$
assumes *H*: $bitlen\ |m2| \leq e1 - e2 - k1 - 2\ m1 \neq 0\ m2 \neq 0\ e1 \geq e2$
shows *float-plus-down p* (*Float m1 e1*) (*Float m2 e2*) =
float-round-down p (*Float* (*m1* * 2 ^ (*Suc* (*Suc k1*)) + *sgn m2*) (*e1* − *int k1* − 2))
⟨*proof*⟩

lemma *compute-float-plus-down-naive*[*code*]: *float-plus-down p x y* = *float-round-down p* (*x* + *y*)
⟨*proof*⟩ **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 ^ (*Suc* (*Suc k1*)) + *sgn m2*) (*e1* − *int k1* − 2)))
else float-plus-down p (*Float m2 e2*) (*Float m1 e1*)))
⟨*proof*⟩ **lemma** *compute-float-plus-up*[*code*]: *float-plus-up p x y* = − *float-plus-down p* (−*x*) (−*y*)
⟨*proof*⟩

lemma *mantissa-zero*[*simp*]: *mantissa 0* = *0*
⟨*proof*⟩ **lemma** *compute-float-less*[*code*]: $a < b \longleftrightarrow$ *is-float-pos* (*float-plus-down 0 b* (− *a*))
⟨*proof*⟩ **lemma** *compute-float-le*[*code*]: $a \leq b \longleftrightarrow$ *is-float-nonneg* (*float-plus-down*

0 b (− a))
⟨*proof*⟩

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
⟨*proof*⟩

lemma *real-of-Float-int*[*simp*]: *real-of-float (Float n 0) = real n*
⟨*proof*⟩

lemma *float-zero*[*simp*]: *real-of-float (Float 0 e) = 0*
⟨*proof*⟩

lemma *abs-div-2-less*: *a ≠ 0 ⟹ a ≠ −1 ⟹ |(a::int) div 2| < |a|*
⟨*proof*⟩

lemma *lapprox-rat*: *real-of-float (lapprox-rat prec x y) ≤ real-of-int x / real-of-int y*
⟨*proof*⟩

lemma *mult-div-le*:
fixes *a b* :: *int*
assumes *b > 0*
shows *a ≥ b ∗ (a div b)*
⟨*proof*⟩

lemma *lapprox-rat-nonneg*:
fixes *n x y*
assumes *0 ≤ x* **and** *0 ≤ y*
shows *0 ≤ real-of-float (lapprox-rat n x y)*
⟨*proof*⟩

lemma *rapprox-rat*: *real-of-int x / real-of-int y ≤ real-of-float (rapprox-rat prec x y)*
⟨*proof*⟩

lemma *rapprox-rat-le1*:

fixes *n x y*
assumes *xy*: $0 \leq x\ 0 < y\ x \leq y$
shows *real-of-float* (*rapprox-rat n x y*) ≤ 1
⟨*proof*⟩

lemma *rapprox-rat-nonneg-nonpos*: $0 \leq x \Longrightarrow y \leq 0 \Longrightarrow$ *real-of-float* (*rapprox-rat n x y*) ≤ 0
⟨*proof*⟩

lemma *rapprox-rat-nonpos-nonneg*: $x \leq 0 \Longrightarrow 0 \leq y \Longrightarrow$ *real-of-float* (*rapprox-rat n x y*) ≤ 0
⟨*proof*⟩

lemma *real-divl*: *real-divl prec x y* $\leq x\ /\ y$
⟨*proof*⟩

lemma *real-divr*: $x\ /\ y \leq$ *real-divr prec x y*
⟨*proof*⟩

lemma *float-divl*: *real-of-float* (*float-divl prec x y*) $\leq x\ /\ y$
⟨*proof*⟩

lemma *real-divl-lower-bound*:
$0 \leq x \Longrightarrow 0 \leq y \Longrightarrow 0 \leq$ *real-divl prec x y*
⟨*proof*⟩

lemma *float-divl-lower-bound*:
$0 \leq x \Longrightarrow 0 \leq y \Longrightarrow 0 \leq$ *real-of-float* (*float-divl prec x y*)
⟨*proof*⟩

lemma *exponent-1*: *exponent 1* $= 0$
⟨*proof*⟩

lemma *mantissa-1*: *mantissa 1* $= 1$
⟨*proof*⟩

lemma *bitlen-1*: *bitlen 1* $= 1$
⟨*proof*⟩

lemma *mantissa-eq-zero-iff*: *mantissa x* $= 0 \longleftrightarrow x = 0$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *float-upper-bound*: $x \leq$ *2 powr* (*bitlen* $|$*mantissa x*$|$ $+$ *exponent x*)
⟨*proof*⟩

lemma *real-divl-pos-less1-bound*:
assumes $0 < x\ x \leq 1$
shows $1 \leq$ *real-divl prec 1 x*

⟨*proof*⟩

lemma *float-divl-pos-less1-bound*:
0 < real-of-float x ⟹ real-of-float x ≤ 1 ⟹ prec ≥ 1 ⟹ 1 ≤ real-of-float (float-divl prec 1 x)
⟨*proof*⟩

lemma *float-divr*: *real-of-float x / real-of-float y ≤ real-of-float (float-divr prec x y)*
⟨*proof*⟩

lemma *real-divr-pos-less1-lower-bound*:
assumes *0 < x*
and *x ≤ 1*
shows *1 ≤ real-divr prec 1 x*
⟨*proof*⟩

lemma *float-divr-pos-less1-lower-bound*: *0 < x ⟹ x ≤ 1 ⟹ 1 ≤ float-divr prec 1 x*
⟨*proof*⟩

lemma *real-divr-nonpos-pos-upper-bound*:
x ≤ 0 ⟹ 0 ≤ y ⟹ real-divr prec x y ≤ 0
⟨*proof*⟩

lemma *float-divr-nonpos-pos-upper-bound*:
real-of-float x ≤ 0 ⟹ 0 ≤ real-of-float y ⟹ real-of-float (float-divr prec x y) ≤ 0
⟨*proof*⟩

lemma *real-divr-nonneg-neg-upper-bound*:
0 ≤ x ⟹ y ≤ 0 ⟹ real-divr prec x y ≤ 0
⟨*proof*⟩

lemma *float-divr-nonneg-neg-upper-bound*:
0 ≤ real-of-float x ⟹ real-of-float y ≤ 0 ⟹ real-of-float (float-divr prec x y) ≤ 0
⟨*proof*⟩

lemma *truncate-up-nonneg-mono*:
assumes *0 ≤ x x ≤ y*
shows *truncate-up prec x ≤ truncate-up prec y*
⟨*proof*⟩

lemma *truncate-up-switch-sign-mono*:
assumes *x ≤ 0 0 ≤ y*
shows *truncate-up prec x ≤ truncate-up prec y*
⟨*proof*⟩

lemma *truncate-down-switch-sign-mono*:
assumes $x \leq 0$
and $0 \leq y$
and $x \leq y$
shows *truncate-down prec x* $\leq$ *truncate-down prec y*
⟨*proof*⟩

lemma *truncate-down-nonneg-mono*:
assumes $0 \leq x$ $x \leq y$
shows *truncate-down prec x* $\leq$ *truncate-down prec y*
⟨*proof*⟩

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* $(-x)$
⟨*proof*⟩

lemma *truncate-down-mono*: $x \leq y \Longrightarrow$ *truncate-down p x* $\leq$ *truncate-down p y*
⟨*proof*⟩

lemma *truncate-up-mono*: $x \leq y \Longrightarrow$ *truncate-up p x* $\leq$ *truncate-up p y*
⟨*proof*⟩

lemma *Float-le-zero-iff*: *Float a b* $\leq 0 \longleftrightarrow a \leq 0$
⟨*proof*⟩

lemma *real-of-float-pprt*[*simp*]:
fixes *a* :: *float*
shows *real-of-float* (*pprt a*) $=$ *pprt* (*real-of-float a*)
⟨*proof*⟩

lemma *real-of-float-nprt*[*simp*]:
fixes *a* :: *float*
shows *real-of-float* (*nprt a*) $=$ *nprt* (*real-of-float a*)
⟨*proof*⟩

context
begin

lift-definition *int-floor-fl* :: *float* ⇒ *int* **is** *floor* ⟨*proof*⟩ **lemma** *compute-int-floor-fl*[*code*]:
int-floor-fl (*Float m e*) $=$ (*if* $0 \leq e$ *then* $m * 2$ ^ *nat e else m div* (2 ^ (*nat* $(-e)$)))
⟨*proof*⟩

lift-definition *floor-fl* :: *float* ⇒ *float* **is** $\lambda x.$ *real-of-int* $\lfloor x \rfloor$
⟨*proof*⟩ **lemma** *compute-floor-fl*[*code*]:
floor-fl (*Float m e*) $=$ (*if* $0 \leq e$ *then Float m e else Float* (*m div* (2 ^ (*nat* $(-e)$))) 0)
⟨*proof*⟩

end

lemma *floor-fl*: *real-of-float* (*floor-fl x*) ≤ *real-of-float x*
⟨*proof*⟩

lemma *int-floor-fl*: *real-of-int* (*int-floor-fl x*) ≤ *real-of-float x*
⟨*proof*⟩

lemma *floor-pos-exp*: *exponent* (*floor-fl x*) ≥ *0*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

end

40 Less common functions on lists

theory *More-List*
imports *Main*
begin

definition *strip-while* :: ('*a* ⇒ *bool*) ⇒ '*a list* ⇒ '*a list*
where
strip-while P = *rev* ∘ *dropWhile P* ∘ *rev*

lemma *strip-while-rev* [*simp*]:
strip-while P (*rev xs*) = *rev* (*dropWhile P xs*)
⟨*proof*⟩

lemma *strip-while-Nil* [*simp*]:
strip-while P [] = []
⟨*proof*⟩

lemma *strip-while-append* [*simp*]:
¬ *P x* ⟹ *strip-while P* (*xs* @ [*x*]) = *xs* @ [*x*]
⟨*proof*⟩

lemma *strip-while-append-rec* [*simp*]:
P x ⟹ *strip-while P* (*xs* @ [*x*]) = *strip-while P xs*
⟨*proof*⟩

lemma *strip-while-Cons* [*simp*]:
$\neg\ P\ x \Longrightarrow$ *strip-while* $P\ (x\ \#\ xs) = x\ \#$ *strip-while* $P\ xs$
⟨*proof*⟩

lemma *strip-while-eq-Nil* [*simp*]:
strip-while $P\ xs = [] \longleftrightarrow (\forall\, x{\in}set\ xs.\ P\ x)$
⟨*proof*⟩

lemma *strip-while-eq-Cons-rec*:
strip-while $P\ (x\ \#\ xs) = x\ \#$ *strip-while* $P\ xs \longleftrightarrow \neg\ (P\ x \wedge (\forall\, x{\in}set\ xs.\ P\ x))$
⟨*proof*⟩

lemma *strip-while-not-last* [*simp*]:
$\neg\ P$ (*last* xs) $\Longrightarrow$ *strip-while* $P\ xs = xs$
⟨*proof*⟩

lemma *split-strip-while-append*:
fixes $xs :: 'a\ list$
obtains $ys\ zs :: 'a\ list$
where *strip-while* $P\ xs = ys$ **and** $\forall\, x{\in}set\ zs.\ P\ x$ **and** $xs = ys\ @\ zs$
⟨*proof*⟩

lemma *strip-while-snoc* [*simp*]:
strip-while $P\ (xs\ @\ [x]) =$ (*if* $P\ x$ *then strip-while* $P\ xs$ *else* $xs\ @\ [x]$)
⟨*proof*⟩

lemma *strip-while-map*:
strip-while P (*map* $f\ xs$) = *map* f (*strip-while* $(P \circ f)\ xs$)
⟨*proof*⟩

definition *no-leading* :: $('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow bool$
where
no-leading $P\ xs \longleftrightarrow (xs \neq [] \longrightarrow \neg\ P$ (*hd* xs))

lemma *no-leading-Nil* [*simp*, *intro*!]:
no-leading P []
⟨*proof*⟩

lemma *no-leading-Cons* [*simp*, *intro*!]:
no-leading $P\ (x\ \#\ xs) \longleftrightarrow \neg\ P\ x$
⟨*proof*⟩

lemma *no-leading-append* [*simp*]:
no-leading $P\ (xs\ @\ ys) \longleftrightarrow$ *no-leading* $P\ xs \wedge (xs = [] \longrightarrow$ *no-leading* $P\ ys)$
⟨*proof*⟩

lemma *no-leading-dropWhile* [*simp*]:
no-leading P (*dropWhile* $P\ xs$)

⟨*proof*⟩

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*⟩

lemma *dropWhile-idem-iff*:
dropWhile P xs = *xs* ⟷ *no-leading P xs*
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *no-trailing-Nil* [*simp*, *intro*!]:
no-trailing P []
⟨*proof*⟩

lemma *no-trailing-Cons* [*simp*]:
no-trailing P (*x* # *xs*) ⟷ *no-trailing P xs* ∧ (*xs* = [] ⟶ ¬ *P x*)
⟨*proof*⟩

lemma *no-trailing-append-Cons* [*simp*]:
no-trailing P (*xs* @ *y* # *ys*) ⟷ *no-trailing P* (*y* # *ys*)
⟨*proof*⟩

lemma *no-trailing-strip-while* [*simp*]:
no-trailing P (*strip-while P xs*)
⟨*proof*⟩

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*⟩

lemma *strip-while-idem-iff*:
strip-while P xs = *xs* ⟷ *no-trailing P xs*
⟨*proof*⟩

lemma *no-trailing-map*:
no-trailing P (*map f xs*) = *no-trailing* (*P* ∘ *f*) *xs*

⟨*proof*⟩

lemma *no-trailing-upt* [*simp*]:
no-trailing P [*n*..<*m*] ⟷ (*n* < *m* ⟶ ¬ *P* (*m* − *1*))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *nth-default-beyond*:
length xs ≤ *n* ⟹ *nth-default dflt xs n* = *dflt*
⟨*proof*⟩

lemma *nth-default-Nil* [*simp*]:
nth-default dflt [] *n* = *dflt*
⟨*proof*⟩

lemma *nth-default-Cons*:
nth-default dflt (*x* # *xs*) *n* = (*case n of 0* ⇒ *x* | *Suc n'* ⇒ *nth-default dflt xs n'*)
⟨*proof*⟩

lemma *nth-default-Cons-0* [*simp*]:
nth-default dflt (*x* # *xs*) *0* = *x*
⟨*proof*⟩

lemma *nth-default-Cons-Suc* [*simp*]:
nth-default dflt (*x* # *xs*) (*Suc n*) = *nth-default dflt xs n*
⟨*proof*⟩

lemma *nth-default-replicate-dflt* [*simp*]:
nth-default dflt (*replicate n dflt*) *m* = *dflt*
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *nth-default-append-trailing* [*simp*]:
nth-default dflt (*xs* @ *replicate n dflt*) = *nth-default dflt xs*
⟨*proof*⟩

lemma *nth-default-snoc-default* [*simp*]:

nth-default dflt (xs @ [dflt]) = nth-default dflt xs
⟨*proof*⟩

lemma *nth-default-eq-dflt-iff*:
nth-default dflt xs k = dflt ⟷ (k < length xs ⟶ xs ! k = dflt)
⟨*proof*⟩

lemma *in-enumerate-iff-nth-default-eq*:
x ≠ dflt ⟹ (n, x) ∈ set (enumerate 0 xs) ⟷ nth-default dflt xs n = x
⟨*proof*⟩

lemma *last-conv-nth-default*:
assumes *xs ≠ []*
shows *last xs = nth-default dflt xs (length xs − 1)*
⟨*proof*⟩

lemma *nth-default-map-eq*:
f dflt′ = dflt ⟹ nth-default dflt (map f xs) n = f (nth-default dflt′ xs n)
⟨*proof*⟩

lemma *finite-nth-default-neq-default* [*simp*]:
finite {k. nth-default dflt xs k ≠ dflt}
⟨*proof*⟩

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))
⟨*proof*⟩

lemma *map-nth-default*:
map (nth-default x xs) [0..<length xs] = xs
⟨*proof*⟩

lemma *range-nth-default* [*simp*]:
range (nth-default dflt xs) = insert dflt (set xs)
⟨*proof*⟩

lemma *nth-strip-while*:
assumes *n < length (strip-while P xs)*
shows *strip-while P xs ! n = xs ! n*
⟨*proof*⟩

lemma *length-strip-while-le*:
length (strip-while P xs) ≤ length xs
⟨*proof*⟩

lemma *nth-default-strip-while-dflt* [*simp*]:
nth-default dflt (strip-while (op = dflt) xs) = nth-default dflt xs
⟨*proof*⟩

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*⟩

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 ## [] = []
⟨*proof*⟩

lemma *cCons-Cons-eq* [*simp*]:
x ## y # ys = *x # y # ys*
⟨*proof*⟩

lemma *cCons-append-Cons-eq* [*simp*]:
x ## xs @ y # ys = *x # xs @ y # ys*
⟨*proof*⟩

lemma *cCons-not-0-eq* [*simp*]:
x ≠ *0* ⟹ *x ## xs* = *x # xs*
⟨*proof*⟩

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*⟩

lemma *tl-cCons* [*simp*]:
tl (*x ## xs*) = *xs*
⟨*proof*⟩

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* ⟨*proof*⟩

setup-lifting *type-definition-poly*

lemma *poly-eq-iff*: $p = q \longleftrightarrow (\forall\, n.\ coeff\ p\ n = coeff\ q\ n)$
  ⟨*proof*⟩

lemma *poly-eqI*: $(\bigwedge n.\ coeff\ p\ n = coeff\ q\ n) \Longrightarrow p = q$
  ⟨*proof*⟩

lemma *MOST-coeff-eq-0*: $\forall_\infty\ n.\ coeff\ p\ n = 0$
  ⟨*proof*⟩

41.3 Degree of a polynomial

definition *degree* :: *'a::zero poly* ⇒ *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*⟩

lemma *le-degree*: *coeff p n* ≠ *0* ⟹ *n* ≤ *degree p*
  ⟨*proof*⟩

lemma *degree-le*: $\forall\, i{>}n.$ *coeff p i* = *0* ⟹ *degree p* ≤ *n*
  ⟨*proof*⟩

lemma *less-degree-imp*: *n* < *degree p* ⟹ $\exists\, i{>}n.$ *coeff p i* ≠ *0*
  ⟨*proof*⟩

41.4 The zero polynomial

instantiation *poly* :: (*zero*) *zero*
begin

lift-definition *zero-poly* :: *'a poly*
  is *λ-. 0* ⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *coeff-0* [*simp*]:
  coeff 0 n = *0*

⟨*proof*⟩

lemma *degree-0* [*simp*]:
degree 0 = 0
⟨*proof*⟩

lemma *leading-coeff-neq-0*:
assumes *p ≠ 0*
shows *coeff p (degree p) ≠ 0*
⟨*proof*⟩

lemma *leading-coeff-0-iff* [*simp*]:
coeff p (degree p) = 0 ⟷ p = 0
⟨*proof*⟩

41.5 List-style constructor for polynomials

lift-definition *pCons* :: *'a::zero ⇒ 'a poly ⇒ 'a poly*
is *λa p. case-nat a (coeff p)*
⟨*proof*⟩

lemmas *coeff-pCons = pCons.rep-eq*

lemma *coeff-pCons-0* [*simp*]:
coeff (pCons a p) 0 = a
⟨*proof*⟩

lemma *coeff-pCons-Suc* [*simp*]:
coeff (pCons a p) (Suc n) = coeff p n
⟨*proof*⟩

lemma *degree-pCons-le*:
degree (pCons a p) ≤ Suc (degree p)
⟨*proof*⟩

lemma *degree-pCons-eq*:
p ≠ 0 ⟹ degree (pCons a p) = Suc (degree p)
⟨*proof*⟩

lemma *degree-pCons-0*:
degree (pCons a 0) = 0
⟨*proof*⟩

lemma *degree-pCons-eq-if* [*simp*]:
degree (pCons a p) = (if p = 0 then 0 else Suc (degree p))
⟨*proof*⟩

lemma *pCons-0-0* [*simp*]:
pCons 0 0 = 0

⟨*proof*⟩

lemma *pCons-eq-iff* [*simp*]:
pCons a p = *pCons b q* ⟷ *a* = *b* ∧ *p* = *q*
⟨*proof*⟩

lemma *pCons-eq-0-iff* [*simp*]:
pCons a p = *0* ⟷ *a* = *0* ∧ *p* = *0*
⟨*proof*⟩

lemma *pCons-cases* [*cases type*: *poly*]:
obtains (*pCons*) *a q* **where** *p* = *pCons a q*
⟨*proof*⟩

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*⟩

lemma *degree-eq-zeroE*:
fixes *p* :: *'a*::*zero poly*
assumes *degree p* = *0*
obtains *a* **where** *p* = *pCons a 0*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *Poly-eq-0*:
Poly as = 0 ⟷ (∃ n. as = replicate n 0)
⟨*proof*⟩

lemma *degree-Poly*: *degree (Poly xs) ≤ length xs*
⟨*proof*⟩

definition *coeffs* :: *'a poly ⇒ 'a::zero list*
where
coeffs p = (if p = 0 then [] else map (λi. coeff p i) [0 ..< Suc (degree p)])

lemma *coeffs-eq-Nil* [*simp*]:
coeffs p = [] ⟷ p = 0
⟨*proof*⟩

lemma *not-0-coeffs-not-Nil*:
p ≠ 0 ⟹ coeffs p ≠ []
⟨*proof*⟩

lemma *coeffs-0-eq-Nil* [*simp*]:
coeffs 0 = []
⟨*proof*⟩

lemma *coeffs-pCons-eq-cCons* [*simp*]:
coeffs (pCons a p) = a ## coeffs p
⟨*proof*⟩

lemma *length-coeffs*: *p ≠ 0 ⟹ length (coeffs p) = degree p + 1*
⟨*proof*⟩

lemma *coeffs-nth*:
assumes *p ≠ 0 n ≤ degree p*
shows *coeffs p ! n = coeff p n*
⟨*proof*⟩

lemma *not-0-cCons-eq* [*simp*]:
p ≠ 0 ⟹ a ## coeffs p = a # coeffs p
⟨*proof*⟩

lemma *Poly-coeffs* [*simp*, *code abstype*]:
Poly (coeffs p) = p
⟨*proof*⟩

lemma *coeffs-Poly* [*simp*]:
coeffs (Poly as) = strip-while (HOL.eq 0) as
⟨*proof*⟩

lemma *last-coeffs-not-0*:

$p \neq 0 \Longrightarrow last\ (coeffs\ p) \neq 0$
⟨*proof*⟩

lemma *strip-while-coeffs* [*simp*]:
$strip\text{-}while\ (HOL.eq\ 0)\ (coeffs\ p) = coeffs\ p$
⟨*proof*⟩

lemma *coeffs-eq-iff*:
$p = q \longleftrightarrow coeffs\ p = coeffs\ q$ (**is** $?P \longleftrightarrow ?Q$)
⟨*proof*⟩

lemma *coeff-Poly-eq*:
$coeff\ (Poly\ xs)\ n = nth\text{-}default\ 0\ xs\ n$
⟨*proof*⟩

lemma *nth-default-coeffs-eq*:
$nth\text{-}default\ 0\ (coeffs\ p) = coeff\ p$
⟨*proof*⟩

lemma [*code*]:
$coeff\ p = nth\text{-}default\ 0\ (coeffs\ p)$
⟨*proof*⟩

lemma *coeffs-eqI*:
assumes *coeff*: $\bigwedge n.\ coeff\ p\ n = nth\text{-}default\ 0\ xs\ n$
assumes *zero*: $xs \neq [] \Longrightarrow last\ xs \neq 0$
shows $coeffs\ p = xs$
⟨*proof*⟩

lemma *degree-eq-length-coeffs* [*code*]:
$degree\ p = length\ (coeffs\ p) - 1$
⟨*proof*⟩

lemma *length-coeffs-degree*:
$p \neq 0 \Longrightarrow length\ (coeffs\ p) = Suc\ (degree\ p)$
⟨*proof*⟩

lemma [*code abstract*]:
$coeffs\ 0 = []$
⟨*proof*⟩

lemma [*code abstract*]:
$coeffs\ (pCons\ a\ p) = a\ \#\#\ coeffs\ p$
⟨*proof*⟩

instantiation *poly* :: ({*zero*, *equal*}) *equal*
begin

definition

[*code*]: *HOL.equal* (*p*::'*a poly*) *q* ⟷ *HOL.equal* (*coeffs p*) (*coeffs q*)

instance
⟨*proof*⟩

end

lemma [*code nbe*]: *HOL.equal* (*p* :: - *poly*) *p* ⟷ *True*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *fold-coeffs-pCons-eq* [*simp*]:
f 0 = *id* ⟹ *fold-coeffs f* (*pCons a p*) = *f a* ∘ *fold-coeffs f p*
⟨*proof*⟩

lemma *fold-coeffs-pCons-0-0-eq* [*simp*]:
fold-coeffs f (*pCons 0 0*) = *id*
⟨*proof*⟩

lemma *fold-coeffs-pCons-coeff-not-0-eq* [*simp*]:
a ≠ *0* ⟹ *fold-coeffs f* (*pCons a p*) = *f a* ∘ *fold-coeffs f p*
⟨*proof*⟩

lemma *fold-coeffs-pCons-not-0-0-eq* [*simp*]:
p ≠ *0* ⟹ *fold-coeffs f* (*pCons a p*) = *f a* ∘ *fold-coeffs f p*
⟨*proof*⟩

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
⟨*proof*⟩

lemma *poly-pCons* [*simp*]:
*poly (pCons a p) x = a + x * poly p x*
⟨*proof*⟩

lemma *poly-altdef*:
*poly p (x :: 'a :: {comm-semiring-0, semiring-1}) = (∑ i≤degree p. coeff p i * x ^ i)*
⟨*proof*⟩

lemma *poly-0-coeff-0*: *poly p 0 = coeff p 0*
⟨*proof*⟩

41.11 Monomials

lift-definition *monom* :: *'a ⇒ nat ⇒ 'a::zero poly*
is *λa m n. if m = n then a else 0*
⟨*proof*⟩

lemma *coeff-monom* [*simp*]:
coeff (monom a m) n = (if m = n then a else 0)
⟨*proof*⟩

lemma *monom-0*:
monom a 0 = pCons a 0
⟨*proof*⟩

lemma *monom-Suc*:
monom a (Suc n) = pCons 0 (monom a n)
⟨*proof*⟩

lemma *monom-eq-0* [*simp*]: *monom 0 n = 0*
⟨*proof*⟩

lemma *monom-eq-0-iff* [*simp*]: *monom a n = 0 ⟷ a = 0*
⟨*proof*⟩

lemma *monom-eq-iff* [*simp*]: *monom a n = monom b n ⟷ a = b*
⟨*proof*⟩

lemma *degree-monom-le*: *degree (monom a n) ≤ n*
⟨*proof*⟩

lemma *degree-monom-eq*: *a ≠ 0 ⟹ degree (monom a n) = n*
⟨*proof*⟩

lemma *coeffs-monom* [*code abstract*]:
coeffs (*monom a n*) = (*if a = 0 then* [] *else replicate n 0* @ [*a*])
⟨*proof*⟩

lemma *fold-coeffs-monom* [*simp*]:
a ≠ *0* ⟹ *fold-coeffs f* (*monom a n*) = *f 0* ^^ *n* ∘ *f a*
⟨*proof*⟩

lemma *poly-monom*:
fixes *a x* :: *'a*::{*comm-semiring-1*}
shows *poly* (*monom a n*) *x* = *a* ∗ *x* ^ *n*
⟨*proof*⟩

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 λ*p q n*. *coeff p n* + *coeff q n*
⟨*proof*⟩

lemma *coeff-add* [*simp*]: *coeff* (*p* + *q*) *n* = *coeff p n* + *coeff q n*
⟨*proof*⟩

instance
⟨*proof*⟩

end

instantiation *poly* :: (*cancel-comm-monoid-add*) *cancel-comm-monoid-add*
begin

lift-definition *minus-poly* :: *'a poly* ⇒ *'a poly* ⇒ *'a poly*
is λ*p q n*. *coeff p n* − *coeff q n*
⟨*proof*⟩

lemma *coeff-diff* [*simp*]: *coeff* (*p* − *q*) *n* = *coeff p n* − *coeff q n*
⟨*proof*⟩

instance
⟨*proof*⟩

end

instantiation *poly* :: (*ab-group-add*) *ab-group-add*
begin

lift-definition *uminus-poly* :: *'a poly* ⇒ *'a poly*

is $\lambda p\ n.\ -\ coeff\ p\ n$
⟨*proof*⟩

lemma *coeff-minus* [*simp*]: $coeff\ (-\ p)\ n = -\ coeff\ p\ n$
⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *add-pCons* [*simp*]:
$pCons\ a\ p\ +\ pCons\ b\ q = pCons\ (a\ +\ b)\ (p\ +\ q)$
⟨*proof*⟩

lemma *minus-pCons* [*simp*]:
$-\ pCons\ a\ p = pCons\ (-\ a)\ (-\ p)$
⟨*proof*⟩

lemma *diff-pCons* [*simp*]:
$pCons\ a\ p\ -\ pCons\ b\ q = pCons\ (a\ -\ b)\ (p\ -\ q)$
⟨*proof*⟩

lemma *degree-add-le-max*: $degree\ (p\ +\ q) \leq max\ (degree\ p)\ (degree\ q)$
⟨*proof*⟩

lemma *degree-add-le*:
$[\![degree\ p \leq n;\ degree\ q \leq n]\!] \Longrightarrow degree\ (p\ +\ q) \leq n$
⟨*proof*⟩

lemma *degree-add-less*:
$[\![degree\ p < n;\ degree\ q < n]\!] \Longrightarrow degree\ (p\ +\ q) < n$
⟨*proof*⟩

lemma *degree-add-eq-right*:
$degree\ p < degree\ q \Longrightarrow degree\ (p\ +\ q) = degree\ q$
⟨*proof*⟩

lemma *degree-add-eq-left*:
$degree\ q < degree\ p \Longrightarrow degree\ (p\ +\ q) = degree\ p$
⟨*proof*⟩

lemma *degree-minus* [*simp*]:
$degree\ (-\ p) = degree\ p$
⟨*proof*⟩

lemma *degree-diff-le-max*:
fixes $p\ q :: 'a :: ab\text{-}group\text{-}add\ poly$
shows $degree\ (p\ -\ q) \leq max\ (degree\ p)\ (degree\ q)$

⟨*proof*⟩

lemma *degree-diff-le*:
fixes *p q* :: *'a* :: *ab-group-add poly*
assumes *degree p* ≤ *n* **and** *degree q* ≤ *n*
shows *degree* (*p* − *q*) ≤ *n*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *add-monom*: *monom a n* + *monom b n* = *monom* (*a* + *b*) *n*
⟨*proof*⟩

lemma *diff-monom*: *monom a n* − *monom b n* = *monom* (*a* − *b*) *n*
⟨*proof*⟩

lemma *minus-monom*: − *monom a n* = *monom* (−*a*) *n*
⟨*proof*⟩

lemma *coeff-setsum*: *coeff* (∑ *x*∈*A*. *p x*) *i* = (∑ *x*∈*A*. *coeff* (*p x*) *i*)
⟨*proof*⟩

lemma *monom-setsum*: *monom* (∑ *x*∈*A*. *a x*) *n* = (∑ *x*∈*A*. *monom* (*a x*) *n*)
⟨*proof*⟩

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*⟩

lemma *coeffs-uminus* [*code abstract*]:
coeffs (− *p*) = *map* (λ*a*. − *a*) (*coeffs p*)
⟨*proof*⟩

lemma [*code*]:
fixes *p q* :: *'a*::*ab-group-add poly*
shows *p* − *q* = *p* + − *q*
⟨*proof*⟩

lemma *poly-add* [*simp*]: *poly* (*p* + *q*) *x* = *poly p x* + *poly q x*

⟨*proof*⟩

lemma *poly-minus* [*simp*]:
fixes x :: $'a$::*comm-ring*
shows *poly* $(-p)$ $x = -$ *poly* p x
⟨*proof*⟩

lemma *poly-diff* [*simp*]:
fixes x :: $'a$::*comm-ring*
shows *poly* $(p - q)$ $x =$ *poly* p $x -$ *poly* q x
⟨*proof*⟩

lemma *poly-setsum*: *poly* $(\sum k{\in}A.\ p\ k)$ $x = (\sum k{\in}A.$ *poly* $(p\ k)$ $x)$
⟨*proof*⟩

lemma *degree-setsum-le*: *finite* $S \Longrightarrow (\bigwedge p\ .\ p \in S \Longrightarrow$ *degree* $(f\ p) \leq n)$
$\Longrightarrow$ *degree* $(setsum\ f\ S) \leq n$
⟨*proof*⟩

lemma *poly-as-sum-of-monoms'*:
assumes n: *degree* $p \leq n$
shows $(\sum i{\leq}n.$ *monom* $(coeff\ p\ i)\ i) = p$
⟨*proof*⟩

lemma *poly-as-sum-of-monoms*: $(\sum i{\leq}$*degree* $p.$ *monom* $(coeff\ p\ i)\ i) = p$
⟨*proof*⟩

lemma *Poly-snoc*: *Poly* $(xs\ @\ [x])$ = *Poly* xs + *monom* x $(length\ xs)$
⟨*proof*⟩

41.13 Multiplication by a constant, polynomial multiplication and the unit polynomial

lift-definition *smult* :: $'a$::*comm-semiring-0* $\Rightarrow$ $'a$ *poly* $\Rightarrow$ $'a$ *poly*
is $\lambda a\ p\ n.\ a * coeff\ p\ n$
⟨*proof*⟩

lemma *coeff-smult* [*simp*]:
coeff $(smult\ a\ p)\ n = a * coeff\ p\ n$
⟨*proof*⟩

lemma *degree-smult-le*: *degree* $(smult\ a\ p) \leq$ *degree* p
⟨*proof*⟩

lemma *smult-smult* [*simp*]: *smult* a $(smult\ b\ p) =$ *smult* $(a * b)$ p
⟨*proof*⟩

lemma *smult-0-right* [*simp*]: *smult* $a\ 0 = 0$
⟨*proof*⟩

lemma *smult-0-left* [*simp*]: *smult 0 p = 0*
⟨*proof*⟩

lemma *smult-1-left* [*simp*]: *smult (1::'a::comm-semiring-1) p = p*
⟨*proof*⟩

lemma *smult-add-right*:
smult a (p + q) = smult a p + smult a q
⟨*proof*⟩

lemma *smult-add-left*:
smult (a + b) p = smult a p + smult b p
⟨*proof*⟩

lemma *smult-minus-right* [*simp*]:
smult (a::'a::comm-ring) (− p) = − smult a p
⟨*proof*⟩

lemma *smult-minus-left* [*simp*]:
smult (− a::'a::comm-ring) p = − smult a p
⟨*proof*⟩

lemma *smult-diff-right*:
smult (a::'a::comm-ring) (p − q) = smult a p − smult a q
⟨*proof*⟩

lemma *smult-diff-left*:
smult (a − b::'a::comm-ring) p = smult a p − smult b p
⟨*proof*⟩

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)*
⟨*proof*⟩

lemma *smult-monom*: *smult a (monom b n) = monom (a * b) n*
⟨*proof*⟩

lemma *degree-smult-eq* [*simp*]:
fixes *a* :: *'a::idom*
shows *degree (smult a p) = (if a = 0 then 0 else degree p)*
⟨*proof*⟩

lemma *smult-eq-0-iff* [*simp*]:
fixes *a* :: *'a::idom*

shows *smult a p = 0 ⟷ a = 0 ∨ p = 0*
⟨*proof*⟩

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*))
⟨*proof*⟩

instantiation *poly* :: (*comm-semiring-0*) *comm-semiring-0*
begin

definition
*p * q = fold-coeffs* (*λa p. smult a q + pCons 0 p*) *p 0*

lemma *mult-poly-0-left*: (*0*::*'a poly*) * *q* = *0*
⟨*proof*⟩

lemma *mult-pCons-left* [*simp*]:
*pCons a p * q = smult a q + pCons 0* (*p * q*)
⟨*proof*⟩

lemma *mult-poly-0-right*: *p * (0*::*'a poly*) = *0*
⟨*proof*⟩

lemma *mult-pCons-right* [*simp*]:
*p * pCons a q = smult a p + pCons 0* (*p * q*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *mult-smult-right* [*simp*]:
*p * smult a q = smult a* (*p * q*)
⟨*proof*⟩

lemma *mult-poly-add-left*:
fixes *p q r* :: *'a poly*
shows (*p + q*) * *r* = *p * r + q * r*
⟨*proof*⟩

instance
⟨*proof*⟩

end

instance *poly* :: (*comm-semiring-0-cancel*) *comm-semiring-0-cancel* ⟨*proof*⟩

lemma *coeff-mult*:
coeff (*p* ∗ *q*) *n* = (∑ *i*≤*n*. *coeff p i* ∗ *coeff q* (*n*−*i*))
⟨*proof*⟩

lemma *degree-mult-le*: *degree* (*p* ∗ *q*) ≤ *degree p* + *degree q*
⟨*proof*⟩

lemma *mult-monom*: *monom a m* ∗ *monom b n* = *monom* (*a* ∗ *b*) (*m* + *n*)
⟨*proof*⟩

instantiation *poly* :: (*comm-semiring-1*) *comm-semiring-1*
begin

definition *one-poly-def*: *1* = *pCons 1 0*

instance
⟨*proof*⟩

end

instance *poly* :: (*comm-ring*) *comm-ring* ⟨*proof*⟩

instance *poly* :: (*comm-ring-1*) *comm-ring-1* ⟨*proof*⟩

lemma *coeff-1* [*simp*]: *coeff 1 n* = (*if n* = *0 then 1 else 0*)
⟨*proof*⟩

lemma *monom-eq-1* [*simp*]:
monom 1 0 = *1*
⟨*proof*⟩

lemma *degree-1* [*simp*]: *degree 1* = *0*
⟨*proof*⟩

lemma *coeffs-1-eq* [*simp*, *code abstract*]:
coeffs 1 = [*1*]
⟨*proof*⟩

lemma *degree-power-le*:
degree (*p* ^ *n*) ≤ *degree p* ∗ *n*
⟨*proof*⟩

lemma *poly-smult* [*simp*]:
poly (*smult a p*) *x* = *a* ∗ *poly p x*
⟨*proof*⟩

lemma *poly-mult* [*simp*]:

poly (*p* ∗ *q*) *x* = *poly p x* ∗ *poly q x*
⟨*proof*⟩

lemma *poly-1* [*simp*]:
poly 1 x = *1*
⟨*proof*⟩

lemma *poly-power* [*simp*]:
fixes *p* :: *'a*::{*comm-semiring-1*} *poly*
shows *poly* (*p* ^ *n*) *x* = *poly p x* ^ *n*
⟨*proof*⟩

lemma *poly-setprod*: *poly* (∏ *k*∈*A*. *p k*) *x* = (∏ *k*∈*A*. *poly* (*p k*) *x*)
⟨*proof*⟩

lemma *degree-setprod-setsum-le*: *finite S* ⟹ *degree* (*setprod f S*) ≤ *setsum* (*degree* ∘ *f*) *S*
⟨*proof*⟩

41.14 Conversions from natural numbers

lemma *of-nat-poly*: *of-nat n* = [:*of-nat n* :: *'a* :: *comm-semiring-1*:]
⟨*proof*⟩

lemma *degree-of-nat* [*simp*]: *degree* (*of-nat n*) = *0*
⟨*proof*⟩

lemma *degree-numeral* [*simp*]: *degree* (*numeral n*) = *0*
⟨*proof*⟩

lemma *numeral-poly*: *numeral n* = [:*numeral n*:]
⟨*proof*⟩

41.15 Lemmas about divisibility

lemma *dvd-smult*: *p dvd q* ⟹ *p dvd smult a q*
⟨*proof*⟩

lemma *dvd-smult-cancel*:
fixes *a* :: *'a* :: *field*
shows *p dvd smult a q* ⟹ *a* ≠ *0* ⟹ *p dvd q*
⟨*proof*⟩

lemma *dvd-smult-iff*:
fixes *a* :: *'a*::*field*
shows *a* ≠ *0* ⟹ *p dvd smult a q* ⟷ *p dvd q*
⟨*proof*⟩

lemma *smult-dvd-cancel*:
smult a p dvd q ⟹ *p dvd q*

⟨*proof*⟩

lemma *smult-dvd*:
fixes *a* :: *'a*::*field*
shows *p dvd q* ⟹ *a* ≠ *0* ⟹ *smult a p dvd q*
⟨*proof*⟩

lemma *smult-dvd-iff*:
fixes *a* :: *'a*::*field*
shows *smult a p dvd q* ⟷ (*if a* = *0 then q* = *0 else p dvd q*)
⟨*proof*⟩

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*)
⟨*proof*⟩

instance *poly* :: (*idom*) *idom*
⟨*proof*⟩

lemma *degree-mult-eq*:
fixes *p q* :: *'a*::*semidom poly*
shows ⟦*p* ≠ *0*; *q* ≠ *0*⟧ ⟹ *degree* (*p* ∗ *q*) = *degree p* + *degree q*
⟨*proof*⟩

lemma *degree-mult-right-le*:
fixes *p q* :: *'a*::*semidom poly*
assumes *q* ≠ *0*
shows *degree p* ≤ *degree* (*p* ∗ *q*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *dvd-imp-degree-le*:
fixes *p q* :: *'a*::*semidom poly*
shows ⟦*p dvd q*; *q* ≠ *0*⟧ ⟹ *degree p* ≤ *degree q*
⟨*proof*⟩

lemma *divides-degree*:
assumes *pq*: *p dvd* (*q* :: *'a* :: *semidom poly*)
shows *degree p* ≤ *degree q* ∨ *q* = *0*
⟨*proof*⟩

41.17 Polynomials form an ordered integral domain

definition *pos-poly* :: *'a::linordered-idom poly* ⇒ *bool*
where
pos-poly p ⟷ *0 < coeff p (degree p)*

lemma *pos-poly-pCons*:
pos-poly (pCons a p) ⟷ *pos-poly p* ∨ *(p = 0* ∧ *0 < a)*
⟨*proof*⟩

lemma *not-pos-poly-0* [*simp*]: ¬ *pos-poly 0*
⟨*proof*⟩

lemma *pos-poly-add*: ⟦*pos-poly p*; *pos-poly q*⟧ ⟹ *pos-poly (p + q)*
⟨*proof*⟩

lemma *pos-poly-mult*: ⟦*pos-poly p*; *pos-poly q*⟧ ⟹ *pos-poly (p * q)*
⟨*proof*⟩

lemma *pos-poly-total*: *p = 0* ∨ *pos-poly p* ∨ *pos-poly (− p)*
⟨*proof*⟩

lemma *last-coeffs-eq-coeff-degree*:
p ≠ 0 ⟹ *last (coeffs p) = coeff p (degree p)*
⟨*proof*⟩

lemma *pos-poly-coeffs* [*code*]:
pos-poly p ⟷ *(let as = coeffs p in as ≠ []* ∧ *last as > 0)* (**is** *?P* ⟷ *?Q*)
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *synthetic-divmod-pCons* [*simp*]:
synthetic-divmod (*pCons a p*) *c* = (λ(*q*, *r*). (*pCons r q*, *a* + *c* ∗ *r*)) (*synthetic-divmod p c*)
⟨*proof*⟩

lemma *synthetic-div-0* [*simp*]:
synthetic-div 0 c = *0*
⟨*proof*⟩

lemma *synthetic-div-unique-lemma*: *smult c p* = *pCons a p* ⟹ *p* = *0*
⟨*proof*⟩

lemma *snd-synthetic-divmod*:
snd (*synthetic-divmod p c*) = *poly p c*
⟨*proof*⟩

lemma *synthetic-div-pCons* [*simp*]:
synthetic-div (*pCons a p*) *c* = *pCons* (*poly p c*) (*synthetic-div p c*)
⟨*proof*⟩

lemma *synthetic-div-eq-0-iff*:
synthetic-div p c = *0* ⟷ *degree p* = *0*
⟨*proof*⟩

lemma *degree-synthetic-div*:
degree (*synthetic-div p c*) = *degree p* − *1*
⟨*proof*⟩

lemma *synthetic-div-correct*:
p + *smult c* (*synthetic-div p c*) = *pCons* (*poly p c*) (*synthetic-div p c*)
⟨*proof*⟩

lemma *synthetic-div-unique*:
$p + smult\ c\ q = pCons\ r\ q \Longrightarrow r = poly\ p\ c \wedge q = synthetic\text{-}div\ p\ c$
⟨*proof*⟩

lemma *synthetic-div-correct'*:
fixes $c :: 'a::comm\text{-}ring\text{-}1$
shows $[:-c,\ 1:] * synthetic\text{-}div\ p\ c + [:poly\ p\ c:] = p$
⟨*proof*⟩

lemma *poly-eq-0-iff-dvd*:
fixes $c :: 'a::idom$
shows $poly\ p\ c = 0 \longleftrightarrow [:-c,\ 1:]\ dvd\ p$
⟨*proof*⟩

lemma *dvd-iff-poly-eq-0*:
fixes $c :: 'a::idom$
shows $[:c,\ 1:]\ dvd\ p \longleftrightarrow poly\ p\ (-c) = 0$
⟨*proof*⟩

lemma *poly-roots-finite*:
fixes $p :: 'a::idom\ poly$
shows $p \neq 0 \Longrightarrow finite\ \{x.\ poly\ p\ x = 0\}$
⟨*proof*⟩

lemma *poly-eq-poly-eq-iff*:
fixes $p\ q :: 'a::\{idom,ring\text{-}char\text{-}0\}\ poly$
shows $poly\ p = poly\ q \longleftrightarrow p = q$ (**is** $?P \longleftrightarrow ?Q$)
⟨*proof*⟩

lemma *poly-all-0-iff-0*:
fixes $p :: 'a::\{ring\text{-}char\text{-}0,\ idom\}\ poly$
shows $(\forall x.\ poly\ p\ x = 0) \longleftrightarrow p = 0$
⟨*proof*⟩

41.19 Long division of polynomials

definition *pdivmod-rel* :: $'a::field\ poly \Rightarrow 'a\ poly \Rightarrow 'a\ poly \Rightarrow 'a\ poly \Rightarrow bool$
where
pdivmod-rel x y q r ⟷
$x = q * y + r \wedge (if\ y = 0\ then\ q = 0\ else\ r = 0 \vee degree\ r < degree\ y)$

lemma *pdivmod-rel-0*:
pdivmod-rel 0 y 0 0
⟨*proof*⟩

lemma *pdivmod-rel-by-0*:
pdivmod-rel x 0 0 x
⟨*proof*⟩

lemma *eq-zero-or-degree-less*:
  assumes *degree p* $\leq$ *n* **and** *coeff p n* = *0*
  shows *p* = *0* $\vee$ *degree p* < *n*
⟨*proof*⟩

lemma *pdivmod-rel-pCons*:
  assumes *rel*: *pdivmod-rel x y q r*
  assumes *y*: *y* $\neq$ *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*⟩

lemma *pdivmod-rel-exists*: ∃ *q r*. *pdivmod-rel x y q r*
⟨*proof*⟩

lemma *pdivmod-rel-unique*:
  assumes *1*: *pdivmod-rel x y q1 r1*
  assumes *2*: *pdivmod-rel x y q2 r2*
  shows *q1* = *q2* $\wedge$ *r1* = *r2*
⟨*proof*⟩

lemma *pdivmod-rel-0-iff*: *pdivmod-rel 0 y q r* $\longleftrightarrow$ *q* = *0* $\wedge$ *r* = *0*
⟨*proof*⟩

lemma *pdivmod-rel-by-0-iff*: *pdivmod-rel x 0 q r* $\longleftrightarrow$ *q* = *0* $\wedge$ *r* = *x*
⟨*proof*⟩

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 $\Longrightarrow$ *x div y* = *q*
⟨*proof*⟩

lemma *mod-poly-eq*:
  pdivmod-rel x y q r $\Longrightarrow$ *x mod y* = *r*
⟨*proof*⟩

lemma *pdivmod-rel*:
pdivmod-rel x y (*x div y*) (*x mod y*)
⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *is-unit-monom-0*:
fixes *a* :: *'a*::*field*
assumes *a* ≠ *0*
shows *is-unit* (*monom a 0*)
⟨*proof*⟩

lemma *is-unit-triv*:
fixes *a* :: *'a*::*field*
assumes *a* ≠ *0*
shows *is-unit* [:*a*:]
⟨*proof*⟩

lemma *is-unit-iff-degree*:
assumes *p* ≠ *0*
shows *is-unit p* ⟷ *degree p* = *0* (**is** *?P* ⟷ *?Q*)
⟨*proof*⟩

lemma *is-unit-pCons-iff*:
is-unit (*pCons a p*) ⟷ *p* = *0* ∧ *a* ≠ *0* (**is** *?P* ⟷ *?Q*)
⟨*proof*⟩

lemma *is-unit-monom-trival*:
fixes *p* :: *'a*::*field poly*
assumes *is-unit p*
shows *monom* (*coeff p* (*degree p*)) *0* = *p*
⟨*proof*⟩

lemma *is-unit-polyE*:
assumes *is-unit p*
obtains *a* **where** *p* = *monom a 0* **and** *a* ≠ *0*
⟨*proof*⟩

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*⟩

end

lemma *unit-factor-monom* [*simp*]:
unit-factor (*monom a n*) =
(*if a* = *0 then 0 else monom a 0*)
⟨*proof*⟩

lemma *unit-factor-pCons* [*simp*]:
unit-factor (*pCons a p*) =
(*if p* = *0 then monom a 0 else unit-factor p*)
⟨*proof*⟩

lemma *normalize-monom* [*simp*]:
normalize (*monom a n*) =
(*if a* = *0 then 0 else monom 1 n*)
⟨*proof*⟩

lemma *degree-mod-less*:
y ≠ *0* ⟹ *x mod y* = *0* ∨ *degree* (*x mod y*) < *degree y*
⟨*proof*⟩

lemma *div-poly-less*: *degree x* < *degree y* ⟹ *x div y* = *0*
⟨*proof*⟩

lemma *mod-poly-less*: *degree x* < *degree y* ⟹ *x mod y* = *x*
⟨*proof*⟩

lemma *pdivmod-rel-smult-left*:
pdivmod-rel x y q r
⟹ *pdivmod-rel* (*smult a x*) *y* (*smult a q*) (*smult a r*)
⟨*proof*⟩

lemma *div-smult-left*: (*smult a x*) *div y* = *smult a* (*x div y*)
⟨*proof*⟩

lemma *mod-smult-left*: (*smult a x*) *mod y* = *smult a* (*x mod y*)
⟨*proof*⟩

lemma *poly-div-minus-left* [*simp*]:
fixes *x y* :: *'a*::*field poly*
shows (− *x*) *div y* = − (*x div y*)
⟨*proof*⟩

lemma *poly-mod-minus-left* [*simp*]:
fixes *x y* :: *'a*::*field poly*

shows $(-\ x)\ mod\ y = -\ (x\ mod\ y)$
⟨*proof*⟩

lemma *pdivmod-rel-add-left*:
assumes *pdivmod-rel x y q r*
assumes *pdivmod-rel x′ y q′ r′*
shows *pdivmod-rel* $(x + x')\ y\ (q + q')\ (r + r')$
⟨*proof*⟩

lemma *poly-div-add-left*:
fixes $x\ y\ z :: 'a$::*field poly*
shows $(x + y)\ div\ z = x\ div\ z + y\ div\ z$
⟨*proof*⟩

lemma *poly-mod-add-left*:
fixes $x\ y\ z :: 'a$::*field poly*
shows $(x + y)\ mod\ z = x\ mod\ z + y\ mod\ z$
⟨*proof*⟩

lemma *poly-div-diff-left*:
fixes $x\ y\ z :: 'a$::*field poly*
shows $(x - y)\ div\ z = x\ div\ z - y\ div\ z$
⟨*proof*⟩

lemma *poly-mod-diff-left*:
fixes $x\ y\ z :: 'a$::*field poly*
shows $(x - y)\ mod\ z = x\ mod\ z - y\ mod\ z$
⟨*proof*⟩

lemma *pdivmod-rel-smult-right*:
$[\![a \neq 0;\ pdivmod\text{-}rel\ x\ y\ q\ r]\!]$
$\Longrightarrow pdivmod\text{-}rel\ x\ (smult\ a\ y)\ (smult\ (inverse\ a)\ q)\ r$
⟨*proof*⟩

lemma *div-smult-right*:
$a \neq 0 \Longrightarrow x\ div\ (smult\ a\ y) = smult\ (inverse\ a)\ (x\ div\ y)$
⟨*proof*⟩

lemma *mod-smult-right*: $a \neq 0 \Longrightarrow x\ mod\ (smult\ a\ y) = x\ mod\ y$
⟨*proof*⟩

lemma *poly-div-minus-right* [*simp*]:
fixes $x\ y :: 'a$::*field poly*
shows $x\ div\ (-\ y) = -\ (x\ div\ y)$
⟨*proof*⟩

lemma *poly-mod-minus-right* [*simp*]:
fixes $x\ y :: 'a$::*field poly*
shows $x\ mod\ (-\ y) = x\ mod\ y$

⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *poly-div-mult-right*:
fixes *x y z* :: *'a*::*field poly*
shows *x div* (*y* ∗ *z*) = (*x div y*) *div z*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *mod-poly-code* [*code*]:
p mod q = *snd* (*pdivmod p q*)
⟨*proof*⟩

lemma *pdivmod-0*:
pdivmod 0 q = (*0*, *0*)
⟨*proof*⟩

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*)))
⟨*proof*⟩

lemma *pdivmod-fold-coeffs* [*code*]:

pdivmod p q = (if q = 0 then (0, p)
else fold-coeffs (λa (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))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *degree-linear-power*:
fixes *a* :: *'a::comm-semiring-1*
shows *degree ([:a, 1:] ^ n) = n*
⟨*proof*⟩

lemma *order-1*: *[:−a, 1:] ^ order a p dvd p*
⟨*proof*⟩

lemma *order-2*: *p ≠ 0 ⟹ ¬ [:−a, 1:] ^ Suc (order a p) dvd p*
⟨*proof*⟩

lemma *order*:
p ≠ 0 ⟹ [:−a, 1:] ^ order a p dvd p ∧ ¬ [:−a, 1:] ^ Suc (order a p) dvd p
⟨*proof*⟩

lemma *order-degree*:
assumes *p*: *p ≠ 0*
shows *order a p ≤ degree p*
⟨*proof*⟩

lemma *order-root*: *poly p a = 0 ⟷ p = 0 ∨ order a p ≠ 0*
⟨*proof*⟩

lemma *order-0I*: *poly p a ≠ 0 ⟹ order a p = 0*
⟨*proof*⟩

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*⟩

lemma *dropWhile-replicate-append*:
$dropWhile\ (op=\ a)\ (replicate\ n\ a\ @\ ys) = dropWhile\ (op=\ a)\ ys$
⟨*proof*⟩

lemma *Poly-append-replicate-0*: $Poly\ (xs\ @\ replicate\ n\ 0) = Poly\ xs$
⟨*proof*⟩

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*⟩

41.22 Composition of polynomials

definition $pcompose :: 'a::comm\text{-}semiring\text{-}0\ poly \Rightarrow 'a\ poly \Rightarrow 'a\ poly$
where
$pcompose\ p\ q = fold\text{-}coeffs\ (\lambda a\ c.\ [:a:] + q * c)\ p\ 0$

notation *pcompose* (**infixl** $\circ_p$ *71*)

lemma *pcompose-0* [*simp*]:
$pcompose\ 0\ q = 0$
⟨*proof*⟩

lemma *pcompose-pCons*:
$pcompose\ (pCons\ a\ p)\ q = [:a:] + q * pcompose\ p\ q$
⟨*proof*⟩

lemma *pcompose-1*:
fixes $p :: 'a :: comm\text{-}semiring\text{-}1\ poly$
shows $pcompose\ 1\ p = 1$
⟨*proof*⟩

lemma *poly-pcompose*:
$poly\ (pcompose\ p\ q)\ x = poly\ p\ (poly\ q\ x)$
⟨*proof*⟩

lemma *degree-pcompose-le*:
$degree\ (pcompose\ p\ q) \leq degree\ p * degree\ q$
⟨*proof*⟩

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*⟩

lemma *pcompose-uminus*:
fixes *p r* :: *'a* :: *comm-ring poly*
shows *pcompose* (−*p*) *r* = −*pcompose p r*
⟨*proof*⟩

lemma *pcompose-diff*:
fixes *p q r* :: *'a* :: *comm-ring poly*
shows *pcompose* (*p* − *q*) *r* = *pcompose p r* − *pcompose q r*
⟨*proof*⟩

lemma *pcompose-smult*:
fixes *p r* :: *'a* :: *comm-semiring-0 poly*
shows *pcompose* (*smult a p*) *r* = *smult a* (*pcompose p r*)
⟨*proof*⟩

lemma *pcompose-mult*:
fixes *p q r* :: *'a* :: *comm-semiring-0 poly*
shows *pcompose* (*p* ∗ *q*) *r* = *pcompose p r* ∗ *pcompose q r*
⟨*proof*⟩

lemma *pcompose-assoc*:
pcompose p (*pcompose q r* :: *'a* :: *comm-semiring-0 poly*) =
pcompose (*pcompose p q*) *r*
⟨*proof*⟩

lemma *pcompose-idR*[*simp*]:
fixes *p* :: *'a* :: *comm-semiring-1 poly*
shows *pcompose p* [: *0*, *1* :] = *p*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *pcompose-const*[*simp*]:*pcompose* [:*a*:] *q* = [:*a*:] ⟨*proof*⟩

lemma *pcompose-0'*: *pcompose p 0* = [:*coeff p 0*:]
⟨*proof*⟩

lemma *degree-pcompose*:
fixes *p q*:: *'a*::*semidom poly*
shows *degree* (*pcompose p q*) = *degree p* ∗ *degree q*
⟨*proof*⟩

lemma *pcompose-eq-0*:
fixes *p q*:: *'a* :: *semidom poly*
assumes *pcompose p q* = *0 degree q* > *0*
shows *p* = *0*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *lead-coeff-0*[*simp*]:*lead-coeff 0* =*0*
⟨*proof*⟩

lemma *lead-coeff-mult*:
fixes *p q*::*'a* ::*idom poly*
shows *lead-coeff* (*p* ∗ *q*) = *lead-coeff p* ∗ *lead-coeff q*
⟨*proof*⟩

lemma *lead-coeff-add-le*:
assumes *degree p* < *degree q*
shows *lead-coeff* (*p*+*q*) = *lead-coeff q*
⟨*proof*⟩

lemma *lead-coeff-minus*:
lead-coeff (−*p*) = − *lead-coeff p*
⟨*proof*⟩

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* ^ (*degree p*)
⟨*proof*⟩

lemma *lead-coeff-smult*:
lead-coeff (*smult c p* :: *'a* :: *idom poly*) = *c* ∗ *lead-coeff p*
⟨*proof*⟩

lemma *lead-coeff-1* [*simp*]: *lead-coeff 1 = 1*
⟨*proof*⟩

lemma *lead-coeff-of-nat* [*simp*]:
lead-coeff (*of-nat n*) = (*of-nat n* :: *'a* :: {*comm-semiring-1*,*semiring-char-0*})
⟨*proof*⟩

lemma *lead-coeff-numeral* [*simp*]:
lead-coeff (*numeral n*) = *numeral n*
⟨*proof*⟩

lemma *lead-coeff-power*:
lead-coeff (*p ^ n* :: *'a* :: *idom poly*) = *lead-coeff p ^ n*
⟨*proof*⟩

lemma *lead-coeff-nonzero*: *p ≠ 0 ⟹ lead-coeff p ≠ 0*
⟨*proof*⟩

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*))
⟨*proof*⟩

termination *pderiv*
⟨*proof*⟩

lemma *pderiv-0* [*simp*]:
pderiv 0 = 0
⟨*proof*⟩

lemma *pderiv-pCons*:
pderiv (*pCons a p*) = *p + pCons 0* (*pderiv p*)
⟨*proof*⟩

lemma *pderiv-1* [*simp*]: *pderiv 1 = 0*
⟨*proof*⟩

lemma *pderiv-of-nat* [*simp*]: *pderiv* (*of-nat n*) = *0*
and *pderiv-numeral* [*simp*]: *pderiv* (*numeral m*) = *0*
⟨*proof*⟩

lemma *coeff-pderiv*: *coeff* (*pderiv p*) *n* = *of-nat* (*Suc n*) ∗ *coeff p* (*Suc n*)
⟨*proof*⟩

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*⟩

lemma *map-upt-Suc*: *map f* [*0* ..< *Suc n*] = *f 0* # *map* (λ *i*. *f* (*Suc i*)) [*0* ..< *n*]
⟨*proof*⟩

lemma *coeffs-pderiv-code* [*code abstract*]:
coeffs (*pderiv p*) = *pderiv-coeffs* (*coeffs p*) ⟨*proof*⟩

context
assumes *SORT-CONSTRAINT*('*a*::{*semidom*, *semiring-char-0*})
begin

lemma *pderiv-eq-0-iff*:
pderiv (*p* :: '*a poly*) = *0* ⟷ *degree p* = *0*
⟨*proof*⟩

lemma *degree-pderiv*: *degree* (*pderiv* (*p* :: '*a poly*)) = *degree p* − *1*
⟨*proof*⟩

lemma *not-dvd-pderiv*:
assumes *degree* (*p* :: '*a poly*) ≠ *0*
shows ¬ *p dvd pderiv p*
⟨*proof*⟩

lemma *dvd-pderiv-iff* [*simp*]: (*p* :: '*a poly*) *dvd pderiv p* ⟷ *degree p* = *0*
⟨*proof*⟩

end

lemma *pderiv-singleton* [*simp*]: *pderiv* [:*a*:] = *0*
⟨*proof*⟩

lemma *pderiv-add*: *pderiv* (*p* + *q*) = *pderiv p* + *pderiv q*
⟨*proof*⟩

lemma *pderiv-minus*: *pderiv* (− *p* :: '*a* :: *idom poly*) = − *pderiv p*
⟨*proof*⟩

lemma *pderiv-diff*: *pderiv* (*p* − *q*) = *pderiv p* − *pderiv q*
⟨*proof*⟩

lemma *pderiv-smult*: *pderiv* (*smult a p*) = *smult a* (*pderiv p*)
⟨*proof*⟩

lemma *pderiv-mult*: *pderiv* (*p* ∗ *q*) = *p* ∗ *pderiv q* + *q* ∗ *pderiv p*
⟨*proof*⟩

lemma *pderiv-power-Suc*:
pderiv (*p* ^ *Suc n*) = *smult* (*of-nat* (*Suc n*)) (*p* ^ *n*) ∗ *pderiv p*
⟨*proof*⟩

lemma *pderiv-setprod*: *pderiv* (*setprod f* (*as*)) =
(∑ *a* ∈ *as*. *setprod f* (*as* − {*a*}) ∗ *pderiv* (*f a*))
⟨*proof*⟩

lemma *DERIV-pow2*: *DERIV* (%*x*. *x* ^ *Suc n*) *x* :> *real* (*Suc n*) ∗ (*x* ^ *n*)
⟨*proof*⟩
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*
⟨*proof*⟩

lemma *poly-DERIV* [*simp*]: *DERIV* (%*x*. *poly p x*) *x* :> *poly* (*pderiv p*) *x*
⟨*proof*⟩

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*⟩

Consequences of the derivative theorem above

lemma *poly-differentiable*[*simp*]: (%*x*. *poly p x*) *differentiable* (*at x*::*real filter*)
⟨*proof*⟩

lemma *poly-isCont*[*simp*]: *isCont* (%*x*. *poly p x*) (*x*::*real*)
⟨*proof*⟩

lemma *poly-IVT-pos*: [| *a* < *b*; *poly p* (*a*::*real*) < *0*; *0* < *poly p b* |]
==> ∃ *x*. *a* < *x* & *x* < *b* & (*poly p x* = *0*)
⟨*proof*⟩

lemma *poly-IVT-neg*: [| (*a*::*real*) < *b*; *0* < *poly p a*; *poly p b* < *0* |]
==> ∃ *x*. *a* < *x* & *x* < *b* & (*poly p x* = *0*)
⟨*proof*⟩

lemma *poly-IVT*:
fixes *p*::*real poly*
assumes *a*<*b* **and** *poly p a* ∗ *poly p b* < *0*

shows $\exists x{>}a.\ x < b \wedge poly\ p\ x = 0$
⟨*proof*⟩

lemma *poly-MVT*: $(a{::}real) < b \Longrightarrow$
$\exists x.\ a < x\ \&\ x < b\ \&\ (poly\ p\ b - poly\ p\ a = (b - a) * poly\ (pderiv\ p)\ x)$
⟨*proof*⟩

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*⟩

lemma *poly-pinfty-gt-lc*:
fixes p:: *real poly*
assumes *lead-coeff p* > 0
shows $\exists\ n.\ \forall\ x \geq n.\ poly\ p\ x \geq lead\text{-}coeff\ p$ ⟨*proof*⟩

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* $\Rightarrow$ *bool* **where**
$algebraic\ x \longleftrightarrow (\exists p.\ (\forall i.\ coeff\ p\ i \in \mathbb{Z}) \wedge p \neq 0 \wedge poly\ p\ x = 0)$

lemma *algebraicI*:
assumes $\bigwedge i.\ coeff\ p\ i \in \mathbb{Z}\ p \neq 0\ poly\ p\ x = 0$
shows *algebraic x*
⟨*proof*⟩

lemma *algebraicE*:
assumes *algebraic x*
obtains p **where** $\bigwedge i.\ coeff\ p\ i \in \mathbb{Z}\ p \neq 0\ poly\ p\ x = 0$
⟨*proof*⟩

lemma *quotient-of-denom-pos'*: $snd\ (quotient\text{-}of\ x) > 0$
⟨*proof*⟩

lemma *of-int-div-in-Ints*:
$b\ dvd\ a \Longrightarrow of\text{-}int\ a\ div\ of\text{-}int\ b \in (\mathbb{Z} :: 'a :: ring\text{-}div\ set)$
⟨*proof*⟩

lemma *of-int-divide-in-Ints*:
b dvd a $\Longrightarrow$ *of-int a / of-int b* $\in$ ($\mathbb{Z}$ :: *'a* :: *field set*)
⟨*proof*⟩

lemma *algebraic-altdef*:
fixes *p* :: *'a* :: *field-char-0 poly*
shows *algebraic x* $\longleftrightarrow$ ($\exists$ *p*. ($\forall$ *i*. *coeff p i* $\in$ $\mathbb{Q}$) $\wedge$ *p* $\neq$ *0* $\wedge$ *poly p x* = *0*)
⟨*proof*⟩

Lemmas for Derivatives

lemma *order-unique-lemma*:
fixes *p* :: *'a*::*idom poly*
assumes [:−*a*, *1*:] ^ *n dvd p* ¬ [:−*a*, *1*:] ^ *Suc n dvd p*
shows *n* = *order a p*
⟨*proof*⟩

lemma *lemma-order-pderiv1*:
pderiv ([:− *a*, *1*:] ^ *Suc n* * *q*) = [:− *a*, *1*:] ^ *Suc n* * *pderiv q* +
smult (*of-nat* (*Suc n*)) (*q* * [:− *a*, *1*:] ^ *n*)
⟨*proof*⟩

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*:] ^ *n* * *q*
and *nd*: ~ [:− *a*, *1*:] *dvd q*
shows *n* = *Suc* (*order a* (*pderiv p*))
⟨*proof*⟩

lemma *order-decomp*:
assumes *p* $\neq$ *0*
shows $\exists$ *q*. *p* = [:− *a*, *1*:] ^ *order a p* * *q* $\wedge$ ¬ [:− *a*, *1*:] *dvd q*
⟨*proof*⟩

lemma *order-pderiv*:
⟦*pderiv p* $\neq$ *0*; *order a* (*p* :: *'a* :: *field-char-0 poly*) $\neq$ *0*⟧ $\Longrightarrow$
(*order a p* = *Suc* (*order a* (*pderiv p*)))
⟨*proof*⟩

lemma *order-mult*: *p* * *q* $\neq$ *0* $\Longrightarrow$ *order a* (*p* * *q*) = *order a p* + *order a q*
⟨*proof*⟩

lemma *order-smult*:
assumes *c* $\neq$ *0*
shows *order x* (*smult c p*) = *order x p*
⟨*proof*⟩

lemma *order-1-eq-0* [*simp*]:*order x 1 = 0*
⟨*proof*⟩

lemma *order-power-n-n*: *order a* ([:−*a*,*1*:]^*n*)=*n*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*}:]
⟨*proof*⟩

lemma *rsquarefree-roots*:
fixes *p* :: *'a* :: *field-char-0 poly*
shows *rsquarefree p* = (∀ *a*. ¬(*poly p a = 0* ∧ *poly* (*pderiv p*) *a = 0*))
⟨*proof*⟩

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*⟩

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* ≠ (*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*⟩

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*⟩

function *gcd-eucl* :: '*a* ⇒ '*a* ⇒ '*a*
where
gcd-eucl a b = (*if b* = *0 then normalize a else gcd-eucl b* (*a mod b*))
⟨*proof*⟩

termination
⟨*proof*⟩

declare *gcd-eucl.simps* [*simp del*]

lemma *gcd-eucl-induct* [*case-names zero mod*]:
assumes *H1*: $\bigwedge b.\ P\ b\ 0$
and *H2*: $\bigwedge a\ b.\ b \neq 0 \Longrightarrow P\ b\ (a\ mod\ b) \Longrightarrow P\ a\ b$
shows $P\ a\ b$
⟨*proof*⟩

definition *lcm-eucl* :: $'a \Rightarrow 'a \Rightarrow 'a$
where
$lcm\text{-}eucl\ a\ b = normalize\ (a * b)\ div\ gcd\text{-}eucl\ a\ b$

definition *Lcm-eucl* :: $'a\ set \Rightarrow 'a$ — Somewhat complicated definition of Lcm that has the advantage of working for infinite sets as well
where
$Lcm\text{-}eucl\ A = (if\ \exists l.\ l \neq 0 \wedge (\forall a{\in}A.\ a\ dvd\ l)\ then$
$\quad let\ l = SOME\ l.\ l \neq 0 \wedge (\forall a{\in}A.\ a\ dvd\ l) \wedge euclidean\text{-}size\ l =$
$\quad\quad (LEAST\ n.\ \exists l.\ l \neq 0 \wedge (\forall a{\in}A.\ a\ dvd\ l) \wedge euclidean\text{-}size\ l = n)$
$\quad\quad in\ normalize\ l$
$\quad else\ 0)$

definition *Gcd-eucl* :: $'a\ set \Rightarrow 'a$
where
$Gcd\text{-}eucl\ A = Lcm\text{-}eucl\ \{d.\ \forall a{\in}A.\ d\ dvd\ a\}$

declare *Lcm-eucl-def Gcd-eucl-def* [*code del*]

lemma *gcd-eucl-0*:
$gcd\text{-}eucl\ a\ 0 = normalize\ a$
⟨*proof*⟩

lemma *gcd-eucl-0-left*:
$gcd\text{-}eucl\ 0\ a = normalize\ a$
⟨*proof*⟩

lemma *gcd-eucl-non-0*:
$b \neq 0 \Longrightarrow gcd\text{-}eucl\ a\ b = gcd\text{-}eucl\ b\ (a\ mod\ b)$
⟨*proof*⟩

lemma *gcd-eucl-dvd1* [*iff*]: $gcd\text{-}eucl\ a\ b\ dvd\ a$
and *gcd-eucl-dvd2* [*iff*]: $gcd\text{-}eucl\ a\ b\ dvd\ b$
⟨*proof*⟩

lemma *normalize-gcd-eucl* [*simp*]:
$normalize\ (gcd\text{-}eucl\ a\ b) = gcd\text{-}eucl\ a\ b$
⟨*proof*⟩

lemma *gcd-eucl-greatest*:
fixes *k a b* :: *'a*
shows *k dvd a* $\Longrightarrow$ *k dvd b* $\Longrightarrow$ *k dvd gcd-eucl a b*
⟨*proof*⟩

lemma *eq-gcd-euclI*:
fixes *gcd* :: *'a* $\Rightarrow$ *'a* $\Rightarrow$ *'a*
assumes $\bigwedge$*a b*. *gcd a b dvd a* $\bigwedge$*a b*. *gcd a b dvd b* $\bigwedge$*a b*. *normalize* (*gcd a b*) = *gcd a b*
$\bigwedge$*a b k*. *k dvd a* $\Longrightarrow$ *k dvd b* $\Longrightarrow$ *k dvd gcd a b*
shows *gcd* = *gcd-eucl*
⟨*proof*⟩

lemma *gcd-eucl-zero* [*simp*]:
gcd-eucl a b = *0* $\longleftrightarrow$ *a* = *0* $\wedge$ *b* = *0*
⟨*proof*⟩

lemma *dvd-Lcm-eucl* [*simp*]: *a* $\in$ *A* $\Longrightarrow$ *a dvd Lcm-eucl A*
and *Lcm-eucl-least*: ($\bigwedge$*a*. *a* $\in$ *A* $\Longrightarrow$ *a dvd b*) $\Longrightarrow$ *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*⟩

lemma *normalize-Lcm-eucl* [*simp*]:
normalize (*Lcm-eucl A*) = *Lcm-eucl A*
⟨*proof*⟩

lemma *eq-Lcm-euclI*:
fixes *lcm* :: *'a set* $\Rightarrow$ *'a*
assumes $\bigwedge$*A a*. *a* $\in$ *A* $\Longrightarrow$ *a dvd lcm A* **and** $\bigwedge$*A c*. ($\bigwedge$*a*. *a* $\in$ *A* $\Longrightarrow$ *a dvd c*) $\Longrightarrow$ *lcm A dvd c*
$\bigwedge$*A*. *normalize* (*lcm A*) = *lcm A* **shows** *lcm* = *Lcm-eucl*
⟨*proof*⟩

end

class *euclidean-ring* = *euclidean-semiring* + *idom*
begin

subclass *ring-div* ⟨*proof*⟩

function *euclid-ext-aux* :: *'a* $\Rightarrow$ - **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*))
⟨*proof*⟩

termination ⟨*proof*⟩

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*))
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *euclid-ext-left-0*:
euclid-ext 0 a = (0, 1 div unit-factor a, normalize a)
⟨*proof*⟩

lemma *euclid-ext-correct'*:
*case euclid-ext x y of (a,b,c) ⇒ a * x + b * y = c ∧ c = gcd-eucl x y*
⟨*proof*⟩

lemma *euclid-ext-gcd-eucl*:
(case euclid-ext x y of (a,b,c) ⇒ c) = gcd-eucl x y
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *euclid-ext'-0*: *euclid-ext' a 0 = (1 div unit-factor a, 0)*
⟨*proof*⟩

lemma *euclid-ext'-left-0*: *euclid-ext' 0 a = (0, 1 div unit-factor a)*
⟨*proof*⟩

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*
⟨*proof*⟩

subclass *semiring-Gcd*
⟨*proof*⟩

lemma *gcd-non-0*:
$b \neq 0 \Longrightarrow gcd\ a\ b = gcd\ b\ (a\ mod\ b)$
⟨*proof*⟩

lemmas *gcd-0* = *gcd-0-right*
lemmas *dvd-gcd-iff* = *gcd-greatest-iff*
lemmas *gcd-greatest-iff* = *dvd-gcd-iff*

lemma *gcd-mod1* [*simp*]:
$gcd\ (a\ mod\ b)\ b = gcd\ a\ b$
⟨*proof*⟩

lemma *gcd-mod2* [*simp*]:
$gcd\ a\ (b\ mod\ a) = gcd\ a\ b$
⟨*proof*⟩

lemma *euclidean-size-gcd-le1* [*simp*]:
assumes $a \neq 0$
shows $euclidean\text{-}size\ (gcd\ a\ b) \leq euclidean\text{-}size\ a$
⟨*proof*⟩

lemma *euclidean-size-gcd-le2* [*simp*]:
$b \neq 0 \Longrightarrow euclidean\text{-}size\ (gcd\ a\ b) \leq euclidean\text{-}size\ b$
⟨*proof*⟩

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*⟩

lemma *euclidean-size-gcd-less2*:
assumes $b \neq 0$ **and** $\neg b\ dvd\ a$
shows $euclidean\text{-}size\ (gcd\ a\ b) < euclidean\text{-}size\ b$
⟨*proof*⟩

lemma *euclidean-size-lcm-le1*:
assumes $a \neq 0$ **and** $b \neq 0$
shows $euclidean\text{-}size\ a \leq euclidean\text{-}size\ (lcm\ a\ b)$
⟨*proof*⟩

lemma *euclidean-size-lcm-le2*:
a ≠ *0* ⟹ *b* ≠ *0* ⟹ *euclidean-size b* ≤ *euclidean-size* (*lcm a b*)
⟨*proof*⟩

lemma *euclidean-size-lcm-less1*:
assumes *b* ≠ *0* **and** ¬*b dvd a*
shows *euclidean-size a* < *euclidean-size* (*lcm a b*)
⟨*proof*⟩

lemma *euclidean-size-lcm-less2*:
assumes *a* ≠ *0* **and** ¬*a dvd b*
shows *euclidean-size b* < *euclidean-size* (*lcm a b*)
⟨*proof*⟩

lemma *Lcm-eucl-set* [*code*]:
Lcm-eucl (*set xs*) = *foldl lcm-eucl 1 xs*
⟨*proof*⟩

lemma *Gcd-eucl-set* [*code*]:
Gcd-eucl (*set xs*) = *foldl gcd-eucl 0 xs*
⟨*proof*⟩

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* ⟨*proof*⟩
subclass *ring-gcd* ⟨*proof*⟩

lemma *euclid-ext-gcd* [*simp*]:
(*case euclid-ext a b of* (-, - , *t*) ⇒ *t*) = *gcd a b*
⟨*proof*⟩

lemma *euclid-ext-gcd'* [*simp*]:
euclid-ext a b = (*r*, *s*, *t*) ⟹ *t* = *gcd a b*
⟨*proof*⟩

lemma *euclid-ext-correct*:
case euclid-ext x y of (*a*,*b*,*c*) ⇒ *a* ∗ *x* + *b* ∗ *y* = *c* ∧ *c* = *gcd x y*
⟨*proof*⟩

lemma *euclid-ext'-correct*:
fst (*euclid-ext' a b*) ∗ *a* + *snd* (*euclid-ext' a b*) ∗ *b* = *gcd a b*
⟨*proof*⟩

lemma *bezout*: $\exists\, s\ t.\ s * a + t * b = gcd\ a\ b$
⟨*proof*⟩

end

42.1 Typical instances

instantiation *nat* :: *euclidean-semiring*
begin

definition [*simp*]:
euclidean-size-nat = (*id* :: *nat* ⇒ *nat*)

instance ⟨*proof*⟩

end

instantiation *int* :: *euclidean-ring*
begin

definition [*simp*]:
euclidean-size-int = (*nat* ∘ *abs* :: *int* ⇒ *nat*)

instance
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *euclidean-size-poly-not-0* [*simp*]:
p ≠ *0* ⟹ *euclidean-size p* = *2* ^ *degree p*
⟨*proof*⟩

instance
⟨*proof*⟩

end

instance *nat* :: *euclidean-semiring-gcd*
⟨*proof*⟩

instance *int* :: *euclidean-ring-gcd*
⟨*proof*⟩

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 ⟨*proof*⟩
end

lemma *poly-gcd-monic*:
lead-coeff (*gcd x y*) = (*if x* = *0* ∧ *y* = *0 then 0 else 1*)
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *poly-gcd-code* [*code*]:
gcd x y = (*if y* = *0 then normalize x else gcd y* (*x mod* (*y* :: *- poly*)))
⟨*proof*⟩

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*
  ⟨*proof*⟩

notation *fps-nth* (**infixl** $ *75*)

lemma *expand-fps-eq*: *p* = *q* ⟷ (∀ *n*. *p* $ *n* = *q* $ *n*)
  ⟨*proof*⟩

lemma *fps-ext*: (⋀*n*. *p* $ *n* = *q* $ *n*) ⟹ *p* = *q*
  ⟨*proof*⟩

lemma *fps-nth-Abs-fps* [*simp*]: *Abs-fps f* $ *n* = *f n*
  ⟨*proof*⟩

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 ⟨*proof*⟩
end

lemma *fps-zero-nth* [*simp*]: *0* $ *n* = *0*
  ⟨*proof*⟩

instantiation *fps* :: ({*one*, *zero*}) *one*
begin
  definition *fps-one-def*: *1* = *Abs-fps* (λ*n*. *if n* = *0 then 1 else 0*)
  instance ⟨*proof*⟩
end

lemma *fps-one-nth* [*simp*]: *1* $ *n* = (*if n* = *0 then 1 else 0*)
  ⟨*proof*⟩

instantiation *fps* :: (*plus*) *plus*
begin
  definition *fps-plus-def*: *op* + = (λ*f g*. *Abs-fps* (λ*n*. *f* $ *n* + *g* $ *n*))
  instance ⟨*proof*⟩
end

lemma *fps-add-nth* [*simp*]: (*f* + *g*) $ *n* = *f* $ *n* + *g* $ *n*

⟨*proof*⟩

instantiation *fps* :: (*minus*) *minus*
begin
definition *fps-minus-def*: *op* − = (λ*f g*. *Abs-fps* (λ*n*. *f* $ *n* − *g* $ *n*))
instance ⟨*proof*⟩
end

lemma *fps-sub-nth* [*simp*]: (*f* − *g*) $ *n* = *f* $ *n* − *g* $ *n*
⟨*proof*⟩

instantiation *fps* :: (*uminus*) *uminus*
begin
definition *fps-uminus-def*: *uminus* = (λ*f*. *Abs-fps* (λ*n*. − (*f* $ *n*)))
instance ⟨*proof*⟩
end

lemma *fps-neg-nth* [*simp*]: (− *f*) $ *n* = − (*f* $ *n*)
⟨*proof*⟩

instantiation *fps* :: ({*comm-monoid-add*, *times*}) *times*
begin
definition *fps-times-def*: *op* ∗ = (λ*f g*. *Abs-fps* (λ*n*. ∑*i*=*0*..*n*. *f* $ *i* ∗ *g* $ (*n* − *i*)))
instance ⟨*proof*⟩
end

lemma *fps-mult-nth*: (*f* ∗ *g*) $ *n* = (∑*i*=*0*..*n*. *f*$*i* ∗ *g*$(*n* − *i*))
⟨*proof*⟩

lemma *fps-mult-nth-0* [*simp*]: (*f* ∗ *g*) $ *0* = *f* $ *0* ∗ *g* $ *0*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *cond-value-iff*: *f* (*if b then x else y*) = (*if b then f x else f y*)
⟨*proof*⟩

lemma *cond-application-beta*: (*if b then f else g*) *x* = (*if b then f x else g x*)
⟨*proof*⟩

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*⟩

instance *fps* :: (*ab-semigroup-add*) *ab-semigroup-add*
⟨*proof*⟩

lemma *fps-mult-assoc-lemma*:
fixes *k* :: *nat*
and *f* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *'a*::*comm-monoid-add*
shows (∑*j*=*0*..*k*. ∑*i*=*0*..*j*. *f i* (*j* − *i*) (*n* − *j*)) =
(∑*j*=*0*..*k*. ∑*i*=*0*..*k* − *j*. *f j i* (*n* − *j* − *i*))
⟨*proof*⟩

instance *fps* :: (*semiring-0*) *semigroup-mult*
⟨*proof*⟩

lemma *fps-mult-commute-lemma*:
fixes *n* :: *nat*
and *f* :: *nat* ⇒ *nat* ⇒ *'a*::*comm-monoid-add*
shows (∑*i*=*0*..*n*. *f i* (*n* − *i*)) = (∑*i*=*0*..*n*. *f* (*n* − *i*) *i*)
⟨*proof*⟩

instance *fps* :: (*comm-semiring-0*) *ab-semigroup-mult*
⟨*proof*⟩

instance *fps* :: (*monoid-add*) *monoid-add*
⟨*proof*⟩

instance *fps* :: (*comm-monoid-add*) *comm-monoid-add*
⟨*proof*⟩

instance *fps* :: (*semiring-1*) *monoid-mult*
⟨*proof*⟩

instance *fps* :: (*cancel-semigroup-add*) *cancel-semigroup-add*
⟨*proof*⟩

instance *fps* :: (*cancel-ab-semigroup-add*) *cancel-ab-semigroup-add*
⟨*proof*⟩

instance *fps* :: (*cancel-comm-monoid-add*) *cancel-comm-monoid-add* ⟨*proof*⟩

instance *fps* :: (*group-add*) *group-add*
⟨*proof*⟩

instance *fps* :: (*ab-group-add*) *ab-group-add*
⟨*proof*⟩

instance *fps* :: (*zero-neq-one*) *zero-neq-one*
⟨*proof*⟩

instance *fps* :: (*semiring-0*) *semiring*
⟨*proof*⟩

instance *fps* :: (*semiring-0*) *semiring-0*
⟨*proof*⟩

instance *fps* :: (*semiring-0-cancel*) *semiring-0-cancel* ⟨*proof*⟩

instance *fps* :: (*semiring-1*) *semiring-1* ⟨*proof*⟩

43.3 Selection of the nth power of the implicit variable in the infinite sum

lemma *fps-nonzero-nth*: $f \neq 0 \longleftrightarrow (\exists\ n.\ f\ \$n \neq 0)$
⟨*proof*⟩

lemma *fps-nonzero-nth-minimal*: $f \neq 0 \longleftrightarrow (\exists\, n.\ f\ \$\ n \neq 0 \wedge (\forall\, m < n.\ f\ \$\ m = 0))$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *fps-eq-iff*: $f = g \longleftrightarrow (\forall\, n.\ f\ \$\ n = g\ \$n)$
⟨*proof*⟩

lemma *fps-setsum-nth*: *setsum f S* \$ *n* = *setsum* $(\lambda k.\ (f\ k)\ \$\ n)\ S$
⟨*proof*⟩

43.4 Injection of the basic ring elements and multiplication by scalars

definition *fps-const c* = *Abs-fps* $(\lambda n.$ *if n* = *0 then c else 0*$)$

lemma *fps-nth-fps-const* [*simp*]: *fps-const c* \$ *n* = (*if n* = *0 then c else 0*)
⟨*proof*⟩

lemma *fps-const-0-eq-0* [*simp*]: *fps-const 0* = *0*
⟨*proof*⟩

lemma *fps-const-1-eq-1* [*simp*]: *fps-const 1* = *1*

⟨*proof*⟩

lemma *fps-const-neg* [*simp*]: − (*fps-const* (*c*::′*a*::*ring*)) = *fps-const* (− *c*)
⟨*proof*⟩

lemma *fps-const-add* [*simp*]: *fps-const* (*c*::′*a*::*monoid-add*) + *fps-const d* = *fps-const* (*c* + *d*)
⟨*proof*⟩

lemma *fps-const-sub* [*simp*]: *fps-const* (*c*::′*a*::*group-add*) − *fps-const d* = *fps-const* (*c* − *d*)
⟨*proof*⟩

lemma *fps-const-mult*[*simp*]: *fps-const* (*c*::′*a*::*ring*) ∗ *fps-const d* = *fps-const* (*c* ∗ *d*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *fps-const-add-right*: *f* + *fps-const* (*c*::′*a*::*monoid-add*) =
Abs-fps (λ*n*. *if n* = *0 then f*$*0* + *c else f*$*n*)
⟨*proof*⟩

lemma *fps-const-mult-left*: *fps-const* (*c*::′*a*::*semiring-0*) ∗ *f* = *Abs-fps* (λ*n*. *c* ∗ *f*$*n*)
⟨*proof*⟩

lemma *fps-const-mult-right*: *f* ∗ *fps-const* (*c*::′*a*::*semiring-0*) = *Abs-fps* (λ*n*. *f*$*n* ∗ *c*)
⟨*proof*⟩

lemma *fps-mult-left-const-nth* [*simp*]: (*fps-const* (*c*::′*a*::*semiring-1*) ∗ *f*)$*n* = *c*∗ *f*$*n*
⟨*proof*⟩

lemma *fps-mult-right-const-nth* [*simp*]: (*f* ∗ *fps-const* (*c*::′*a*::*semiring-1*))$*n* = *f*$*n* ∗ *c*
⟨*proof*⟩

43.5 Formal power series form an integral domain

instance *fps* :: (*ring*) *ring* ⟨*proof*⟩

instance *fps* :: (*ring-1*) *ring-1*
⟨*proof*⟩

instance *fps* :: (*comm-ring-1*) *comm-ring-1*

⟨*proof*⟩

instance *fps* :: (*ring-no-zero-divisors*) *ring-no-zero-divisors*
⟨*proof*⟩

instance *fps* :: (*ring-1-no-zero-divisors*) *ring-1-no-zero-divisors* ⟨*proof*⟩

instance *fps* :: (*idom*) *idom* ⟨*proof*⟩

lemma *numeral-fps-const*: *numeral k* = *fps-const* (*numeral k*)
⟨*proof*⟩

lemma *neg-numeral-fps-const*:
(− *numeral k* :: *'a* :: *ring-1 fps*) = *fps-const* (− *numeral k*)
⟨*proof*⟩

lemma *fps-numeral-nth*: *numeral n* $ *i* = (*if i* = *0 then numeral n else 0*)
⟨*proof*⟩

lemma *fps-numeral-nth-0* [*simp*]: *numeral n* $ *0* = *numeral n*
⟨*proof*⟩

43.6 The eXtractor series X

lemma *minus-one-power-iff*: (− (*1*::*'a*::*comm-ring-1*)) ^ *n* = (*if even n then 1 else* − *1*)
⟨*proof*⟩

definition *X* = *Abs-fps* (*λn. 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*⟩

lemma *X-mult-right-nth*[*simp*]:
((*f* :: *'a*::*comm-semiring-1 fps*) ∗ *X*) $*n* = (*if n* = *0 then 0 else f* $ (*n* − *1*))
⟨*proof*⟩

lemma *X-power-iff*: *X*^*k* = *Abs-fps* (*λn. if n* = *k then 1*::*'a*::*comm-ring-1 else 0*)
⟨*proof*⟩

lemma *X-nth*[*simp*]: *X*$*n* = (*if n* = *1 then 1 else 0*)
⟨*proof*⟩

lemma *X-power-nth*[*simp*]: (*X*^*k*) $*n* = (*if n* = *k then 1 else 0*::*'a*::*comm-ring-1*)
⟨*proof*⟩

lemma *X-power-mult-nth*: (*X*^*k* ∗ (*f* :: *'a*::*comm-ring-1 fps*)) $*n* = (*if n* < *k then 0 else f* $ (*n* − *k*))

⟨*proof*⟩

lemma *X-power-mult-right-nth*:
((*f* :: 'a::*comm-ring-1 fps*) * *X*^*k*) $*n* = (*if n* < *k then 0 else f* $ (*n* − *k*))
⟨*proof*⟩

lemma *X-neq-fps-const* [*simp*]: (*X* :: 'a :: *zero-neq-one fps*) ≠ *fps-const c*
⟨*proof*⟩

lemma *X-neq-zero* [*simp*]: (*X* :: 'a :: *zero-neq-one fps*) ≠ *0*
⟨*proof*⟩

lemma *X-neq-one* [*simp*]: (*X* :: 'a :: *zero-neq-one fps*) ≠ *1*
⟨*proof*⟩

lemma *X-neq-numeral* [*simp*]: (*X* :: 'a :: {*semiring-1*,*zero-neq-one*} *fps*) ≠ *numeral c*
⟨*proof*⟩

lemma *X-pow-eq-X-pow-iff* [*simp*]:
(*X* :: ('a :: {*comm-ring-1*}) *fps*) ^ *m* = *X* ^ *n* ⟷ *m* = *n*
⟨*proof*⟩

43.7 Subdegrees

definition *subdegree* :: ('a::*zero*) *fps* ⇒ *nat* **where**
subdegree f = (*if f* = *0 then 0 else LEAST n*. *f*$*n* ≠ *0*)

lemma *subdegreeI*:
assumes *f* $ *d* ≠ *0* **and** ⋀*i*. *i* < *d* ⟹ *f* $ *i* = *0*
shows *subdegree f* = *d*
⟨*proof*⟩

lemma *nth-subdegree-nonzero* [*simp*,*intro*]: *f* ≠ *0* ⟹ *f* $ *subdegree f* ≠ *0*
⟨*proof*⟩

lemma *nth-less-subdegree-zero* [*dest*]: *n* < *subdegree f* ⟹ *f* $ *n* = *0*
⟨*proof*⟩

lemma *subdegree-geI*:
assumes *f* ≠ *0* ⋀*i*. *i* < *n* ⟹ *f*$*i* = *0*
shows *subdegree f* ≥ *n*
⟨*proof*⟩

lemma *subdegree-greaterI*:
assumes *f* ≠ *0* ⋀*i*. *i* ≤ *n* ⟹ *f*$*i* = *0*
shows *subdegree f* > *n*
⟨*proof*⟩

lemma *subdegree-leI*:
f $ n ≠ 0 ⟹ subdegree f ≤ n
⟨*proof*⟩

lemma *subdegree-0* [*simp*]: *subdegree 0 = 0*
⟨*proof*⟩

lemma *subdegree-1* [*simp*]: *subdegree (1 :: ('a :: zero-neq-one) fps) = 0*
⟨*proof*⟩

lemma *subdegree-X* [*simp*]: *subdegree (X :: ('a :: zero-neq-one) fps) = 1*
⟨*proof*⟩

lemma *subdegree-fps-const* [*simp*]: *subdegree (fps-const c) = 0*
⟨*proof*⟩

lemma *subdegree-numeral* [*simp*]: *subdegree (numeral n) = 0*
⟨*proof*⟩

lemma *subdegree-eq-0-iff*: *subdegree f = 0 ⟷ f = 0 ∨ f $ 0 ≠ 0*
⟨*proof*⟩

lemma *subdegree-eq-0* [*simp*]: *f $ 0 ≠ 0 ⟹ subdegree f = 0*
⟨*proof*⟩

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*⟩

lemma *subdegree-mult* [*simp*]:
assumes *f ≠ 0 g ≠ 0*
shows *subdegree ((f :: ('a :: {ring-no-zero-divisors}) fps) * g) = subdegree f + subdegree g*
⟨*proof*⟩

lemma *subdegree-power* [*simp*]:
*subdegree ((f :: ('a :: ring-1-no-zero-divisors) fps) ^ n) = n * subdegree f*
⟨*proof*⟩

lemma *subdegree-uminus* [*simp*]:
subdegree (−(f::('a::group-add) fps)) = subdegree f
⟨*proof*⟩

lemma *subdegree-minus-commute* [*simp*]:
subdegree (f−(g::('a::group-add) fps)) = subdegree (g − f)
⟨*proof*⟩

lemma *subdegree-add-ge*:
assumes $f \neq -(g :: ('a :: \{group\text{-}add\})\ fps)$
shows *subdegree* $(f + g) \geq min\ (subdegree\ f)\ (subdegree\ g)$
⟨*proof*⟩

lemma *subdegree-add-eq1*:
assumes $f \neq 0$
assumes *subdegree* $f <$ *subdegree* $(g :: ('a :: \{group\text{-}add\})\ fps)$
shows *subdegree* $(f + g) =$ *subdegree* f
⟨*proof*⟩

lemma *subdegree-add-eq2*:
assumes $g \neq 0$
assumes *subdegree* $g <$ *subdegree* $(f :: ('a :: \{ab\text{-}group\text{-}add\})\ fps)$
shows *subdegree* $(f + g) =$ *subdegree* g
⟨*proof*⟩

lemma *subdegree-diff-eq1*:
assumes $f \neq 0$
assumes *subdegree* $f <$ *subdegree* $(g :: ('a :: \{ab\text{-}group\text{-}add\})\ fps)$
shows *subdegree* $(f - g) =$ *subdegree* f
⟨*proof*⟩

lemma *subdegree-diff-eq2*:
assumes $g \neq 0$
assumes *subdegree* $g <$ *subdegree* $(f :: ('a :: \{ab\text{-}group\text{-}add\})\ fps)$
shows *subdegree* $(f - g) =$ *subdegree* g
⟨*proof*⟩

lemma *subdegree-diff-ge* [*simp*]:
assumes $f \neq (g :: ('a :: \{group\text{-}add\})\ fps)$
shows *subdegree* $(f - g) \geq min\ (subdegree\ f)\ (subdegree\ g)$
⟨*proof*⟩

43.8 Shifting and slicing

definition *fps-shift* :: *nat* ⇒ *'a fps* ⇒ *'a fps* **where**
fps-shift $n\ f =$ *Abs-fps* $(\lambda i.\ f\ \$\ (i + n))$

lemma *fps-shift-nth* [*simp*]: *fps-shift* $n\ f\ \$\ i = f\ \$\ (i + n)$
⟨*proof*⟩

lemma *fps-shift-0* [*simp*]: *fps-shift* $0\ f = f$
⟨*proof*⟩

lemma *fps-shift-zero* [*simp*]: *fps-shift* $n\ 0 = 0$
⟨*proof*⟩

lemma *fps-shift-one*: *fps-shift n 1 = (if n = 0 then 1 else 0)*
⟨*proof*⟩

lemma *fps-shift-fps-const*: *fps-shift n (fps-const c) = (if n = 0 then fps-const c else 0)*
⟨*proof*⟩

lemma *fps-shift-numeral*: *fps-shift n (numeral c) = (if n = 0 then numeral c else 0)*
⟨*proof*⟩

lemma *fps-shift-X-power* [*simp*]:
n ≤ m ⟹ fps-shift n (X ^ m) = (X ^ (m − n) ::'a::comm-ring-1 fps)
⟨*proof*⟩

lemma *fps-shift-times-X-power*:
*n ≤ subdegree f ⟹ fps-shift n f * X ^ n = (f :: 'a :: comm-ring-1 fps)*
⟨*proof*⟩

lemma *fps-shift-times-X-power'* [*simp*]:
*fps-shift n (f * X^n) = (f :: 'a :: comm-ring-1 fps)*
⟨*proof*⟩

lemma *fps-shift-times-X-power''*:
*m ≤ n ⟹ fps-shift n (f * X^m) = fps-shift (n − m) (f :: 'a :: comm-ring-1 fps)*
⟨*proof*⟩

lemma *fps-shift-subdegree* [*simp*]:
n ≤ subdegree f ⟹ subdegree (fps-shift n f) = subdegree (f :: 'a :: comm-ring-1 fps) − n
⟨*proof*⟩

lemma *subdegree-decompose*:
*f = fps-shift (subdegree f) f * X ^ subdegree (f :: ('a :: comm-ring-1) fps)*
⟨*proof*⟩

lemma *subdegree-decompose'*:
*n ≤ subdegree (f :: ('a :: comm-ring-1) fps) ⟹ f = fps-shift n f * X^n*
⟨*proof*⟩

lemma *fps-shift-fps-shift*:
fps-shift (m + n) f = fps-shift m (fps-shift n f)
⟨*proof*⟩

lemma *fps-shift-add*:
fps-shift n (f + g) = fps-shift n f + fps-shift n g
⟨*proof*⟩

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*⟩

lemma *fps-shift-mult-right*:
assumes $n \leq$ *subdegree* $(g :: 'b :: \{$*comm-ring-1*$\}$ *fps*)
shows *fps-shift* n $(g*h) = h *$ *fps-shift* n g
⟨*proof*⟩

lemma *nth-subdegree-zero-iff* [*simp*]: $f \$$ *subdegree* $f = 0 \longleftrightarrow f = 0$
⟨*proof*⟩

lemma *fps-shift-subdegree-zero-iff* [*simp*]:
fps-shift (*subdegree* f) $f = 0 \longleftrightarrow f = 0$
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *fps-cutoff-zero-iff*: *fps-cutoff* n $f = 0 \longleftrightarrow (f = 0 \vee n \leq$ *subdegree* $f)$
⟨*proof*⟩

lemma *fps-cutoff-0* [*simp*]: *fps-cutoff* 0 $f = 0$
⟨*proof*⟩

lemma *fps-cutoff-zero* [*simp*]: *fps-cutoff* n $0 = 0$
⟨*proof*⟩

lemma *fps-cutoff-one*: *fps-cutoff* n $1 =$ (*if* $n = 0$ *then* 0 *else* 1)
⟨*proof*⟩

lemma *fps-cutoff-fps-const*: *fps-cutoff* n (*fps-const* c) = (*if* $n = 0$ *then* 0 *else* *fps-const* c)
⟨*proof*⟩

lemma *fps-cutoff-numeral*: *fps-cutoff* n (*numeral* c) = (*if* $n = 0$ *then* 0 *else* *numeral* c)
⟨*proof*⟩

lemma *fps-shift-cutoff*:
fps-shift n $(f :: ('a ::$ *comm-ring-1*$)$ *fps*$) * X\hat{}n +$ *fps-cutoff* n $f = f$
⟨*proof*⟩

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 ^ *subdegree* (*a* − *b*)))

lemma *dist-fps-ge0*: *dist* (*a* :: 'a *fps*) *b* ≥ *0*
⟨*proof*⟩

lemma *dist-fps-sym*: *dist* (*a* :: 'a *fps*) *b* = *dist b a*
⟨*proof*⟩

instance ⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*⟩

lemma *fps-sum-rep-nth*: (*setsum* (λ*i*. *fps-const*(*a*$*i*)∗*X*^*i*) {*0*..*m*})$*n* =
(*if n* ≤ *m then a*$*n else 0*::′*a*::*comm-ring-1*)
⟨*proof*⟩

lemma *fps-notation*: (λ*n*. *setsum* (λ*i*. *fps-const*(*a*$*i*) ∗ *X*^*i*) {*0*..*n*}) ⟶ *a*
(**is** *?s* ⟶ *a*)
⟨*proof*⟩

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* (λ*i*. *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 ⟨*proof*⟩

end

lemma *fps-inverse-zero* [*simp*]:
inverse (*0* :: ′*a*::{*comm-monoid-add*,*inverse*,*times*,*uminus*} *fps*) = *0*
⟨*proof*⟩

lemma *fps-inverse-one* [*simp*]: *inverse* (*1* :: ′*a*::{*division-ring*,*zero-neq-one*} *fps*) = *1*
⟨*proof*⟩

lemma *inverse-mult-eq-1* [*intro*]:
assumes *f0*: *f*$*0* ≠ (*0*::′*a*::*field*)
shows *inverse f* ∗ *f* = *1*
⟨*proof*⟩

lemma *fps-inverse-0-iff* [*simp*]: (*inverse f*) $ *0* = (*0*::′*a*::*division-ring*) ⟷ *f* $ *0* = *0*

⟨*proof*⟩

lemma *fps-inverse-nth-0* [*simp*]: *inverse f $ 0 = inverse* (*f $ 0* :: *'a* :: *division-ring*)
⟨*proof*⟩

lemma *fps-inverse-eq-0-iff*[*simp*]: *inverse f* = (*0*:: (*'a*::*division-ring*) *fps*) ⟷ *f $ 0 = 0*
⟨*proof*⟩

lemma *fps-inverse-idempotent*[*intro*, *simp*]:
assumes *f0*: *f$0* ≠ (*0*::*'a*::*field*)
shows *inverse* (*inverse f*) = *f*
⟨*proof*⟩

lemma *fps-inverse-unique*:
assumes *fg*: (*f* :: *'a* :: *field fps*) ∗ *g* = *1*
shows *inverse f* = *g*
⟨*proof*⟩

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*⟩

lemma *fps-inverse-mult*: *inverse* (*f* ∗ *g* :: *'a*::*field fps*) = *inverse f* ∗ *inverse g*
⟨*proof*⟩

lemma *fps-inverse-gp*: *inverse* (*Abs-fps*(λ*n*. (*1*::*'a*::*field*))) =
Abs-fps (λ*n*. *if n*= *0 then 1 else if n*=*1 then* − *1 else 0*)
⟨*proof*⟩

lemma *subdegree-inverse* [*simp*]: *subdegree* (*inverse* (*f*::*'a*::*field fps*)) = *0*
⟨*proof*⟩

lemma *fps-is-unit-iff* [*simp*]: (*f* :: *'a* :: *field fps*) *dvd 1* ⟷ *f $ 0* ≠ *0*
⟨*proof*⟩

lemma *subdegree-eq-0'* [*simp*]: (*f* :: *'a* :: *field fps*) *dvd 1* ⟹ *subdegree f* = *0*
⟨*proof*⟩

lemma *fps-unit-dvd* [*simp*]: (*f $ 0* :: *'a* :: *field*) ≠ *0* ⟹ *f dvd g*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *fps-times-divide-eq*:
assumes *g* ≠ *0* **and** *subdegree f* ≥ *subdegree* (*g* :: *'a fps*)
shows *f div g* ∗ *g* = *f*
⟨*proof*⟩

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*⟩

context
begin
private lemma *fps-divide-cancel-aux1*:
assumes *h*$*0* ≠ (*0* :: *'a* :: *field*)
shows (*h* ∗ *f*) *div* (*h* ∗ *g*) = *f div g*
⟨*proof*⟩ **lemma** *fps-divide-cancel-aux2*:
(*f* ∗ *X*^*m*) *div* (*g* ∗ *X*^*m*) = *f div* (*g* :: *'a* :: *field fps*)
⟨*proof*⟩

instance ⟨*proof*⟩

end
end

lemma *subdegree-mod*:
assumes *f* ≠ *0 subdegree f* < *subdegree g*
shows *subdegree* (*f mod g*) = *subdegree f*
⟨*proof*⟩

lemma *fps-divide-nth-0* [*simp*]: *g* $ *0* ≠ *0* ⟹ (*f div g*) $ *0* = *f* $ *0* / (*g* $ *0* :: - :: *field*)
⟨*proof*⟩

lemma *dvd-imp-subdegree-le*:
(*f* :: *'a* :: *idom fps*) *dvd g* ⟹ *g* ≠ *0* ⟹ *subdegree f* ≤ *subdegree g*

⟨*proof*⟩

lemma *fps-dvd-iff*:
assumes (*f* :: *'a* :: *field fps*) ≠ *0 g* ≠ *0*
shows *f dvd g* ⟷ *subdegree f* ≤ *subdegree g*
⟨*proof*⟩

lemma *fps-const-inverse*: *inverse* (*fps-const* (*a*::*'a*::*field*)) = *fps-const* (*inverse a*)
⟨*proof*⟩

lemma *fps-const-divide*: *fps-const* (*x* :: - :: *field*) / *fps-const y* = *fps-const* (*x* / *y*)
⟨*proof*⟩

lemma *inverse-fps-numeral*:
inverse (*numeral n* :: (*'a* :: *field-char-0*) *fps*) = *fps-const* (*inverse* (*numeral n*))
⟨*proof*⟩

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*⟩

end

instance *fps* :: (*field*) *algebraic-semidom* ⟨*proof*⟩

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* ^ *subdegree f*)

instance ⟨*proof*⟩

end

instantiation *fps* :: (*field*) *euclidean-ring-gcd*
begin

definition *fps-gcd-def*: $(gcd :: 'a\ fps \Rightarrow -) = gcd\text{-}eucl$
definition *fps-lcm-def*: $(lcm :: 'a\ fps \Rightarrow -) = lcm\text{-}eucl$
definition *fps-Gcd-def*: $(Gcd :: 'a\ fps\ set \Rightarrow -) = Gcd\text{-}eucl$
definition *fps-Lcm-def*: $(Lcm :: 'a\ fps\ set \Rightarrow -) = Lcm\text{-}eucl$
instance ⟨*proof*⟩
end

lemma *fps-gcd*:
  assumes [*simp*]: $f \neq 0\ g \neq 0$
  shows $gcd\ f\ g = X\ \hat{}\ min\ (subdegree\ f)\ (subdegree\ g)$
⟨*proof*⟩

lemma *fps-gcd-altdef*: $gcd\ (f :: 'a :: field\ fps)\ g =$
  $(if\ f = 0 \wedge g = 0\ then\ 0\ else$
   $if\ f = 0\ then\ X\ \hat{}\ subdegree\ g\ else$
   $if\ g = 0\ then\ X\ \hat{}\ subdegree\ f\ else$
     $X\ \hat{}\ min\ (subdegree\ f)\ (subdegree\ g))$
  ⟨*proof*⟩

lemma *fps-lcm*:
  assumes [*simp*]: $f \neq 0\ g \neq 0$
  shows $lcm\ f\ g = X\ \hat{}\ max\ (subdegree\ f)\ (subdegree\ g)$
⟨*proof*⟩

lemma *fps-lcm-altdef*: $lcm\ (f :: 'a :: field\ fps)\ g =$
  $(if\ f = 0 \vee g = 0\ then\ 0\ else\ X\ \hat{}\ max\ (subdegree\ f)\ (subdegree\ g))$
  ⟨*proof*⟩

lemma *fps-Gcd*:
  assumes $A - \{0\} \neq \{\}$
  shows $Gcd\ A = X\ \hat{}\ (INF\ f{:}A-\{0\}.\ subdegree\ f)$
⟨*proof*⟩

lemma *fps-Gcd-altdef*: $Gcd\ (A :: 'a :: field\ fps\ set) =$
  $(if\ A \subseteq \{0\}\ then\ 0\ else\ X\ \hat{}\ (INF\ f{:}A-\{0\}.\ subdegree\ f))$
  ⟨*proof*⟩

lemma *fps-Lcm*:
  assumes $A \neq \{\}\ 0 \notin A\ bdd\text{-}above\ (subdegree`A)$
  shows $Lcm\ A = X\ \hat{}\ (SUP\ f{:}A.\ subdegree\ f)$
⟨*proof*⟩

lemma *fps-Lcm-altdef*:
  $Lcm\ (A :: 'a :: field\ fps\ set) =$
    $(if\ 0 \in A \vee \neg bdd\text{-}above\ (subdegree`A)\ then\ 0\ else$
     $if\ A = \{\}\ then\ 1\ else\ X\ \hat{}\ (SUP\ f{:}A.\ subdegree\ f))$
⟨*proof*⟩

43.12 Formal Derivatives, and the MacLaurin theorem around 0

definition *fps-deriv f = Abs-fps (λn. of-nat (n + 1) * f $ (n + 1))*

lemma *fps-deriv-nth*[*simp*]: *fps-deriv f $ n = of-nat (n +1) * f $ (n + 1)*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *fps-deriv-X*[*simp*]: *fps-deriv X = 1*
⟨*proof*⟩

lemma *fps-deriv-neg*[*simp*]:
fps-deriv (− (f:: 'a::comm-ring-1 fps)) = − (fps-deriv f)
⟨*proof*⟩

lemma *fps-deriv-add*[*simp*]:
fps-deriv ((f:: 'a::comm-ring-1 fps) + g) = fps-deriv f + fps-deriv g
⟨*proof*⟩

lemma *fps-deriv-sub*[*simp*]:
fps-deriv ((f:: 'a::comm-ring-1 fps) − g) = fps-deriv f − fps-deriv g
⟨*proof*⟩

lemma *fps-deriv-const*[*simp*]: *fps-deriv (fps-const c) = 0*
⟨*proof*⟩

lemma *fps-deriv-mult-const-left*[*simp*]:
*fps-deriv (fps-const (c::'a::comm-ring-1) * f) = fps-const c * fps-deriv f*
⟨*proof*⟩

lemma *fps-deriv-0*[*simp*]: *fps-deriv 0 = 0*
⟨*proof*⟩

lemma *fps-deriv-1*[*simp*]: *fps-deriv 1 = 0*
⟨*proof*⟩

lemma *fps-deriv-mult-const-right*[*simp*]:
*fps-deriv (f * fps-const (c::'a::comm-ring-1)) = fps-deriv f * fps-const c*
⟨*proof*⟩

lemma *fps-deriv-setsum*:
fps-deriv (*setsum f S*) = *setsum* (λ*i*. *fps-deriv* (*f i* :: *'a*::*comm-ring-1 fps*)) *S*
⟨*proof*⟩

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*⟩

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*⟩

lemma *fps-deriv-eq-iff-ex*:
(*fps-deriv f* = *fps-deriv g*) ⟷ (∃ *c*::*'a*::{*idom*,*semiring-char-0*}. *f* = *fps-const c* + *g*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *fps-nth-deriv-neg*[*simp*]:
fps-nth-deriv n (− (*f* :: *'a*::*comm-ring-1 fps*)) = − (*fps-nth-deriv n f*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *fps-nth-deriv-0*[*simp*]: *fps-nth-deriv n 0* = *0*

⟨*proof*⟩

lemma *fps-nth-deriv-1*[*simp*]: *fps-nth-deriv n 1* = (*if n* = *0 then 1 else 0*)
⟨*proof*⟩

lemma *fps-nth-deriv-const*[*simp*]:
fps-nth-deriv n (*fps-const c*) = (*if n* = *0 then fps-const c else 0*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *fps-nth-deriv-setsum*:
fps-nth-deriv n (*setsum f S*) = *setsum* (λ*i*. *fps-nth-deriv n* (*f i* :: ′*a*::*comm-ring-1 fps*)) *S*
⟨*proof*⟩

lemma *fps-deriv-maclauren-0*:
(*fps-nth-deriv k* (*f* :: ′*a*::*comm-semiring-1 fps*)) $ *0* = *of-nat* (*fact k*) ∗ *f* $ *k*
⟨*proof*⟩

43.13 Powers

lemma *fps-power-zeroth-eq-one*: *a*$*0* =*1* ⟹ *a*^*n* $ *0* = (*1*::′*a*::*semiring-1*)
⟨*proof*⟩

lemma *fps-power-first-eq*: (*a* :: ′*a*::*comm-ring-1 fps*) $ *0* =*1* ⟹ *a*^*n* $ *1* = *of-nat n* ∗ *a*$*1*
⟨*proof*⟩

lemma *startsby-one-power*:*a* $ *0* = (*1*::′*a*::*comm-ring-1*) ⟹ *a*^*n* $ *0* = *1*
⟨*proof*⟩

lemma *startsby-zero-power*:*a* $*0* = (*0*::′*a*::*comm-ring-1*) ⟹ *n* > *0* ⟹ *a*^*n* $*0* = *0*
⟨*proof*⟩

lemma *startsby-power*:*a* $*0* = (*v*::′*a*::*comm-ring-1*) ⟹ *a*^*n* $*0* = *v*^*n*
⟨*proof*⟩

lemma *startsby-zero-power-iff*[*simp*]: *a*^*n* $*0* = (*0*::′*a*::*idom*) ⟷ *n* ≠ *0* ∧ *a*$*0* = *0*

⟨*proof*⟩

lemma *startsby-zero-power-prefix*:
assumes *a0*: *a $ 0 = (0::'a::idom)*
shows *∀ n < k. a ^ k $ n = 0*
⟨*proof*⟩

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}*
⟨*proof*⟩

lemma *startsby-zero-power-nth-same*:
assumes *a0*: *a$0 = (0::'a::idom)*
shows *a^n $ n = (a$1) ^ n*
⟨*proof*⟩

lemma *fps-inverse-power*:
fixes *a :: 'a::field fps*
shows *inverse (a^n) = inverse a ^ n*
⟨*proof*⟩

lemma *fps-deriv-power*:
*fps-deriv (a ^ n) = fps-const (of-nat n :: 'a::comm-ring-1) * fps-deriv a * a ^ (n − 1)*
⟨*proof*⟩

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*⟩

lemma *fps-inverse-deriv'*:
fixes *a :: 'a::field fps*
assumes *a0*: *a $ 0 ≠ 0*
shows *fps-deriv (inverse a) = − fps-deriv a / a²*
⟨*proof*⟩

lemma *inverse-mult-eq-1'*:
assumes *f0*: *f$0 ≠ (0::'a::field)*
shows *f * inverse f = 1*
⟨*proof*⟩

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^2*

⟨*proof*⟩

lemma *fps-inverse-gp'*: *inverse* (*Abs-fps* (λ*n*. *1*::'*a*::*field*)) = *1* − *X*
⟨*proof*⟩

lemma *fps-nth-deriv-X*[*simp*]: *fps-nth-deriv n X* = (*if n* = *0 then X else if n*=*1 then 1 else 0*)
⟨*proof*⟩

lemma *fps-inverse-X-plus1*: *inverse* (*1* + *X*) = *Abs-fps* (λ*n*. (− (*1*::'*a*::*field*)) ^ *n*)
(**is** - = *?r*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

43.15 Composition of FPSs

definition *fps-compose* :: '*a*::*semiring-1 fps* ⇒ '*a fps* ⇒ '*a fps* (**infixl** *oo 55*)
where *a oo b* = *Abs-fps* (λ*n*. *setsum* (λ*i*. *a*$*i* ∗ (*b* ^ *i*$*n*)) {*0*..*n*})

lemma *fps-compose-nth*: (*a oo b*)$*n* = *setsum* (λ*i*. *a*$*i* ∗ (*b* ^ *i*$*n*)) {*0*..*n*}
⟨*proof*⟩

lemma *fps-compose-nth-0* [*simp*]: (*f oo g*) $ *0* = *f* $ *0*
⟨*proof*⟩

lemma *fps-compose-X*[*simp*]: *a oo X* = (*a* :: '*a*::*comm-ring-1 fps*)
⟨*proof*⟩

lemma *fps-const-compose*[*simp*]: *fps-const* (*a*::'*a*::*comm-ring-1*) *oo b* = *fps-const a*
⟨*proof*⟩

lemma *numeral-compose*[*simp*]: (*numeral k* :: '*a*::*comm-ring-1 fps*) *oo b* = *numeral k*
⟨*proof*⟩

lemma *neg-numeral-compose*[*simp*]: (− *numeral k* :: *'a*::*comm-ring-1 fps*) *oo b* = − *numeral k*
⟨*proof*⟩

lemma *X-fps-compose-startby0*[*simp*]: *a*$*0* = *0* ⟹ *X oo a* = (*a* :: *'a*::*comm-ring-1 fps*)
⟨*proof*⟩

43.16 Rules from Herbert Wilf's Generatingfunctionology

43.16.1 Rule 1

lemma *fps-power-mult-eq-shift*:
X^*Suc k* ∗ *Abs-fps* (λ*n*. *a* (*n* + *Suc k*)) =
Abs-fps a − *setsum* (λ*i*. *fps-const* (*a i* :: *'a*::*comm-ring-1*) ∗ *X*^*i*) {*0* .. *k*}
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *XD-mult-const*[*simp*]:*XD* (*fps-const* (*c*::*'a*::*comm-ring-1*) ∗ *a*) = *fps-const c* ∗ *XD a*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *XDN-linear*:
(*XD* ^^ *n*) (*fps-const c* ∗ *a* + *fps-const d* ∗ *b*) =
fps-const c ∗ (*XD* ^^ *n*) *a* + *fps-const d* ∗ (*XD* ^^ *n*) (*b* :: *'a*::*comm-ring-1 fps*)
⟨*proof*⟩

lemma *fps-mult-X-deriv-shift*: *X*∗ *fps-deriv a* = *Abs-fps* (λ*n*. *of-nat n*∗ *a*$*n*)
⟨*proof*⟩

lemma *fps-mult-XD-shift*:
(*XD* ^^ *k*) (*a* :: *'a*::*comm-ring-1 fps*) = *Abs-fps* (λ*n*. (*of-nat n* ^ *k*) ∗ *a*$*n*)
⟨*proof*⟩

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*⟩

lemma *fps-divide-X-minus1-setsum*:
$a\ /((1::'a::field\ fps) - X) = Abs\text{-}fps\ (\lambda n.\ setsum\ (\lambda i.\ a\ \$\ i)\ \{0..n\})$
⟨*proof*⟩

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 $natpermute\ n\ k = \{l :: nat\ list.\ length\ l = k \wedge listsum\ l = n\}$

lemma *natlist-trivial-1*: $natpermute\ n\ 1 = \{[n]\}$
⟨*proof*⟩

lemma *append-natpermute-less-eq*:
assumes $xs\ @\ ys \in natpermute\ n\ k$
shows $listsum\ xs \leq n$
and $listsum\ ys \leq n$
⟨*proof*⟩

lemma *natpermute-split*:
assumes $h \leq k$
shows $natpermute\ n\ k =$
$(\bigcup m \in \{0..n\}.\ \{l1\ @\ l2\ |l1\ l2.\ l1 \in natpermute\ m\ h \wedge l2 \in natpermute\ (n - m)\ (k - h)\})$
(**is** $?L = ?R$ **is** - = $(\bigcup m \in \{0..n\}.\ ?S\ m)$)
⟨*proof*⟩

lemma *natpermute-0*: $natpermute\ n\ 0 = (if\ n = 0\ then\ \{[]\}\ else\ \{\})$
⟨*proof*⟩

lemma *natpermute-0'*[*simp*]: $natpermute\ 0\ k = (if\ k = 0\ then\ \{[]\}\ else\ \{replicate\ k\ 0\})$
⟨*proof*⟩

lemma *natpermute-finite*: *finite* $(natpermute\ n\ k)$
⟨*proof*⟩

lemma *natpermute-contain-maximal*:
$\{xs \in natpermute\ n\ (k + 1).\ n \in set\ xs\} = (\bigcup i \in \{0\ ..\ k\}.\ \{(replicate\ (k + 1)\ 0)\ [i:=n]\})$
(**is** $?A = ?B$)
⟨*proof*⟩

The general form.

lemma *fps-setprod-nth*:
fixes $m :: nat$
and $a :: nat \Rightarrow 'a::comm\text{-}ring\text{-}1\ fps$
shows $(setprod\ a\ \{0\ ..\ m\})\ \$\ n =$
$setsum\ (\lambda v.\ setprod\ (\lambda j.\ (a\ j)\ \$\ (v!j))\ \{0..m\})\ (natpermute\ n\ (m+1))$
(**is** *?P m n*)
⟨*proof*⟩

The special form for powers.

lemma *fps-power-nth-Suc*:
fixes $m :: nat$
and $a :: 'a::comm\text{-}ring\text{-}1\ fps$
shows $(a\ \hat{}\ Suc\ m)\$n = setsum\ (\lambda v.\ setprod\ (\lambda j.\ a\ \$\ (v!j))\ \{0..m\})\ (natpermute\ n\ (m+1))$
⟨*proof*⟩

lemma *fps-power-nth*:
fixes $m :: nat$
and $a :: 'a::comm\text{-}ring\text{-}1\ fps$
shows $(a\ \hat{}m)\$n =$
$(if\ m=0\ then\ 1\$n\ else\ setsum\ (\lambda v.\ setprod\ (\lambda j.\ a\ \$\ (v!j))\ \{0..m\ -\ 1\})\ (natpermute\ n\ m))$
⟨*proof*⟩

lemma *fps-nth-power-0*:
fixes $m :: nat$
and $a :: 'a::comm\text{-}ring\text{-}1\ fps$
shows $(a\ \hat{}m)\$0 = (a\$0)\ \hat{}\ m$
⟨*proof*⟩

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*⟩

43.17 Radicals

declare *setprod.cong* [*fundef-cong*]

function *radical* :: $(nat \Rightarrow 'a \Rightarrow 'a) \Rightarrow nat \Rightarrow 'a::field\ fps \Rightarrow nat \Rightarrow 'a$
where
$radical\ r\ 0\ a\ 0 = 1$
| $radical\ r\ 0\ a\ (Suc\ n) = 0$
| $radical\ r\ (Suc\ k)\ a\ 0 = r\ (Suc\ k)\ (a\$0)$
| $radical\ r\ (Suc\ k)\ a\ (Suc\ n) =$
$(a\$\ Suc\ n\ -\ setsum\ (\lambda xs.\ setprod\ (\lambda j.\ radical\ r\ (Suc\ k)\ a\ (xs\ !\ j))\ \{0..k\})$
$\{xs.\ xs \in natpermute\ (Suc\ n)\ (Suc\ k) \wedge Suc\ n \notin set\ xs\})\ /$
$(of\text{-}nat\ (Suc\ k) * (radical\ r\ (Suc\ k)\ a\ 0)\ \hat{}k)$

⟨*proof*⟩

termination *radical*
⟨*proof*⟩

definition *fps-radical r n a = Abs-fps (radical r n a)*

lemma *fps-radical0*[*simp*]: *fps-radical r 0 a = 1*
⟨*proof*⟩

lemma *fps-radical-nth-0*[*simp*]: *fps-radical r n a $ 0 = (if n = 0 then 1 else r n (a$0))*
⟨*proof*⟩

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*⟩

lemma *natpermute-max-card*:
assumes *n0*: *n ≠ 0*
shows *card {xs ∈ natpermute n (k + 1). n ∈ set xs} = k + 1*
⟨*proof*⟩

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*⟩

lemma *eq-divide-imp'*:
fixes *c* :: *'a*::*field*
shows *c ≠ 0 ⟹ a * c = b ⟹ a = b / c*
⟨*proof*⟩

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*⟩

lemma *radical-power*:
assumes *r0*: *r (Suc k) ((a$0) ^ Suc k) = a$0*

and *a0*: $(a\$0 :: 'a::field\text{-}char\text{-}0) \neq 0$
shows $(fps\text{-}radical\ r\ (Suc\ k)\ (a\ \hat{}\ Suc\ k)) = a$
⟨*proof*⟩

lemma *fps-deriv-radical*:
fixes $a :: 'a::field\text{-}char\text{-}0\ fps$
assumes *r0*: $(r\ (Suc\ k)\ (a\$0))\ \hat{}\ Suc\ k = a\$0$
and *a0*: $a\$0 \neq 0$
shows $fps\text{-}deriv\ (fps\text{-}radical\ r\ (Suc\ k)\ a) =$
$fps\text{-}deriv\ a\ /\ (fps\text{-}const\ (of\text{-}nat\ (Suc\ k)) * (fps\text{-}radical\ r\ (Suc\ k)\ a)\ \hat{}\ k)$
⟨*proof*⟩

lemma *radical-mult-distrib*:
fixes $a :: 'a::field\text{-}char\text{-}0\ fps$
assumes *k*: $k > 0$
and *ra0*: $r\ k\ (a\ \$\ 0)\ \hat{}\ k = a\ \$\ 0$
and *rb0*: $r\ k\ (b\ \$\ 0)\ \hat{}\ k = b\ \$\ 0$
and *a0*: $a\ \$\ 0 \neq 0$
and *b0*: $b\ \$\ 0 \neq 0$
shows $r\ k\ ((a * b)\ \$\ 0) = r\ k\ (a\ \$\ 0) * r\ k\ (b\ \$\ 0) \longleftrightarrow$
$fps\text{-}radical\ r\ k\ (a * b) = fps\text{-}radical\ r\ k\ a * fps\text{-}radical\ r\ k\ b$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *fps-divide-1* [*simp*]: $(a :: 'a::field\ fps)\ /\ 1 = a$
⟨*proof*⟩

lemma *radical-divide*:
fixes $a :: 'a::field\text{-}char\text{-}0\ fps$
assumes *kp*: $k > 0$
and *ra0*: $(r\ k\ (a\ \$\ 0))\ \hat{}\ k = a\ \$\ 0$
and *rb0*: $(r\ k\ (b\ \$\ 0))\ \hat{}\ k = b\ \$\ 0$
and *a0*: $a\$0 \neq 0$
and *b0*: $b\$0 \neq 0$
shows $r\ k\ ((a\ \$\ 0)\ /\ (b\$0)) = r\ k\ (a\$0)\ /\ r\ k\ (b\ \$\ 0) \longleftrightarrow$
$fps\text{-}radical\ r\ k\ (a/b) = fps\text{-}radical\ r\ k\ a\ /\ fps\text{-}radical\ r\ k\ b$
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *radical-inverse*:
fixes $a :: 'a::field\text{-}char\text{-}0\ fps$
assumes *k*: $k > 0$
and *ra0*: $r\ k\ (a\ \$\ 0)\ \hat{}\ k = a\ \$\ 0$
and *r1*: $(r\ k\ 1)\hat{}k = 1$
and *a0*: $a\$0 \neq 0$
shows $r\ k\ (inverse\ (a\ \$\ 0)) = r\ k\ 1\ /\ (r\ k\ (a\ \$\ 0)) \longleftrightarrow$
$fps\text{-}radical\ r\ k\ (inverse\ a) = fps\text{-}radical\ r\ k\ 1\ /\ fps\text{-}radical\ r\ k\ a$

⟨*proof*⟩

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*⟩

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*⟩

43.19 Finite FPS (i.e. polynomials) and X

lemma *fps-poly-sum-X*:
assumes ∀ *i* > *n*. *a$i* = (*0*::*'a*::*comm-ring-1*)
shows *a* = *setsum* (λ*i*. *fps-const* (*a$i*) * *X*^*i*) {*0*..*n*} (**is** *a* = *?r*)
⟨*proof*⟩

43.20 Compositional inverses

fun *compinv* :: *'a fps* ⇒ *nat* ⇒ *'a*::*field*
where
compinv a 0 = *X$0*
| *compinv a* (*Suc n*) =
(*X$ Suc n* − *setsum* (λ*i*. (*compinv a i*) * (*a*^*i*)*$Suc n*) {*0* .. *n*}) / (*a$1*) ^ *Suc n*

definition *fps-inv a* = *Abs-fps* (*compinv a*)

lemma *fps-inv*:
assumes *a0*: *a$0* = *0*
and *a1*: *a$1* ≠ *0*
shows *fps-inv a oo a* = *X*
⟨*proof*⟩

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* (λ*i*. (*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*: $a\$1 \neq 0$
shows *fps-ginv b a oo a = b*
⟨*proof*⟩

lemma *fps-inv-ginv*: *fps-inv = fps-ginv X*
⟨*proof*⟩

lemma *fps-compose-1*[*simp*]: *1 oo a = 1*
⟨*proof*⟩

lemma *fps-compose-0*[*simp*]: *0 oo a = 0*
⟨*proof*⟩

lemma *fps-compose-0-right*[*simp*]: *a oo 0 = fps-const (a $ 0)*
⟨*proof*⟩

lemma *fps-compose-add-distrib*: $(a + b)\ oo\ c = (a\ oo\ c) + (b\ oo\ c)$
⟨*proof*⟩

lemma *fps-compose-setsum-distrib*: $(setsum\ f\ S)\ oo\ a = setsum\ (\lambda i.\ f\ i\ oo\ a)\ S$
⟨*proof*⟩

lemma *convolution-eq*:
$setsum\ (\lambda i.\ a\ (i :: nat) * b\ (n - i))\ \{0\ ..\ n\} =$
$setsum\ (\lambda(i,j).\ a\ i * b\ j)\ \{(i,j).\ i \leq n \wedge j \leq n \wedge i + j = n\}$
⟨*proof*⟩

lemma *product-composition-lemma*:
assumes *c0*: $c\$0 = (0::'a::idom)$
and *d0*: $d\$0 = 0$
shows $((a\ oo\ c) * (b\ oo\ d))\$n =$
$setsum\ (\lambda(k,m).\ a\$k * b\$m * (c\hat{}k * d\hat{}m)\ \$\ n)\ \{(k,m).\ k + m \leq n\}$ (**is** *?l = ?r*)
⟨*proof*⟩

lemma *product-composition-lemma'*:
assumes *c0*: $c\$0 = (0::'a::idom)$
and *d0*: $d\$0 = 0$
shows $((a\ oo\ c) * (b\ oo\ d))\$n =$
$setsum\ (\lambda k.\ setsum\ (\lambda m.\ a\$k * b\$m * (c\hat{}k * d\hat{}m)\ \$\ n)\ \{0..n\})\ \{0..n\}$ (**is** *?l = ?r*)
⟨*proof*⟩

lemma *setsum-pair-less-iff*:
$setsum\ (\lambda((k::nat),m).\ a\ k * b\ m * c\ (k + m))\ \{(k,m).\ k + m \leq n\} =$
$setsum\ (\lambda s.\ setsum\ (\lambda i.\ a\ i * b\ (s - i) * c\ s)\ \{0..s\})\ \{0..n\}$
(**is** *?l = ?r*)
⟨*proof*⟩

lemma *fps-compose-mult-distrib-lemma*:
assumes *c0*: $c\$0 = (0::'a::idom)$
shows $((a \ oo \ c) * (b \ oo \ c))\$n = setsum \ (\lambda s. \ setsum \ (\lambda i. \ a\$i * b\$(s - i)) * (c \hat{} s) \ \$ \ n) \ \{0..s\}) \ \{0..n\}$
⟨*proof*⟩

lemma *fps-compose-mult-distrib*:
assumes *c0*: $c \ \$ \ 0 = (0::'a::idom)$
shows $(a * b) \ oo \ c = (a \ oo \ c) * (b \ oo \ c)$
⟨*proof*⟩

lemma *fps-compose-setprod-distrib*:
assumes *c0*: $c\$0 = (0::'a::idom)$
shows $setprod \ a \ S \ oo \ c = setprod \ (\lambda k. \ a \ k \ oo \ c) \ S$
⟨*proof*⟩

lemma *fps-compose-power*:
assumes *c0*: $c\$0 = (0::'a::idom)$
shows $(a \ oo \ c) \hat{} n = a \hat{} n \ oo \ c$
⟨*proof*⟩

lemma *fps-compose-uminus*: $- \ (a::'a::ring\text{-}1 \ fps) \ oo \ c = - \ (a \ oo \ c)$
⟨*proof*⟩

lemma *fps-compose-sub-distrib*: $(a - b) \ oo \ (c::'a::ring\text{-}1 \ fps) = (a \ oo \ c) - (b \ oo \ c)$
⟨*proof*⟩

lemma *X-fps-compose*: $X \ oo \ a = Abs\text{-}fps \ (\lambda n. \ if \ n = 0 \ then \ (0::'a::comm\text{-}ring\text{-}1) \ else \ a\$n)$
⟨*proof*⟩

lemma *fps-inverse-compose*:
assumes *b0*: $(b\$0 :: 'a::field) = 0$
and *a0*: $a\$0 \neq 0$
shows $inverse \ a \ oo \ b = inverse \ (a \ oo \ b)$
⟨*proof*⟩

lemma *fps-divide-compose*:
assumes *c0*: $(c\$0 :: 'a::field) = 0$
and *b0*: $b\$0 \neq 0$
shows $(a/b) \ oo \ c = (a \ oo \ c) \ / \ (b \ oo \ c)$
⟨*proof*⟩

lemma *gp*:
assumes *a0*: $a\$0 = (0::'a::field)$
shows $(Abs\text{-}fps \ (\lambda n. \ 1)) \ oo \ a = 1/(1 - a)$
(**is** *?one oo a = -*)

⟨*proof*⟩

lemma *fps-const-power* [*simp*]: *fps-const* (*c*::'*a*::*ring-1*) ^ *n* = *fps-const* (*c*^*n*)
⟨*proof*⟩

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*⟩

lemma *fps-const-mult-apply-left*: *fps-const c* ∗ (*a oo b*) = (*fps-const c* ∗ *a*) *oo b*
⟨*proof*⟩

lemma *fps-const-mult-apply-right*:
(*a oo b*) ∗ *fps-const* (*c*::'*a*::*comm-semiring-1*) = (*fps-const c* ∗ *a*) *oo b*
⟨*proof*⟩

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*⟩

lemma *fps-X-power-compose*:
assumes *a0*: *a*$*0*=*0*
shows *X*^*k oo a* = (*a*::'*a*::*idom fps*) ^*k*
(**is** *?l* = *?r*)
⟨*proof*⟩

lemma *fps-inv-right*:
assumes *a0*: *a*$*0* = *0*
and *a1*: *a*$*1* ≠ *0*
shows *a oo fps-inv a* = *X*
⟨*proof*⟩

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*⟩

lemma *fps-inv-idempotent*:
assumes *a0*: *a*$*0* = *0*
and *a1*: *a*$*1* ≠ *0*
shows *fps-inv* (*fps-inv a*) = *a*
⟨*proof*⟩

lemma *fps-ginv-ginv*:
assumes *a0*: $a\$0 = 0$
and *a1*: $a\$1 \neq 0$
and *c0*: $c\$0 = 0$
and *c1*: $c\$1 \neq 0$
shows *fps-ginv b* (*fps-ginv c a*) = *b oo a oo fps-inv c*
⟨*proof*⟩

lemma *fps-ginv-deriv*:
assumes *a0*:$a\$0 = (0::'a::field)$
and *a1*: $a\$1 \neq 0$
shows *fps-deriv* (*fps-ginv b a*) = (*fps-deriv b* / *fps-deriv a*) *oo fps-ginv X a*
⟨*proof*⟩

43.21 Elementary series

43.21.1 Exponential series

definition *E x* = *Abs-fps* (λ*n*. *x*^*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*⟩

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*⟩

lemma *E-add-mult*: *E* (*a* + *b*) = *E* (*a*::'*a*::*field-char-0*) ∗ *E b* (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *E-nth*[*simp*]: *E a* $ *n* = *a*^*n* / *of-nat* (*fact n*)
⟨*proof*⟩

lemma *E0*[*simp*]: *E* (*0*::'*a*::*field*) = *1*
⟨*proof*⟩

lemma *E-neg*: *E* (− *a*) = *inverse* (*E* (*a*::'*a*::*field-char-0*))
⟨*proof*⟩

lemma *E-nth-deriv*[*simp*]: *fps-nth-deriv n* (*E* (*a*::'*a*::*field-char-0*)) = (*fps-const a*)^*n* ∗ (*E a*)
⟨*proof*⟩

lemma *X-compose-E*[*simp*]: *X oo E* (*a*::'*a*::*field*) = *E a* − *1*
⟨*proof*⟩

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*⟩

lemma *E-power-mult*: (*E* (*c*::'*a*::*field-char-0*)) ^ *n* = *E* (*of-nat n* ∗ *c*)
⟨*proof*⟩

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*⟩

lemma *Ec-E1-eq*: *E* (*1*::'*a*::*field-char-0*) *oo* (*fps-const c* ∗ *X*) = *E c*
⟨*proof*⟩

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*))
⟨*proof*⟩

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*) ^ (*n* − *1*) / *of-nat n*)

lemma *fps-deriv-L*: *fps-deriv* (*L c*) = *fps-const* (*1*/*c*) ∗ *inverse* (*1* + *X*)
⟨*proof*⟩

lemma *L-nth*: *L c* $ *n* = (*if n* = *0 then 0 else 1*/*c* ∗ ((− *1*) ^ (*n* − *1*) / *of-nat n*))
⟨*proof*⟩

lemma *L-0*[*simp*]: *L c* $ *0* = *0* ⟨*proof*⟩

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*⟩

lemma *L-mult-add*:
assumes *c0*: $c \neq 0$
and *d0*: $d \neq 0$
shows *L c* + *L d* = *fps-const* (*c*+*d*) ∗ *L* (*c*∗*d*)
(**is** *?r* = *?l*)
⟨*proof*⟩

43.21.3 Binomial series

definition *fps-binomial a = Abs-fps (λn. a gchoose n)*

lemma *fps-binomial-nth*[*simp*]: *fps-binomial a $ n = a gchoose n*
⟨*proof*⟩

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*⟩

lemma *fps-binomial-deriv*: *fps-deriv (fps-binomial c) = fps-const c * fps-binomial c / (1 + X)*
⟨*proof*⟩

lemma *fps-binomial-add-mult*: *fps-binomial (c+d) = fps-binomial c * fps-binomial d* (**is** *?l = ?r*)
⟨*proof*⟩

lemma *fps-binomial-minus-one*: *fps-binomial (− 1) = inverse (1 + X)*
(**is** *?l = inverse ?r*)
⟨*proof*⟩

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*⟩

lemma *binomial-Vandermonde*:
*setsum (λk. (a choose k) * (b choose (n − k))) {0..n} = (a + b) choose n*
⟨*proof*⟩

lemma *binomial-Vandermonde-same*: *setsum (λk. (n choose k)²) {0..n} = (2 * n) choose n*
⟨*proof*⟩

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*⟩

lemma *Vandermonde-pochhammer*:
fixes *a* :: *'a*::*field-char-0*

assumes c: $\forall\, i \in \{0..< n\}.\ c \neq -\ of\text{-}nat\ i$
shows *setsum* $(\lambda k.\ (pochhammer\ a\ k * pochhammer\ (-\ (of\text{-}nat\ n))\ k)\ /$
$(of\text{-}nat\ (fact\ k) * pochhammer\ c\ k))\ \{0..n\} = pochhammer\ (c - a)\ n\ /\ pochhammer\ c\ n$
⟨*proof*⟩

43.21.4 Formal trigonometric functions

definition *fps-sin* $(c::'a::field\text{-}char\text{-}0) =$
$Abs\text{-}fps\ (\lambda n.\ if\ even\ n\ then\ 0\ else\ (-\ 1)\ \hat{}((n - 1)\ div\ 2) * c\hat{}n\ /(of\text{-}nat\ (fact\ n)))$

definition *fps-cos* $(c::'a::field\text{-}char\text{-}0) =$
$Abs\text{-}fps\ (\lambda n.\ if\ even\ n\ then\ (-\ 1)\ \hat{}\ (n\ div\ 2) * c\hat{}n\ /\ (of\text{-}nat\ (fact\ n))\ else\ 0)$

lemma *fps-sin-deriv*:
$fps\text{-}deriv\ (fps\text{-}sin\ c) = fps\text{-}const\ c * fps\text{-}cos\ c$
(**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *fps-cos-deriv*: $fps\text{-}deriv\ (fps\text{-}cos\ c) = fps\text{-}const\ (-\ c) * (fps\text{-}sin\ c)$
(**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *fps-sin-cos-sum-of-squares*: $(fps\text{-}cos\ c)^2 + (fps\text{-}sin\ c)^2 = 1$
(**is** *?lhs = -*)
⟨*proof*⟩

lemma *fps-sin-nth-0* [*simp*]: $fps\text{-}sin\ c\ \$\ 0 = 0$
⟨*proof*⟩

lemma *fps-sin-nth-1* [*simp*]: $fps\text{-}sin\ c\ \$\ 1 = c$
⟨*proof*⟩

lemma *fps-sin-nth-add-2*:
$fps\text{-}sin\ c\ \$\ (n + 2) = -\ (c * c * fps\text{-}sin\ c\ \$\ n\ /\ (of\text{-}nat\ (n + 1) * of\text{-}nat\ (n + 2)))$
⟨*proof*⟩

lemma *fps-cos-nth-0* [*simp*]: $fps\text{-}cos\ c\ \$\ 0 = 1$
⟨*proof*⟩

lemma *fps-cos-nth-1* [*simp*]: $fps\text{-}cos\ c\ \$\ 1 = 0$
⟨*proof*⟩

lemma *fps-cos-nth-add-2*:
$fps\text{-}cos\ c\ \$\ (n + 2) = -\ (c * c * fps\text{-}cos\ c\ \$\ n\ /\ (of\text{-}nat\ (n + 1) * of\text{-}nat\ (n + 2)))$
⟨*proof*⟩

lemma *nat-induct2*: $P\ 0 \Longrightarrow P\ 1 \Longrightarrow (\bigwedge n.\ P\ n \Longrightarrow P\ (n + 2)) \Longrightarrow P\ (n::nat)$
⟨*proof*⟩

lemma *nat-add-1-add-1*: $(n::nat) + 1 + 1 = n + 2$
⟨*proof*⟩

lemma *eq-fps-sin*:
assumes *0*: $a\ \$\ 0 = 0$
and *1*: $a\ \$\ 1 = c$
and *2*: $fps\text{-}deriv\ (fps\text{-}deriv\ a) = -\ (fps\text{-}const\ c * fps\text{-}const\ c * a)$
shows $a = fps\text{-}sin\ c$
⟨*proof*⟩

lemma *eq-fps-cos*:
assumes *0*: $a\ \$\ 0 = 1$
and *1*: $a\ \$\ 1 = 0$
and *2*: $fps\text{-}deriv\ (fps\text{-}deriv\ a) = -\ (fps\text{-}const\ c * fps\text{-}const\ c * a)$
shows $a = fps\text{-}cos\ c$
⟨*proof*⟩

lemma *mult-nth-0* [*simp*]: $(a * b)\ \$\ 0 = a\ \$\ 0 * b\ \$\ 0$
⟨*proof*⟩

lemma *mult-nth-1* [*simp*]: $(a * b)\ \$\ 1 = a\ \$\ 0 * b\ \$\ 1 + a\ \$\ 1 * b\ \$\ 0$
⟨*proof*⟩

lemma *fps-sin-add*: $fps\text{-}sin\ (a + b) = fps\text{-}sin\ a * fps\text{-}cos\ b + fps\text{-}cos\ a * fps\text{-}sin\ b$
⟨*proof*⟩

lemma *fps-cos-add*: $fps\text{-}cos\ (a + b) = fps\text{-}cos\ a * fps\text{-}cos\ b - fps\text{-}sin\ a * fps\text{-}sin\ b$
⟨*proof*⟩

lemma *fps-sin-even*: $fps\text{-}sin\ (-\ c) = -\ fps\text{-}sin\ c$
⟨*proof*⟩

lemma *fps-cos-odd*: $fps\text{-}cos\ (-\ c) = fps\text{-}cos\ c$
⟨*proof*⟩

definition $fps\text{-}tan\ c = fps\text{-}sin\ c\ /\ fps\text{-}cos\ c$

lemma *fps-tan-deriv*: $fps\text{-}deriv\ (fps\text{-}tan\ c) = fps\text{-}const\ c\ /\ (fps\text{-}cos\ c)^2$
⟨*proof*⟩

Connection to E c over the complex numbers — Euler and de Moivre.

lemma *Eii-sin-cos*: $E\ (ii * c) = fps\text{-}cos\ c + fps\text{-}const\ ii * fps\text{-}sin\ c$
(**is** *?l = ?r*)
⟨*proof*⟩

lemma *E-minus-ii-sin-cos*: $E\ (-\ (ii * c)) = fps\text{-}cos\ c - fps\text{-}const\ ii * fps\text{-}sin\ c$
⟨*proof*⟩

lemma *fps-const-minus*: $fps\text{-}const\ (c::'a::group\text{-}add) - fps\text{-}const\ d = fps\text{-}const\ (c - d)$
⟨*proof*⟩

lemma *fps-numeral-fps-const*: $numeral\ i = fps\text{-}const\ (numeral\ i :: 'a::comm\text{-}ring\text{-}1)$
⟨*proof*⟩

lemma *fps-cos-Eii*: $fps\text{-}cos\ c = (E\ (ii * c) + E\ (-\ ii * c))\ /\ fps\text{-}const\ 2$
⟨*proof*⟩

lemma *fps-sin-Eii*: $fps\text{-}sin\ c = (E\ (ii * c) - E\ (-\ ii * c))\ /\ fps\text{-}const\ (2*ii)$
⟨*proof*⟩

lemma *fps-tan-Eii*:
$fps\text{-}tan\ c = (E\ (ii * c) - E\ (-\ ii * c))\ /\ (fps\text{-}const\ ii * (E\ (ii * c) + E\ (-\ ii * c)))$
⟨*proof*⟩

lemma *fps-demoivre*:
$(fps\text{-}cos\ a + fps\text{-}const\ ii * fps\text{-}sin\ a)\ \hat{}\ n =$
$fps\text{-}cos\ (of\text{-}nat\ n * a) + fps\text{-}const\ ii * fps\text{-}sin\ (of\text{-}nat\ n * a)$
⟨*proof*⟩

43.22 Hypergeometric series

definition $F\ as\ bs\ (c::'a::\{field\text{-}char\text{-}0,field\}) =$
$Abs\text{-}fps\ (\lambda n.\ (foldl\ (\lambda r\ a.\ r * pochhammer\ a\ n)\ 1\ as * c\ \hat{}\ n)\ /$
$(foldl\ (\lambda r\ b.\ r * pochhammer\ b\ n)\ 1\ bs * of\text{-}nat\ (fact\ n)))$

lemma *F-nth*[*simp*]: $F\ as\ bs\ c\ \$\ n =$
$(foldl\ (\lambda r\ a.\ r * pochhammer\ a\ n)\ 1\ as * c\ \hat{}\ n)\ /$
$(foldl\ (\lambda r\ b.\ r * pochhammer\ b\ n)\ 1\ bs * of\text{-}nat\ (fact\ n))$
⟨*proof*⟩

lemma *foldl-mult-start*:
fixes $v :: 'a::comm\text{-}ring\text{-}1$
shows $foldl\ (\lambda r\ x.\ r * f\ x)\ v\ as * x = foldl\ (\lambda r\ x.\ r * f\ x)\ (v * x)\ as$
⟨*proof*⟩

lemma *foldr-mult-foldl*:
fixes $v :: 'a::comm\text{-}ring\text{-}1$
shows $foldr\ (\lambda x\ r.\ r * f\ x)\ as\ v = foldl\ (\lambda r\ x.\ r * f\ x)\ v\ as$
⟨*proof*⟩

lemma *F-nth-alt*:
$F\ as\ bs\ c\ \$\ n = foldr\ (\lambda a\ r.\ r * pochhammer\ a\ n)\ as\ (c\ \hat{}\ n)\ /$

foldr (λ*b r*. *r* ∗ *pochhammer b n*) *bs* (*of-nat* (*fact n*))
⟨*proof*⟩

lemma *F-E*[*simp*]: *F* [] [] *c* = *E c*
⟨*proof*⟩

lemma *F-1-0*[*simp*]: *F* [*1*] [] *c* = *1*/(*1* − *fps-const c* ∗ *X*)
⟨*proof*⟩

lemma *F-B*[*simp*]: *F* [−*a*] [] (− *1*) = *fps-binomial a*
⟨*proof*⟩

lemma *F-0*[*simp*]: *F as bs c* $ *0* = *1*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *XD-nth*[*simp*]: *XD a* $ *n* = (*if n* = *0 then 0 else of-nat n* ∗ *a*$*n*)
⟨*proof*⟩

lemma *XD-0th*[*simp*]: *XD a* $ *0* = *0*
⟨*proof*⟩
lemma *XD-Suc*[*simp*]: *XD a* $ *Suc n* = *of-nat* (*Suc n*) ∗ *a* $ *Suc n*
⟨*proof*⟩

definition *XDp c a* = *XD a* + *fps-const c* ∗ *a*

lemma *XDp-nth*[*simp*]: *XDp c a* $ *n* = (*c* + *of-nat n*) ∗ *a*$*n*
⟨*proof*⟩

lemma *XDp-commute*: *XDp b* ∘ *XDp* (*c*::′*a*::*comm-ring-1*) = *XDp c* ∘ *XDp b*
⟨*proof*⟩

lemma *XDp0* [*simp*]: *XDp 0* = *XD*
⟨*proof*⟩

lemma *XDp-fps-integral* [*simp*]: *XDp 0* (*fps-integral a c*) = *X* ∗ *a*
⟨*proof*⟩

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* ^ *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* ^ *k* / (*pochhammer* (− *of-nat* (*m* + *n*)) *k* ∗ *of-nat* (*fact k*))
else 0)
⟨*proof*⟩

lemma *setsum-eq-if*: *setsum f* {(*n*::*nat*) .. *m*} = (*if m* < *n then 0 else f n* + *setsum f* {*n*+*1* .. *m*})
⟨*proof*⟩

lemma *pochhammer-rec-if*: *pochhammer a n* = (*if n* = *0 then 1 else a* ∗ *pochhammer* (*a* + *1*) (*n* − *1*))
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *dist-less-imp-nth-equal*:
assumes *dist f g* < *inverse* (*2* ^ *i*)
and*j* ≤ *i*
shows *f* $ *j* = *g* $ *j*
⟨*proof*⟩

lemma *nth-equal-imp-dist-less*:
assumes ⋀*j*. *j* ≤ *i* ⟹ *f* $ *j* = *g* $ *j*
shows *dist f g* < *inverse* (*2* ^ *i*)
⟨*proof*⟩

lemma *dist-less-eq-nth-equal*: *dist f g* < *inverse* (*2* ^ *i*) ⟷ (∀ *j* ≤ *i*. *f* $ *j* = *g* $ *j*)
⟨*proof*⟩

instance *fps* :: (*comm-ring-1*) *complete-space*
⟨*proof*⟩

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*]:
$fractrel\ x\ y \longleftrightarrow snd\ x \neq 0 \wedge snd\ y \neq 0 \wedge fst\ x * snd\ y = fst\ y * snd\ x$
⟨*proof*⟩

lemma *symp-fractrel*: *symp fractrel*
⟨*proof*⟩

lemma *transp-fractrel*: *transp fractrel*
⟨*proof*⟩

lemma *part-equivp-fractrel*: *part-equivp fractrel*
⟨*proof*⟩

end

quotient-type (**overloaded**) $'a\ fract = 'a :: idom \times 'a$ / *partial*: *fractrel*
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *Fract-cases* [*cases type*: *fract*]:
obtains (*Fract*) $a\ b$ **where** $q = Fract\ a\ b\ b \neq 0$
⟨*proof*⟩

lemma *Fract-induct* [*case-names Fract*, *induct type*: *fract*]:

$(\bigwedge a\ b.\ b \neq 0 \Longrightarrow P\ (Fract\ a\ b)) \Longrightarrow P\ q$
⟨*proof*⟩

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 $\bigwedge a\ c.\ Fract\ 0\ a = Fract\ 0\ c$
⟨*proof*⟩

instantiation *fract* :: (*idom*) {*comm-ring-1*,*power*}
begin

lift-definition *zero-fract* :: 'a *fract* **is** (*0*, *1*) ⟨*proof*⟩

lemma *Zero-fract-def*: *0* = *Fract 0 1*
⟨*proof*⟩

lift-definition *one-fract* :: 'a *fract* **is** (*1*, *1*) ⟨*proof*⟩

lemma *One-fract-def*: *1* = *Fract 1 1*
⟨*proof*⟩

lift-definition *plus-fract* :: 'a *fract* ⇒ 'a *fract* ⇒ 'a *fract*
is $\lambda q\ r.\ (fst\ q * snd\ r + fst\ r * snd\ q,\ snd\ q * snd\ r)$
⟨*proof*⟩

lemma *add-fract* [*simp*]:
$[\![\ b \neq 0;\ d \neq 0\ ]\!] \Longrightarrow Fract\ a\ b + Fract\ c\ d = Fract\ (a * d + c * b)\ (b * d)$
⟨*proof*⟩

lift-definition *uminus-fract* :: 'a *fract* ⇒ 'a *fract*
is $\lambda x.\ (-\ fst\ x,\ snd\ x)$
⟨*proof*⟩

lemma *minus-fract* [*simp*]:
fixes *a b* :: 'a::*idom*
shows $-\ Fract\ a\ b = Fract\ (-\ a)\ b$
⟨*proof*⟩

lemma *minus-fract-cancel* [*simp*]: $Fract\ (-\ a)\ (-\ b) = Fract\ a\ b$
⟨*proof*⟩

definition *diff-fract-def*: $q - r = q + -\ (r::'a\ fract)$

lemma *diff-fract* [*simp*]:
$[\![\ b \neq 0;\ d \neq 0\ ]\!] \Longrightarrow Fract\ a\ b - Fract\ c\ d = Fract\ (a * d - c * b)\ (b * d)$
⟨*proof*⟩

lift-definition *times-fract* :: 'a *fract* ⇒ 'a *fract* ⇒ 'a *fract*

is $\lambda q\ r.\ (fst\ q * fst\ r,\ snd\ q * snd\ r)$
⟨*proof*⟩

lemma *mult-fract* [*simp*]: $Fract\ (a::'a::idom)\ b * Fract\ c\ d = Fract\ (a * c)\ (b * d)$
⟨*proof*⟩

lemma *mult-fract-cancel*:
$c \neq 0 \Longrightarrow Fract\ (c * a)\ (c * b) = Fract\ a\ b$
⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *of-nat-fract*: $of\text{-}nat\ k = Fract\ (of\text{-}nat\ k)\ 1$
⟨*proof*⟩

lemma *Fract-of-nat-eq*: $Fract\ (of\text{-}nat\ k)\ 1 = of\text{-}nat\ k$
⟨*proof*⟩

lemma *fract-collapse*:
$Fract\ 0\ k = 0$
$Fract\ 1\ 1 = 1$
$Fract\ k\ 0 = 0$
⟨*proof*⟩

lemma *fract-expand*:
$0 = Fract\ 0\ 1$
$1 = Fract\ 1\ 1$
⟨*proof*⟩

lemma *Fract-cases-nonzero*:
obtains $(Fract)\ a\ b$ **where** $q = Fract\ a\ b$ **and** $b \neq 0$ **and** $a \neq 0$
$\mid (0)\ q = 0$
⟨*proof*⟩

44.1.3 The field of rational numbers

context *idom*
begin

subclass *ring-no-zero-divisors* ⟨*proof*⟩

end

instantiation *fract* :: (*idom*) *field*
begin

lift-definition *inverse-fract* :: *'a fract ⇒ 'a fract*
is *λx. if fst x = 0 then (0, 1) else (snd x, fst x)*
⟨*proof*⟩

lemma *inverse-fract* [*simp*]: *inverse (Fract a b) = Fract (b::'a::idom) a*
⟨*proof*⟩

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)*
⟨*proof*⟩

instance
⟨*proof*⟩

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 ≠ 0 b' ≠ 0 d ≠ 0 d' ≠ 0*
assumes *eq1*: *a * b' = a' * b*
assumes *eq2*: *c * d' = c' * d*
shows *((a * d) * (b * d) ≤ (c * b) * (b * d)) ⟷ ((a' * d') * (b' * d') ≤ (c' * b') * (b' * d'))*
⟨*proof*⟩

lift-definition *less-eq-fract* :: *'a fract ⇒ 'a fract ⇒ bool*
is *λq r. (fst q * snd r) * (snd q * snd r) ≤ (fst r * snd q) * (snd q * snd r)*
⟨*proof*⟩

definition *less-fract-def*: *z < (w::'a fract) ⟷ z ≤ w ∧ ¬ w ≤ z*

lemma *le-fract* [*simp*]:
*⟦ b ≠ 0; d ≠ 0 ⟧ ⟹ Fract a b ≤ Fract c d ⟷ (a * d) * (b * d) ≤ (c * b) * (b * d)*
⟨*proof*⟩

lemma *less-fract* [*simp*]:
*⟦ b ≠ 0; d ≠ 0 ⟧ ⟹ Fract a b < Fract c d ⟷ (a * d) * (b * d) < (c * b) * (b * d)*
⟨*proof*⟩

instance

⟨*proof*⟩

end

instantiation *fract* :: (*linordered-idom*) {*distrib-lattice*,*abs-if*,*sgn-if*}
begin

definition *abs-fract-def2*: $|q| = (\mathit{if}\ q < 0\ \mathit{then}\ -q\ \mathit{else}\ (q::'a\ \mathit{fract}))$

definition *sgn-fract-def*:
$\mathit{sgn}\ (q::'a\ \mathit{fract}) = (\mathit{if}\ q = 0\ \mathit{then}\ 0\ \mathit{else}\ \mathit{if}\ 0 < q\ \mathit{then}\ 1\ \mathit{else}\ -1)$

theorem *abs-fract* [*simp*]: $|\mathit{Fract}\ a\ b| = \mathit{Fract}\ |a|\ |b|$
⟨*proof*⟩

definition *inf-fract-def*:
$(\mathit{inf} :: 'a\ \mathit{fract} \Rightarrow 'a\ \mathit{fract} \Rightarrow 'a\ \mathit{fract}) = \mathit{min}$

definition *sup-fract-def*:
$(\mathit{sup} :: 'a\ \mathit{fract} \Rightarrow 'a\ \mathit{fract} \Rightarrow 'a\ \mathit{fract}) = \mathit{max}$

instance
⟨*proof*⟩

end

instance *fract* :: (*linordered-idom*) *linordered-field*
⟨*proof*⟩

lemma *fract-induct-pos* [*case-names Fract*]:
fixes $P :: 'a::\mathit{linordered\text{-}idom}\ \mathit{fract} \Rightarrow \mathit{bool}$
assumes *step*: $\bigwedge a\ b.\ 0 < b \Longrightarrow P\ (\mathit{Fract}\ a\ b)$
shows $P\ q$
⟨*proof*⟩

lemma *zero-less-Fract-iff*: $0 < b \Longrightarrow 0 < \mathit{Fract}\ a\ b \longleftrightarrow 0 < a$
⟨*proof*⟩

lemma *Fract-less-zero-iff*: $0 < b \Longrightarrow \mathit{Fract}\ a\ b < 0 \longleftrightarrow a < 0$
⟨*proof*⟩

lemma *zero-le-Fract-iff*: $0 < b \Longrightarrow 0 \leq \mathit{Fract}\ a\ b \longleftrightarrow 0 \leq a$
⟨*proof*⟩

lemma *Fract-le-zero-iff*: $0 < b \Longrightarrow \mathit{Fract}\ a\ b \leq 0 \longleftrightarrow a \leq 0$
⟨*proof*⟩

lemma *one-less-Fract-iff*: $0 < b \Longrightarrow 1 < \mathit{Fract}\ a\ b \longleftrightarrow b < a$
⟨*proof*⟩

lemma *Fract-less-one-iff*: $0 < b \Longrightarrow Fract\ a\ b < 1 \longleftrightarrow a < b$
⟨*proof*⟩

lemma *one-le-Fract-iff*: $0 < b \Longrightarrow 1 \leq Fract\ a\ b \longleftrightarrow b \leq a$
⟨*proof*⟩

lemma *Fract-le-one-iff*: $0 < b \Longrightarrow Fract\ a\ b \leq 1 \longleftrightarrow a \leq b$
⟨*proof*⟩

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*
⟨*proof*⟩

setup-lifting *type-definition-fset*

45.2 Basic operations and type class instantiations

instantiation *fset* :: (*finite*) *finite*
begin
instance ⟨*proof*⟩
end

instantiation *fset* :: (*type*) {*bounded-lattice-bot*, *distrib-lattice*, *minus*}
begin

interpretation *lifting-syntax* ⟨*proof*⟩

lift-definition *bot-fset* :: *'a fset* **is** {} **parametric** *empty-transfer* ⟨*proof*⟩

lift-definition *less-eq-fset* :: *'a fset* ⇒ *'a fset* ⇒ *bool* **is** *subset-eq* **parametric** *subset-transfer*
⟨*proof*⟩

definition *less-fset* :: *'a fset* ⇒ *'a fset* ⇒ *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* <

⟨*proof*⟩

lift-definition *sup-fset* :: *'a fset* ⇒ *'a fset* ⇒ *'a fset* **is** *union* **parametric** *union-transfer*
⟨*proof*⟩

lift-definition *inf-fset* :: *'a fset* ⇒ *'a fset* ⇒ *'a fset* **is** *inter* **parametric** *inter-transfer*
⟨*proof*⟩

lift-definition *minus-fset* :: *'a fset* ⇒ *'a fset* ⇒ *'a fset* **is** *minus* **parametric** *Diff-transfer*
⟨*proof*⟩

instance
⟨*proof*⟩

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 ⟨*proof*⟩
end

instantiation *fset* :: (*type*) *conditionally-complete-lattice*
begin

interpretation *lifting-syntax* ⟨*proof*⟩

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* {})
⟨*proof*⟩

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* {})
⟨*proof*⟩

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* ⟨*proof*⟩

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* {}) ⟨*proof*⟩

lift-definition *Sup-fset* :: *'a fset set* ⇒ *'a fset* **is** λ*A*. *if finite* (*Sup A*) *then Sup A else* {}
parametric *Sup-fset-transfer* ⟨*proof*⟩

lemma *finite-Sup*: ∃ *z*. *finite z* ∧ (∀ *a*. *a* ∈ *X* ⟶ *a* ≤ *z*) ⟹ *finite* (*Sup X*)
⟨*proof*⟩

lemma *transfer-bdd-below*[*transfer-rule*]: (*rel-set* (*pcr-fset op* =) ===> *op* =) *bdd-below bdd-below*
⟨*proof*⟩

instance
⟨*proof*⟩
end

instantiation *fset* :: (*finite*) *complete-lattice*
begin

lift-definition *top-fset* :: *'a fset* **is** *UNIV* **parametric** *right-total-UNIV-transfer UNIV-transfer*
⟨*proof*⟩

instance
⟨*proof*⟩

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* ⟨*proof*⟩

instance
⟨*proof*⟩

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*
⟨*proof*⟩

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* ⟨*proof*⟩

abbreviation *notin-fset* :: *'a* ⇒ *'a fset* ⇒ *bool* (**infix** |∉| *50*) **where** *x* |∉| *S* ≡ ¬ (*x* |∈| *S*)

context
begin

interpretation *lifting-syntax* ⟨*proof*⟩

lift-definition *ffilter* :: (*'a* ⇒ *bool*) ⇒ *'a fset* ⇒ *'a fset* **is** *Set.filter*
parametric *Lifting-Set.filter-transfer* ⟨*proof*⟩

lift-definition *fPow* :: *'a fset* ⇒ *'a fset fset* **is** *Pow* **parametric** *Pow-transfer*
⟨*proof*⟩

lift-definition *fcard* :: *'a fset* ⇒ *nat* **is** *card* **parametric** *card-transfer* ⟨*proof*⟩

lift-definition *fimage* :: (*'a* ⇒ *'b*) ⇒ *'a fset* ⇒ *'b fset* (**infixr** |‘| *90*) **is** *image*
parametric *image-transfer* ⟨*proof*⟩

lift-definition *fthe-elem* :: *'a fset* ⇒ *'a* **is** *the-elem* ⟨*proof*⟩

lift-definition *fbind* :: *'a fset* ⇒ (*'a* ⇒ *'b fset*) ⇒ *'b fset* **is** *Set.bind* **parametric**

bind-transfer
⟨*proof*⟩

lift-definition *ffUnion* :: *'a fset fset* ⇒ *'a fset* **is** *Union* **parametric** *Union-transfer*
⟨*proof*⟩

lift-definition *fBall* :: *'a fset* ⇒ (*'a* ⇒ *bool*) ⇒ *bool* **is** *Ball* **parametric** *Ball-transfer*
⟨*proof*⟩
lift-definition *fBex* :: *'a fset* ⇒ (*'a* ⇒ *bool*) ⇒ *bool* **is** *Bex* **parametric** *Bex-transfer*
⟨*proof*⟩

lift-definition *ffold* :: (*'a* ⇒ *'b* ⇒ *'b*) ⇒ *'b* ⇒ *'a fset* ⇒ *'b* **is** *Finite-Set.fold*
⟨*proof*⟩

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* = {||}
⟨*proof*⟩

lemma *femptyE* [*elim*!]: *a* |∈| {||} ⟹ *P*
⟨*proof*⟩

45.5.3 *fset*

lemmas *fset-simps*[*simp*] = *bot-fset.rep-eq finsert.rep-eq*

lemma *finite-fset* [*simp*]:
shows *finite* (*fset S*)
⟨*proof*⟩

lemmas *fset-cong* = *fset-inject*

lemma *filter-fset* [*simp*]:
shows *fset* (*ffilter P xs*) = *Collect P* ∩ *fset xs*
⟨*proof*⟩

lemma *notin-fset*: *x* |∉| *S* ⟷ *x* ∉ *fset S* ⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *eq-ffilter*:
(*ffilter P A* = *ffilter Q A*) = (∀ *x*. *x* |∈| *A* ⟶ *P x* = *Q x*)
⟨*proof*⟩

lemma *pfsubset-ffilter*:
(⋀*x*. *x* |∈| *A* ⟹ *P x* ⟹ *Q x*) ⟹ (*x* |∈| *A* & ¬ *P x* & *Q x*) ⟹
ffilter P A |⊂| *ffilter Q A*
⟨*proof*⟩

45.5.5 *finsert*

lemma *set-finsert*:
assumes *x* |∈| *A*
obtains *B* **where** *A* = *finsert x B* **and** *x* |∉| *B*
⟨*proof*⟩

lemma *mk-disjoint-finsert*: *a* |∈| *A* ⟹ ∃ *B*. *A* = *finsert a B* ∧ *a* |∉| *B*
⟨*proof*⟩

45.5.6 *fimage*

lemma *subset-fimage-iff*: (*B* |⊆| *f*|‘|*A*) = (∃ *AA*. *AA* |⊆| *A* ∧ *B* = *f*|‘|*AA*)
⟨*proof*⟩

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*))
⋀*A P*. (¬ *fBex A P*) = *fBall A* (λ*x*. ¬ *P x*)
⟨*proof*⟩

lemma *ball-simps* [*simp*, *no-atp*]:
⋀*A P Q*. *fBall A* (λ*x*. *P x* ∨ *Q*) = (*fBall A P* ∨ *Q*)
⋀*A P Q*. *fBall A* (λ*x*. *P* ∨ *Q x*) = (*P* ∨ *fBall A Q*)
⋀*A P Q*. *fBall A* (λ*x*. *P* ⟶ *Q x*) = (*P* ⟶ *fBall A Q*)
⋀*A P Q*. *fBall A* (λ*x*. *P x* ⟶ *Q*) = (*fBex A P* ⟶ *Q*)
⋀*P*. *fBall* {||} *P* = *True*
⋀*a B P*. *fBall* (*finsert a B*) *P* = (*P a* ∧ *fBall B P*)
⋀*A P f*. *fBall* (*f* |‘| *A*) *P* = *fBall A* (λ*x*. *P* (*f x*))
⋀*A P*. (¬ *fBall A P*) = *fBex A* (λ*x*. ¬ *P x*)
⟨*proof*⟩

lemma *atomize-fBall*:
(⋀*x*. *x* |∈| *A* ==> *P x*) == *Trueprop* (*fBall A* (λ*x*. *P x*))
⟨*proof*⟩

end

45.5.8 *fcard*

lemma *fcard-fempty*:
fcard {||} = *0*
⟨*proof*⟩

lemma *fcard-finsert-disjoint*:
x |∉| *A* ⟹ *fcard* (*finsert x A*) = *Suc* (*fcard A*)
⟨*proof*⟩

lemma *fcard-finsert-if*:
fcard (*finsert x A*) = (*if x* |∈| *A then fcard A else Suc* (*fcard A*))
⟨*proof*⟩

lemma *card-0-eq* [*simp*, *no-atp*]:
fcard A = *0* ⟷ *A* = {||}
⟨*proof*⟩

lemma *fcard-Suc-fminus1*:
x |∈| *A* ⟹ *Suc* (*fcard* (*A* |−| {|*x*|})) = *fcard A*
⟨*proof*⟩

lemma *fcard-fminus-fsingleton*:
x |∈| *A* ⟹ *fcard* (*A* |−| {|*x*|}) = *fcard A* − *1*
⟨*proof*⟩

lemma *fcard-fminus-fsingleton-if*:
fcard (*A* |−| {|*x*|}) = (*if x* |∈| *A then fcard A* − *1 else fcard A*)
⟨*proof*⟩

lemma *fcard-fminus-finsert*[*simp*]:
assumes *a* |∈| *A* **and** *a* |∉| *B*
shows *fcard* (*A* |−| *finsert a B*) = *fcard* (*A* |−| *B*) − *1*
⟨*proof*⟩

lemma *fcard-finsert*: *fcard* (*finsert x A*) = *Suc* (*fcard* (*A* |−| {|*x*|}))
⟨*proof*⟩

lemma *fcard-finsert-le*: *fcard A* ≤ *fcard* (*finsert x A*)
⟨*proof*⟩

lemma *fcard-mono*:
A |⊆| *B* ⟹ *fcard A* ≤ *fcard B*
⟨*proof*⟩

lemma *fcard-seteq*: *A* |⊆| *B* ⟹ *fcard B* ≤ *fcard A* ⟹ *A* = *B*

⟨*proof*⟩

lemma *pfsubset-fcard-mono*: $A \mid\subset\mid B \Longrightarrow fcard\ A < fcard\ B$
⟨*proof*⟩

lemma *fcard-funion-finter*:
$fcard\ A + fcard\ B = fcard\ (A \mid\cup\mid B) + fcard\ (A \mid\cap\mid B)$
⟨*proof*⟩

lemma *fcard-funion-disjoint*:
$A \mid\cap\mid B = \{||\} \Longrightarrow fcard\ (A \mid\cup\mid B) = fcard\ A + fcard\ B$
⟨*proof*⟩

lemma *fcard-funion-fsubset*:
$B \mid\subseteq\mid A \Longrightarrow fcard\ (A \mid-\mid B) = fcard\ A - fcard\ B$
⟨*proof*⟩

lemma *diff-fcard-le-fcard-fminus*:
$fcard\ A - fcard\ B \leq fcard(A \mid-\mid B)$
⟨*proof*⟩

lemma *fcard-fminus1-less*: $x \mid\in\mid A \Longrightarrow fcard\ (A \mid-\mid \{|x|\}) < fcard\ A$
⟨*proof*⟩

lemma *fcard-fminus2-less*:
$x \mid\in\mid A \Longrightarrow y \mid\in\mid A \Longrightarrow fcard\ (A \mid-\mid \{|x|\} \mid-\mid \{|y|\}) < fcard\ A$
⟨*proof*⟩

lemma *fcard-fminus1-le*: $fcard\ (A \mid-\mid \{|x|\}) \leq fcard\ A$
⟨*proof*⟩

lemma *fcard-pfsubset*: $A \mid\subseteq\mid B \Longrightarrow fcard\ A < fcard\ B \Longrightarrow A < B$
⟨*proof*⟩

45.5.9 *ffold*

context *comp-fun-commute*
begin
lemmas *ffold-empty*[*simp*] = *fold-empty*[*Transfer.transferred*]

lemma *ffold-finsert* [*simp*]:
assumes $x \mid\notin\mid A$
shows $ffold\ f\ z\ (finsert\ x\ A) = f\ x\ (ffold\ f\ z\ A)$
⟨*proof*⟩

lemma *ffold-fun-left-comm*:
$f\ x\ (ffold\ f\ z\ A) = ffold\ f\ (f\ x\ z)\ A$
⟨*proof*⟩

lemma *ffold-finsert2*:
x |∉| *A* ⟹ *ffold f z* (*finsert x A*) = *ffold f* (*f x z*) *A*
⟨*proof*⟩

lemma *ffold-rec*:
assumes *x* |∈| *A*
shows *ffold f z A* = *f x* (*ffold f z* (*A* |−| {|*x*|}))
⟨*proof*⟩

lemma *ffold-finsert-fremove*:
ffold f z (*finsert x A*) = *f x* (*ffold f z* (*A* |−| {|*x*|}))
⟨*proof*⟩
end

lemma *ffold-fimage*:
assumes *inj-on g* (*fset A*)
shows *ffold f z* (*g* |‘| *A*) = *ffold* (*f* ∘ *g*) *z A*
⟨*proof*⟩

lemma *ffold-cong*:
assumes *comp-fun-commute f comp-fun-commute g*
⋀*x*. *x* |∈| *A* ⟹ *f x* = *g x*
and *s* = *t* **and** *A* = *B*
shows *ffold f s A* = *ffold g t B*
⟨*proof*⟩

context *comp-fun-idem*
begin

lemma *ffold-finsert-idem*:
ffold f z (*finsert x A*) = *f x* (*ffold f z A*)
⟨*proof*⟩

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*
⟨*proof*⟩

end

45.6 Choice in fsets

lemma *fset-choice*:
assumes ∀*x*. *x* |∈| *A* ⟶ (∃*y*. *P x y*)
shows ∃*f*. ∀*x*. *x* |∈| *A* ⟶ *P x* (*f x*)
⟨*proof*⟩

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
⟨*proof*⟩

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*⟩

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*⟩

lemma *fset-card-induct*:
assumes *empty-fset-case*: $P\ \{||\}$
and *card-fset-Suc-case*: $\bigwedge S\ T.\ \mathit{Suc}\ (\mathit{fcard}\ S) = (\mathit{fcard}\ T) \Longrightarrow P\ S \Longrightarrow P\ T$
shows $P\ S$
⟨*proof*⟩

lemma *fset-strong-cases*:
obtains $xs = \{||\}$
| $ys\ x$ **where** $x\ |\notin|\ ys$ **and** $xs = \mathit{finsert}\ x\ ys$
⟨*proof*⟩

lemma *fset-induct2*:
$P\ \{||\}\ \{||\} \Longrightarrow$
$(\bigwedge x\ xs.\ x\ |\notin|\ xs \Longrightarrow P\ (\mathit{finsert}\ x\ xs)\ \{||\}) \Longrightarrow$
$(\bigwedge y\ ys.\ y\ |\notin|\ ys \Longrightarrow P\ \{||\}\ (\mathit{finsert}\ y\ ys)) \Longrightarrow$
$(\bigwedge x\ xs\ y\ ys.\ [\![P\ xs\ ys;\ x\ |\notin|\ xs;\ y\ |\notin|\ ys]\!] \Longrightarrow P\ (\mathit{finsert}\ x\ xs)\ (\mathit{finsert}\ y\ ys)) \Longrightarrow$
$P\ xsa\ ysa$
⟨*proof*⟩

45.8 Setup for Lifting/Transfer

45.8.1 Relator and predicator properties

lift-definition *rel-fset* :: $('a \Rightarrow 'b \Rightarrow \mathit{bool}) \Rightarrow 'a\ \mathit{fset} \Rightarrow 'b\ \mathit{fset} \Rightarrow \mathit{bool}$ **is** *rel-set*
parametric *rel-set-transfer* ⟨*proof*⟩

lemma *rel-fset-alt-def*: *rel-fset* $R = (\lambda A\ B.\ (\forall x.\exists y.\ x|\in|A \longrightarrow y|\in|B \wedge R\ x\ y)$
$\wedge\ (\forall y.\ \exists x.\ y|\in|B \longrightarrow x|\in|A \wedge R\ x\ y))$
⟨*proof*⟩

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*⟩

45.8.2 Transfer rules for the Transfer package

Unconditional transfer rules

context
begin

interpretation *lifting-syntax* ⟨*proof*⟩

lemmas *fempty-transfer* [*transfer-rule*] = *empty-transfer*[*Transfer.transferred*]

lemma *finsert-transfer* [*transfer-rule*]:
(*A* ===> *rel-fset A* ===> *rel-fset A*) *finsert finsert*
⟨*proof*⟩

lemma *funion-transfer* [*transfer-rule*]:
(*rel-fset A* ===> *rel-fset A* ===> *rel-fset A*) *funion funion*
⟨*proof*⟩

lemma *ffUnion-transfer* [*transfer-rule*]:
(*rel-fset* (*rel-fset A*) ===> *rel-fset A*) *ffUnion ffUnion*
⟨*proof*⟩

lemma *fimage-transfer* [*transfer-rule*]:
((*A* ===> *B*) ===> *rel-fset A* ===> *rel-fset B*) *fimage fimage*
⟨*proof*⟩

lemma *fBall-transfer* [*transfer-rule*]:
(*rel-fset A* ===> (*A* ===> *op* =) ===> *op* =) *fBall fBall*
⟨*proof*⟩

lemma *fBex-transfer* [*transfer-rule*]:
(*rel-fset A* ===> (*A* ===> *op* =) ===> *op* =) *fBex fBex*
⟨*proof*⟩

lemma *fPow-transfer* [*transfer-rule*]:
(*rel-fset A* ===> *rel-fset* (*rel-fset A*)) *fPow fPow*
⟨*proof*⟩

lemma *rel-fset-transfer* [*transfer-rule*]:
((*A* ===> *B* ===> *op* =) ===> *rel-fset A* ===> *rel-fset B* ===> *op* =)
rel-fset rel-fset
⟨*proof*⟩

lemma *bind-transfer* [*transfer-rule*]:
(*rel-fset A* ===> (*A* ===> *rel-fset B*) ===> *rel-fset B*) *fbind fbind*
⟨*proof*⟩

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* |∈|)
⟨*proof*⟩

lemma *finter-transfer* [*transfer-rule*]:
assumes *bi-unique A*
shows (*rel-fset A* ===> *rel-fset A* ===> *rel-fset A*) *finter finter*
⟨*proof*⟩

lemma *fminus-transfer* [*transfer-rule*]:
assumes *bi-unique A*
shows (*rel-fset A* ===> *rel-fset A* ===> *rel-fset A*) (*op* |−|) (*op* |−|)
⟨*proof*⟩

lemma *fsubset-transfer* [*transfer-rule*]:
assumes *bi-unique A*
shows (*rel-fset A* ===> *rel-fset A* ===> *op* =) (*op* |⊆|) (*op* |⊆|)
⟨*proof*⟩

lemma *fSup-transfer* [*transfer-rule*]:
bi-unique A ⟹ (*rel-set* (*rel-fset A*) ===> *rel-fset A*) *Sup Sup*
⟨*proof*⟩

lemma *fInf-transfer* [*transfer-rule*]:
assumes *bi-unique A* **and** *bi-total A*
shows (*rel-set* (*rel-fset A*) ===> *rel-fset A*) *Inf Inf*
⟨*proof*⟩

lemma *ffilter-transfer* [*transfer-rule*]:
assumes *bi-unique A*
shows ((*A* ===> *op*=) ===> *rel-fset A* ===> *rel-fset A*) *ffilter ffilter*
⟨*proof*⟩

lemma *card-transfer* [*transfer-rule*]:
bi-unique A ⟹ (*rel-fset A* ===> *op* =) *fcard fcard*
⟨*proof*⟩

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 $\longleftrightarrow$ $(\forall\, t \in \mathit{fset}\ a.\ \exists\, u \in \mathit{fset}\ b.\ R\ t\ u) \wedge (\forall\, t \in \mathit{fset}\ b.\ \exists\, u \in \mathit{fset}\ a.\ R\ u\ t)$
⟨*proof*⟩

lemma *fset-to-fset*: *finite A* $\Longrightarrow$ *fset* (*the-inv fset A*) = *A*
⟨*proof*⟩

lemma *rel-fset-aux*:
$(\forall\, t \in \mathit{fset}\ a.\ \exists\, u \in \mathit{fset}\ b.\ R\ t\ u) \wedge (\forall\, u \in \mathit{fset}\ b.\ \exists\, t \in \mathit{fset}\ a.\ R\ t\ u) \longleftrightarrow$
$((\mathit{BNF\text{-}Def.Grp}\ \{a.\ \mathit{fset}\ a \subseteq \{(a,\ b).\ R\ a\ b\}\}\ (\mathit{fimage}\ \mathit{fst}))^{-1-1}\ \mathit{OO}$
BNF-Def.Grp $\{a.\ \mathit{fset}\ a \subseteq \{(a,\ b).\ R\ a\ b\}\}$ (*fimage snd*)) *a b* (**is** *?L* = *?R*)
⟨*proof*⟩

bnf *'a fset*
map: *fimage*
sets: *fset*
bd: *natLeq*
wits: {||}
rel: *rel-fset*
⟨*proof*⟩

lemma *rel-fset-fset*: *rel-set* χ (*fset A1*) (*fset A2*) = *rel-fset* χ *A1 A2*
⟨*proof*⟩

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** $\lambda f.$ *setsum* (*Suc* ∘ *f*)
⟨*proof*⟩
end

instantiation *fset* :: (*type*) *size* **begin**
definition *size-fset* **where**
size-fset-overloaded-def: *size-fset* = *FSet.size-fset* (λ-. *0*)
instance ⟨*proof*⟩
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*)
⟨*proof*⟩
including *fset.lifting* ⟨*proof*⟩

⟨*ML*⟩

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* ⟷
rel-set χ (*Inl* −‘ *A1*) (*Inl* −‘ *A2*) ∧ *rel-set* φ (*Inr* −‘ *A1*) (*Inr* −‘ *A2*)
(**is** *?L* ⟷ *?Rl* ∧ *?Rr*)
⟨*proof*⟩

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 ⟨*proof*⟩

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 (λA. f A orelse Quickcheck-Exhaustive.full-exhaustive (λx. if fst x |∈| fst A then None else f (valtermify-finsert x A)) (i − 1)) (i − 1)))

instance ⟨*proof*⟩

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 ∘→ (λx. random-aux-fset i j ∘→ (λs. 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 ∘→ (λx. random-aux-fset (i − 1) j ∘→ (λs. Pair (valtermify-finsert x s))))])
⟨*proof*⟩

definition *random-fset i = random-aux-fset i i*

instance ⟨*proof*⟩

end

no-notation *scomp* (**infixl** *∘→ 60*)

end

46 Pi and Function Sets

theory *FuncSet*
imports *Hilbert-Choice Main*
begin

definition $Pi :: 'a\ set \Rightarrow ('a \Rightarrow 'b\ set) \Rightarrow ('a \Rightarrow 'b)\ set$
  where $Pi\ A\ B = \{f.\ \forall x.\ x \in A \longrightarrow f\ x \in B\ x\}$

definition $extensional :: 'a\ set \Rightarrow ('a \Rightarrow 'b)\ set$
  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 :: $pttrn \Rightarrow 'a\ set \Rightarrow 'b\ set \Rightarrow ('a \Rightarrow 'b)\ set$ ((*3*Π *-∈-./ -*) *10*)
  -lam :: $pttrn \Rightarrow 'a\ set \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b)$ ((*3*λ*-∈-./ -*) [*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*!]: $(\bigwedge x.\ x \in A \Longrightarrow f\ x \in B\ x) \Longrightarrow f \in Pi\ A\ B$
  ⟨*proof*⟩

lemma *Pi-I'*[*simp*]: $(\bigwedge x.\ x \in A \longrightarrow f\ x \in B\ x) \Longrightarrow f \in Pi\ A\ B$
  ⟨*proof*⟩

lemma *funcsetI*: $(\bigwedge x.\ x \in A \Longrightarrow f\ x \in B) \Longrightarrow f \in A \rightarrow B$
  ⟨*proof*⟩

lemma *Pi-mem*: $f \in Pi\ A\ B \Longrightarrow x \in A \Longrightarrow f\ x \in B\ x$
  ⟨*proof*⟩

lemma *Pi-iff*: $f \in Pi\ I\ X \longleftrightarrow (\forall i{\in}I.\ f\ i \in X\ i)$
  ⟨*proof*⟩

lemma *PiE* [*elim*]: $f \in Pi\ A\ B \Longrightarrow (f\ x \in B\ x \Longrightarrow Q) \Longrightarrow (x \notin A \Longrightarrow Q) \Longrightarrow Q$
  ⟨*proof*⟩

lemma *Pi-cong*: $(\bigwedge w.\ w \in A \Longrightarrow f\ w = g\ w) \Longrightarrow f \in Pi\ A\ B \longleftrightarrow g \in Pi\ A\ B$
⟨*proof*⟩

lemma *funcset-id* [*simp*]: $(\lambda x.\ x) \in A \to A$
⟨*proof*⟩

lemma *funcset-mem*: $f \in A \to B \Longrightarrow x \in A \Longrightarrow f\ x \in B$
⟨*proof*⟩

lemma *funcset-image*: $f \in A \to B \Longrightarrow f\ `\ A \subseteq B$
⟨*proof*⟩

lemma *image-subset-iff-funcset*: $F\ `\ A \subseteq B \longleftrightarrow F \in A \to B$
⟨*proof*⟩

lemma *Pi-eq-empty*[*simp*]: $(\Pi\ x \in A.\ B\ x) = \{\} \longleftrightarrow (\exists\, x{\in}A.\ B\ x = \{\})$
⟨*proof*⟩

lemma *Pi-empty* [*simp*]: $Pi\ \{\}\ B = UNIV$
⟨*proof*⟩

lemma *Pi-Int*: $Pi\ I\ E \cap Pi\ I\ F = (\Pi\ i{\in}I.\ E\ i \cap F\ i)$
⟨*proof*⟩

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*⟩

lemma *Pi-UNIV* [*simp*]: $A \to UNIV = UNIV$
⟨*proof*⟩

Covariance of Pi-sets in their second argument

lemma *Pi-mono*: $(\bigwedge x.\ x \in A \Longrightarrow B\ x \subseteq C\ x) \Longrightarrow Pi\ A\ B \subseteq Pi\ A\ C$
⟨*proof*⟩

Contravariance of Pi-sets in their first argument

lemma *Pi-anti-mono*: $A' \subseteq A \Longrightarrow Pi\ A\ B \subseteq Pi\ A'\ B$
⟨*proof*⟩

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*⟩

lemma *Pi-split-domain*[*simp*]: $x \in Pi\ (I \cup J)\ X \longleftrightarrow x \in Pi\ I\ X \wedge x \in Pi\ J\ X$
⟨*proof*⟩

lemma *Pi-split-insert-domain*[*simp*]: $x \in Pi\ (insert\ i\ I)\ X \longleftrightarrow x \in Pi\ I\ X \wedge x\ i \in X\ i$
⟨*proof*⟩

lemma *Pi-cancel-fupd-range*[*simp*]: $i \notin I \Longrightarrow x \in Pi\ I\ (B(i := b)) \longleftrightarrow x \in Pi\ I\ B$
⟨*proof*⟩

lemma *Pi-cancel-fupd*[*simp*]: $i \notin I \Longrightarrow x(i := a) \in Pi\ I\ B \longleftrightarrow x \in Pi\ I\ B$
⟨*proof*⟩

lemma *Pi-fupd-iff*: $i \in I \Longrightarrow f \in Pi\ I\ (B(i := A)) \longleftrightarrow f \in Pi\ (I - \{i\})\ B \wedge f\ i \in A$
⟨*proof*⟩

46.2 Composition With a Restricted Domain: *compose*

lemma *funcset-compose*: $f \in A \to B \Longrightarrow g \in B \to C \Longrightarrow compose\ A\ g\ f \in A \to C$
⟨*proof*⟩

lemma *compose-assoc*:
assumes $f \in A \to B$
and $g \in B \to C$
and $h \in C \to D$
shows $compose\ A\ h\ (compose\ A\ g\ f) = compose\ A\ (compose\ B\ h\ g)\ f$
⟨*proof*⟩

lemma *compose-eq*: $x \in A \Longrightarrow compose\ A\ g\ f\ x = g\ (f\ x)$
⟨*proof*⟩

lemma *surj-compose*: $f\ `\ A = B \Longrightarrow g\ `\ B = C \Longrightarrow compose\ A\ g\ f\ `\ A = C$
⟨*proof*⟩

46.3 Bounded Abstraction: *restrict*

lemma *restrict-cong*: $I = J \Longrightarrow (\bigwedge i.\ i \in J =simp=> f\ i = g\ i) \Longrightarrow restrict\ f\ I = restrict\ g\ J$
⟨*proof*⟩

lemma *restrict-in-funcset*: $(\bigwedge x.\ x \in A \Longrightarrow f\ x \in B) \Longrightarrow (\lambda x{\in}A.\ f\ x) \in A \to B$
⟨*proof*⟩

lemma *restrictI*[*intro!*]: $(\bigwedge x.\ x \in A \Longrightarrow f\ x \in B\ x) \Longrightarrow (\lambda x{\in}A.\ f\ x) \in Pi\ A\ B$
⟨*proof*⟩

lemma *restrict-apply*[*simp*]: $(\lambda y{\in}A.\ f\ y)\ x = (if\ x \in A\ then\ f\ x\ else\ undefined)$

⟨*proof*⟩

lemma *restrict-apply'*: $x \in A \Longrightarrow (\lambda y{\in}A.\ f\ y)\ x = f\ x$
⟨*proof*⟩

lemma *restrict-ext*: $(\bigwedge x.\ x \in A \Longrightarrow f\ x = g\ x) \Longrightarrow (\lambda x{\in}A.\ f\ x) = (\lambda x{\in}A.\ g\ x)$
⟨*proof*⟩

lemma *restrict-UNIV*: $restrict\ f\ UNIV = f$
⟨*proof*⟩

lemma *inj-on-restrict-eq* [*simp*]: $inj\text{-}on\ (restrict\ f\ A)\ A = inj\text{-}on\ f\ A$
⟨*proof*⟩

lemma *Id-compose*: $f \in A \to B \Longrightarrow f \in extensional\ A \Longrightarrow compose\ A\ (\lambda y{\in}B.\ y)\ f = f$
⟨*proof*⟩

lemma *compose-Id*: $g \in A \to B \Longrightarrow g \in extensional\ A \Longrightarrow compose\ A\ g\ (\lambda x{\in}A.\ x) = g$
⟨*proof*⟩

lemma *image-restrict-eq* [*simp*]: $(restrict\ f\ A)\ `\ A = f\ `\ A$
⟨*proof*⟩

lemma *restrict-restrict*[*simp*]: $restrict\ (restrict\ f\ A)\ B = restrict\ f\ (A \cap B)$
⟨*proof*⟩

lemma *restrict-fupd*[*simp*]: $i \notin I \Longrightarrow restrict\ (f\ (i := x))\ I = restrict\ f\ I$
⟨*proof*⟩

lemma *restrict-upd*[*simp*]: $i \notin I \Longrightarrow (restrict\ f\ I)(i := y) = restrict\ (f(i := y))\ (insert\ i\ I)$
⟨*proof*⟩

lemma *restrict-Pi-cancel*: $restrict\ x\ I \in Pi\ I\ A \longleftrightarrow x \in Pi\ I\ A$
⟨*proof*⟩

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 $bij\text{-}betw\ f\ A\ B$
⟨*proof*⟩

lemma *bij-betw-imp-funcset*: *bij-betw f A B* $\Longrightarrow$ $f \in A \to B$
⟨*proof*⟩

lemma *inj-on-compose*: *bij-betw f A B* $\Longrightarrow$ *inj-on g B* $\Longrightarrow$ *inj-on* (*compose A g f*) *A*
⟨*proof*⟩

lemma *bij-betw-compose*: *bij-betw f A B* $\Longrightarrow$ *bij-betw g B C* $\Longrightarrow$ *bij-betw* (*compose A g f*) *A C*
⟨*proof*⟩

lemma *bij-betw-restrict-eq* [*simp*]: *bij-betw* (*restrict f A*) *A B* = *bij-betw f A B*
⟨*proof*⟩

46.5 Extensionality

lemma *extensional-empty*[*simp*]: *extensional* {} = {$\lambda x.$ *undefined*}
⟨*proof*⟩

lemma *extensional-arb*: $f \in$ *extensional A* $\Longrightarrow$ $x \notin A$ $\Longrightarrow$ *f x* = *undefined*
⟨*proof*⟩

lemma *restrict-extensional* [*simp*]: *restrict f A* $\in$ *extensional A*
⟨*proof*⟩

lemma *compose-extensional* [*simp*]: *compose A f g* $\in$ *extensional A*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *extensional-restrict*: $f \in$ *extensional A* $\Longrightarrow$ *restrict f A* = *f*
⟨*proof*⟩

lemma *extensional-subset*: $f \in$ *extensional A* $\Longrightarrow$ $A \subseteq B$ $\Longrightarrow$ $f \in$ *extensional B*
⟨*proof*⟩

lemma *inv-into-funcset*: *f ' A* = *B* $\Longrightarrow$ ($\lambda x \in B.$ *inv-into A f x*) $\in B \to A$
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *compose-id-inv-into*: f ' $A = B \Longrightarrow$ *compose* B f $(\lambda y{\in}B.$ *inv-into* A f $y)$ $= (\lambda x{\in}B.\ x)$
⟨*proof*⟩

lemma *extensional-insert*[*intro*, *simp*]:
assumes $a \in$ *extensional* (*insert* i I)
shows $a(i := b) \in$ *extensional* (*insert* i I)
⟨*proof*⟩

lemma *extensional-Int*[*simp*]: *extensional* $I \cap$ *extensional* $I' =$ *extensional* $(I \cap I')$
⟨*proof*⟩

lemma *extensional-UNIV*[*simp*]: *extensional UNIV* $=$ *UNIV*
⟨*proof*⟩

lemma *restrict-extensional-sub*[*intro*]: $A \subseteq B \Longrightarrow$ *restrict* f $A \in$ *extensional* B
⟨*proof*⟩

lemma *extensional-insert-undefined*[*intro*, *simp*]:
$a \in$ *extensional* (*insert* i I) $\Longrightarrow$ $a(i :=$ *undefined*$) \in$ *extensional* I
⟨*proof*⟩

lemma *extensional-insert-cancel*[*intro*, *simp*]:
$a \in$ *extensional* $I \Longrightarrow a \in$ *extensional* (*insert* i I)
⟨*proof*⟩

46.6 Cardinality

lemma *card-inj*: $f \in A \to B \Longrightarrow$ *inj-on* f $A \Longrightarrow$ *finite* $B \Longrightarrow$ *card* $A \leq$ *card* B
⟨*proof*⟩

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
⟨*proof*⟩

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*)
-*PiE* :: *pttrn* $\Rightarrow$ $'a$ *set* $\Rightarrow$ $'b$ *set* $\Rightarrow$ $('a \Rightarrow 'b)$ *set* ((*3PIE* -:-./ -) *10*)
syntax
-*PiE* :: *pttrn* $\Rightarrow$ $'a$ *set* $\Rightarrow$ $'b$ *set* $\Rightarrow$ $('a \Rightarrow 'b)$ *set* ((*3*Π_E -∈-./ -) *10*)

translations
$\Pi_E\ x{\in}A.\ B \rightleftharpoons CONST\ Pi_E\ A\ (\lambda x.\ B)$

abbreviation *extensional-funcset* :: *'a set* ⇒ *'b set* ⇒ (*'a* ⇒ *'b*) *set* (**infixr** $\rightarrow_E$ *60*)
where $A \rightarrow_E B \equiv (\Pi_E\ i{\in}A.\ B)$

lemma *extensional-funcset-def*: *extensional-funcset S T* = (*S* → *T*) ∩ *extensional S*
⟨*proof*⟩

lemma *PiE-empty-domain*[*simp*]: *PiE* {} *T* = {λ*x*. *undefined*}
⟨*proof*⟩

lemma *PiE-UNIV-domain*: *PiE UNIV T* = *Pi UNIV T*
⟨*proof*⟩

lemma *PiE-empty-range*[*simp*]: *i* ∈ *I* ⟹ *F i* = {} ⟹ (Π$_E$ *i*∈*I*. *F i*) = {}
⟨*proof*⟩

lemma *PiE-eq-empty-iff*: $Pi_E\ I\ F = \{\} \longleftrightarrow (\exists\, i{\in}I.\ F\ i = \{\})$
⟨*proof*⟩

lemma *PiE-arb*: *f* ∈ *PiE S T* ⟹ *x* ∉ *S* ⟹ *f x* = *undefined*
⟨*proof*⟩

lemma *PiE-mem*: *f* ∈ *PiE S T* ⟹ *x* ∈ *S* ⟹ *f x* ∈ *T x*
⟨*proof*⟩

lemma *PiE-fun-upd*: *y* ∈ *T x* ⟹ *f* ∈ *PiE S T* ⟹ *f*(*x* := *y*) ∈ *PiE* (*insert x S*) *T*
⟨*proof*⟩

lemma *fun-upd-in-PiE*: *x* ∉ *S* ⟹ *f* ∈ *PiE* (*insert x S*) *T* ⟹ *f*(*x* := *undefined*) ∈ *PiE S T*
⟨*proof*⟩

lemma *PiE-insert-eq*: *PiE* (*insert x S*) *T* = (λ(*y*, *g*). *g*(*x* := *y*)) ‘ (*T x* × *PiE S T*)
⟨*proof*⟩

lemma *PiE-Int*: $Pi_E\ I\ A \cap Pi_E\ I\ B = Pi_E\ I\ (\lambda x.\ A\ x \cap B\ x)$
⟨*proof*⟩

lemma *PiE-cong*: (⋀*i*. *i*∈*I* ⟹ *A i* = *B i*) ⟹ $Pi_E\ I\ A = Pi_E\ I\ B$
⟨*proof*⟩

lemma *PiE-E* [*elim*]:
assumes *f* ∈ *PiE A B*

obtains $x \in A$ **and** $f\ x \in B\ x$
| $x \notin A$ **and** $f\ x = \mathit{undefined}$
⟨*proof*⟩

lemma *PiE-I*[*intro*!]:
$(\bigwedge x.\ x \in A \Longrightarrow f\ x \in B\ x) \Longrightarrow (\bigwedge x.\ x \notin A \Longrightarrow f\ x = \mathit{undefined}) \Longrightarrow f \in \mathit{PiE}\ A\ B$
⟨*proof*⟩

lemma *PiE-mono*: $(\bigwedge x.\ x \in A \Longrightarrow B\ x \subseteq C\ x) \Longrightarrow \mathit{PiE}\ A\ B \subseteq \mathit{PiE}\ A\ C$
⟨*proof*⟩

lemma *PiE-iff*: $f \in \mathit{PiE}\ I\ X \longleftrightarrow (\forall i{\in}I.\ f\ i \in X\ i) \wedge f \in \mathit{extensional}\ I$
⟨*proof*⟩

lemma *PiE-restrict*[*simp*]: $f \in \mathit{PiE}\ A\ B \Longrightarrow \mathit{restrict}\ f\ A = f$
⟨*proof*⟩

lemma *restrict-PiE*[*simp*]: $\mathit{restrict}\ f\ I \in \mathit{PiE}\ I\ S \longleftrightarrow f \in \mathit{Pi}\ I\ S$
⟨*proof*⟩

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*: $\mathit{Pi}_E\ I\ F = \mathit{Pi}_E\ I\ F'$
and $i \in I$
shows $F\ i \subseteq F'\ i$
⟨*proof*⟩

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 $\mathit{Pi}_E\ I\ F = \mathit{Pi}_E\ I\ F' \longleftrightarrow (\forall i{\in}I.\ F\ i = F'\ i)$
⟨*proof*⟩

lemma *PiE-eq-iff*:
$\mathit{Pi}_E\ I\ F = \mathit{Pi}_E\ I\ F' \longleftrightarrow (\forall i{\in}I.\ F\ i = F'\ i) \vee ((\exists i{\in}I.\ F\ i = \{\}) \wedge (\exists i{\in}I.\ F'\ i = \{\}))$
⟨*proof*⟩

lemma *extensional-funcset-fun-upd-restricts-rangeI*:
$\forall y \in S.\ f\ x \neq f\ y \Longrightarrow f \in (\mathit{insert}\ x\ S) \to_E T \Longrightarrow f(x := \mathit{undefined}) \in S \to_E (T - \{f\ x\})$
⟨*proof*⟩

lemma *extensional-funcset-fun-upd-extends-rangeI*:
assumes $a \in T$ $f \in S \to_E (T - \{a\})$
shows $f(x := a) \in \mathit{insert}\ x\ S \to_E T$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *extensional-funcset-extend-domain-inj-on-eq*:
assumes $x \notin S$
shows $\{f.\ f \in (insert\ x\ S) \rightarrow_E T \wedge inj\text{-}on\ f\ (insert\ x\ S)\} =$
$(\lambda(y,\ g).\ g(x{:=}y))$ ' $\{(y,\ g).\ y \in T \wedge g \in S \rightarrow_E (T - \{y\}) \wedge inj\text{-}on\ g\ S\}$
⟨*proof*⟩

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-on g S*$\}$
⟨*proof*⟩

46.7.2 Cardinality

lemma *finite-PiE*: *finite S* $\Longrightarrow$ $(\bigwedge i.\ i \in S \Longrightarrow$ *finite* $(T\ i)) \Longrightarrow$ *finite* $(\Pi_E\ i \in S.\ T\ i)$
⟨*proof*⟩

lemma *inj-combinator*: $x \notin S \Longrightarrow$ *inj-on* $(\lambda(y,\ g).\ g(x := y))$ $(T\ x \times Pi_E\ S\ T)$
⟨*proof*⟩

lemma *card-PiE*: *finite S* $\Longrightarrow$ *card* $(\Pi_E\ i \in S.\ T\ i) = (\prod\ i{\in}S.\ card\ (T\ i))$
⟨*proof*⟩

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 ⟨*proof*⟩

end

lemma *inverse-fun-apply* [*simp*]:
inverse f x = inverse (f x)
⟨*proof*⟩

lemma *divide-fun-apply* [*simp*]:
(f / g) x = f x / g x
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *fun-right-inverse*:
fixes *f :: 'b ⇒ 'a::field*
shows *zero-free f ⟹ f ∗ inverse f = 1*
⟨*proof*⟩

lemma *fun-divide-inverse*:
fixes *f g :: 'b ⇒ 'a::field*
shows *f / g = f ∗ inverse g*
⟨*proof*⟩

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 ⇒ 'a ⇒ bool* **where**
equiv x y ⟷ x ≤ y ∧ y ≤ x

notation
equiv (*op* $\approx$) **and**
equiv ((-/ $\approx$ -) [*51*, *51*] *50*)

lemma *refl* [*iff*]:
$x \approx x$
⟨*proof*⟩

lemma *trans*:
$x \approx y \Longrightarrow y \approx z \Longrightarrow x \approx z$
⟨*proof*⟩

lemma *antisym*:
$x \leq y \Longrightarrow y \leq x \Longrightarrow x \approx y$
⟨*proof*⟩

lemma *less-le*: $x < y \longleftrightarrow x \leq y \wedge \neg\, x \approx y$
⟨*proof*⟩

lemma *le-less*: $x \leq y \longleftrightarrow x < y \vee x \approx y$
⟨*proof*⟩

lemma *le-imp-less-or-eq*: $x \leq y \Longrightarrow x < y \vee x \approx y$
⟨*proof*⟩

lemma *less-imp-not-eq*: $x < y \Longrightarrow x \approx y \longleftrightarrow \mathit{False}$
⟨*proof*⟩

lemma *less-imp-not-eq2*: $x < y \Longrightarrow y \approx x \longleftrightarrow \mathit{False}$
⟨*proof*⟩

lemma *neq-le-trans*: $\neg\, a \approx b \Longrightarrow a \leq b \Longrightarrow a < b$
⟨*proof*⟩

lemma *le-neq-trans*: $a \leq b \Longrightarrow \neg\, a \approx b \Longrightarrow a < b$
⟨*proof*⟩

lemma *antisym-conv*: $y \leq x \Longrightarrow x \leq y \longleftrightarrow x \approx y$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *log-zero* [*simp*]: *log 0* = *0*
  ⟨*proof*⟩

lemma *log-one* [*simp*]: *log 1* = *0*
  ⟨*proof*⟩

lemma *log-Suc-zero* [*simp*]: *log* (*Suc 0*) = *0*
  ⟨*proof*⟩

lemma *log-rec*: *n* ≥ *2* ⟹ *log n* = *Suc* (*log* (*n div 2*))
  ⟨*proof*⟩

lemma *log-twice* [*simp*]: *n* ≠ *0* ⟹ *log* (*2* ∗ *n*) = *Suc* (*log n*)
  ⟨*proof*⟩

lemma *log-half* [*simp*]: *log* (*n div 2*) = *log n* − *1*
⟨*proof*⟩

lemma *log-exp* [*simp*]: *log* (*2* ^ *n*) = *n*
  ⟨*proof*⟩

lemma *log-mono*: *mono log*
⟨*proof*⟩

lemma *log-exp2-le*:
  assumes *n* > *0*
  shows *2* ^ *log n* ≤ *n*
⟨*proof*⟩

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*⟩

lemma [*code*]: *sqrt n* = *Max* (*Set.filter* $(\lambda m.\ m^2 \leq n)$ $\{0..n\}$)
⟨*proof*⟩

lemma *sqrt-inverse-power2* [*simp*]: *sqrt* $(n^2) = n$
⟨*proof*⟩

lemma *sqrt-zero* [*simp*]: *sqrt 0* = *0*
⟨*proof*⟩

lemma *sqrt-one* [*simp*]: *sqrt 1* = *1*
⟨*proof*⟩

lemma *mono-sqrt*: *mono sqrt*
⟨*proof*⟩

lemma *sqrt-greater-zero-iff* [*simp*]: *sqrt n* > *0* $\longleftrightarrow$ *n* > *0*
⟨*proof*⟩

lemma *sqrt-power2-le* [*simp*]: $(sqrt\ n)^2 \leq n$
⟨*proof*⟩

lemma *sqrt-le*: *sqrt n* $\leq$ *n*
⟨*proof*⟩

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 $f :: 'a \Rightarrow 'b$::*order*
assumes *mono f*
assumes $f\ x < f\ y$
obtains $x < y$
⟨*proof*⟩

end

lemma (**in** *semidom-divide*) *power-diff*:
fixes $a :: 'a$
assumes $a \neq 0$
assumes $m \geq n$
shows $a\ \hat{}\ (m - n) = (a\ \hat{}\ m)\ div\ (a\ \hat{}\ n)$
⟨*proof*⟩

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: $(\lambda n.\ 0) \lesssim (\lambda n.\ k) \lesssim$ *Discrete.log* $\lesssim$ *Discrete.sqrt* $\lesssim$ *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* :: $(nat \Rightarrow nat) \Rightarrow (nat \Rightarrow nat) \Rightarrow bool$ (**infix** $\lesssim$ *50*)
where
$f \lesssim g \longleftrightarrow (\exists\, c{>}0.\ \exists\, n.\ \forall\, m{>}n.\ f\ m \leq c * g\ m)$

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 $\exists\, c{>}0.\ \exists\, n.\ \forall\, m{>}n.\ f\ m \leq c * g\ m$
shows $f \lesssim g$
⟨*proof*⟩

lemma *not-less-eq-funI*:
assumes $\bigwedge c\ n.\ c > 0 \Longrightarrow \exists\, m{>}n.\ c * g\ m < f\ m$
shows $\neg\ f \lesssim g$
⟨*proof*⟩

lemma *less-eq-funE* [*elim?*]:
assumes $f \lesssim g$
obtains $n\ c$ **where** $c > 0$ **and** $\bigwedge m.\ m > n \Longrightarrow f\ m \leq c * g\ m$
⟨*proof*⟩

lemma *not-less-eq-funE*:
assumes $\neg\ f \lesssim g$ **and** $c > 0$
obtains m **where** $m > n$ **and** $c * g\ m < f\ m$
⟨*proof*⟩

50.4 The $\approx$ relation, the equivalence relation induced by $\lesssim$

definition *equiv-fun* :: $(nat \Rightarrow nat) \Rightarrow (nat \Rightarrow nat) \Rightarrow bool$ (**infix** $\cong$ *50*)
where
$f \cong g \longleftrightarrow$
$(\exists\, c_1{>}0.\ \exists\, c_2{>}0.\ \exists\, n.\ \forall\, m{>}n.\ f\ m \leq c_1 * g\ m \wedge g\ m \leq c_2 * f\ m)$

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 $\exists\, c_1{>}0.\ \exists\, c_2{>}0.\ \exists\, n.\ \forall\, m{>}n.\ f\ m \leq c_1 * g\ m \wedge g\ m \leq c_2 * f\ m$
shows $f \cong g$
⟨*proof*⟩

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 $\neg\ f \cong g$
⟨*proof*⟩

lemma *equiv-funE*:
assumes $f \cong g$
obtains $n\ c_1\ c_2$ **where** $c_1 > 0$ **and** $c_2 > 0$
and $\bigwedge m.\ m > n \Longrightarrow f\ m \leq c_1 * g\ m \wedge g\ m \leq c_2 * f\ m$
⟨*proof*⟩

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 $c_1 * f\ m < g\ m \vee c_2 * g\ m < f\ m$
⟨*proof*⟩

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
$f \prec g \longleftrightarrow f \lesssim g \wedge \neg\ g \lesssim f$

lemma *less-funI*:
assumes $\exists\, c{>}0.\ \exists\, n.\ \forall\, m{>}n.\ f\ m \leq c * g\ m$
and $\bigwedge c\ n.\ c > 0 \Longrightarrow \exists\, m{>}n.\ c * f\ m < g\ m$
shows $f \prec g$
⟨*proof*⟩

lemma *not-less-funI*:
assumes $\bigwedge c\ n.\ c > 0 \Longrightarrow \exists\, m{>}n.\ c * g\ m < f\ m$
and $\exists\, c{>}0.\ \exists\, n.\ \forall\, m{>}n.\ g\ m \leq c * f\ m$
shows $\neg\ f \prec g$
⟨*proof*⟩

lemma *less-funE* [*elim?*]:
assumes $f \prec g$
obtains $n\ c$ **where** $c > 0$ **and** $\bigwedge m.\ m > n \Longrightarrow f\ m \leq c * g\ m$
and $\bigwedge c\ n.\ c > 0 \Longrightarrow \exists\, m{>}n.\ c * f\ m < g\ m$
⟨*proof*⟩

lemma *not-less-funE*:
assumes $\neg\ f \prec g$ **and** $c > 0$
obtains m **where** $m > n$ **and** $c * g\ m < f\ m$
$\mid d\ q$ **where** $\bigwedge m.\ d > 0 \Longrightarrow m > q \Longrightarrow g\ q \leq d * f\ q$
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

lemma
  assumes *strict-mono f*
  shows $(\lambda n.\ f\ n + k) \cong f$
⟨*proof*⟩

lemma
  $(\lambda n.\ Suc\ k * f\ n) \cong f$
⟨*proof*⟩

lemma
  $f \lesssim (\lambda n.\ f\ n + g\ n)$
  ⟨*proof*⟩

lemma
  $(\lambda\text{-}.\ 0) \prec (\lambda n.\ Suc\ k)$
  ⟨*proof*⟩

lemma
  $(\lambda\text{-}.\ k) \prec Discrete.log$
⟨*proof*⟩

$Discrete.log \prec Discrete.sqrt$

lemma
  $Discrete.sqrt \prec id$
⟨*proof*⟩

lemma

$id \prec (\lambda n.\ n^2)$
⟨*proof*⟩

lemma
$(\lambda n.\ n\ \hat{}\ k) \prec (\lambda n.\ n\ \hat{}\ Suc\ k)$
⟨*proof*⟩

$(\lambda n.\ n^k) \prec op\ \hat{}\ 2$

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*: $cmod\ w \leq cmod\ (w + z) + norm\ z$
⟨*proof*⟩

51.2 Basic lemmas about polynomials

lemma *poly-bound-exists*:
fixes $p :: 'a::\{comm\text{-}semiring\text{-}0, real\text{-}normed\text{-}div\text{-}algebra\}\ poly$
shows $\exists m.\ m > 0 \wedge (\forall z.\ norm\ z \leq r \longrightarrow norm\ (poly\ p\ z) \leq m)$
⟨*proof*⟩

Offsetting the variable in a polynomial gives another of same degree

definition *offset-poly* :: $'a::comm\text{-}semiring\text{-}0\ poly \Rightarrow 'a \Rightarrow 'a\ poly$
where $offset\text{-}poly\ p\ h = fold\text{-}coeffs\ (\lambda a\ q.\ smult\ h\ q + pCons\ a\ q)\ p\ 0$

lemma *offset-poly-0*: $offset\text{-}poly\ 0\ h = 0$
⟨*proof*⟩

lemma *offset-poly-pCons*:
$offset\text{-}poly\ (pCons\ a\ p)\ h =$
$smult\ h\ (offset\text{-}poly\ p\ h) + pCons\ a\ (offset\text{-}poly\ p\ h)$
⟨*proof*⟩

lemma *offset-poly-single*: $offset\text{-}poly\ [:a:]\ h = [:a:]$
⟨*proof*⟩

lemma *poly-offset-poly*: $poly\ (offset\text{-}poly\ p\ h)\ x = poly\ p\ (h + x)$
⟨*proof*⟩

lemma *offset-poly-eq-0-lemma*: $smult\ c\ p + pCons\ a\ p = 0 \Longrightarrow p = 0$

⟨*proof*⟩

lemma *offset-poly-eq-0-iff*: *offset-poly p h* = *0* ⟷ *p* = *0*
⟨*proof*⟩

lemma *degree-offset-poly*: *degree* (*offset-poly p h*) = *degree p*
⟨*proof*⟩

definition *psize p* = (*if p* = *0 then 0 else Suc* (*degree p*))

lemma *psize-eq-0-iff* [*simp*]: *psize p* = *0* ⟷ *p* = *0*
⟨*proof*⟩

lemma *poly-offset*:
fixes *p* :: *'a*::*comm-ring-1 poly*
shows ∃ *q*. *psize q* = *psize p* ∧ (∀ *x*. *poly q x* = *poly p* (*a* + *x*))
⟨*proof*⟩

An alternative useful formulation of completeness of the reals

lemma *real-sup-exists*:
assumes *ex*: ∃ *x*. *P x*
and *bz*: ∃ *z*. ∀ *x*. *P x* ⟶ *x* < *z*
shows ∃ *s*::*real*. ∀ *y*. (∃ *x*. *P x* ∧ *y* < *x*) ⟷ *y* < *s*
⟨*proof*⟩

51.3 Fundamental theorem of algebra

lemma *unimodular-reduce-norm*:
assumes *md*: *cmod z* = *1*
shows *cmod* (*z* + *1*) < *1* ∨ *cmod* (*z* − *1*) < *1* ∨ *cmod* (*z* + *ii*) < *1* ∨ *cmod* (*z* − *ii*) < *1*
⟨*proof*⟩

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*
⟨*proof*⟩

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*|
⟨*proof*⟩

lemma *bolzano-weierstrass-complex-disc*:
assumes *r*: ∀ *n*. *cmod* (*s n*) ≤ *r*
shows ∃ *f z*. *subseq f* ∧ (∀ *e* >*0*. ∃ *N*. ∀ *n* ≥ *N*. *cmod* (*s* (*f n*) − *z*) < *e*)
⟨*proof*⟩

Polynomial is continuous.

lemma *poly-cont*:
fixes p :: $'a$::{*comm-semiring-0*,*real-normed-div-algebra*} *poly*
assumes *ep*: $e > 0$
shows $\exists\, d > 0.\ \forall\, w.\ 0 < norm\ (w - z) \wedge norm\ (w - z) < d \longrightarrow norm\ (poly\ p\ w - poly\ p\ z) < e$
⟨*proof*⟩

Hence a polynomial attains minimum on a closed disc in the complex plane.

lemma *poly-minimum-modulus-disc*: $\exists\, z.\ \forall\, w.\ cmod\ w \leq r \longrightarrow cmod\ (poly\ p\ z) \leq cmod\ (poly\ p\ w)$
⟨*proof*⟩

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 $\exists\, r.\ \forall\, z.\ r \leq norm\ z \longrightarrow d \leq norm\ (poly\ (pCons\ a\ p)\ z)$
⟨*proof*⟩

Hence polynomial's modulus attains its minimum somewhere.

lemma *poly-minimum-modulus*: $\exists\, z. \forall\, w.\ cmod\ (poly\ p\ z) \leq cmod\ (poly\ p\ w)$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *poly-replicate-append*: *poly* (*monom 1 n* $*$ *p*) (*x*::$'a$::*comm-ring-1*) = *x*^*n* $*$ *poly p x*
⟨*proof*⟩

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 $\exists\, k\ a\ q.\ a \neq 0 \wedge Suc\ (psize\ q + k) = psize\ p \wedge (\forall\, z.\ poly\ p\ z = z\hat{}k * poly\ (pCons\ a\ q)\ z)$
⟨*proof*⟩

lemma *poly-decompose*:
assumes *nc*: $\neg$ *constant* (*poly p*)
shows $\exists\, k\ a\ q.\ a \neq (0::'a::idom) \wedge k \neq 0 \wedge$
$psize\ q + k + 1 = psize\ p \wedge$
$(\forall\, z.\ poly\ p\ z = poly\ p\ 0 + z\hat{}k * poly\ (pCons\ a\ q)\ z)$
⟨*proof*⟩

Fundamental theorem of algebra

lemma *fundamental-theorem-of-algebra*:
assumes *nc*: ¬ *constant* (*poly p*)
shows ∃ *z*::*complex*. *poly p z* = *0*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *nullstellensatz-univariate*:
(∀ *x*. *poly p x* = (*0*::*complex*) ⟶ *poly q x* = *0*) ⟷
p dvd (*q* ^ (*degree p*)) ∨ (*p* = *0* ∧ *q* = *0*)
⟨*proof*⟩

Useful lemma

lemma *constant-degree*:
fixes *p* :: *'a*::{*idom*,*ring-char-0*} *poly*
shows *constant* (*poly p*) ⟷ *degree p* = *0* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *mpoly-sub-conv*:
fixes *x* :: *'a*::*comm-ring-1*
shows *poly p x* − *poly q x* = *poly p x* + −*1* ∗ *poly q x*
⟨*proof*⟩

lemma *poly-pad-rule*: *poly p x* = *0* ⟹ *poly* (*pCons 0 p*) *x* = *0*

⟨*proof*⟩

lemma *poly-cancel-eq-conv*:
fixes x :: $'a$::*field*
shows $x = 0 \Longrightarrow a \neq 0 \Longrightarrow y = 0 \longleftrightarrow a * y - b * x = 0$
⟨*proof*⟩

lemma *poly-divides-pad-rule*:
fixes p:: ($'a$::*comm-ring-1*) *poly*
assumes *pq*: *p dvd q*
shows *p dvd* (*pCons 0 q*)
⟨*proof*⟩

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*⟩

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*⟩

lemma *basic-cqe-conv1*:
$(\exists x.\ poly\ p\ x = 0 \wedge poly\ 0\ x \neq 0) \longleftrightarrow False$
$(\exists x.\ poly\ 0\ x \neq 0) \longleftrightarrow False$
$(\exists x.\ poly\ [:c:]\ x \neq 0) \longleftrightarrow c \neq 0$
$(\exists x.\ poly\ 0\ x = 0) \longleftrightarrow True$
$(\exists x.\ poly\ [:c:]\ x = 0) \longleftrightarrow c = 0$
⟨*proof*⟩

lemma *basic-cqe-conv2*:
assumes *l*: $p \neq 0$
shows $\exists x.\ poly\ (pCons\ a\ (pCons\ b\ p))\ x = (0$::*complex*$)$
⟨*proof*⟩

lemma *basic-cqe-conv-2b*: $(\exists x.\ poly\ p\ x \neq (0$::*complex*$)) \longleftrightarrow p \neq 0$
⟨*proof*⟩

lemma *basic-cqe-conv3*:
fixes *p q* :: *complex poly*
assumes *l*: $p \neq 0$
shows $(\exists x.\ poly\ (pCons\ a\ p)\ x = 0 \wedge poly\ q\ x \neq 0) \longleftrightarrow \neg\ (pCons\ a\ p)\ dvd\ (q$
^ *psize p*)

⟨*proof*⟩

lemma *basic-cqe-conv4*:
fixes *p q* :: *complex poly*
assumes *h*: ⋀*x*. *poly* (*q* ^ *n*) *x* = *poly r x*
shows *p dvd* (*q* ^ *n*) ⟷ *p dvd r*
⟨*proof*⟩

lemma *poly-const-conv*:
fixes *x* :: 'a::*comm-ring-1*
shows *poly* [:*c*:] *x* = *y* ⟷ *c* = *y*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *less-funE*:
assumes *less-fun f g*
obtains *k* **where** *f k* < *g k* **and** ⋀*k'*. *k'* < *k* ⟹ *f k'* = *g k'*
⟨*proof*⟩

lemma *less-fun-asym*:
assumes *less-fun f g*
shows ¬ *less-fun g f*
⟨*proof*⟩

lemma *less-fun-irrefl*:
¬ *less-fun f f*
⟨*proof*⟩

lemma *less-fun-trans*:
assumes *less-fun f g* **and** *less-fun g h*
shows *less-fun f h*
⟨*proof*⟩

lemma *order-less-fun*:
  class.order (λ*f g*. *less-fun f g* ∨ *f* = *g*) *less-fun*
  ⟨*proof*⟩

lemma *less-fun-trichotomy*:
  assumes *finite* {*k*. *f k* ≠ *g k*}
  shows *less-fun f g* ∨ *f* = *g* ∨ *less-fun g f*
⟨*proof*⟩

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* ⇒ '*a*) ⇒ '*a*
where
  expand-set: *G g* = *comm-monoid-set.F f 1 g* {*a*. *g a* ≠ *1*}

interpretation *F*: *comm-monoid-set f 1*
  ⟨*proof*⟩

lemma *expand-superset*:
  assumes *finite A* **and** {*a*. *g a* ≠ *1*} ⊆ *A*
  shows *G g* = *F.F g A*
  ⟨*proof*⟩

lemma *conditionalize*:
  assumes *finite A*
  shows *F.F g A* = *G* (λ*a*. *if a* ∈ *A then g a else 1*)
  ⟨*proof*⟩

lemma *neutral* [*simp*]:
  G (λ*a*. *1*) = *1*
  ⟨*proof*⟩

lemma *update* [*simp*]:
  assumes *finite* {*a*. *g a* ≠ *1*}

assumes $g\ a = 1$
shows $G\ (g(a := b)) = b * G\ g$
⟨*proof*⟩

lemma *infinite* [*simp*]:
$\neg$ *finite* $\{a.\ g\ a \neq 1\} \Longrightarrow G\ g = 1$
⟨*proof*⟩

lemma *cong*:
assumes $\bigwedge a.\ g\ a = h\ a$
shows $G\ g = G\ h$
⟨*proof*⟩

lemma *strong-cong* [*cong*]:
assumes $\bigwedge a.\ g\ a = h\ a$
shows $G\ (\lambda a.\ g\ a) = G\ (\lambda a.\ h\ a)$
⟨*proof*⟩

lemma *not-neutral-obtains-not-neutral*:
assumes $G\ g \neq 1$
obtains a **where** $g\ a \neq 1$
⟨*proof*⟩

lemma *reindex-cong*:
assumes *bij* l
assumes $g \circ l = h$
shows $G\ g = G\ h$
⟨*proof*⟩

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*⟩

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*⟩

lemma *cartesian-product*:
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 $G\ (\lambda a.\ G\ (g\ a)) = G\ (\lambda(a,\ b).\ g\ a\ b)$
⟨*proof*⟩

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*⟩

lemma *delta* [*simp*]:
$G\ (\lambda b.\ \textit{if}\ b = a\ \textit{then}\ g\ b\ \textit{else}\ 1) = g\ a$
⟨*proof*⟩

lemma *delta'* [*simp*]:
$G\ (\lambda b.\ \textit{if}\ a = b\ \textit{then}\ g\ b\ \textit{else}\ 1) = g\ a$
⟨*proof*⟩

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*⟩

end

syntax (*ASCII*)
-Sum-any :: *pttrn* $\Rightarrow$ *'a* $\Rightarrow$ *'a*::*comm-monoid-add* (*(3SUM -. -) [0, 10] 10*)
syntax
-Sum-any :: *pttrn* $\Rightarrow$ *'a* $\Rightarrow$ *'a*::*comm-monoid-add* (*(3$\sum$ -. -) [0, 10] 10*)
translations
$\sum a.\ b \rightleftharpoons$ *CONST Sum-any* $(\lambda a.\ b)$

lemma *Sum-any-left-distrib*:
fixes *r* :: *'a* :: *semiring-0*
assumes *finite* $\{a.\ g\ a \neq 0\}$
shows *Sum-any g* $*$ *r* $= (\sum n.\ g\ n * r)$
⟨*proof*⟩

lemma *Sum-any-right-distrib*:
fixes *r* :: *'a* :: *semiring-0*
assumes *finite* $\{a.\ g\ a \neq 0\}$

shows $r * \mathit{Sum\text{-}any}\ g = (\sum n.\ r * g\ n)$
⟨*proof*⟩

lemma *Sum-any-product*:
fixes $f\ g :: 'b \Rightarrow 'a$::*semiring-0*
assumes *finite* $\{a.\ f\ a \neq 0\}$ **and** *finite* $\{b.\ g\ b \neq 0\}$
shows $\mathit{Sum\text{-}any}\ f * \mathit{Sum\text{-}any}\ g = (\sum a.\ \sum b.\ f\ a * g\ b)$
⟨*proof*⟩

lemma *Sum-any-eq-zero-iff* [*simp*]:
fixes $f :: 'a \Rightarrow nat$
assumes *finite* $\{a.\ f\ a \neq 0\}$
shows $\mathit{Sum\text{-}any}\ f = 0 \longleftrightarrow f = (\lambda\text{-}.\ 0)$
⟨*proof*⟩

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*⟩

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*$\prod$ -. -) [*0*, *10*] *10*)
translations
$\prod a.\ b == \mathit{CONST}\ \mathit{Prod\text{-}any}\ (\lambda a.\ b)$

lemma *Prod-any-zero*:
fixes $f :: 'b \Rightarrow 'a$ :: *comm-semiring-1*
assumes *finite* $\{a.\ f\ a \neq 1\}$
assumes $f\ a = 0$
shows $(\prod a.\ f\ a) = 0$
⟨*proof*⟩

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 $f\ a \neq 0$
⟨*proof*⟩

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*⟩

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*]
⟨*proof*⟩

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*))
  ⟨*proof*⟩

lemma [*code*]:
  size-iarray f as = *Suc* (*size-list f* (*IArray.list-of as*))
  ⟨*proof*⟩

lemma [*code*]:
  rec-iarray f as = *f* (*IArray.list-of as*)
  ⟨*proof*⟩

lemma [*code*]:
  case-iarray f as = *f* (*IArray.list-of as*)
  ⟨*proof*⟩

lemma [*code*]:
  set-iarray as = *set* (*IArray.list-of as*)
  ⟨*proof*⟩

lemma [*code*]:
  map-iarray f as = *IArray* (*map f* (*IArray.list-of as*))
  ⟨*proof*⟩

lemma [*code*]:
  rel-iarray r as bs = *list-all2 r* (*IArray.list-of as*) (*IArray.list-of bs*)
  ⟨*proof*⟩

lemma [*code*]:
  HOL.equal as bs ⟷ *HOL.equal* (*IArray.list-of as*) (*IArray.list-of bs*)
  ⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma [*code*]:
as !! *n* = *IArray.sub'* (*as*, *integer-of-nat n*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma [*code*]:
IArray.length as = *nat-of-integer* (*IArray.length' as*)
⟨*proof*⟩

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*))
⟨*proof*⟩

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 ≤ *y* ⟷ (*case x of Bot* ⇒ *True* | *Value x* ⇒ (*case y of Bot* ⇒ *False* | *Value y* ⇒ *x* ≤ *y*))

definition *less-bot* **where**
x < *y* ⟷ (*case y of Bot* ⇒ *False* | *Value y* ⇒ (*case x of Bot* ⇒ *True* | *Value x* ⇒ *x* < *y*))

lemma *less-eq-bot-Bot* [*simp*]: *Bot* ≤ *x*
⟨*proof*⟩

lemma *less-eq-bot-Bot-code* [*code*]: *Bot* ≤ *x* ⟷ *True*
⟨*proof*⟩

lemma *less-eq-bot-Bot-is-Bot*: *x* ≤ *Bot* ⟹ *x* = *Bot*
⟨*proof*⟩

lemma *less-eq-bot-Value-Bot* [*simp*, *code*]: *Value x* ≤ *Bot* ⟷ *False*
⟨*proof*⟩

lemma *less-eq-bot-Value* [*simp*, *code*]: *Value x* ≤ *Value y* ⟷ *x* ≤ *y*
⟨*proof*⟩

lemma *less-bot-Bot* [*simp*, *code*]: *x* < *Bot* ⟷ *False*

⟨*proof*⟩

lemma *less-bot-Bot-is-Value*: $Bot < x \Longrightarrow \exists\, z.\ x = Value\ z$
⟨*proof*⟩

lemma *less-bot-Bot-Value* [*simp*]: $Bot < Value\ x$
⟨*proof*⟩

lemma *less-bot-Bot-Value-code* [*code*]: $Bot < Value\ x \longleftrightarrow True$
⟨*proof*⟩

lemma *less-bot-Value* [*simp*, *code*]: $Value\ x < Value\ y \longleftrightarrow x < y$
⟨*proof*⟩

instance
⟨*proof*⟩

end

instance *bot* :: (*order*) *order*
⟨*proof*⟩

instance *bot* :: (*linorder*) *linorder*
⟨*proof*⟩

instantiation *bot* :: (*order*) *bot*
begin
definition $bot = Bot$
instance ⟨*proof*⟩
end

instantiation *bot* :: (*top*) *top*
begin
definition $top = Value\ top$
instance ⟨*proof*⟩
end

instantiation *bot* :: (*semilattice-inf*) *semilattice-inf*
begin

definition *inf-bot*
where
$inf\ x\ y =$
(*case* x *of*
$Bot \Rightarrow Bot$
| $Value\ v \Rightarrow$
(*case* y *of*
$Bot \Rightarrow Bot$
| $Value\ v' \Rightarrow Value\ (inf\ v\ v')$))

instance
⟨*proof*⟩

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
⟨*proof*⟩

end

instance *bot* :: (*lattice*) *bounded-lattice-bot*
⟨*proof*⟩

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 ≤ *y* ⟷ (*case y of Top* ⇒ *True* | *Value y* ⇒ (*case x of Top* ⇒ *False* | *Value x* ⇒ *x* ≤ *y*))

definition *less-top* **where**
x < *y* ⟷ (*case x of Top* ⇒ *False* | *Value x* ⇒ (*case y of Top* ⇒ *True* | *Value y* ⇒ *x* < *y*))

lemma *less-eq-top-Top* [*simp*]: *x* ≤ *Top*
⟨*proof*⟩

lemma *less-eq-top-Top-code* [*code*]: *x* ≤ *Top* ⟷ *True*
⟨*proof*⟩

lemma *less-eq-top-is-Top*: *Top* ≤ *x* ⟹ *x* = *Top*

⟨*proof*⟩

lemma *less-eq-top-Top-Value* [*simp*, *code*]: *Top* ≤ *Value x* ⟷ *False*
⟨*proof*⟩

lemma *less-eq-top-Value-Value* [*simp*, *code*]: *Value x* ≤ *Value y* ⟷ *x* ≤ *y*
⟨*proof*⟩

lemma *less-top-Top* [*simp*, *code*]: *Top* < *x* ⟷ *False*
⟨*proof*⟩

lemma *less-top-Top-is-Value*: *x* < *Top* ⟹ ∃ *z*. *x* = *Value z*
⟨*proof*⟩

lemma *less-top-Value-Top* [*simp*]: *Value x* < *Top*
⟨*proof*⟩

lemma *less-top-Value-Top-code* [*code*]: *Value x* < *Top* ⟷ *True*
⟨*proof*⟩

lemma *less-top-Value* [*simp*, *code*]: *Value x* < *Value y* ⟷ *x* < *y*
⟨*proof*⟩

instance
⟨*proof*⟩

end

instance *top* :: (*order*) *order*
⟨*proof*⟩

instance *top* :: (*linorder*) *linorder*
⟨*proof*⟩

instantiation *top* :: (*order*) *top*
begin
definition *top* = *Top*
instance ⟨*proof*⟩
end

instantiation *top* :: (*bot*) *bot*
begin
definition *bot* = *Value bot*
instance ⟨*proof*⟩
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
⟨*proof*⟩

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
⟨*proof*⟩

end

instance *top* :: (*lattice*) *bounded-lattice-top*
⟨*proof*⟩

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
  ⟨proof⟩

lemma [simp]: y ≤ Top
  ⟨proof⟩

lemma greater-than-two-values:
  assumes a ≠ b Value a ≤ z Value b ≤ z
  shows z = Top
  ⟨proof⟩

lemma lesser-than-two-values:
  assumes a ≠ b z ≤ Value a z ≤ Value b
  shows z = Bot
  ⟨proof⟩

instance
  ⟨proof⟩

end

instantiation flat-complete-lattice :: (type) bot
begin
  definition bot = Bot
  instance ⟨proof⟩
end

instantiation flat-complete-lattice :: (type) top
begin
  definition top = Top
  instance ⟨proof⟩
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
⟨*proof*⟩

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*⟩

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*)
  ⟨*proof*⟩

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* ∈ *sset s* **and** ⋀*s*. *P* (*shd s*) *s* **and** ⋀*s y*. ⟦*y* ∈ *sset* (*stl s*); *P y* (*stl s*)⟧ ⟹ *P y s*
  shows *P y s*
⟨*proof*⟩

lemma *smap-ctr*: *smap f s* = *x* ## *s'* ⟷ *f* (*shd s*) = *x* ∧ *smap f* (*stl s*) = *s'*
  ⟨*proof*⟩

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*
⟨*proof*⟩

lemma *shift-append*[*simp*]: (*xs* @ *ys*) @− *s* = *xs* @− *ys* @− *s*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *sset-shift*[*simp*]: *sset* (*xs* @− *s*) = *set xs* ∪ *sset s*
⟨*proof*⟩

lemma *shift-left-inj*[*simp*]: *xs* @− *s1* = *xs* @− *s2* ⟷ *s1* = *s2*
⟨*proof*⟩

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* ∈ *A*; *s* ∈ *streams A*⟧ ⟹ *a* ## *s* ∈ *streams A*

end

lemma *in-streams*: *stl s* ∈ *streams S* ⟹ *shd s* ∈ *S* ⟹ *s* ∈ *streams S*
⟨*proof*⟩

lemma *streamsE*: *s* ∈ *streams A* ⟹ (*shd s* ∈ *A* ⟹ *stl s* ∈ *streams A* ⟹ *P*)
⟹ *P*
⟨*proof*⟩

lemma *Stream-image*: *x* ## *y* ∈ (*op* ## *x'*) ' *Y* ⟷ *x* = *x'* ∧ *y* ∈ *Y*
⟨*proof*⟩

lemma *shift-streams*: ⟦*w* ∈ *lists A*; *s* ∈ *streams A*⟧ ⟹ *w* @− *s* ∈ *streams A*
⟨*proof*⟩

lemma *streams-Stream*: *x* ## *s* ∈ *streams A* ⟷ *x* ∈ *A* ∧ *s* ∈ *streams A*
⟨*proof*⟩

lemma *streams-stl*: $s \in streams\ A \Longrightarrow stl\ s \in streams\ A$
⟨*proof*⟩

lemma *streams-shd*: $s \in streams\ A \Longrightarrow shd\ s \in A$
⟨*proof*⟩

lemma *sset-streams*:
assumes $sset\ s \subseteq A$
shows $s \in streams\ A$
⟨*proof*⟩

lemma *streams-sset*:
assumes $s \in streams\ A$
shows $sset\ s \subseteq A$
⟨*proof*⟩

lemma *streams-iff-sset*: $s \in streams\ A \longleftrightarrow sset\ s \subseteq A$
⟨*proof*⟩

lemma *streams-mono*: $s \in streams\ A \Longrightarrow A \subseteq B \Longrightarrow s \in streams\ B$
⟨*proof*⟩

lemma *streams-mono2*: $S \subseteq T \Longrightarrow streams\ S \subseteq streams\ T$
⟨*proof*⟩

lemma *smap-streams*: $s \in streams\ A \Longrightarrow (\bigwedge x.\ x \in A \Longrightarrow f\ x \in B) \Longrightarrow smap\ f\ s \in streams\ B$
⟨*proof*⟩

lemma *streams-empty*: $streams\ \{\} = \{\}$
⟨*proof*⟩

lemma *streams-UNIV*[*simp*]: $streams\ UNIV = UNIV$
⟨*proof*⟩

55.3 nth, take, drop for streams

primrec *snth* :: *'a stream* ⇒ *nat* ⇒ *'a* (**infixl** !! *100*) **where**
$s\ !!\ 0 = shd\ s$
| $s\ !!\ Suc\ n = stl\ s\ !!\ n$

lemma *snth-Stream*: $(x\ \#\#\ s)\ !!\ Suc\ i = s\ !!\ i$
⟨*proof*⟩

lemma *snth-smap*[*simp*]: $smap\ f\ s\ !!\ n = f\ (s\ !!\ n)$
⟨*proof*⟩

lemma *shift-snth-less*[*simp*]: $p < length\ xs \Longrightarrow (xs\ @-\ s)\ !!\ p = xs\ !\ p$
⟨*proof*⟩

lemma *shift-snth-ge*[*simp*]: $p \geq$ *length xs* $\Longrightarrow$ (*xs* @− *s*) !! *p* = *s* !! (*p* − *length xs*)
⟨*proof*⟩

lemma *shift-snth*: (*xs* @− *s*) !! *n* = (*if n* < *length xs then xs* ! *n else s* !! (*n* − *length xs*))
⟨*proof*⟩

lemma *snth-sset*[*simp*]: *s* !! *n* ∈ *sset s*
⟨*proof*⟩

lemma *sset-range*: *sset s* = *range* (*snth s*)
⟨*proof*⟩

lemma *streams-iff-snth*: *s* ∈ *streams X* ⟷ (∀ *n*. *s* !! *n* ∈ *X*)
⟨*proof*⟩

lemma *snth-in*: *s* ∈ *streams X* $\Longrightarrow$ *s* !! *n* ∈ *X*
⟨*proof*⟩

primrec *stake* :: *nat* ⇒ *'a stream* ⇒ *'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*
⟨*proof*⟩

lemma *stake-smap*[*simp*]: *stake n* (*smap f s*) = *map f* (*stake n s*)
⟨*proof*⟩

lemma *take-stake*: *take n* (*stake m s*) = *stake* (*min n m*) *s*
⟨*proof*⟩

primrec *sdrop* :: *nat* ⇒ *'a stream* ⇒ *'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*
⟨*proof*⟩

lemma *sdrop-smap*[*simp*]: *sdrop n* (*smap f s*) = *smap f* (*sdrop n s*)
⟨*proof*⟩

lemma *sdrop-stl*: *sdrop n* (*stl s*) = *stl* (*sdrop n s*)
⟨*proof*⟩

lemma *drop-stake*: *drop n* (*stake m s*) = *stake* (*m* − *n*) (*sdrop n s*)

⟨*proof*⟩

lemma *stake-sdrop*: *stake n s @− sdrop n s = s*
⟨*proof*⟩

lemma *id-stake-snth-sdrop*:
s = stake i s @− s !! i ## sdrop (Suc i) s
⟨*proof*⟩

lemma *smap-alt*: *smap f s = s' ⟷ (∀ n. f (s !! n) = s' !! n)* (**is** *?L = ?R*)
⟨*proof*⟩

lemma *stake-invert-Nil*[*iff*]: *stake n s = [] ⟷ n = 0*
⟨*proof*⟩

lemma *sdrop-shift*: *sdrop i (w @− s) = drop i w @− sdrop (i − length w) s*
⟨*proof*⟩

lemma *stake-shift*: *stake i (w @− s) = take i w @ stake (i − length w) s*
⟨*proof*⟩

lemma *stake-add*[*simp*]: *stake m s @ stake n (sdrop m s) = stake (m + n) s*
⟨*proof*⟩

lemma *sdrop-add*[*simp*]: *sdrop n (sdrop m s) = sdrop (m + n) s*
⟨*proof*⟩

lemma *sdrop-snth*: *sdrop n s !! m = s !! (n + m)*
⟨*proof*⟩

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)
⟨*proof*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

lemma *stream-all-shift*[*simp*]: *stream-all P* (*xs* @− *s*) = (*list-all P xs* ∧ *stream-all P s*)
⟨*proof*⟩

lemma *stream-all-Stream*: *stream-all P* (*x* ## *X*) ⟷ *P x* ∧ *stream-all P X*
⟨*proof*⟩

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*⟩

lemma *cycle-Cons*[*code*]: *cycle* (*x* # *xs*) = *x* ## *cycle* (*xs* @ [*x*])
⟨*proof*⟩

lemma *cycle-rotated*: ⟦*v* ≠ []; *cycle u* = *v* @− *s*⟧ ⟹ *cycle* (*tl u* @ [*hd u*]) = *tl v* @− *s*
⟨*proof*⟩

lemma *stake-append*: *stake n* (*u* @− *s*) = *take* (*min* (*length u*) *n*) *u* @ *stake* (*n* − *length u*) *s*
⟨*proof*⟩

lemma *stake-cycle-le*[*simp*]:
assumes *u* ≠ [] *n* < *length u*
shows *stake n* (*cycle u*) = *take n u*
⟨*proof*⟩

lemma *stake-cycle-eq*[*simp*]: *u* ≠ [] ⟹ *stake* (*length u*) (*cycle u*) = *u*
⟨*proof*⟩

lemma *sdrop-cycle-eq*[*simp*]: *u* ≠ [] ⟹ *sdrop* (*length u*) (*cycle u*) = *cycle u*
⟨*proof*⟩

lemma *stake-cycle-eq-mod-0*[*simp*]: ⟦*u* ≠ []; *n mod length u* = *0*⟧ ⟹
stake n (*cycle u*) = *concat* (*replicate* (*n div length u*) *u*)
⟨*proof*⟩

lemma *sdrop-cycle-eq-mod-0*[*simp*]: ⟦*u* ≠ []; *n mod length u* = *0*⟧ ⟹
sdrop n (*cycle u*) = *cycle u*
⟨*proof*⟩

lemma *stake-cycle*: *u* ≠ [] ⟹
stake n (*cycle u*) = *concat* (*replicate* (*n div length u*) *u*) @ *take* (*n mod length u*) *u*
⟨*proof*⟩

lemma *sdrop-cycle*: *u* ≠ [] ⟹ *sdrop n* (*cycle u*) = *cycle* (*rotate* (*n mod length u*) *u*)
⟨*proof*⟩

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*]
⟨*proof*⟩

lemma *snth-siterate*[*simp*]: *siterate f x* !! *n* = (*f*^^*n*) *x*
⟨*proof*⟩

lemma *sdrop-siterate*[*simp*]: *sdrop n* (*siterate f x*) = *siterate f* ((*f*^^*n*) *x*)
⟨*proof*⟩

lemma *stake-siterate*[*simp*]: *stake n* (*siterate f x*) = *map* (λ*n*. (*f*^^*n*) *x*) [*0* ..< *n*]
⟨*proof*⟩

lemma *sset-siterate*: *sset* (*siterate f x*) = {(*f*^^*n*) *x* | *n*. *True*}
⟨*proof*⟩

lemma *smap-siterate*: *smap f* (*siterate f x*) = *siterate f* (*f x*)
⟨*proof*⟩

55.7 stream repeating a single element

abbreviation *sconst* ≡ *siterate id*

lemma *shift-replicate-sconst*[*simp*]: *replicate n x* @− *sconst x* = *sconst x*
⟨*proof*⟩

lemma *sset-sconst*[*simp*]: *sset* (*sconst x*) = {*x*}
⟨*proof*⟩

lemma *sconst-alt*: *s* = *sconst x* ⟷ *sset s* = {*x*}
⟨*proof*⟩

lemma *sconst-cycle*: *sconst x = cycle [x]*
⟨*proof*⟩

lemma *smap-sconst*: *smap f (sconst x) = sconst (f x)*
⟨*proof*⟩

lemma *sconst-streams*: *x ∈ A ⟹ sconst x ∈ streams A*
⟨*proof*⟩

55.8 stream of natural numbers

abbreviation *fromN ≡ siterate Suc*

abbreviation *nats ≡ fromN 0*

lemma *sset-fromN*[*simp*]: *sset (fromN n) = {n ..}*
⟨*proof*⟩

lemma *stream-smap-fromN*: *s = smap (λj. let i = j − n in s !! i) (fromN n)*
⟨*proof*⟩

lemma *stream-smap-nats*: *s = smap (snth s) nats*
⟨*proof*⟩

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)*
⟨*proof*⟩

lemma *flat-Stream*[*simp*]: *xs ≠ [] ⟹ flat (xs ## ws) = xs @− flat ws*
⟨*proof*⟩

lemma *flat-unfold*: *shd ws ≠ [] ⟹ flat ws = shd ws @− flat (stl ws)*
⟨*proof*⟩

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)))
⟨*proof*⟩

lemma *sset-flat*[*simp*]: *∀ xs ∈ sset s. xs ≠ [] ⟹*
sset (flat s) = (⋃ xs ∈ sset s. set xs) (**is** *?P ⟹ ?L = ?R*)
⟨*proof*⟩

55.10 merge a stream of streams

definition *smerge* :: *'a stream stream ⇒ 'a stream* **where**
smerge ss = flat (smap (λn. map (λs. s !! n) (stake (Suc n) ss) @ stake n (ss !! n)) nats)

lemma *stake-nth*[*simp*]: *m < n ⟹ stake n s ! m = s !! m*
⟨*proof*⟩

lemma *snth-sset-smerge*: *ss !! n !! m ∈ sset (smerge ss)*
⟨*proof*⟩

lemma *sset-smerge*: *sset (smerge ss) = UNION (sset ss) sset*
⟨*proof*⟩

55.11 product of two streams

definition *sproduct* :: *'a stream ⇒ 'b stream ⇒ ('a × 'b) stream* **where**
sproduct s1 s2 = smerge (smap (λx. smap (Pair x) s2) s1)

lemma *sset-sproduct*: *sset (sproduct s1 s2) = sset s1 × sset s2*
⟨*proof*⟩

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
⟨*proof*⟩

lemma *sinterleave-snth*[*simp*]:
even n ⟹ sinterleave s1 s2 !! n = s1 !! (n div 2)
odd n ⟹ sinterleave s1 s2 !! n = s2 !! (n div 2)
⟨*proof*⟩

lemma *sset-sinterleave*: *sset (sinterleave s1 s2) = sset s1 ∪ sset s2*
⟨*proof*⟩

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)*
⟨*proof*⟩

lemma *snth-szip*[*simp*]: *szip s1 s2 !! n = (s1 !! n, s2 !! n)*
⟨*proof*⟩

lemma *stake-szip*[*simp*]:
stake n (szip s1 s2) = zip (stake n s1) (stake n s2)
⟨*proof*⟩

lemma *sdrop-szip*[*simp*]: *sdrop n (szip s1 s2) = szip (sdrop n s1) (sdrop n s2)*
⟨*proof*⟩

lemma *smap-szip-fst*:
smap (λx. f (fst x)) (szip s1 s2) = smap f s1
⟨*proof*⟩

lemma *smap-szip-snd*:
smap (λx. g (snd x)) (szip s1 s2) = smap g s2
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *smap2-szip*:
smap2 f s1 s2 = smap (case-prod f) (szip s1 s2)
⟨*proof*⟩

lemma *smap-smap2*[*simp*]:
smap f (smap2 g s1 s2) = smap2 (λx y. f (g x y)) s1 s2
⟨*proof*⟩

lemma *smap2-alt*:
(smap2 f s1 s2 = s) = (∀ n. f (s1 !! n) (s2 !! n) = s !! n)
⟨*proof*⟩

lemma *snth-smap2*[*simp*]:
smap2 f s1 s2 !! n = f (s1 !! n) (s2 !! n)
⟨*proof*⟩

lemma *stake-smap2*[*simp*]:
stake n (smap2 f s1 s2) = map (case-prod f) (zip (stake n s1) (stake n s2))
⟨*proof*⟩

lemma *sdrop-smap2*[*simp*]:

sdrop n (smap2 f s1 s2) = smap2 f (sdrop n s1) (sdrop n s2)
⟨*proof*⟩

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*
⟨*proof*⟩

interpretation *prefix-bot*: *order-bot Nil prefixeq prefix*
⟨*proof*⟩

lemma *prefixeqI* [*intro?*]: *ys* = *xs* @ *zs* ⟹ *prefixeq xs ys*
⟨*proof*⟩

lemma *prefixeqE* [*elim?*]:
assumes *prefixeq xs ys*
obtains *zs* **where** *ys* = *xs* @ *zs*
⟨*proof*⟩

lemma *prefixI'* [*intro?*]: *ys* = *xs* @ *z* # *zs* ⟹ *prefix xs ys*
⟨*proof*⟩

lemma *prefixE'* [*elim?*]:
assumes *prefix xs ys*
obtains *z zs* **where** *ys* = *xs* @ *z* # *zs*
⟨*proof*⟩

lemma *prefixI* [*intro?*]: *prefixeq xs ys* ⟹ *xs* ≠ *ys* ⟹ *prefix xs ys*
⟨*proof*⟩

lemma *prefixE* [*elim?*]:
fixes *xs ys* :: *'a list*
assumes *prefix xs ys*

obtains *prefixeq xs ys* **and** $xs \neq ys$
⟨*proof*⟩

56.2 Basic properties of prefixes

theorem *Nil-prefixeq* [*iff*]: *prefixeq* [] *xs*
⟨*proof*⟩

theorem *prefixeq-Nil* [*simp*]: (*prefixeq xs* []) = (*xs* = [])
⟨*proof*⟩

lemma *prefixeq-snoc* [*simp*]: *prefixeq xs* (*ys* @ [*y*]) ⟷ *xs* = *ys* @ [*y*] ∨ *prefixeq xs ys*
⟨*proof*⟩

lemma *Cons-prefixeq-Cons* [*simp*]: *prefixeq* (*x* # *xs*) (*y* # *ys*) = (*x* = *y* ∧ *prefixeq xs ys*)
⟨*proof*⟩

lemma *prefixeq-code* [*code*]:
prefixeq [] *xs* ⟷ *True*
prefixeq (*x* # *xs*) [] ⟷ *False*
prefixeq (*x* # *xs*) (*y* # *ys*) ⟷ *x* = *y* ∧ *prefixeq xs ys*
⟨*proof*⟩

lemma *same-prefixeq-prefixeq* [*simp*]: *prefixeq* (*xs* @ *ys*) (*xs* @ *zs*) = *prefixeq ys zs*
⟨*proof*⟩

lemma *same-prefixeq-nil* [*iff*]: *prefixeq* (*xs* @ *ys*) *xs* = (*ys* = [])
⟨*proof*⟩

lemma *prefixeq-prefixeq* [*simp*]: *prefixeq xs ys* ⟹ *prefixeq xs* (*ys* @ *zs*)
⟨*proof*⟩

lemma *append-prefixeqD*: *prefixeq* (*xs* @ *ys*) *zs* ⟹ *prefixeq xs zs*
⟨*proof*⟩

theorem *prefixeq-Cons*: *prefixeq xs* (*y* # *ys*) = (*xs* = [] ∨ (∃ *zs*. *xs* = *y* # *zs* ∧ *prefixeq zs ys*))
⟨*proof*⟩

theorem *prefixeq-append*:
prefixeq xs (*ys* @ *zs*) = (*prefixeq xs ys* ∨ (∃ *us*. *xs* = *ys* @ *us* ∧ *prefixeq us zs*))
⟨*proof*⟩

lemma *append-one-prefixeq*:
prefixeq xs ys ⟹ *length xs* < *length ys* ⟹ *prefixeq* (*xs* @ [*ys* ! *length xs*]) *ys*
⟨*proof*⟩

theorem *prefixeq-length-le*: *prefixeq xs ys* $\Longrightarrow$ *length xs* $\leq$ *length ys*
⟨*proof*⟩

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
⟨*proof*⟩

lemma *set-mono-prefixeq*: *prefixeq xs ys* $\Longrightarrow$ *set xs* $\subseteq$ *set ys*
⟨*proof*⟩

lemma *take-is-prefixeq*: *prefixeq* (*take n xs*) *xs*
⟨*proof*⟩

lemma *map-prefixeqI*: *prefixeq xs ys* $\Longrightarrow$ *prefixeq* (*map f xs*) (*map f ys*)
⟨*proof*⟩

lemma *prefixeq-length-less*: *prefix xs ys* $\Longrightarrow$ *length xs* $<$ *length ys*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *take-prefix*: *prefix xs ys* $\Longrightarrow$ *prefix* (*take n xs*) *ys*
⟨*proof*⟩

lemma *not-prefixeq-cases*:
assumes *pfx*: ¬ *prefixeq ps ls*
obtains
(*c1*) *ps* $\neq$ [] **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* $\neq$ *a*
⟨*proof*⟩

lemma *not-prefixeq-induct* [*consumes 1*, *case-names Nil Neq Eq*]:
assumes *np*: ¬ *prefixeq ps ls*
and *base*: $\bigwedge$*x xs*. *P* (*x*#*xs*) []
and *r1*: $\bigwedge$*x xs y ys*. *x* $\neq$ *y* $\Longrightarrow$ *P* (*x*#*xs*) (*y*#*ys*)
and *r2*: $\bigwedge$*x xs y ys*. ⟦ *x* = *y*; ¬ *prefixeq xs ys*; *P xs ys* ⟧ $\Longrightarrow$ *P* (*x*#*xs*) (*y*#*ys*)
shows *P ps ls* ⟨*proof*⟩

56.3 Parallel lists

definition *parallel* :: '*a list* $\Rightarrow$ '*a list* $\Rightarrow$ *bool* (**infixl** ∥ *50*)
where (*xs* ∥ *ys*) = (¬ *prefixeq xs ys* $\wedge$ ¬ *prefixeq ys xs*)

lemma *parallelI* [*intro*]: ¬ *prefixeq xs ys* ⟹ ¬ *prefixeq ys xs* ⟹ *xs* ‖ *ys*
⟨*proof*⟩

lemma *parallelE* [*elim*]:
assumes *xs* ‖ *ys*
obtains ¬ *prefixeq xs ys* ∧ ¬ *prefixeq ys xs*
⟨*proof*⟩

theorem *prefixeq-cases*:
obtains *prefixeq xs ys* | *prefix ys xs* | *xs* ‖ *ys*
⟨*proof*⟩

theorem *parallel-decomp*:
xs ‖ *ys* ⟹ ∃ *as b bs c cs*. *b* ≠ *c* ∧ *xs* = *as* @ *b* # *bs* ∧ *ys* = *as* @ *c* # *cs*
⟨*proof*⟩

lemma *parallel-append*: *a* ‖ *b* ⟹ *a* @ *c* ‖ *b* @ *d*
⟨*proof*⟩

lemma *parallel-appendI*: *xs* ‖ *ys* ⟹ *x* = *xs* @ *xs'* ⟹ *y* = *ys* @ *ys'* ⟹ *x* ‖ *y*
⟨*proof*⟩

lemma *parallel-commute*: *a* ‖ *b* ⟷ *b* ‖ *a*
⟨*proof*⟩

56.4 Suffix order on lists

definition *suffixeq* :: *'a list* ⇒ *'a list* ⇒ *bool*
where *suffixeq xs ys* = (∃ *zs*. *ys* = *zs* @ *xs*)

definition *suffix* :: *'a list* ⇒ *'a list* ⇒ *bool*
where *suffix xs ys* ⟷ (∃ *us*. *ys* = *us* @ *xs* ∧ *us* ≠ [])

lemma *suffix-imp-suffixeq*:
suffix xs ys ⟹ *suffixeq xs ys*
⟨*proof*⟩

lemma *suffixeqI* [*intro?*]: *ys* = *zs* @ *xs* ⟹ *suffixeq xs ys*
⟨*proof*⟩

lemma *suffixeqE* [*elim?*]:
assumes *suffixeq xs ys*
obtains *zs* **where** *ys* = *zs* @ *xs*
⟨*proof*⟩

lemma *suffixeq-refl* [*iff*]: *suffixeq xs xs*
⟨*proof*⟩
lemma *suffix-trans*:

suffix xs ys $\Longrightarrow$ *suffix ys zs* $\Longrightarrow$ *suffix xs zs*
⟨*proof*⟩
lemma *suffixeq-trans*: ⟦*suffixeq xs ys*; *suffixeq ys zs*⟧ $\Longrightarrow$ *suffixeq xs zs*
⟨*proof*⟩
lemma *suffixeq-antisym*: ⟦*suffixeq xs ys*; *suffixeq ys xs*⟧ $\Longrightarrow$ *xs* = *ys*
⟨*proof*⟩

lemma *suffixeq-tl* [*simp*]: *suffixeq* (*tl xs*) *xs*
⟨*proof*⟩

lemma *suffix-tl* [*simp*]: *xs* ≠ [] $\Longrightarrow$ *suffix* (*tl xs*) *xs*
⟨*proof*⟩

lemma *Nil-suffixeq* [*iff*]: *suffixeq* [] *xs*
⟨*proof*⟩
lemma *suffixeq-Nil* [*simp*]: (*suffixeq xs* []) = (*xs* = [])
⟨*proof*⟩

lemma *suffixeq-ConsI*: *suffixeq xs ys* $\Longrightarrow$ *suffixeq xs* (*y* # *ys*)
⟨*proof*⟩
lemma *suffixeq-ConsD*: *suffixeq* (*x* # *xs*) *ys* $\Longrightarrow$ *suffixeq xs ys*
⟨*proof*⟩

lemma *suffixeq-appendI*: *suffixeq xs ys* $\Longrightarrow$ *suffixeq xs* (*zs* @ *ys*)
⟨*proof*⟩
lemma *suffixeq-appendD*: *suffixeq* (*zs* @ *xs*) *ys* $\Longrightarrow$ *suffixeq xs ys*
⟨*proof*⟩

lemma *suffix-set-subset*:
suffix xs ys $\Longrightarrow$ *set xs* ⊆ *set ys* ⟨*proof*⟩

lemma *suffixeq-set-subset*:
suffixeq xs ys $\Longrightarrow$ *set xs* ⊆ *set ys* ⟨*proof*⟩

lemma *suffixeq-ConsD2*: *suffixeq* (*x* # *xs*) (*y* # *ys*) $\Longrightarrow$ *suffixeq xs ys*
⟨*proof*⟩

lemma *suffixeq-to-prefixeq* [*code*]: *suffixeq xs ys* ⟷ *prefixeq* (*rev xs*) (*rev ys*)
⟨*proof*⟩

lemma *distinct-suffixeq*: *distinct ys* $\Longrightarrow$ *suffixeq xs ys* $\Longrightarrow$ *distinct xs*
⟨*proof*⟩

lemma *suffixeq-map*: *suffixeq xs ys* $\Longrightarrow$ *suffixeq* (*map f xs*) (*map f ys*)
⟨*proof*⟩

lemma *suffixeq-drop*: *suffixeq* (*drop n as*) *as*
⟨*proof*⟩

lemma *suffixeq-take*: *suffixeq xs ys* $\Longrightarrow$ *ys* = *take* (*length ys* − *length xs*) *ys* @ *xs*
⟨*proof*⟩

lemma *suffixeq-suffix-reflclp-conv*: *suffixeq* = *suffix*$^{==}$
⟨*proof*⟩

lemma *parallelD1*: *x* ∥ *y* $\Longrightarrow$ ¬ *prefixeq x y*
⟨*proof*⟩

lemma *parallelD2*: *x* ∥ *y* $\Longrightarrow$ ¬ *prefixeq y x*
⟨*proof*⟩

lemma *parallel-Nil1* [*simp*]: ¬ *x* ∥ []
⟨*proof*⟩

lemma *parallel-Nil2* [*simp*]: ¬ [] ∥ *x*
⟨*proof*⟩

lemma *Cons-parallelI1*: *a* ≠ *b* $\Longrightarrow$ *a* # *as* ∥ *b* # *bs*
⟨*proof*⟩

lemma *Cons-parallelI2*: ⟦ *a* = *b*; *as* ∥ *bs* ⟧ $\Longrightarrow$ *a* # *as* ∥ *b* # *bs*
⟨*proof*⟩

lemma *not-equal-is-parallel*:
assumes *neq*: *xs* ≠ *ys*
and *len*: *length xs* = *length ys*
shows *xs* ∥ *ys*
⟨*proof*⟩

lemma *suffix-reflclp-conv*: *suffix*$^{==}$ = *suffixeq*
⟨*proof*⟩

lemma *suffix-lists*: *suffix xs ys* $\Longrightarrow$ *ys* ∈ *lists A* $\Longrightarrow$ *xs* ∈ *lists A*
⟨*proof*⟩

56.5 Homeomorphic embedding on lists

inductive *list-emb* :: ('*a* ⇒ '*a* ⇒ *bool*) ⇒ '*a list* ⇒ '*a list* ⇒ *bool*
for *P* :: ('*a* ⇒ '*a* ⇒ *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 ⋀*x y*. *P x y* ⟶ *Q x y*
shows *list-emb P xs ys* ⟶ *list-emb Q xs ys*
⟨*proof*⟩

lemma *list-emb-Nil2* [*simp*]:
assumes *list-emb P xs* [] **shows** *xs* = []
⟨*proof*⟩

lemma *list-emb-refl*:
assumes ⋀*x*. *x* ∈ *set xs* ⟹ *P x x*
shows *list-emb P xs xs*
⟨*proof*⟩

lemma *list-emb-Cons-Nil* [*simp*]: *list-emb P* (*x*#*xs*) [] = *False*
⟨*proof*⟩

lemma *list-emb-append2* [*intro*]: *list-emb P xs ys* ⟹ *list-emb P xs* (*zs* @ *ys*)
⟨*proof*⟩

lemma *list-emb-prefix* [*intro*]:
assumes *list-emb P xs ys* **shows** *list-emb P xs* (*ys* @ *zs*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *list-emb-suffix*:
assumes *list-emb P xs ys* **and** *suffix ys zs*
shows *list-emb P xs zs*
⟨*proof*⟩

lemma *list-emb-suffixeq*:
assumes *list-emb P xs ys* **and** *suffixeq ys zs*
shows *list-emb P xs zs*
⟨*proof*⟩

lemma *list-emb-length*: *list-emb P xs ys* ⟹ *length xs* ≤ *length ys*
⟨*proof*⟩

lemma *list-emb-trans*:
assumes ⋀*x y z*. ⟦*x* ∈ *set xs*; *y* ∈ *set ys*; *z* ∈ *set zs*; *P x y*; *P y z*⟧ ⟹ *P x z*
shows ⟦*list-emb P xs ys*; *list-emb P ys zs*⟧ ⟹ *list-emb P xs zs*
⟨*proof*⟩

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*
⟨*proof*⟩

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*) ⟨*proof*⟩

lemma *sublisteq-same-length*:
assumes *sublisteq xs ys* **and** *length xs* = *length ys* **shows** *xs* = *ys*
⟨*proof*⟩

lemma *not-sublisteq-length* [*simp*]: *length ys* < *length xs* ⟹ ¬ *sublisteq xs ys*
⟨*proof*⟩

lemma [*code*]:
list-emb P [] *ys* ⟷ *True*
list-emb P (*x*#*xs*) [] ⟷ *False*
⟨*proof*⟩

lemma *sublisteq-Cons′*: *sublisteq* (*x*#*xs*) *ys* ⟹ *sublisteq xs ys*
⟨*proof*⟩

lemma *sublisteq-Cons2′*:
assumes *sublisteq* (*x*#*xs*) (*x*#*ys*) **shows** *sublisteq xs ys*
⟨*proof*⟩

lemma *sublisteq-Cons2-neq*:
assumes *sublisteq* (*x*#*xs*) (*y*#*ys*)
shows *x* ≠ *y* ⟹ *sublisteq* (*x*#*xs*) *ys*
⟨*proof*⟩

lemma *sublisteq-Cons2-iff* [*simp*, *code*]:
sublisteq (*x*#*xs*) (*y*#*ys*) = (*if x* = *y then sublisteq xs ys else sublisteq* (*x*#*xs*) *ys*)
⟨*proof*⟩

lemma *sublisteq-append′*: *sublisteq* (*zs* @ *xs*) (*zs* @ *ys*) ⟷ *sublisteq xs ys*
⟨*proof*⟩

lemma *sublisteq-refl* [*simp*, *intro*!]: *sublisteq xs xs* ⟨*proof*⟩

lemma *sublisteq-antisym*:
assumes *sublisteq xs ys* **and** *sublisteq ys xs*
shows *xs* = *ys*
⟨*proof*⟩

lemma *sublisteq-trans*: *sublisteq xs ys* $\Longrightarrow$ *sublisteq ys zs* $\Longrightarrow$ *sublisteq xs zs*
⟨*proof*⟩

lemma *sublisteq-append-le-same-iff*: *sublisteq* (*xs* @ *ys*) *ys* $\longleftrightarrow$ *xs* = []
⟨*proof*⟩

lemma *list-emb-append-mono*:
⟦ *list-emb P xs xs'*; *list-emb P ys ys'* ⟧ $\Longrightarrow$ *list-emb P* (*xs*@*ys*) (*xs'*@*ys'*)
⟨*proof*⟩

56.7 Appending elements

lemma *sublisteq-append* [*simp*]:
sublisteq (*xs* @ *zs*) (*ys* @ *zs*) $\longleftrightarrow$ *sublisteq xs ys* (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *sublisteq-drop-many*: *sublisteq xs ys* $\Longrightarrow$ *sublisteq xs* (*zs* @ *ys*)
⟨*proof*⟩

lemma *sublisteq-rev-drop-many*: *sublisteq xs ys* $\Longrightarrow$ *sublisteq xs* (*ys* @ *zs*)
⟨*proof*⟩

56.8 Relation to standard list operations

lemma *sublisteq-map*:
assumes *sublisteq xs ys* **shows** *sublisteq* (*map f xs*) (*map f ys*)
⟨*proof*⟩

lemma *sublisteq-filter-left* [*simp*]: *sublisteq* (*filter P xs*) *xs*
⟨*proof*⟩

lemma *sublisteq-filter* [*simp*]:
assumes *sublisteq xs ys* **shows** *sublisteq* (*filter P xs*) (*filter P ys*)
⟨*proof*⟩

lemma *sublisteq xs ys* $\longleftrightarrow$ ($\exists$ *N*. *xs* = *sublist ys N*) (**is** *?L* = *?R*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *shift-prefix-cases*:
assumes *xl* @− *xs* = *yl* @− *ys*
shows *prefixeq xl yl* ∨ *prefixeq yl xl*
⟨*proof*⟩

59 Linear temporal logic

abbreviation (*input*) *IMPL* (**infix** *impl 60*)
where φ *impl* ψ ≡ λ *xs*. φ *xs* ⟶ ψ *xs*

abbreviation (*input*) *OR* (**infix** *or 60*)
where φ *or* ψ ≡ λ *xs*. φ *xs* ∨ ψ *xs*

abbreviation (*input*) *AND* (**infix** *aand 60*)
where φ *aand* ψ ≡ λ *xs*. φ *xs* ∧ ψ *xs*

abbreviation (*input*) *not* φ ≡ λ *xs*. ¬ φ *xs*

abbreviation (*input*) *true* ≡ λ *xs*. *True*

abbreviation (*input*) *false* ≡ λ *xs*. *False*

lemma *impl-not-or*: φ *impl* ψ = (*not* φ) *or* ψ
⟨*proof*⟩

lemma *not-or*: *not* (φ *or* ψ) = (*not* φ) *aand* (*not* ψ)
⟨*proof*⟩

lemma *not-aand*: *not* (φ *aand* ψ) = (*not* φ) *or* (*not* ψ)
⟨*proof*⟩

lemma *non-not*[*simp*]: *not* (*not* φ) = φ ⟨*proof*⟩

fun *holds* **where** *holds P xs* ⟷ *P* (*shd xs*)
fun *nxt* **where** *nxt* φ *xs* = φ (*stl xs*)

definition *HLD s* = *holds* (λ*x*. *x* ∈ *s*)

abbreviation *HLD-nxt* (**infixr** · *65*) **where**
s · *P* ≡ *HLD s aand nxt P*

context
notes [[*inductive-internals*]]
begin

inductive *ev* **for** φ **where**
base: $\varphi\ xs \Longrightarrow ev\ \varphi\ xs$
|
step: $ev\ \varphi\ (stl\ xs) \Longrightarrow ev\ \varphi\ xs$

coinductive *alw* **for** φ **where**
alw: $[\![\varphi\ xs;\ alw\ \varphi\ (stl\ xs)]\!] \Longrightarrow alw\ \varphi\ xs$

coinductive *UNTIL* (**infix** *until 60*) **for** $\varphi\ \psi$ **where**
base: $\psi\ xs \Longrightarrow (\varphi\ until\ \psi)\ xs$
|
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 $holds\ Q\ xs$
⟨*proof*⟩

lemma *holds-aand*:
$(holds\ P\ aand\ holds\ Q)\ steps \longleftrightarrow holds\ (\lambda\ step.\ P\ step \wedge Q\ step)\ steps$ ⟨*proof*⟩

lemma *HLD-iff*: $HLD\ s\ \omega \longleftrightarrow shd\ \omega \in s$
⟨*proof*⟩

lemma *HLD-Stream*[*simp*]: $HLD\ X\ (x\ \#\#\ \omega) \longleftrightarrow x \in X$
⟨*proof*⟩

lemma *nxt-mono*:
assumes *nxt*: $nxt\ \varphi\ xs$ **and** *0*: $\bigwedge xs.\ \varphi\ xs \Longrightarrow \psi\ xs$
shows $nxt\ \psi\ xs$
⟨*proof*⟩

declare *ev.intros*[*intro*]
declare *alw.cases*[*elim*]

lemma *ev-induct-strong*[*consumes 1*, *case-names base step*]:
$ev\ \varphi\ x \Longrightarrow (\bigwedge xs.\ \varphi\ xs \Longrightarrow P\ xs) \Longrightarrow (\bigwedge xs.\ ev\ \varphi\ (stl\ xs) \Longrightarrow \neg\ \varphi\ xs \Longrightarrow P\ (stl\ xs) \Longrightarrow P\ xs) \Longrightarrow P\ x$
⟨*proof*⟩

lemma *alw-coinduct*[*consumes 1*, *case-names alw stl*]:
$X\ x \Longrightarrow (\bigwedge x.\ X\ x \Longrightarrow \varphi\ x) \Longrightarrow (\bigwedge x.\ X\ x \Longrightarrow \neg\ alw\ \varphi\ (stl\ x) \Longrightarrow X\ (stl\ x)) \Longrightarrow alw\ \varphi\ x$
⟨*proof*⟩

lemma *ev-mono*:
assumes *ev*: *ev* φ *xs* **and** *0*: $\bigwedge$ *xs*. φ *xs* $\Longrightarrow$ ψ *xs*
shows *ev* ψ *xs*
⟨*proof*⟩

lemma *alw-mono*:
assumes *alw*: *alw* φ *xs* **and** *0*: $\bigwedge$ *xs*. φ *xs* $\Longrightarrow$ ψ *xs*
shows *alw* ψ *xs*
⟨*proof*⟩

lemma *until-monoL*:
assumes *until*: (φ*1 until* ψ) *xs* **and** *0*: $\bigwedge$ *xs*. φ*1 xs* $\Longrightarrow$ φ*2 xs*
shows (φ*2 until* ψ) *xs*
⟨*proof*⟩

lemma *until-monoR*:
assumes *until*: (φ *until* ψ*1*) *xs* **and** *0*: $\bigwedge$ *xs*. ψ*1 xs* $\Longrightarrow$ ψ*2 xs*
shows (φ *until* ψ*2*) *xs*
⟨*proof*⟩

lemma *until-mono*:
assumes *until*: (φ*1 until* ψ*1*) *xs* **and**
0: $\bigwedge$ *xs*. φ*1 xs* $\Longrightarrow$ φ*2 xs* $\bigwedge$ *xs*. ψ*1 xs* $\Longrightarrow$ ψ*2 xs*
shows (φ*2 until* ψ*2*) *xs*
⟨*proof*⟩

lemma *until-false*: φ *until false* = *alw* φ
⟨*proof*⟩

lemma *ev-nxt*: *ev* φ = (φ *or nxt* (*ev* φ))
⟨*proof*⟩

lemma *alw-nxt*: *alw* φ = (φ *aand nxt* (*alw* φ))
⟨*proof*⟩

lemma *ev-ev*[*simp*]: *ev* (*ev* φ) = *ev* φ
⟨*proof*⟩

lemma *alw-alw*[*simp*]: *alw* (*alw* φ) = *alw* φ
⟨*proof*⟩

lemma *ev-shift*:
assumes *ev* φ *xs*
shows *ev* φ (*xl* @− *xs*)
⟨*proof*⟩

lemma *ev-imp-shift*:
assumes *ev* φ *xs* **shows** $\exists$ *xl xs2*. *xs* = *xl* @− *xs2* $\wedge$ φ *xs2*
⟨*proof*⟩

lemma *alw-ev-shift*: $alw\ \varphi\ xs1 \Longrightarrow ev\ (alw\ \varphi)\ (xl\ @-\ xs1)$
⟨*proof*⟩

lemma *alw-shift*:
assumes $alw\ \varphi\ (xl\ @-\ xs)$
shows $alw\ \varphi\ xs$
⟨*proof*⟩

lemma *ev-ex-nxt*:
assumes $ev\ \varphi\ xs$
shows $\exists\ n.\ (nxt\ \hat{}\hat{}\ n)\ \varphi\ xs$
⟨*proof*⟩

lemma *alw-sdrop*:
assumes $alw\ \varphi\ xs$ **shows** $alw\ \varphi\ (sdrop\ n\ xs)$
⟨*proof*⟩

lemma *nxt-sdrop*: $(nxt\ \hat{}\hat{}\ n)\ \varphi\ xs \longleftrightarrow \varphi\ (sdrop\ n\ xs)$
⟨*proof*⟩

definition $wait\ \varphi\ xs \equiv LEAST\ n.\ (nxt\ \hat{}\hat{}\ n)\ \varphi\ xs$

lemma *nxt-wait*:
assumes $ev\ \varphi\ xs$ **shows** $(nxt\ \hat{}\hat{}\ (wait\ \varphi\ xs))\ \varphi\ xs$
⟨*proof*⟩

lemma *nxt-wait-least*:
assumes *ev*: $ev\ \varphi\ xs$ **and** *nxt*: $(nxt\ \hat{}\hat{}\ n)\ \varphi\ xs$ **shows** $wait\ \varphi\ xs \leq n$
⟨*proof*⟩

lemma *sdrop-wait*:
assumes $ev\ \varphi\ xs$ **shows** $\varphi\ (sdrop\ (wait\ \varphi\ xs)\ xs)$
⟨*proof*⟩

lemma *sdrop-wait-least*:
assumes *ev*: $ev\ \varphi\ xs$ **and** *nxt*: $\varphi\ (sdrop\ n\ xs)$ **shows** $wait\ \varphi\ xs \leq n$
⟨*proof*⟩

lemma *nxt-ev*: $(nxt\ \hat{}\hat{}\ n)\ \varphi\ xs \Longrightarrow ev\ \varphi\ xs$
⟨*proof*⟩

lemma *not-ev*: $not\ (ev\ \varphi) = alw\ (not\ \varphi)$
⟨*proof*⟩

lemma *not-alw*: $not\ (alw\ \varphi) = ev\ (not\ \varphi)$
⟨*proof*⟩

lemma *not-ev-not*[*simp*]: $not\ (ev\ (not\ \varphi)) = alw\ \varphi$

⟨*proof*⟩

lemma *not-alw-not*[*simp*]: *not* (*alw* (*not* φ)) = *ev* φ
⟨*proof*⟩

lemma *alw-ev-sdrop*:
assumes *alw* (*ev* φ) (*sdrop m xs*)
shows *alw* (*ev* φ) *xs*
⟨*proof*⟩

lemma *ev-alw-imp-alw-ev*:
assumes *ev* (*alw* φ) *xs* **shows** *alw* (*ev* φ) *xs*
⟨*proof*⟩

lemma *alw-aand*: *alw* (φ *aand* ψ) = *alw* φ *aand alw* ψ
⟨*proof*⟩

lemma *ev-or*: *ev* (φ *or* ψ) = *ev* φ *or ev* ψ
⟨*proof*⟩

lemma *ev-alw-aand*:
assumes φ: *ev* (*alw* φ) *xs* **and** ψ: *ev* (*alw* ψ) *xs*
shows *ev* (*alw* (φ *aand* ψ)) *xs*
⟨*proof*⟩

lemma *ev-alw-alw-impl*:
assumes *ev* (*alw* φ) *xs* **and** *alw* (*alw* φ *impl ev* ψ) *xs*
shows *ev* ψ *xs*
⟨*proof*⟩

lemma *ev-alw-stl*[*simp*]: *ev* (*alw* φ) (*stl x*) ⟷ *ev* (*alw* φ) *x*
⟨*proof*⟩

lemma *alw-alw-impl-ev*:
alw (*alw* φ *impl ev* ψ) = (*ev* (*alw* φ) *impl alw* (*ev* ψ)) (**is** *?A* = *?B*)
⟨*proof*⟩

lemma *ev-alw-impl*:
assumes *ev* φ *xs* **and** *alw* (φ *impl* ψ) *xs* **shows** *ev* ψ *xs*
⟨*proof*⟩

lemma *ev-alw-impl-ev*:
assumes *ev* φ *xs* **and** *alw* (φ *impl ev* ψ) *xs* **shows** *ev* ψ *xs*
⟨*proof*⟩

lemma *alw-mp*:
assumes *alw* φ *xs* **and** *alw* (φ *impl* ψ) *xs*
shows *alw* ψ *xs*
⟨*proof*⟩

lemma *all-imp-alw*:
assumes $\bigwedge xs.\ \varphi\ xs$ **shows** *alw* φ *xs*
⟨*proof*⟩

lemma *alw-impl-ev-alw*:
assumes *alw* (φ *impl ev* ψ) *xs*
shows *alw* (*ev* φ *impl ev* ψ) *xs*
⟨*proof*⟩

lemma *ev-holds-sset*:
ev (*holds P*) *xs* ⟷ ($\exists$ $x \in$ *sset xs*. *P x*) (**is** *?L* ⟷ *?R*)
⟨*proof*⟩

lemma *alw-invar*:
assumes φ *xs* **and** *alw* (φ *impl nxt* φ) *xs*
shows *alw* φ *xs*
⟨*proof*⟩

lemma *variance*:
assumes *1*: φ *xs* **and** *2*: *alw* (φ *impl* (ψ *or nxt* φ)) *xs*
shows (*alw* φ *or ev* ψ) *xs*
⟨*proof*⟩

lemma *ev-alw-imp-nxt*:
assumes *e*: *ev* φ *xs* **and** *a*: *alw* (φ *impl* (*nxt* φ)) *xs*
shows *ev* (*alw* φ) *xs*
⟨*proof*⟩

inductive *ev-at* :: ('*a stream* ⇒ *bool*) ⇒ *nat* ⇒ '*a stream* ⇒ *bool* **for** *P* :: '*a stream* ⇒ *bool* **where**
base: *P* ω ⟹ *ev-at P 0* ω
| *step*:¬ *P* ω ⟹ *ev-at P n* (*stl* ω) ⟹ *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)
⟨*proof*⟩

lemma *ev-at-HLD-imp-snth*: *ev-at* (*HLD X*) *n* ω ⟹ ω !! *n* ∈ *X*
⟨*proof*⟩

lemma *ev-at-HLD-single-imp-snth*: *ev-at* (*HLD* {*x*}) *n* ω ⟹ ω !! *n* = *x*
⟨*proof*⟩

lemma *ev-at-unique*: *ev-at P n* ω ⟹ *ev-at P m* ω ⟹ *n* = *m*

⟨*proof*⟩

lemma *ev-iff-ev-at*: *ev P ω* ⟷ (∃ *n*. *ev-at P n ω*)
⟨*proof*⟩

lemma *ev-at-shift*: *ev-at* (*HLD X*) *i* (*stake* (*Suc i*) *ω* @− *ω'* :: *'s stream*) ⟷ *ev-at* (*HLD X*) *i ω*
⟨*proof*⟩

lemma *ev-iff-ev-at-unqiue*: *ev P ω* ⟷ (∃!*n*. *ev-at P n ω*)
⟨*proof*⟩

lemma *alw-HLD-iff-streams*: *alw* (*HLD X*) *ω* ⟷ *ω* ∈ *streams X*
⟨*proof*⟩

lemma *not-HLD*: *not* (*HLD X*) = *HLD* (− *X*)
⟨*proof*⟩

lemma *not-alw-iff*: ¬ (*alw P ω*) ⟷ *ev* (*not P*) *ω*
⟨*proof*⟩

lemma *not-ev-iff*: ¬ (*ev P ω*) ⟷ *alw* (*not P*) *ω*
⟨*proof*⟩

lemma *ev-Stream*: *ev P* (*x* ## *s*) ⟷ *P* (*x* ## *s*) ∨ *ev P s*
⟨*proof*⟩

lemma *alw-ev-imp-ev-alw*:
assumes *alw* (*ev P*) *ω* **shows** *ev* (*P aand alw* (*ev P*)) *ω*
⟨*proof*⟩

lemma *ev-False*: *ev* (λ*x*. *False*) *ω* ⟷ *False*
⟨*proof*⟩

lemma *alw-False*: *alw* (λ*x*. *False*) *ω* ⟷ *False*
⟨*proof*⟩

lemma *ev-iff-sdrop*: *ev P ω* ⟷ (∃ *m*. *P* (*sdrop m ω*))
⟨*proof*⟩

lemma *alw-iff-sdrop*: *alw P ω* ⟷ (∀ *m*. *P* (*sdrop m ω*))
⟨*proof*⟩

lemma *infinite-iff-alw-ev*: *infinite* {*m*. *P* (*sdrop m ω*)} ⟷ *alw* (*ev P*) *ω*
⟨*proof*⟩

lemma *alw-inv*:
assumes *stl*: ⋀*s*. *f* (*stl s*) = *stl* (*f s*)
shows *alw P* (*f s*) ⟷ *alw* (λ*x*. *P* (*f x*)) *s*

⟨*proof*⟩

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*⟩

lemma *alw-smap*: $alw\ P\ (smap\ f\ s) \longleftrightarrow alw\ (\lambda x.\ P\ (smap\ f\ x))\ s$
⟨*proof*⟩

lemma *ev-smap*: $ev\ P\ (smap\ f\ s) \longleftrightarrow ev\ (\lambda x.\ P\ (smap\ f\ x))\ s$
⟨*proof*⟩

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*⟩

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*⟩

lemma *alwD*: $alw\ P\ x \Longrightarrow P\ x$
⟨*proof*⟩

lemma *alw-alwD*: $alw\ P\ \omega \Longrightarrow alw\ (alw\ P)\ \omega$
⟨*proof*⟩

lemma *alw-ev-stl*: $alw\ (ev\ P)\ (stl\ \omega) \longleftrightarrow alw\ (ev\ P)\ \omega$
⟨*proof*⟩

lemma *holds-Stream*: $holds\ P\ (x\ \#\#\ s) \longleftrightarrow P\ x$
⟨*proof*⟩

lemma *holds-eq1*[*simp*]: $holds\ (op = x) = HLD\ \{x\}$
⟨*proof*⟩

lemma *holds-eq2*[*simp*]: $holds\ (\lambda y.\ y = x) = HLD\ \{x\}$
⟨*proof*⟩

lemma *not-holds-eq*[*simp*]: $holds\ (-\ op = x) = not\ (HLD\ \{x\})$
⟨*proof*⟩

Strong until

context
notes [[*inductive-internals*]]
begin

inductive *suntil* (**infix** *suntil 60*) **for** φ ψ **where**
base: $\psi\ \omega \Longrightarrow (\varphi\ \mathit{suntil}\ \psi)\ \omega$
| *step*: $\varphi\ \omega \Longrightarrow (\varphi\ \mathit{suntil}\ \psi)\ (\mathit{stl}\ \omega) \Longrightarrow (\varphi\ \mathit{suntil}\ \psi)\ \omega$

inductive-simps *suntil-Stream*: $(\varphi\ \mathit{suntil}\ \psi)\ (x\ \#\#\ s)$

end

lemma *suntil-induct-strong*[*consumes 1*, *case-names base step*]:
$(\varphi\ \mathit{suntil}\ \psi)\ x \Longrightarrow$
$(\bigwedge\omega.\ \psi\ \omega \Longrightarrow P\ \omega) \Longrightarrow$
$(\bigwedge\omega.\ \varphi\ \omega \Longrightarrow \neg\ \psi\ \omega \Longrightarrow (\varphi\ \mathit{suntil}\ \psi)\ (\mathit{stl}\ \omega) \Longrightarrow P\ (\mathit{stl}\ \omega) \Longrightarrow P\ \omega) \Longrightarrow P\ x$
⟨*proof*⟩

lemma *ev-suntil*: $(\varphi\ \mathit{suntil}\ \psi)\ \omega \Longrightarrow \mathit{ev}\ \psi\ \omega$
⟨*proof*⟩

lemma *suntil-inv*:
assumes *stl*: $\bigwedge s.\ f\ (\mathit{stl}\ s) = \mathit{stl}\ (f\ s)$
shows $(P\ \mathit{suntil}\ Q)\ (f\ s) \longleftrightarrow ((\lambda x.\ P\ (f\ x))\ \mathit{suntil}\ (\lambda x.\ Q\ (f\ x)))\ s$
⟨*proof*⟩

lemma *suntil-smap*: $(P\ \mathit{suntil}\ Q)\ (\mathit{smap}\ f\ s) \longleftrightarrow ((\lambda x.\ P\ (\mathit{smap}\ f\ x))\ \mathit{suntil}\ (\lambda x.\ Q\ (\mathit{smap}\ f\ x)))\ s$
⟨*proof*⟩

lemma *hld-smap*: $\mathit{HLD}\ x\ (\mathit{smap}\ f\ s) = \mathit{holds}\ (\lambda y.\ f\ y \in x)\ s$
⟨*proof*⟩

lemma *suntil-mono*:
assumes *eq*: $\bigwedge\omega.\ P\ \omega \Longrightarrow \mathit{Q1}\ \omega \Longrightarrow \mathit{Q2}\ \omega\ \bigwedge\omega.\ P\ \omega \Longrightarrow \mathit{R1}\ \omega \Longrightarrow \mathit{R2}\ \omega$
assumes $*$: $(\mathit{Q1}\ \mathit{suntil}\ \mathit{R1})\ \omega\ \mathit{alw}\ P\ \omega$ **shows** $(\mathit{Q2}\ \mathit{suntil}\ \mathit{R2})\ \omega$
⟨*proof*⟩

lemma *suntil-cong*:
$\mathit{alw}\ P\ \omega \Longrightarrow (\bigwedge\omega.\ P\ \omega \Longrightarrow \mathit{Q1}\ \omega \longleftrightarrow \mathit{Q2}\ \omega) \Longrightarrow (\bigwedge\omega.\ P\ \omega \Longrightarrow \mathit{R1}\ \omega \longleftrightarrow \mathit{R2}\ \omega) \Longrightarrow$
$(\mathit{Q1}\ \mathit{suntil}\ \mathit{R1})\ \omega \longleftrightarrow (\mathit{Q2}\ \mathit{suntil}\ \mathit{R2})\ \omega$
⟨*proof*⟩

lemma *ev-suntil-iff*: $\mathit{ev}\ (P\ \mathit{suntil}\ Q)\ \omega \longleftrightarrow \mathit{ev}\ Q\ \omega$
⟨*proof*⟩

lemma *true-suntil*: $((\lambda\text{-}.\ \mathit{True})\ \mathit{suntil}\ P) = \mathit{ev}\ P$
⟨*proof*⟩

lemma *suntil-lfp*: $(\varphi\ \mathit{suntil}\ \psi) = \mathit{lfp}\ (\lambda P\ s.\ \psi\ s \vee (\varphi\ s \wedge P\ (\mathit{stl}\ s)))$
⟨*proof*⟩

lemma *sfilter-P*[*simp*]: $P\ (shd\ s) \Longrightarrow sfilter\ P\ s = shd\ s\ \#\#\ sfilter\ P\ (stl\ s)$
⟨*proof*⟩

lemma *sfilter-not-P*[*simp*]: $\neg\ P\ (shd\ s) \Longrightarrow sfilter\ P\ s = sfilter\ P\ (stl\ s)$
⟨*proof*⟩

lemma *sfilter-eq*:
assumes $ev\ (holds\ P)\ s$
shows $sfilter\ P\ s = x\ \#\#\ s' \longleftrightarrow$
$P\ x \wedge (not\ (holds\ P)\ suntil\ (HLD\ \{x\}\ aand\ nxt\ (\lambda s.\ sfilter\ P\ s = s')))\ s$
⟨*proof*⟩

lemma *sfilter-streams*:
$alw\ (ev\ (holds\ P))\ \omega \Longrightarrow \omega \in streams\ A \Longrightarrow sfilter\ P\ \omega \in streams\ \{x{\in}A.\ P\ x\}$
⟨*proof*⟩

lemma *alw-sfilter*:
assumes $*$: $alw\ (ev\ (holds\ P))\ s$
shows $alw\ Q\ (sfilter\ P\ s) \longleftrightarrow alw\ (\lambda x.\ Q\ (sfilter\ P\ x))\ s$
⟨*proof*⟩

lemma *ev-sfilter*:
assumes $*$: $alw\ (ev\ (holds\ P))\ s$
shows $ev\ Q\ (sfilter\ P\ s) \longleftrightarrow ev\ (\lambda x.\ Q\ (sfilter\ P\ x))\ s$
⟨*proof*⟩

lemma *holds-sfilter*:
assumes $ev\ (holds\ Q)\ s$ **shows** $holds\ P\ (sfilter\ Q\ s) \longleftrightarrow (not\ (holds\ Q)\ suntil\ (holds\ (Q\ aand\ P)))\ s$
⟨*proof*⟩

lemma *suntil-aand-nxt*:
$(\varphi\ suntil\ (\varphi\ aand\ nxt\ \psi))\ \omega \longleftrightarrow (\varphi\ aand\ nxt\ (\varphi\ suntil\ \psi))\ \omega$
⟨*proof*⟩

lemma *alw-sconst*: $alw\ P\ (sconst\ x) \longleftrightarrow P\ (sconst\ x)$
⟨*proof*⟩

lemma *ev-sconst*: $ev\ P\ (sconst\ x) \longleftrightarrow P\ (sconst\ x)$
⟨*proof*⟩

lemma *suntil-sconst*: $(\varphi\ suntil\ \psi)\ (sconst\ x) \longleftrightarrow \psi\ (sconst\ x)$
⟨*proof*⟩

lemma *hld-smap'*: $HLD\ x\ (smap\ f\ s) = HLD\ (f\ -`\ x)\ s$
⟨*proof*⟩

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 $x *_s xs \equiv map\ (op * x)\ xs$

lemma *scale1*[*simp*]: (*1*::'*a*::*monoid-mult*) $*_s$ *xs* = *xs*
⟨*proof*⟩

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 ⟨*proof*⟩

end

instantiation *list* :: ({*zero*, *uminus*}) *uminus*
begin

definition
list-uminus-def: *uminus* = *map uminus*

instance ⟨*proof*⟩

end

instantiation *list* :: ({*zero*,*minus*}) *minus*
begin

definition
list-diff-def: *op* − = *zipwith0* (*op* −)

instance ⟨*proof*⟩

end

lemma *zipwith0-Nil*[*simp*]: *zipwith0 f* [] *ys* = *map* (*f 0*) *ys*
⟨*proof*⟩

lemma *list-add-Nil*[*simp*]: [] + *xs* = (*xs*::'*a*::*monoid-add list*)
⟨*proof*⟩

lemma *list-add-Nil2*[*simp*]: *xs* + [] = (*xs*::'*a*::*monoid-add list*)
⟨*proof*⟩

lemma *list-add-Cons*[*simp*]: (*x*#*xs*) + (*y*#*ys*) = (*x*+*y*)#(*xs*+*ys*)
⟨*proof*⟩

lemma *list-diff-Nil*[*simp*]: [] − *xs* = −(*xs*::'*a*::*group-add list*)
⟨*proof*⟩

lemma *list-diff-Nil2*[*simp*]: *xs* − [] = (*xs*::'*a*::*group-add list*)
⟨*proof*⟩

lemma *list-diff-Cons-Cons*[*simp*]: (*x*#*xs*) − (*y*#*ys*) = (*x*−*y*)#(*xs*−*ys*)
⟨*proof*⟩

lemma *list-uminus-Cons*[*simp*]: −(*x*#*xs*) = (−*x*)#(−*xs*)
⟨*proof*⟩

lemma *self-list-diff*:
xs − *xs* = *replicate* (*length*(*xs*::'*a*::*group-add list*)) *0*
⟨*proof*⟩

lemma *list-add-assoc*: **fixes** *xs* :: '*a*::*monoid-add list*
shows (*xs*+*ys*)+*zs* = *xs*+(*ys*+*zs*)
⟨*proof*⟩

60.2 Inner product

definition *iprod* :: '*a*::*ring list* ⇒ '*a list* ⇒ '*a* (⟨-,-⟩) **where**
⟨*xs*,*ys*⟩ = (∑ (*x*,*y*) ← *zip xs ys*. *x*∗*y*)

lemma *iprod-Nil*[*simp*]: ⟨[],*ys*⟩ = *0*
⟨*proof*⟩

lemma *iprod-Nil2*[*simp*]: ⟨*xs*,[]⟩ = *0*
⟨*proof*⟩

lemma *iprod-Cons*[*simp*]: ⟨*x*#*xs*,*y*#*ys*⟩ = *x*∗*y* + ⟨*xs*,*ys*⟩

⟨*proof*⟩

lemma *iprod0-if-coeffs0*: ∀ *c*∈*set cs*. *c* = *0* ⟹ ⟨*cs*,*xs*⟩ = *0*
⟨*proof*⟩

lemma *iprod-uminus*[*simp*]: ⟨−*xs*,*ys*⟩ = −⟨*xs*,*ys*⟩
⟨*proof*⟩

lemma *iprod-left-add-distrib*: ⟨*xs* + *ys*,*zs*⟩ = ⟨*xs*,*zs*⟩ + ⟨*ys*,*zs*⟩
⟨*proof*⟩

lemma *iprod-left-diff-distrib*: ⟨*xs* − *ys*, *zs*⟩ = ⟨*xs*,*zs*⟩ − ⟨*ys*,*zs*⟩
⟨*proof*⟩

lemma *iprod-assoc*: ⟨*x* ∗$_s$ *xs*, *ys*⟩ = *x* ∗ ⟨*xs*,*ys*⟩
⟨*proof*⟩

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* ⇒ *'a*::*ord* ⇒ *bool* (**infixl** ∗<= *70*)
where *S* ∗<= *x* = (*ALL y*: *S*. *y* ≤ *x*)

definition *setge* :: *'a*::*ord* ⇒ *'a set* ⇒ *bool* (**infixl** <=∗ *70*)
where *x* <=∗ *S* = (*ALL y*: *S*. *x* ≤ *y*)

61.1 Rules for the Relations ∗<= and <=∗

lemma *setleI*: *ALL y*: *S*. *y* ≤ *x* ⟹ *S* ∗<= *x*
⟨*proof*⟩

lemma *setleD*: *S* ∗<= *x* ⟹ *y*: *S* ⟹ *y* ≤ *x*
⟨*proof*⟩

lemma *setgeI*: *ALL y*: *S*. *x* ≤ *y* ⟹ *x* <=∗ *S*
⟨*proof*⟩

lemma *setgeD*: *x* <=∗ *S* ⟹ *y*: *S* ⟹ *x* ≤ *y*
⟨*proof*⟩

definition *leastP* :: $('a \Rightarrow bool) \Rightarrow 'a::ord \Rightarrow bool$
where $leastP\ P\ x = (P\ x \wedge x <=* Collect\ P)$

definition *isUb* :: $'a\ set \Rightarrow 'a\ set \Rightarrow 'a::ord \Rightarrow bool$
where $isUb\ R\ S\ x = (S\ *<=\ x \wedge x{:}\ R)$

definition *isLub* :: $'a\ set \Rightarrow 'a\ set \Rightarrow 'a::ord \Rightarrow bool$
where $isLub\ R\ S\ x = leastP\ (isUb\ R\ S)\ x$

definition *ubs* :: $'a\ set \Rightarrow 'a::ord\ set \Rightarrow 'a\ set$
where $ubs\ R\ S = Collect\ (isUb\ R\ S)$

61.2 Rules about the Operators *leastP*, *ub* and *lub*

lemma *leastPD1*: $leastP\ P\ x \Longrightarrow P\ x$
⟨*proof*⟩

lemma *leastPD2*: $leastP\ P\ x \Longrightarrow x <=* Collect\ P$
⟨*proof*⟩

lemma *leastPD3*: $leastP\ P\ x \Longrightarrow y{:}\ Collect\ P \Longrightarrow x \leq y$
⟨*proof*⟩

lemma *isLubD1*: $isLub\ R\ S\ x \Longrightarrow S\ *<=\ x$
⟨*proof*⟩

lemma *isLubD1a*: $isLub\ R\ S\ x \Longrightarrow x{:}\ R$
⟨*proof*⟩

lemma *isLub-isUb*: $isLub\ R\ S\ x \Longrightarrow isUb\ R\ S\ x$
⟨*proof*⟩

lemma *isLubD2*: $isLub\ R\ S\ x \Longrightarrow y : S \Longrightarrow y \leq x$
⟨*proof*⟩

lemma *isLubD3*: $isLub\ R\ S\ x \Longrightarrow leastP\ (isUb\ R\ S)\ x$
⟨*proof*⟩

lemma *isLubI1*: $leastP(isUb\ R\ S)\ x \Longrightarrow isLub\ R\ S\ x$
⟨*proof*⟩

lemma *isLubI2*: $isUb\ R\ S\ x \Longrightarrow x <=* Collect\ (isUb\ R\ S) \Longrightarrow isLub\ R\ S\ x$
⟨*proof*⟩

lemma *isUbD*: $isUb\ R\ S\ x \Longrightarrow y : S \Longrightarrow y \leq x$
⟨*proof*⟩

lemma *isUbD2*: $isUb\ R\ S\ x \Longrightarrow S\ *<=\ x$
⟨*proof*⟩

lemma *isUbD2a*: *isUb R S x* $\Longrightarrow$ *x*: *R*
⟨*proof*⟩

lemma *isUbI*: *S* *<= *x* $\Longrightarrow$ *x*: *R* $\Longrightarrow$ *isUb R S x*
⟨*proof*⟩

lemma *isLub-le-isUb*: *isLub R S x* $\Longrightarrow$ *isUb R S y* $\Longrightarrow$ *x* $\leq$ *y*
⟨*proof*⟩

lemma *isLub-ubs*: *isLub R S x* $\Longrightarrow$ *x* <=* *ubs R S*
⟨*proof*⟩

lemma *isLub-unique*: [| *isLub R S x*; *isLub R S y* |] ==> *x* = (*y*::'*a*::*linorder*)
⟨*proof*⟩

lemma *isUb-UNIV-I*: ($\bigwedge$*y*. *y* $\in$ *S* $\Longrightarrow$ *y* $\leq$ *u*) $\Longrightarrow$ *isUb UNIV S u*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *greatestPD2*: *greatestP P x* $\Longrightarrow$ *Collect P* *<= *x*
⟨*proof*⟩

lemma *greatestPD3*: *greatestP P x* $\Longrightarrow$ *y*: *Collect P* $\Longrightarrow$ *x* $\geq$ *y*
⟨*proof*⟩

lemma *isGlbD1*: *isGlb R S x* $\Longrightarrow$ *x* <=* *S*
⟨*proof*⟩

lemma *isGlbD1a*: *isGlb R S x* $\Longrightarrow$ *x*: *R*
⟨*proof*⟩

lemma *isGlb-isLb*: $isGlb\ R\ S\ x \Longrightarrow isLb\ R\ S\ x$
⟨*proof*⟩

lemma *isGlbD2*: $isGlb\ R\ S\ x \Longrightarrow y : S \Longrightarrow y \geq x$
⟨*proof*⟩

lemma *isGlbD3*: $isGlb\ R\ S\ x \Longrightarrow greatestP\ (isLb\ R\ S)\ x$
⟨*proof*⟩

lemma *isGlbI1*: $greatestP\ (isLb\ R\ S)\ x \Longrightarrow isGlb\ R\ S\ x$
⟨*proof*⟩

lemma *isGlbI2*: $isLb\ R\ S\ x \Longrightarrow Collect\ (isLb\ R\ S)\ *<=\ x \Longrightarrow isGlb\ R\ S\ x$
⟨*proof*⟩

lemma *isLbD*: $isLb\ R\ S\ x \Longrightarrow y : S \Longrightarrow y \geq x$
⟨*proof*⟩

lemma *isLbD2*: $isLb\ R\ S\ x \Longrightarrow x\ <=*\ S$
⟨*proof*⟩

lemma *isLbD2a*: $isLb\ R\ S\ x \Longrightarrow x{:}\ R$
⟨*proof*⟩

lemma *isLbI*: $x\ <=*\ S \Longrightarrow x{:}\ R \Longrightarrow isLb\ R\ S\ x$
⟨*proof*⟩

lemma *isGlb-le-isLb*: $isGlb\ R\ S\ x \Longrightarrow isLb\ R\ S\ y \Longrightarrow x \geq y$
⟨*proof*⟩

lemma *isGlb-ubs*: $isGlb\ R\ S\ x \Longrightarrow lbs\ R\ S\ *<=\ x$
⟨*proof*⟩

lemma *isGlb-unique*: $[|\ isGlb\ R\ S\ x;\ isGlb\ R\ S\ y\ |] ==> x = (y::'a::linorder)$
⟨*proof*⟩

lemma *bdd-above-setle*: $bdd\text{-}above\ A \longleftrightarrow (\exists\, a.\ A\ *<=\ a)$
⟨*proof*⟩

lemma *bdd-below-setge*: $bdd\text{-}below\ A \longleftrightarrow (\exists\, a.\ a\ <=*\ A)$
⟨*proof*⟩

lemma *isLub-cSup*:
$(S::'a :: conditionally\text{-}complete\text{-}lattice\ set) \neq \{\} \Longrightarrow (\exists\, b.\ S\ *<=\ b) \Longrightarrow isLub\ UNIV\ S\ (Sup\ S)$
⟨*proof*⟩

lemma *isGlb-cInf*:
$(S::'a :: conditionally\text{-}complete\text{-}lattice\ set) \neq \{\} \Longrightarrow (\exists\, b.\ b\ <=*\ S) \Longrightarrow isGlb$

UNIV S (*Inf S*)
⟨*proof*⟩

lemma *cSup-le*: (*S*::'*a*::*conditionally-complete-lattice set*) ≠ {} ⟹ *S* *<= *b* ⟹ *Sup S* ≤ *b*
⟨*proof*⟩

lemma *cInf-ge*: (*S*::'*a* :: *conditionally-complete-lattice set*) ≠ {} ⟹ *b* <=* *S* ⟹ *Inf S* ≥ *b*
⟨*proof*⟩

lemma *cSup-bounds*:
fixes *S* :: '*a* :: *conditionally-complete-lattice set*
shows *S* ≠ {} ⟹ *a* <=* *S* ⟹ *S* *<= *b* ⟹ *a* ≤ *Sup S* ∧ *Sup S* ≤ *b*
⟨*proof*⟩

lemma *cSup-unique*: (*S*::'*a* :: {*conditionally-complete-linorder*, *no-bot*} *set*) *<= *b* ⟹ (∀ *b'*<*b*. ∃ *x*∈*S*. *b'* < *x*) ⟹ *Sup S* = *b*
⟨*proof*⟩

lemma *cInf-unique*: *b* <=* (*S*::'*a* :: {*conditionally-complete-linorder*, *no-top*} *set*) ⟹ (∀ *b'*>*b*. ∃ *x*∈*S*. *b'* > *x*) ⟹ *Inf S* = *b*
⟨*proof*⟩

Use completeness of reals (supremum property) to show that any bounded sequence has a least upper bound

lemma *reals-complete*: ∃ *X*. *X* ∈ *S* ⟹ ∃ *Y*. *isUb* (*UNIV*::*real set*) *S Y* ⟹ ∃ *t*. *isLub* (*UNIV* :: *real set*) *S t*
⟨*proof*⟩

lemma *Bseq-isUb*: ⋀*X* :: *nat* ⇒ *real*. *Bseq X* ⟹ ∃ *U*. *isUb* (*UNIV*::*real set*) {*x*. ∃ *n*. *X n* = *x*} *U*
⟨*proof*⟩

lemma *Bseq-isLub*: ⋀*X* :: *nat* ⇒ *real*. *Bseq X* ⟹ ∃ *U*. *isLub* (*UNIV*::*real set*) {*x*. ∃ *n*. *X n* = *x*} *U*
⟨*proof*⟩

lemma *isLub-mono-imp-LIMSEQ*:
fixes *X* :: *nat* ⇒ *real*
assumes *u*: *isLub UNIV* {*x*. ∃ *n*. *X n* = *x*} *u*
assumes *X*: ∀ *m n*. *m* ≤ *n* ⟶ *X m* ≤ *X n*
shows *X* ⟶ *u*
⟨*proof*⟩

lemmas *real-isGlb-unique* = *isGlb-unique*[**where** '*a*=*real*]

lemma *real-le-inf-subset*: *t* ≠ {} ⟹ *t* ⊆ *s* ⟹ ∃ *b*. *b* <=* *s* ⟹ *Inf s* ≤ *Inf* (*t*::*real set*)

⟨*proof*⟩

lemma *real-ge-sup-subset*: $t \neq \{\} \Longrightarrow t \subseteq s \Longrightarrow \exists\, b.\ s\ *<=\ b \Longrightarrow Sup\ s \geq Sup\ (t::real\ set)$
⟨*proof*⟩

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
$map\text{-}of\ xs = foldr\ (\lambda(k,\ v)\ m.\ m(k \mapsto v))\ xs\ Map.empty$
⟨*proof*⟩

context
begin

interpretation *lifting-syntax* ⟨*proof*⟩

lemma *empty-parametric*:
$(A\ ===>\ rel\text{-}option\ B)\ Map.empty\ Map.empty$
⟨*proof*⟩

lemma *lookup-parametric*: $((A\ ===>\ B)\ ===>\ A\ ===>\ B)\ (\lambda m\ k.\ m\ k)\ (\lambda m\ k.\ m\ k)$
⟨*proof*⟩

lemma *update-parametric*:
assumes [*transfer-rule*]: *bi-unique A*
shows $(A\ ===>\ B\ ===>\ (A\ ===>\ rel\text{-}option\ B)\ ===>\ A\ ===>\ rel\text{-}option\ B)$
$(\lambda k\ v\ m.\ m(k \mapsto v))\ (\lambda k\ v\ m.\ m(k \mapsto v))$
⟨*proof*⟩

lemma *delete-parametric*:
assumes [*transfer-rule*]: *bi-unique A*
shows $(A\ ===>\ (A\ ===>\ rel\text{-}option\ B)\ ===>\ A\ ===>\ rel\text{-}option\ B)$
$(\lambda k\ m.\ m(k := None))\ (\lambda k\ m.\ m(k := None))$
⟨*proof*⟩

lemma *is-none-parametric* [*transfer-rule*]:
$(rel\text{-}option\ A\ ===>\ HOL.eq)\ Option.is\text{-}none\ Option.is\text{-}none$
⟨*proof*⟩

lemma *dom-parametric*:
assumes [*transfer-rule*]: *bi-total A*
shows ((*A* ===> *rel-option B*) ===> *rel-set A*) *dom dom*
⟨*proof*⟩

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*
⟨*proof*⟩

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*))))
⟨*proof*⟩

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*)))
⟨*proof*⟩

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*⟩

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*))
⟨*proof*⟩

end

62.2 Type definition and primitive operations

typedef ('*a*, '*b*) *mapping* = *UNIV* :: ('*a* ⇀ '*b*) *set*
morphisms *rep Mapping*
⟨*proof*⟩

setup-lifting *type-definition-mapping*

lift-definition *empty* :: ('*a*, '*b*) *mapping*
is *Map.empty* **parametric** *empty-parametric* ⟨*proof*⟩

lift-definition *lookup* :: ('*a*, '*b*) *mapping* ⇒ '*a* ⇒ '*b option*
is λ*m k*. *m k* **parametric** *lookup-parametric* ⟨*proof*⟩

declare [[*code drop*: *Mapping.lookup*]]
⟨*ML*⟩

lift-definition *update* :: '*a* ⇒ '*b* ⇒ ('*a*, '*b*) *mapping* ⇒ ('*a*, '*b*) *mapping*
is λ*k v m*. *m*(*k* ↦ *v*) **parametric** *update-parametric* ⟨*proof*⟩

lift-definition *delete* :: '*a* ⇒ ('*a*, '*b*) *mapping* ⇒ ('*a*, '*b*) *mapping*
is λ*k m*. *m*(*k* := *None*) **parametric** *delete-parametric* ⟨*proof*⟩

lift-definition *keys* :: ('*a*, '*b*) *mapping* ⇒ '*a set*
is *dom* **parametric** *dom-parametric* ⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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* ⟨*proof*⟩

declare [[*code drop*: *map*]]

62.3 Functorial structure

functor *map*: *map*
⟨*proof*⟩

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
λk f m. (case m k of None ⇒ m
| Some v ⇒ m (k ↦ (f v))) **parametric** *map-entry-parametric* ⟨*proof*⟩

lemma *map-entry-code* [*code*]:
map-entry k f m = (case lookup m k of None ⇒ m
| Some v ⇒ update k (f v) m)
⟨*proof*⟩

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
⟨*proof*⟩

end

context
begin

interpretation *lifting-syntax* ⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

end

62.5 Properties

lemma *lookup-update*:
lookup (*update k v m*) *k* = *Some v*
⟨*proof*⟩

lemma *lookup-update-neq*:
k ≠ *k'* ⟹ *lookup* (*update k v m*) *k'* = *lookup m k'*
⟨*proof*⟩

lemma *lookup-empty*:
lookup empty k = *None*
⟨*proof*⟩

lemma *keys-is-none-rep* [*code-unfold*]:
k ∈ *keys m* ⟷ ¬ (*Option.is-none* (*lookup m k*))
⟨*proof*⟩

lemma *update-update*:
update k v (*update k w m*) = *update k v m*
k ≠ *l* ⟹ *update k v* (*update l w m*) = *update l w* (*update k v m*)
⟨*proof*⟩

lemma *update-delete* [*simp*]:
update k v (*delete k m*) = *update k v m*
⟨*proof*⟩

lemma *delete-update*:
delete k (*update k v m*) = *delete k m*
k ≠ *l* ⟹ *delete k* (*update l v m*) = *update l v* (*delete k m*)
⟨*proof*⟩

lemma *delete-empty* [*simp*]:
delete k empty = *empty*
⟨*proof*⟩

lemma *replace-update*:
k ∉ *keys m* ⟹ *replace k v m* = *m*
k ∈ *keys m* ⟹ *replace k v m* = *update k v m*

⟨*proof*⟩

lemma *size-empty* [*simp*]:
size empty = 0
⟨*proof*⟩

lemma *size-update*:
finite (*keys m*) ⟹ *size* (*update k v m*) =
(*if k* ∈ *keys m then size m else Suc* (*size m*))
⟨*proof*⟩

lemma *size-delete*:
size (*delete k m*) = (*if k* ∈ *keys m then size m* − *1 else size m*)
⟨*proof*⟩

lemma *size-tabulate* [*simp*]:
size (*tabulate ks f*) = *length* (*remdups ks*)
⟨*proof*⟩

lemma *bulkload-tabulate*:
bulkload xs = *tabulate* [*0*..<*length xs*] (*nth xs*)
⟨*proof*⟩

lemma *is-empty-empty* [*simp*]:
is-empty empty
⟨*proof*⟩

lemma *is-empty-update* [*simp*]:
¬ *is-empty* (*update k v m*)
⟨*proof*⟩

lemma *is-empty-delete*:
is-empty (*delete k m*) ⟷ *is-empty m* ∨ *keys m* = {*k*}
⟨*proof*⟩

lemma *is-empty-replace* [*simp*]:
is-empty (*replace k v m*) ⟷ *is-empty m*
⟨*proof*⟩

lemma *is-empty-default* [*simp*]:
¬ *is-empty* (*default k v m*)
⟨*proof*⟩

lemma *is-empty-map-entry* [*simp*]:
is-empty (*map-entry k f m*) ⟷ *is-empty m*
⟨*proof*⟩

lemma *is-empty-map-default* [*simp*]:
¬ *is-empty* (*map-default k v f m*)

⟨*proof*⟩

lemma *keys-dom-lookup*:
keys m = *dom* (*Mapping.lookup m*)
⟨*proof*⟩

lemma *keys-empty* [*simp*]:
keys empty = {}
⟨*proof*⟩

lemma *keys-update* [*simp*]:
keys (*update k v m*) = *insert k* (*keys m*)
⟨*proof*⟩

lemma *keys-delete* [*simp*]:
keys (*delete k m*) = *keys m* − {*k*}
⟨*proof*⟩

lemma *keys-replace* [*simp*]:
keys (*replace k v m*) = *keys m*
⟨*proof*⟩

lemma *keys-default* [*simp*]:
keys (*default k v m*) = *insert k* (*keys m*)
⟨*proof*⟩

lemma *keys-map-entry* [*simp*]:
keys (*map-entry k f m*) = *keys m*
⟨*proof*⟩

lemma *keys-map-default* [*simp*]:
keys (*map-default k v f m*) = *insert k* (*keys m*)
⟨*proof*⟩

lemma *keys-tabulate* [*simp*]:
keys (*tabulate ks f*) = *set ks*
⟨*proof*⟩

lemma *keys-bulkload* [*simp*]:
keys (*bulkload xs*) = {*0..<length xs*}
⟨*proof*⟩

lemma *distinct-ordered-keys* [*simp*]:
distinct (*ordered-keys m*)
⟨*proof*⟩

lemma *ordered-keys-infinite* [*simp*]:
¬ *finite* (*keys m*) ⟹ *ordered-keys m* = []
⟨*proof*⟩

lemma *ordered-keys-empty* [*simp*]:
ordered-keys empty = []
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *ordered-keys-delete* [*simp*]:
ordered-keys (*delete k m*) = *remove1 k* (*ordered-keys m*)
⟨*proof*⟩

lemma *ordered-keys-replace* [*simp*]:
ordered-keys (*replace k v m*) = *ordered-keys m*
⟨*proof*⟩

lemma *ordered-keys-default* [*simp*]:
k ∈ *keys m* ⟹ *ordered-keys* (*default k v m*) = *ordered-keys m*
finite (*keys m*) ⟹ *k* ∉ *keys m* ⟹ *ordered-keys* (*default k v m*) = *insort k* (*ordered-keys m*)
⟨*proof*⟩

lemma *ordered-keys-map-entry* [*simp*]:
ordered-keys (*map-entry k f m*) = *ordered-keys m*
⟨*proof*⟩

lemma *ordered-keys-map-default* [*simp*]:
k ∈ *keys m* ⟹ *ordered-keys* (*map-default k v f m*) = *ordered-keys m*
finite (*keys m*) ⟹ *k* ∉ *keys m* ⟹ *ordered-keys* (*map-default k v f m*) = *insort k* (*ordered-keys m*)
⟨*proof*⟩

lemma *ordered-keys-tabulate* [*simp*]:
ordered-keys (*tabulate ks f*) = *sort* (*remdups ks*)
⟨*proof*⟩

lemma *ordered-keys-bulkload* [*simp*]:
ordered-keys (*bulkload ks*) = [*0*..<*length ks*]
⟨*proof*⟩

lemma *tabulate-fold*:
tabulate xs f = *fold* (λ*k m*. *update k* (*f k*) *m*) *xs empty*
⟨*proof*⟩

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*⟩

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* ⇒ *'c*] ⇒ *'d* (**infixr** ≫= *54*)

notation (*ASCII*)
bind (**infixr** >>= *54*)

abbreviation (*do-notation*)
bind-do :: [*'a*, *'b* ⇒ *'c*] ⇒ *'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*
⟨*proof*⟩

setup-lifting *type-definition-multiset*

lemma *multiset-eq-iff*: *M* = *N* ⟷ (∀ *a*. *count M a* = *count N a*)
⟨*proof*⟩

lemma *multiset-eqI*: $(\bigwedge x.\ count\ A\ x = count\ B\ x) \Longrightarrow A = B$
⟨*proof*⟩

Preservation of the representing set *multiset*.

lemma *const0-in-multiset*: $(\lambda a.\ 0) \in multiset$
⟨*proof*⟩

lemma *only1-in-multiset*: $(\lambda b.\ if\ b = a\ then\ n\ else\ 0) \in multiset$
⟨*proof*⟩

lemma *union-preserves-multiset*: $M \in multiset \Longrightarrow N \in multiset \Longrightarrow (\lambda a.\ M\ a + N\ a) \in multiset$
⟨*proof*⟩

lemma *diff-preserves-multiset*:
assumes $M \in multiset$
shows $(\lambda a.\ M\ a - N\ a) \in multiset$
⟨*proof*⟩

lemma *filter-preserves-multiset*:
assumes $M \in multiset$
shows $(\lambda x.\ if\ P\ x\ then\ M\ x\ else\ 0) \in multiset$
⟨*proof*⟩

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** $\lambda a.\ 0$
⟨*proof*⟩

abbreviation *Mempty* :: *'a multiset* ({#}) **where**
Mempty ≡ *0*

lift-definition *plus-multiset* :: *'a multiset* ⇒ *'a multiset* ⇒ *'a multiset* **is** $\lambda M\ N.\ (\lambda a.\ M\ a + N\ a)$
⟨*proof*⟩

lift-definition *minus-multiset* :: *'a multiset* ⇒ *'a multiset* ⇒ *'a multiset* **is** $\lambda\ M\ N.\ \lambda a.\ M\ a - N\ a$
⟨*proof*⟩

instance
⟨*proof*⟩

end

lift-definition *single* :: *'a ⇒ 'a multiset* **is** *λa b. if b = a then 1 else 0*
⟨*proof*⟩

syntax
-multiset :: *args ⇒ 'a multiset* ({#(-)#})
translations
{#*x*, *xs*#} == {#*x*#} + {#*xs*#}
{#*x*#} == *CONST single x*

lemma *count-empty* [*simp*]: *count* {#} *a* = *0*
⟨*proof*⟩

lemma *count-single* [*simp*]: *count* {#*b*#} *a* = (*if b = a then 1 else 0*)
⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *not-in-iff*:
  x ∉# *M* ⟷ *count M x* = *0*
  ⟨*proof*⟩

lemma *count-greater-zero-iff* [*simp*]:
  count M x > *0* ⟷ *x* ∈# *M*
  ⟨*proof*⟩

lemma *count-inI*:
  assumes *count M x* = *0* ⟹ *False*
  shows *x* ∈# *M*
⟨*proof*⟩

lemma *in-countE*:
  assumes *x* ∈# *M*
  obtains *n* **where** *count M x* = *Suc n*
⟨*proof*⟩

lemma *count-greater-eq-Suc-zero-iff* [*simp*]:
  count M x ≥ *Suc 0* ⟷ *x* ∈# *M*
  ⟨*proof*⟩

lemma *count-greater-eq-one-iff* [*simp*]:
  count M x ≥ *1* ⟷ *x* ∈# *M*
  ⟨*proof*⟩

lemma *set-mset-empty* [*simp*]:
set-mset {#} = {}
⟨*proof*⟩

lemma *set-mset-single* [*simp*]:
set-mset {#*b*#} = {*b*}
⟨*proof*⟩

lemma *set-mset-eq-empty-iff* [*simp*]:
set-mset M = {} ⟷ *M* = {#}
⟨*proof*⟩

lemma *finite-set-mset* [*iff*]:
finite (*set-mset M*)
⟨*proof*⟩

65.3.2 Union

lemma *count-union* [*simp*]:
count (*M* + *N*) *a* = *count M a* + *count N a*
⟨*proof*⟩

lemma *set-mset-union* [*simp*]:
set-mset (*M* + *N*) = *set-mset M* ∪ *set-mset N*
⟨*proof*⟩

65.3.3 Difference

instance *multiset* :: (*type*) *comm-monoid-diff*
⟨*proof*⟩

lemma *count-diff* [*simp*]:
count (*M* − *N*) *a* = *count M a* − *count N a*
⟨*proof*⟩

lemma *in-diff-count*:
a ∈# *M* − *N* ⟷ *count N a* < *count M a*
⟨*proof*⟩

lemma *count-in-diffI*:
assumes ⋀*n*. *count N x* = *n* + *count M x* ⟹ *False*
shows *x* ∈# *M* − *N*
⟨*proof*⟩

lemma *in-diff-countE*:
assumes *x* ∈# *M* − *N*
obtains *n* **where** *count M x* = *Suc n* + *count N x*
⟨*proof*⟩

lemma *in-diffD*:
assumes $a \in\# M - N$
shows $a \in\# M$
⟨*proof*⟩

lemma *set-mset-diff*:
$\mathit{set\text{-}mset}\ (M - N) = \{a.\ \mathit{count}\ N\ a < \mathit{count}\ M\ a\}$
⟨*proof*⟩

lemma *diff-empty* [*simp*]: $M - \{\#\} = M \wedge \{\#\} - M = \{\#\}$
⟨*proof*⟩

lemma *diff-cancel* [*simp*]: $A - A = \{\#\}$
⟨*proof*⟩

lemma *diff-union-cancelR* [*simp*]: $M + N - N = (M::'a\ \mathit{multiset})$
⟨*proof*⟩

lemma *diff-union-cancelL* [*simp*]: $N + M - N = (M::'a\ \mathit{multiset})$
⟨*proof*⟩

lemma *diff-right-commute*:
fixes $M\ N\ Q :: 'a\ \mathit{multiset}$
shows $M - N - Q = M - Q - N$
⟨*proof*⟩

lemma *diff-add*:
fixes $M\ N\ Q :: 'a\ \mathit{multiset}$
shows $M - (N + Q) = M - N - Q$
⟨*proof*⟩

lemma *insert-DiffM*: $x \in\# M \Longrightarrow \{\#x\#\} + (M - \{\#x\#\}) = M$
⟨*proof*⟩

lemma *insert-DiffM2* [*simp*]: $x \in\# M \Longrightarrow M - \{\#x\#\} + \{\#x\#\} = M$
⟨*proof*⟩

lemma *diff-union-swap*: $a \neq b \Longrightarrow M - \{\#a\#\} + \{\#b\#\} = M + \{\#b\#\} - \{\#a\#\}$
⟨*proof*⟩

lemma *diff-union-single-conv*:
$a \in\# J \Longrightarrow I + J - \{\#a\#\} = I + (J - \{\#a\#\})$
⟨*proof*⟩

lemma *mset-add* [*elim?*]:
assumes $a \in\# A$
obtains B **where** $A = B + \{\#a\#\}$
⟨*proof*⟩

lemma *union-iff*:
$a \in\# A + B \longleftrightarrow a \in\# A \vee a \in\# B$
⟨*proof*⟩

65.3.4 Equality of multisets

lemma *single-not-empty* [*simp*]: $\{\#a\#\} \neq \{\#\} \wedge \{\#\} \neq \{\#a\#\}$
⟨*proof*⟩

lemma *single-eq-single* [*simp*]: $\{\#a\#\} = \{\#b\#\} \longleftrightarrow a = b$
⟨*proof*⟩

lemma *union-eq-empty* [*iff*]: $M + N = \{\#\} \longleftrightarrow M = \{\#\} \wedge N = \{\#\}$
⟨*proof*⟩

lemma *empty-eq-union* [*iff*]: $\{\#\} = M + N \longleftrightarrow M = \{\#\} \wedge N = \{\#\}$
⟨*proof*⟩

lemma *multi-self-add-other-not-self* [*simp*]: $M = M + \{\#x\#\} \longleftrightarrow \mathit{False}$
⟨*proof*⟩

lemma *diff-single-trivial*: $\neg\ x \in\# M \Longrightarrow M - \{\#x\#\} = M$
⟨*proof*⟩

lemma *diff-single-eq-union*: $x \in\# M \Longrightarrow M - \{\#x\#\} = N \longleftrightarrow M = N + \{\#x\#\}$
⟨*proof*⟩

lemma *union-single-eq-diff*: $M + \{\#x\#\} = N \Longrightarrow M = N - \{\#x\#\}$
⟨*proof*⟩

lemma *union-single-eq-member*: $M + \{\#x\#\} = N \Longrightarrow x \in\# N$
⟨*proof*⟩

lemma *union-is-single*:
$M + N = \{\#a\#\} \longleftrightarrow M = \{\#a\#\} \wedge N = \{\#\} \vee M = \{\#\} \wedge N = \{\#a\#\}$
(**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *single-is-union*: $\{\#a\#\} = M + N \longleftrightarrow \{\#a\#\} = M \wedge N = \{\#\} \vee M = \{\#\} \wedge \{\#a\#\} = N$
⟨*proof*⟩

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 ⟷ ?rhs*)

⟨*proof*⟩

lemma *insert-noteq-member*:
assumes *BC*: $B + \{\#b\#\} = C + \{\#c\#\}$
and *bnotc*: $b \neq c$
shows $c \in\# B$
⟨*proof*⟩

lemma *add-eq-conv-ex*:
$(M + \{\#a\#\} = N + \{\#b\#\}) =$
$(M = N \wedge a = b \vee (\exists K.\ M = K + \{\#b\#\} \wedge N = K + \{\#a\#\}))$
⟨*proof*⟩

lemma *multi-member-split*: $x \in\# M \Longrightarrow \exists A.\ M = A + \{\#x\#\}$
⟨*proof*⟩

lemma *multiset-add-sub-el-shuffle*:
assumes $c \in\# B$
and $b \neq c$
shows $B - \{\#c\#\} + \{\#b\#\} = B + \{\#b\#\} - \{\#c\#\}$
⟨*proof*⟩

65.3.5 Pointwise ordering induced by count

definition *subseteq-mset* :: *'a multiset* ⇒ *'a multiset* ⇒ *bool* (**infix** $\subseteq\#$ *50*)
where $A \subseteq\# B = (\forall a.\ count\ A\ a \leq count\ B\ a)$

definition *subset-mset* :: *'a multiset* ⇒ *'a multiset* ⇒ *bool* (**infix** $\subset\#$ *50*)
where $A \subset\# B = (A \subseteq\# B \wedge A \neq B)$

abbreviation (*input*) *supseteq-mset* :: *'a multiset* ⇒ *'a multiset* ⇒ *bool* (**infix** $\supseteq\#$ *50*)
where *supseteq-mset A B* ≡ $B \subseteq\# A$

abbreviation (*input*) *supset-mset* :: *'a multiset* ⇒ *'a multiset* ⇒ *bool* (**infix** $\supset\#$ *50*)
where *supset-mset A B* ≡ $B \subset\# A$

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 $\subset\#$
⟨*proof*⟩

lemma *mset-less-eqI*:
$(\bigwedge a.\ count\ A\ a \leq count\ B\ a) \Longrightarrow A \subseteq\# B$
⟨*proof*⟩

lemma *mset-less-eq-count*:
$A \subseteq\# B \Longrightarrow count\ A\ a \leq count\ B\ a$
⟨*proof*⟩

lemma *mset-le-exists-conv*: $(A::'a\ multiset) \subseteq\# B \longleftrightarrow (\exists\, C.\ B = A + C)$
⟨*proof*⟩

interpretation *subset-mset*: *ordered-cancel-comm-monoid-diff* $op + 0\ op \leq\#\ op <\#\ op -$
⟨*proof*⟩

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*]: $(A::'a\ multiset) + C \subseteq\# B + C \longleftrightarrow A \subseteq\# B$
⟨*proof*⟩

lemma *mset-le-mono-add-left-cancel* [*simp*]: $C + (A::'a\ multiset) \subseteq\# C + B \longleftrightarrow A \subseteq\# B$
⟨*proof*⟩

lemma *mset-le-mono-add*: $(A::'a\ multiset) \subseteq\# B \Longrightarrow C \subseteq\# D \Longrightarrow A + C \subseteq\# B + D$
⟨*proof*⟩

lemma *mset-le-add-left* [*simp*]: $(A::'a\ multiset) \subseteq\# A + B$
⟨*proof*⟩

lemma *mset-le-add-right* [*simp*]: $B \subseteq\# (A::'a\ multiset) + B$
⟨*proof*⟩

lemma *single-subset-iff* [*simp*]:
$\{\#a\#\} \subseteq\# M \longleftrightarrow a \in\# M$
⟨*proof*⟩

lemma *mset-le-single*: $a \in\# B \Longrightarrow \{\#a\#\} \subseteq\# B$
⟨*proof*⟩

lemma *multiset-diff-union-assoc*:
fixes $A\ B\ C\ D :: 'a\ multiset$
shows $C \subseteq\# B \Longrightarrow A + B - C = A + (B - C)$

⟨*proof*⟩

lemma *mset-le-multiset-union-diff-commute*:
fixes $A\ B\ C\ D :: 'a\ multiset$
shows $B \subseteq\# A \Longrightarrow A - B + C = A + C - B$
⟨*proof*⟩

lemma *diff-le-self*[*simp*]:
$(M::'a\ multiset) - N \subseteq\# M$
⟨*proof*⟩

lemma *mset-leD*:
assumes $A \subseteq\# B$ **and** $x \in\# A$
shows $x \in\# B$
⟨*proof*⟩

lemma *mset-lessD*:
$A \subset\# B \Longrightarrow x \in\# A \Longrightarrow x \in\# B$
⟨*proof*⟩

lemma *set-mset-mono*:
$A \subseteq\# B \Longrightarrow set\text{-}mset\ A \subseteq set\text{-}mset\ B$
⟨*proof*⟩

lemma *mset-le-insertD*:
$A + \{\#x\#\} \subseteq\# B \Longrightarrow x \in\# B \wedge A \subset\# B$
⟨*proof*⟩

lemma *mset-less-insertD*:
$A + \{\#x\#\} \subset\# B \Longrightarrow x \in\# B \wedge A \subset\# B$
⟨*proof*⟩

lemma *mset-less-of-empty*[*simp*]: $A \subset\# \{\#\} \longleftrightarrow False$
⟨*proof*⟩

lemma *empty-le* [*simp*]: $\{\#\} \subseteq\# A$
⟨*proof*⟩

lemma *insert-subset-eq-iff*:
$\{\#a\#\} + A \subseteq\# B \longleftrightarrow a \in\# B \wedge A \subseteq\# B - \{\#a\#\}$
⟨*proof*⟩

lemma *insert-union-subset-iff*:
$\{\#a\#\} + A \subset\# B \longleftrightarrow a \in\# B \wedge A \subset\# B - \{\#a\#\}$
⟨*proof*⟩

lemma *subset-eq-diff-conv*:
$A - C \subseteq\# B \longleftrightarrow A \subseteq\# B + C$
⟨*proof*⟩

lemma *le-empty* [*simp*]: $M \subseteq\# \{\#\} \longleftrightarrow M = \{\#\}$
⟨*proof*⟩

lemma *multi-psub-of-add-self* [*simp*]: $A \subset\# A + \{\#x\#\}$
⟨*proof*⟩

lemma *multi-psub-self* [*simp*]: $(A::'a\ multiset) \subset\# A = False$
⟨*proof*⟩

lemma *mset-less-add-bothsides*: $N + \{\#x\#\} \subset\# M + \{\#x\#\} \Longrightarrow N \subset\# M$
⟨*proof*⟩

lemma *mset-less-empty-nonempty*: $\{\#\} \subset\# S \longleftrightarrow S \neq \{\#\}$
⟨*proof*⟩

lemma *mset-less-diff-self*: $c \in\# B \Longrightarrow B - \{\#c\#\} \subset\# B$
⟨*proof*⟩

65.3.6 Intersection

definition *inf-subset-mset* :: $'a\ multiset \Rightarrow 'a\ multiset \Rightarrow 'a\ multiset$ (**infixl** $\#\cap$ *70*) **where**
multiset-inter-def: $inf\text{-}subset\text{-}mset\ A\ B = A - (A - B)$

interpretation *subset-mset*: *semilattice-inf inf-subset-mset op* $\subseteq\#$ *op* $\subset\#$
⟨*proof*⟩

lemma *multiset-inter-count* [*simp*]:
fixes $A\ B :: 'a\ multiset$
shows $count\ (A\ \#\cap\ B)\ x = min\ (count\ A\ x)\ (count\ B\ x)$
⟨*proof*⟩

lemma *set-mset-inter* [*simp*]:
$set\text{-}mset\ (A\ \#\cap\ B) = set\text{-}mset\ A \cap set\text{-}mset\ B$
⟨*proof*⟩

lemma *diff-intersect-left-idem* [*simp*]:
$M - M\ \#\cap\ N = M - N$
⟨*proof*⟩

lemma *diff-intersect-right-idem* [*simp*]:
$M - N\ \#\cap\ M = M - N$
⟨*proof*⟩

lemma *multiset-inter-single*: $a \neq b \Longrightarrow \{\#a\#\}\ \#\cap\ \{\#b\#\} = \{\#\}$
⟨*proof*⟩

lemma *multiset-union-diff-commute*:

assumes $B \#\cap C = \{\#\}$
shows $A + B - C = A - C + B$
⟨*proof*⟩

lemma *disjunct-not-in*:
$A \#\cap B = \{\#\} \longleftrightarrow (\forall a.\ a \notin\# A \vee a \notin\# B)$ (**is** $?P \longleftrightarrow ?Q$)
⟨*proof*⟩

lemma *empty-inter* [*simp*]: $\{\#\} \#\cap M = \{\#\}$
⟨*proof*⟩

lemma *inter-empty* [*simp*]: $M \#\cap \{\#\} = \{\#\}$
⟨*proof*⟩

lemma *inter-add-left1*: $\neg\ x \in\# N \Longrightarrow (M + \{\#x\#\}) \#\cap N = M \#\cap N$
⟨*proof*⟩

lemma *inter-add-left2*: $x \in\# N \Longrightarrow (M + \{\#x\#\}) \#\cap N = (M \#\cap (N - \{\#x\#\})) + \{\#x\#\}$
⟨*proof*⟩

lemma *inter-add-right1*: $\neg\ x \in\# N \Longrightarrow N \#\cap (M + \{\#x\#\}) = N \#\cap M$
⟨*proof*⟩

lemma *inter-add-right2*: $x \in\# N \Longrightarrow N \#\cap (M + \{\#x\#\}) = ((N - \{\#x\#\}) \#\cap M) + \{\#x\#\}$
⟨*proof*⟩

lemma *disjunct-set-mset-diff*:
assumes $M \#\cap N = \{\#\}$
shows *set-mset* $(M - N) =$ *set-mset* M
⟨*proof*⟩

lemma *at-most-one-mset-mset-diff*:
assumes $a \notin\# M - \{\#a\#\}$
shows *set-mset* $(M - \{\#a\#\}) =$ *set-mset* $M - \{a\}$
⟨*proof*⟩

lemma *more-than-one-mset-mset-diff*:
assumes $a \in\# M - \{\#a\#\}$
shows *set-mset* $(M - \{\#a\#\}) =$ *set-mset* M
⟨*proof*⟩

lemma *inter-iff*:
$a \in\# A \#\cap B \longleftrightarrow a \in\# A \wedge a \in\# B$
⟨*proof*⟩

lemma *inter-union-distrib-left*:
$A \#\cap B + C = (A + C) \#\cap (B + C)$

⟨*proof*⟩

lemma *inter-union-distrib-right*:
$C + A \mathbin{\#\cap} B = (C + A) \mathbin{\#\cap} (C + B)$
⟨*proof*⟩

lemma *inter-subset-eq-union*:
$A \mathbin{\#\cap} B \subseteq\# A + B$
⟨*proof*⟩

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*⟩

lemma *sup-subset-mset-count* [*simp*]: — FIXME irregular fact name
count (*A* #∪ *B*) *x* = *max* (*count A x*) (*count B x*)
⟨*proof*⟩

lemma *set-mset-sup* [*simp*]:
set-mset (*A* #∪ *B*) = *set-mset A* ∪ *set-mset B*
⟨*proof*⟩

lemma *empty-sup* [*simp*]: {#} #∪ *M* = *M*
⟨*proof*⟩

lemma *sup-empty* [*simp*]: *M* #∪ {#} = *M*
⟨*proof*⟩

lemma *sup-union-left1*: ¬ *x* ∈# *N* ⟹ (*M* + {#*x*#}) #∪ *N* = (*M* #∪ *N*) + {#*x*#}
⟨*proof*⟩

lemma *sup-union-left2*: *x* ∈# *N* ⟹ (*M* + {#*x*#}) #∪ *N* = (*M* #∪ (*N* − {#*x*#})) + {#*x*#}
⟨*proof*⟩

lemma *sup-union-right1*: ¬ *x* ∈# *N* ⟹ *N* #∪ (*M* + {#*x*#}) = (*N* #∪ *M*) + {#*x*#}
⟨*proof*⟩

lemma *sup-union-right2*: *x* ∈# *N* ⟹ *N* #∪ (*M* + {#*x*#}) = ((*N* − {#*x*#}) #∪ *M*) + {#*x*#}
⟨*proof*⟩

lemma *sup-union-distrib-left*:
$A \#\cup B + C = (A + C) \#\cup (B + C)$
⟨*proof*⟩

lemma *union-sup-distrib-right*:
$C + A \#\cup B = (C + A) \#\cup (C + B)$
⟨*proof*⟩

lemma *union-diff-inter-eq-sup*:
$A + B - A \#\cap B = A \#\cup B$
⟨*proof*⟩

lemma *union-diff-sup-eq-inter*:
$A + B - A \#\cup B = A \#\cap B$
⟨*proof*⟩

65.3.8 Subset is an order

interpretation *subset-mset*: *order op* $\leq\#$ *op* $<\#$ ⟨*proof*⟩

65.3.9 Filter (with comprehension syntax)

Multiset comprehension

lift-definition *filter-mset* :: $('a \Rightarrow bool) \Rightarrow 'a\ multiset \Rightarrow 'a\ multiset$
is $\lambda P\ M.\ \lambda x.\ if\ P\ x\ then\ M\ x\ else\ 0$
⟨*proof*⟩

syntax (*ASCII*)
-MCollect :: *pttrn* $\Rightarrow$ *'a multiset* $\Rightarrow$ *bool* $\Rightarrow$ *'a multiset* (($1\{\#$ - $:\#$ -./ $-\#\}$))
syntax
-MCollect :: *pttrn* $\Rightarrow$ *'a multiset* $\Rightarrow$ *bool* $\Rightarrow$ *'a multiset* (($1\{\#$ - $\in\#$ -./ $-\#\}$))
translations
$\{\#x \in\# M.\ P\#\} == CONST\ filter\text{-}mset\ (\lambda x.\ P)\ M$

lemma *count-filter-mset* [*simp*]:
count (*filter-mset P M*) *a* = (*if P a then count M a else 0*)
⟨*proof*⟩

lemma *set-mset-filter* [*simp*]:
set-mset (*filter-mset P M*) = {*a* ∈ *set-mset M*. *P a*}
⟨*proof*⟩

lemma *filter-empty-mset* [*simp*]: *filter-mset P* {#} = {#}
⟨*proof*⟩

lemma *filter-single-mset* [*simp*]: *filter-mset P* {#*x*#} = (*if P x then* {#*x*#} *else* {#})
⟨*proof*⟩

lemma *filter-union-mset* [*simp*]: *filter-mset P* (*M* + *N*) = *filter-mset P M* + *filter-mset P N*
⟨*proof*⟩

lemma *filter-diff-mset* [*simp*]: *filter-mset P* (*M* − *N*) = *filter-mset P M* − *filter-mset P N*
⟨*proof*⟩

lemma *filter-inter-mset* [*simp*]: *filter-mset P* (*M* #∩ *N*) = *filter-mset P M* #∩ *filter-mset P N*
⟨*proof*⟩

lemma *multiset-filter-subset*[*simp*]: *filter-mset f M* ⊆# *M*
⟨*proof*⟩

lemma *multiset-filter-mono*:
assumes *A* ⊆# *B*
shows *filter-mset f A* ⊆# *filter-mset f B*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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 ⟨*proof*⟩

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*
⟨*proof*⟩

lemma *size-empty* [*simp*]: *size* {#} = *0*
⟨*proof*⟩

lemma *size-multiset-single* [*simp*]: *size-multiset f* {#*b*#} = *Suc* (*f b*)
⟨*proof*⟩

lemma *size-single* [*simp*]: *size* {#*b*#} = *1*
⟨*proof*⟩

lemma *setsum-wcount-Int*:
finite A ⟹ *setsum* (*wcount f N*) (*A* ∩ *set-mset N*) = *setsum* (*wcount f N*) *A*
⟨*proof*⟩

lemma *size-multiset-union* [*simp*]:
size-multiset f (*M* + *N*::'*a multiset*) = *size-multiset f M* + *size-multiset f N*
⟨*proof*⟩

lemma *size-union* [*simp*]: *size* (*M* + *N*::'*a multiset*) = *size M* + *size N*
⟨*proof*⟩

lemma *size-multiset-eq-0-iff-empty* [*iff*]:
size-multiset f M = *0* ⟷ *M* = {#}
⟨*proof*⟩

lemma *size-eq-0-iff-empty* [*iff*]: (*size M* = *0*) = (*M* = {#})
⟨*proof*⟩

lemma *nonempty-has-size*: (*S* ≠ {#}) = (*0* < *size S*)
⟨*proof*⟩

lemma *size-eq-Suc-imp-elem*: *size M* = *Suc n* ⟹ ∃ *a*. *a* ∈# *M*
⟨*proof*⟩

lemma *size-eq-Suc-imp-eq-union*:
assumes *size M* = *Suc n*
shows ∃ *a N*. *M* = *N* + {#*a*#}
⟨*proof*⟩

lemma *size-mset-mono*:
fixes *A B* :: '*a multiset*
assumes *A* ⊆# *B*
shows *size A* ≤ *size B*
⟨*proof*⟩

lemma *size-filter-mset-lesseq*[*simp*]: *size* (*filter-mset f M*) ≤ *size M*
⟨*proof*⟩

lemma *size-Diff-submset*:
$M \subseteq\# M' \Longrightarrow size\ (M' - M) = size\ M' - size(M::'a\ multiset)$
⟨*proof*⟩

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*⟩

lemma *multi-nonempty-split*: $M \neq \{\#\} \Longrightarrow \exists A\ a.\ M = A + \{\#a\#\}$
⟨*proof*⟩

lemma *multiset-cases* [*cases type*]:
obtains (*empty*) $M = \{\#\}$
| (*add*) $N\ x$ **where** $M = N + \{\#x\#\}$
⟨*proof*⟩

lemma *multi-drop-mem-not-eq*: $c \in\# B \Longrightarrow B - \{\#c\#\} \neq B$
⟨*proof*⟩

lemma *multiset-partition*: $M = \{\#\ x{\in}\#M.\ P\ x\ \#\} + \{\#\ x{\in}\#M.\ \neg\ P\ x\ \#\}$
⟨*proof*⟩

lemma *mset-less-size*: $(A::'a\ multiset) \subset\# B \Longrightarrow size\ A < size\ B$
⟨*proof*⟩

lemma *size-1-singleton-mset*: $size\ M = 1 \Longrightarrow \exists a.\ M = \{\#a\#\}$
⟨*proof*⟩

65.4.1 Strong induction and subset induction for multisets

Well-foundedness of strict subset relation

lemma *wf-less-mset-rel*: *wf* $\{(M, N :: 'a\ multiset).\ M \subset\# N\}$
⟨*proof*⟩

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 $P\ B$
⟨*proof*⟩

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*⟩

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* $(\lambda x.\ f\ x\ \hat{}\hat{}\ count\ M\ x)\ s\ (set\text{-}mset\ M)$

lemma *fold-mset-empty* [*simp*]: *fold-mset f s* $\{\#\} = s$
⟨*proof*⟩

context *comp-fun-commute*
begin

lemma *fold-mset-insert*: *fold-mset f s* $(M + \{\#x\#\}) = f\ x$ (*fold-mset f s M*)
⟨*proof*⟩

corollary *fold-mset-single* [*simp*]: *fold-mset f s* $\{\#x\#\} = f\ x\ s$
⟨*proof*⟩

lemma *fold-mset-fun-left-comm*: $f\ x$ (*fold-mset f s M*) = *fold-mset f* (*f x s*) *M*
⟨*proof*⟩

lemma *fold-mset-union* [*simp*]: *fold-mset f s* $(M + N)$ = *fold-mset f* (*fold-mset f s M*) *N*
⟨*proof*⟩

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*⟩

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* :: $('a \Rightarrow 'b) \Rightarrow 'a\ multiset \Rightarrow 'b\ multiset$ **where**
image-mset f = *fold-mset* (*plus* ∘ *single* ∘ *f*) $\{\#\}$

lemma *comp-fun-commute-mset-image*: *comp-fun-commute* (*plus* ∘ *single* ∘ *f*)
⟨*proof*⟩

lemma *image-mset-empty* [*simp*]: *image-mset f* {#} = {#}
⟨*proof*⟩

lemma *image-mset-single* [*simp*]: *image-mset f* {#*x*#} = {#*f x*#}
⟨*proof*⟩

lemma *image-mset-union* [*simp*]: *image-mset f* (*M* + *N*) = *image-mset f M* + *image-mset f N*
⟨*proof*⟩

corollary *image-mset-insert*: *image-mset f* (*M* + {#*a*#}) = *image-mset f M* + {#*f a*#}
⟨*proof*⟩

lemma *set-image-mset* [*simp*]: *set-mset* (*image-mset f M*) = *image f* (*set-mset M*)
⟨*proof*⟩

lemma *size-image-mset* [*simp*]: *size* (*image-mset f M*) = *size M*
⟨*proof*⟩

lemma *image-mset-is-empty-iff* [*simp*]: *image-mset f M* = {#} ⟷ *M* = {#}
⟨*proof*⟩

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*
⟨*proof*⟩

functor *image-mset*: *image-mset*
⟨*proof*⟩

declare
image-mset.id [*simp*]
image-mset.identity [*simp*]

lemma *image-mset-id*[*simp*]: *image-mset id x* = *x*
⟨*proof*⟩

lemma *image-mset-cong*: (⋀*x*. *x* ∈# *M* ⟹ *f x* = *g x*) ⟹ {#*f x*. *x* ∈# *M*#}
= {#*g x*. *x* ∈# *M*#}
⟨*proof*⟩

lemma *image-mset-cong-pair*:
(∀ *x y*. (*x*, *y*) ∈# *M* ⟶ *f x y* = *g x y*) ⟹ {#*f x y*. (*x*, *y*) ∈# *M*#} = {#*g x*
y. (*x*, *y*) ∈# *M*#}
⟨*proof*⟩

65.7 Further conversions

primrec *mset* :: *'a list* ⇒ *'a multiset* **where**
mset [] = {#} |
mset (*a* # *x*) = *mset x* + {# *a* #}

lemma *in-multiset-in-set*:
x ∈# *mset xs* ⟷ *x* ∈ *set xs*
⟨*proof*⟩

lemma *count-mset*:
count (*mset xs*) *x* = *length* (*filter* (λ*y*. *x* = *y*) *xs*)
⟨*proof*⟩

lemma *mset-zero-iff*[*simp*]: (*mset x* = {#}) = (*x* = [])
⟨*proof*⟩

lemma *mset-zero-iff-right*[*simp*]: ({#} = *mset x*) = (*x* = [])
⟨*proof*⟩

lemma *set-mset-mset*[*simp*]: *set-mset* (*mset x*) = *set x*
⟨*proof*⟩

lemma *set-mset-comp-mset* [*simp*]: *set-mset* ∘ *mset* = *set*
⟨*proof*⟩

lemma *size-mset* [*simp*]: *size* (*mset xs*) = *length xs*
⟨*proof*⟩

lemma *mset-append* [*simp*]: *mset* (*xs* @ *ys*) = *mset xs* + *mset ys*
⟨*proof*⟩

lemma *mset-filter*: *mset* (*filter P xs*) = {#*x* ∈# *mset xs*. *P x* #}
⟨*proof*⟩

lemma *mset-rev* [*simp*]:
mset (*rev xs*) = *mset xs*
⟨*proof*⟩

lemma *surj-mset*: *surj mset*
⟨*proof*⟩

lemma *distinct-count-atmost-1*:
distinct x = (∀ *a*. *count* (*mset x*) *a* = (*if a* ∈ *set x then 1 else 0*))
⟨*proof*⟩

lemma *mset-eq-setD*:
assumes *mset xs* = *mset ys*
shows *set xs* = *set ys*
⟨*proof*⟩

lemma *set-eq-iff-mset-eq-distinct*:
distinct x ⟹ *distinct y* ⟹
(*set x* = *set y*) = (*mset x* = *mset y*)
⟨*proof*⟩

lemma *set-eq-iff-mset-remdups-eq*:
(*set x* = *set y*) = (*mset* (*remdups x*) = *mset* (*remdups y*))
⟨*proof*⟩

lemma *mset-compl-union* [*simp*]: *mset* [*x*←*xs*. *P x*] + *mset* [*x*←*xs*. ¬*P x*] = *mset xs*
⟨*proof*⟩

lemma *nth-mem-mset*: *i* < *length ls* ⟹ (*ls* ! *i*) ∈# *mset ls*
⟨*proof*⟩

lemma *mset-remove1*[*simp*]: *mset* (*remove1 a xs*) = *mset xs* − {#*a*#}
⟨*proof*⟩

lemma *mset-eq-length*:
assumes *mset xs* = *mset ys*
shows *length xs* = *length ys*
⟨*proof*⟩

lemma *mset-eq-length-filter*:
assumes *mset xs* = *mset ys*
shows *length* (*filter* (λ*x*. *z* = *x*) *xs*) = *length* (*filter* (λ*y*. *z* = *y*) *ys*)
⟨*proof*⟩

lemma *fold-multiset-equiv*:

assumes f: $\bigwedge x\ y.\ x \in set\ xs \Longrightarrow y \in set\ xs \Longrightarrow f\ x \circ f\ y = f\ y \circ f\ x$
and *equiv*: $mset\ xs = mset\ ys$
shows $List.fold\ f\ xs = List.fold\ f\ ys$
⟨*proof*⟩

lemma *mset-insort* [*simp*]: $mset\ (insort\ x\ xs) = mset\ xs + \{\#x\#\}$
⟨*proof*⟩

lemma *mset-map*: $mset\ (map\ f\ xs) = image\text{-}mset\ f\ (mset\ xs)$
⟨*proof*⟩

global-interpretation *mset-set*: *folding* $\lambda x\ M.\ \{\#x\#\} + M\ \{\#\}$
defines $mset\text{-}set = folding.F\ (\lambda x\ M.\ \{\#x\#\} + M)\ \{\#\}$
⟨*proof*⟩

lemma *count-mset-set* [*simp*]:
finite $A \Longrightarrow x \in A \Longrightarrow count\ (mset\text{-}set\ A)\ x = 1$ (**is** *PROP ?P*)
$\neg$ *finite* $A \Longrightarrow count\ (mset\text{-}set\ A)\ x = 0$ (**is** *PROP ?Q*)
$x \notin A \Longrightarrow count\ (mset\text{-}set\ A)\ x = 0$ (**is** *PROP ?R*)
⟨*proof*⟩

lemma *elem-mset-set*[*simp*, *intro*]: *finite* $A \Longrightarrow x \in\# mset\text{-}set\ A \longleftrightarrow x \in A$
⟨*proof*⟩

context *linorder*
begin

definition *sorted-list-of-multiset* :: $'a\ multiset \Rightarrow 'a\ list$
where
$sorted\text{-}list\text{-}of\text{-}multiset\ M = fold\text{-}mset\ insort\ []\ M$

lemma *sorted-list-of-multiset-empty* [*simp*]:
$sorted\text{-}list\text{-}of\text{-}multiset\ \{\#\} = []$
⟨*proof*⟩

lemma *sorted-list-of-multiset-singleton* [*simp*]:
$sorted\text{-}list\text{-}of\text{-}multiset\ \{\#x\#\} = [x]$
⟨*proof*⟩

lemma *sorted-list-of-multiset-insert* [*simp*]:
$sorted\text{-}list\text{-}of\text{-}multiset\ (M + \{\#x\#\}) = List.insort\ x\ (sorted\text{-}list\text{-}of\text{-}multiset\ M)$
⟨*proof*⟩

end

lemma *mset-sorted-list-of-multiset* [*simp*]:
$mset\ (sorted\text{-}list\text{-}of\text{-}multiset\ M) = M$
⟨*proof*⟩

lemma *sorted-list-of-multiset-mset* [*simp*]:
sorted-list-of-multiset (*mset xs*) = *sort xs*
⟨*proof*⟩

lemma *finite-set-mset-mset-set*[*simp*]:
finite A ⟹ *set-mset* (*mset-set A*) = *A*
⟨*proof*⟩

lemma *infinite-set-mset-mset-set*:
¬ *finite A* ⟹ *set-mset* (*mset-set A*) = {}
⟨*proof*⟩

lemma *set-sorted-list-of-multiset* [*simp*]:
set (*sorted-list-of-multiset M*) = *set-mset M*
⟨*proof*⟩

lemma *sorted-list-of-mset-set* [*simp*]:
sorted-list-of-multiset (*mset-set A*) = *sorted-list-of-set A*
⟨*proof*⟩

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* = {#}
⟨*proof*⟩

lemma *replicate-mset-Suc*[*simp*]: *replicate-mset* (*Suc n*) *x* = *replicate-mset n x* + {#*x*#}
⟨*proof*⟩

lemma *in-replicate-mset*[*simp*]: *x* ∈# *replicate-mset n y* ⟷ *n* > *0* ∧ *x* = *y*
⟨*proof*⟩

lemma *count-replicate-mset*[*simp*]: *count* (*replicate-mset n x*) *y* = (*if y* = *x then n else 0*)
⟨*proof*⟩

lemma *set-mset-replicate-mset-subset*[*simp*]: *set-mset* (*replicate-mset n x*) = (*if n* = *0 then* {} *else* {*x*})
⟨*proof*⟩

lemma *size-replicate-mset*[*simp*]: *size* (*replicate-mset n M*) = *n*
⟨*proof*⟩

lemma *count-le-replicate-mset-le*: *n* ≤ *count M x* ⟷ *replicate-mset n x* ⊆# *M*
⟨*proof*⟩

lemma *filter-eq-replicate-mset*: $\{\#y \in\# D.\ y = x\#\} = replicate\text{-}mset\ (count\ D\ x)\ x$
⟨*proof*⟩

lemma *replicate-count-mset-eq-filter-eq*:
$replicate\ (count\ (mset\ xs)\ k)\ k = filter\ (HOL.eq\ k)\ xs$
⟨*proof*⟩

lemma *replicate-mset-eq-empty-iff* [*simp*]:
$replicate\text{-}mset\ n\ a = \{\#\} \longleftrightarrow n = 0$
⟨*proof*⟩

lemma *replicate-mset-eq-iff*:
$replicate\text{-}mset\ m\ a = replicate\text{-}mset\ n\ b \longleftrightarrow$
$m = 0 \wedge n = 0 \vee m = n \wedge a = b$
⟨*proof*⟩

65.9 Big operators

no-notation *times* (**infixl** $*$ *70*)
no-notation *Groups.one* (*1*)

locale *comm-monoid-mset* = *comm-monoid*
begin

definition $F :: 'a\ multiset \Rightarrow 'a$
where *eq-fold*: $F\ M = fold\text{-}mset\ f\ 1\ M$

lemma *empty* [*simp*]: $F\ \{\#\} = 1$
⟨*proof*⟩

lemma *singleton* [*simp*]: $F\ \{\#x\#\} = x$
⟨*proof*⟩

lemma *union* [*simp*]: $F\ (M + N) = F\ M * F\ N$
⟨*proof*⟩

end

lemma *comp-fun-commute-plus-mset*[*simp*]: *comp-fun-commute* (*op* + :: *'a multiset* ⇒ - ⇒ -)
⟨*proof*⟩

declare *comp-fun-commute.fold-mset-insert*[*OF comp-fun-commute-plus-mset*, *simp*]

lemma *in-mset-fold-plus-iff*[*iff*]: $x \in\# fold\text{-}mset\ (op\ +)\ M\ NN \longleftrightarrow x \in\# M \vee (\exists N.\ N \in\# NN \wedge x \in\# N)$
⟨*proof*⟩

notation *times* (**infixl** $*$ *70*)
notation *Groups.one* (*1*)

context *comm-monoid-add*
begin

sublocale *msetsum*: *comm-monoid-mset plus 0*
  defines *msetsum* = *msetsum.F* ⟨*proof*⟩

lemma (**in** *semiring-1*) *msetsum-replicate-mset* [*simp*]:
  msetsum (*replicate-mset n a*) = *of-nat n* $*$ *a*
  ⟨*proof*⟩

lemma *setsum-unfold-msetsum*:
  setsum f A = *msetsum* (*image-mset f* (*mset-set A*))
  ⟨*proof*⟩

end

lemma *msetsum-diff*:
  fixes *M N* :: ('*a* :: *ordered-cancel-comm-monoid-diff*) *multiset*
  shows *N* ⊆# *M* ⟹ *msetsum* (*M* − *N*) = *msetsum M* − *msetsum N*
  ⟨*proof*⟩

lemma *size-eq-msetsum*: *size M* = *msetsum* (*image-mset* (λ-. *1*) *M*)
⟨*proof*⟩

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*)
  ⟨*proof*⟩

lemma *in-Union-mset-iff*[*iff*]: *x* ∈# ⋃# *MM* ⟷ (∃ *M*. *M* ∈# *MM* ∧ *x* ∈# *M*)
  ⟨*proof*⟩

lemma *count-setsum*:

count (*setsum f A*) *x* = *setsum* (λ*a*. *count* (*f a*) *x*) *A*
⟨*proof*⟩

lemma *setsum-eq-empty-iff*:
assumes *finite A*
shows *setsum f A* = {#} ⟷ (∀ *a*∈*A*. *f a* = {#})
⟨*proof*⟩

context *comm-monoid-mult*
begin

sublocale *msetprod*: *comm-monoid-mset times 1*
defines *msetprod* = *msetprod.F* ⟨*proof*⟩

lemma *msetprod-empty*:
msetprod {#} = *1*
⟨*proof*⟩

lemma *msetprod-singleton*:
msetprod {#*x*#} = *x*
⟨*proof*⟩

lemma *msetprod-Un*:
msetprod (*A* + *B*) = *msetprod A* ∗ *msetprod B*
⟨*proof*⟩

lemma *msetprod-replicate-mset* [*simp*]:
msetprod (*replicate-mset n a*) = *a* ^ *n*
⟨*proof*⟩

lemma *setprod-unfold-msetprod*:
setprod f A = *msetprod* (*image-mset f* (*mset-set A*))
⟨*proof*⟩

lemma *msetprod-multiplicity*:
msetprod M = *setprod* (λ*x*. *x* ^ *count M x*) (*set-mset M*)
⟨*proof*⟩

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 $x \in\# A$
shows x *dvd* *msetprod* A
⟨*proof*⟩

lemma (**in** *semidom*) *msetprod-zero-iff* [*iff*]:
$msetprod\ A = 0 \longleftrightarrow 0 \in\# A$
⟨*proof*⟩

lemma (**in** *semidom-divide*) *msetprod-diff*:
assumes $B \subseteq\# A$ **and** $0 \notin\# B$
shows $msetprod\ (A - B) = msetprod\ A\ div\ msetprod\ B$
⟨*proof*⟩

lemma (**in** *semidom-divide*) *msetprod-minus*:
assumes $a \in\# A$ **and** $a \neq 0$
shows $msetprod\ (A - \{\#a\#\}) = msetprod\ A\ div\ a$
⟨*proof*⟩

lemma (**in** *normalization-semidom*) *normalized-msetprodI*:
assumes $\bigwedge a.\ a \in\# A \Longrightarrow normalize\ a = a$
shows $normalize\ (msetprod\ A) = msetprod\ A$
⟨*proof*⟩

65.10 Alternative representations

65.10.1 Lists

context *linorder*
begin

lemma *mset-insort* [*simp*]:
$mset\ (insort\text{-}key\ k\ x\ xs) = \{\#x\#\} + mset\ xs$
⟨*proof*⟩

lemma *mset-sort* [*simp*]:
$mset\ (sort\text{-}key\ k\ xs) = mset\ xs$
⟨*proof*⟩

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 $sort\text{-}key\ f\ xs = ys$
⟨*proof*⟩

lemma *properties-for-sort*:
assumes *multiset*: $mset\ ys = mset\ xs$

and *sorted ys*
shows *sort xs = ys*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*))
⟨*proof*⟩

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*⟩

end

hide-const (**open**) *part*

lemma *mset-remdups-le*: *mset* (*remdups xs*) ⊆# *mset xs*
⟨*proof*⟩

lemma *mset-update*:
i < *length ls* ⟹ *mset* (*ls*[*i* := *v*]) = *mset ls* − {#*ls* ! *i*#} + {#*v*#}
⟨*proof*⟩

lemma *mset-swap*:
i < *length ls* ⟹ *j* < *length ls* ⟹
mset (*ls*[*j* := *ls* ! *i*, *i* := *ls* ! *j*]) = *mset ls*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *mult1E*:
assumes (*N*, *M*) ∈ *mult1 r*
obtains *a M0 K* **where** *M* = *M0* + {#*a*#} *N* = *M0* + *K* ⋀*b*. *b* ∈# *K* ⟹
(*b*, *a*) ∈ *r*
⟨*proof*⟩

lemma *not-less-empty* [*iff*]: (*M*, {#}) ∉ *mult1 r*
⟨*proof*⟩

lemma *less-add*:
assumes *mult1*: (*N*, *M0* + {#*a*#}) ∈ *mult1 r*
shows
(∃ *M*. (*M*, *M0*) ∈ *mult1 r* ∧ *N* = *M* + {#*a*#}) ∨

$(\exists K.\ (\forall b.\ b \in\# K \longrightarrow (b,\ a) \in r) \wedge N = M0 + K)$
⟨*proof*⟩

lemma *all-accessible*:
assumes *wf r*
shows $\forall M.\ M \in \mathit{Wellfounded.acc}\ (\mathit{mult1}\ r)$
⟨*proof*⟩

theorem *wf-mult1*: $\mathit{wf}\ r \Longrightarrow \mathit{wf}\ (\mathit{mult1}\ r)$
⟨*proof*⟩

theorem *wf-mult*: $\mathit{wf}\ r \Longrightarrow \mathit{wf}\ (\mathit{mult}\ r)$
⟨*proof*⟩

65.11.2 Closure-free presentation

One direction.

lemma *mult-implies-one-step*:
$\mathit{trans}\ r \Longrightarrow (M,\ N) \in \mathit{mult}\ r \Longrightarrow$
$\exists I\ J\ K.\ N = I + J \wedge M = I + K \wedge J \neq \{\#\} \wedge$
$(\forall k \in \mathit{set\text{-}mset}\ K.\ \exists j \in \mathit{set\text{-}mset}\ J.\ (k,\ j) \in r)$
⟨*proof*⟩

lemma *one-step-implies-mult-aux*:
$\forall I\ J\ K.\ \mathit{size}\ J = n \wedge J \neq \{\#\} \wedge (\forall k \in \mathit{set\text{-}mset}\ K.\ \exists j \in \mathit{set\text{-}mset}\ J.\ (k,\ j) \in r)$
$\longrightarrow (I + K,\ I + J) \in \mathit{mult}\ r$
⟨*proof*⟩

lemma *one-step-implies-mult*:
$J \neq \{\#\} \Longrightarrow \forall k \in \mathit{set\text{-}mset}\ K.\ \exists j \in \mathit{set\text{-}mset}\ J.\ (k,\ j) \in r$
$\Longrightarrow (I + K,\ I + J) \in \mathit{mult}\ r$
⟨*proof*⟩

65.11.3 Partial-order properties

lemma (**in** *order*) *mult1-lessE*:
assumes $(N,\ M) \in \mathit{mult1}\ \{(a,\ b).\ a < b\}$
obtains *a M0 K* **where** $M = M0 + \{\#a\#\}\ N = M0 + K$
$a \notin\# K\ \bigwedge b.\ b \in\# K \Longrightarrow b < a$
⟨*proof*⟩

definition *less-multiset* :: *'a*::*order multiset* ⇒ *'a multiset* ⇒ *bool* (**infix** #⊂# *50*)
where $M' \#\subset\# M \longleftrightarrow (M',\ M) \in \mathit{mult}\ \{(x',\ x).\ x' < x\}$

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*⟩

lemma *mult-less-irrefl* [*elim*!]:
fixes *M* :: *'a*::*order multiset*
shows *M* #⊂# *M* ⟹ *R*
⟨*proof*⟩

65.11.4 Monotonicity of multiset union

lemma *mult1-union*: (*B*, *D*) ∈ *mult1 r* ⟹ (*C* + *B*, *C* + *D*) ∈ *mult1 r*
⟨*proof*⟩

lemma *union-less-mono2*: *B* #⊂# *D* ⟹ *C* + *B* #⊂# *C* + (*D*::*'a*::*order multiset*)
⟨*proof*⟩

lemma *union-less-mono1*: *B* #⊂# *D* ⟹ *B* + *C* #⊂# *D* + (*C*::*'a*::*order multiset*)
⟨*proof*⟩

lemma *union-less-mono*:
fixes *A B C D* :: *'a*::*order multiset*
shows *A* #⊂# *C* ⟹ *B* #⊂# *D* ⟹ *A* + *B* #⊂# *C* + *D*
⟨*proof*⟩

interpretation *multiset-order*: *ordered-ab-semigroup-add plus le-multiset less-multiset*
⟨*proof*⟩

65.11.5 Termination proofs with multiset orders

lemma *multi-member-skip*: *x* ∈# *XS* ⟹ *x* ∈# {# *y* #} + *XS*
and *multi-member-this*: *x* ∈# {# *x* #} + *XS*
and *multi-member-last*: *x* ∈# {# *x* #}
⟨*proof*⟩

definition *ms-strict* = *mult pair-less*
definition *ms-weak* = *ms-strict* ∪ *Id*

lemma *ms-reduction-pair*: *reduction-pair* (*ms-strict*, *ms-weak*)
⟨*proof*⟩

lemma *smsI*:
(*set-mset A*, *set-mset B*) ∈ *max-strict* ⟹ (*Z* + *A*, *Z* + *B*) ∈ *ms-strict*
⟨*proof*⟩

lemma *wmsI*:
(*set-mset A*, *set-mset B*) ∈ *max-strict* ∨ *A* = {#} ∧ *B* = {#}
⟹ (*Z* + *A*, *Z* + *B*) ∈ *ms-weak*
⟨*proof*⟩

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*#}
⟨*proof*⟩

lemma *pw-leq-split*:
assumes *pw-leq X Y*
shows ∃ *A B Z*. *X* = *A* + *Z* ∧ *Y* = *B* + *Z* ∧ ((*set-mset A*, *set-mset B*) ∈ *max-strict* ∨ (*B* = {#} ∧ *A* = {#}))
⟨*proof*⟩

lemma
assumes *pwleq*: *pw-leq Z Z'*
shows *ms-strictI*: (*set-mset A*, *set-mset B*) ∈ *max-strict* ⟹ (*Z* + *A*, *Z'* + *B*) ∈ *ms-strict*
and *ms-weakI1*: (*set-mset A*, *set-mset B*) ∈ *max-strict* ⟹ (*Z* + *A*, *Z'* + *B*) ∈ *ms-weak*
and *ms-weakI2*: (*Z* + {#}, *Z'* + {#}) ∈ *ms-weak*
⟨*proof*⟩

lemma *empty-neutral*: {#} + *x* = *x x* + {#} = *x*
and *nonempty-plus*: {# *x* #} + *rs* ≠ {#}
and *nonempty-single*: {# *x* #} ≠ {#}
⟨*proof*⟩

⟨*ML*⟩

65.12 Legacy theorem bindings

lemmas *multi-count-eq* = *multiset-eq-iff* [*symmetric*]

lemma *union-commute*: *M* + *N* = *N* + (*M*::'*a multiset*)
⟨*proof*⟩

lemma *union-assoc*: (*M* + *N*) + *K* = *M* + (*N* + (*K*::'*a multiset*))
⟨*proof*⟩

lemma *union-lcomm*: *M* + (*N* + *K*) = *N* + (*M* + (*K*::'*a multiset*))
⟨*proof*⟩

lemmas *union-ac = union-assoc union-commute union-lcomm*

lemma *union-right-cancel*: $M + K = N + K \longleftrightarrow M = (N::'a\ multiset)$
⟨*proof*⟩

lemma *union-left-cancel*: $K + M = K + N \longleftrightarrow M = (N::'a\ multiset)$
⟨*proof*⟩

lemma *multi-union-self-other-eq*: $(A::'a\ multiset) + X = A + Y \Longrightarrow X = Y$
⟨*proof*⟩

lemma *mset-less-trans*: $(M::'a\ multiset) \subset\# K \Longrightarrow K \subset\# N \Longrightarrow M \subset\# N$
⟨*proof*⟩

lemma *multiset-inter-commute*: $A \#\cap B = B \#\cap A$
⟨*proof*⟩

lemma *multiset-inter-assoc*: $A \#\cap (B \#\cap C) = A \#\cap B \#\cap C$
⟨*proof*⟩

lemma *multiset-inter-left-commute*: $A \#\cap (B \#\cap C) = B \#\cap (A \#\cap C)$
⟨*proof*⟩

lemmas *multiset-inter-ac =*
multiset-inter-commute
multiset-inter-assoc
multiset-inter-left-commute

lemma *mult-less-not-refl*: $\neg\ M \#\subset\#\ (M::'a::order\ multiset)$
⟨*proof*⟩

lemma *mult-less-trans*: $K \#\subset\# M \Longrightarrow M \#\subset\# N \Longrightarrow K \#\subset\# (N::'a::order\ multiset)$
⟨*proof*⟩

lemma *mult-less-not-sym*: $M \#\subset\# N \Longrightarrow \neg\ N \#\subset\# (M::'a::order\ multiset)$
⟨*proof*⟩

lemma *mult-less-asym*: $M \#\subset\# N \Longrightarrow (\neg\ P \Longrightarrow N \#\subset\# (M::'a::order\ multiset)) \Longrightarrow P$
⟨*proof*⟩

⟨*ML*⟩

65.13 Naive implementation using lists

code-datatype *mset*

lemma [*code*]: $\{\#\} = mset\ []$
⟨*proof*⟩

lemma [*code*]: $\{\#x\#\} = mset\ [x]$
⟨*proof*⟩

lemma *union-code* [*code*]: $mset\ xs + mset\ ys = mset\ (xs\ @\ ys)$
⟨*proof*⟩

lemma [*code*]: $image\text{-}mset\ f\ (mset\ xs) = mset\ (map\ f\ xs)$
⟨*proof*⟩

lemma [*code*]: $filter\text{-}mset\ f\ (mset\ xs) = mset\ (filter\ f\ xs)$
⟨*proof*⟩

lemma [*code*]: $mset\ xs - mset\ ys = mset\ (fold\ remove1\ ys\ xs)$
⟨*proof*⟩

lemma [*code*]:
$mset\ xs\ \#\cap\ mset\ ys =$
$mset\ (snd\ (fold\ (\lambda x\ (ys,\ zs).$
$if\ x \in set\ ys\ then\ (remove1\ x\ ys,\ x\ \#\ zs)\ else\ (ys,\ zs))\ xs\ (ys,\ [])))$
⟨*proof*⟩

lemma [*code*]:
$mset\ xs\ \#\cup\ mset\ ys =$
$mset\ (case\text{-}prod\ append\ (fold\ (\lambda x\ (ys,\ zs).\ (remove1\ x\ ys,\ x\ \#\ zs))\ xs\ (ys,\ [])))$
⟨*proof*⟩

declare *in-multiset-in-set* [*code-unfold*]

lemma [*code*]: $count\ (mset\ xs)\ x = fold\ (\lambda y.\ if\ x = y\ then\ Suc\ else\ id)\ xs\ 0$
⟨*proof*⟩

declare *set-mset-mset* [*code*]

declare *sorted-list-of-multiset-mset* [*code*]

lemma [*code*]: — not very efficient, but representation-ignorant!
$mset\text{-}set\ A = mset\ (sorted\text{-}list\text{-}of\text{-}set\ A)$
⟨*proof*⟩

declare *size-mset* [*code*]

fun *ms-lesseq-impl* :: *'a list* ⇒ *'a list* ⇒ *bool option* **where**
$ms\text{-}lesseq\text{-}impl\ []\ ys = Some\ (ys \neq [])$
| $ms\text{-}lesseq\text{-}impl\ (Cons\ x\ xs)\ ys = (case\ List.extract\ (op = x)\ ys\ of$
$None \Rightarrow None$
| $Some\ (ys1,\text{-},ys2) \Rightarrow ms\text{-}lesseq\text{-}impl\ xs\ (ys1\ @\ ys2))$

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*⟩

lemma [*code*]: *mset xs* ⊆# *mset ys* ⟷ *ms-lesseq-impl xs ys* ≠ *None*
⟨*proof*⟩

lemma [*code*]: *mset xs* ⊂# *mset ys* ⟷ *ms-lesseq-impl xs ys* = *Some True*
⟨*proof*⟩

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*
⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma [*code*]: *msetsum* (*mset xs*) = *listsum xs*
⟨*proof*⟩

lemma [*code*]: *msetprod* (*mset xs*) = *fold times xs 1*
⟨*proof*⟩

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 ⟨*proof*⟩

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 ⟨*proof*⟩

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*)#}
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *list-all2-reorder-left-invariance*:
assumes *rel*: *list-all2 R xs ys* **and** *ms-x*: *mset xs'* = *mset xs*
shows ∃ *ys'*. *list-all2 R xs' ys'* ∧ *mset ys'* = *mset ys*
⟨*proof*⟩

lemma *ex-mset*: ∃ *xs*. *mset xs* = *X*
⟨*proof*⟩

inductive *pred-mset* :: ('*a* ⇒ *bool*) ⇒ '*a multiset* ⇒ *bool*

where
  pred-mset P {#}
| ⟦*P a*; *pred-mset P M*⟧ ⟹ *pred-mset P* (*M* + {#*a*#})

bnf *'a multiset*
  map: *image-mset*
  sets: *set-mset*
  bd: *natLeq*
  wits: {#}
  rel: *rel-mset*
  pred: *pred-mset*
⟨*proof*⟩

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 {#} {#}*
⟨*proof*⟩

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*⟩

lemma *rel-mset'-imp-rel-mset*: *rel-mset' R M N* ⟹ *rel-mset R M N*
  ⟨*proof*⟩

lemma *rel-mset-size*: *rel-mset R M N* ⟹ *size M* = *size N*
  ⟨*proof*⟩

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*
⟨*proof*⟩

lemma *multiset-induct2-size*[*consumes 1*, *case-names empty add*]:
  assumes *c*: *size M* = *size N*
    and *empty*: *P {#} {#}*

and *add*: $\bigwedge M\ N\ a\ b.\ P\ M\ N \Longrightarrow P\ (M + \{\#a\#\})\ (N + \{\#b\#\})$
shows *P M N*
⟨*proof*⟩

lemma *msed-map-invL*:
assumes *image-mset f* $(M + \{\#a\#\}) = N$
shows $\exists N1.\ N = N1 + \{\#f\ a\#\} \wedge$ *image-mset f M* $=$ *N1*
⟨*proof*⟩

lemma *msed-map-invR*:
assumes *image-mset f M* $= N + \{\#b\#\}$
shows $\exists M1\ a.\ M = M1 + \{\#a\#\} \wedge f\ a = b \wedge$ *image-mset f M1* $= N$
⟨*proof*⟩

lemma *msed-rel-invL*:
assumes *rel-mset R* $(M + \{\#a\#\})\ N$
shows $\exists N1\ b.\ N = N1 + \{\#b\#\} \wedge R\ a\ b \wedge$ *rel-mset R M N1*
⟨*proof*⟩

lemma *msed-rel-invR*:
assumes *rel-mset R M* $(N + \{\#b\#\})$
shows $\exists M1\ a.\ M = M1 + \{\#a\#\} \wedge R\ a\ b \wedge$ *rel-mset R M1 N*
⟨*proof*⟩

lemma *rel-mset-imp-rel-mset′*:
assumes *rel-mset R M N*
shows *rel-mset′ R M N*
⟨*proof*⟩

lemma *rel-mset-rel-mset′*: *rel-mset R M N* $=$ *rel-mset′ R M N*
⟨*proof*⟩

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* $\circ$ *image-mset f* $=$ *size-multiset* $(g \circ f)$
⟨*proof*⟩

⟨*ML*⟩

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* $\leq$)
⟨*proof*⟩

lemma *antisymP-le*: *antisymP* (*op* $\leq$)
⟨*proof*⟩

lemma *transp-le*: *transp* (*op* $\leq$)
⟨*proof*⟩

lemma *irreflp-less*: *irreflp* (*op* $<$)
⟨*proof*⟩

lemma *antisymP-less*: *antisymP* (*op* $<$)
⟨*proof*⟩

lemma *transp-less*: *transp* (*op* $<$)
⟨*proof*⟩

lemmas *le-trans* = *transp-le*[*unfolded transp-def*, *rule-format*]

lemma *order-mult*: *class.order*
(λ*M N*. (*M*, *N*) ∈ *mult* {(*x*, *y*). *x* < *y*} ∨ *M* = *N*)
(λ*M N*. (*M*, *N*) ∈ *mult* {(*x*, *y*). *x* < *y*})
(**is** *class.order ?le ?less*)
⟨*proof*⟩

The Dershowitz–Manna ordering:

definition *less-multiset*$_{DM}$ **where**
less-multiset$_{DM}$ *M N* ⟷
(∃ *X Y*. *X* ≠ {#} ∧ *X* ≤# *N* ∧ *M* = (*N* − *X*) + *Y* ∧ (∀ *k*. *k* ∈# *Y* ⟶ (∃ *a*. *a* ∈# *X* ∧ *k* < *a*)))

The Huet–Oppen ordering:

definition *less-multiset*$_{HO}$ **where**
less-multiset$_{HO}$ *M N* ⟷ *M* ≠ *N* ∧ (∀ *y*. *count N y* < *count M y* ⟶ (∃ *x*. *y* < *x* ∧ *count M x* < *count N x*))

lemma *mult-imp-less-multiset*$_{HO}$:
(*M*, *N*) ∈ *mult* {(*x*, *y*). *x* < *y*} ⟹ *less-multiset*$_{HO}$ *M N*

⟨*proof*⟩

lemma *less-multiset$_{DM}$-imp-mult*:
less-multiset$_{DM}$ M N ⟹ (*M*, *N*) ∈ *mult* {(*x*, *y*). *x* < *y*}
⟨*proof*⟩

lemma *less-multiset$_{HO}$-imp-less-multiset$_{DM}$*: *less-multiset$_{HO}$ M N* ⟹ *less-multiset$_{DM}$ M N*
⟨*proof*⟩

lemma *mult-less-multiset$_{DM}$*: (*M*, *N*) ∈ *mult* {(*x*, *y*). *x* < *y*} ⟷ *less-multiset$_{DM}$ M N*
⟨*proof*⟩

lemma *mult-less-multiset$_{HO}$*: (*M*, *N*) ∈ *mult* {(*x*, *y*). *x* < *y*} ⟷ *less-multiset$_{HO}$ M N*
⟨*proof*⟩

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* ≤ *b*}
⟨*proof*⟩

lemma *total-less*: *total* {(*a* :: '*a*, *b*). *a* < *b*}
⟨*proof*⟩

lemma *linorder-mult*: *class.linorder*
(λ*M N*. (*M*, *N*) ∈ *mult* {(*x*, *y*). *x* < *y*} ∨ *M* = *N*)
(λ*M N*. (*M*, *N*) ∈ *mult* {(*x*, *y*). *x* < *y*})
⟨*proof*⟩

end

lemma *less-multiset-less-multiset$_{HO}$*:
M #⊂# *N* ⟷ *less-multiset$_{HO}$ M N*
⟨*proof*⟩

lemmas *less-multiset$_{DM}$* = *mult$_{DM}$*[*folded less-multiset-def*]
lemmas *less-multiset$_{HO}$* = *mult$_{HO}$*[*folded less-multiset-def*]

lemma *le-multiset$_{HO}$*:
fixes *M N* :: ('*a* :: *linorder*) *multiset*
shows *M* #⊆# *N* ⟷ (∀ *y*. *count N y* < *count M y* ⟶ (∃ *x*. *y* < *x* ∧ *count*

$M\ x < count\ N\ x))$
⟨*proof*⟩

lemma *wf-less-multiset*: $wf\ \{(M :: ('a :: wellorder)\ multiset, N).\ M\ \#\subset\#\ N\}$
⟨*proof*⟩

lemma *order-multiset*: *class.order*
$(le\text{-}multiset :: ('a :: order)\ multiset \Rightarrow ('a :: order)\ multiset \Rightarrow bool)$
$(less\text{-}multiset :: ('a :: order)\ multiset \Rightarrow ('a :: order)\ multiset \Rightarrow bool)$
⟨*proof*⟩

lemma *linorder-multiset*: *class.linorder*
$(le\text{-}multiset :: ('a :: linorder)\ multiset \Rightarrow ('a :: linorder)\ multiset \Rightarrow bool)$
$(less\text{-}multiset :: ('a :: linorder)\ multiset \Rightarrow ('a :: linorder)\ multiset \Rightarrow bool)$
⟨*proof*⟩

interpretation *multiset-linorder*: *linorder*
$le\text{-}multiset :: ('a :: linorder)\ multiset \Rightarrow ('a :: linorder)\ multiset \Rightarrow bool$
$less\text{-}multiset :: ('a :: linorder)\ multiset \Rightarrow ('a :: linorder)\ multiset \Rightarrow bool$
⟨*proof*⟩

interpretation *multiset-wellorder*: *wellorder*
$le\text{-}multiset :: ('a :: wellorder)\ multiset \Rightarrow ('a :: wellorder)\ multiset \Rightarrow bool$
$less\text{-}multiset :: ('a :: wellorder)\ multiset \Rightarrow ('a :: wellorder)\ multiset \Rightarrow bool$
⟨*proof*⟩

lemma *le-multiset-total*:
fixes $M\ N :: ('a :: linorder)\ multiset$
shows $\neg\ M\ \#\subseteq\#\ N \Longrightarrow N\ \#\subseteq\#\ M$
⟨*proof*⟩

lemma *less-eq-imp-le-multiset*:
fixes $M\ N :: ('a :: linorder)\ multiset$
shows $M\ \leq\#\ N \Longrightarrow M\ \#\subseteq\#\ N$
⟨*proof*⟩

lemma *less-multiset-right-total*:
fixes $M :: ('a :: linorder)\ multiset$
shows $M\ \#\subset\#\ M + \{\#undefined\#\}$
⟨*proof*⟩

lemma *le-multiset-empty-left*[*simp*]:
fixes $M :: ('a :: linorder)\ multiset$
shows $\{\#\}\ \#\subseteq\#\ M$
⟨*proof*⟩

lemma *le-multiset-empty-right*[*simp*]:
fixes $M :: ('a :: linorder)\ multiset$
shows $M \neq \{\#\} \Longrightarrow \neg\ M\ \#\subseteq\#\ \{\#\}$

⟨*proof*⟩

lemma *less-multiset-empty-left*[*simp*]:
fixes $M :: ('a :: linorder)\ multiset$
shows $M \neq \{\#\} \Longrightarrow \{\#\} \#\subset\# M$
⟨*proof*⟩

lemma *less-multiset-empty-right*[*simp*]:
fixes $M :: ('a :: linorder)\ multiset$
shows $\neg\ M \#\subset\# \{\#\}$
⟨*proof*⟩

lemma
fixes $M\ N :: ('a :: linorder)\ multiset$
shows
le-multiset-plus-left[*simp*]: $N \#\subseteq\# (M + N)$ **and**
le-multiset-plus-right[*simp*]: $M \#\subseteq\# (M + N)$
⟨*proof*⟩

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*]: $M + N \#\subset\# M + N' \longleftrightarrow N \#\subset\# N'$
⟨*proof*⟩

lemma *add-eq-self-empty-iff*: $M + N = M \longleftrightarrow N = \{\#\}$
⟨*proof*⟩

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*]: $N \neq \{\#\} \Longrightarrow M \#\subset\# M + N$
⟨*proof*⟩

lemma *ex-gt-imp-less-multiset*: $(\exists\, y :: 'a :: linorder.\ y \in\# N \wedge (\forall\, x.\ x \in\# M \longrightarrow x < y)) \Longrightarrow M \#\subset\# N$
⟨*proof*⟩

lemma *ex-gt-count-imp-less-multiset*:
$(\forall\, y :: 'a :: linorder.\ y \in\# M + N \longrightarrow y \leq x) \Longrightarrow count\ M\ x < count\ N\ x \Longrightarrow M \#\subset\# N$
⟨*proof*⟩

lemma *union-less-diff-plus*: $P \leq\# M \Longrightarrow N \#\subset\# P \Longrightarrow M - P + N \#\subset\# M$
⟨*proof*⟩

end

67 Numeral Syntax for Types

theory *Numeral-Type*
imports *Cardinality*
begin

67.1 Numeral Types

typedef *num0* = *UNIV* :: *nat set* ⟨*proof*⟩
typedef *num1* = *UNIV* :: *unit set* ⟨*proof*⟩

typedef *'a bit0* = {*0* ..< *2* ∗ *int CARD*('a::*finite*)}
⟨*proof*⟩

typedef *'a bit1* = {*0* ..< *1* + *2* ∗ *int CARD*('a::*finite*)}
⟨*proof*⟩

lemma *card-num0* [*simp*]: *CARD* (*num0*) = *0*
⟨*proof*⟩

lemma *infinite-num0*: ¬ *finite* (*UNIV* :: *num0 set*)
⟨*proof*⟩

lemma *card-num1* [*simp*]: *CARD*(*num1*) = *1*
⟨*proof*⟩

lemma *card-bit0* [*simp*]: *CARD*('a *bit0*) = *2* ∗ *CARD*('a::*finite*)
⟨*proof*⟩

lemma *card-bit1* [*simp*]: *CARD*('a *bit1*) = *Suc* (*2* ∗ *CARD*('a::*finite*))
⟨*proof*⟩

instance *num1* :: *finite*
⟨*proof*⟩

instance *bit0* :: (*finite*) *card2*
⟨*proof*⟩

instance *bit1* :: (*finite*) *card2*
⟨*proof*⟩

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*: $0 < n$
⟨*proof*⟩

lemmas *definitions* =
zero-def one-def add-def mult-def minus-def diff-def

lemma *Rep-less-n*: $Rep\ x < n$
⟨*proof*⟩

lemma *Rep-le-n*: $Rep\ x \leq n$
⟨*proof*⟩

lemma *Rep-inject-sym*: $x = y \longleftrightarrow Rep\ x = Rep\ y$
⟨*proof*⟩

lemma *Rep-inverse*: $Abs\ (Rep\ x) = x$
⟨*proof*⟩

lemma *Abs-inverse*: $m \in \{0..<n\} \Longrightarrow Rep\ (Abs\ m) = m$
⟨*proof*⟩

lemma *Rep-Abs-mod*: $Rep\ (Abs\ (m\ mod\ n)) = m\ mod\ n$
⟨*proof*⟩

lemma *Rep-Abs-0*: $Rep\ (Abs\ 0) = 0$
⟨*proof*⟩

lemma *Rep-0*: $Rep\ 0 = 0$
⟨*proof*⟩

lemma *Rep-Abs-1*: $Rep\ (Abs\ 1) = 1$
⟨*proof*⟩

lemma *Rep-1*: $Rep\ 1 = 1$
⟨*proof*⟩

lemma *Rep-mod*: $Rep\ x\ mod\ n = Rep\ x$
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *of-int-eq*: *of-int z* = *Abs* (*z mod n*)
⟨*proof*⟩

lemma *Rep-numeral*:
Rep (*numeral w*) = *numeral w mod n*
⟨*proof*⟩

lemma *iszero-numeral*:
iszero (*numeral w*::'*a*) ⟷ *numeral w mod n* = *0*
⟨*proof*⟩

lemma *cases*:
assumes *1*: ⋀*z*. ⟦(*x*::'*a*) = *of-int z*; *0* ≤ *z*; *z* < *n*⟧ ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *induct*:
(⋀*z*. ⟦*0* ≤ *z*; *z* < *n*⟧ ⟹ *P* (*of-int z*)) ⟹ *P* (*x*::'*a*)
⟨*proof*⟩

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*) ⟷ *True*
⟨*proof*⟩

instance
⟨*proof*⟩

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 ⟨*proof*⟩

end

interpretation *bit0*:
mod-type int CARD(*'a*::*finite bit0*)
Rep-bit0 :: *'a*::*finite bit0* ⇒ *int*
Abs-bit0 :: *int* ⇒ *'a*::*finite bit0*
⟨*proof*⟩

interpretation *bit1*:
mod-type int CARD(*'a*::*finite bit1*)
Rep-bit1 :: *'a*::*finite bit1* ⇒ *int*
Abs-bit1 :: *int* ⇒ *'a*::*finite bit1*
⟨*proof*⟩

instance *bit0* :: (*finite*) *comm-ring-1*
⟨*proof*⟩

instance *bit1* :: (*finite*) *comm-ring-1*

⟨*proof*⟩

interpretation *bit0*:
mod-ring int CARD('*a*::*finite bit0*)
Rep-bit0 :: '*a*::*finite bit0* ⇒ *int*
Abs-bit0 :: *int* ⇒ '*a*::*finite bit0*
⟨*proof*⟩

interpretation *bit1*:
mod-ring int CARD('*a*::*finite bit1*)
Rep-bit1 :: '*a*::*finite bit1* ⇒ *int*
Abs-bit1 :: *int* ⇒ '*a*::*finite bit1*
⟨*proof*⟩

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* ≤ *Rep-bit0 b*
definition $a < b \longleftrightarrow$ *Rep-bit1 a* < *Rep-bit1 b*
definition $a \leq b \longleftrightarrow$ *Rep-bit1 a* ≤ *Rep-bit1 b*

instance
⟨*proof*⟩
end

lemma (**in** *preorder*) *tranclp-less*: *op* $<^{++}$ = *op* <
⟨*proof*⟩

instance *bit0* **and** *bit1* :: (*finite*) *wellorder*
⟨*proof*⟩

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 ⟨*proof*⟩
end

lemma *equal-num0-code* [*code*]:
  equal-class.equal Num0 Num0 = *True*
⟨*proof*⟩

code-datatype *1* :: *num1*

instantiation *num1* :: *equal* **begin**
definition *equal-num1* :: *num1* ⇒ *num1* ⇒ *bool*
  where *equal-num1* = *op* =
instance ⟨*proof*⟩
end

lemma *equal-num1-code* [*code*]:
  equal-class.equal (*1* :: *num1*) *1* = *True*
⟨*proof*⟩

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
  ⟨*proof*⟩
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
  ⟨*proof*⟩
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*))

⟨*proof*⟩

lemma *inj-on-Abs-bit0*:
inj-on (*Abs-bit0* :: *int* ⇒ *'a bit0*) {*0..<2* ∗ *int CARD*(*'a* :: *finite*)}
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *inj-on-Abs-bit1*:
inj-on (*Abs-bit1* :: *int* ⇒ *'a bit1*) {*0..<1* + *2* ∗ *int CARD*(*'a* :: *finite*)}
⟨*proof*⟩

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
⟨*proof*⟩

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*⟩

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*⟩

end

instantiation *bit0* **and** *bit1* :: (*finite*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*('*a bit0*) *True*
definition *finite-UNIV* = *Phantom*('*a bit1*) *True*
instance ⟨*proof*⟩
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 ⟨*proof*⟩
end

67.6 Syntax

syntax
-NumeralType :: *num-token* => *type* (-)
-NumeralType0 :: *type* (*0*)
-NumeralType1 :: *type* (*1*)

translations
(*type*) *1* == (*type*) *num1*
(*type*) *0* == (*type*) *num0*

⟨*ML*⟩

67.7 Examples

lemma *CARD*(*0*) = *0* ⟨*proof*⟩
lemma *CARD*(*17*) = *17* ⟨*proof*⟩
lemma *8* ∗ *11* ^ *3* − *6* = (*2*::*5*) ⟨*proof*⟩

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* ⌢ *x* == λ*n*. *if n* < *length w then w*!*n else x* (*n* − *length w*)

definition
iter :: *'a list* ⇒ *'a word* ((-^ω) [*1000*])
where *iter w* == *if w* = [] *then undefined else* (λ*n*. *w*!(*n mod* (*length w*)))

lemma *conc-empty*[*simp*]: [] ⌢ *w* = *w*
⟨*proof*⟩

lemma *conc-fst*[*simp*]: *n* < *length w* ⟹ (*w* ⌢ *x*) *n* = *w*!*n*
⟨*proof*⟩

lemma *conc-snd*[*simp*]: ¬(*n* < *length w*) ⟹ (*w* ⌢ *x*) *n* = *x* (*n* − *length w*)
⟨*proof*⟩

lemma *iter-nth* [*simp*]: *0* < *length w* ⟹ *w*^ω *n* = *w*!(*n mod* (*length w*))
⟨*proof*⟩

lemma *conc-conc*[*simp*]: *u* ⌢ *v* ⌢ *w* = (*u* @ *v*) ⌢ *w* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *range-conc*[*simp*]: *range* (*w*$_1$ ⌢ *w*$_2$) = *set w*$_1$ ∪ *range w*$_2$
⟨*proof*⟩

lemma *iter-unroll*: *0* < *length w* ⟹ *w*^ω = *w* ⌢ *w*^ω
⟨*proof*⟩

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)*
⟨*proof*⟩

lemma *suffix-0* [*simp*]: *suffix 0 x = x*
⟨*proof*⟩

lemma *suffix-suffix* [*simp*]: *suffix m (suffix k x) = suffix (k+m) x*
⟨*proof*⟩

lemma *subsequence-append*: *prefix (i + j) w = prefix i w @ (w [i → i + j])*
⟨*proof*⟩

lemma *subsequence-drop*[*simp*]: *drop i (w [j → k]) = w [j + i → k]*
⟨*proof*⟩

lemma *subsequence-empty*[*simp*]: *w [i → j] = [] ⟷ j ≤ i*
⟨*proof*⟩

lemma *subsequence-length*[*simp*]: *length (subsequence w i j) = j − i*
⟨*proof*⟩

lemma *subsequence-nth*[*simp*]: *k < j − i ⟹ (w [i → j]) ! k = w (i + k)*
⟨*proof*⟩

lemma *subseq-to-zero*[*simp*]: *w[i→0] = []*
⟨*proof*⟩

lemma *subseq-to-smaller*[*simp*]: *i≥j ⟹ w[i→j] = []*
⟨*proof*⟩

lemma *subseq-to-Suc*[*simp*]: *i≤j ⟹ w [i → Suc j] = w [i → j] @ [w j]*
⟨*proof*⟩

lemma *subsequence-singleton*[*simp*]: *w [i → Suc i] = [w i]*
⟨*proof*⟩

lemma *subsequence-prefix-suffix*: *prefix (j − i) (suffix i w) = w [i → j]*
⟨*proof*⟩

lemma *prefix-suffix*: *x = prefix n x ⌢ (suffix n x)*
⟨*proof*⟩

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*⟩

lemma *set-subsequence*[*simp*]: *set* $(w[i \rightarrow j]) = w`\{i..<j\}$
⟨*proof*⟩

lemma *subsequence-take*[*simp*]: *take* i $(w\ [j \rightarrow k]) = w\ [j \rightarrow min\ (j + i)\ k]$
⟨*proof*⟩

lemma *subsequence-shift*[*simp*]: $(suffix\ i\ w)\ [j \rightarrow k] = w\ [i + j \rightarrow i + k]$
⟨*proof*⟩

lemma *suffix-subseq-join*[*simp*]: $i \leq j \Longrightarrow v\ [i \rightarrow j] \frown suffix\ j\ v = suffix\ i\ v$
⟨*proof*⟩

lemma *prefix-conc-fst*[*simp*]:
assumes $j \leq length\ w$
shows $prefix\ j\ (w \frown w') = take\ j\ w$
⟨*proof*⟩

lemma *prefix-conc-snd*[*simp*]:
assumes $n \geq length\ u$
shows $prefix\ n\ (u \frown v) = u\ @\ prefix\ (n - length\ u)\ v$
⟨*proof*⟩

lemma *prefix-conc-length*[*simp*]: $prefix\ (length\ w)\ (w \frown w') = w$
⟨*proof*⟩

lemma *suffix-conc-fst*[*simp*]:
assumes $n \leq length\ u$
shows $suffix\ n\ (u \frown v) = drop\ n\ u \frown v$
⟨*proof*⟩

lemma *suffix-conc-snd*[*simp*]:
assumes $n \geq length\ u$
shows $suffix\ n\ (u \frown v) = suffix\ (n - length\ u)\ v$
⟨*proof*⟩

lemma *suffix-conc-length*[*simp*]: $suffix\ (length\ w)\ (w \frown w') = w'$
⟨*proof*⟩

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*⟩

lemma *same-concat-eq*[*iff*]: $u \frown v = u \frown w \longleftrightarrow v = w$
⟨*proof*⟩

lemma *comp-concat*[*simp*]: $f \circ u \frown v = map\ f\ u \frown (f \circ v)$
⟨*proof*⟩

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*⟩

lemma *build-cons*[*simp*]: $(a \# u) \frown v = a \#\# u \frown v$
⟨*proof*⟩

lemma *build-append*[*simp*]: $(w\ @\ a \# u) \frown v = w \frown a \#\# u \frown v$
⟨*proof*⟩

lemma *build-first*[*simp*]: $w\ 0 \#\# suffix\ (Suc\ 0)\ w = w$
⟨*proof*⟩

lemma *build-split*[*intro*]: $w = w\ 0 \#\# suffix\ 1\ w$
⟨*proof*⟩

lemma *build-range*[*simp*]: $range\ (a \#\# w) = insert\ a\ (range\ w)$
⟨*proof*⟩

lemma *suffix-singleton-suffix*[*simp*]: $w\ i \#\# suffix\ (Suc\ i)\ w = suffix\ i\ w$
⟨*proof*⟩

Find the first occurrence of a letter from a given set

lemma *word-first-split-set*:
assumes $A \cap range\ w \neq \{\}$
obtains $u\ a\ v$ **where** $w = u \frown [a] \frown v\ A \cap set\ u = \{\}\ a \in A$
⟨*proof*⟩

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* :: $'a\ word \Rightarrow 'a\ set$
where $limit\ x \equiv \{a\ .\ \exists_\infty n\ .\ x\ n = a\}$

lemma *limit-iff-frequent*: $a \in limit\ x \longleftrightarrow (\exists_\infty n\ .\ x\ n = a)$

⟨*proof*⟩

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*: $(a \in \mathit{limit}\ x) = \mathit{infinite}\ (x -` \{a\})$
⟨*proof*⟩

lemma *two-in-limit-iff*:
$(\{a, b\} \subseteq \mathit{limit}\ x) =$
$((\exists\, n.\ x\ n = a\ ) \wedge (\forall\, n.\ x\ n = a \longrightarrow (\exists\, m{>}n.\ x\ m = b)) \wedge (\forall\, m.\ x\ m = b \longrightarrow (\exists\, n{>}m.\ x\ n = a)))$
(**is** *?lhs* = (*?r1* ∧ *?r2* ∧ *?r3*))
⟨*proof*⟩

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 $\exists\, a.\ a \in \mathit{limit}\ x$
⟨*proof*⟩

lemmas *limit-nonemptyE* = *limit-nonempty*[*THEN exE*]

lemma *limit-inter-INF*:
assumes *hyp*: $\mathit{limit}\ w \cap S \neq \{\}$
shows $\exists_\infty\ n.\ w\ n \in S$
⟨*proof*⟩

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* ($S \cap \mathit{range}\ w$)
shows $\exists\, a.\ a \in \mathit{limit}\ w \cap S$
⟨*proof*⟩

lemma *fin-ex-inf-eq-limit*: *finite* $A \Longrightarrow (\exists_\infty i.\ w\ i \in A) \longleftrightarrow \mathit{limit}\ w \cap A \neq \{\}$
⟨*proof*⟩

lemma *limit-in-range-suffix*: $\mathit{limit}\ x \subseteq \mathit{range}\ (\mathit{suffix}\ k\ x)$
⟨*proof*⟩

lemma *limit-in-range*: $\mathit{limit}\ r \subseteq \mathit{range}\ r$
⟨*proof*⟩

lemmas *limit-in-range-suffixD* = *limit-in-range-suffix*[*THEN subsetD*]

lemma *limit-subset*: $\mathit{limit}\ f \subseteq f\ `\ \{n..\}$

⟨*proof*⟩

theorem *limit-is-suffix*:
assumes *fin*: *finite* (*range x*)
shows ∃ *k*. *limit x* = *range* (*suffix k x*)
⟨*proof*⟩

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*⟩

theorem *limit-suffix* [*simp*]: *limit* (*suffix n x*) = *limit x*
⟨*proof*⟩

theorem *limit-iter* [*simp*]:
assumes *nempty*: *0* < *length w*
shows *limit* w^ω = *set w*
⟨*proof*⟩

lemma *limit-o* [*simp*]:
assumes *a*: *a* ∈ *limit w*
shows *f a* ∈ *limit* (*f* ∘ *w*)
⟨*proof*⟩

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* ∈ *limit* (*f* ∘ *w*)
shows ∃ *a* ∈ (*f* −‘ {*x*}). *a* ∈ *limit w*
⟨*proof*⟩

theorem *limit-inj* [*simp*]:
assumes *inj*: *inj f*
shows *limit* (*f* ∘ *w*) = *f* ‘ (*limit w*)
⟨*proof*⟩

lemma *limit-inter-empty*:
assumes *fin*: *finite* (*range w*)
assumes *hyp*: *limit w* ∩ *S* = {}
shows ∀$_\infty$*n*. *w n* ∉ *S*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

lemma *idx-sequence-inj*:
assumes *iseq*: *idx-sequence idx*
and *eq*: *idx m = idx n*
shows *m = n*
⟨*proof*⟩

lemma *idx-sequence-mono*:
assumes *iseq*: *idx-sequence idx*
and *m*: *m ≤ n*
shows *idx m ≤ idx n*
⟨*proof*⟩

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*)}
⟨*proof*⟩

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*⟩

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*⟩

lemma *idx-sequence-unique-interval*:
assumes *iseq*: *idx-sequence idx*
shows ∃! *k*. *n* ∈ {*idx k* ..< *idx* (*Suc k*) }
⟨*proof*⟩

Now we can define the piecewise construction of a word using an index sequence.

definition *merge* :: *'a word word* ⇒ *nat word* ⇒ *'a word*
where *merge ws idx* ≡ λ*n*. *let i* = *THE i*. *n* ∈ {*idx i* ..< *idx* (*Suc i*) } *in ws i n*

lemma *merge*:
assumes *idx*: *idx-sequence idx*
and *n*: *n* ∈ {*idx i* ..< *idx* (*Suc i*)}
shows *merge ws idx n* = *ws i n*
⟨*proof*⟩

lemma *merge0*:
assumes *idx*: *idx-sequence idx*
shows *merge ws idx 0* = *ws 0 0*
⟨*proof*⟩

lemma *merge-Suc*:
assumes *idx*: *idx-sequence idx*
and *n*: *n* ∈ {*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*⟩

end

⟨*ML*⟩

69 Canonical order on option type

theory *Option-ord*
imports *Option Main*
begin

notation
bot ($\bot$) **and**
top ($\top$) **and**
inf (**infixl** $\sqcap$ *70*) **and**
sup (**infixl** $\sqcup$ *65*) **and**
Inf ($\bigsqcap$- [*900*] *900*) **and**
Sup ($\bigsqcup$- [*900*] *900*)

syntax
-INF1 :: *pttrns* $\Rightarrow$ *'b* $\Rightarrow$ *'b* (($3\bigsqcap$-./ -) [*0*, *10*] *10*)
-INF :: *pttrn* $\Rightarrow$ *'a set* $\Rightarrow$ *'b* $\Rightarrow$ *'b* (($3\bigsqcap$-$\in$-./ -) [*0*, *0*, *10*] *10*)
-SUP1 :: *pttrns* $\Rightarrow$ *'b* $\Rightarrow$ *'b* (($3\bigsqcup$-./ -) [*0*, *10*] *10*)
-SUP :: *pttrn* $\Rightarrow$ *'a set* $\Rightarrow$ *'b* $\Rightarrow$ *'b* (($3\bigsqcup$-$\in$-./ -) [*0*, *0*, *10*] *10*)

instantiation *option* :: (*preorder*) *preorder*
begin

definition *less-eq-option* **where**
$x \leq y \longleftrightarrow$ (*case x of None* $\Rightarrow$ *True* | *Some x* $\Rightarrow$ (*case y of None* $\Rightarrow$ *False* | *Some y* $\Rightarrow$ $x \leq y$))

definition *less-option* **where**
$x < y \longleftrightarrow$ (*case y of None* $\Rightarrow$ *False* | *Some y* $\Rightarrow$ (*case x of None* $\Rightarrow$ *True* | *Some x* $\Rightarrow$ $x < y$))

lemma *less-eq-option-None* [*simp*]: *None* $\leq x$
⟨*proof*⟩

lemma *less-eq-option-None-code* [*code*]: *None* $\leq x \longleftrightarrow$ *True*
⟨*proof*⟩

lemma *less-eq-option-None-is-None*: $x \leq$ *None* $\Longrightarrow x =$ *None*
⟨*proof*⟩

lemma *less-eq-option-Some-None* [*simp*, *code*]: *Some x* $\leq$ *None* $\longleftrightarrow$ *False*
⟨*proof*⟩

lemma *less-eq-option-Some* [*simp*, *code*]: *Some x* $\leq$ *Some y* $\longleftrightarrow x \leq y$
⟨*proof*⟩

lemma *less-option-None* [*simp*, *code*]: $x <$ *None* $\longleftrightarrow$ *False*
⟨*proof*⟩

lemma *less-option-None-is-Some*: $None < x \Longrightarrow \exists z.\ x = Some\ z$
⟨*proof*⟩

lemma *less-option-None-Some* [*simp*]: $None < Some\ x$
⟨*proof*⟩

lemma *less-option-None-Some-code* [*code*]: $None < Some\ x \longleftrightarrow True$
⟨*proof*⟩

lemma *less-option-Some* [*simp*, *code*]: $Some\ x < Some\ y \longleftrightarrow x < y$
⟨*proof*⟩

instance
⟨*proof*⟩

end

instance *option* :: (*order*) *order*
⟨*proof*⟩

instance *option* :: (*linorder*) *linorder*
⟨*proof*⟩

instantiation *option* :: (*order*) *order-bot*
begin

definition *bot-option* **where** $\bot = None$

instance
⟨*proof*⟩

end

instantiation *option* :: (*order-top*) *order-top*
begin

definition *top-option* **where** $\top = Some\ \top$

instance
⟨*proof*⟩

end

instance *option* :: (*wellorder*) *wellorder*
⟨*proof*⟩

instantiation *option* :: (*inf*) *inf*
begin

definition *inf-option* **where**
$x \sqcap y = (\mathit{case}\ x\ \mathit{of}\ \mathit{None} \Rightarrow \mathit{None} \mid \mathit{Some}\ x \Rightarrow (\mathit{case}\ y\ \mathit{of}\ \mathit{None} \Rightarrow \mathit{None} \mid \mathit{Some}\ y \Rightarrow \mathit{Some}\ (x \sqcap y)))$

lemma *inf-None-1* [*simp*, *code*]: $\mathit{None} \sqcap y = \mathit{None}$
⟨*proof*⟩

lemma *inf-None-2* [*simp*, *code*]: $x \sqcap \mathit{None} = \mathit{None}$
⟨*proof*⟩

lemma *inf-Some* [*simp*, *code*]: $\mathit{Some}\ x \sqcap \mathit{Some}\ y = \mathit{Some}\ (x \sqcap y)$
⟨*proof*⟩

instance ⟨*proof*⟩

end

instantiation *option* :: (*sup*) *sup*
begin

definition *sup-option* **where**
$x \sqcup y = (\mathit{case}\ x\ \mathit{of}\ \mathit{None} \Rightarrow y \mid \mathit{Some}\ x' \Rightarrow (\mathit{case}\ y\ \mathit{of}\ \mathit{None} \Rightarrow x \mid \mathit{Some}\ y \Rightarrow \mathit{Some}\ (x' \sqcup y)))$

lemma *sup-None-1* [*simp*, *code*]: $\mathit{None} \sqcup y = y$
⟨*proof*⟩

lemma *sup-None-2* [*simp*, *code*]: $x \sqcup \mathit{None} = x$
⟨*proof*⟩

lemma *sup-Some* [*simp*, *code*]: $\mathit{Some}\ x \sqcup \mathit{Some}\ y = \mathit{Some}\ (x \sqcup y)$
⟨*proof*⟩

instance ⟨*proof*⟩

end

instance *option* :: (*semilattice-inf*) *semilattice-inf*
⟨*proof*⟩

instance *option* :: (*semilattice-sup*) *semilattice-sup*
⟨*proof*⟩

instance *option* :: (*lattice*) *lattice* ⟨*proof*⟩

instance *option* :: (*lattice*) *bounded-lattice-bot* ⟨*proof*⟩

instance *option* :: (*bounded-lattice-top*) *bounded-lattice-top* ⟨*proof*⟩

instance *option* :: (*bounded-lattice-top*) *bounded-lattice* ⟨*proof*⟩

instance *option* :: (*distrib-lattice*) *distrib-lattice*
⟨*proof*⟩

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*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *singleton-None-Sup* [*simp*]: $\bigsqcup$ {*None*} = *None*
  ⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *Some-Inf*:
  Some ($\bigsqcap A$) = $\bigsqcap$ (*Some* ‘ *A*)
  ⟨*proof*⟩

lemma *Some-Sup*:
  A ≠ {} ⟹ *Some* ($\bigsqcup A$) = $\bigsqcup$ (*Some* ‘ *A*)
  ⟨*proof*⟩

lemma *Some-INF*:
  Some ($\bigsqcap x{\in}A.\ f\ x$) = ($\bigsqcap x{\in}A.$ *Some* (*f x*))
  ⟨*proof*⟩

lemma *Some-SUP*:
  A ≠ {} ⟹ *Some* ($\bigsqcup x{\in}A.\ f\ x$) = ($\bigsqcup x{\in}A.$ *Some* (*f x*))
  ⟨*proof*⟩

instance *option* :: (*complete-distrib-lattice*) *complete-distrib-lattice*
⟨*proof*⟩

instance *option* :: (*complete-linorder*) *complete-linorder* ⟨*proof*⟩

no-notation
bot ($\bot$) **and**
top ($\top$) **and**
inf (**infixl** $\sqcap$ *70*) **and**
sup (**infixl** $\sqcup$ *65*) **and**
Inf ($\bigsqcap$- [*900*] *900*) **and**
Sup ($\bigsqcup$- [*900*] *900*)

no-syntax
-INF1 :: *pttrns* $\Rightarrow$ *'b* $\Rightarrow$ *'b* (($3\bigsqcap$-./ -) [*0*, *10*] *10*)
-INF :: *pttrn* $\Rightarrow$ *'a set* $\Rightarrow$ *'b* $\Rightarrow$ *'b* (($3\bigsqcap$-$\in$-./ -) [*0*, *0*, *10*] *10*)
-SUP1 :: *pttrns* $\Rightarrow$ *'b* $\Rightarrow$ *'b* (($3\bigsqcup$-./ -) [*0*, *10*] *10*)
-SUP :: *pttrn* $\Rightarrow$ *'a set* $\Rightarrow$ *'b* $\Rightarrow$ *'b* (($3\bigsqcup$-$\in$-./ -) [*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* $\Rightarrow$ *'a*

primrec *join* :: *'a future* $\Rightarrow$ *'a* **where**
join (*fork f*) = *f* ()

lemma *future-eqI* [*intro*!]:
assumes *join f* = *join g*
shows *f* = *g*
⟨*proof*⟩

code-printing
type-constructor *future* $\rightharpoonup$ (*Eval*) - *future*
| **constant** *fork* $\rightharpoonup$ (*Eval*) *Future.fork*
| **constant** *join* $\rightharpoonup$ (*Eval*) *Future.join*

code-reserved *Eval Future future*

70.2 Parallel lists

definition *map* :: (*'a* $\Rightarrow$ *'b*) $\Rightarrow$ *'a list* $\Rightarrow$ *'b list* **where**
[*simp*]: *map* = *List.map*

definition *forall* :: $('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow bool$ **where**
forall = *list-all*

lemma *forall-all* [*simp*]:
forall P xs $\longleftrightarrow$ $(\forall x{\in}set\ xs.\ P\ x)$
⟨*proof*⟩

definition *exists* :: $('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow bool$ **where**
exists = *list-ex*

lemma *exists-ex* [*simp*]:
exists P xs $\longleftrightarrow$ $(\exists x{\in}set\ xs.\ P\ x)$
⟨*proof*⟩

code-printing
constant *map* $\rightharpoonup$ (*Eval*) *Par'-List.map*
| **constant** *forall* $\rightharpoonup$ (*Eval*) *Par'-List.forall*
| **constant** *exists* $\rightharpoonup$ (*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* :: $'a\ list \Rightarrow 'a\ list \Rightarrow bool$ (- $<\sim\sim>$ - [*50*, *50*] *50*)
where
Nil [*intro*!]: [] $<\sim\sim>$ []
| *swap* [*intro*!]: $y \# x \# l <\sim\sim> x \# y \# l$
| *Cons* [*intro*!]: *xs* $<\sim\sim>$ *ys* $\Longrightarrow$ $z \# xs <\sim\sim> z \# ys$
| *trans* [*intro*]: *xs* $<\sim\sim>$ *ys* $\Longrightarrow$ *ys* $<\sim\sim>$ *zs* $\Longrightarrow$ *xs* $<\sim\sim>$ *zs*

proposition *perm-refl* [*iff*]: $l <\sim\sim> l$
⟨*proof*⟩

71.1 Some examples of rule induction on permutations

proposition *xperm-empty-imp*: [] $<\sim\sim>$ *ys* $\Longrightarrow$ *ys* = []
⟨*proof*⟩

This more general theorem is easier to understand!

proposition *perm-length*: *xs* $<\sim\sim>$ *ys* $\Longrightarrow$ *length xs* = *length ys*

⟨*proof*⟩

proposition *perm-empty-imp*: [] <~~> *xs* ⟹ *xs* = []
⟨*proof*⟩

proposition *perm-sym*: *xs* <~~> *ys* ⟹ *ys* <~~> *xs*
⟨*proof*⟩

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*
⟨*proof*⟩

proposition *perm-append-swap*: *xs* @ *ys* <~~> *ys* @ *xs*
⟨*proof*⟩

proposition *perm-append-single*: *a* # *xs* <~~> *xs* @ [*a*]
⟨*proof*⟩

proposition *perm-rev*: *rev xs* <~~> *xs*
⟨*proof*⟩

proposition *perm-append1*: *xs* <~~> *ys* ⟹ *l* @ *xs* <~~> *l* @ *ys*
⟨*proof*⟩

proposition *perm-append2*: *xs* <~~> *ys* ⟹ *xs* @ *l* <~~> *ys* @ *l*
⟨*proof*⟩

71.3 Further results

proposition *perm-empty* [*iff*]: [] <~~> *xs* ⟷ *xs* = []
⟨*proof*⟩

proposition *perm-empty2* [*iff*]: *xs* <~~> [] ⟷ *xs* = []
⟨*proof*⟩

proposition *perm-sing-imp*: *ys* <~~> *xs* ⟹ *xs* = [*y*] ⟹ *ys* = [*y*]
⟨*proof*⟩

proposition *perm-sing-eq* [*iff*]: *ys* <~~> [*y*] ⟷ *ys* = [*y*]
⟨*proof*⟩

proposition *perm-sing-eq2* [*iff*]: [*y*] <~~> *ys* ⟷ *ys* = [*y*]
⟨*proof*⟩

71.4 Removing elements

proposition *perm-remove*: *x* ∈ *set ys* ⟹ *ys* <~~> *x* # *remove1 x ys*

⟨*proof*⟩

Congruence rule

proposition *perm-remove-perm*: *xs* <~~> *ys* ⟹ *remove1 z xs* <~~> *remove1 z ys*
⟨*proof*⟩

proposition *remove-hd* [*simp*]: *remove1 z* (*z* # *xs*) = *xs*
⟨*proof*⟩

proposition *cons-perm-imp-perm*: *z* # *xs* <~~> *z* # *ys* ⟹ *xs* <~~> *ys*
⟨*proof*⟩

proposition *cons-perm-eq* [*iff*]: *z*#*xs* <~~> *z*#*ys* ⟷ *xs* <~~> *ys*
⟨*proof*⟩

proposition *append-perm-imp-perm*: *zs* @ *xs* <~~> *zs* @ *ys* ⟹ *xs* <~~> *ys*
⟨*proof*⟩

proposition *perm-append1-eq* [*iff*]: *zs* @ *xs* <~~> *zs* @ *ys* ⟷ *xs* <~~> *ys*
⟨*proof*⟩

proposition *perm-append2-eq* [*iff*]: *xs* @ *zs* <~~> *ys* @ *zs* ⟷ *xs* <~~> *ys*
⟨*proof*⟩

theorem *mset-eq-perm*: *mset xs* = *mset ys* ⟷ *xs* <~~> *ys*
⟨*proof*⟩

proposition *mset-le-perm-append*: *mset xs* ≤# *mset ys* ⟷ (∃ *zs*. *xs* @ *zs* <~~> *ys*)
⟨*proof*⟩

proposition *perm-set-eq*: *xs* <~~> *ys* ⟹ *set xs* = *set ys*
⟨*proof*⟩

proposition *perm-distinct-iff*: *xs* <~~> *ys* ⟹ *distinct xs* = *distinct ys*
⟨*proof*⟩

theorem *eq-set-perm-remdups*: *set xs* = *set ys* ⟹ *remdups xs* <~~> *remdups ys*
⟨*proof*⟩

proposition *perm-remdups-iff-eq-set*: *remdups x* <~~> *remdups y* ⟷ *set x* = *set y*
⟨*proof*⟩

theorem *permutation-Ex-bij*:
assumes *xs* <~~> *ys*
shows ∃*f*. *bij-betw f* {..<*length xs*} {..<*length ys*} ∧ (∀ *i*<*length xs*. *xs* ! *i* = *ys* ! (*f i*))

⟨*proof*⟩

proposition *perm-finite*: *finite* {*B*. *B* <~~> *A*}
⟨*proof*⟩

proposition *perm-swap*:
  assumes *i* < *length xs j* < *length xs*
  shows *xs*[*i* := *xs* ! *j*, *j* := *xs* ! *i*] <~~> *xs*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *inv-swap-id*:
  inv (*Fun.swap a b id*) = *Fun.swap a b id*
⟨*proof*⟩

lemma *swap-id-eq*:
  Fun.swap a b id x = (*if x* = *a then b else if x* = *b then a else x*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *permutes-image*: *p permutes S* ⟹ *p* ‘ *S* = *S*
⟨*proof*⟩

lemma *permutes-inj*: *p permutes S* ⟹ *inj p*
⟨*proof*⟩

lemma *permutes-surj*: *p permutes s* ⟹ *surj p*
⟨*proof*⟩

lemma *permutes-bij*: *p permutes s* $\Longrightarrow$ *bij p*
⟨*proof*⟩

lemma *permutes-imp-bij*: *p permutes S* $\Longrightarrow$ *bij-betw p S S*
⟨*proof*⟩

lemma *bij-imp-permutes*: *bij-betw p S S* $\Longrightarrow$ $(\bigwedge x.\ x \notin S \Longrightarrow p\ x = x) \Longrightarrow$ *p permutes S*
⟨*proof*⟩

lemma *permutes-inv-o*:
assumes *pS*: *p permutes S*
shows $p \circ$ *inv p* $= id$
and *inv p* $\circ\ p = id$
⟨*proof*⟩

lemma *permutes-inverses*:
fixes *p* :: $'a \Rightarrow 'a$
assumes *pS*: *p permutes S*
shows *p* (*inv p x*) $= x$
and *inv p* (*p x*) $= x$
⟨*proof*⟩

lemma *permutes-subset*: *p permutes S* $\Longrightarrow S \subseteq T \Longrightarrow$ *p permutes T*
⟨*proof*⟩

lemma *permutes-empty*[*simp*]: *p permutes* $\{\} \longleftrightarrow p = id$
⟨*proof*⟩

lemma *permutes-sing*[*simp*]: *p permutes* $\{a\} \longleftrightarrow p = id$
⟨*proof*⟩

lemma *permutes-univ*: *p permutes UNIV* $\longleftrightarrow (\forall y.\ \exists!x.\ p\ x = y)$
⟨*proof*⟩

lemma *permutes-inv-eq*: *p permutes S* $\Longrightarrow$ *inv p y* $= x \longleftrightarrow p\ x = y$
⟨*proof*⟩

lemma *permutes-swap-id*: $a \in S \Longrightarrow b \in S \Longrightarrow$ *Fun.swap a b id permutes S*
⟨*proof*⟩

lemma *permutes-superset*: *p permutes S* $\Longrightarrow (\forall x \in S - T.\ p\ x = x) \Longrightarrow$ *p permutes T*
⟨*proof*⟩

72.3 Group properties

lemma *permutes-id*: *id permutes S*

⟨*proof*⟩

lemma *permutes-compose*: *p permutes S* ⟹ *q permutes S* ⟹ *q* ∘ *p permutes S*
⟨*proof*⟩

lemma *permutes-inv*:
assumes *pS*: *p permutes S*
shows *inv p permutes S*
⟨*proof*⟩

lemma *permutes-inv-inv*:
assumes *pS*: *p permutes S*
shows *inv* (*inv p*) = *p*
⟨*proof*⟩

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* ∘ *p permutes S*
⟨*proof*⟩

lemma *permutes-insert*: {*p*. *p permutes* (*insert a S*)} =
(λ(*b*,*p*). *Fun.swap a b id* ∘ *p*) ' {(*b*,*p*). *b* ∈ *insert a S* ∧ *p* ∈ {*p*. *p permutes S*}}
⟨*proof*⟩

lemma *card-permutations*:
assumes *Sn*: *card S* = *n*
and *fS*: *finite S*
shows *card* {*p*. *p permutes S*} = *fact n*
⟨*proof*⟩

lemma *finite-permutations*:
assumes *fS*: *finite S*
shows *finite* {*p*. *p permutes S*}
⟨*proof*⟩

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* ∘ *p*) *S*
⟨*proof*⟩

72.6 Various combinations of transpositions with 2, 1 and 0 common elements

lemma *swap-id-common*: *a* ≠ *c* ⟹ *b* ≠ *c* ⟹
Fun.swap a b id ∘ *Fun.swap a c id* = *Fun.swap b c id* ∘ *Fun.swap a b id*
⟨*proof*⟩

lemma *swap-id-common'*: $a \neq b \Longrightarrow a \neq c \Longrightarrow$
Fun.swap a c id $\circ$ *Fun.swap b c id* = *Fun.swap b c id* $\circ$ *Fun.swap a b id*
⟨*proof*⟩

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 $\circ$ *Fun.swap c d id* = *Fun.swap c d id* $\circ$ *Fun.swap a b id*
⟨*proof*⟩

72.7 Permutations as transposition sequences

inductive *swapidseq* :: *nat* ⇒ ('*a* ⇒ '*a*) ⇒ *bool*
where
id[*simp*]: *swapidseq 0 id*
| *comp-Suc*: *swapidseq n p* $\Longrightarrow$ *a* $\neq$ *b* $\Longrightarrow$ *swapidseq* (*Suc n*) (*Fun.swap a b id* $\circ$ *p*)

declare *id*[*unfolded id-def*, *simp*]

definition *permutation p* $\longleftrightarrow$ ($\exists$ *n*. *swapidseq n p*)

72.8 Some closure properties of the set of permutations, with lengths

lemma *permutation-id*[*simp*]: *permutation id*
⟨*proof*⟩

declare *permutation-id*[*unfolded id-def*, *simp*]

lemma *swapidseq-swap*: *swapidseq* (*if a* = *b then 0 else 1*) (*Fun.swap a b id*)
⟨*proof*⟩

lemma *permutation-swap-id*: *permutation* (*Fun.swap a b id*)
⟨*proof*⟩

lemma *swapidseq-comp-add*: *swapidseq n p* $\Longrightarrow$ *swapidseq m q* $\Longrightarrow$ *swapidseq* (*n* + *m*) (*p* $\circ$ *q*)
⟨*proof*⟩

lemma *permutation-compose*: *permutation p* $\Longrightarrow$ *permutation q* $\Longrightarrow$ *permutation* (*p* $\circ$ *q*)
⟨*proof*⟩

lemma *swapidseq-endswap*: *swapidseq n p* $\Longrightarrow$ *a* $\neq$ *b* $\Longrightarrow$ *swapidseq* (*Suc n*) (*p* $\circ$ *Fun.swap a b id*)
⟨*proof*⟩

lemma *swapidseq-inverse-exists*: *swapidseq n p* $\Longrightarrow$ $\exists$ *q*. *swapidseq n q* $\wedge$ *p* $\circ$ *q* = *id* $\wedge$ *q* $\circ$ *p* = *id*

⟨*proof*⟩

lemma *swapidseq-inverse*:
assumes *H*: *swapidseq n p*
shows *swapidseq n* (*inv p*)
⟨*proof*⟩

lemma *permutation-inverse*: *permutation p* ⟹ *permutation* (*inv p*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *swapidseq-id-iff* [*simp*]: *swapidseq 0 p* ⟷ *p* = *id*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *swapidseq-identity-even*:
assumes *swapidseq n* (*id* :: *'a* ⇒ *'a*)
shows *even n*
⟨*proof*⟩

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*⟩

lemma *evenperm-unique*:
assumes *p*: *swapidseq n p*
and *n*:*even n* = *b*
shows *evenperm p* = *b*
⟨*proof*⟩

72.11 And it has the expected composition properties

lemma *evenperm-id*[*simp*]: *evenperm id* = *True*
⟨*proof*⟩

lemma *evenperm-swap*: *evenperm* (*Fun.swap a b id*) = (*a* = *b*)
⟨*proof*⟩

lemma *evenperm-comp*:
assumes *p*: *permutation p*
and *q*:*permutation q*
shows *evenperm* (*p* ∘ *q*) = (*evenperm p* = *evenperm q*)
⟨*proof*⟩

lemma *evenperm-inv*:
assumes *p*: *permutation p*
shows *evenperm* (*inv p*) = *evenperm p*
⟨*proof*⟩

72.12 A more abstract characterization of permutations

lemma *bij-iff*: *bij f* ⟷ (∀ *x*. ∃!*y*. *f y* = *x*)
⟨*proof*⟩

lemma *permutation-bijective*:
assumes *p*: *permutation p*
shows *bij p*
⟨*proof*⟩

lemma *permutation-finite-support*:
assumes *p*: *permutation p*
shows *finite* {*x*. *p x* ≠ *x*}
⟨*proof*⟩

lemma *bij-inv-eq-iff*: *bij p* ⟹ *x* = *inv p y* ⟷ *p x* = *y*
⟨*proof*⟩

lemma *bij-swap-comp*:
assumes *bp*: *bij p*
shows *Fun.swap a b id ∘ p = Fun.swap (inv p a) (inv p b) p*
⟨*proof*⟩

lemma *bij-swap-ompose-bij*: *bij p ⟹ bij (Fun.swap a b id ∘ p)*
⟨*proof*⟩

lemma *permutation-lemma*:
assumes *fS*: *finite S*
and *p*: *bij p*
and *pS*: *∀ x. x∉ S ⟶ p x = x*
shows *permutation p*
⟨*proof*⟩

lemma *permutation*: *permutation p ⟷ bij p ∧ finite {x. p x ≠ x}*
(**is** *?lhs ⟷ ?b ∧ ?f*)
⟨*proof*⟩

lemma *permutation-inverse-works*:
assumes *p*: *permutation p*
shows *inv p ∘ p = id*
and *p ∘ inv p = id*
⟨*proof*⟩

lemma *permutation-inverse-compose*:
assumes *p*: *permutation p*
and *q*: *permutation q*
shows *inv (p ∘ q) = inv q ∘ inv p*
⟨*proof*⟩

72.13 Relation to "permutes"

lemma *permutation-permutes*: *permutation p ⟷ (∃ S. finite S ∧ p permutes S)*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *sign-id*: *sign id = 1*
⟨*proof*⟩

lemma *sign-inverse*: *permutation p ⟹ sign (inv p) = sign p*
⟨*proof*⟩

lemma *sign-compose*: *permutation p ⟹ permutation q ⟹ sign (p ∘ q) = sign p * sign q*
⟨*proof*⟩

lemma *sign-swap-id*: *sign (Fun.swap a b id) = (if a = b then 1 else −1)*
⟨*proof*⟩

lemma *sign-idempotent*: *sign p * sign p = 1*
⟨*proof*⟩

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*⟩

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*⟩

lemma *image-inverse-permutations*: *{inv p |p. p permutes S} = {p. p permutes S}*
⟨*proof*⟩

lemma *image-compose-permutations-left*:
assumes *q*: *q permutes S*
shows *{q ∘ p | p. p permutes S} = {p . p permutes S}*
⟨*proof*⟩

lemma *image-compose-permutations-right*:
assumes *q*: *q permutes S*
shows *{p ∘ q | p. p permutes S} = {p . p permutes S}*
⟨*proof*⟩

lemma *permutes-in-seg*: *p permutes {1 ..n} ⟹ i ∈ {1..n} ⟹ 1 ≤ p i ∧ p i ≤ n*

⟨*proof*⟩

lemma *setsum-permutations-inverse*:
setsum f {*p*. *p permutes S*} = *setsum* (λ*p*. *f*(*inv p*)) {*p*. *p permutes S*}
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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*⟩

lemma *sum-permutations-compose-right*:
assumes *q*: *q permutes S*
shows *setsum f* {*p*. *p permutes S*} = *setsum* (λ*p*. *f*(*p* ∘ *q*)) {*p*. *p permutes S*}
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

72.17 Sum over a set of permutations (could generalize to iteration)

lemma *setsum-over-permutations-insert*:
assumes *fS*: *finite S*
and *aS*: *a* ∉ *S*
shows *setsum f* {*p*. *p permutes* (*insert a S*)} =
setsum (λ*b*. *setsum* (λ*q*. *f* (*Fun.swap a b id* ∘ *q*)) {*p*. *p permutes S*}) (*insert a S*)
⟨*proof*⟩

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* ≠ *0*
shows $a * x^2 + b * x + c = 0 \longleftrightarrow (2 * a * x + b)^2 = discrim\ a\ b\ c$
⟨*proof*⟩

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*⟩

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*⟩

lemma *divide-non-zero*:
fixes $x\ y\ z$ :: *real*
assumes $x \neq 0$
shows $x * y = z \longleftrightarrow y = z\ /\ x$
⟨*proof*⟩

lemma *discriminant-nonneg*:
fixes $a\ b\ c\ x$ :: *real*
assumes $a \neq 0$
and *discrim a b c* ≥ 0
shows $a * x^2 + b * x + c = 0 \longleftrightarrow$
$x = (-b + sqrt\ (discrim\ a\ b\ c))\ /\ (2 * a) \vee$
$x = (-b - sqrt\ (discrim\ a\ b\ c))\ /\ (2 * a)$
⟨*proof*⟩

lemma *discriminant-zero*:
fixes $a\ b\ c\ x$ :: *real*
assumes $a \neq 0$
and *discrim a b c* $= 0$
shows $a * x^2 + b * x + c = 0 \longleftrightarrow x = -b\ /\ (2 * a)$
⟨*proof*⟩

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*⟩

lemma *discriminant-nonneg-ex*:
fixes $a\ b\ c$ :: *real*
assumes $a \neq 0$
and *discrim a b c* ≥ 0
shows $\exists\ x.\ a * x^2 + b * x + c = 0$
⟨*proof*⟩

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*⟩

lemma *discriminant-pos-distinct*:
fixes $a\ b\ c\ x :: real$
assumes $a \neq 0$ **and** $discrim\ a\ b\ c > 0$
shows $\exists\ y.\ x \neq y \wedge a * y^2 + b * y + c = 0$
⟨*proof*⟩

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 $rel\text{-}vset\ R\ xs\ ys \equiv \forall\ x\ y.\ R\ x\ y \longrightarrow x \in xs \longleftrightarrow y \in ys$

lemma *rel-vset-eq* [*id-simps*]:
$rel\text{-}vset\ op = = op =$
⟨*proof*⟩

lemma *rel-vset-equivp*:
assumes *e*: *equivp R*
shows $rel\text{-}vset\ R\ xs\ ys \longleftrightarrow xs = ys \wedge (\forall\ x\ y.\ x \in xs \longrightarrow R\ x\ y \longrightarrow y \in xs)$
⟨*proof*⟩

lemma *set-quotient* [*quot-thm*]:
assumes *Quotient3 R Abs Rep*
shows *Quotient3* (*rel-vset R*) (*vimage Rep*) (*vimage Abs*)
⟨*proof*⟩

declare [[*mapQ3 set* = (*rel-vset*, *set-quotient*)]]

lemma *empty-set-rsp*[*quot-respect*]:
rel-vset R {} {}
⟨*proof*⟩

lemma *collect-rsp*[*quot-respect*]:
assumes *Quotient3 R Abs Rep*
shows ((*R* ===> *op* =) ===> *rel-vset R*) *Collect Collect*
⟨*proof*⟩

lemma *collect-prs*[*quot-preserve*]:
assumes *Quotient3 R Abs Rep*
shows ((*Abs* ---> *id*) ---> *op* -‘ *Rep*) *Collect* = *Collect*
⟨*proof*⟩

lemma *union-rsp*[*quot-respect*]:
assumes *Quotient3 R Abs Rep*
shows (*rel-vset R* ===> *rel-vset R* ===> *rel-vset R*) *op* ∪ *op* ∪
⟨*proof*⟩

lemma *union-prs*[*quot-preserve*]:
assumes *Quotient3 R Abs Rep*
shows (*op* -‘ *Abs* ---> *op* -‘ *Abs* ---> *op* -‘ *Rep*) *op* ∪ = *op* ∪
⟨*proof*⟩

lemma *diff-rsp*[*quot-respect*]:
assumes *Quotient3 R Abs Rep*
shows (*rel-vset R* ===> *rel-vset R* ===> *rel-vset R*) *op* − *op* −
⟨*proof*⟩

lemma *diff-prs*[*quot-preserve*]:
assumes *Quotient3 R Abs Rep*
shows (*op* -‘ *Abs* ---> *op* -‘ *Abs* ---> *op* -‘ *Rep*) *op* − = *op* −
⟨*proof*⟩

lemma *inter-rsp*[*quot-respect*]:
assumes *Quotient3 R Abs Rep*
shows (*rel-vset R* ===> *rel-vset R* ===> *rel-vset R*) *op* ∩ *op* ∩
⟨*proof*⟩

lemma *inter-prs*[*quot-preserve*]:
assumes *Quotient3 R Abs Rep*
shows (*op* -‘ *Abs* ---> *op* -‘ *Abs* ---> *op* -‘ *Rep*) *op* ∩ = *op* ∩

⟨*proof*⟩

lemma *mem-prs*[*quot-preserve*]:
assumes *Quotient3 R Abs Rep*
shows (*Rep* −−−> *op* −‘ *Abs* −−−> *id*) *op* ∈ = *op* ∈
⟨*proof*⟩

lemma *mem-rsp*[*quot-respect*]:
shows (*R* ===> *rel-vset R* ===> *op* =) *op* ∈ *op* ∈
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *rel-prod-eq* [*id-simps*]:
shows *rel-prod* (*op* =) (*op* =) = (*op* =)
⟨*proof*⟩

lemma *prod-equivp* [*quot-equiv*]:
assumes *equivp R1*
assumes *equivp R2*
shows *equivp* (*rel-prod R1 R2*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *fst-rsp* [*quot-respect*]:
assumes *Quotient3 R1 Abs1 Rep1*
assumes *Quotient3 R2 Abs2 Rep2*
shows (*rel-prod R1 R2 ===> R1*) *fst fst*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *snd-rsp* [*quot-respect*]:
assumes *Quotient3 R1 Abs1 Rep1*
assumes *Quotient3 R2 Abs2 Rep2*
shows (*rel-prod R1 R2 ===> R2*) *snd snd*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *case-prod-rsp* [*quot-respect*]:
shows ((*R1 ===> R2 ===> (op =)) ===> (rel-prod R1 R2) ===> (op =)) case-prod case-prod*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma [*quot-respect*]:
shows ((*R2 ===> R2 ===> op =) ===> (R1 ===> R1 ===> op =) ===>*
rel-prod R2 R1 ===> rel-prod R2 R1 ===> op =) rel-prod rel-prod
⟨*proof*⟩

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*
⟨*proof*⟩

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*))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *rel-option-map2*:
rel-option R x (*map-option f y*) ⟷ *rel-option* (λ*x y*. *R x* (*f y*)) *x y*
⟨*proof*⟩

declare
map-option.id [*id-simps*]
option.rel-eq [*id-simps*]

lemma *reflp-rel-option*:
reflp R ⟹ *reflp* (*rel-option R*)
⟨*proof*⟩

lemma *option-symp*:
symp R ⟹ *symp* (*rel-option R*)
⟨*proof*⟩

lemma *option-transp*:
transp R ⟹ *transp* (*rel-option R*)
⟨*proof*⟩

lemma *option-equivp* [*quot-equiv*]:
equivp R $\Longrightarrow$ *equivp* (*rel-option R*)
⟨*proof*⟩

lemma *option-quotient* [*quot-thm*]:
assumes *Quotient3 R Abs Rep*
shows *Quotient3* (*rel-option R*) (*map-option Abs*) (*map-option Rep*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *option-Some-rsp* [*quot-respect*]:
assumes *q*: *Quotient3 R Abs Rep*
shows (*R* ===> *rel-option R*) *Some Some*
⟨*proof*⟩

lemma *option-None-prs* [*quot-preserve*]:
assumes *q*: *Quotient3 R Abs Rep*
shows *map-option Abs None* = *None*
⟨*proof*⟩

lemma *option-Some-prs* [*quot-preserve*]:
assumes *q*: *Quotient3 R Abs Rep*
shows (*Rep* −−−> *map-option Abs*) *Some* = *Some*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *list-all2-eq* [*id-simps*]:
list-all2 (*op* =) = (*op* =)
⟨*proof*⟩

lemma *reflp-list-all2*:
assumes *reflp R*
shows *reflp* (*list-all2 R*)
⟨*proof*⟩

lemma *list-symp*:
assumes *symp R*
shows *symp* (*list-all2 R*)
⟨*proof*⟩

lemma *list-transp*:
assumes *transp R*
shows *transp* (*list-all2 R*)
⟨*proof*⟩

lemma *list-equivp* [*quot-equiv*]:
equivp R ⟹ *equivp* (*list-all2 R*)
⟨*proof*⟩

lemma *list-quotient3* [*quot-thm*]:
assumes *Quotient3 R Abs Rep*
shows *Quotient3* (*list-all2 R*) (*map Abs*) (*map Rep*)
⟨*proof*⟩

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 #*)
⟨*proof*⟩

lemma *cons-rsp* [*quot-respect*]:
assumes *q*: *Quotient3 R Abs Rep*
shows (*R* ===> *list-all2 R* ===> *list-all2 R*) (*op #*) (*op #*)
⟨*proof*⟩

lemma *nil-prs* [*quot-preserve*]:
assumes *q*: *Quotient3 R Abs Rep*
shows *map Abs* [] = []
⟨*proof*⟩

lemma *nil-rsp* [*quot-respect*]:
assumes *q*: *Quotient3 R Abs Rep*
shows *list-all2 R* [] []
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *foldl-rsp*[*quot-respect*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
and *q2*: *Quotient3 R2 Abs2 Rep2*

shows $((R1 ===> R2 ===> R1) ===> R1 ===> list\text{-}all2\ R2 ===> R1)$ *foldl foldl*
⟨*proof*⟩

lemma *foldr-rsp*[*quot-respect*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
and *q2*: *Quotient3 R2 Abs2 Rep2*
shows $((R1 ===> R2 ===> R2) ===> list\text{-}all2\ R1 ===> R2 ===> R2)$ *foldr foldr*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma [*quot-respect*]:
$((R ===> R ===> op =) ===> list\text{-}all2\ R ===> list\text{-}all2\ R ===> op =)$ *list-all2 list-all2*
⟨*proof*⟩

lemma [*quot-preserve*]:
assumes *a*: *Quotient3 R abs1 rep1*
shows $((abs1 ---> abs1 ---> id) ---> map\ rep1 ---> map\ rep1 ---> id)\ list\text{-}all2 = list\text{-}all2$
⟨*proof*⟩

lemma [*quot-preserve*]:
assumes *a*: *Quotient3 R abs1 rep1*
shows $(list\text{-}all2\ ((rep1 ---> rep1 ---> id)\ R)\ l\ m) = (l = m)$
⟨*proof*⟩

lemma *list-all2-find-element*:
assumes *a*: $x \in set\ a$
and *b*: *list-all2 R a b*
shows $\exists y.\ (y \in set\ b \wedge R\ x\ y)$
⟨*proof*⟩

lemma *list-all2-refl*:
assumes *a*: $\bigwedge x\ y.\ R\ x\ y = (R\ x = R\ y)$
shows *list-all2 R x x*
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

lemma *map-sum-id* [*id-simps*]:
map-sum id id = id
⟨*proof*⟩

lemma *rel-sum-eq* [*id-simps*]:
rel-sum (op =) (op =) = (op =)
⟨*proof*⟩

lemma *reflp-rel-sum*:
reflp R1 ⟹ reflp R2 ⟹ reflp (rel-sum R1 R2)
⟨*proof*⟩

lemma *sum-symp*:
symp R1 ⟹ symp R2 ⟹ symp (rel-sum R1 R2)
⟨*proof*⟩

lemma *sum-transp*:
transp R1 ⟹ transp R2 ⟹ transp (rel-sum R1 R2)
⟨*proof*⟩

lemma *sum-equivp* [*quot-equiv*]:
equivp R1 ⟹ equivp R2 ⟹ equivp (rel-sum R1 R2)
⟨*proof*⟩

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)*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *not-equiv-trans1* [*trans*]: $\neg\ x \sim y \Longrightarrow y \sim z \Longrightarrow \neg\ x \sim z$
⟨*proof*⟩

lemma *not-equiv-trans2* [*trans*]: $x \sim y \Longrightarrow \neg\ y \sim z \Longrightarrow \neg\ x \sim z$
⟨*proof*⟩

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**) $'a\ quot = quot :: 'a::eqv\ set\ set$
⟨*proof*⟩

lemma *quotI* [*intro*]: $\{x.\ a \sim x\} \in quot$
⟨*proof*⟩

lemma *quotE* [*elim*]:
assumes $R \in quot$
obtains a **where** $R = \{x.\ a \sim x\}$
⟨*proof*⟩

Abstracted equivalence classes are the canonical representation of elements of a quotient type.

definition $class :: 'a::equiv \Rightarrow 'a\ quot$ $(\lfloor\text{-}\rfloor)$
where $\lfloor a \rfloor = Abs\text{-}quot\ \{x.\ a \sim x\}$

theorem *quot-exhaust*: $\exists\, a.\ A = \lfloor a \rfloor$
⟨*proof*⟩

lemma *quot-cases* [*cases type*: *quot*]:
obtains a **where** $A = \lfloor a \rfloor$
⟨*proof*⟩

80.2 Equality on quotients

Equality of canonical quotient elements coincides with the original relation.

theorem *quot-equality* [*iff?*]: $\lfloor a \rfloor = \lfloor b \rfloor \longleftrightarrow a \sim b$
⟨*proof*⟩

80.3 Picking representing elements

definition $pick :: 'a::equiv\ quot \Rightarrow 'a$
where $pick\ A = (SOME\ a.\ A = \lfloor a \rfloor)$

theorem *pick-equiv* [*intro*]: $pick\ \lfloor a \rfloor \sim a$

⟨*proof*⟩

theorem *pick-inverse* [*intro*]: ⌊*pick A*⌋ = *A*
⟨*proof*⟩

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*⟩

theorem *quot-function*:
assumes ⋀*X Y*. *f X Y* ≡ *g* (*pick X*) (*pick Y*)
and ⋀*x x′ y y′*. ⌊*x*⌋ = ⌊*x′*⌋ ⟹ ⌊*y*⌋ = ⌊*y′*⌋ ⟹ *g x y* = *g x′ y′*
shows *f* ⌊*a*⌋ ⌊*b*⌋ = *g a b*
⟨*proof*⟩

theorem *quot-function′*:
(⋀*X Y*. *f X Y* ≡ *g* (*pick X*) (*pick Y*)) ⟹
(⋀*x x′ y y′*. *x* ∼ *x′* ⟹ *y* ∼ *y′* ⟹ *g x y* = *g x′ y′*) ⟹
f ⌊*a*⌋ ⌊*b*⌋ = *g a b*
⟨*proof*⟩

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* = (∀ *v*∈*V*. ∀ *w*∈*V*. *v*≠*w* ⟶ {*v*,*w*} : *E*)
definition *indep V E* = (∀ *v*∈*V*. ∀ *w*∈*V*. *v*≠*w* ⟶ ¬ {*v*,*w*} : *E*)

lemma *ramsey2*:
∃ *r*≥*1*. ∀ (*V*::′*a set*) (*E*::′*a set set*). *finite V* ∧ *card V* ≥ *r* ⟶
(∃ *R* ⊆ *V*. *card R* = *m* ∧ *clique R E* ∨ *card R* = *n* ∧ *indep R E*)

(**is** $\exists r{\geq}1.$ *?R m n r*)
⟨*proof*⟩

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*⟩

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*⟩

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*))
⟨*proof*⟩

lemma *part-subset*: *part r s YY f* ==> *Y* ⊆ *YY* ==> *part r s Y f*
⟨*proof*⟩

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*|]

$$\Longrightarrow \exists Y'\ t'.\ Y' \subseteq YY\ \&\ \mathit{infinite}\ Y'\ \&\ t' < s\ \&$$
$$(\forall X.\ X \subseteq Y'\ \&\ \mathit{finite}\ X\ \&\ \mathit{card}\ X = r \longrightarrow f\ X = t')$$

⟨*proof*⟩

theorem *Ramsey*:
fixes $s\ r :: nat$ **and** $Z :: 'a\ set$ **and** $f :: 'a\ set \Rightarrow nat$
shows
$$[|\mathit{infinite}\ Z;\ \forall X.\ X \subseteq Z\ \&\ \mathit{finite}\ X\ \&\ \mathit{card}\ X = r \longrightarrow f\ X < s|]$$
$$\Longrightarrow \exists Y\ t.\ Y \subseteq Z\ \&\ \mathit{infinite}\ Y\ \&\ t < s\ \&\ (\forall X.\ X \subseteq Y\ \&\ \mathit{finite}\ X\ \&\ \mathit{card}\ X = r \longrightarrow f\ X = t)$$

⟨*proof*⟩

corollary *Ramsey2*:
fixes $s :: nat$ **and** $Z :: 'a\ set$ **and** $f :: 'a\ set \Rightarrow nat$
assumes *infZ*: *infinite Z*
and *part*: $\forall x \in Z.\ \forall y \in Z.\ x \neq y \longrightarrow f\{x,y\} < s$
shows
$$\exists Y\ t.\ Y \subseteq Z\ \&\ \mathit{infinite}\ Y\ \&\ t < s\ \&\ (\forall x \in Y.\ \forall y \in Y.\ x \neq y \longrightarrow f\{x,y\} = t)$$

⟨*proof*⟩

81.4 Disjunctive Well-Foundedness

An application of Ramsey's theorem to program termination. See [2].

definition *disj-wf* :: $('a * 'a)set \Rightarrow bool$
where $\mathit{disj\text{-}wf}\ r = (\exists T.\ \exists n :: nat.\ (\forall i < n.\ wf(T\ i))\ \&\ r = (\bigcup i < n.\ T\ i))$

definition *transition-idx* :: $[nat \Rightarrow 'a, nat \Rightarrow ('a * 'a)set, nat\ set] \Rightarrow nat$
where
$\mathit{transition\text{-}idx}\ s\ T\ A =$
$(\mathit{LEAST}\ k.\ \exists i\ j.\ A = \{i,j\}\ \&\ i < j\ \&\ (s\ j,\ s\ i) \in T\ k)$

lemma *transition-idx-less*:
$[|i<j;\ (s\ j,\ s\ i) \in T\ k;\ k<n|] \Longrightarrow \mathit{transition\text{-}idx}\ s\ T\ \{i,j\} < n$

⟨*proof*⟩

lemma *transition-idx-in*:
$[|i<j;\ (s\ j,\ s\ i) \in T\ k|] \Longrightarrow (s\ j,\ s\ i) \in T\ (\mathit{transition\text{-}idx}\ s\ T\ \{i,j\})$

⟨*proof*⟩

To be equal to the union of some well-founded relations is equivalent to being the subset of such a union.

lemma *disj-wf*:
$\mathit{disj\text{-}wf}(r) = (\exists T.\ \exists n :: nat.\ (\forall i < n.\ wf(T\ i))\ \&\ r \subseteq (\bigcup i < n.\ T\ i))$

⟨*proof*⟩

theorem *trans-disj-wf-implies-wf*:
  assumes *transr*: *trans r*
    and *dwf*: *disj-wf*(*r*)
  shows *wf r*
⟨*proof*⟩

end

82 Generic reflection and reification

theory *Reflection*
imports *Main*
begin

⟨*ML*⟩

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 ⟨*proof*⟩

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 ⟨*proof*⟩

end

lemmas *len-of-numeral-defs* [*simp*] = *len-num0 len-num1 len-bit0 len-bit1*

instance *num1* :: *len* ⟨*proof*⟩
instance *bit0* :: (*len*) *len* ⟨*proof*⟩
instance *bit1* :: (*len0*) *len* ⟨*proof*⟩

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*
⟨*proof*⟩

lemma *sat-eqI*:
nat-of m = *nat-of n* ⟹ *m* = *n*
⟨*proof*⟩

lemma *sat-eq-iff*:
m = *n* ⟷ *nat-of m* = *nat-of n*
⟨*proof*⟩

lemma *Abs-sat-nat-of* [*code abstype*]:
Abs-sat (*nat-of n*) = *n*
⟨*proof*⟩

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*

⟨*proof*⟩

lemma *nat-of-le-len-of* [*simp*]:
nat-of (*n* :: ('*a*::*len*) *sat*) ≤ *len-of TYPE*('*a*)
⟨*proof*⟩

lemma *min-len-of-nat-of* [*simp*]:
min (*len-of TYPE*('*a*)) (*nat-of* (*n*::('*a*::*len*) *sat*)) = *nat-of n*
⟨*proof*⟩

lemma *min-nat-of-len-of* [*simp*]:
min (*nat-of* (*n*::('*a*::*len*) *sat*)) (*len-of TYPE*('*a*)) = *nat-of n*
⟨*proof*⟩

lemma *Abs-sat'-nat-of* [*simp*]:
Abs-sat' (*nat-of n*) = *n*
⟨*proof*⟩

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
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *nat-of-one-sat* [*simp*, *code abstract*]:
nat-of 1 = *min 1* (*len-of TYPE*('*a*))
⟨*proof*⟩

definition
$x + y = \mathit{Abs\text{-}sat}'\ (\mathit{nat\text{-}of}\ x + \mathit{nat\text{-}of}\ y)$

lemma *nat-of-plus-sat* [*simp*, *code abstract*]:
$\mathit{nat\text{-}of}\ (x + y) = \mathit{min}\ (\mathit{nat\text{-}of}\ x + \mathit{nat\text{-}of}\ y)\ (\mathit{len\text{-}of}\ \mathit{TYPE}('a))$
⟨*proof*⟩

definition
$x - y = \mathit{Abs\text{-}sat}'\ (\mathit{nat\text{-}of}\ x - \mathit{nat\text{-}of}\ y)$

lemma *nat-of-minus-sat* [*simp*, *code abstract*]:
$\mathit{nat\text{-}of}\ (x - y) = \mathit{nat\text{-}of}\ x - \mathit{nat\text{-}of}\ y$
⟨*proof*⟩

definition
$x * y = \mathit{Abs\text{-}sat}'\ (\mathit{nat\text{-}of}\ x * \mathit{nat\text{-}of}\ y)$

lemma *nat-of-times-sat* [*simp*, *code abstract*]:
$\mathit{nat\text{-}of}\ (x * y) = \mathit{min}\ (\mathit{nat\text{-}of}\ x * \mathit{nat\text{-}of}\ y)\ (\mathit{len\text{-}of}\ \mathit{TYPE}('a))$
⟨*proof*⟩

instance
⟨*proof*⟩

end

instantiation *sat* :: (*len*) *ordered-comm-semiring*
begin

instance
⟨*proof*⟩

end

lemma *Abs-sat'-eq-of-nat*: $\mathit{Abs\text{-}sat}'\ n = \mathit{of\text{-}nat}\ n$
⟨*proof*⟩

abbreviation *Sat* :: *nat* ⇒ 'a::*len sat* **where**
$\mathit{Sat} \equiv \mathit{of\text{-}nat}$

lemma *nat-of-Sat* [*simp*]:
$\mathit{nat\text{-}of}\ (\mathit{Sat}\ n :: ('a::\mathit{len})\ \mathit{sat}) = \mathit{min}\ (\mathit{len\text{-}of}\ \mathit{TYPE}('a))\ n$
⟨*proof*⟩

lemma [*code-abbrev*]:
$\mathit{of\text{-}nat}\ (\mathit{numeral}\ k) = (\mathit{numeral}\ k :: 'a::\mathit{len}\ \mathit{sat})$
⟨*proof*⟩

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*
  ⟨*proof*⟩

end

instance *sat* :: (*len*) *finite*
⟨*proof*⟩

instantiation *sat* :: (*len*) *equal*
begin

definition *HOL.equal A B* ⟷ *nat-of A* = *nat-of B*

instance
  ⟨*proof*⟩

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
  ⟨*proof*⟩

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 ⟨*proof*⟩

end

interpretation *Inf-sat*: *semilattice-neutr-set min top* :: *'a*::*len sat*
rewrites

semilattice-neutr-set.F min (*top* :: *'a sat*) = *Inf*
⟨*proof*⟩

interpretation *Sup-sat*: *semilattice-neutr-set max bot* :: *'a*::*len sat*
rewrites
semilattice-neutr-set.F max (*bot* :: *'a sat*) = *Sup*
⟨*proof*⟩

instance *sat* :: (*len*) *complete-lattice*
⟨*proof*⟩

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*⟩

end

87 A table-based implementation of the reflexive transitive closure

theory *Transitive-Closure-Table*
imports *Main*
begin

inductive *rtrancl-path* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a\ list \Rightarrow 'a \Rightarrow bool$
  for $r :: 'a \Rightarrow 'a \Rightarrow bool$
where
  base: *rtrancl-path r x* [] *x*
| *step*: $r\ x\ y \Longrightarrow$ *rtrancl-path r y ys z* $\Longrightarrow$ *rtrancl-path r x* $(y \# ys)\ z$

lemma *rtranclp-eq-rtrancl-path*: $r^{**}\ x\ y \longleftrightarrow (\exists\, xs.$ *rtrancl-path r x xs y*$)$
⟨*proof*⟩

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* $(xs\ @\ ys)\ z$ ⟨*proof*⟩

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*
  ⟨*proof*⟩

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** $set\ xs' \subseteq set\ xs$
  ⟨*proof*⟩

inductive *rtrancl-tab* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$
  for $r :: 'a \Rightarrow 'a \Rightarrow bool$
where
  base: *rtrancl-tab r xs x x*
| *step*: $x \notin set\ xs \Longrightarrow r\ x\ y \Longrightarrow$ *rtrancl-tab r* $(x \# xs)\ y\ z \Longrightarrow$ *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*: $(\{x\} \cup set\ xs) \cap set\ ys = \{\}$
  shows *rtrancl-tab r ys x y*
  ⟨*proof*⟩

lemma *rtrancl-tab-imp-rtrancl-path*:
  assumes *tab*: *rtrancl-tab r ys x y*
  obtains *xs* **where** *rtrancl-path r x xs y*
  ⟨*proof*⟩

lemma *rtranclp-eq-rtrancl-tab-nil*: $r^{**}\ x\ y \longleftrightarrow$ *rtrancl-tab r* [] *x y*
⟨*proof*⟩

declare *rtranclp-rtrancl-eq* [*code del*]
declare *rtranclp-eq-rtrancl-tab-nil* [*THEN iffD2*, *code-pred-intro*]

code-pred *rtranclp*
⟨*proof*⟩

lemma *rtrancl-path-Range*: ⟦ *rtrancl-path R x xs y*; $z \in$ *set xs* ⟧ ⟹ *RangeP R z*
⟨*proof*⟩

lemma *rtrancl-path-Range-end*: ⟦ *rtrancl-path R x xs y*; *xs* ≠ [] ⟧ ⟹ *RangeP R y*
⟨*proof*⟩

lemma *rtrancl-path-nth*:
⟦ *rtrancl-path R x xs y*; *i* < *length xs* ⟧ ⟹ *R* ((*x* # *xs*) ! *i*) (*xs* ! *i*)
⟨*proof*⟩

lemma *rtrancl-path-last*: ⟦ *rtrancl-path R x xs y*; *xs* ≠ [] ⟧ ⟹ *last xs* = *y*
⟨*proof*⟩

lemma *rtrancl-path-mono*:
⟦ *rtrancl-path R x p y*; ⋀*x y*. *R x y* ⟹ *S x y* ⟧ ⟹ *rtrancl-path S x p y*
⟨*proof*⟩

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 $\langle l, x, r\rangle$ = *size1* l + *size1* r
⟨*proof*⟩

lemma *size1-ge0*[*simp*]: $0 < size1\ t$
⟨*proof*⟩

lemma *size-0-iff-Leaf*: $size\ t = 0 \longleftrightarrow t = Leaf$
⟨*proof*⟩

lemma *neq-Leaf-iff*: $(t \neq \langle\rangle) = (\exists\, l\ a\ r.\ t = \langle l, a, r\rangle)$
⟨*proof*⟩

lemma *finite-set-tree*[*simp*]: *finite*(*set-tree t*)
⟨*proof*⟩

lemma *size-map-tree*[*simp*]: $size\ (map\text{-}tree\ f\ t) = size\ t$
⟨*proof*⟩

lemma *size1-map-tree*[*simp*]: $size1\ (map\text{-}tree\ f\ t) = size1\ t$
⟨*proof*⟩

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 ⟨*proof*⟩

end

lemma *height-map-tree*[*simp*]: $height\ (map\text{-}tree\ f\ t) = height\ t$
⟨*proof*⟩

lemma *size1-height*: $size\ t + 1 \leq 2\ \hat{}\ height\ (t::'a\ tree)$
⟨*proof*⟩

88.2 The set of subtrees

fun *subtrees* :: *'a tree* ⇒ *'a tree set* **where**
subtrees $\langle\rangle = \{\langle\rangle\}$ |
subtrees $(\langle l, a, r\rangle) = insert\ \langle l, a, r\rangle\ (subtrees\ l \cup subtrees\ r)$

lemma *set-treeE*: $a \in set\text{-}tree\ t \Longrightarrow \exists\, l\ r.\ \langle l, a, r\rangle \in subtrees\ t$
⟨*proof*⟩

lemma *Node-notin-subtrees-if*[*simp*]: $a \notin$ *set-tree* $t \Longrightarrow$ *Node* $l\ a\ r \notin$ *subtrees* t
⟨*proof*⟩

lemma *in-set-tree-if*: $\langle l, a, r\rangle \in$ *subtrees* $t \Longrightarrow a \in$ *set-tree* t
⟨*proof*⟩

88.3 List of entries

fun *preorder* :: *'a tree* ⇒ *'a list* **where**
preorder $\langle\rangle = [] \mid$
preorder $\langle l, x, r\rangle = x \# $ *preorder* l @ *preorder* r

fun *inorder* :: *'a tree* ⇒ *'a list* **where**
inorder $\langle\rangle = [] \mid$
inorder $\langle l, x, r\rangle =$ *inorder* l @ $[x]$ @ *inorder* r

lemma *set-inorder*[*simp*]: *set* (*inorder* t) = *set-tree* t
⟨*proof*⟩

lemma *set-preorder*[*simp*]: *set* (*preorder* t) = *set-tree* t
⟨*proof*⟩

lemma *length-preorder*[*simp*]: *length* (*preorder* t) = *size* t
⟨*proof*⟩

lemma *length-inorder*[*simp*]: *length* (*inorder* t) = *size* t
⟨*proof*⟩

lemma *preorder-map*: *preorder* (*map-tree* $f\ t$) = *map* f (*preorder* t)
⟨*proof*⟩

lemma *inorder-map*: *inorder* (*map-tree* $f\ t$) = *map* f (*inorder* t)
⟨*proof*⟩

88.4 Binary Search Tree predicate

fun (**in** *linorder*) *bst* :: *'a tree* ⇒ *bool* **where**
bst $\langle\rangle \longleftrightarrow$ *True* $\mid$
bst $\langle l, a, r\rangle \longleftrightarrow$ *bst* $l \wedge$ *bst* $r \wedge (\forall x{\in}$*set-tree* $l.\ x < a) \wedge (\forall x{\in}$*set-tree* $r.\ a < x)$

In case there are duplicates:

fun (**in** *linorder*) *bst-eq* :: *'a tree* ⇒ *bool* **where**
bst-eq $\langle\rangle \longleftrightarrow$ *True* $\mid$
bst-eq $\langle l,a,r\rangle \longleftrightarrow$
bst-eq $l \wedge$ *bst-eq* $r \wedge (\forall x{\in}$*set-tree* $l.\ x \le a) \wedge (\forall x{\in}$*set-tree* $r.\ a \le x)$

lemma (**in** *linorder*) *bst-eq-if-bst*: *bst* $t \Longrightarrow$ *bst-eq* t
⟨*proof*⟩

lemma (**in** *linorder*) *bst-eq-imp-sorted*: *bst-eq t* $\Longrightarrow$ *sorted* (*inorder t*)
⟨*proof*⟩

lemma (**in** *linorder*) *distinct-preorder-if-bst*: *bst t* $\Longrightarrow$ *distinct* (*preorder t*)
⟨*proof*⟩

lemma (**in** *linorder*) *distinct-inorder-if-bst*: *bst t* $\Longrightarrow$ *distinct* (*inorder t*)
⟨*proof*⟩

88.5 The heap predicate

fun *heap* :: *'a*::*linorder tree* ⇒ *bool* **where**
heap Leaf = *True* |
heap (*Node l m r*) =
(*heap l* ∧ *heap r* ∧ (∀ *x* ∈ *set-tree l* ∪ *set-tree r*. *m* ≤ *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* = ⟨⟩
⟨*proof*⟩

lemma *size-mirror*[*simp*]: *size*(*mirror t*) = *size t*
⟨*proof*⟩

lemma *size1-mirror*[*simp*]: *size1*(*mirror t*) = *size1 t*
⟨*proof*⟩

lemma *height-mirror*[*simp*]: *height*(*mirror t*) = *height t*
⟨*proof*⟩

lemma *inorder-mirror*: *inorder*(*mirror t*) = *rev*(*inorder t*)
⟨*proof*⟩

lemma *map-mirror*: *map-tree f* (*mirror t*) = *mirror* (*map-tree f t*)
⟨*proof*⟩

lemma *mirror-mirror*[*simp*]: *mirror*(*mirror t*) = *t*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *size-mset-tree*[*simp*]: *size*(*mset-tree t*) = *size t*
⟨*proof*⟩

lemma *mset-map-tree*: *mset-tree* (*map-tree f t*) = *image-mset f* (*mset-tree t*)
⟨*proof*⟩

lemma *mset-iff-set-tree*: *x* ∈# *mset-tree t* ⟷ *x* ∈ *set-tree t*
⟨*proof*⟩

lemma *mset-preorder*[*simp*]: *mset* (*preorder t*) = *mset-tree t*
⟨*proof*⟩

lemma *mset-inorder*[*simp*]: *mset* (*inorder t*) = *mset-tree t*
⟨*proof*⟩

lemma *map-mirror*: *mset-tree* (*mirror t*) = *mset-tree t*
⟨*proof*⟩

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*⟩

lemma *while-option-stop2*:

while-option b c s = Some t ⟹ EX k. t = (c^^k) s ∧ ¬ b t
⟨*proof*⟩

lemma *while-option-stop*: *while-option b c s = Some t ⟹ ~ b t*
⟨*proof*⟩

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*⟩

lemma *funpow-commute*:
⟦∀ k' < k. f (c ((c^^k') s)) = c' (f ((c^^k') s))⟧ ⟹ f ((c^^k) s) = (c'^^k) (f s)
⟨*proof*⟩

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)*
⟨*proof*⟩

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)*
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *def-while-unfold*:
assumes *fdef*: *f == while test do*
shows *f x = (if test x then f(do x) else x)*
⟨*proof*⟩

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)*

⟨*proof*⟩

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*)
⟨*proof*⟩

Proving termination:

theorem *wf-while-option-Some*:
assumes *wf* {(*t*, *s*). (*P s* ∧ *b s*) ∧ *t* = *c s*}
and !!*s*. *P s* ⟹ *b s* ⟹ *P*(*c s*) **and** *P s*
shows *EX t*. *while-option b c s* = *Some t*
⟨*proof*⟩

lemma *wf-rel-while-option-Some*:
assumes *wf*: *wf R*
assumes *smaller*: ⋀*s*. *P s* ∧ *b s* ⟹ (*c s*, *s*) ∈ *R*
assumes *inv*: ⋀*s*. *P s* ∧ *b s* ⟹ *P*(*c s*)
assumes *init*: *P s*
shows ∃ *t*. *while-option b c s* = *Some t*
⟨*proof*⟩

theorem *measure-while-option-Some*: **fixes** *f* :: *'s* ⇒ *nat*
shows (!!*s*. *P s* ⟹ *b s* ⟹ *P*(*c s*) ∧ *f*(*c s*) < *f s*)
⟹ *P s* ⟹ *EX t*. *while-option b c s* = *Some t*
⟨*proof*⟩

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* ⊆ *C* ⟹ *f X* ⊆ *C* **and** *finite C*
shows ∃ *P*. *while-option* (λ*A*. *f A* ≠ *A*) *f* {} = *Some P*
⟨*proof*⟩

lemma *lfp-the-while-option*:
assumes *mono f* **and** !!*X*. *X* ⊆ *C* ⟹ *f X* ⊆ *C* **and** *finite C*
shows *lfp f* = *the*(*while-option* (λ*A*. *f A* ≠ *A*) *f* {})
⟨*proof*⟩

lemma *lfp-while*:
assumes *mono f* **and** !!*X*. *X* ⊆ *C* ⟹ *f X* ⊆ *C* **and** *finite C*
shows *lfp f* = *while* (λ*A*. *f A* ≠ *A*) *f* {}
⟨*proof*⟩

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 $p :: 'a \Rightarrow bool$
and $f :: 'a \Rightarrow 'a\ list$
and $x :: 'a$
begin

qualified fun *rtrancl-while-test* $:: 'a\ list \times 'a\ set \Rightarrow bool$
where *rtrancl-while-test* $(ws,\text{-}) = (ws \neq [] \wedge p(hd\ ws))$

qualified fun *rtrancl-while-step* $:: 'a\ list \times 'a\ set \Rightarrow 'a\ list \times 'a\ set$
where *rtrancl-while-step* $(ws,\ Z) =$
$(let\ x = hd\ ws;\ new = remdups\ (filter\ (\lambda y.\ y \notin Z)\ (f\ x))$
$in\ (new\ @\ tl\ ws,\ set\ new \cup Z))$

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* $:: 'a\ list \times 'a\ set \Rightarrow bool$
where *rtrancl-while-invariant* $(ws,\ Z) =$
$(x \in Z \wedge set\ ws \subseteq Z \wedge distinct\ ws \wedge \{(x,y).\ y \in set(f\ x)\}\ ``\ (Z - set\ ws) \subseteq Z \wedge$
$Z \subseteq \{(x,y).\ y \in set(f\ x)\}\hat{}*\ ``\ \{x\} \wedge (\forall z{\in}Z - set\ ws.\ p\ z))$

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*⟩

lemma *rtrancl-while-Some*: **assumes** *rtrancl-while* $= Some(ws,Z)$
shows *if* $ws = []$
then $Z = \{(x,y).\ y \in set(f\ x)\}\hat{}*\ ``\ \{x\} \wedge (\forall z{\in}Z.\ p\ z)$
else $\neg p(hd\ ws) \wedge hd\ ws \in \{(x,y).\ y \in set(f\ x)\}\hat{}*\ ``\ \{x\}$
⟨*proof*⟩

lemma *rtrancl-while-finite-Some*:
assumes *finite* $(\{(x,\ y).\ y \in set\ (f\ x)\}^*\ ``\ \{x\})$ (**is** *finite ?Cl*)
shows $\exists y.$ *rtrancl-while* $=$ *Some y*
⟨*proof*⟩

end

end

theory *Rewrite*
imports *Main*

begin

consts *rewrite-HOLE* :: *'a*::{} (⌑)

lemma *eta-expand*:
fixes *f* :: *'a*::{} ⇒ *'b*::{}
shows *f* ≡ λ*x*. *f x* ⟨*proof*⟩

lemma *rewr-imp*:
assumes *PROP A* ≡ *PROP B*
shows (*PROP A* ⟹ *PROP C*) ≡ (*PROP B* ⟹ *PROP C*)
⟨*proof*⟩

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'*))
⟨*proof*⟩

⟨*ML*⟩

end

91 Lexicographic order on lists

theory *List-lexord*
imports *Main*
begin

instantiation *list* :: (*ord*) *ord*
begin

definition
list-less-def: *xs* < *ys* ⟷ (*xs*, *ys*) ∈ *lexord* {(*u*, *v*). *u* < *v*}

definition
list-le-def: (*xs* :: - *list*) ≤ *ys* ⟷ *xs* < *ys* ∨ *xs* = *ys*

instance ⟨*proof*⟩

end

instance *list* :: (*order*) *order*
⟨*proof*⟩

instance *list* :: (*linorder*) *linorder*
⟨*proof*⟩

instantiation *list* :: (*linorder*) *distrib-lattice*

begin

definition (*inf* :: *'a list* ⇒ -) = *min*

definition (*sup* :: *'a list* ⇒ -) = *max*

instance
⟨*proof*⟩

end

lemma *not-less-Nil* [*simp*]: ¬ *x* < []
⟨*proof*⟩

lemma *Nil-less-Cons* [*simp*]: [] < *a* # *x*
⟨*proof*⟩

lemma *Cons-less-Cons* [*simp*]: *a* # *x* < *b* # *y* ⟷ *a* < *b* ∨ *a* = *b* ∧ *x* < *y*
⟨*proof*⟩

lemma *le-Nil* [*simp*]: *x* ≤ [] ⟷ *x* = []
⟨*proof*⟩

lemma *Nil-le-Cons* [*simp*]: [] ≤ *x*
⟨*proof*⟩

lemma *Cons-le-Cons* [*simp*]: *a* # *x* ≤ *b* # *y* ⟷ *a* < *b* ∨ *a* = *b* ∧ *x* ≤ *y*
⟨*proof*⟩

instantiation *list* :: (*order*) *order-bot*
begin

definition *bot* = []

instance
⟨*proof*⟩

end

lemma *less-list-code* [*code*]:
xs < ([]::*'a*::{*equal*, *order*} *list*) ⟷ *False*
[] < (*x*::*'a*::{*equal*, *order*}) # *xs* ⟷ *True*
(*x*::*'a*::{*equal*, *order*}) # *xs* < *y* # *ys* ⟷ *x* < *y* ∨ *x* = *y* ∧ *xs* < *ys*
⟨*proof*⟩

lemma *less-eq-list-code* [*code*]:
x # *xs* ≤ ([]::*'a*::{*equal*, *order*} *list*) ⟷ *False*
[] ≤ (*xs*::*'a*::{*equal*, *order*} *list*) ⟷ *True*
(*x*::*'a*::{*equal*, *order*}) # *xs* ≤ *y* # *ys* ⟷ *x* < *y* ∨ *x* = *y* ∧ *xs* ≤ *ys*

⟨*proof*⟩

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*) ≤ *ys* ⟷ *sublisteq xs ys*

definition
(*xs* :: *'a list*) < *ys* ⟷ *xs* ≤ *ys* ∧ ¬ *ys* ≤ *xs*

instance ⟨*proof*⟩

end

instance *list* :: (*type*) *order*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *less-list-empty* [*simp*]: [] < *xs* ⟷ *xs* ≠ []
⟨*proof*⟩

lemma *less-list-below-empty* [*simp*]: *xs* < [] ⟷ *False*
⟨*proof*⟩

lemma *less-list-drop*: *xs* < *ys* ⟹ *xs* < *x* # *ys*

⟨*proof*⟩

lemma *less-list-take-iff*: $x \# xs < x \# ys \longleftrightarrow xs < ys$
⟨*proof*⟩

lemma *less-list-drop-many*: $xs < ys \Longrightarrow xs < zs @ ys$
⟨*proof*⟩

lemma *less-list-take-many-iff*: $zs @ xs < zs @ ys \longleftrightarrow xs < ys$
⟨*proof*⟩

lemma *less-list-rev-take*: $xs @ zs < ys @ zs \longleftrightarrow xs < ys$
⟨*proof*⟩

end

93 Lexicographic order on product types

theory *Product-Lexorder*
imports *Main*
begin

instantiation *prod* :: (*ord*, *ord*) *ord*
begin

definition
$x \leq y \longleftrightarrow fst\ x < fst\ y \vee fst\ x \leq fst\ y \wedge snd\ x \leq snd\ y$

definition
$x < y \longleftrightarrow fst\ x < fst\ y \vee fst\ x \leq fst\ y \wedge snd\ x < snd\ y$

instance ⟨*proof*⟩

end

lemma *less-eq-prod-simp* [*simp*, *code*]:
$(x1, y1) \leq (x2, y2) \longleftrightarrow x1 < x2 \vee x1 \leq x2 \wedge y1 \leq y2$
⟨*proof*⟩

lemma *less-prod-simp* [*simp*, *code*]:
$(x1, y1) < (x2, y2) \longleftrightarrow x1 < x2 \vee x1 \leq x2 \wedge y1 < y2$
⟨*proof*⟩

A stronger version for partial orders.

lemma *less-prod-def'*:
fixes $x\ y$ :: $'a$::*order* × $'b$::*ord*
shows $x < y \longleftrightarrow fst\ x < fst\ y \vee fst\ x = fst\ y \wedge snd\ x < snd\ y$
⟨*proof*⟩

instance *prod* :: (*preorder*, *preorder*) *preorder*
⟨*proof*⟩

instance *prod* :: (*order*, *order*) *order*
⟨*proof*⟩

instance *prod* :: (*linorder*, *linorder*) *linorder*
⟨*proof*⟩

instantiation *prod* :: (*linorder*, *linorder*) *distrib-lattice*
begin

definition
(*inf* :: $'a \times 'b \Rightarrow$ - $\Rightarrow$ -) = *min*

definition
(*sup* :: $'a \times 'b \Rightarrow$ - $\Rightarrow$ -) = *max*

instance
⟨*proof*⟩

end

instantiation *prod* :: (*bot*, *bot*) *bot*
begin

definition
bot = (*bot*, *bot*)

instance ⟨*proof*⟩

end

instance *prod* :: (*order-bot*, *order-bot*) *order-bot*
⟨*proof*⟩

instantiation *prod* :: (*top*, *top*) *top*
begin

definition
top = (*top*, *top*)

instance ⟨*proof*⟩

end

instance *prod* :: (*order-top*, *order-top*) *order-top*
⟨*proof*⟩

instance *prod* :: (*wellorder*, *wellorder*) *wellorder*
⟨*proof*⟩

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
$x \leq y \longleftrightarrow fst\ x \leq fst\ y \wedge snd\ x \leq snd\ y$

definition
$(x::'a \times 'b) < y \longleftrightarrow x \leq y \wedge \neg\ y \leq x$

instance ⟨*proof*⟩

end

lemma *fst-mono*: $x \leq y \Longrightarrow fst\ x \leq fst\ y$
⟨*proof*⟩

lemma *snd-mono*: $x \leq y \Longrightarrow snd\ x \leq snd\ y$
⟨*proof*⟩

lemma *Pair-mono*: $x \leq x' \Longrightarrow y \leq y' \Longrightarrow (x, y) \leq (x', y')$
⟨*proof*⟩

lemma *Pair-le* [*simp*]: $(a, b) \leq (c, d) \longleftrightarrow a \leq c \wedge b \leq d$
⟨*proof*⟩

instance *prod* :: (*preorder*, *preorder*) *preorder*
⟨*proof*⟩

instance *prod* :: (*order*, *order*) *order*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *fst-inf* [*simp*]: *fst* (*inf x y*) = *inf* (*fst x*) (*fst y*)
⟨*proof*⟩

lemma *snd-inf* [*simp*]: *snd* (*inf x y*) = *inf* (*snd x*) (*snd y*)
⟨*proof*⟩

instance ⟨*proof*⟩

end

instance *prod* :: (*semilattice-inf*, *semilattice-inf*) *semilattice-inf*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *fst-sup* [*simp*]: *fst* (*sup x y*) = *sup* (*fst x*) (*fst y*)
⟨*proof*⟩

lemma *snd-sup* [*simp*]: *snd* (*sup x y*) = *sup* (*snd x*) (*snd y*)
⟨*proof*⟩

instance ⟨*proof*⟩

end

instance *prod* :: (*semilattice-sup*, *semilattice-sup*) *semilattice-sup*
⟨*proof*⟩

instance *prod* :: (*lattice*, *lattice*) *lattice* ⟨*proof*⟩

instance *prod* :: (*distrib-lattice*, *distrib-lattice*) *distrib-lattice*
⟨*proof*⟩

94.3 Top and bottom elements

instantiation *prod* :: (*top*, *top*) *top*
begin

definition
top = (*top*, *top*)

instance ⟨*proof*⟩

end

lemma *fst-top* [*simp*]: *fst top* = *top*
⟨*proof*⟩

lemma *snd-top* [*simp*]: *snd top* = *top*
⟨*proof*⟩

lemma *Pair-top-top*: (*top*, *top*) = *top*
⟨*proof*⟩

instance *prod* :: (*order-top*, *order-top*) *order-top*
⟨*proof*⟩

instantiation *prod* :: (*bot*, *bot*) *bot*
begin

definition
bot = (*bot*, *bot*)

instance ⟨*proof*⟩

end

lemma *fst-bot* [*simp*]: *fst bot* = *bot*
⟨*proof*⟩

lemma *snd-bot* [*simp*]: *snd bot* = *bot*
⟨*proof*⟩

lemma *Pair-bot-bot*: (*bot*, *bot*) = *bot*
⟨*proof*⟩

instance *prod* :: (*order-bot*, *order-bot*) *order-bot*
⟨*proof*⟩

instance *prod* :: (*bounded-lattice*, *bounded-lattice*) *bounded-lattice* ⟨*proof*⟩

instance *prod* :: (*boolean-algebra*, *boolean-algebra*) *boolean-algebra*
⟨*proof*⟩

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 ⟨*proof*⟩

end

instantiation *prod* :: (*Sup*, *Sup*) *Sup*
begin

definition *Sup A* = (*SUP x*:*A*. *fst x*, *SUP x*:*A*. *snd x*)

instance ⟨*proof*⟩

end

instance *prod* :: (*conditionally-complete-lattice*, *conditionally-complete-lattice*)
conditionally-complete-lattice
⟨*proof*⟩

instance *prod* :: (*complete-lattice*, *complete-lattice*) *complete-lattice*
⟨*proof*⟩

lemma *fst-Sup*: *fst* (*Sup A*) = (*SUP x*:*A*. *fst x*)
⟨*proof*⟩

lemma *snd-Sup*: *snd* (*Sup A*) = (*SUP x*:*A*. *snd x*)
⟨*proof*⟩

lemma *fst-Inf*: *fst* (*Inf A*) = (*INF x*:*A*. *fst x*)
⟨*proof*⟩

lemma *snd-Inf*: *snd* (*Inf A*) = (*INF x*:*A*. *snd x*)
⟨*proof*⟩

lemma *fst-SUP*: *fst* (*SUP x*:*A*. *f x*) = (*SUP x*:*A*. *fst* (*f x*))
⟨*proof*⟩

lemma *snd-SUP*: *snd* (*SUP x*:*A*. *f x*) = (*SUP x*:*A*. *snd* (*f x*))
⟨*proof*⟩

lemma *fst-INF*: *fst* (*INF x*:*A*. *f x*) = (*INF x*:*A*. *fst* (*f x*))
⟨*proof*⟩

lemma *snd-INF*: *snd* (*INF x*:*A*. *f x*) = (*INF x*:*A*. *snd* (*f x*))
⟨*proof*⟩

lemma *SUP-Pair*: $(SUP\ x{:}A.\ (f\ x,\ g\ x)) = (SUP\ x{:}A.\ f\ x,\ SUP\ x{:}A.\ g\ x)$
⟨*proof*⟩

lemma *INF-Pair*: $(INF\ x{:}A.\ (f\ x,\ g\ x)) = (INF\ x{:}A.\ f\ x,\ INF\ x{:}A.\ g\ x)$
⟨*proof*⟩

Alternative formulations for set infima and suprema over the product of two complete lattices:

lemma *INF-prod-alt-def*:
$INFIMUM\ A\ f = (INFIMUM\ A\ (fst\ o\ f),\ INFIMUM\ A\ (snd\ o\ f))$
⟨*proof*⟩

lemma *SUP-prod-alt-def*:
$SUPREMUM\ A\ f = (SUPREMUM\ A\ (fst\ o\ f),\ SUPREMUM\ A\ (snd\ o\ f))$
⟨*proof*⟩

94.5 Complete distributive lattices

instance *prod* :: (*complete-distrib-lattice*, *complete-distrib-lattice*) *complete-distrib-lattice*
⟨*proof*⟩

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*: $bot = Inf\text{-}fin\ UNIV$
assumes *top-def*: $top = Sup\text{-}fin\ UNIV$
assumes *Inf-def*: $Inf\ A = Finite\text{-}Set.fold\ inf\ top\ A$
assumes *Sup-def*: $Sup\ A = Finite\text{-}Set.fold\ sup\ bot\ A$

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*: $(bot{::}'a{::}finite\text{-}lattice\text{-}complete) \leq x$
⟨*proof*⟩

instance *finite-lattice-complete* ⊆ *order-bot*
⟨*proof*⟩

lemma *finite-lattice-complete-top-greatest*: $(top::'a::finite\text{-}lattice\text{-}complete) \geq x$
⟨*proof*⟩

instance *finite-lattice-complete* ⊆ *order-top*
⟨*proof*⟩

instance *finite-lattice-complete* ⊆ *bounded-lattice* ⟨*proof*⟩

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*: $Inf\ \{\} = (top :: 'a::finite\text{-}lattice\text{-}complete)$
⟨*proof*⟩

lemma *finite-lattice-complete-Sup-empty*: $Sup\ \{\} = (bot :: 'a::finite\text{-}lattice\text{-}complete)$
⟨*proof*⟩

lemma *finite-lattice-complete-Inf-insert*:
fixes $A :: 'a::finite\text{-}lattice\text{-}complete\ set$
shows $Inf\ (insert\ x\ A) = inf\ x\ (Inf\ A)$
⟨*proof*⟩

lemma *finite-lattice-complete-Sup-insert*:
fixes $A :: 'a::finite\text{-}lattice\text{-}complete\ set$
shows $Sup\ (insert\ x\ A) = sup\ x\ (Sup\ A)$
⟨*proof*⟩

lemma *finite-lattice-complete-Inf-lower*:
$(x::'a::finite\text{-}lattice\text{-}complete) \in A \Longrightarrow Inf\ A \leq x$
⟨*proof*⟩

lemma *finite-lattice-complete-Inf-greatest*:
$\forall\, x::'a::finite\text{-}lattice\text{-}complete \in A.\ z \leq x \Longrightarrow z \leq Inf\ A$
⟨*proof*⟩

lemma *finite-lattice-complete-Sup-upper*:
$(x::'a::finite\text{-}lattice\text{-}complete) \in A \Longrightarrow Sup\ A \geq x$
⟨*proof*⟩

lemma *finite-lattice-complete-Sup-least*:
$\forall\, x::'a::finite\text{-}lattice\text{-}complete \in A.\ z \geq x \Longrightarrow z \geq Sup\ A$
⟨*proof*⟩

instance *finite-lattice-complete* ⊆ *complete-lattice*
⟨*proof*⟩

The product of two finite lattices is already a finite lattice.

lemma *finite-bot-prod*:
$(bot :: ('a::finite\text{-}lattice\text{-}complete \times 'b::finite\text{-}lattice\text{-}complete)) =$

Inf-fin UNIV
⟨*proof*⟩

lemma *finite-top-prod*:
(*top* :: ('*a*::*finite-lattice-complete* × '*b*::*finite-lattice-complete*)) =
Sup-fin UNIV
⟨*proof*⟩

lemma *finite-Inf-prod*:
Inf(*A* :: ('*a*::*finite-lattice-complete* × '*b*::*finite-lattice-complete*) *set*) =
Finite-Set.fold inf top A
⟨*proof*⟩

lemma *finite-Sup-prod*:
Sup (*A* :: ('*a*::*finite-lattice-complete* × '*b*::*finite-lattice-complete*) *set*) =
Finite-Set.fold sup bot A
⟨*proof*⟩

instance *prod* :: (*finite-lattice-complete*, *finite-lattice-complete*) *finite-lattice-complete*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *finite-top-fun*: (*top* :: ('*a*::*finite* ⇒ '*b*::*finite-lattice-complete*)) = *Sup-fin UNIV*
⟨*proof*⟩

lemma *finite-Inf-fun*:
Inf (*A*::('*a*::*finite* ⇒ '*b*::*finite-lattice-complete*) *set*) =
Finite-Set.fold inf top A
⟨*proof*⟩

lemma *finite-Sup-fun*:
Sup (*A*::('*a*::*finite* ⇒ '*b*::*finite-lattice-complete*) *set*) =
Finite-Set.fold sup bot A
⟨*proof*⟩

instance *fun* :: (*finite*, *finite-lattice-complete*) *finite-lattice-complete*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *finite-distrib-lattice-complete-inf-Sup*:
inf (*x*::'*a*::*finite-distrib-lattice-complete*) (*Sup A*) = (*SUP y*:*A*. *inf x y*)
⟨*proof*⟩

instance *finite-distrib-lattice-complete* ⊆ *complete-distrib-lattice*
⟨*proof*⟩

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
⟨*proof*⟩

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*
⟨*proof*⟩

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-lattice*) *y* ≤ *x*
⟨*proof*⟩

lemma *linorder-lattice-inf-le2*: *inf* (*x*::'*a*::*linorder-lattice*) *y* ≤ *y*
⟨*proof*⟩

lemma *linorder-lattice-inf-greatest*:
(*x*::'*a*::*linorder-lattice*) ≤ *y* ⟹ *x* ≤ *z* ⟹ *x* ≤ *inf y z*
⟨*proof*⟩

lemma *linorder-lattice-sup-ge1*: *sup* (*x*::*'a*::*linorder-lattice*) *y* ≥ *x*
⟨*proof*⟩

lemma *linorder-lattice-sup-ge2*: *sup* (*x*::*'a*::*linorder-lattice*) *y* ≥ *y*
⟨*proof*⟩

lemma *linorder-lattice-sup-least*:
(*x*::*'a*::*linorder-lattice*) ≥ *y* ⟹ *x* ≥ *z* ⟹ *x* ≥ *sup y z*
⟨*proof*⟩

lemma *linorder-lattice-sup-inf-distrib1*:
sup (*x*::*'a*::*linorder-lattice*) (*inf y z*) = *inf* (*sup x y*) (*sup x z*)
⟨*proof*⟩

instance *linorder-lattice* ⊆ *distrib-lattice*
⟨*proof*⟩

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* ⟨*proof*⟩

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* ⟨*proof*⟩

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*)
⟨*proof*⟩

termination *gcd* :: - *poly* ⇒ -
⟨*proof*⟩

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 ⟨*proof*⟩

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*
⟨*proof*⟩

lemma *poly-gcd-greatest*:
fixes *k x y* :: - *poly*
shows *k dvd x* ⟹ *k dvd y* ⟹ *k dvd gcd x y*
⟨*proof*⟩

lemma *dvd-poly-gcd-iff* [*iff*]:
fixes *k x y* :: - *poly*
shows *k dvd gcd x y* ⟷ *k dvd x* ∧ *k dvd y*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *poly-gcd-zero-iff* [*simp*]:
fixes *x y* :: - *poly*
shows *gcd x y* = *0* ⟷ *x* = *0* ∧ *y* = *0*
⟨*proof*⟩

lemma *poly-gcd-0-0* [*simp*]:
gcd (*0*::- *poly*) *0* = *0*
⟨*proof*⟩

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*⟩

lemma *poly-gcd-unique*:
  fixes *d x y* :: - *poly*
  assumes *dvd1*: *d dvd x* **and** *dvd2*: *d dvd y*
    and *greatest*: $\bigwedge k.\ k\ dvd\ x \Longrightarrow k\ dvd\ y \Longrightarrow k\ dvd\ d$
    and *monic*: *coeff d* (*degree d*) = (*if x* = *0* ∧ *y* = *0 then 0 else 1*)
  shows *gcd x y* = *d*
⟨*proof*⟩

instance *poly* :: (*field*) *semiring-gcd*
⟨*proof*⟩

lemma *poly-gcd-1-left* [*simp*]: *gcd 1 y* = (*1* :: - *poly*)
⟨*proof*⟩

lemma *poly-gcd-1-right* [*simp*]: *gcd x 1* = (*1* :: - *poly*)
⟨*proof*⟩

lemma *poly-gcd-minus-left* [*simp*]: *gcd* (− *x*) *y* = *gcd x* (*y* :: - *poly*)
⟨*proof*⟩

lemma *poly-gcd-minus-right* [*simp*]: *gcd x* (− *y*) = *gcd x* (*y* :: - *poly*)
⟨*proof*⟩

lemma *poly-gcd-code* [*code*]:
  gcd x y = (*if y* = *0 then normalize x else gcd y* (*x mod* (*y* :: - *poly*)))
  ⟨*proof*⟩

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* ⟨*proof*⟩

code-datatype *Mapping*

lemma *lookup-Mapping* [*simp*, *code*]:
  Mapping.lookup (*Mapping xs*) = *map-of xs*
  ⟨*proof*⟩

lemma *keys-Mapping* [*simp*, *code*]:
  Mapping.keys (*Mapping xs*) = *set* (*map fst xs*)
  ⟨*proof*⟩

lemma *empty-Mapping* [*code*]:
Mapping.empty = *Mapping* []
⟨*proof*⟩

lemma *is-empty-Mapping* [*code*]:
Mapping.is-empty (*Mapping xs*) ⟷ *List.null xs*
⟨*proof*⟩

lemma *update-Mapping* [*code*]:
Mapping.update k v (*Mapping xs*) = *Mapping* (*AList.update k v xs*)
⟨*proof*⟩

lemma *delete-Mapping* [*code*]:
Mapping.delete k (*Mapping xs*) = *Mapping* (*AList.delete k xs*)
⟨*proof*⟩

lemma *ordered-keys-Mapping* [*code*]:
Mapping.ordered-keys (*Mapping xs*) = *sort* (*remdups* (*map fst xs*))
⟨*proof*⟩

lemma *size-Mapping* [*code*]:
Mapping.size (*Mapping xs*) = *length* (*remdups* (*map fst xs*))
⟨*proof*⟩

lemma *tabulate-Mapping* [*code*]:
Mapping.tabulate ks f = *Mapping* (*map* (λ*k*. (*k*, *f k*)) *ks*)
⟨*proof*⟩

lemma *bulkload-Mapping* [*code*]:
Mapping.bulkload vs = *Mapping* (*map* (λ*n*. (*n*, *vs* ! *n*)) [*0*..<*length vs*])
⟨*proof*⟩

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*⟩

lemma [*code nbe*]:
HOL.equal (*x* :: ('*a*, '*b*) *mapping*) *x* ⟷ *True*
⟨*proof*⟩

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-nat = (λ*f g n*. *if n* = *0 then f else g* (*n* − *1*))
⟨*proof*⟩

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 ⋀*n*. *f* (*Suc n*) ≡ *h n*
assumes *f 0* ≡ *g*
shows *f n* ≡ *if n* = *0 then g else h* (*n* − *1*)
⟨*proof*⟩

The rule above is built into a preprocessor that is plugged into the code generator.

⟨*ML*⟩

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
⟨*proof*⟩

lemma [*code*]:
(1::*nat) = Numeral1*
⟨*proof*⟩

lemma [*code-abbrev*]: *Numeral1 = (1*::*nat)*
⟨*proof*⟩

lemma [*code*]:
Suc n = n + 1
⟨*proof*⟩

98.2 Basic arithmetic

context
begin

lemma [*code*, *code del*]:
(plus :: *nat ⇒* -*) = plus* ⟨*proof*⟩

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)*
⟨*proof*⟩

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)
⟨*proof*⟩ **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 (λq. 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))
⟨*proof*⟩

lemma [*code*, *code del*]:
(*minus* :: *nat* ⇒ -) = *minus* ⟨*proof*⟩

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)*
⟨*proof*⟩

lemma [*code*, *code del*]:
(*times* :: *nat* ⇒ -) = *times* ⟨*proof*⟩

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)*
⟨*proof*⟩

lemma [*code*, *code del*]:
(*HOL.equal* :: *nat* ⇒ -) = *HOL.equal* ⟨*proof*⟩

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*
⟨*proof*⟩

lemma *equal-nat-refl* [*code nbe*]:
HOL.equal (n::*nat) n* ⟷ *True*
⟨*proof*⟩

lemma [*code*, *code del*]:
(*less-eq* :: *nat* ⇒ -) = *less-eq* ⟨*proof*⟩

lemma *less-eq-nat-code* [*code*]:
0 ≤ (n::*nat)* ⟷ *True*

nat-of-num k $\leq$ *0* $\longleftrightarrow$ *False*
nat-of-num k $\leq$ *nat-of-num l* $\longleftrightarrow$ *k* $\leq$ *l*
⟨*proof*⟩

lemma [*code*, *code del*]:
(*less* :: *nat* $\Rightarrow$ -) = *less* ⟨*proof*⟩

lemma *less-nat-code* [*code*]:
(*m*::*nat*) < *0* $\longleftrightarrow$ *False*
0 < *nat-of-num l* $\longleftrightarrow$ *True*
nat-of-num k < *nat-of-num l* $\longleftrightarrow$ *k* < *l*
⟨*proof*⟩

lemma [*code*, *code del*]:
Divides.divmod-nat = *Divides.divmod-nat* ⟨*proof*⟩

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*)
⟨*proof*⟩

end

98.3 Conversions

lemma [*code*, *code del*]:
of-nat = *of-nat* ⟨*proof*⟩

lemma *of-nat-code* [*code*]:
of-nat 0 = *0*
of-nat (*nat-of-num k*) = *numeral k*
⟨*proof*⟩

code-identifier
code-module *Code-Binary-Nat* $\rightharpoonup$
(*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* $\rightharpoonup$

```
    (SML) char
    and (OCaml) char
    and (Haskell) Prelude.Char
    and (Scala) Char

⟨ML⟩

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*⟩

101 Setup for Numerals

⟨*ML*⟩

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*
⟨*proof*⟩

lemma [*code*]:
Int.Pos = *int-of-integer* ∘ *integer-of-num*
⟨*proof*⟩

lemma [*code*]:
Int.Neg = *int-of-integer* ∘ *uminus* ∘ *integer-of-num*
⟨*proof*⟩

lemma [*code-abbrev*]:
int-of-integer (*numeral k*) = *Int.Pos k*
⟨*proof*⟩

lemma [*code-abbrev*]:
int-of-integer (− *numeral k*) = *Int.Neg k*
⟨*proof*⟩

lemma [*code*, *symmetric*, *code-post*]:
0 = *int-of-integer 0*
⟨*proof*⟩

lemma [*code*, *symmetric*, *code-post*]:
1 = *int-of-integer 1*
⟨*proof*⟩

lemma [*code-post*]:
int-of-integer (− *1*) = − *1*
⟨*proof*⟩

lemma [*code*]:
k + *l* = *int-of-integer* (*of-int k* + *of-int l*)
⟨*proof*⟩

lemma [*code*]:
− *k* = *int-of-integer* (− *of-int k*)
⟨*proof*⟩

lemma [*code*]:
k − *l* = *int-of-integer* (*of-int k* − *of-int l*)
⟨*proof*⟩

lemma [*code*]:
Int.dup k = *int-of-integer* (*Code-Numeral.dup* (*of-int k*))
⟨*proof*⟩

declare [[*code drop*: *Int.sub*]]

lemma [*code*]:
k ∗ *l* = *int-of-integer* (*of-int k* ∗ *of-int l*)
⟨*proof*⟩

lemma [*code*]:
k div l = *int-of-integer* (*of-int k div of-int l*)
⟨*proof*⟩

lemma [*code*]:
k mod l = *int-of-integer* (*of-int k mod of-int l*)
⟨*proof*⟩

lemma [*code*]:
divmod m n = *map-prod int-of-integer int-of-integer* (*divmod m n*)
⟨*proof*⟩

lemma [*code*]:
HOL.equal k l = *HOL.equal* (*of-int k* :: *integer*) (*of-int l*)
⟨*proof*⟩

lemma [*code*]:
k ≤ *l* ⟷ (*of-int k* :: *integer*) ≤ *of-int l*
⟨*proof*⟩

lemma [*code*]:
k < *l* ⟷ (*of-int k* :: *integer*) < *of-int l*

⟨*proof*⟩

lemma *gcd-int-of-integer* [*code*]:
gcd (*int-of-integer x*) (*int-of-integer y*) = *int-of-integer* (*gcd x y*)
⟨*proof*⟩

lemma *lcm-int-of-integer* [*code*]:
lcm (*int-of-integer x*) (*int-of-integer y*) = *int-of-integer* (*lcm x y*)
⟨*proof*⟩

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*⟩

declare *of-int-code-if* [*code*]

lemma [*code*]:
nat = *nat-of-integer* ∘ *of-int*
including *integer.lifting* ⟨*proof*⟩

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*
⟨*proof*⟩

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
  ⟨proof⟩

lemma [code-unfold del]:
  0 ≡ (of-rat 0 :: real)
  ⟨proof⟩
```

lemma [*code-unfold del*]:
1 ≡ (*of-rat 1* :: *real*)
⟨*proof*⟩

lemma [*code-unfold del*]:
numeral k ≡ (*of-rat* (*numeral k*) :: *real*)
⟨*proof*⟩

lemma [*code-unfold del*]:
− *numeral k* ≡ (*of-rat* (− *numeral k*) :: *real*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma [*code*, *code del*]:
(*HOL.equal* :: *real*=>*real*=>*bool*) = (*HOL.equal* :: *real* => *real* => *bool*)
⟨*proof*⟩

lemma [*code*, *code del*]:
(*plus* :: *real* => *real* => *real*) = *plus*
⟨*proof*⟩

lemma [*code*, *code del*]:
(*uminus* :: *real* => *real*) = *uminus*
⟨*proof*⟩

lemma [*code*, *code del*]:
(*minus* :: *real* => *real* => *real*) = *minus*
⟨*proof*⟩

lemma [*code*, *code del*]:
(*times* :: *real* => *real* => *real*) = *times*

⟨*proof*⟩

lemma [*code*, *code del*]:
(*divide* :: *real* => *real* => *real*) = *divide*
⟨*proof*⟩

lemma [*code*]:
fixes *r* :: *real*
shows *inverse r* = *1* / *r*
⟨*proof*⟩

notepad
begin
⟨*proof*⟩
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*
⟨*proof*⟩

lemma [*code-post*]:
Nat 0 = *0*
Nat 1 = *1*
Nat (*numeral k*) = *numeral k*
⟨*proof*⟩

lemma [*code-abbrev*]:
integer-of-nat = *of-nat*
⟨*proof*⟩

lemma [*code-unfold*]:
Int.nat (*int-of-integer k*) = *nat-of-integer k*
⟨*proof*⟩

lemma [*code abstype*]:
Code-Target-Nat.Nat (*integer-of-nat n*) = *n*
⟨*proof*⟩

lemma [*code abstract*]:
integer-of-nat (*nat-of-integer k*) = *max 0 k*
⟨*proof*⟩

lemma [*code-abbrev*]:
nat-of-integer (*numeral k*) = *nat-of-num k*
⟨*proof*⟩

lemma [*code abstract*]:
integer-of-nat (*nat-of-num n*) = *integer-of-num n*
⟨*proof*⟩

lemma [*code abstract*]:
integer-of-nat 0 = *0*
⟨*proof*⟩

lemma [*code abstract*]:
integer-of-nat 1 = *1*
⟨*proof*⟩

lemma [*code*]:
Suc n = *n* + *1*
⟨*proof*⟩

lemma [*code abstract*]:
integer-of-nat (*m* + *n*) = *of-nat m* + *of-nat n*
⟨*proof*⟩

lemma [*code abstract*]:
integer-of-nat (*m* − *n*) = *max 0* (*of-nat m* − *of-nat n*)
⟨*proof*⟩

lemma [*code abstract*]:
integer-of-nat (*m* ∗ *n*) = *of-nat m* ∗ *of-nat n*
⟨*proof*⟩

lemma [*code abstract*]:
integer-of-nat (*m div n*) = *of-nat m div of-nat n*
⟨*proof*⟩

lemma [*code abstract*]:
integer-of-nat (*m mod n*) = *of-nat m mod of-nat n*
⟨*proof*⟩

lemma [*code*]:
Divides.divmod-nat m n = (*m div n*, *m mod n*)
⟨*proof*⟩

lemma [*code*]:
divmod m n = *map-prod nat-of-integer nat-of-integer* (*divmod m n*)
⟨*proof*⟩

lemma [*code*]:
HOL.equal m n = *HOL.equal* (*of-nat m* :: *integer*) (*of-nat n*)
⟨*proof*⟩

lemma [*code*]:
m ≤ *n* ⟷ (*of-nat m* :: *integer*) ≤ *of-nat n*
⟨*proof*⟩

lemma [*code*]:
m < *n* ⟷ (*of-nat m* :: *integer*) < *of-nat n*
⟨*proof*⟩

lemma *num-of-nat-code* [*code*]:
num-of-nat = *num-of-integer* ∘ *of-nat*
⟨*proof*⟩

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*⟩

declare *of-nat-code-if* [*code*]

definition *int-of-nat* :: *nat* ⇒ *int* **where**
[*code-abbrev*]: *int-of-nat* = *of-nat*

lemma [*code*]:
int-of-nat n = *int-of-integer* (*of-nat n*)
⟨*proof*⟩

lemma [*code abstract*]:
integer-of-nat (*nat k*) = *max 0* (*integer-of-int k*)
including *integer.lifting* ⟨*proof*⟩

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))
⟨*proof*⟩

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* ⟨*proof*⟩

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) ⟹ *distinct (map fst (List.filter P xs))*
⟨*proof*⟩

105.2 Type $('key, 'value)\ alist$

typedef $('key, 'value)\ alist = \{xs :: ('key \times 'value)\ list.\ (distinct \circ map\ fst)\ xs\}$
morphisms *impl-of Alist*
⟨*proof*⟩

setup-lifting *type-definition-alist*

lemma *alist-ext*: $impl\text{-}of\ xs = impl\text{-}of\ ys \Longrightarrow xs = ys$
⟨*proof*⟩

lemma *alist-eq-iff*: $xs = ys \longleftrightarrow impl\text{-}of\ xs = impl\text{-}of\ ys$
⟨*proof*⟩

lemma *impl-of-distinct* [*simp*, *intro*]: $distinct\ (map\ fst\ (impl\text{-}of\ xs))$
⟨*proof*⟩

lemma *Alist-impl-of* [*code abstype*]: $Alist\ (impl\text{-}of\ xs) = xs$
⟨*proof*⟩

105.3 Primitive operations

lift-definition *lookup* :: $('key, 'value)\ alist \Rightarrow 'key \Rightarrow 'value\ option$ **is** *map-of*
⟨*proof*⟩

lift-definition *empty* :: $('key, 'value)\ alist$ **is** $[]$
⟨*proof*⟩

lift-definition *update* :: $'key \Rightarrow 'value \Rightarrow ('key, 'value)\ alist \Rightarrow ('key, 'value)\ alist$
is *AList.update*
⟨*proof*⟩

lift-definition *delete* :: $'key \Rightarrow ('key, 'value)\ alist \Rightarrow ('key, 'value)\ alist$
is *AList.delete*
⟨*proof*⟩

lift-definition *map-entry* ::
$'key \Rightarrow ('value \Rightarrow 'value) \Rightarrow ('key, 'value)\ alist \Rightarrow ('key, 'value)\ alist$
is *AList.map-entry*
⟨*proof*⟩

lift-definition *filter* :: $('key \times 'value \Rightarrow bool) \Rightarrow ('key, 'value)\ alist \Rightarrow ('key, 'value)\ alist$
is *List.filter*
⟨*proof*⟩

lift-definition *map-default* ::
$'key \Rightarrow 'value \Rightarrow ('value \Rightarrow 'value) \Rightarrow ('key, 'value)\ alist \Rightarrow ('key, 'value)\ alist$
is *AList.map-default*

⟨*proof*⟩

105.4 Abstract operation properties

lemma *lookup-empty* [*simp*]: *lookup empty k* = *None*
⟨*proof*⟩

lemma *lookup-delete* [*simp*]: *lookup* (*delete k al*) = (*lookup al*)(*k* := *None*)
⟨*proof*⟩

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
⟨*proof*⟩

end

105.5.2 Size

instantiation *alist* :: (*type*, *type*) *size*
begin

definition *size* (*al* :: ('*a*, '*b*) *alist*) = *length* (*impl-of al*)

instance ⟨*proof*⟩

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 ∘→ (λk. Quickcheck-Random.random j ∘→
(λv. random-aux-alist (i − 1) j ∘→ (λa. 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 ⟨*proof*⟩

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
(λa. Quickcheck-Exhaustive.exhaustive
(λk. Quickcheck-Exhaustive.exhaustive (λv. f (update k v a)) (i − 1)) (i − 1))
(i − 1))

instance ⟨*proof*⟩

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 ⟨*proof*⟩

end

106 alist is a BNF

lift-bnf (*dead* '*k*, *set*: '*v*) *alist* [*wits*: [] :: ('*k* × '*v*) *list*] **for** *map*: *map rel*: *rel*
⟨*proof*⟩

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*]: {#} = {#} ⟨*proof*⟩

lemma [*code*, *code del*]: *single* = *single* ⟨*proof*⟩

lemma [*code*, *code del*]: *plus* = (*plus* :: *'a multiset* ⇒ -) ⟨*proof*⟩

lemma [*code*, *code del*]: *minus* = (*minus* :: *'a multiset* ⇒ -) ⟨*proof*⟩

lemma [*code*, *code del*]: *inf-subset-mset* = (*inf-subset-mset* :: *'a multiset* ⇒ -) ⟨*proof*⟩

lemma [*code*, *code del*]: *sup-subset-mset* = (*sup-subset-mset* :: *'a multiset* ⇒ -) ⟨*proof*⟩

lemma [*code*, *code del*]: *image-mset* = *image-mset* ⟨*proof*⟩

lemma [*code*, *code del*]: *filter-mset* = *filter-mset* ⟨*proof*⟩

lemma [*code*, *code del*]: *count* = *count* ⟨*proof*⟩

lemma [*code*, *code del*]: *size* = (*size* :: - *multiset* ⇒ *nat*) ⟨*proof*⟩

lemma [*code*, *code del*]: *msetsum* = *msetsum* ⟨*proof*⟩

lemma [*code*, *code del*]: *msetprod* = *msetprod* ⟨*proof*⟩

lemma [*code*, *code del*]: *set-mset* = *set-mset* ⟨*proof*⟩

lemma [*code*, *code del*]: *sorted-list-of-multiset* = *sorted-list-of-multiset* ⟨*proof*⟩

lemma [*code*, *code del*]: *subset-mset* = *subset-mset* ⟨*proof*⟩

lemma [*code*, *code del*]: *subseteq-mset* = *subseteq-mset* ⟨*proof*⟩

lemma [*code*, *code del*]: *equal-multiset-inst.equal-multiset* = *equal-multiset-inst.equal-multiset* ⟨*proof*⟩

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*
 ⟨*proof*⟩

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*)
 ⟨*proof*⟩

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′*))))
⟨*proof*⟩

lemma *distinct-join-raw*:
assumes *distinct* (*map fst xs*)
shows *distinct* (*map fst* (*join-raw f xs ys*))
⟨*proof*⟩

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′*)))
⟨*proof*⟩

lemma *distinct-subtract-entries-raw*:
assumes *distinct* (*map fst xs*)
shows *distinct* (*map fst* (*subtract-entries-raw xs ys*))
⟨*proof*⟩

Operations on alists with distinct keys

lift-definition *join* :: ('*a* ⇒ '*b* × '*b* ⇒ '*b*) ⇒ ('*a*, '*b*) *alist* ⇒ ('*a*, '*b*) *alist* ⇒ ('*a*, '*b*) *alist*
is *join-raw*
⟨*proof*⟩

lift-definition *subtract-entries* :: ('*a*, ('*b* :: *minus*)) *alist* ⇒ ('*a*, '*b*) *alist* ⇒ ('*a*, '*b*) *alist*
is *subtract-entries-raw*
⟨*proof*⟩

Implementing multisets by means of association lists

definition *count-of* :: ('*a* × *nat*) *list* ⇒ '*a* ⇒ *nat*
where *count-of xs x* = (*case map-of xs x of None* ⇒ *0* | *Some n* ⇒ *n*)

lemma *count-of-multiset*: *count-of xs* ∈ *multiset*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *count-of-empty*: *x* ∉ *fst* ' *set xs* ⟹ *count-of xs x* = *0*
⟨*proof*⟩

lemma *count-of-filter*: *count-of* (*List.filter* (*P* ∘ *fst*) *xs*) *x* = (*if P x then count-of xs x else 0*)
⟨*proof*⟩

lemma *count-of-map-default* [*simp*]:
count-of (*map-default x b* (λ*x*. *x* + *b*) *xs*) *y* =
(*if x = y then count-of xs x* + *b else count-of xs y*)
⟨*proof*⟩

lemma *count-of-join-raw*:
distinct (*map fst ys*) ⟹
count-of xs x + *count-of ys x* = *count-of* (*join-raw* (λ*x* (*x*, *y*). *x* + *y*) *xs ys*) *x*
⟨*proof*⟩

lemma *count-of-subtract-entries-raw*:
distinct (*map fst ys*) ⟹
count-of xs x − *count-of ys x* = *count-of* (*subtract-entries-raw xs ys*) *x*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *Mempty-Bag* [*code*]: {#} = *Bag* (*DAList.empty*)
⟨*proof*⟩

lemma *single-Bag* [*code*]: {#*x*#} = *Bag* (*DAList.update x 1 DAList.empty*)
⟨*proof*⟩

lemma *union-Bag* [*code*]: *Bag xs* + *Bag ys* = *Bag* (*join* (λ*x* (*n1*, *n2*). *n1* + *n2*) *xs ys*)
⟨*proof*⟩

lemma *minus-Bag* [*code*]: *Bag xs* − *Bag ys* = *Bag* (*subtract-entries xs ys*)
⟨*proof*⟩

lemma *filter-Bag* [*code*]: *filter-mset P* (*Bag xs*) = *Bag* (*DAList.filter* (*P* ∘ *fst*) *xs*)
⟨*proof*⟩

lemma *mset-eq* [*code*]: *HOL.equal* (*m1*::'*a*::*equal multiset*) *m2* ⟷ *m1* ≤# *m2* ∧ *m2* ≤# *m1*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *mset-less-eq-Bag* [*code*]:
Bag xs ≤# (*A* :: '*a multiset*) ⟷ (∀(*x*, *n*) ∈ *set* (*DAList.impl-of xs*). *n* ≤ *count A x*)
⟨*proof*⟩

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*

⟨*proof*⟩

end

context
begin

private lift-definition *single-alist-entry* :: *'a* ⇒ *'b* ⇒ (*'a*, *'b*) *alist* **is** λ*a b*. [(*a*, *b*)]
⟨*proof*⟩

lemma *image-mset-Bag* [*code*]:
image-mset f (*Bag ms*) =
DAList-Multiset.fold (λ*a n m*. *Bag* (*single-alist-entry* (*f a*) *n*) + *m*) {#} *ms*
⟨*proof*⟩

end

lemma *msetsum-Bag*[*code*]: *msetsum* (*Bag ms*) = *DAList-Multiset.fold* (λ*a n*. ((*op* + *a*) ^^ *n*)) *0 ms*
⟨*proof*⟩

lemma *msetprod-Bag*[*code*]: *msetprod* (*Bag ms*) = *DAList-Multiset.fold* (λ*a n*. ((*op* ∗ *a*) ^^ *n*)) *1 ms*
⟨*proof*⟩

lemma *size-fold*: *size A* = *fold-mset* (λ-. *Suc*) *0 A* (**is** - = *fold-mset ?f* - -)
⟨*proof*⟩

lemma *size-Bag*[*code*]: *size* (*Bag ms*) = *DAList-Multiset.fold* (λ*a n*. *op* + *n*) *0 ms*
⟨*proof*⟩

lemma *set-mset-fold*: *set-mset A* = *fold-mset insert* {} *A* (**is** - = *fold-mset ?f* - -)
⟨*proof*⟩

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*
⟨*proof*⟩

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 ⟨*proof*⟩

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*⟩

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*

⟨*proof*⟩

lemma *entry-in-tree-keys*:
assumes $(k, v) \in set\ (entries\ t)$
shows $k \in set\ (keys\ t)$
⟨*proof*⟩

lemma *keys-entries*:
$k \in set\ (keys\ t) \longleftrightarrow (\exists\, v.\ (k, v) \in set\ (entries\ t))$
⟨*proof*⟩

lemma *non-empty-rbt-keys*:
$t \neq rbt.Empty \Longrightarrow keys\ t \neq []$
⟨*proof*⟩

108.2.2 Search tree properties

context *ord* **begin**

definition *rbt-less* :: $'a \Rightarrow ('a, 'b)\ rbt \Rightarrow bool$
where
rbt-less-prop: $rbt\text{-}less\ k\ t \longleftrightarrow (\forall\, x{\in}set\ (keys\ t).\ x < k)$

abbreviation *rbt-less-symbol* (**infix** $|\ll$ *50*)
where $t\ |\ll\ x \equiv rbt\text{-}less\ x\ t$

definition *rbt-greater* :: $'a \Rightarrow ('a, 'b)\ rbt \Rightarrow bool$ (**infix** $\ll|$ *50*)
where
rbt-greater-prop: $rbt\text{-}greater\ k\ t = (\forall\, x{\in}set\ (keys\ t).\ k < x)$

lemma *rbt-less-simps* [*simp*]:
$Empty\ |\ll\ k = True$
$Branch\ c\ lt\ kt\ v\ rt\ |\ll\ k \longleftrightarrow kt < k \wedge lt\ |\ll\ k \wedge rt\ |\ll\ k$
⟨*proof*⟩

lemma *rbt-greater-simps* [*simp*]:
$k \ll|\ Empty = True$
$k \ll|\ (Branch\ c\ lt\ kt\ v\ rt) \longleftrightarrow k < kt \wedge k \ll|\ lt \wedge k \ll|\ rt$
⟨*proof*⟩

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*: $l\ |\ll\ u \Longrightarrow u \leq v \Longrightarrow l\ |\ll\ v$
and *rbt-less-trans*: $t\ |\ll\ x \Longrightarrow x < y \Longrightarrow t\ |\ll\ y$
and *rbt-greater-eq-trans*: $u \leq v \Longrightarrow v \ll|\ r \Longrightarrow u \ll|\ r$

and *rbt-greater-trans*: $x < y \Longrightarrow y \ll| t \Longrightarrow x \ll| t$
⟨*proof*⟩

primrec *rbt-sorted* :: $('a, 'b)$ *rbt* ⇒ *bool*
where
rbt-sorted Empty = *True*
| *rbt-sorted* (*Branch c l k v r*) = (*l* |≪ *k* ∧ *k* ≪| *r* ∧ *rbt-sorted l* ∧ *rbt-sorted r*)

end

context *linorder* **begin**

lemma *rbt-sorted-entries*:
rbt-sorted t ⟹ *List.sorted* (*map fst* (*entries t*))
⟨*proof*⟩

lemma *distinct-entries*:
rbt-sorted t ⟹ *distinct* (*map fst* (*entries t*))
⟨*proof*⟩

lemma *distinct-keys*:
rbt-sorted t ⟹ *distinct* (*keys t*)
⟨*proof*⟩

108.2.3 Tree lookup

primrec (**in** *ord*) *rbt-lookup* :: $('a, 'b)$ *rbt* ⇒ $'a \rightharpoonup 'b$
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*)
⟨*proof*⟩

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*⟩

lemma *finite-dom-rbt-lookup* [*simp*, *intro*!]: *finite* (*dom* (*rbt-lookup t*))
⟨*proof*⟩

end

context *ord* **begin**

lemma *rbt-lookup-rbt-less*[*simp*]: *t* |≪ *k* ⟹ *rbt-lookup t k* = *None*

⟨*proof*⟩

lemma *rbt-lookup-rbt-greater*[*simp*]: $k \ll\!| \ t \Longrightarrow$ *rbt-lookup t k* = *None*
⟨*proof*⟩

lemma *rbt-lookup-Empty*: *rbt-lookup Empty* = *empty*
⟨*proof*⟩

end

context *linorder* **begin**

lemma *map-of-entries*:
rbt-sorted t $\Longrightarrow$ *map-of* (*entries t*) = *rbt-lookup t*
⟨*proof*⟩

lemma *rbt-lookup-in-tree*: *rbt-sorted t* $\Longrightarrow$ *rbt-lookup t k* = *Some v* $\longleftrightarrow$ (*k*, *v*) $\in$ *set* (*entries t*)
⟨*proof*⟩

lemma *set-entries-inject*:
assumes *rbt-sorted*: *rbt-sorted t1 rbt-sorted t2*
shows *set* (*entries t1*) = *set* (*entries t2*) $\longleftrightarrow$ *entries t1* = *entries t2*
⟨*proof*⟩

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*⟩

lemma *entries-rbt-lookup*:
assumes *rbt-sorted t1 rbt-sorted t2*
shows *entries t1* = *entries t2* $\longleftrightarrow$ *rbt-lookup t1* = *rbt-lookup t2*
⟨*proof*⟩

lemma *rbt-lookup-from-in-tree*:
assumes *rbt-sorted t1 rbt-sorted t2*
and $\bigwedge v$. (*k*, *v*) $\in$ *set* (*entries t1*) $\longleftrightarrow$ (*k*, *v*) $\in$ *set* (*entries t2*)
shows *rbt-lookup t1 k* = *rbt-lookup t2 k*
⟨*proof*⟩

end

108.2.4 Red-black properties

primrec *color-of* :: (*'a*, *'b*) *rbt* $\Rightarrow$ *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* ⟨*proof*⟩

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* ⟨*proof*⟩

theorem *Empty-is-rbt* [*simp*]:
is-rbt Empty ⟨*proof*⟩

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)*
⟨*proof*⟩

lemma *balance-bheight*: *bheight l = bheight r* ⟹ *bheight (balance l k v r) = Suc (bheight l)*
⟨*proof*⟩

lemma *balance-inv2*:
assumes *inv2 l inv2 r bheight l = bheight r*
shows *inv2 (balance l k v r)*
⟨*proof*⟩

context *ord* **begin**

lemma *balance-rbt-greater*[*simp*]: *(v ≪| balance a k x b) = (v ≪| a ∧ v ≪| b ∧ v < k)*
⟨*proof*⟩

lemma *balance-rbt-less*[*simp*]: *(balance a k x b |≪ v) = (a |≪ v ∧ b |≪ v ∧ k < v)*
⟨*proof*⟩

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)*
⟨*proof*⟩

lemma *entries-balance* [*simp*]:
entries (balance l k v r) = entries l @ (k, v) # entries r
⟨*proof*⟩

lemma *keys-balance* [*simp*]:
keys (balance l k v r) = keys l @ k # keys r
⟨*proof*⟩

lemma *balance-in-tree*:
entry-in-tree k x (balance l v y r) ⟷ *entry-in-tree k x l ∨ k = v ∧ x = y ∨ entry-in-tree k x r*
⟨*proof*⟩

lemma (**in** *linorder*) *rbt-lookup-balance*[*simp*]:

fixes $k :: 'a$
assumes *rbt-sorted l rbt-sorted r l* $|\ll k\ k \ll|$ *r*
shows *rbt-lookup* (*balance l k v r*) *x* = *rbt-lookup* (*Branch B l k v r*) *x*
⟨*proof*⟩

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*) ⟨*proof*⟩
lemma *paint-inv1*[*simp*]: *inv1l t* ⟹ *inv1* (*paint B t*) ⟨*proof*⟩
lemma *paint-inv2*[*simp*]: *inv2 t* ⟹ *inv2* (*paint c t*) ⟨*proof*⟩
lemma *paint-color-of*[*simp*]: *color-of* (*paint B t*) = *B* ⟨*proof*⟩
lemma *paint-in-tree*[*simp*]: *entry-in-tree k x* (*paint c t*) = *entry-in-tree k x t* ⟨*proof*⟩

context *ord* **begin**

lemma *paint-rbt-sorted*[*simp*]: *rbt-sorted t* ⟹ *rbt-sorted* (*paint c t*) ⟨*proof*⟩
lemma *paint-rbt-lookup*[*simp*]: *rbt-lookup* (*paint c t*) = *rbt-lookup t* ⟨*proof*⟩
lemma *paint-rbt-greater*[*simp*]: (*v* $\ll|$ *paint c t*) = (*v* $\ll|$ *t*) ⟨*proof*⟩
lemma *paint-rbt-less*[*simp*]: (*paint c t* $|\ll$ *v*) = (*t* $|\ll$ *v*) ⟨*proof*⟩

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*)
⟨*proof*⟩

end

context *linorder* **begin**

lemma *ins-rbt-greater*[*simp*]: (*v* $\ll|$ *rbt-ins f* (*k* :: '*a*) *x t*) = (*v* $\ll|$ *t* ∧ *k* > *v*)
⟨*proof*⟩
lemma *ins-rbt-less*[*simp*]: (*rbt-ins f k x t* $|\ll$ *v*) = (*t* $|\ll$ *v* ∧ *k* < *v*)
⟨*proof*⟩
lemma *ins-rbt-sorted*[*simp*]: *rbt-sorted t* ⟹ *rbt-sorted* (*rbt-ins f k x t*)

⟨*proof*⟩

lemma *keys-ins*: *set* (*keys* (*rbt-ins f k v t*)) = { *k* } ∪ *set* (*keys t*)
⟨*proof*⟩

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*
⟨*proof*⟩

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* ⟹ *rbt-sorted* (*rbt-insert-with-key f* (*k* :: *'a*) *x t*)
⟨*proof*⟩

theorem *rbt-insertwk-is-rbt*:
assumes *inv*: *is-rbt t*
shows *is-rbt* (*rbt-insert-with-key f k x t*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *rbt-insertw-rbt-sorted*: *rbt-sorted t* ⟹ *rbt-sorted* (*rbt-insert-with f k v t*)
⟨*proof*⟩
theorem *rbt-insertw-is-rbt*: *is-rbt t* ⟹ *is-rbt* (*rbt-insert-with f k v t*)

⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *rbt-insert-rbt-sorted*: *rbt-sorted t* ⟹ *rbt-sorted* (*rbt-insert k v t*)
⟨*proof*⟩
theorem *rbt-insert-is-rbt* [*simp*]: *is-rbt t* ⟹ *is-rbt* (*rbt-insert k v t*)
⟨*proof*⟩

lemma *rbt-lookup-rbt-insert*:
assumes *is-rbt t*
shows *rbt-lookup* (*rbt-insert k v t*) = (*rbt-lookup t*)(*k*↦*v*)
⟨*proof*⟩

end

108.4 Deletion

lemma *bheight-paintR'*[*simp*]: *color-of t* = *B* ⟹ *bheight* (*paint R t*) = *bheight t* − *1*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *balance-left-inv1*: ⟦*inv1l a*; *inv1 b*; *color-of b = B*⟧ ⟹ *inv1* (*balance-left a k x b*)
⟨*proof*⟩

lemma *balance-left-inv1l*: ⟦ *inv1l lt*; *inv1 rt* ⟧ ⟹ *inv1l* (*balance-left lt k x rt*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *balance-left-rbt-less*:
fixes *k* :: *'a*
assumes *a* |≪ *k b* |≪ *k x < k*
shows *balance-left a x t b* |≪ *k*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *balance-right-inv1*: ⟦*inv1 a*; *inv1l b*; *color-of a* = *B*⟧ ⟹ *inv1* (*balance-right a k x b*)
⟨*proof*⟩

lemma *balance-right-inv1l*: ⟦ *inv1 lt*; *inv1l rt* ⟧ ⟹*inv1l* (*balance-right lt k x rt*)
⟨*proof*⟩

lemma (**in** *linorder*) *balance-right-rbt-sorted*:
⟦ *rbt-sorted l*; *rbt-sorted r*; *rbt-less k l*; *k* ≪| *r* ⟧ ⟹ *rbt-sorted* (*balance-right l k v r*)
⟨*proof*⟩

context *order* **begin**

lemma *balance-right-rbt-greater*:
fixes *k* :: *'a*
assumes *k* ≪| *a k* ≪| *b k* < *x*
shows *k* ≪| *balance-right a x t b*
⟨*proof*⟩

lemma *balance-right-rbt-less*:
fixes *k* :: *'a*
assumes *a* |≪ *k b* |≪ *k x* < *k*
shows *balance-right a x t b* |≪ *k*
⟨*proof*⟩

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*)
⟨*proof*⟩

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)*
⟨*proof*⟩

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)
⟨*proof*⟩

context *linorder* **begin**

lemma *combine-rbt-greater*[*simp*]:
fixes *k* :: *'a*
assumes *k ≪| l k ≪| r*
shows *k ≪| combine l r*
⟨*proof*⟩

lemma *combine-rbt-less*[*simp*]:
fixes *k* :: *'a*
assumes *l |≪ k r |≪ k*
shows *combine l r |≪ k*
⟨*proof*⟩

lemma *combine-rbt-sorted*:
fixes *k* :: *'a*
assumes *rbt-sorted l rbt-sorted r l |≪ k k ≪| r*
shows *rbt-sorted (combine l r)*
⟨*proof*⟩

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)*
⟨*proof*⟩

context *ord* **begin**

fun
rbt-del-from-left :: *'a ⇒ ('a,'b) rbt ⇒ 'a ⇒ 'b ⇒ ('a,'b) rbt ⇒ ('a,'b) rbt* **and**
rbt-del-from-right :: *'a ⇒ ('a,'b) rbt ⇒ 'a ⇒ 'b ⇒ ('a,'b) rbt ⇒ ('a,'b) rbt* **and**
rbt-del :: *'a⇒ ('a,'b) rbt ⇒ ('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*⟧ ⟹
inv2 (*rbt-del-from-left* $x\ lt\ k\ v\ rt$) ∧
bheight (*rbt-del-from-left* $x\ lt\ k\ v\ rt$) = *bheight lt* ∧
(*color-of lt* = B ∧ *color-of rt* = B ∧ *inv1* (*rbt-del-from-left* $x\ lt\ k\ v\ rt$) ∨
(*color-of lt* ≠ B ∨ *color-of rt* ≠ B) ∧ *inv1l* (*rbt-del-from-left* $x\ lt\ k\ v\ rt$))
and ⟦*inv2 rt*; *bheight lt* = *bheight rt*; *inv1 rt*⟧ ⟹
inv2 (*rbt-del-from-right* $x\ lt\ k\ v\ rt$) ∧
bheight (*rbt-del-from-right* $x\ lt\ k\ v\ rt$) = *bheight lt* ∧
(*color-of lt* = B ∧ *color-of rt* = B ∧ *inv1* (*rbt-del-from-right* $x\ lt\ k\ v\ rt$) ∨
(*color-of lt* ≠ B ∨ *color-of rt* ≠ B) ∧ *inv1l* (*rbt-del-from-right* $x\ lt\ k\ v\ rt$))
and *rbt-del-inv1-inv2*: *inv2* (*rbt-del* $x\ lt$) ∧ (*color-of lt* = R ∧ *bheight* (*rbt-del* x lt) = *bheight lt* ∧ *inv1* (*rbt-del* $x\ lt$)
∨ *color-of lt* = B ∧ *bheight* (*rbt-del* $x\ lt$) = *bheight lt* − 1 ∧ *inv1l* (*rbt-del* $x\ lt$))
⟨*proof*⟩

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
⟨*proof*⟩

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* $x\ lt$
⟨*proof*⟩

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* $\Longrightarrow$ *rbt-sorted* (*rbt-del x lt*)
⟨*proof*⟩

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*⟧ $\Longrightarrow$ *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*⟧ $\Longrightarrow$ *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*⟧ $\Longrightarrow$ *entry-in-tree k v* (*rbt-del x t*) = (*False* ∨ (*x* ≠ *k* ∧ *entry-in-tree k v t*))
⟨*proof*⟩

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*⟩

lemma *rbt-delete-in-tree*:
assumes *is-rbt t*
shows *entry-in-tree k v* (*rbt-delete x t*) = (*x* ≠ *k* ∧ *entry-in-tree k v t*)
⟨*proof*⟩

lemma *rbt-lookup-rbt-delete*:
assumes *is-rbt*: *is-rbt t*
shows *rbt-lookup* (*rbt-delete k t*) = (*rbt-lookup t*)|`(−{*k*})
⟨*proof*⟩

end

108.5 Modifying existing entries

context *ord* **begin**

primrec
rbt-map-entry :: *'a* ⇒ (*'b* ⇒ *'b*) ⇒ (*'a*, *'b*) *rbt* ⇒ (*'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* ⟨*proof*⟩
lemma *rbt-map-entry-inv1*: *inv1* (*rbt-map-entry k f t*) = *inv1 t* ⟨*proof*⟩
lemma *rbt-map-entry-inv2*: *inv2* (*rbt-map-entry k f t*) = *inv2 t bheight* (*rbt-map-entry*

k f t) = *bheight t* ⟨*proof*⟩
lemma *rbt-map-entry-rbt-greater*: *rbt-greater a* (*rbt-map-entry k f t*) = *rbt-greater a t* ⟨*proof*⟩
lemma *rbt-map-entry-rbt-less*: *rbt-less a* (*rbt-map-entry k f t*) = *rbt-less a t* ⟨*proof*⟩
lemma *rbt-map-entry-rbt-sorted*: *rbt-sorted* (*rbt-map-entry k f t*) = *rbt-sorted t*
⟨*proof*⟩

theorem *rbt-map-entry-is-rbt* [*simp*]: *is-rbt* (*rbt-map-entry k f t*) = *is-rbt t*
⟨*proof*⟩

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*))
⟨*proof*⟩

108.6 Mapping all entries

primrec
map :: ('*a* ⇒ '*b* ⇒ '*c*) ⇒ ('*a*, '*b*) *rbt* ⇒ ('*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*)
⟨*proof*⟩
lemma *map-keys* [*simp*]: *keys* (*map f t*) = *keys t* ⟨*proof*⟩
lemma *map-color-of*: *color-of* (*map f t*) = *color-of t* ⟨*proof*⟩
lemma *map-inv1*: *inv1* (*map f t*) = *inv1 t* ⟨*proof*⟩
lemma *map-inv2*: *inv2* (*map f t*) = *inv2 t bheight* (*map f t*) = *bheight t* ⟨*proof*⟩

context *ord* **begin**

lemma *map-rbt-greater*: *rbt-greater k* (*map f t*) = *rbt-greater k t* ⟨*proof*⟩
lemma *map-rbt-less*: *rbt-less k* (*map f t*) = *rbt-less k t* ⟨*proof*⟩
lemma *map-rbt-sorted*: *rbt-sorted* (*map f t*) = *rbt-sorted t* ⟨*proof*⟩
theorem *map-is-rbt* [*simp*]: *is-rbt* (*map f t*) = *is-rbt t*
⟨*proof*⟩

end

theorem (**in** *linorder*) *rbt-lookup-map*: *rbt-lookup* (*map f t*) *x* = *map-option* (*f x*) (*rbt-lookup t x*)
⟨*proof*⟩

hide-const (**open**) *map*

108.7 Folding over entries

definition *fold* :: $('a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'c) \Rightarrow ('a, 'b)\ rbt \Rightarrow 'c \Rightarrow 'c$ **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
⟨*proof*⟩

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))
⟨*proof*⟩

fun *foldi* :: $('c \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'c) \Rightarrow ('a :: linorder, 'b)\ rbt \Rightarrow 'c \Rightarrow$
$'c$
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* :: $('a \times 'b)\ list \Rightarrow ('a, 'b)\ rbt$ **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)
⟨*proof*⟩

lemma *rbt-lookup-rbt-bulkload*:
rbt-lookup (rbt-bulkload xs) = map-of xs
⟨*proof*⟩

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'))
⟨proof⟩

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′))
⟨*proof*⟩

lemma *Suc-double-half*: *Suc (2 ∗ n) div 2 = n*
⟨*proof*⟩

lemma *div2-plus-div2*: *n div 2 + n div 2 = (n :: nat) − n mod 2*
⟨*proof*⟩

lemma *rbtreeify-f-rec-aux-lemma*:
⟦k − n div 2 = Suc k′; n ≤ k; n mod 2 = Suc 0⟧
⟹ k′ − n div 2 = k − n
⟨*proof*⟩

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′))
⟨*proof*⟩

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′))
⟨*proof*⟩

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*⟩

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*⟩

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*))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *bheight-rbtreeify-f*: $n \leq length\ kvs$
$\Longrightarrow bheight\ (fst\ (rbtreeify\text{-}f\ n\ kvs)) = plog2\ n$
and *bheight-rbtreeify-g*: $n \leq Suc\ (length\ kvs)$
$\Longrightarrow bheight\ (fst\ (rbtreeify\text{-}g\ n\ kvs)) = plog2\ n$
⟨*proof*⟩

lemma *bheight-rbtreeify-f-eq-plog2I*:
$[\![\ rbtreeify\text{-}f\ n\ kvs = (t,\ kvs');\ n \leq length\ kvs\ ]\!]$
$\Longrightarrow bheight\ t = plog2\ n$
⟨*proof*⟩

lemma *bheight-rbtreeify-g-eq-plog2I*:
$[\![\ rbtreeify\text{-}g\ n\ kvs = (t,\ kvs');\ n \leq Suc\ (length\ kvs)\ ]\!]$
$\Longrightarrow bheight\ t = plog2\ n$
⟨*proof*⟩

hide-const (**open**) *plog2*

lemma *inv2-rbtreeify-f*: $n \leq length\ kvs$
$\Longrightarrow inv2\ (fst\ (rbtreeify\text{-}f\ n\ kvs))$
and *inv2-rbtreeify-g*: $n \leq Suc\ (length\ kvs)$
$\Longrightarrow inv2\ (fst\ (rbtreeify\text{-}g\ n\ kvs))$
⟨*proof*⟩

lemma $n \leq length\ kvs \Longrightarrow True$
and *color-of-rbtreeify-g*:
$[\![\ n \leq Suc\ (length\ kvs);\ 0 < n\ ]\!]$
$\Longrightarrow color\text{-}of\ (fst\ (rbtreeify\text{-}g\ n\ kvs)) = B$
⟨*proof*⟩

lemma *entries-rbtreeify-f-append*:
$n \leq length\ kvs$
$\Longrightarrow entries\ (fst\ (rbtreeify\text{-}f\ n\ kvs))\ @\ snd\ (rbtreeify\text{-}f\ n\ kvs) = kvs$
and *entries-rbtreeify-g-append*:
$n \leq Suc\ (length\ kvs)$
$\Longrightarrow entries\ (fst\ (rbtreeify\text{-}g\ n\ kvs))\ @\ snd\ (rbtreeify\text{-}g\ n\ kvs) = kvs$
⟨*proof*⟩

lemma *length-entries-rbtreeify-f*:
$n \leq length\ kvs \Longrightarrow length\ (entries\ (fst\ (rbtreeify\text{-}f\ n\ kvs))) = n$
and *length-entries-rbtreeify-g*:
$n \leq Suc\ (length\ kvs) \Longrightarrow length\ (entries\ (fst\ (rbtreeify\text{-}g\ n\ kvs))) = n - 1$
⟨*proof*⟩

lemma *rbtreeify-f-conv-drop*:
$n \leq length\ kvs \Longrightarrow snd\ (rbtreeify\text{-}f\ n\ kvs) = drop\ n\ kvs$
⟨*proof*⟩

lemma *rbtreeify-g-conv-drop*:

$n \leq Suc\ (length\ kvs) \Longrightarrow snd\ (rbtreeify\text{-}g\ n\ kvs) = drop\ (n - 1)\ kvs$
⟨*proof*⟩

lemma *entries-rbtreeify-f* [*simp*]:
$n \leq length\ kvs \Longrightarrow entries\ (fst\ (rbtreeify\text{-}f\ n\ kvs)) = take\ n\ kvs$
⟨*proof*⟩

lemma *entries-rbtreeify-g* [*simp*]:
$n \leq Suc\ (length\ kvs) \Longrightarrow$
$entries\ (fst\ (rbtreeify\text{-}g\ n\ kvs)) = take\ (n - 1)\ kvs$
⟨*proof*⟩

lemma *keys-rbtreeify-f* [*simp*]: $n \leq length\ kvs$
$\Longrightarrow keys\ (fst\ (rbtreeify\text{-}f\ n\ kvs)) = take\ n\ (map\ fst\ kvs)$
⟨*proof*⟩

lemma *keys-rbtreeify-g* [*simp*]: $n \leq Suc\ (length\ kvs)$
$\Longrightarrow keys\ (fst\ (rbtreeify\text{-}g\ n\ kvs)) = take\ (n - 1)\ (map\ fst\ kvs)$
⟨*proof*⟩

lemma *rbtreeify-fD*:
$[\![\ rbtreeify\text{-}f\ n\ kvs = (t,\ kvs');\ n \leq length\ kvs\ ]\!]$
$\Longrightarrow entries\ t = take\ n\ kvs \wedge kvs' = drop\ n\ kvs$
⟨*proof*⟩

lemma *rbtreeify-gD*:
$[\![\ rbtreeify\text{-}g\ n\ kvs = (t,\ kvs');\ n \leq Suc\ (length\ kvs)\ ]\!]$
$\Longrightarrow entries\ t = take\ (n - 1)\ kvs \wedge kvs' = drop\ (n - 1)\ kvs$
⟨*proof*⟩

lemma *entries-rbtreeify* [*simp*]: $entries\ (rbtreeify\ kvs) = kvs$
⟨*proof*⟩

context *linorder* **begin**

lemma *rbt-sorted-rbtreeify-f*:
$[\![\ n \leq length\ kvs;\ sorted\ (map\ fst\ kvs);\ distinct\ (map\ fst\ kvs)\ ]\!]$
$\Longrightarrow rbt\text{-}sorted\ (fst\ (rbtreeify\text{-}f\ n\ kvs))$
and *rbt-sorted-rbtreeify-g*:
$[\![\ n \leq Suc\ (length\ kvs);\ sorted\ (map\ fst\ kvs);\ distinct\ (map\ fst\ kvs)\ ]\!]$
$\Longrightarrow rbt\text{-}sorted\ (fst\ (rbtreeify\text{-}g\ n\ kvs))$
⟨*proof*⟩

lemma *rbt-sorted-rbtreeify*:
$[\![\ sorted\ (map\ fst\ kvs);\ distinct\ (map\ fst\ kvs)\ ]\!] \Longrightarrow rbt\text{-}sorted\ (rbtreeify\ kvs)$
⟨*proof*⟩

lemma *is-rbt-rbtreeify*:
$[\![\ sorted\ (map\ fst\ kvs);\ distinct\ (map\ fst\ kvs)\ ]\!]$

$\Longrightarrow$ *is-rbt* (*rbtreeify kvs*)
⟨*proof*⟩

lemma *rbt-lookup-rbtreeify*:
⟦ *sorted* (*map fst kvs*); *distinct* (*map fst kvs*) ⟧ $\Longrightarrow$
rbt-lookup (*rbtreeify kvs*) = *map-of kvs*
⟨*proof*⟩

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*
⟨*proof*⟩
termination ⟨*proof*⟩

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 - [] = []*
⟨*proof*⟩
termination ⟨*proof*⟩

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)
⟨*proof*⟩

lemma *sorted-sunion-with* [*simp*]:
⟦ sorted (map fst xs); sorted (map fst ys) ⟧
⟹ sorted (map fst (sunion-with f xs ys))
⟨*proof*⟩

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*⟩

lemma *map-of-sunion-with*:
⟦ *sorted* (*map fst xs*); *sorted* (*map fst ys*) ⟧
⟹ *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*))
⟨*proof*⟩

lemma *set-fst-sinter-with* [*simp*]:
⟦ *sorted* (*map fst xs*); *sorted* (*map fst ys*) ⟧
⟹ *set* (*map fst* (*sinter-with f xs ys*)) = *set* (*map fst xs*) ∩ *set* (*map fst ys*)
⟨*proof*⟩

lemma *set-fst-sinter-with-subset1*:
set (*map fst* (*sinter-with f xs ys*)) ⊆ *set* (*map fst xs*)
⟨*proof*⟩

lemma *set-fst-sinter-with-subset2*:
set (*map fst* (*sinter-with f xs ys*)) ⊆ *set* (*map fst ys*)
⟨*proof*⟩

lemma *sorted-sinter-with* [*simp*]:
⟦ *sorted* (*map fst xs*); *sorted* (*map fst ys*) ⟧
⟹ *sorted* (*map fst* (*sinter-with f xs ys*))
⟨*proof*⟩

lemma *distinct-sinter-with* [*simp*]:
⟦ *distinct* (*map fst xs*); *distinct* (*map fst ys*) ⟧
⟹ *distinct* (*map fst* (*sinter-with f xs ys*))
⟨*proof*⟩

lemma *map-of-sinter-with*:
⟦ *sorted* (*map fst xs*); *sorted* (*map fst ys*) ⟧
⟹ *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*))
⟨*proof*⟩

end

lemma *distinct-map-of-rev*: *distinct* (*map fst xs*) ⟹ *map-of* (*rev xs*) = *map-of xs*

⟨*proof*⟩

lemma *map-map-filter*:
map f (*List.map-filter g xs*) = *List.map-filter* (*map-option f* ∘ *g*) *xs*
⟨*proof*⟩

lemma *map-filter-map-option-const*:
List.map-filter (λ*x*. *map-option* (λ*y*. *f x*) (*g* (*f x*))) *xs* = *filter* (λ*x*. *g x* ≠ *None*) (*map f xs*)
⟨*proof*⟩

lemma *set-map-filter*: *set* (*List.map-filter P xs*) = *the* ‘ (*P* ‘ *set xs* − {*None*})
⟨*proof*⟩

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'*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *is-rbt-fold-rbt-insertwk*:
assumes *is-rbt t1*
shows *is-rbt* (*fold* (*rbt-insert-with-key f*) *t2 t1*)
⟨*proof*⟩

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*⟩

lemma *is-rbt-rbt-unionwk* [*simp*]:
⟦ *is-rbt t1*; *is-rbt t2* ⟧ ⟹ *is-rbt* (*rbt-union-with-key f t1 t2*)
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *rbt-unionw-is-rbt*: ⟦ *is-rbt lt*; *is-rbt rt* ⟧ ⟹ *is-rbt* (*rbt-union-with f lt rt*)
⟨*proof*⟩

lemma *rbt-union-is-rbt*: ⟦ *is-rbt lt*; *is-rbt rt* ⟧ ⟹ *is-rbt* (*rbt-union lt rt*)
⟨*proof*⟩

lemma *rbt-lookup-rbt-union*:
⟦ *rbt-sorted s*; *rbt-sorted t* ⟧ ⟹
rbt-lookup (*rbt-union s t*) = *rbt-lookup s* ++ *rbt-lookup t*
⟨*proof*⟩

lemma *rbt-interwk-is-rbt* [*simp*]:

⟦ *rbt-sorted t1*; *rbt-sorted t2* ⟧ ⟹ *is-rbt* (*rbt-inter-with-key f t1 t2*)
⟨*proof*⟩

lemma *rbt-interw-is-rbt*:
⟦ *rbt-sorted t1*; *rbt-sorted t2* ⟧ ⟹ *is-rbt* (*rbt-inter-with f t1 t2*)
⟨*proof*⟩

lemma *rbt-inter-is-rbt*:
⟦ *rbt-sorted t1*; *rbt-sorted t2* ⟧ ⟹ *is-rbt* (*rbt-inter t1 t2*)
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *rbt-lookup-rbt-inter*:
⟦ *rbt-sorted t1*; *rbt-sorted t2* ⟧
⟹ *rbt-lookup* (*rbt-inter t1 t2*) = *rbt-lookup t2* |‘ *dom* (*rbt-lookup t1*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *entries-code* [*code*]:
entries = *gen-entries* []
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *keys-code* [*code*]:
keys = *gen-keys* []
⟨*proof*⟩

Restore original type constraints for constants

⟨*ML*⟩

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**) $('a, 'b)\ rbt = \{t :: ('a::linorder, 'b)\ RBT\text{-}Impl.rbt.\ is\text{-}rbt\ t\}$
  morphisms *impl-of RBT*
⟨*proof*⟩

lemma *rbt-eq-iff*:
  $t1 = t2 \longleftrightarrow impl\text{-}of\ t1 = impl\text{-}of\ t2$
  ⟨*proof*⟩

lemma *rbt-eqI*:
  $impl\text{-}of\ t1 = impl\text{-}of\ t2 \Longrightarrow t1 = t2$
  ⟨*proof*⟩

lemma *is-rbt-impl-of* [*simp*, *intro*]:
  $is\text{-}rbt\ (impl\text{-}of\ t)$
  ⟨*proof*⟩

lemma *RBT-impl-of* [*simp*, *code abstype*]:
  $RBT\ (impl\text{-}of\ t) = t$
  ⟨*proof*⟩

109.2 Primitive operations

setup-lifting *type-definition-rbt*

lift-definition $lookup :: ('a::linorder, 'b)\ rbt \Rightarrow 'a \rightharpoonup 'b$ **is** *rbt-lookup* ⟨*proof*⟩

lift-definition $empty :: ('a::linorder, 'b)\ rbt$ **is** *RBT-Impl.Empty*
⟨*proof*⟩

lift-definition $insert :: 'a::linorder \Rightarrow 'b \Rightarrow ('a, 'b)\ rbt \Rightarrow ('a, 'b)\ rbt$ **is** *rbt-insert*

⟨*proof*⟩

lift-definition $delete :: 'a::linorder \Rightarrow ('a, 'b)\ rbt \Rightarrow ('a, 'b)\ rbt$ **is** *rbt-delete*
⟨*proof*⟩

lift-definition $entries :: ('a::linorder, 'b)\ rbt \Rightarrow ('a \times 'b)\ list$ **is** *RBT-Impl.entries*
⟨*proof*⟩

lift-definition $keys :: ('a::linorder, 'b)\ rbt \Rightarrow 'a\ list$ **is** *RBT-Impl.keys* ⟨*proof*⟩

lift-definition $bulkload :: ('a::linorder \times 'b)\ list \Rightarrow ('a, 'b)\ rbt$ **is** *rbt-bulkload*
⟨*proof*⟩

lift-definition $map\text{-}entry :: 'a \Rightarrow ('b \Rightarrow 'b) \Rightarrow ('a::linorder, 'b)\ rbt \Rightarrow ('a, 'b)\ rbt$
is *rbt-map-entry*
⟨*proof*⟩

lift-definition *map* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('a::linorder, 'b)\ rbt \Rightarrow ('a, 'c)\ rbt$ **is** *RBT-Impl.map*
⟨*proof*⟩

lift-definition *fold* :: $('a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'c) \Rightarrow ('a::linorder, 'b)\ rbt \Rightarrow 'c \Rightarrow 'c$ **is** *RBT-Impl.fold* ⟨*proof*⟩

lift-definition *union* :: $('a::linorder, 'b)\ rbt \Rightarrow ('a, 'b)\ rbt \Rightarrow ('a, 'b)\ rbt$ **is** *rbt-union*
⟨*proof*⟩

lift-definition *foldi* :: $('c \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'c) \Rightarrow ('a :: linorder, 'b)\ rbt \Rightarrow 'c \Rightarrow 'c$
is *RBT-Impl.foldi* ⟨*proof*⟩

109.3 Derived operations

definition *is-empty* :: $('a::linorder, 'b)\ rbt \Rightarrow bool$ **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*
⟨*proof*⟩

lemma *lookup-impl-of*:
rbt-lookup (*impl-of t*) = *lookup t*
⟨*proof*⟩

lemma *entries-impl-of*:
RBT-Impl.entries (*impl-of t*) = *entries t*
⟨*proof*⟩

lemma *keys-impl-of*:
RBT-Impl.keys (*impl-of t*) = *keys t*
⟨*proof*⟩

lemma *lookup-keys*:
dom (*lookup t*) = *set* (*keys t*)
⟨*proof*⟩

lemma *lookup-empty* [*simp*]:
lookup empty = *Map.empty*
⟨*proof*⟩

lemma *lookup-insert* [*simp*]:
lookup (*insert k v t*) = (*lookup t*)(*k* ↦ *v*)
⟨*proof*⟩

lemma *lookup-delete* [*simp*]:
lookup (*delete k t*) = (*lookup t*)(*k* := *None*)
⟨*proof*⟩

lemma *map-of-entries* [*simp*]:
map-of (*entries t*) = *lookup t*
⟨*proof*⟩

lemma *entries-lookup*:
entries t1 = *entries t2* ⟷ *lookup t1* = *lookup t2*
⟨*proof*⟩

lemma *lookup-bulkload* [*simp*]:
lookup (*bulkload xs*) = *map-of xs*
⟨*proof*⟩

lemma *lookup-map-entry* [*simp*]:
lookup (*map-entry k f t*) = (*lookup t*)(*k* := *map-option f* (*lookup t k*))
⟨*proof*⟩

lemma *lookup-map* [*simp*]:
lookup (*map f t*) *k* = *map-option* (*f k*) (*lookup t k*)
⟨*proof*⟩

lemma *fold-fold*:
fold f t = *List.fold* (*case-prod f*) (*entries t*)
⟨*proof*⟩

lemma *impl-of-empty*:
impl-of empty = *RBT-Impl.Empty*
⟨*proof*⟩

lemma *is-empty-empty* [*simp*]:
is-empty t ⟷ *t* = *empty*
⟨*proof*⟩

lemma *RBT-lookup-empty* [*simp*]:
rbt-lookup t = *Map.empty* ⟷ *t* = *RBT-Impl.Empty*
⟨*proof*⟩

lemma *lookup-empty-empty* [*simp*]:
lookup t = *Map.empty* ⟷ *t* = *empty*
⟨*proof*⟩

lemma *sorted-keys* [*iff*]:
sorted (*keys t*)
⟨*proof*⟩

lemma *distinct-keys* [*iff*]:
distinct (*keys t*)
⟨*proof*⟩

lemma *finite-dom-lookup* [*simp*, *intro*!]: *finite* (*dom* (*lookup t*))
⟨*proof*⟩

lemma *lookup-union*: *lookup* (*union s t*) = *lookup s* ++ *lookup t*
⟨*proof*⟩

lemma *lookup-in-tree*: (*lookup t k* = *Some v*) = ((*k*, *v*) ∈ *set* (*entries t*))
⟨*proof*⟩

lemma *keys-entries*: (*k* ∈ *set* (*keys t*)) = (∃ *v*. (*k*, *v*) ∈ *set* (*entries t*))
⟨*proof*⟩

lemma *fold-def-alt*:
fold f t = *List.fold* (*case-prod f*) (*entries t*)
⟨*proof*⟩

lemma *distinct-entries*: *distinct* (*List.map fst* (*entries t*))
⟨*proof*⟩

lemma *non-empty-keys*: *t* ≠ *empty* ⟹ *keys t* ≠ []
⟨*proof*⟩

lemma *keys-def-alt*:
keys t = *List.map fst* (*entries t*)
⟨*proof*⟩

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

⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

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* ⇒ '*b* ⇒ ('*a*, '*b*) *RBT.rbt* ⇒ ('*a*, '*b*) *RBT.rbt*

Updates the map at a given position. $O(\log n)$

RBT.delete::'*a* ⇒ ('*a*, '*b*) *RBT.rbt* ⇒ ('*a*, '*b*) *RBT.rbt*

Deletes a map entry at a given position. $O(\log n)$

RBT.entries::('*a*, '*b*) *RBT.rbt* ⇒ ('*a* × '*b*) *list*

Return a corresponding key-value list for a tree.

RBT.bulkload::('*a* × '*b*) *list* ⇒ ('*a*, '*b*) *RBT.rbt*

Builds a tree from a key-value list.

RBT.map-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* $\mapsto$ *?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* ⟨*proof*⟩

lemma [*code*, *code del*]:
  Set.is-empty = *Set.is-empty* ⟨*proof*⟩

lemma [*code*, *code del*]:
  uminus-set-inst.uminus-set = *uminus-set-inst.uminus-set* ⟨*proof*⟩

lemma [*code*, *code del*]:
  Set.member = *Set.member* ⟨*proof*⟩

lemma [*code*, *code del*]:
  Set.insert = *Set.insert* ⟨*proof*⟩

lemma [*code*, *code del*]:
  Set.remove = *Set.remove* ⟨*proof*⟩

lemma [*code*, *code del*]:
  UNIV = *UNIV* ⟨*proof*⟩

lemma [*code*, *code del*]:
  Set.filter = *Set.filter* ⟨*proof*⟩

lemma [*code*, *code del*]:
  image = *image* ⟨*proof*⟩

lemma [*code*, *code del*]:
  Set.subset-eq = *Set.subset-eq* ⟨*proof*⟩

lemma [*code*, *code del*]:
  Ball = *Ball* ⟨*proof*⟩

lemma [*code*, *code del*]:
Bex = *Bex* ⟨*proof*⟩

lemma [*code*, *code del*]:
can-select = *can-select* ⟨*proof*⟩

lemma [*code*, *code del*]:
Set.union = *Set.union* ⟨*proof*⟩

lemma [*code*, *code del*]:
minus-set-inst.minus-set = *minus-set-inst.minus-set* ⟨*proof*⟩

lemma [*code*, *code del*]:
Set.inter = *Set.inter* ⟨*proof*⟩

lemma [*code*, *code del*]:
card = *card* ⟨*proof*⟩

lemma [*code*, *code del*]:
the-elem = *the-elem* ⟨*proof*⟩

lemma [*code*, *code del*]:
Pow = *Pow* ⟨*proof*⟩

lemma [*code*, *code del*]:
setsum = *setsum* ⟨*proof*⟩

lemma [*code*, *code del*]:
setprod = *setprod* ⟨*proof*⟩

lemma [*code*, *code del*]:
Product-Type.product = *Product-Type.product* ⟨*proof*⟩

lemma [*code*, *code del*]:
Id-on = *Id-on* ⟨*proof*⟩

lemma [*code*, *code del*]:
Image = *Image* ⟨*proof*⟩

lemma [*code*, *code del*]:
trancl = *trancl* ⟨*proof*⟩

lemma [*code*, *code del*]:
relcomp = *relcomp* ⟨*proof*⟩

lemma [*code*, *code del*]:
wf = *wf* ⟨*proof*⟩

lemma [*code*, *code del*]:
Min = *Min* ⟨*proof*⟩

lemma [*code*, *code del*]:
Inf-fin = *Inf-fin* ⟨*proof*⟩

lemma [*code*, *code del*]:
INFIMUM = *INFIMUM* ⟨*proof*⟩

lemma [*code*, *code del*]:
Max = *Max* ⟨*proof*⟩

lemma [*code*, *code del*]:
Sup-fin = *Sup-fin* ⟨*proof*⟩

lemma [*code*, *code del*]:
SUPREMUM = *SUPREMUM* ⟨*proof*⟩

lemma [*code*, *code del*]:
(*Inf* :: *'a set set* ⇒ *'a set*) = *Inf* ⟨*proof*⟩

lemma [*code*, *code del*]:
(*Sup* :: *'a set set* ⇒ *'a set*) = *Sup* ⟨*proof*⟩

lemma [*code*, *code del*]:
sorted-list-of-set = *sorted-list-of-set* ⟨*proof*⟩

lemma [*code*, *code del*]:
List.map-project = *List.map-project* ⟨*proof*⟩

lemma [*code*, *code del*]:
List.Bleast = *List.Bleast* ⟨*proof*⟩

114 Lemmas

114.1 Auxiliary lemmas

lemma [*simp*]: $x \neq \mathit{Some}\ () \longleftrightarrow x = \mathit{None}$
⟨*proof*⟩

lemma *Set-set-keys*: *Set* x = *dom* (*RBT.lookup* x)
⟨*proof*⟩

lemma *finite-Set* [*simp*, *intro*!]: *finite* (*Set* x)
⟨*proof*⟩

lemma *set-keys*: *Set* t = *set*(*RBT.keys* t)
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *finite-fold-fold-keys*:
assumes *comp-fun-commute f*
shows *Finite-Set.fold f A* (*Set t*) = *fold-keys f t A*
⟨*proof*⟩

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*
⟨*proof*⟩

114.3 foldi and Ball

lemma *Ball-False*: *RBT-Impl.fold* (λ*k v s*. *s* ∧ *P k*) *t False* = *False*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩ **including** *rbt.lifting* ⟨*proof*⟩

114.4 foldi and Bex

lemma *Bex-True*: *RBT-Impl.fold* (λ*k v s*. *s* ∨ *P k*) *t True* = *True*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩ **including** *rbt.lifting* ⟨*proof*⟩

114.5 folding over non empty trees and selecting the minimal and maximal element

definition *rbt-fold1-keys* :: ('*a* ⇒ '*a* ⇒ '*a*) ⇒ ('*a*::*linorder*, '*b*) *RBT-Impl.rbt* ⇒ '*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* ⇒ '*a*
where *rbt-min t* = *rbt-fold1-keys min t*

lemma *key-le-right*: *rbt-sorted* (*Branch c lt k v rt*) ⟹ (⋀*x*. *x* ∈*set* (*RBT-Impl.keys rt*) ⟹ *k* ≤ *x*)
⟨*proof*⟩

lemma *left-le-key*: *rbt-sorted* (*Branch c lt k v rt*) ⟹ (⋀*x*. *x* ∈*set* (*RBT-Impl.keys lt*) ⟹ *x* ≤ *k*)
⟨*proof*⟩

lemma *fold-min-triv*:
fixes *k* :: - :: *linorder*
shows (∀ *x*∈*set xs*. *k* ≤ *x*) ⟹ *List.fold min xs k* = *k*
⟨*proof*⟩

lemma *rbt-min-simps*:
is-rbt (*Branch c RBT-Impl.Empty k v rt*) ⟹ *rbt-min* (*Branch c RBT-Impl.Empty k v rt*) = *k*
⟨*proof*⟩

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 ≠ *rbt.Empty* ⟹ *rbt-min-opt* (*Branch c t1 k* () *t2*) = *rbt-min-opt t1*
⟨*proof*⟩

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 ⋀*color t1 a b t2*. *P t1* ⟹ *P t2* ⟹ *t1* = *rbt.Empty* ⟹ *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 $P\ t$
⟨*proof*⟩

lemma *rbt-min-opt-in-set*:
fixes $t :: ('a :: linorder,\ unit)\ RBT\text{-}Impl.rbt$
assumes $t \neq rbt.Empty$
shows $rbt\text{-}min\text{-}opt\ t \in set\ (RBT\text{-}Impl.keys\ t)$
⟨*proof*⟩

lemma *rbt-min-opt-is-min*:
fixes $t :: ('a :: linorder,\ unit)\ RBT\text{-}Impl.rbt$
assumes *rbt-sorted t*
assumes $t \neq rbt.Empty$
shows $\bigwedge y.\ y \in set\ (RBT\text{-}Impl.keys\ t) \Longrightarrow y \geq rbt\text{-}min\text{-}opt\ t$
⟨*proof*⟩

lemma *rbt-min-eq-rbt-min-opt*:
assumes $t \neq RBT\text{-}Impl.Empty$
assumes *is-rbt t*
shows $rbt\text{-}min\ t = rbt\text{-}min\text{-}opt\ t$
⟨*proof*⟩

definition $rbt\text{-}max :: ('a::linorder,\ unit)\ RBT\text{-}Impl.rbt \Rightarrow 'a$
where $rbt\text{-}max\ t = rbt\text{-}fold1\text{-}keys\ max\ t$

lemma *fold-max-triv*:
fixes $k :: - :: linorder$
shows $(\forall x{\in}set\ xs.\ x \leq k) \Longrightarrow List.fold\ max\ xs\ k = k$
⟨*proof*⟩

lemma *fold-max-rev-eq*:
fixes $xs :: ('a :: linorder)\ list$
assumes $xs \neq []$
shows $List.fold\ max\ (tl\ xs)\ (hd\ xs) = List.fold\ max\ (tl\ (rev\ xs))\ (hd\ (rev\ xs))$
⟨*proof*⟩

lemma *rbt-max-simps*:
assumes $is\text{-}rbt\ (Branch\ c\ lt\ k\ v\ RBT\text{-}Impl.Empty)$
shows $rbt\text{-}max\ (Branch\ c\ lt\ k\ v\ RBT\text{-}Impl.Empty) = k$
⟨*proof*⟩

fun *rbt-max-opt* **where**
$rbt\text{-}max\text{-}opt\ (Branch\ c\ lt\ k\ v\ RBT\text{-}Impl.Empty) = k\ |$
$rbt\text{-}max\text{-}opt\ (Branch\ c\ lt\ k\ v\ (Branch\ rc\ rlc\ rk\ rv\ rrt)) = rbt\text{-}max\text{-}opt\ (Branch\ rc\ rlc\ rk\ rv\ rrt)$

lemma *rbt-max-opt-Branch*:
t2 ≠ *rbt.Empty* ⟹ *rbt-max-opt* (*Branch c t1 k* () *t2*) = *rbt-max-opt t2*
⟨*proof*⟩

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 ⋀*color t1 a b t2*. *P t1* ⟹ *P t2* ⟹ *t2* = *rbt.Empty* ⟹ *P* (*Branch color t1 a b t2*)
assumes ⋀*color t1 a b t2*. *P t1* ⟹ *P t2* ⟹ *t2* ≠ *rbt.Empty* ⟹ *P* (*Branch color t1 a b t2*)
shows *P t*
⟨*proof*⟩

lemma *rbt-max-opt-in-set*:
fixes *t* :: ('*a* :: *linorder*, *unit*) *RBT-Impl.rbt*
assumes *t* ≠ *rbt.Empty*
shows *rbt-max-opt t* ∈ *set* (*RBT-Impl.keys t*)
⟨*proof*⟩

lemma *rbt-max-opt-is-max*:
fixes *t* :: ('*a* :: *linorder*, *unit*) *RBT-Impl.rbt*
assumes *rbt-sorted t*
assumes *t* ≠ *rbt.Empty*
shows ⋀*y*. *y* ∈ *set* (*RBT-Impl.keys t*) ⟹ *y* ≤ *rbt-max-opt t*
⟨*proof*⟩

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*⟩

context includes *rbt.lifting* **begin**
lift-definition *fold1-keys* :: ('*a* ⇒ '*a* ⇒ '*a*) ⇒ ('*a*::*linorder*, '*b*) *rbt* ⇒ '*a*
is *rbt-fold1-keys* ⟨*proof*⟩

lemma *fold1-keys-def-alt*:
fold1-keys f t = *List.fold f* (*tl* (*RBT.keys t*)) (*hd* (*RBT.keys t*))
⟨*proof*⟩

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*⟩

lift-definition *r-min* :: ('*a* :: *linorder*, *unit*) *rbt* ⇒ '*a* **is** *rbt-min* ⟨*proof*⟩

lift-definition *r-min-opt* :: ('*a* :: *linorder*, *unit*) *rbt* ⇒ '*a* **is** *rbt-min-opt* ⟨*proof*⟩

lemma *r-min-alt-def*: *r-min t* = *fold1-keys min t*
⟨*proof*⟩

lemma *r-min-eq-r-min-opt*:
assumes ¬ (*RBT.is-empty t*)
shows *r-min t* = *r-min-opt t*
⟨*proof*⟩

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*⟩

lift-definition *r-max* :: ('*a* :: *linorder*, *unit*) *rbt* ⇒ '*a* **is** *rbt-max* ⟨*proof*⟩

lift-definition *r-max-opt* :: ('*a* :: *linorder*, *unit*) *rbt* ⇒ '*a* **is** *rbt-max-opt* ⟨*proof*⟩

lemma *r-max-alt-def*: *r-max t* = *fold1-keys max t*
⟨*proof*⟩

lemma *r-max-eq-r-max-opt*:
assumes ¬ (*RBT.is-empty t*)
shows *r-max t* = *r-max-opt t*
⟨*proof*⟩

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*⟩

end

115 Code equations

code-datatype *Set Coset*

declare *list.set*[*code*]

lemma *empty-Set* [*code*]:
Set.empty = Set RBT.empty
⟨*proof*⟩

lemma *UNIV-Coset* [*code*]:
UNIV = Coset RBT.empty
⟨*proof*⟩

lemma *is-empty-Set* [*code*]:
Set.is-empty (Set t) = RBT.is-empty t
⟨*proof*⟩

lemma *compl-code* [*code*]:
− Set xs = Coset xs
− Coset xs = Set xs
⟨*proof*⟩

lemma *member-code* [*code*]:
x ∈ (Set t) = (RBT.lookup t x = Some ())
x ∈ (Coset t) = (RBT.lookup t x = None)
⟨*proof*⟩

lemma *insert-code* [*code*]:
Set.insert x (Set t) = Set (RBT.insert x () t)
Set.insert x (Coset t) = Coset (RBT.delete x t)
⟨*proof*⟩

lemma *remove-code* [*code*]:
Set.remove x (Set t) = Set (RBT.delete x t)
Set.remove x (Coset t) = Coset (RBT.insert x () t)
⟨*proof*⟩

lemma *union-Set* [*code*]:
Set t ∪ A = fold-keys Set.insert t A
⟨*proof*⟩

lemma *inter-Set* [*code*]:
A ∩ Set t = rbt-filter (λk. k ∈ A) t
⟨*proof*⟩

lemma *minus-Set* [*code*]:
A − Set t = fold-keys Set.remove t A
⟨*proof*⟩

lemma *union-Coset* [*code*]:
Coset t ∪ A = − rbt-filter (λk. k ∉ A) t
⟨*proof*⟩

lemma *union-Set-Set* [*code*]:
Set t1 ∪ *Set t2* = *Set* (*RBT.union t1 t2*)
⟨*proof*⟩

lemma *inter-Coset* [*code*]:
A ∩ *Coset t* = *fold-keys Set.remove t A*
⟨*proof*⟩

lemma *inter-Coset-Coset* [*code*]:
Coset t1 ∩ *Coset t2* = *Coset* (*RBT.union t1 t2*)
⟨*proof*⟩

lemma *minus-Coset* [*code*]:
A − *Coset t* = *rbt-filter* (λ*k*. *k* ∈ *A*) *t*
⟨*proof*⟩

lemma *filter-Set* [*code*]:
Set.filter P (*Set t*) = (*rbt-filter P t*)
⟨*proof*⟩

lemma *image-Set* [*code*]:
image f (*Set t*) = *fold-keys* (λ*k A*. *Set.insert* (*f k*) *A*) *t* {}
⟨*proof*⟩

lemma *Ball-Set* [*code*]:
Ball (*Set t*) *P* ⟷ *RBT.foldi* (λ*s*. *s* = *True*) (λ*k v s*. *s* ∧ *P k*) *t True*
⟨*proof*⟩

lemma *Bex-Set* [*code*]:
Bex (*Set t*) *P* ⟷ *RBT.foldi* (λ*s*. *s* = *False*) (λ*k v s*. *s* ∨ *P k*) *t False*
⟨*proof*⟩

lemma *subset-code* [*code*]:
Set t ≤ *B* ⟷ (∀ *x*∈*Set t*. *x* ∈ *B*)
A ≤ *Coset t* ⟷ (∀ *y*∈*Set t*. *y* ∉ *A*)
⟨*proof*⟩

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* ≤ *Set t2*))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *card-Set* [*code*]:

card (Set t) = fold-keys (λ- n. n + 1) t 0
⟨*proof*⟩

lemma *setsum-Set* [*code*]:
setsum f (Set xs) = fold-keys (plus o f) xs 0
⟨*proof*⟩

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*⟩

lemma *Pow-Set* [*code*]: *Pow (Set t) = fold-keys (λx A. A ∪ Set.insert x ' A) t {{}}*
⟨*proof*⟩

lemma *product-Set* [*code*]:
Product-Type.product (Set t1) (Set t2) =
fold-keys (λx A. fold-keys (λy. Set.insert (x, y)) t2 A) t1 {}
⟨*proof*⟩

lemma *Id-on-Set* [*code*]: *Id-on (Set t) = fold-keys (λx. Set.insert (x, x)) t {}*
⟨*proof*⟩

lemma *Image-Set* [*code*]:
(Set t) `` S = fold-keys (λ(x,y) A. if x ∈ S then Set.insert y A else A) t {}
⟨*proof*⟩

lemma *trancl-set-ntrancl* [*code*]:
trancl (Set t) = ntrancl (card (Set t) − 1) (Set t)
⟨*proof*⟩

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*⟩

lemma *wf-set* [*code*]:
wf (Set t) = acyclic (Set t)
⟨*proof*⟩

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*⟩

lemma *Inf-fin-set-fold* [*code*]:
Inf-fin (*Set t*) = *Min* (*Set t*)
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

lemma *Sup-fin-set-fold* [*code*]:
Sup-fin (*Set t*) = *Max* (*Set t*)
⟨*proof*⟩

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*⟩

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*⟩

lemma *sorted-list-set*[*code*]: *sorted-list-of-set* (*Set t*) = *RBT.keys t*
⟨*proof*⟩

lemma *Bleast-code* [*code*]:
  Bleast (*Set t*) *P* =
    (*case filter P* (*RBT.keys t*) *of*
      x # *xs* ⇒ *x*
    | [] ⇒ *abort-Bleast* (*Set t*) *P*)
⟨*proof*⟩

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*⟩

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*: $ALL\ R.\ wf\ R\ -->$
$(ALL\ P.\ (ALL\ x.\ (ALL\ y.\ (y,x):R\ -->\ P\ y)\ -->\ P\ x)\ -->\ (ALL\ x.\ P\ x))$
⟨*proof*⟩

lemma *tfl-cut-def*: $cut\ f\ r\ x \equiv (\lambda y.\ if\ (y,x) \in r\ then\ f\ y\ else\ undefined)$
⟨*proof*⟩

lemma *tfl-cut-apply*: $ALL\ f\ R.\ (x,a):R\ -->\ (cut\ f\ R\ a)(x) = f(x)$
⟨*proof*⟩

lemma *tfl-wfrec*:
$ALL\ M\ R\ f.\ (f=wfrec\ R\ M)\ -->\ wf\ R\ -->\ (ALL\ x.\ f\ x = M\ (cut\ f\ R\ x)\ x)$
⟨*proof*⟩

lemma *tfl-eq-True*: $(x = True)\ -->\ x$
⟨*proof*⟩

lemma *tfl-rev-eq-mp*: $(x = y)\ -->\ y\ -->\ x$
⟨*proof*⟩

lemma *tfl-simp-thm*: $(x\ -->\ y)\ -->\ (x = x')\ -->\ (x'\ -->\ y)$
⟨*proof*⟩

lemma *tfl-P-imp-P-iff-True*: $P \Longrightarrow P = True$
⟨*proof*⟩

lemma *tfl-imp-trans*: $(A \longrightarrow B) \Longrightarrow (B \longrightarrow C) \Longrightarrow (A \longrightarrow C)$
⟨*proof*⟩

lemma *tfl-disj-assoc*: $(a \vee b) \vee c \equiv a \vee (b \vee c)$
⟨*proof*⟩

lemma *tfl-disjE*: $P \vee Q \Longrightarrow P \longrightarrow R \Longrightarrow Q \longrightarrow R \Longrightarrow R$
⟨*proof*⟩

lemma *tfl-exE*: $\exists x.\ P\ x \Longrightarrow \forall x.\ P\ x \longrightarrow Q \Longrightarrow Q$
⟨*proof*⟩

⟨*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* ⇒ *'a* (*NOT - [70] 71*)
    and *bitAND* :: *'a* ⇒ *'a* ⇒ *'a* (**infixr** *AND 64*)
    and *bitOR* :: *'a* ⇒ *'a* ⇒ *'a* (**infixr** *OR 59*)
    and *bitXOR* :: *'a* ⇒ *'a* ⇒ *'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* ⇒ *nat* ⇒ *bool* (**infixl** !! *100*)
    and *lsb* :: *'a* ⇒ *bool*
    and *set-bit* :: *'a* ⇒ *nat* ⇒ *bool* ⇒ *'a*
    and *set-bits* :: (*nat* ⇒ *bool*) ⇒ *'a* (**binder** *BITS 10*)
    and *shiftl* :: *'a* ⇒ *nat* ⇒ *'a* (**infixl** << *55*)
    and *shiftr* :: *'a* ⇒ *nat* ⇒ *'a* (**infixl** >> *55*)

class *bitss* = *bits* +
  fixes *msb* :: *'a* ⇒ *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 ⟨*proof*⟩

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)
⟨*proof*⟩

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
⟨*proof*⟩

lemma *bit-ops-same* [*simp*]:
(x::bit) AND x = x
(x::bit) OR x = x
(x::bit) XOR x = 0
⟨*proof*⟩

lemma *bit-not-not* [*simp*]: *NOT (NOT (x::bit)) = x*
⟨*proof*⟩

lemma *bit-or-def*: *(b::bit) OR c = NOT (NOT b AND NOT c)*
⟨*proof*⟩

lemma *bit-xor-def*: *(b::bit) XOR c = (b AND NOT c) OR (NOT b AND c)*
⟨*proof*⟩

lemma *bit-NOT-eq-1-iff* [*simp*]: *NOT (b::bit) = 1 ⟷ b = 0*
⟨*proof*⟩

lemma *bit-AND-eq-1-iff* [*simp*]: *(a::bit) AND b = 1 ⟷ a = 1 ∧ b = 1*
⟨*proof*⟩

end

120 Useful Numerical Lemmas

theory *Misc-Numeric*
imports *Main*

begin

lemma *mod-2-neq-1-eq-eq-0*:
fixes k :: *int*
shows $k \ mod \ 2 \neq 1 \longleftrightarrow k \ mod \ 2 = 0$
⟨*proof*⟩

lemma *z1pmod2*:
fixes b :: *int*
shows $(2 * b + 1) \ mod \ 2 = (1::int)$
⟨*proof*⟩

lemma *diff-le-eq'*:
$a - b \leq c \longleftrightarrow a \leq b + (c::int)$
⟨*proof*⟩

lemma *emep1*:
fixes $n \ d$ :: *int*
shows $even \ n \Longrightarrow even \ d \Longrightarrow 0 \leq d \Longrightarrow (n + 1) \ mod \ d = (n \ mod \ d) + 1$
⟨*proof*⟩

lemma *int-mod-ge*:
$a < n \Longrightarrow 0 < (n :: int) \Longrightarrow a \leq a \ mod \ n$
⟨*proof*⟩

lemma *int-mod-ge'*:
$b < 0 \Longrightarrow 0 < (n :: int) \Longrightarrow b + n \leq b \ mod \ n$
⟨*proof*⟩

lemma *int-mod-le'*:
$(0::int) \leq b - n \Longrightarrow b \ mod \ n \leq b - n$
⟨*proof*⟩

lemma *zless2*:
$0 < (2 :: int)$
⟨*proof*⟩

lemma *zless2p*:
$0 < (2 \ \hat{} \ n :: int)$
⟨*proof*⟩

lemma *zle2p*:
$0 \leq (2 \ \hat{} \ n :: int)$
⟨*proof*⟩

lemma *m1mod2k*:
$-1 \ mod \ 2 \ \hat{} \ n = (2 \ \hat{} \ n - 1 :: int)$
⟨*proof*⟩

lemma *p1mod22k'*:
fixes *b* :: *int*
shows (*1* + *2* * *b*) *mod* (*2* * *2* ^ *n*) = *1* + *2* * (*b mod* *2* ^ *n*)
⟨*proof*⟩

lemma *p1mod22k*:
fixes *b* :: *int*
shows (*2* * *b* + *1*) *mod* (*2* * *2* ^ *n*) = *2* * (*b mod* *2* ^ *n*) + *1*
⟨*proof*⟩

lemma *int-mod-lem*:
(*0* :: *int*) < *n* ==> (*0* <= *b* & *b* < *n*) = (*b mod* *n* = *b*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *Bit-B1*:
k BIT True = *k* + *k* + *1*
⟨*proof*⟩

lemma *Bit-B0-2t*: *k BIT False* = *2* * *k*
⟨*proof*⟩

lemma *Bit-B1-2t*: *k BIT True* = *2* * *k* + *1*
⟨*proof*⟩

definition *bin-last* :: *int* ⇒ *bool*
where
bin-last w ⟷ *w mod* *2* = *1*

lemma *bin-last-odd*:
bin-last = *odd*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *bin-rest-BIT* [*simp*]: *bin-rest* (*x BIT b*) = *x*
⟨*proof*⟩

lemma *bin-last-BIT* [*simp*]: *bin-last* (*x BIT b*) = *b*
⟨*proof*⟩

lemma *BIT-eq-iff* [*iff*]: *u BIT b* = *v BIT c* ⟷ *u* = *v* ∧ *b* = *c*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *Bit-eq-0-iff*: *w BIT b* = *0* ⟷ *w* = *0* ∧ ¬ *b*
⟨*proof*⟩

lemma *Bit-eq-m1-iff*: *w BIT b* = −*1* ⟷ *w* = −*1* ∧ *b*
⟨*proof*⟩

lemma *BitM-inc*: *Num.BitM* (*Num.inc w*) = *Num.Bit1 w*
⟨*proof*⟩

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*
⟨*proof*⟩

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))
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *less-Bits*:
v BIT b < w BIT c ⟷ v < w ∨ v ≤ w ∧ ¬ b ∧ c
⟨*proof*⟩

lemma *le-Bits*:
v BIT b ≤ w BIT c ⟷ v < w ∨ v ≤ w ∧ (¬ b ∨ c)
⟨*proof*⟩

lemma *pred-BIT-simps* [*simp*]:
x BIT False − 1 = (x − 1) BIT True
x BIT True − 1 = x BIT False
⟨*proof*⟩

lemma *succ-BIT-simps* [*simp*]:
x BIT False + 1 = x BIT True
x BIT True + 1 = (x + 1) BIT False
⟨*proof*⟩

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
⟨*proof*⟩

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*

⟨*proof*⟩

lemma *B-mod-2'*:
X = 2 ==> (w BIT True) mod X = 1 & (w BIT False) mod X = 0
⟨*proof*⟩

lemma *bin-ex-rl*: *EX w b. w BIT b = bin*
⟨*proof*⟩

lemma *bin-exhaust*:
assumes *Q*: *⋀x b. bin = x BIT b ⟹ Q*
shows *Q*
⟨*proof*⟩

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|
⟨*proof*⟩

lemma *bin-induct*:
assumes *PPls*: *P 0*
and *PMin*: *P (- 1)*
and *PBit*: *!!bin bit. P bin ==> P (bin BIT bit)*
shows *P bin*
⟨*proof*⟩

lemma *Bit-div2* [*simp*]: *(w BIT b) div 2 = w*
⟨*proof*⟩

lemma *bin-nth-eq-iff*:
bin-nth x = bin-nth y ⟷ x = y
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *bin-nth-zero* [*simp*]: *¬ bin-nth 0 n*
⟨*proof*⟩

lemma *bin-nth-1* [*simp*]: *bin-nth 1 n ⟷ n = 0*
⟨*proof*⟩

lemma *bin-nth-minus1* [*simp*]: *bin-nth* (− *1*) *n*
⟨*proof*⟩

lemma *bin-nth-0-BIT*: *bin-nth* (*w BIT b*) *0* ⟷ *b*
⟨*proof*⟩

lemma *bin-nth-Suc-BIT*: *bin-nth* (*w BIT b*) (*Suc n*) = *bin-nth w n*
⟨*proof*⟩

lemma *bin-nth-minus* [*simp*]: *0* < *n* ==> *bin-nth* (*w BIT b*) *n* = *bin-nth w* (*n* − *1*)
⟨*proof*⟩

lemma *bin-nth-numeral*:
bin-rest x = *y* ⟹ *bin-nth x* (*numeral n*) = *bin-nth y* (*pred-numeral n*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *bin-sign-rest* [*simp*]:
bin-sign (*bin-rest w*) = *bin-sign w*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *bintrunc-mod2p*: *bintrunc n w* = (*w mod 2* ^ *n*)
⟨*proof*⟩

lemma *sbintrunc-mod2p*: *sbintrunc n w* = (*w* + *2* ^ *n*) *mod 2* ^ (*Suc n*) − *2* ^ *n*
⟨*proof*⟩

121.3 Simplifications for (s)bintrunc

lemma *bintrunc-n-0* [*simp*]: *bintrunc n 0* = *0*
⟨*proof*⟩

lemma *sbintrunc-n-0* [*simp*]: *sbintrunc n 0* = *0*
⟨*proof*⟩

lemma *sbintrunc-n-minus1* [*simp*]: *sbintrunc n* (− *1*) = *−1*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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
⟨*proof*⟩

lemma *bin-sign-lem*: *(bin-sign (sbintrunc n bin) = −1) = bin-nth bin n*
⟨*proof*⟩

lemma *nth-bintr*: *bin-nth (bintrunc m w) n = (n < m & bin-nth w n)*
⟨*proof*⟩

lemma *nth-sbintr*:
bin-nth (sbintrunc m w) n =
(if n < m then bin-nth w n else bin-nth w m)
⟨*proof*⟩

lemma *bin-nth-Bit*:
bin-nth (w BIT b) n = (n = 0 & b | (EX m. n = Suc m & bin-nth w m))
⟨*proof*⟩

lemma *bin-nth-Bit0*:
bin-nth (numeral (Num.Bit0 w)) n ⟷
(∃ m. n = Suc m ∧ bin-nth (numeral w) m)
⟨*proof*⟩

lemma *bin-nth-Bit1*:
bin-nth (numeral (Num.Bit1 w)) n ⟷
n = 0 ∨ (∃ m. n = Suc m ∧ bin-nth (numeral w) m)
⟨*proof*⟩

lemma *bintrunc-bintrunc-l*:
n <= m ==> (bintrunc m (bintrunc n w) = bintrunc n w)
⟨*proof*⟩

lemma *sbintrunc-sbintrunc-l*:
n <= m ==> (sbintrunc m (sbintrunc n w) = sbintrunc n w)
⟨*proof*⟩

lemma *bintrunc-bintrunc-ge*:
n <= m ==> (bintrunc n (bintrunc m w) = bintrunc n w)
⟨*proof*⟩

lemma *bintrunc-bintrunc-min* [*simp*]:
bintrunc m (bintrunc n w) = bintrunc (min m n) w
⟨*proof*⟩

lemma *sbintrunc-sbintrunc-min* [*simp*]:
sbintrunc m (*sbintrunc n w*) = *sbintrunc* (*min m n*) *w*
⟨*proof*⟩

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
⟨*proof*⟩

lemma *sbintrunc-minus*:
0 < n ==> sbintrunc (Suc (n − 1)) w = sbintrunc n w
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

lemmas *sbintrunc-Suc-Ialts =*
sbintrunc-Suc-Is [*THEN sbintrunc-Suc-lem*]

lemma *sbintrunc-bintrunc-lt*:

$m > n \Longrightarrow$ *sbintrunc* n (*bintrunc* m w) = *sbintrunc* n w
⟨*proof*⟩

lemma *bintrunc-sbintrunc-le*:
$m <= Suc\ n \Longrightarrow$ *bintrunc* m (*sbintrunc* n w) = *bintrunc* m w
⟨*proof*⟩

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*]:
$0 < n \Longrightarrow$ *bintrunc* n (*sbintrunc* $(n - 1)$ w) = *bintrunc* n w
⟨*proof*⟩

lemma *sbintrunc-bintrunc'* [*simp*]:
$0 < n \Longrightarrow$ *sbintrunc* $(n - 1)$ (*bintrunc* n w) = *sbintrunc* $(n - 1)$ w
⟨*proof*⟩

lemma *bin-sbin-eq-iff*:
bintrunc (*Suc* n) x = *bintrunc* (*Suc* n) $y \longleftrightarrow$
sbintrunc n x = *sbintrunc* n y
⟨*proof*⟩

lemma *bin-sbin-eq-iff'*:
$0 < n \Longrightarrow$ *bintrunc* n x = *bintrunc* n $y \longleftrightarrow$
sbintrunc $(n - 1)$ x = *sbintrunc* $(n - 1)$ y
⟨*proof*⟩

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
⟨*proof*⟩

lemma *sbintrunc-numeral*:
sbintrunc (*numeral* k) x =
sbintrunc (*pred-numeral* k) (*bin-rest* x) *BIT bin-last* x

⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *no-bintr-alt1*: *bintrunc n* = (λ*w*. *w mod 2* ^ *n* :: *int*)
⟨*proof*⟩

lemma *range-bintrunc*: *range* (*bintrunc n*) = {*i*. *0* <= *i* & *i* < *2* ^ *n*}
⟨*proof*⟩

lemma *no-sbintr-alt2*:
sbintrunc n = (%*w*. (*w* + *2* ^ *n*) *mod 2* ^ *Suc n* − *2* ^ *n* :: *int*)
⟨*proof*⟩

lemma *range-sbintrunc*:
range (*sbintrunc n*) = {*i*. − (*2* ^ *n*) <= *i* & *i* < *2* ^ *n*}
⟨*proof*⟩

lemma *sb-inc-lem*:
(*a*::*int*) + *2*^*k* < *0* ⟹ *a* + *2*^*k* + *2*^(*Suc k*) <= (*a* + *2*^*k*) *mod 2*^(*Suc k*)
⟨*proof*⟩

lemma *sb-inc-lem'*:
(*a*::*int*) < − (*2*^*k*) ⟹ *a* + *2*^*k* + *2*^(*Suc k*) <= (*a* + *2*^*k*) *mod 2*^(*Suc k*)
⟨*proof*⟩

lemma *sbintrunc-inc*:

$x < - (2\hat{}n) ==> x + 2\hat{}(Suc\ n) <= sbintrunc\ n\ x$
⟨*proof*⟩

lemma *sb-dec-lem*:
$(0::int) \leq - (2\ \hat{}\ k) + a \Longrightarrow (a + 2\ \hat{}\ k)\ mod\ (2 * 2\ \hat{}\ k) \leq - (2\ \hat{}\ k) + a$
⟨*proof*⟩

lemma *sb-dec-lem'*:
$(2::int)\ \hat{}\ k \leq a \Longrightarrow (a + 2\ \hat{}\ k)\ mod\ (2 * 2\ \hat{}\ k) \leq - (2\ \hat{}\ k) + a$
⟨*proof*⟩

lemma *sbintrunc-dec*:
$x >= (2\ \hat{}\ n) ==> x - 2\ \hat{}\ (Suc\ n) >= sbintrunc\ n\ x$
⟨*proof*⟩

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*^*n*::*int*, *folded bintrunc-mod2p*] **for** *n*
lemmas *bintr-ariths* =
brdmods' [**where** *c*=*2*^*n*::*int*, *folded bintrunc-mod2p*] **for** *n*

lemmas *m2pths* = *pos-mod-sign pos-mod-bound* [*OF zless2p*]

lemma *bintr-ge0*: $0 \leq bintrunc\ n\ w$
⟨*proof*⟩

lemma *bintr-lt2p*: $bintrunc\ n\ w < 2\ \hat{}\ n$
⟨*proof*⟩

lemma *bintr-Min*: $bintrunc\ n\ (-1) = 2\ \hat{}\ n - 1$
⟨*proof*⟩

lemma *sbintr-ge*: $- (2\ \hat{}\ n) \leq sbintrunc\ n\ w$

⟨*proof*⟩

lemma *sbintr-lt*: *sbintrunc n w* < *2* ^ *n*
⟨*proof*⟩

lemma *sign-Pls-ge-0*:
(*bin-sign bin* = *0*) = (*bin* >= (*0* :: *int*))
⟨*proof*⟩

lemma *sign-Min-lt-0*:
(*bin-sign bin* = −*1*) = (*bin* < (*0* :: *int*))
⟨*proof*⟩

lemma *bin-rest-trunc*:
(*bin-rest* (*bintrunc n bin*)) = *bintrunc* (*n* − *1*) (*bin-rest bin*)
⟨*proof*⟩

lemma *bin-rest-power-trunc*:
(*bin-rest* ^^ *k*) (*bintrunc n bin*) =
bintrunc (*n* − *k*) ((*bin-rest* ^^ *k*) *bin*)
⟨*proof*⟩

lemma *bin-rest-trunc-i*:
bintrunc n (*bin-rest bin*) = *bin-rest* (*bintrunc* (*Suc n*) *bin*)
⟨*proof*⟩

lemma *bin-rest-strunc*:
bin-rest (*sbintrunc* (*Suc n*) *bin*) = *sbintrunc n* (*bin-rest bin*)
⟨*proof*⟩

lemma *bintrunc-rest* [*simp*]:
bintrunc n (*bin-rest* (*bintrunc n bin*)) = *bin-rest* (*bintrunc n bin*)
⟨*proof*⟩

lemma *sbintrunc-rest* [*simp*]:
sbintrunc n (*bin-rest* (*sbintrunc n bin*)) = *bin-rest* (*sbintrunc n bin*)
⟨*proof*⟩

lemma *bintrunc-rest'*:
bintrunc n o bin-rest o bintrunc n = *bin-rest o bintrunc n*
⟨*proof*⟩

lemma *sbintrunc-rest'* :
sbintrunc n o bin-rest o sbintrunc n = *bin-rest o sbintrunc n*
⟨*proof*⟩

lemma *rco-lem*:
f o g o f = *g o f* ==> *f o* (*g o f*) ^^ *n* = *g* ^^ *n o f*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*))
⟨*proof*⟩

termination
⟨*proof*⟩

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 ⟨*proof*⟩

end

122.1.1 Basic simplification rules

lemma *int-not-BIT* [*simp*]:
NOT (*w BIT b*) = (*NOT w*) *BIT* (¬ *b*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *int-not-not* [*simp*]: *NOT* (*NOT* (*x*::*int*)) = *x*
⟨*proof*⟩

lemma *int-and-0* [*simp*]: (*0*::*int*) *AND x* = *0*
⟨*proof*⟩

lemma *int-and-m1* [*simp*]: (*−1*::*int*) *AND x* = *x*
⟨*proof*⟩

lemma *int-and-Bits* [*simp*]:
(*x BIT b*) *AND* (*y BIT c*) = (*x AND y*) *BIT* (*b* ∧ *c*)
⟨*proof*⟩

lemma *int-or-zero* [*simp*]: (*0*::*int*) *OR x* = *x*
⟨*proof*⟩

lemma *int-or-minus1* [*simp*]: (*−1*::*int*) *OR x* = *−1*
⟨*proof*⟩

lemma *int-or-Bits* [*simp*]:
(*x BIT b*) *OR* (*y BIT c*) = (*x OR y*) *BIT* (*b* ∨ *c*)

⟨*proof*⟩

lemma *int-xor-zero* [*simp*]: (*0*::*int*) *XOR x* = *x*
⟨*proof*⟩

lemma *int-xor-Bits* [*simp*]:
(*x BIT b*) *XOR* (*y BIT c*) = (*x XOR y*) *BIT* ((*b* ∨ *c*) ∧ ¬ (*b* ∧ *c*))
⟨*proof*⟩

122.1.2 Binary destructors

lemma *bin-rest-NOT* [*simp*]: *bin-rest* (*NOT x*) = *NOT* (*bin-rest x*)
⟨*proof*⟩

lemma *bin-last-NOT* [*simp*]: *bin-last* (*NOT x*) ⟷ ¬ *bin-last x*
⟨*proof*⟩

lemma *bin-rest-AND* [*simp*]: *bin-rest* (*x AND y*) = *bin-rest x AND bin-rest y*
⟨*proof*⟩

lemma *bin-last-AND* [*simp*]: *bin-last* (*x AND y*) ⟷ *bin-last x* ∧ *bin-last y*
⟨*proof*⟩

lemma *bin-rest-OR* [*simp*]: *bin-rest* (*x OR y*) = *bin-rest x OR bin-rest y*
⟨*proof*⟩

lemma *bin-last-OR* [*simp*]: *bin-last* (*x OR y*) ⟷ *bin-last x* ∨ *bin-last y*
⟨*proof*⟩

lemma *bin-rest-XOR* [*simp*]: *bin-rest* (*x XOR y*) = *bin-rest x XOR bin-rest y*
⟨*proof*⟩

lemma *bin-last-XOR* [*simp*]: *bin-last* (*x XOR y*) ⟷ (*bin-last x* ∨ *bin-last y*) ∧
¬ (*bin-last x* ∧ *bin-last y*)
⟨*proof*⟩

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*)
⟨*proof*⟩

122.1.3 Derived properties

lemma *int-xor-minus1* [*simp*]: (−*1*::*int*) *XOR x* = *NOT x*
⟨*proof*⟩

lemma *int-xor-extra-simps* [*simp*]:
w XOR (*0*::*int*) = *w*

$w\ XOR\ (-1::int) = NOT\ w$
⟨*proof*⟩

lemma *int-or-extra-simps* [*simp*]:
$w\ OR\ (0::int) = w$
$w\ OR\ (-1::int) = -1$
⟨*proof*⟩

lemma *int-and-extra-simps* [*simp*]:
$w\ AND\ (0::int) = 0$
$w\ AND\ (-1::int) = w$
⟨*proof*⟩

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: $!!y::int.\ x\ XOR\ y = y\ XOR\ x$
⟨*proof*⟩

lemma *bin-ops-same* [*simp*]:
$(x::int)\ AND\ x = x$
$(x::int)\ OR\ x = x$
$(x::int)\ XOR\ x = 0$
⟨*proof*⟩

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*:
$!!y::int.\ x\ AND\ (y\ OR\ x) = x\ \&\ x\ OR\ (y\ AND\ x) = x$
⟨*proof*⟩

lemma *bbw-ao-absorbs-other*:
$x\ AND\ (x\ OR\ y) = x \wedge (y\ AND\ x)\ OR\ x = (x::int)$
$(y\ OR\ x)\ AND\ x = x \wedge x\ OR\ (x\ AND\ y) = (x::int)$
$(x\ OR\ y)\ AND\ x = x \wedge (x\ AND\ y)\ OR\ x = (x::int)$
⟨*proof*⟩

lemmas *bbw-ao-absorbs* [*simp*] = *bbw-ao-absorb* *bbw-ao-absorbs-other*

lemma *int-xor-not*:
$!!y::int.\ (NOT\ x)\ XOR\ y = NOT\ (x\ XOR\ y)\ \&$
$x\ XOR\ (NOT\ y) = NOT\ (x\ XOR\ y)$
⟨*proof*⟩

lemma *int-and-assoc*:
$(x\ AND\ y)\ AND\ (z::int) = x\ AND\ (y\ AND\ z)$
⟨*proof*⟩

lemma *int-or-assoc*:
$(x\ OR\ y)\ OR\ (z::int) = x\ OR\ (y\ OR\ z)$
⟨*proof*⟩

lemma *int-xor-assoc*:
$(x\ XOR\ y)\ XOR\ (z::int) = x\ XOR\ (y\ XOR\ z)$
⟨*proof*⟩

lemmas *bbw-assocs* = *int-and-assoc int-or-assoc int-xor-assoc*

lemma *bbw-lcs* [*simp*]:
$(y::int)\ AND\ (x\ AND\ z) = x\ AND\ (y\ AND\ z)$
$(y::int)\ OR\ (x\ OR\ z) = x\ OR\ (y\ OR\ z)$
$(y::int)\ XOR\ (x\ XOR\ z) = x\ XOR\ (y\ XOR\ z)$
⟨*proof*⟩

lemma *bbw-not-dist*:
$!!y::int.\ NOT\ (x\ OR\ y) = (NOT\ x)\ AND\ (NOT\ y)$
$!!y::int.\ NOT\ (x\ AND\ y) = (NOT\ x)\ OR\ (NOT\ y)$
⟨*proof*⟩

lemma *bbw-oa-dist*:
$!!y\ z::int.\ (x\ AND\ y)\ OR\ z =$
$(x\ OR\ z)\ AND\ (y\ OR\ z)$
⟨*proof*⟩

lemma *bbw-ao-dist*:
$!!y\ z::int.\ (x\ OR\ y)\ AND\ z =$
$(x\ AND\ z)\ OR\ (y\ AND\ z)$
⟨*proof*⟩

122.1.4 Simplification with numerals

Cases for *0* and -1 are already covered by other simp rules.

lemma *bin-rl-eqI*: $[\![bin\text{-}rest\ x = bin\text{-}rest\ y;\ bin\text{-}last\ x = bin\text{-}last\ y]\!] \Longrightarrow x = y$
⟨*proof*⟩

lemma *bin-rest-neg-numeral-BitM* [*simp*]:
$bin\text{-}rest\ (-\ numeral\ (Num.BitM\ w)) = -\ numeral\ w$
⟨*proof*⟩

lemma *bin-last-neg-numeral-BitM* [*simp*]:
$bin\text{-}last\ (-\ numeral\ (Num.BitM\ w))$

⟨*proof*⟩

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
⟨*proof*⟩

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)
⟨*proof*⟩

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*))
⟨*proof*⟩

122.1.5 Interactions with arithmetic

lemma *plus-and-or* [*rule-format*]:
ALL y::*int*. (*x AND y*) + (*x OR y*) = *x* + *y*
⟨*proof*⟩

lemma *le-int-or*:
bin-sign (*y*::*int*) = *0* ==> *x* <= *x OR y*
⟨*proof*⟩

lemmas *int-and-le* =
xtrans(*3*) [*OF bbw-ao-absorbs* (*2*) [*THEN conjunct2*, *symmetric*] *le-int-or*]

lemma *bin-add-not*: $x + NOT\ x = (-1::int)$
⟨*proof*⟩

122.1.6 Truncating results of bit-wise operations

lemma *bin-trunc-ao*:
$!!x\ y.\ (bintrunc\ n\ x)\ AND\ (bintrunc\ n\ y) = bintrunc\ n\ (x\ AND\ y)$
$!!x\ y.\ (bintrunc\ n\ x)\ OR\ (bintrunc\ n\ y) = bintrunc\ n\ (x\ OR\ y)$
⟨*proof*⟩

lemma *bin-trunc-xor*:
$!!x\ y.\ bintrunc\ n\ (bintrunc\ n\ x\ XOR\ bintrunc\ n\ y) =$
$bintrunc\ n\ (x\ XOR\ y)$
⟨*proof*⟩

lemma *bin-trunc-not*:
$!!x.\ bintrunc\ n\ (NOT\ (bintrunc\ n\ x)) = bintrunc\ n\ (NOT\ x)$
⟨*proof*⟩

lemma *bintr-bintr-i*:
$x = bintrunc\ n\ y \Longrightarrow bintrunc\ n\ x = bintrunc\ n\ y$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *bin-sc-sc-same* [*simp*]:
bin-sc n c (*bin-sc n b w*) = *bin-sc n c w*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *bin-nth-sc-gen*:

bin-nth (*bin-sc n b w*) *m* = (*if m* = *n then b else bin-nth w m*)
⟨*proof*⟩

lemma *bin-sc-nth* [*simp*]:
(*bin-sc n* (*bin-nth w n*) *w*) = *w*
⟨*proof*⟩

lemma *bin-sign-sc* [*simp*]:
bin-sign (*bin-sc n b w*) = *bin-sign w*
⟨*proof*⟩

lemma *bin-sc-bintr* [*simp*]:
bintrunc m (*bin-sc n x* (*bintrunc m* (*w*))) = *bintrunc m* (*bin-sc n x w*)
⟨*proof*⟩

lemma *bin-clr-le*:
bin-sc n False w <= *w*
⟨*proof*⟩

lemma *bin-set-ge*:
bin-sc n True w >= *w*
⟨*proof*⟩

lemma *bintr-bin-clr-le*:
bintrunc n (*bin-sc m False w*) <= *bintrunc n w*
⟨*proof*⟩

lemma *bintr-bin-set-ge*:
bintrunc n (*bin-sc m True w*) >= *bintrunc n w*
⟨*proof*⟩

lemma *bin-sc-FP* [*simp*]: *bin-sc n False 0* = *0*
⟨*proof*⟩

lemma *bin-sc-TM* [*simp*]: *bin-sc n True* (− *1*) = − *1*
⟨*proof*⟩

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*
⟨*proof*⟩

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*

⟨*proof*⟩

122.3 Splitting and concatenation

definition *bin-rcat* :: *nat* ⇒ *int list* ⇒ *int*
where
bin-rcat n = *foldl* (λ*u 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*
⟨*proof*⟩

lemma *bin-cat-Suc-Bit*:
bin-cat w (*Suc n*) (*v BIT b*) = *bin-cat w n v BIT b*
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *bin-nth-split*:
bin-split n c = (*a*, *b*) ==>
(*ALL k*. *bin-nth a k* = *bin-nth c* (*n* + *k*)) &

$(ALL\ k.\ bin\text{-}nth\ b\ k = (k < n\ \&\ bin\text{-}nth\ c\ k))$
⟨*proof*⟩

lemma *bin-cat-assoc*:
$bin\text{-}cat\ (bin\text{-}cat\ x\ m\ y)\ n\ z = bin\text{-}cat\ x\ (m + n)\ (bin\text{-}cat\ y\ n\ z)$
⟨*proof*⟩

lemma *bin-cat-assoc-sym*:
$bin\text{-}cat\ x\ m\ (bin\text{-}cat\ y\ n\ z) = bin\text{-}cat\ (bin\text{-}cat\ x\ (m - n)\ y)\ (min\ m\ n)\ z$
⟨*proof*⟩

lemma *bin-cat-zero* [*simp*]: $bin\text{-}cat\ 0\ n\ w = bintrunc\ n\ w$
⟨*proof*⟩

lemma *bintr-cat1*:
$bintrunc\ (k + n)\ (bin\text{-}cat\ a\ n\ b) = bin\text{-}cat\ (bintrunc\ k\ a)\ n\ b$
⟨*proof*⟩

lemma *bintr-cat*: $bintrunc\ m\ (bin\text{-}cat\ a\ n\ b) =$
$bin\text{-}cat\ (bintrunc\ (m - n)\ a)\ n\ (bintrunc\ (min\ m\ n)\ b)$
⟨*proof*⟩

lemma *bintr-cat-same* [*simp*]:
$bintrunc\ n\ (bin\text{-}cat\ a\ n\ b) = bintrunc\ n\ b$
⟨*proof*⟩

lemma *cat-bintr* [*simp*]:
$bin\text{-}cat\ a\ n\ (bintrunc\ n\ b) = bin\text{-}cat\ a\ n\ b$
⟨*proof*⟩

lemma *split-bintrunc*:
$bin\text{-}split\ n\ c = (a,\ b) ==> b = bintrunc\ n\ c$
⟨*proof*⟩

lemma *bin-cat-split*:
$bin\text{-}split\ n\ w = (u,\ v) ==> w = bin\text{-}cat\ u\ n\ v$
⟨*proof*⟩

lemma *bin-split-cat*:
$bin\text{-}split\ n\ (bin\text{-}cat\ v\ n\ w) = (v,\ bintrunc\ n\ w)$
⟨*proof*⟩

lemma *bin-split-zero* [*simp*]: $bin\text{-}split\ n\ 0 = (0,\ 0)$
⟨*proof*⟩

lemma *bin-split-minus1* [*simp*]:
$bin\text{-}split\ n\ (-\ 1) = (-\ 1,\ bintrunc\ n\ (-\ 1))$
⟨*proof*⟩

lemma *bin-split-trunc*:
bin-split (min m n) c = (a, b) ==>
bin-split n (bintrunc m c) = (bintrunc (m − n) a, b)
⟨*proof*⟩

lemma *bin-split-trunc1*:
bin-split n c = (a, b) ==>
bin-split n (bintrunc m c) = (bintrunc (m − n) a, bintrunc m b)
⟨*proof*⟩

lemma *bin-cat-num*:
*bin-cat a n b = a * 2 ^ n + bintrunc n b*
⟨*proof*⟩

lemma *bin-split-num*:
bin-split n b = (b div 2 ^ n, b mod 2 ^ n)
⟨*proof*⟩

122.4 Miscellaneous lemmas

lemma *nth-2p-bin*:
bin-nth (2 ^ n) m = (m = n)
⟨*proof*⟩

lemma *ex-eq-or*:
(EX m. n = Suc m & (m = k | P m)) = (n = Suc k | (EX m. n = Suc m & P m))
⟨*proof*⟩

lemma *power-BIT*: *2 ^ (Suc n) − 1 = (2 ^ n − 1) BIT True*
⟨*proof*⟩

lemma *mod-BIT*:
bin BIT bit mod 2 ^ Suc n = (bin mod 2 ^ n) BIT bit
⟨*proof*⟩

lemma *AND-mod*:
fixes *x* :: *int*
shows *x AND 2 ^ n − 1 = x mod 2 ^ n*
⟨*proof*⟩

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* = []
⟨*proof*⟩

lemma *map2-Nil2* [*simp*, *code*]:
map2 f xs [] = []
⟨*proof*⟩

lemma *map2-Cons* [*simp*, *code*]:
map2 f (*x* # *xs*) (*y* # *ys*) = *f x y* # *map2 f xs ys*
⟨*proof*⟩

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* :: $'a \Rightarrow nat \Rightarrow 'a\ list \Rightarrow 'a\ list$
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*
⟨*proof*⟩

lemma *bin-to-bl-aux-zero-minus-simp* [*simp*]:
0 < *n* ⟹ *bin-to-bl-aux n 0 bl* =
bin-to-bl-aux (*n* − *1*) *0* (*False* # *bl*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *bin-to-bl-aux-one-minus-simp* [*simp*]:
0 < *n* ⟹ *bin-to-bl-aux n 1 bl* =
bin-to-bl-aux (*n* − *1*) *0* (*True* # *bl*)
⟨*proof*⟩

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)
⟨*proof*⟩

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)
⟨*proof*⟩

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)
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *bin-to-bl-aux-append*:
bin-to-bl-aux n w bs @ cs = bin-to-bl-aux n w (bs @ cs)
⟨*proof*⟩

lemma *bl-to-bin-append*:
bl-to-bin (bs @ cs) = bl-to-bin-aux cs (bl-to-bin bs)
⟨*proof*⟩

lemma *bin-to-bl-aux-alt*:
bin-to-bl-aux n w bs = bin-to-bl n w @ bs
⟨*proof*⟩

lemma *bin-to-bl-0* [*simp*]: *bin-to-bl 0 bs = []*
⟨*proof*⟩

lemma *size-bin-to-bl-aux*:
size (bin-to-bl-aux n w bs) = n + length bs
⟨*proof*⟩

lemma *size-bin-to-bl* [*simp*]: *size (bin-to-bl n w) = n*
⟨*proof*⟩

lemma *bin-bl-bin'*:
bl-to-bin (bin-to-bl-aux n w bs) =
bl-to-bin-aux bs (bintrunc n w)
⟨*proof*⟩

lemma *bin-bl-bin* [*simp*]: *bl-to-bin (bin-to-bl n w) = bintrunc n w*

⟨*proof*⟩

lemma *bl-bin-bl'*:
bin-to-bl (*n* + *length bs*) (*bl-to-bin-aux bs w*) =
bin-to-bl-aux n w bs
⟨*proof*⟩

lemma *bl-bin-bl* [*simp*]: *bin-to-bl* (*length bs*) (*bl-to-bin bs*) = *bs*
⟨*proof*⟩

lemma *bl-to-bin-inj*:
bl-to-bin bs = *bl-to-bin cs* ==> *length bs* = *length cs* ==> *bs* = *cs*
⟨*proof*⟩

lemma *bl-to-bin-False* [*simp*]: *bl-to-bin* (*False* # *bl*) = *bl-to-bin bl*
⟨*proof*⟩

lemma *bl-to-bin-Nil* [*simp*]: *bl-to-bin* [] = *0*
⟨*proof*⟩

lemma *bin-to-bl-zero-aux*:
bin-to-bl-aux n 0 bl = *replicate n False* @ *bl*
⟨*proof*⟩

lemma *bin-to-bl-zero*: *bin-to-bl n 0* = *replicate n False*
⟨*proof*⟩

lemma *bin-to-bl-minus1-aux*:
bin-to-bl-aux n (− *1*) *bl* = *replicate n True* @ *bl*
⟨*proof*⟩

lemma *bin-to-bl-minus1*: *bin-to-bl n* (− *1*) = *replicate n True*
⟨*proof*⟩

lemma *bl-to-bin-rep-F*:
bl-to-bin (*replicate n False* @ *bl*) = *bl-to-bin bl*
⟨*proof*⟩

lemma *bin-to-bl-trunc* [*simp*]:
n <= *m* ==> *bin-to-bl n* (*bintrunc m w*) = *bin-to-bl n w*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *bin-to-bl-bintr*:
bin-to-bl n (*bintrunc m bin*) =

replicate $(n - m)$ *False* @ *bin-to-bl* $(min\ n\ m)$ *bin*
⟨*proof*⟩

lemma *bl-to-bin-rep-False*: *bl-to-bin* (*replicate n False*) = *0*
⟨*proof*⟩

lemma *len-bin-to-bl-aux*:
length (*bin-to-bl-aux n w bs*) = *n* + *length bs*
⟨*proof*⟩

lemma *len-bin-to-bl* [*simp*]: *length* (*bin-to-bl n w*) = *n*
⟨*proof*⟩

lemma *sign-bl-bin'*:
bin-sign (*bl-to-bin-aux bs w*) = *bin-sign w*
⟨*proof*⟩

lemma *sign-bl-bin*: *bin-sign* (*bl-to-bin bs*) = *0*
⟨*proof*⟩

lemma *bl-sbin-sign-aux*:
hd (*bin-to-bl-aux* (*Suc n*) *w bs*) =
(*bin-sign* (*sbintrunc n w*) = −*1*)
⟨*proof*⟩

lemma *bl-sbin-sign*:
hd (*bin-to-bl* (*Suc n*) *w*) = (*bin-sign* (*sbintrunc n w*) = −*1*)
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *bin-nth-of-bl*: *bin-nth* (*bl-to-bin bl*) *n* = (*n* < *length bl* & *rev bl* ! *n*)
⟨*proof*⟩

lemma *bin-nth-bl*: *n* < *m* ⟹ *bin-nth w n* = *nth* (*rev* (*bin-to-bl m w*)) *n*
⟨*proof*⟩

lemma *nth-rev*:
n < *length xs* ⟹ *rev xs* ! *n* = *xs* ! (*length xs* − *1* − *n*)
⟨*proof*⟩

lemma *nth-rev-alt*: *n* < *length ys* ⟹ *ys* ! *n* = *rev ys* ! (*length ys* − *Suc n*)
⟨*proof*⟩

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 ! $(n - m)$)
⟨*proof*⟩

lemma *nth-bin-to-bl*: $n < m \Longrightarrow$ (*bin-to-bl* m w) ! n = *bin-nth* w $(m -$ *Suc* $n)$
⟨*proof*⟩

lemma *bl-to-bin-lt2p-aux*:
bl-to-bin-aux bs $w < (w + 1) * (2 \hat{} \textit{length}\ bs)$
⟨*proof*⟩

lemma *bl-to-bin-lt2p-drop*:
bl-to-bin $bs < 2 \hat{} \textit{length}$ (*dropWhile Not* bs)
⟨*proof*⟩

lemma *bl-to-bin-lt2p*: *bl-to-bin* $bs < 2 \hat{} \textit{length}\ bs$
⟨*proof*⟩

lemma *bl-to-bin-ge2p-aux*:
bl-to-bin-aux bs $w >= w * (2 \hat{} \textit{length}\ bs)$
⟨*proof*⟩

lemma *bl-to-bin-ge0*: *bl-to-bin* $bs >= 0$
⟨*proof*⟩

lemma *butlast-rest-bin*:
butlast (*bin-to-bl* n w) = *bin-to-bl* $(n - 1)$ (*bin-rest* w)
⟨*proof*⟩

lemma *butlast-bin-rest*:
butlast bl = *bin-to-bl* (*length* $bl -$ *Suc* 0) (*bin-rest* (*bl-to-bin* bl))
⟨*proof*⟩

lemma *butlast-rest-bl2bin-aux*:
$bl \sim= [] \Longrightarrow$
bl-to-bin-aux (*butlast* bl) w = *bin-rest* (*bl-to-bin-aux* bl w)
⟨*proof*⟩

lemma *butlast-rest-bl2bin*:
bl-to-bin (*butlast* bl) = *bin-rest* (*bl-to-bin* bl)
⟨*proof*⟩

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*⟩

lemma *trunc-bl2bin*:
bintrunc m (*bl-to-bin* bl) = *bl-to-bin* (*drop* (*length* $bl - m$) bl)
⟨*proof*⟩

lemma *trunc-bl2bin-len* [*simp*]:
bintrunc (*length bl*) (*bl-to-bin bl*) = *bl-to-bin bl*
⟨*proof*⟩

lemma *bl2bin-drop*:
bl-to-bin (*drop k bl*) = *bintrunc* (*length bl* − *k*) (*bl-to-bin bl*)
⟨*proof*⟩

lemma *nth-rest-power-bin*:
bin-nth ((*bin-rest* ^^ *k*) *w*) *n* = *bin-nth w* (*n* + *k*)
⟨*proof*⟩

lemma *take-rest-power-bin*:
m <= *n* ==> *take m* (*bin-to-bl n w*) = *bin-to-bl m* ((*bin-rest* ^^ (*n* − *m*)) *w*)
⟨*proof*⟩

lemma *hd-butlast*: *size xs* > *1* ==> *hd* (*butlast xs*) = *hd xs*
⟨*proof*⟩

lemma *last-bin-last'*:
size xs > *0* ⟹ *last xs* ⟷ *bin-last* (*bl-to-bin-aux xs w*)
⟨*proof*⟩

lemma *last-bin-last*:
size xs > *0* ==> *last xs* ⟷ *bin-last* (*bl-to-bin xs*)
⟨*proof*⟩

lemma *bin-last-last*:
bin-last w ⟷ *last* (*bin-to-bl* (*Suc n*) *w*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *bl-not-aux-bin*:
map Not (bin-to-bl-aux n w cs) =
bin-to-bl-aux n (NOT w) (map Not cs)
⟨*proof*⟩

lemma *bl-not-bin*: *map Not (bin-to-bl n w) = bin-to-bl n (NOT w)*
⟨*proof*⟩

lemma *bl-and-bin*:
map2 (op ∧) (bin-to-bl n v) (bin-to-bl n w) = bin-to-bl n (v AND w)
⟨*proof*⟩

lemma *bl-or-bin*:
map2 (op ∨) (bin-to-bl n v) (bin-to-bl n w) = bin-to-bl n (v OR w)
⟨*proof*⟩

lemma *bl-xor-bin*:
map2 (λx y. x ≠ y) (bin-to-bl n v) (bin-to-bl n w) = bin-to-bl n (v XOR w)
⟨*proof*⟩

lemma *drop-bin2bl-aux*:
drop m (bin-to-bl-aux n bin bs) =
bin-to-bl-aux (n − m) bin (drop (m − n) bs)
⟨*proof*⟩

lemma *drop-bin2bl*: *drop m (bin-to-bl n bin) = bin-to-bl (n − m) bin*
⟨*proof*⟩

lemma *take-bin2bl-lem1*:
take m (bin-to-bl-aux m w bs) = bin-to-bl m w
⟨*proof*⟩

lemma *take-bin2bl-lem*:
take m (bin-to-bl-aux (m + n) w bs) =
take m (bin-to-bl (m + n) w)
⟨*proof*⟩

lemma *bin-split-take*:
bin-split n c = (a, b) ⟹
bin-to-bl m a = take m (bin-to-bl (m + n) c)
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *nth-takefill*: *m < n ⟹*
takefill fill n l ! m = (if m < length l then l ! m else fill)

⟨*proof*⟩

lemma *takefill-alt*:
takefill fill n l = *take n l* @ *replicate* (*n* − *length l*) *fill*
⟨*proof*⟩

lemma *takefill-replicate* [*simp*]:
takefill fill n (*replicate m fill*) = *replicate n fill*
⟨*proof*⟩

lemma *takefill-le'*:
n = *m* + *k* ⟹ *takefill x m* (*takefill x n l*) = *takefill x m l*
⟨*proof*⟩

lemma *length-takefill* [*simp*]: *length* (*takefill fill n l*) = *n*
⟨*proof*⟩

lemma *take-takefill'*:
!!*w n*. *n* = *k* + *m* ==> *take k* (*takefill fill n w*) = *takefill fill k w*
⟨*proof*⟩

lemma *drop-takefill*:
!!*w*. *drop k* (*takefill fill* (*m* + *k*) *w*) = *takefill fill m* (*drop k w*)
⟨*proof*⟩

lemma *takefill-le* [*simp*]:
m ≤ *n* ⟹ *takefill x m* (*takefill x n l*) = *takefill x m l*
⟨*proof*⟩

lemma *take-takefill* [*simp*]:
m ≤ *n* ⟹ *take m* (*takefill fill n w*) = *takefill fill m w*
⟨*proof*⟩

lemma *takefill-append*:
takefill fill (*m* + *length xs*) (*xs* @ *w*) = *xs* @ (*takefill fill m w*)
⟨*proof*⟩

lemma *takefill-same'*:
l = *length xs* ==> *takefill fill l xs* = *xs*
⟨*proof*⟩

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*)))
⟨*proof*⟩

lemma *bl-bin-bl-rtf*:
bin-to-bl n (*bl-to-bin bl*) = *rev* (*takefill False n* (*rev bl*))

⟨*proof*⟩

lemma *bl-bin-bl-rep-drop*:
bin-to-bl n (*bl-to-bin bl*) =
replicate (*n* − *length bl*) *False* @ *drop* (*length bl* − *n*) *bl*
⟨*proof*⟩

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*))))
⟨*proof*⟩

lemma *takefill-minus*:
0 < *n* ==> *takefill fill* (*Suc* (*n* − 1)) *w* = *takefill fill n w*
⟨*proof*⟩

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*) []
⟨*proof*⟩

lemma *takefill-numeral-Cons* [*simp*]:
takefill fill (*numeral k*) (*x* # *xs*) = *x* # *takefill fill* (*pred-numeral k*) *xs*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *bl-to-bin-aux-alt*:
bl-to-bin-aux bs w = *bin-cat w* (*length bs*) (*bl-to-bin bs*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *bl-to-bin-app-cat-alt*:
bin-cat (*bl-to-bin cs*) *n w* = *bl-to-bin* (*cs* @ *bin-to-bl n w*)
⟨*proof*⟩

lemma *mask-lem*: (*bl-to-bin* (*True* # *replicate n False*)) =
(*bl-to-bin* (*replicate n True*)) + *1*
⟨*proof*⟩

lemma *length-bl-of-nth* [*simp*]: *length* (*bl-of-nth n f*) = *n*
⟨*proof*⟩

lemma *nth-bl-of-nth* [*simp*]:
m < *n* ⟹ *rev* (*bl-of-nth n f*) ! *m* = *f m*
⟨*proof*⟩

lemma *bl-of-nth-inj*:
(!!*k*. *k* < *n* ==> *f k* = *g k*) ==> *bl-of-nth n f* = *bl-of-nth n g*
⟨*proof*⟩

lemma *bl-of-nth-nth-le*:
n ≤ *length xs* ⟹ *bl-of-nth n* (*nth* (*rev xs*)) = *drop* (*length xs* − *n*) *xs*
⟨*proof*⟩

lemma *bl-of-nth-nth* [*simp*]: *bl-of-nth* (*length xs*) (*op* ! (*rev xs*)) = *xs*
⟨*proof*⟩

lemma *size-rbl-pred*: *length* (*rbl-pred bl*) = *length bl*
⟨*proof*⟩

lemma *size-rbl-succ*: *length* (*rbl-succ bl*) = *length bl*
⟨*proof*⟩

lemma *size-rbl-add*:
!!*cl*. *length* (*rbl-add bl cl*) = *length bl*
⟨*proof*⟩

lemma *size-rbl-mult*:
!!*cl*. *length* (*rbl-mult bl cl*) = *length bl*
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *rbl-succ*:
rbl-succ (*rev* (*bin-to-bl n bin*)) = *rev* (*bin-to-bl n* (*bin* + *1*))
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *rbl-add-app2*:
!!*blb*. *length blb* >= *length bla* ==>
rbl-add bla (*blb* @ *blc*) = *rbl-add bla blb*
⟨*proof*⟩

lemma *rbl-add-take2*:
!!*blb*. *length blb* >= *length bla* ==>
rbl-add bla (*take* (*length bla*) *blb*) = *rbl-add bla blb*
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *rbl-mult-app2*:
!!*blb*. *length blb* >= *length bla* ==>
rbl-mult bla (*blb* @ *blc*) = *rbl-mult bla blb*

⟨*proof*⟩

lemma *rbl-mult-take2*:
length blb >= length bla ==>
rbl-mult bla (take (length bla) blb) = rbl-mult bla blb
⟨*proof*⟩

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))
⟨*proof*⟩

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))
⟨*proof*⟩

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))
⟨*proof*⟩

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))*
⟨*proof*⟩

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)))
⟨*proof*⟩

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))
⟨*proof*⟩

123.3 Repeated splitting or concatenation

lemma *sclem*:
*size (concat (map (bin-to-bl n) xs)) = length xs * n*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *bin-rcat-bl*:
(*bin-rcat n wl*) = *bl-to-bin* (*concat* (*map* (*bin-to-bl n*) *wl*))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *bin-rsplitl-aux-append*:
bin-rsplitl-aux n m c (*bs* @ *cs*) = *bin-rsplitl-aux n m c bs* @ *cs*
⟨*proof*⟩

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*
⟨*proof*⟩

lemmas *bin-split-minus-simp* =
bin-split.Suc [*THEN* [*2*] *bin-split-minus* [*symmetric*, *THEN trans*]]

lemma *bin-split-pred-simp* [*simp*]:

$(0::nat) < numeral\ bin \Longrightarrow$
$bin\text{-}split\ (numeral\ bin)\ w =$
$(let\ (w1,\ w2) = bin\text{-}split\ (numeral\ bin - 1)\ (bin\text{-}rest\ w)$
$\ in\ (w1,\ w2\ BIT\ bin\text{-}last\ w))$
⟨*proof*⟩

lemma *bin-rsplit-aux-simp-alt*:
$bin\text{-}rsplit\text{-}aux\ n\ m\ c\ bs =$
$(if\ m = 0 \vee n = 0$
$\ then\ bs$
$\ else\ let\ (a,\ b) = bin\text{-}split\ n\ c\ in\ bin\text{-}rsplit\ n\ (m - n,\ a)\ @\ b\ \#\ bs)$
⟨*proof*⟩

lemmas *bin-rsplit-simp-alt* =
$trans\ [OF\ bin\text{-}rsplit\text{-}def\ bin\text{-}rsplit\text{-}aux\text{-}simp\text{-}alt]$

lemmas *bthrs* = *bin-rsplit-simp-alt* [*THEN* [*2*] *trans*]

lemma *bin-rsplit-size-sign'* [*rule-format*] :
$[\![n > 0;\ rev\ sw = bin\text{-}rsplit\ n\ (nw,\ w)]\!] \Longrightarrow$
$(ALL\ v{:}\ set\ sw.\ bintrunc\ n\ v = v)$
⟨*proof*⟩

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*] :
$n > 0 \Longrightarrow m < n \Longrightarrow (ALL\ w\ k\ nw.\ rev\ sw = bin\text{-}rsplit\ n\ (nw,\ w) \longrightarrow$
$k < size\ sw \longrightarrow bin\text{-}nth\ (sw\ !\ k)\ m = bin\text{-}nth\ w\ (k * n + m))$
⟨*proof*⟩

lemma *bin-rsplit-all*:
$0 < nw \Longrightarrow nw <= n \Longrightarrow bin\text{-}rsplit\ n\ (nw,\ w) = [bintrunc\ n\ w]$
⟨*proof*⟩

lemma *bin-rsplit-l* [*rule-format*] :
$ALL\ bin.\ bin\text{-}rsplitl\ n\ (m,\ bin) = bin\text{-}rsplit\ n\ (m,\ bintrunc\ m\ bin)$
⟨*proof*⟩

lemma *bin-rsplit-rcat* [*rule-format*] :
$n > 0 \longrightarrow bin\text{-}rsplit\ n\ (n * size\ ws,\ bin\text{-}rcat\ n\ ws) = map\ (bintrunc\ n)\ ws$
⟨*proof*⟩

lemma *bin-rsplit-aux-len-le* [*rule-format*] :
$\forall ws\ m.\ n \neq 0 \longrightarrow ws = bin\text{-}rsplit\text{-}aux\ n\ nw\ w\ bs \longrightarrow$
$length\ ws \leq m \longleftrightarrow nw + length\ bs * n \leq m * n$
⟨*proof*⟩

lemma *bin-rsplit-len-le*:

$n \neq 0 \dashrightarrow ws = \textit{bin-rsplit}\ n\ (nw, w) \dashrightarrow (\textit{length}\ ws <= m) = (nw <= m * n)$
⟨*proof*⟩

lemma *bin-rsplit-aux-len*:
$n \neq 0 \Longrightarrow \textit{length}\ (\textit{bin-rsplit-aux}\ n\ nw\ w\ cs) =$
$(nw + n - 1)\ \textit{div}\ n + \textit{length}\ cs$
⟨*proof*⟩

lemma *bin-rsplit-len*:
$n \neq 0 \Longrightarrow \textit{length}\ (\textit{bin-rsplit}\ n\ (nw, w)) = (nw + n - 1)\ \textit{div}\ n$
⟨*proof*⟩

lemma *bin-rsplit-aux-len-indep*:
$n \neq 0 \Longrightarrow \textit{length}\ bs = \textit{length}\ cs \Longrightarrow$
$\textit{length}\ (\textit{bin-rsplit-aux}\ n\ nw\ v\ bs) =$
$\textit{length}\ (\textit{bin-rsplit-aux}\ n\ nw\ w\ cs)$
⟨*proof*⟩

lemma *bin-rsplit-len-indep*:
$n \neq 0 \Longrightarrow \textit{length}\ (\textit{bin-rsplit}\ n\ (nw, v)) = \textit{length}\ (\textit{bin-rsplit}\ n\ (nw, w))$
⟨*proof*⟩

Even more bit operations

instantiation *int* :: *bitss*
begin

definition [*iff*]:
$i\ !!\ n \longleftrightarrow \textit{bin-nth}\ i\ n$

definition
$\textit{lsb}\ i = (i :: \textit{int})\ !!\ 0$

definition
$\textit{set-bit}\ i\ n\ b = \textit{bin-sc}\ n\ b\ i$

definition
$\textit{set-bits}\ f =$
$(\textit{if}\ \exists n.\ \forall n' \geq n.\ \neg\ f\ n'\ \textit{then}$
$\textit{let}\ n = \textit{LEAST}\ n.\ \forall n' \geq n.\ \neg\ f\ n'$
$\textit{in bl-to-bin}\ (\textit{rev}\ (\textit{map}\ f\ [0..<n]))$
$\textit{else if}\ \exists n.\ \forall n' \geq n.\ f\ n'\ \textit{then}$
$\textit{let}\ n = \textit{LEAST}\ n.\ \forall n' \geq n.\ f\ n'$
$\textit{in sbintrunc}\ n\ (\textit{bl-to-bin}\ (\textit{True}\ \#\ \textit{rev}\ (\textit{map}\ f\ [0..<n])))$
$\textit{else}\ 0 :: \textit{int})$

definition
$\textit{shiftl}\ x\ n = (x :: \textit{int}) * 2\ \hat{}\ n$

definition
shiftr x n = (*x* :: *int*) *div 2* ^ *n*

definition
msb x ⟷ (*x* :: *int*) < *0*

instance ⟨*proof*⟩

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* ⟹ ∀ *x*. *Rep x* ∈ *A* **and**
tdD2: *type-definition Rep Abs A* ⟹ ∀ *x*. *Abs* (*Rep x*) = *x* **and**
tdD3: *type-definition Rep Abs A* ⟹ ∀ *y*. *y* ∈ *A* ⟶ *Rep* (*Abs y*) = *y*
⟨*proof*⟩

lemma *td-nat-int*:
type-definition int nat (*Collect* (*op* <= *0*))
⟨*proof*⟩

context *type-definition*
begin

declare *Rep* [*iff*] *Rep-inverse* [*simp*] *Rep-inject* [*simp*]

lemma *Abs-eqD*: *Abs x* = *Abs y* ==> *x* ∈ *A* ==> *y* ∈ *A* ==> *x* = *y*
⟨*proof*⟩

lemma *Abs-inverse'*:
r : *A* ==> *Abs r* = *a* ==> *Rep a* = *r*
⟨*proof*⟩

lemma *Rep-comp-inverse*:
Rep o f = *g* ==> *Abs o g* = *f*
⟨*proof*⟩

lemma *Rep-eqD* [*elim*!]: *Rep x* = *Rep y* ==> *x* = *y*
⟨*proof*⟩

lemma *Rep-inverse'*: *Rep a = r ==> Abs r = a*
⟨*proof*⟩

lemma *comp-Abs-inverse*:
f o Abs = g ==> g o Rep = f
⟨*proof*⟩

lemma *set-Rep*:
A = range Rep
⟨*proof*⟩

lemma *set-Rep-Abs*: *A = range (Rep o Abs)*
⟨*proof*⟩

lemma *Abs-inj-on*: *inj-on Abs A*
⟨*proof*⟩

lemma *image*: *Abs ' A = UNIV*
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

end

interpretation *nat-int*: *type-definition int nat Collect* (*op <= 0*)
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *fn-comm-power*:
fa o tr = tr o fr ==> fa ^^ n o tr = tr o fr ^^ n
⟨*proof*⟩

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*: ⋀*x*. *Rep* (*Abs x*) = *norm x*
begin

lemma *Abs-norm* [*simp*]:
Abs (*norm x*) = *Abs x*
⟨*proof*⟩

lemma *td-th*:
g o Abs = f ==> f (*Rep x*) = *g x*
⟨*proof*⟩

lemma *eq-norm′*: *Rep o Abs = norm*
⟨*proof*⟩

lemma *norm-Rep* [*simp*]: *norm* (*Rep x*) = *Rep x*
⟨*proof*⟩

lemmas *td = td-thm*

lemma *set-iff-norm*: *w* : *A* ⟷ *w = norm w*
⟨*proof*⟩

lemma *inverse-norm*:
(*Abs n = w*) = (*Rep w = norm n*)
⟨*proof*⟩

lemma *norm-eq-iff*:
(*norm x = norm y*) = (*Abs x = Abs y*)
⟨*proof*⟩

lemma *norm-comps*:
Abs o norm = Abs
norm o Rep = Rep
norm o norm = norm
⟨*proof*⟩

lemmas *norm-norm* [*simp*] = *norm-comps*

lemma *fns5*:
Rep o fa o Abs = fr ==>
fr o norm = fr & norm o fr = fr
⟨*proof*⟩

lemma *fns2*:
Abs o fr o Rep = fa ==>
(norm o fr o norm = fr o norm) = (Rep o fa = fr o Rep)
⟨*proof*⟩

lemma *fns3*:
Abs o fr o Rep = fa ==>
(norm o fr o norm = norm o fr) = (fa o Abs = Abs o fr)
⟨*proof*⟩

lemma *fns*:
fr o norm = norm o fr ==>
(fa o Abs = Abs o fr) = (Rep o fa = fr o Rep)
⟨*proof*⟩

lemma *range-norm*:
range (Rep o Abs) = A
⟨*proof*⟩

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 ⟹ a ^ n = a * a ^ (n − 1)*
⟨*proof*⟩

lemma *funpow-minus-simp*:
0 < n ⟹ f ^^ n = f ∘ f ^^ (n − 1)
⟨*proof*⟩

lemma *power-numeral*:
$a \hat{} numeral\ k = a * a \hat{} (pred\text{-}numeral\ k)$
⟨*proof*⟩

lemma *funpow-numeral* [*simp*]:
$f \hat{}\hat{} numeral\ k = f \circ f \hat{}\hat{} (pred\text{-}numeral\ k)$
⟨*proof*⟩

lemma *replicate-numeral* [*simp*]:
$replicate\ (numeral\ k)\ x = x \# replicate\ (pred\text{-}numeral\ k)\ x$
⟨*proof*⟩

lemma *rco-alt*: $(f\ o\ g) \hat{}\hat{} n\ o\ f = f\ o\ (g\ o\ f) \hat{}\hat{} n$
⟨*proof*⟩

lemma *list-exhaust-size-gt0*:
assumes *y*: $\bigwedge a\ list.\ y = a \# list \Longrightarrow P$
shows $0 < length\ y \Longrightarrow P$
⟨*proof*⟩

lemma *list-exhaust-size-eq0*:
assumes *y*: $y = [] \Longrightarrow P$
shows $length\ y = 0 \Longrightarrow P$
⟨*proof*⟩

lemma *size-Cons-lem-eq*:
$y = xa \# list \Longrightarrow size\ y = Suc\ k \Longrightarrow size\ list = k$
⟨*proof*⟩

lemmas *ls-splits* = *prod.split prod.split-asm if-split-asm*

lemma *not-B1-is-B0*: $y \neq (1::bit) \Longrightarrow y = (0::bit)$
⟨*proof*⟩

lemma *B1-ass-B0*:
assumes *y*: $y = (0::bit) \Longrightarrow y = (1::bit)$
shows $y = (1::bit)$
⟨*proof*⟩
lemmas *n2s-ths* [*THEN eq-reflection*] = *add-2-eq-Suc add-2-eq-Suc'*

lemmas *s2n-ths* = *n2s-ths* [*symmetric*]

lemma *and-len*: $xs = ys \Longrightarrow xs = ys\ \&\ length\ xs = length\ ys$
⟨*proof*⟩

lemma *size-if*: $size\ (if\ p\ then\ xs\ else\ ys) = (if\ p\ then\ size\ xs\ else\ size\ ys)$
⟨*proof*⟩

lemma *tl-if*: $tl\ (if\ p\ then\ xs\ else\ ys) = (if\ p\ then\ tl\ xs\ else\ tl\ ys)$

⟨*proof*⟩

lemma *hd-if*: *hd* (*if p then xs else ys*) = (*if p then hd xs else hd ys*)
⟨*proof*⟩

lemma *if-Not-x*: (*if p then* ~ *x else x*) = (*p* = (~ *x*))
⟨*proof*⟩

lemma *if-x-Not*: (*if p then x else* ~ *x*) = (*p* = *x*)
⟨*proof*⟩

lemma *if-same-and*: (*If p x y* & *If p u v*) = (*if p then x* & *u else y* & *v*)
⟨*proof*⟩

lemma *if-same-eq*: (*If p x y* = (*If p u v*)) = (*if p then x* = (*u*) *else y* = (*v*))
⟨*proof*⟩

lemma *if-same-eq-not*:
(*If p x y* = (~ *If p u v*)) = (*if p then x* = (~*u*) *else y* = (~*v*))
⟨*proof*⟩

lemma *if-Cons*: (*if p then x # xs else y # ys*) = *If p x y # If p xs ys*
⟨*proof*⟩

lemma *if-single*:
(*if xc then* [*xab*] *else* [*an*]) = [*if xc then xab else an*]
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *nonemptyE*: *S* ~= {} ==> (!!*x*. *x* : *S* ==> *R*) ==> *R* ⟨*proof*⟩

lemma *gt-or-eq-0*: *0* < *y* ∨ *0* = (*y*::*nat*) ⟨*proof*⟩

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*: $j = k + i \Longrightarrow j - i = (k :: nat)$ ⟨*proof*⟩

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*
⟨*proof*⟩

lemmas *eme1p* = *emep1* [*simplified add.commute*]

lemma *le-diff-eq'*: $(a \leq c - b) = (b + a \leq (c::int))$ ⟨*proof*⟩

lemma *less-diff-eq'*: $(a < c - b) = (b + a < (c::int))$ ⟨*proof*⟩

lemma *diff-less-eq'*: $(a - b < c) = (a < b + (c::int))$ ⟨*proof*⟩

lemmas *m1mod22k* = *mult-pos-pos* [*OF zless2 zless2p*, *THEN zmod-minus1*]

lemma *z1pdiv2*:
$(2 * b + 1)$ *div 2* = (*b*::*int*) ⟨*proof*⟩

lemmas *zdiv-le-dividend* = *xtr3* [*OF div-by-1* [*symmetric*] *zdiv-mono2*,
simplified int-one-le-iff-zero-less, *simplified*]

lemma *axxbyy*:
$a + m + m = b + n + n \Longrightarrow (a = 0 \mid a = 1) \Longrightarrow (b = 0 \mid b = 1) \Longrightarrow$
$a = b \;\&\; m = (n :: int)$ ⟨*proof*⟩

lemma *axxmod2*:
$(1 + x + x)$ *mod 2* = $(1 :: int) \;\&\; (0 + x + x)$ *mod 2* = $(0 :: int)$ ⟨*proof*⟩

lemma *axxdiv2*:
$(1 + x + x)$ *div 2* = $(x :: int) \;\&\; (0 + x + x)$ *div 2* = $(x :: int)$ ⟨*proof*⟩

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*:
$((a + x)\ mod\ m = (b + x)\ mod\ m) = (a\ mod\ m = b\ mod\ (m :: nat))$
⟨*proof*⟩

lemma *nat-minus-mod*: $(n - n\ mod\ m)\ mod\ m = (0 :: nat)$
⟨*proof*⟩

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*:
$!!b.\ b < n ==> a\ mod\ n = b\ mod\ n ==> a\ mod\ n = (b :: nat)$
⟨*proof*⟩

lemmas *nat-mod-eq'* = *refl* [*THEN* [*2*] *nat-mod-eq*]

lemma *nat-mod-lem*:
$(0 :: nat) < n ==> b < n = (b\ mod\ n = b)$
⟨*proof*⟩

lemma *mod-nat-add*:
$(x :: nat) < z ==> y < z ==>$
$(x + y)\ mod\ z = (if\ x + y < z\ then\ x + y\ else\ x + y - z)$
⟨*proof*⟩

lemma *mod-nat-sub*:
$(x :: nat) < z ==> (x - y)\ mod\ z = x - y$
⟨*proof*⟩

lemma *int-mod-eq*:
$(0 :: int) <= b ==> b < n ==> a\ mod\ n = b\ mod\ n ==> a\ mod\ n = b$
⟨*proof*⟩

lemmas *int-mod-eq'* = *mod-pos-pos-trivial*

lemma *int-mod-le*: $(0::int) <= a ==> a\ mod\ n <= a$
⟨*proof*⟩

lemma *mod-add-if-z*:
$(x :: int) < z ==> y < z ==> 0 <= y ==> 0 <= x ==> 0 <= z ==>$
$(x + y)\ mod\ z = (if\ x + y < z\ then\ x + y\ else\ x + y - z)$
⟨*proof*⟩

lemma *mod-sub-if-z*:
$(x :: int) < z ==> y < z ==> 0 <= y ==> 0 <= x ==> 0 <= z ==>$
$(x - y)\ mod\ z = (if\ y <= x\ then\ x - y\ else\ x - y + z)$
⟨*proof*⟩

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*: $m\ ^{\sim}= (0 :: int) ==> (a + m * n)\ div\ m = a\ div\ m + n$
⟨*proof*⟩

lemma *mod-power-lem*:
$a > 1 ==> a\ \hat{}\ n\ mod\ a\ \hat{}\ m = (if\ m <= n\ then\ 0\ else\ (a :: int)\ \hat{}\ n)$
⟨*proof*⟩

lemma *pl-pl-rels*:
$a + b = c + d ==>$
$a >= c\ \&\ b <= d \mid a <= c\ \&\ b >= (d :: nat)$ ⟨*proof*⟩

lemmas *pl-pl-rels'* = *add.commute* [*THEN* [*2*] *trans*, *THEN pl-pl-rels*]

lemma *minus-eq*: $(m - k = m) = (k = 0 \mid m = (0 :: nat))$ ⟨*proof*⟩

lemma *pl-pl-mm*: $(a :: nat) + b = c + d ==> a - c = d - b$ ⟨*proof*⟩

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*:
$0 < c ==> (a >= b * c) = (a\ div\ c >= (b :: nat))$
⟨*proof*⟩

lemmas *td-gal-lt* = *td-gal* [*simplified not-less* [*symmetric*], *simplified*]

lemma *div-mult-le*: $(a :: nat)\ div\ b * b <= a$
⟨*proof*⟩

lemmas *sdl* = *split-div-lemma* [*THEN iffD1*, *symmetric*]

lemma *given-quot*: $f > (0 :: nat) ==> (f * l + (f - 1)) \ div \ f = l$
⟨*proof*⟩

lemma *given-quot-alt*: $f > (0 :: nat) ==> (l * f + f - Suc\ 0) \ div \ f = l$
⟨*proof*⟩

lemma *diff-mod-le*: $(a::nat) < d ==> b \ dvd \ d ==> a - a \ mod \ b <= d - b$
⟨*proof*⟩

lemma *less-le-mult'*:
$w * c < b * c ==> 0 \leq c ==> (w + 1) * c \leq b * (c::int)$
⟨*proof*⟩

lemma *less-le-mult*:
$w * c < b * c \Longrightarrow 0 \leq c \Longrightarrow w * c + c \leq b * (c::int)$
⟨*proof*⟩

lemmas *less-le-mult-minus* = *iffD2* [*OF le-diff-eq less-le-mult*,
simplified left-diff-distrib]

lemma *gen-minus*: $0 < n ==> f\ n = f\ (Suc\ (n - 1))$
⟨*proof*⟩

lemma *mpl-lem*: $j <= (i :: nat) ==> k < j ==> i - j + k < i$ ⟨*proof*⟩

lemma *nonneg-mod-div*:
$0 <= a ==> 0 <= b ==> 0 <= (a \ mod \ b :: int) \ \& \ 0 <= a \ div \ b$
⟨*proof*⟩

declare *iszero-0* [*intro*]

lemma *min-pm* [*simp*]:
$min\ a\ b + (a - b) = (a :: nat)$
⟨*proof*⟩

lemma *min-pm1* [*simp*]:
$a - b + min\ a\ b = (a :: nat)$
⟨*proof*⟩

lemma *rev-min-pm* [*simp*]:
$min\ b\ a + (a - b) = (a :: nat)$
⟨*proof*⟩

lemma *rev-min-pm1* [*simp*]:
$a - b + min\ b\ a = (a :: nat)$
⟨*proof*⟩

lemma *min-minus* [*simp*]:
min m (*m* − *k*) = (*m* − *k* :: *nat*)
⟨*proof*⟩

lemma *min-minus'* [*simp*]:
min (*m* − *k*) *m* = (*m* − *k* :: *nat*)
⟨*proof*⟩

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* ⟨*proof*⟩

lemma *uint-nonnegative*:
0 ≤ *uint w*
⟨*proof*⟩

lemma *uint-bounded*:
fixes *w* :: *'a*::*len0 word*
shows *uint w* < *2* ^ *len-of TYPE*(*'a*)
⟨*proof*⟩

lemma *uint-idem*:
fixes *w* :: *'a*::*len0 word*
shows *uint w mod 2* ^ *len-of TYPE*(*'a*) = *uint w*
⟨*proof*⟩

lemma *word-uint-eq-iff*:
a = *b* ⟷ *uint a* = *uint b*
⟨*proof*⟩

lemma *word-uint-eqI*:

uint a = uint b ⟹ a = b
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *word-of-int-uint*:
word-of-int (uint w) = w
⟨*proof*⟩

lemma *split-word-all*:
(⋀x::'a::len0 word. PROP P x) ≡ (⋀x. PROP P (word-of-int x))
⟨*proof*⟩

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}
⟨*proof*⟩

lemma *sints-num*:
sints n = {i. − (2 ^ (n − 1)) ≤ i ∧ i < 2 ^ (n − 1)}
⟨*proof*⟩

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 ⟨*proof*⟩

end

lemma *word-size-gt-0* [*iff*]:
0 < *size* (*w*::'a::*len word*)
⟨*proof*⟩

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
⟨*proof*⟩

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 ⟷ *target-size c* ≤ *source-size c*

definition *of-bl* :: *bool list* ⇒ *'a*::*len0 word*
where
of-bl bl = *word-of-int* (*bl-to-bin bl*)

definition *to-bl* :: *'a*::*len0 word* ⇒ *bool list*
where
to-bl w = *bin-to-bl* (*len-of TYPE* (*'a*)) (*uint w*)

definition *word-reverse* :: *'a*::*len0 word* ⇒ *'a word*
where
word-reverse w = *of-bl* (*rev* (*to-bl w*))

definition *word-int-case* :: (*int* ⇒ *'b*) ⇒ *'a*::*len0 word* ⇒ *'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* ⇒ *'a*::*len0 word* ⇒ *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* :: - ⇒ *'a*::*len0 word* ⇒ *bool*)
⟨*proof*⟩

lemma *reflp-word*:
reflp (λ*x y*. *bintrunc* (*len-of TYPE*(*'a*::*len0*)) *x* = *bintrunc* (*len-of TYPE*(*'a*)) *y*)
⟨*proof*⟩

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* ⇒ *int*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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 ⟨*proof*⟩

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*)))
(λ*w*::*int*. *w mod 2* ^ *len-of TYPE*(*'a*))

⟨*proof*⟩

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*)
⟨*proof*⟩

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*)))
⟨*proof*⟩

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*))
⟨*proof*⟩

127.6 Arithmetic operations

lift-definition *word-succ* :: *'a*::*len0 word* ⇒ *'a word* **is** λ*x*. *x* + *1*
⟨*proof*⟩

lift-definition *word-pred* :: *'a*::*len0 word* ⇒ *'a word* **is** λ*x*. *x* − *1*
⟨*proof*⟩

instantiation *word* :: (*len0*) {*neg-numeral*, *Divides.div*, *comm-monoid-mult*, *comm-ring*}
begin

lift-definition *zero-word* :: *'a word* **is** *0* ⟨*proof*⟩

lift-definition *one-word* :: *'a word* **is** *1* ⟨*proof*⟩

lift-definition *plus-word* :: *'a word* ⇒ *'a word* ⇒ *'a word* **is** *op* +
⟨*proof*⟩

lift-definition *minus-word* :: *'a word* ⇒ *'a word* ⇒ *'a word* **is** *op* −
⟨*proof*⟩

lift-definition *uminus-word* :: *'a word* ⇒ *'a word* **is** *uminus*
⟨*proof*⟩

lift-definition *times-word* :: *'a word* ⇒ *'a word* ⇒ *'a word* **is** *op* ∗

⟨*proof*⟩

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
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

lemma *word-of-nat*: *of-nat n* = *word-of-int* (*int n*)
⟨*proof*⟩

lemma *word-of-int*: *of-int* = *word-of-int*
⟨*proof*⟩

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
⟨*proof*⟩

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*
⟨*proof*⟩

lift-definition *bitAND-word* :: *'a word* ⇒ *'a word* ⇒ *'a word* **is** *bitAND*
⟨*proof*⟩

lift-definition *bitOR-word* :: *'a word* ⇒ *'a word* ⇒ *'a word* **is** *bitOR*
⟨*proof*⟩

lift-definition *bitXOR-word* :: *'a word* ⇒ *'a word* ⇒ *'a word* **is** *bitXOR*
⟨*proof*⟩

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 ⟨*proof*⟩

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-of-int (uint a XOR uint b)*
⟨*proof*⟩

instantiation *word* :: *(len) bitss*
begin

definition
word-msb-def:
msb a ⟷ bin-sign (sint a) = −1

instance ⟨*proof*⟩

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)
⟨*proof*⟩

lemma *uint-sint*:
uint w = bintrunc (len-of TYPE('a)) (sint (w :: 'a :: len word))
⟨*proof*⟩

lemma *bintr-uint*:
fixes $w :: 'a::len0\ word$
shows $len\text{-}of\ TYPE('a) \leq n \Longrightarrow bintrunc\ n\ (uint\ w) = uint\ w$
⟨*proof*⟩

lemma *wi-bintr*:
$len\text{-}of\ TYPE('a::len0) \leq n \Longrightarrow$
$word\text{-}of\text{-}int\ (bintrunc\ n\ w) = (word\text{-}of\text{-}int\ w :: 'a\ word)$
⟨*proof*⟩

lemma *td-ext-sbin*:
$td\text{-}ext\ (sint :: 'a\ word \Rightarrow int)\ word\text{-}of\text{-}int\ (sints\ (len\text{-}of\ TYPE('a::len)))$
$(sbintrunc\ (len\text{-}of\ TYPE('a) - 1))$
⟨*proof*⟩

lemma *td-ext-sint*:
$td\text{-}ext\ (sint :: 'a\ word \Rightarrow int)\ word\text{-}of\text{-}int\ (sints\ (len\text{-}of\ TYPE('a::len)))$
$(\lambda w.\ (w + 2\ \hat{}\ (len\text{-}of\ TYPE('a) - 1))\ mod\ 2\ \hat{}\ len\text{-}of\ TYPE('a) -$
$2\ \hat{}\ (len\text{-}of\ TYPE('a) - 1))$
⟨*proof*⟩

interpretation *word-sint*:
$td\text{-}ext\ sint :: 'a::len\ word => int$
$word\text{-}of\text{-}int$
$sints\ (len\text{-}of\ TYPE('a::len))$
$\%w.\ (w + 2\hat{}(len\text{-}of\ TYPE('a::len) - 1))\ mod\ 2\hat{}len\text{-}of\ TYPE('a::len) -$
$2\ \hat{}\ (len\text{-}of\ TYPE('a::len) - 1)$
⟨*proof*⟩

interpretation *word-sbin*:
$td\text{-}ext\ sint :: 'a::len\ word => int$
$word\text{-}of\text{-}int$
$sints\ (len\text{-}of\ TYPE('a::len))$
$sbintrunc\ (len\text{-}of\ TYPE('a::len) - 1)$
⟨*proof*⟩

lemmas *int-word-sint* = *td-ext-sint* [*THEN td-ext.eq-norm*]

lemmas *td-sint* = *word-sint.td*

lemma *to-bl-def'*:
$(to\text{-}bl :: 'a :: len0\ word => bool\ list) =$
$bin\text{-}to\text{-}bl\ (len\text{-}of\ TYPE('a))\ o\ uint$
⟨*proof*⟩

lemmas *word-reverse-no-def* [*simp*] = *word-reverse-def* [*of numeral w*] **for** *w*

lemma *uints-mod*: *uints n = range (λw. w mod 2 ^ n)*
⟨*proof*⟩

lemma *word-numeral-alt*:
numeral b = word-of-int (numeral b)
⟨*proof*⟩

declare *word-numeral-alt* [*symmetric*, *code-abbrev*]

lemma *word-neg-numeral-alt*:
− numeral b = word-of-int (− numeral b)
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *uint-bintrunc* [*simp*]:
uint (numeral bin :: 'a word) =
bintrunc (len-of TYPE ('a :: len0)) (numeral bin)
⟨*proof*⟩

lemma *uint-bintrunc-neg* [*simp*]: *uint (− numeral bin :: 'a word) =*
bintrunc (len-of TYPE ('a :: len0)) (− numeral bin)
⟨*proof*⟩

lemma *sint-sbintrunc* [*simp*]:
sint (numeral bin :: 'a word) =
sbintrunc (len-of TYPE ('a :: len) − 1) (numeral bin)
⟨*proof*⟩

lemma *sint-sbintrunc-neg* [*simp*]: *sint (− numeral bin :: 'a word) =*
sbintrunc (len-of TYPE ('a :: len) − 1) (− numeral bin)
⟨*proof*⟩

lemma *unat-bintrunc* [*simp*]:
unat (numeral bin :: 'a :: len0 word) =
nat (bintrunc (len-of TYPE('a)) (numeral bin))
⟨*proof*⟩

lemma *unat-bintrunc-neg* [*simp*]:
unat (− numeral bin :: 'a :: len0 word) =
nat (bintrunc (len-of TYPE('a)) (− numeral bin))
⟨*proof*⟩

lemma *size-0-eq*: *size (w :: 'a :: len0 word) = 0 ⟹ v = w*

⟨*proof*⟩

lemma *uint-ge-0* [*iff*]: *0* ≤ *uint* (*x*::*'a*::*len0 word*)
⟨*proof*⟩

lemma *uint-lt2p* [*iff*]: *uint* (*x*::*'a*::*len0 word*) < *2* ^ *len-of TYPE*(*'a*)
⟨*proof*⟩

lemma *sint-ge*: − (*2* ^ (*len-of TYPE*(*'a*) − *1*)) ≤ *sint* (*x*::*'a*::*len word*)
⟨*proof*⟩

lemma *sint-lt*: *sint* (*x*::*'a*::*len word*) < *2* ^ (*len-of TYPE*(*'a*) − *1*)
⟨*proof*⟩

lemma *sign-uint-Pls* [*simp*]:
bin-sign (*uint x*) = *0*
⟨*proof*⟩

lemma *uint-m2p-neg*: *uint* (*x*::*'a*::*len0 word*) − *2* ^ *len-of TYPE*(*'a*) < *0*
⟨*proof*⟩

lemma *uint-m2p-not-non-neg*:
¬ *0* ≤ *uint* (*x*::*'a*::*len0 word*) − *2* ^ *len-of TYPE*(*'a*)
⟨*proof*⟩

lemma *lt2p-lem*:
len-of TYPE(*'a*) ≤ *n* ⟹ *uint* (*w* :: *'a*::*len0 word*) < *2* ^ *n*
⟨*proof*⟩

lemma *uint-le-0-iff* [*simp*]: *uint x* ≤ *0* ⟷ *uint x* = *0*
⟨*proof*⟩

lemma *uint-nat*: *uint w* = *int* (*unat w*)
⟨*proof*⟩

lemma *uint-numeral*:
uint (*numeral b* :: *'a* :: *len0 word*) = *numeral b mod 2* ^ *len-of TYPE*(*'a*)
⟨*proof*⟩

lemma *uint-neg-numeral*:
uint (− *numeral b* :: *'a* :: *len0 word*) = − *numeral b mod 2* ^ *len-of TYPE*(*'a*)
⟨*proof*⟩

lemma *unat-numeral*:
unat (*numeral b*::*'a*::*len0 word*) = *numeral b mod 2* ^ *len-of TYPE* (*'a*)
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *word-of-int-0* [*simp*, *code-post*]:
word-of-int 0 = 0
⟨*proof*⟩

lemma *word-of-int-1* [*simp*, *code-post*]:
word-of-int 1 = 1
⟨*proof*⟩

lemma *word-of-int-neg-1* [*simp*]: *word-of-int (- 1) = - 1*
⟨*proof*⟩

lemma *word-of-int-numeral* [*simp*] :
(word-of-int (numeral bin) :: 'a :: len0 word) = (numeral bin)
⟨*proof*⟩

lemma *word-of-int-neg-numeral* [*simp*]:
(word-of-int (- numeral bin) :: 'a :: len0 word) = (- numeral bin)
⟨*proof*⟩

lemma *word-int-case-wi*:
word-int-case f (word-of-int i :: 'b word) =
f (i mod 2 ^ len-of TYPE('b::len0))
⟨*proof*⟩

lemma *word-int-split*:
P (word-int-case f x) =
(ALL i. x = (word-of-int i :: 'b :: len0 word) &
0 <= i & i < 2 ^ len-of TYPE('b) --> P (f i))
⟨*proof*⟩

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 ^ len-of TYPE('b::len0) & ~ P (f n)))
⟨*proof*⟩

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 ^ size w*
⟨*proof*⟩

lemma *sint-range-size*:
- (2 ^ (size w - Suc 0)) <= sint w & sint w < 2 ^ (size w - Suc 0)
⟨*proof*⟩

lemma *sint-above-size*: $2 \hat{} (size\ (w::'a::len\ word) - 1) \leq x \Longrightarrow sint\ w < x$
⟨*proof*⟩

lemma *sint-below-size*:
$x \leq - (2 \hat{} (size\ (w::'a::len\ word) - 1)) \Longrightarrow x \leq sint\ w$
⟨*proof*⟩

127.13 Testing bits

lemma *test-bit-eq-iff*: $(test\text{-}bit\ (u::'a::len0\ word) = test\text{-}bit\ v) = (u = v)$
⟨*proof*⟩

lemma *test-bit-size* [*rule-format*] : $(w::'a::len0\ word)\ !!\ n \longrightarrow n < size\ w$
⟨*proof*⟩

lemma *word-eq-iff*:
fixes $x\ y :: 'a::len0\ word$
shows $x = y \longleftrightarrow (\forall n{<}len\text{-}of\ TYPE('a).\ x\ !!\ n = y\ !!\ n)$
⟨*proof*⟩

lemma *word-eqI* [*rule-format*]:
fixes $u :: 'a::len0\ word$
shows $(ALL\ n.\ n < size\ u \longrightarrow u\ !!\ n = v\ !!\ n) \Longrightarrow u = v$
⟨*proof*⟩

lemma *word-eqD*: $(u::'a::len0\ word) = v \Longrightarrow u\ !!\ x = v\ !!\ x$
⟨*proof*⟩

lemma *test-bit-bin'*: $w\ !!\ n = (n < size\ w\ \&\ bin\text{-}nth\ (uint\ w)\ n)$
⟨*proof*⟩

lemmas *test-bit-bin* = *test-bit-bin'* [*unfolded word-size*]

lemma *bin-nth-uint-imp*:
$bin\text{-}nth\ (uint\ (w::'a::len0\ word))\ n \Longrightarrow n < len\text{-}of\ TYPE('a)$
⟨*proof*⟩

lemma *bin-nth-sint*:
fixes $w :: 'a::len\ word$
shows $len\text{-}of\ TYPE('a) \leq n \Longrightarrow$
$bin\text{-}nth\ (sint\ w)\ n = bin\text{-}nth\ (sint\ w)\ (len\text{-}of\ TYPE('a) - 1)$
⟨*proof*⟩

lemma *td-bl*:
type-definition $(to\text{-}bl :: 'a::len0\ word \Rightarrow bool\ list)$
of-bl
$\{bl.\ length\ bl = len\text{-}of\ TYPE('a)\}$
⟨*proof*⟩

interpretation *word-bl*:
type-definition to-bl :: *'a*::*len0 word* => *bool list*
of-bl
{*bl*. *length bl* = *len-of TYPE*(*'a*::*len0*)}
⟨*proof*⟩

lemmas *word-bl-Rep'* = *word-bl.Rep* [*unfolded mem-Collect-eq*, *iff*]

lemma *word-size-bl*: *size w* = *size* (*to-bl w*)
⟨*proof*⟩

lemma *to-bl-use-of-bl*:
(*to-bl w* = *bl*) = (*w* = *of-bl bl* ∧ *length bl* = *length* (*to-bl w*))
⟨*proof*⟩

lemma *to-bl-word-rev*: *to-bl* (*word-reverse w*) = *rev* (*to-bl w*)
⟨*proof*⟩

lemma *word-rev-rev* [*simp*] : *word-reverse* (*word-reverse w*) = *w*
⟨*proof*⟩

lemma *word-rev-gal*: *word-reverse w* = *u* ⟹ *word-reverse u* = *w*
⟨*proof*⟩

lemma *word-rev-gal'*: *u* = *word-reverse w* ⟹ *w* = *word-reverse u*
⟨*proof*⟩

lemma *length-bl-gt-0* [*iff*]: *0* < *length* (*to-bl* (*x*::*'a*::*len word*))
⟨*proof*⟩

lemma *bl-not-Nil* [*iff*]: *to-bl* (*x*::*'a*::*len word*) ≠ []
⟨*proof*⟩

lemma *length-bl-neq-0* [*iff*]: *length* (*to-bl* (*x*::*'a*::*len word*)) ≠ *0*
⟨*proof*⟩

lemma *hd-bl-sign-sint*: *hd* (*to-bl w*) = (*bin-sign* (*sint w*) = −*1*)
⟨*proof*⟩

lemma *of-bl-drop'*:
lend = *length bl* − *len-of TYPE* (*'a* :: *len0*) ⟹
of-bl (*drop lend bl*) = (*of-bl bl* :: *'a word*)
⟨*proof*⟩

lemma *test-bit-of-bl*:
(*of-bl bl*::*'a*::*len0 word*) !! *n* = (*rev bl* ! *n* ∧ *n* < *len-of TYPE*(*'a*) ∧ *n* < *length bl*)
⟨*proof*⟩

lemma *no-of-bl*:
(*numeral bin* ::'*a*::*len0 word*) = *of-bl* (*bin-to-bl* (*len-of TYPE* ('*a*)) (*numeral bin*))
⟨*proof*⟩

lemma *uint-bl*: *to-bl w* = *bin-to-bl* (*size w*) (*uint w*)
⟨*proof*⟩

lemma *to-bl-bin*: *bl-to-bin* (*to-bl w*) = *uint w*
⟨*proof*⟩

lemma *to-bl-of-bin*:
to-bl (*word-of-int bin*::'*a*::*len0 word*) = *bin-to-bl* (*len-of TYPE*('*a*)) *bin*
⟨*proof*⟩

lemma *to-bl-numeral* [*simp*]:
to-bl (*numeral bin*::'*a*::*len0 word*) =
bin-to-bl (*len-of TYPE*('*a*)) (*numeral bin*)
⟨*proof*⟩

lemma *to-bl-neg-numeral* [*simp*]:
to-bl (− *numeral bin*::'*a*::*len0 word*) =
bin-to-bl (*len-of TYPE*('*a*)) (− *numeral bin*)
⟨*proof*⟩

lemma *to-bl-to-bin* [*simp*] : *bl-to-bin* (*to-bl w*) = *uint w*
⟨*proof*⟩

lemma *uint-bl-bin*:
fixes *x* :: '*a*::*len0 word*
shows *bl-to-bin* (*bin-to-bl* (*len-of TYPE*('*a*)) (*uint x*)) = *uint x*
⟨*proof*⟩

lemma *uints-unats*: *uints n* = *int* ‘ *unats n*
⟨*proof*⟩

lemma *unats-uints*: *unats n* = *nat* ‘ *uints n*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *num-of-sbintr'*:
sbintrunc (*len-of TYPE*('*a* :: *len*) − *1*) (*numeral a*) = (*numeral b*) ⟹
numeral a = (*numeral b* :: '*a word*)
⟨*proof*⟩

lemma *num-abs-bintr*:
(*numeral x* :: '*a word*) =
word-of-int (*bintrunc* (*len-of TYPE*('*a*::*len0*)) (*numeral x*))
⟨*proof*⟩

lemma *num-abs-sbintr*:
(*numeral x* :: '*a word*) =
word-of-int (*sbintrunc* (*len-of TYPE*('*a*::*len*) − *1*) (*numeral x*))
⟨*proof*⟩

lemma *ucast-id*: *ucast w* = *w*
⟨*proof*⟩

lemma *scast-id*: *scast w* = *w*
⟨*proof*⟩

lemma *ucast-bl*: *ucast w* = *of-bl* (*to-bl w*)
⟨*proof*⟩

lemma *nth-ucast*:
(*ucast w*::'*a*::*len0 word*) !! *n* = (*w* !! *n* & *n* < *len-of TYPE*('*a*))
⟨*proof*⟩

lemma *ucast-bintr* [*simp*]:
ucast (*numeral w* ::'*a*::*len0 word*) =
word-of-int (*bintrunc* (*len-of TYPE*('*a*)) (*numeral w*))
⟨*proof*⟩

lemma *scast-sbintr* [*simp*]:
scast (*numeral w* ::'*a*::*len word*) =
word-of-int (*sbintrunc* (*len-of TYPE*('*a*) − *Suc 0*) (*numeral w*))
⟨*proof*⟩

lemma *source-size*: *source-size* (*c*::'*a*::*len0 word* ⇒ -) = *len-of TYPE*('*a*)
⟨*proof*⟩

lemma *target-size*: *target-size* (*c*::- ⇒ '*b*::*len0 word*) = *len-of TYPE*('*b*)
⟨*proof*⟩

lemma *is-down*:
fixes *c* :: *'a::len0 word* ⇒ *'b::len0 word*
shows *is-down c* ⟷ *len-of TYPE('b)* ≤ *len-of TYPE('a)*
⟨*proof*⟩

lemma *is-up*:
fixes *c* :: *'a::len0 word* ⇒ *'b::len0 word*
shows *is-up c* ⟷ *len-of TYPE('a)* ≤ *len-of TYPE('b)*
⟨*proof*⟩

lemmas *is-up-down* = *trans* [*OF is-up is-down* [*symmetric*]]

lemma *down-cast-same* [*OF refl*]: *uc* = *ucast* ⟹ *is-down uc* ⟹ *uc* = *scast*
⟨*proof*⟩

lemma *word-rev-tf*:
to-bl (*of-bl bl::'a::len0 word*) =
rev (*takefill False* (*len-of TYPE('a)*) (*rev bl*))
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *ucast-down-drop* [*OF refl*]:
uc = *ucast* ⟹ *source-size uc* = *target-size uc* + *n* ⟹
to-bl (*uc w*) = *drop n* (*to-bl w*)
⟨*proof*⟩

lemma *scast-down-drop* [*OF refl*]:
sc = *scast* ⟹ *source-size sc* = *target-size sc* + *n* ⟹
to-bl (*sc w*) = *drop n* (*to-bl w*)
⟨*proof*⟩

lemma *sint-up-scast* [*OF refl*]:

sc = scast ⟹ is-up sc ⟹ sint (sc w) = sint w
⟨*proof*⟩

lemma *uint-up-ucast* [*OF refl*]:
uc = ucast ⟹ is-up uc ⟹ uint (uc w) = uint w
⟨*proof*⟩

lemma *ucast-up-ucast* [*OF refl*]:
uc = ucast ⟹ is-up uc ⟹ ucast (uc w) = ucast w
⟨*proof*⟩

lemma *scast-up-scast* [*OF refl*]:
sc = scast ⟹ is-up sc ⟹ scast (sc w) = scast w
⟨*proof*⟩

lemma *ucast-of-bl-up* [*OF refl*]:
w = of-bl bl ⟹ size bl <= size w ⟹ ucast w = of-bl bl
⟨*proof*⟩

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) ⟹
surj (ucast :: 'a word => 'b word)
⟨*proof*⟩

lemma *up-scast-surj*:
is-up (scast :: 'b::len word => 'a::len word) ⟹
surj (scast :: 'a word => 'b word)
⟨*proof*⟩

lemma *down-scast-inj*:
is-down (scast :: 'b::len word => 'a::len word) ⟹
inj-on (ucast :: 'a word => 'b word) A
⟨*proof*⟩

lemma *down-ucast-inj*:
is-down (ucast :: 'b::len0 word => 'a::len0 word) ⟹
inj-on (ucast :: 'a word => 'b word) A
⟨*proof*⟩

lemma *of-bl-append-same*: *of-bl (X @ to-bl w) = w*
⟨*proof*⟩

lemma *ucast-down-wi* [*OF refl*]:
uc = ucast ⟹ *is-down uc* ⟹ *uc* (*word-of-int x*) = *word-of-int x*
⟨*proof*⟩

lemma *ucast-down-no* [*OF refl*]:
uc = ucast ⟹ *is-down uc* ⟹ *uc* (*numeral bin*) = *numeral bin*
⟨*proof*⟩

lemma *ucast-down-bl* [*OF refl*]:
uc = ucast ⟹ *is-down uc* ⟹ *uc* (*of-bl bl*) = *of-bl bl*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *signed-linorder*: *class.linorder word-sle word-sless*
⟨*proof*⟩

interpretation *signed*: *linorder word-sle word-sless*
⟨*proof*⟩

lemma *udvdI*:
0 ≤ *n* ⟹ *uint b* = *n* * *uint a* ⟹ *a udvd b*
⟨*proof*⟩

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-of-int* (− *1*)
⟨*proof*⟩

lemma *word-0-bl* [*simp*]: *of-bl* [] = *0*

⟨*proof*⟩

lemma *word-1-bl*: *of-bl* [*True*] = *1*
⟨*proof*⟩

lemma *uint-eq-0* [*simp*]: *uint 0* = *0*
⟨*proof*⟩

lemma *of-bl-0* [*simp*]: *of-bl* (*replicate n False*) = *0*
⟨*proof*⟩

lemma *to-bl-0* [*simp*]:
to-bl (*0*::*'a*::*len0 word*) = *replicate* (*len-of TYPE*(*'a*)) *False*
⟨*proof*⟩

lemma *uint-0-iff*:
uint x = *0* ⟷ *x* = *0*
⟨*proof*⟩

lemma *unat-0-iff*:
unat x = *0* ⟷ *x* = *0*
⟨*proof*⟩

lemma *unat-0* [*simp*]:
unat 0 = *0*
⟨*proof*⟩

lemma *size-0-same'*:
size w = *0* ⟹ *w* = (*v* :: *'a* :: *len0 word*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *ucast-0* [*simp*]: *ucast 0* = *0*
⟨*proof*⟩

lemma *sint-0* [*simp*]: *sint 0* = *0*
⟨*proof*⟩

lemma *scast-0* [*simp*]: *scast 0* = *0*
⟨*proof*⟩

lemma *sint-n1* [*simp*] : *sint* (− *1*) = − *1*

⟨*proof*⟩

lemma *scast-n1* [*simp*]: *scast* (− *1*) = − *1*
⟨*proof*⟩

lemma *uint-1* [*simp*]: *uint* (*1*::*'a*::*len word*) = *1*
⟨*proof*⟩

lemma *unat-1* [*simp*]: *unat* (*1*::*'a*::*len word*) = *1*
⟨*proof*⟩

lemma *ucast-1* [*simp*]: *ucast* (*1*::*'a*::*len word*) = *1*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *uint-cong*: *x* = *y* ⟹ *uint x* = *uint y*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *word-sp-01* [*simp*] :
word-succ (− *1*) = *0* & *word-succ 0* = *1* & *word-pred 0* = − *1* & *word-pred 1* = *0*
⟨*proof*⟩

lemma *word-of-int-Ex*:
∃ *y*. *x* = *word-of-int y*
⟨*proof*⟩

127.16 Order on fixed-length words

lemma *word-zero-le* [*simp*] :
0 <= (*y* :: *'a* :: *len0 word*)
⟨*proof*⟩

lemma *word-m1-ge* [*simp*] : *word-pred 0* >= *y*
⟨*proof*⟩

lemma *word-n1-ge* [*simp*]: *y* ≤ (−*1*::*'a*::*len0 word*)
⟨*proof*⟩

lemmas *word-not-simps* [*simp*] =
word-zero-le [*THEN leD*] *word-m1-ge* [*THEN leD*] *word-n1-ge* [*THEN leD*]

lemma *word-gt-0*: $0 < y \longleftrightarrow 0 \neq (y :: 'a :: len0\ word)$
⟨*proof*⟩

lemmas *word-gt-0-no* [*simp*] = *word-gt-0* [*of numeral y*] **for** *y*

lemma *word-sless-alt*: $(a <s b) = (sint\ a < sint\ b)$
⟨*proof*⟩

lemma *word-le-nat-alt*: $(a <= b) = (unat\ a <= unat\ b)$
⟨*proof*⟩

lemma *word-less-nat-alt*: $(a < b) = (unat\ a < unat\ b)$
⟨*proof*⟩

lemma *wi-less*:
$(word\text{-}of\text{-}int\ n < (word\text{-}of\text{-}int\ m :: 'a :: len0\ word)) =$
$(n\ mod\ 2\ \hat{}\ len\text{-}of\ TYPE('a) < m\ mod\ 2\ \hat{}\ len\text{-}of\ TYPE('a))$
⟨*proof*⟩

lemma *wi-le*:
$(word\text{-}of\text{-}int\ n <= (word\text{-}of\text{-}int\ m :: 'a :: len0\ word)) =$
$(n\ mod\ 2\ \hat{}\ len\text{-}of\ TYPE('a) <= m\ mod\ 2\ \hat{}\ len\text{-}of\ TYPE('a))$
⟨*proof*⟩

lemma *udvd-nat-alt*: $a\ udvd\ b = (EX\ n>=0.\ unat\ b = n * unat\ a)$
⟨*proof*⟩

lemma *udvd-iff-dvd*: $x\ udvd\ y \longleftrightarrow unat\ x\ dvd\ unat\ y$
⟨*proof*⟩

lemmas *unat-mono* = *word-less-nat-alt* [*THEN iffD1*]

lemma *unat-minus-one*:
assumes $w \neq 0$
shows $unat\ (w - 1) = unat\ w - 1$
⟨*proof*⟩

lemma *measure-unat*: $p\ ^{\sim}= 0 \Longrightarrow unat\ (p - 1) < unat\ p$
⟨*proof*⟩

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*]:
$uint\ (x :: 'a :: len0\ word) - uint\ (y :: 'b :: len0\ word) <$

$2 \hat{} \mathit{len\text{-}of}\ \mathit{TYPE}('a)$
⟨*proof*⟩

127.17 Conditions for the addition (etc) of two words to overflow

lemma *uint-add-lem*:
$(\mathit{uint}\ x + \mathit{uint}\ y < 2 \hat{} \mathit{len\text{-}of}\ \mathit{TYPE}('a)) =$
$(\mathit{uint}\ (x + y :: 'a :: \mathit{len0}\ \mathit{word}) = \mathit{uint}\ x + \mathit{uint}\ y)$
⟨*proof*⟩

lemma *uint-mult-lem*:
$(\mathit{uint}\ x * \mathit{uint}\ y < 2 \hat{} \mathit{len\text{-}of}\ \mathit{TYPE}('a)) =$
$(\mathit{uint}\ (x * y :: 'a :: \mathit{len0}\ \mathit{word}) = \mathit{uint}\ x * \mathit{uint}\ y)$
⟨*proof*⟩

lemma *uint-sub-lem*:
$(\mathit{uint}\ x >= \mathit{uint}\ y) = (\mathit{uint}\ (x - y) = \mathit{uint}\ x - \mathit{uint}\ y)$
⟨*proof*⟩

lemma *uint-add-le*: $\mathit{uint}\ (x + y) <= \mathit{uint}\ x + \mathit{uint}\ y$
⟨*proof*⟩

lemma *uint-sub-ge*: $\mathit{uint}\ (x - y) >= \mathit{uint}\ x - \mathit{uint}\ y$
⟨*proof*⟩

lemma *mod-add-if-z*:
$(x :: \mathit{int}) < z ==> y < z ==> 0 <= y ==> 0 <= x ==> 0 <= z ==>$
$(x + y)\ \mathit{mod}\ z = (\mathit{if}\ x + y < z\ \mathit{then}\ x + y\ \mathit{else}\ x + y - z)$
⟨*proof*⟩

lemma *uint-plus-if'*:
$\mathit{uint}\ ((a::'a\ \mathit{word}) + b) =$
$(\mathit{if}\ \mathit{uint}\ a + \mathit{uint}\ b < 2 \hat{} \mathit{len\text{-}of}\ \mathit{TYPE}('a::\mathit{len0})\ \mathit{then}\ \mathit{uint}\ a + \mathit{uint}\ b$
$\mathit{else}\ \mathit{uint}\ a + \mathit{uint}\ b - 2 \hat{} \mathit{len\text{-}of}\ \mathit{TYPE}('a))$
⟨*proof*⟩

lemma *mod-sub-if-z*:
$(x :: \mathit{int}) < z ==> y < z ==> 0 <= y ==> 0 <= x ==> 0 <= z ==>$
$(x - y)\ \mathit{mod}\ z = (\mathit{if}\ y <= x\ \mathit{then}\ x - y\ \mathit{else}\ x - y + z)$
⟨*proof*⟩

lemma *uint-sub-if'*:
$\mathit{uint}\ ((a::'a\ \mathit{word}) - b) =$
$(\mathit{if}\ \mathit{uint}\ b \leq \mathit{uint}\ a\ \mathit{then}\ \mathit{uint}\ a - \mathit{uint}\ b$
$\mathit{else}\ \mathit{uint}\ a - \mathit{uint}\ b + 2 \hat{} \mathit{len\text{-}of}\ \mathit{TYPE}('a::\mathit{len0}))$
⟨*proof*⟩

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
⟨*proof*⟩

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)
⟨*proof*⟩

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))
⟨*proof*⟩

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*
⟨*proof*⟩

⟨*ML*⟩

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)
⟨*proof*⟩

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)*
⟨*proof*⟩

lemma *no-olen-add'*:
fixes *x :: 'a::len0 word*
shows *(x ≤ y + x) = (uint y + uint x < 2 ^ len-of TYPE('a))*
⟨*proof*⟩

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*) ~= *0* ⟹ (*1* < *x*) = (*0* < *x* − *1*)
⟨*proof*⟩

lemma *word-le-sub1*:
(*x* :: *'a* :: *len word*) ~= *0* ⟹ (*1* <= *x*) = (*0* <= *x* − *1*)
⟨*proof*⟩

lemma *sub-wrap-lt*:
((*x* :: *'a* :: *len0 word*) < *x* − *z*) = (*x* < *z*)
⟨*proof*⟩

lemma *sub-wrap*:
((*x* :: *'a* :: *len0 word*) <= *x* − *z*) = (*z* = *0* | *x* < *z*)
⟨*proof*⟩

lemma *plus-minus-not-NULL-ab*:
(*x* :: *'a* :: *len0 word*) <= *ab* − *c* ⟹ *c* <= *ab* ⟹ *c* ~= *0* ⟹ *x* + *c* ~= *0*
⟨*proof*⟩

lemma *plus-minus-no-overflow-ab*:
(*x* :: *'a* :: *len0 word*) <= *ab* − *c* ⟹ *c* <= *ab* ⟹ *x* <= *x* + *c*
⟨*proof*⟩

lemma *le-minus'*:
(*a* :: *'a* :: *len0 word*) + *c* <= *b* ⟹ *a* <= *a* + *c* ⟹ *c* <= *b* − *a*
⟨*proof*⟩

lemma *le-plus'*:
(*a* :: *'a* :: *len0 word*) <= *b* ⟹ *c* <= *b* − *a* ⟹ *a* + *c* <= *b*
⟨*proof*⟩

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* ⟹ *x* <= *x* + *z* ⟹ *x* + *y* <= *x* + *z*
⟨*proof*⟩

lemma *word-less-minus-cancel*:

$y - x < z - x \Longrightarrow x <= z \Longrightarrow (y :: 'a :: len0\ word) < z$
⟨*proof*⟩

lemma *word-less-minus-mono-left*:
$(y :: 'a :: len0\ word) < z \Longrightarrow x <= y \Longrightarrow y - x < z - x$
⟨*proof*⟩

lemma *word-less-minus-mono*:
$a < c \Longrightarrow d < b \Longrightarrow a - b < a \Longrightarrow c - d < c$
$\Longrightarrow a - b < c - (d::'a::len\ word)$
⟨*proof*⟩

lemma *word-le-minus-cancel*:
$y - x <= z - x \Longrightarrow x <= z \Longrightarrow (y :: 'a :: len0\ word) <= z$
⟨*proof*⟩

lemma *word-le-minus-mono-left*:
$(y :: 'a :: len0\ word) <= z \Longrightarrow x <= y \Longrightarrow y - x <= z - x$
⟨*proof*⟩

lemma *word-le-minus-mono*:
$a <= c \Longrightarrow d <= b \Longrightarrow a - b <= a \Longrightarrow c - d <= c$
$\Longrightarrow a - b <= c - (d::'a::len\ word)$
⟨*proof*⟩

lemma *plus-le-left-cancel-wrap*:
$(x :: 'a :: len0\ word) + y' < x \Longrightarrow x + y < x \Longrightarrow (x + y' < x + y) = (y' < y)$
⟨*proof*⟩

lemma *plus-le-left-cancel-nowrap*:
$(x :: 'a :: len0\ word) <= x + y' \Longrightarrow x <= x + y \Longrightarrow$
$(x + y' < x + y) = (y' < y)$
⟨*proof*⟩

lemma *word-plus-mono-right2*:
$(a :: 'a :: len0\ word) <= a + b \Longrightarrow c <= b \Longrightarrow a <= a + c$
⟨*proof*⟩

lemma *word-less-add-right*:
$(x :: 'a :: len0\ word) < y - z \Longrightarrow z <= y \Longrightarrow x + z < y$
⟨*proof*⟩

lemma *word-less-sub-right*:
$(x :: 'a :: len0\ word) < y + z \Longrightarrow y <= x \Longrightarrow x - y < z$
⟨*proof*⟩

lemma *word-le-plus-either*:
$(x :: 'a :: len0\ word) <= y \mid x <= z \Longrightarrow y <= y + z \Longrightarrow x <= y + z$
⟨*proof*⟩

lemma *word-less-nowrapI*:
$(x :: 'a :: len0\ word) < z - k \Longrightarrow k <= z \Longrightarrow 0 < k \Longrightarrow x < x + k$
⟨*proof*⟩

lemma *inc-le*: $(i :: 'a :: len\ word) < m \Longrightarrow i + 1 <= m$
⟨*proof*⟩

lemma *inc-i*:
$(1 :: 'a :: len\ word) <= i \Longrightarrow i < m \Longrightarrow 1 <= (i + 1)\ \&\ i + 1 <= m$
⟨*proof*⟩

lemma *udvd-incr-lem*:
$up < uq \Longrightarrow up = ua + n * uint\ K \Longrightarrow$
$uq = ua + n' * uint\ K \Longrightarrow up + uint\ K <= uq$
⟨*proof*⟩

lemma *udvd-incr'*:
$p < q \Longrightarrow uint\ p = ua + n * uint\ K \Longrightarrow$
$uint\ q = ua + n' * uint\ K \Longrightarrow p + K <= q$
⟨*proof*⟩

lemma *udvd-decr'*:
$p < q \Longrightarrow uint\ p = ua + n * uint\ K \Longrightarrow$
$uint\ q = ua + n' * uint\ K \Longrightarrow p <= q - K$
⟨*proof*⟩

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'*:
$xy < k \Longrightarrow z\ udvd\ xy \Longrightarrow z\ udvd\ k \Longrightarrow xy <= k - z$
⟨*proof*⟩

lemma *udvd-incr2-K*:
$p < a + s \Longrightarrow a <= a + s \Longrightarrow K\ udvd\ s \Longrightarrow K\ udvd\ p - a \Longrightarrow a <= p \Longrightarrow$
$0 < K \Longrightarrow p <= p + K\ \&\ p + K <= a + s$
⟨*proof*⟩

lemma *word-succ-rbl*:
$to\text{-}bl\ w = bl \Longrightarrow to\text{-}bl\ (word\text{-}succ\ w) = (rev\ (rbl\text{-}succ\ (rev\ bl)))$
⟨*proof*⟩

lemma *word-pred-rbl*:
$to\text{-}bl\ w = bl \Longrightarrow to\text{-}bl\ (word\text{-}pred\ w) = (rev\ (rbl\text{-}pred\ (rev\ bl)))$
⟨*proof*⟩

lemma *word-add-rbl*:
to-bl v = vbl ⟹ to-bl w = wbl ⟹
to-bl (v + w) = (rev (rbl-add (rev vbl) (rev wbl)))
⟨*proof*⟩

lemma *word-mult-rbl*:
to-bl v = vbl ⟹ to-bl w = wbl ⟹
*to-bl (v * w) = (rev (rbl-mult (rev vbl) (rev wbl)))*
⟨*proof*⟩

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
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *iszero-word-no* [*simp*]:
iszero (numeral bin :: 'a :: len word) =
iszero (bintrunc (len-of TYPE('a)) (numeral bin))
⟨*proof*⟩

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 ^ n)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*) ⟹ *y* : *unats* (*len-of TYPE* ('*a*))
⟨*proof*⟩

lemma *word-nchotomy*:
ALL w. *EX n*. (*w* :: '*a* :: *len word*) = *of-nat n* & *n* < 2 ^ *len-of TYPE* ('*a*)
⟨*proof*⟩

lemma *of-nat-eq*:
fixes *w* :: '*a*::*len word*
shows (*of-nat n* = *w*) = (∃ *q*. *n* = *unat w* + *q* * 2 ^ *len-of TYPE*('*a*))
⟨*proof*⟩

lemma *of-nat-eq-size*:
(*of-nat n* = *w*) = (*EX q*. *n* = *unat w* + *q* * 2 ^ *size w*)
⟨*proof*⟩

lemma *of-nat-0*:
(*of-nat m* = (*0*::'*a*::*len word*)) = (∃ *q*. *m* = *q* * 2 ^ *len-of TYPE*('*a*))
⟨*proof*⟩

lemma *of-nat-2p* [*simp*]:
of-nat (2 ^ *len-of TYPE*('*a*)) = (*0*::'*a*::*len word*)
⟨*proof*⟩

lemma *of-nat-gt-0*: *of-nat k* ~= *0* ⟹ *0* < *k*
⟨*proof*⟩

lemma *of-nat-neq-0*:
0 < *k* ⟹ *k* < 2 ^ *len-of TYPE* ('*a* :: *len*) ⟹ *of-nat k* ~= (*0* :: '*a word*)
⟨*proof*⟩

lemma *Abs-fnat-hom-add*:
of-nat a + *of-nat b* = *of-nat* (*a* + *b*)
⟨*proof*⟩

lemma *Abs-fnat-hom-mult*:
of-nat a * *of-nat b* = (*of-nat* (*a* * *b*) :: '*a* :: *len word*)
⟨*proof*⟩

lemma *Abs-fnat-hom-Suc*:
word-succ (*of-nat a*) = *of-nat* (*Suc a*)

⟨*proof*⟩

lemma *Abs-fnat-hom-0*: $(0::'a::len\ word) = of\text{-}nat\ 0$
⟨*proof*⟩

lemma *Abs-fnat-hom-1*: $(1::'a::len\ word) = of\text{-}nat\ (Suc\ 0)$
⟨*proof*⟩

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*:
$a + b = of\text{-}nat\ (unat\ a + unat\ b)$
⟨*proof*⟩

lemma *word-arith-nat-mult*:
$a * b = of\text{-}nat\ (unat\ a * unat\ b)$
⟨*proof*⟩

lemma *word-arith-nat-Suc*:
$word\text{-}succ\ a = of\text{-}nat\ (Suc\ (unat\ a))$
⟨*proof*⟩

lemma *word-arith-nat-div*:
$a\ div\ b = of\text{-}nat\ (unat\ a\ div\ unat\ b)$
⟨*proof*⟩

lemma *word-arith-nat-mod*:
$a\ mod\ b = of\text{-}nat\ (unat\ a\ mod\ unat\ b)$
⟨*proof*⟩

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*: $x = y \Longrightarrow unat\ x = unat\ y$
⟨*proof*⟩

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*:
$(unat\ x + unat\ y < 2\ \hat{}\ len\text{-}of\ TYPE('a)) =$

(*unat* (*x* + *y* :: 'a :: *len word*) = *unat x* + *unat y*)
⟨*proof*⟩

lemma *unat-mult-lem*:
(*unat x* * *unat y* < 2 ^ *len-of TYPE*('a)) =
(*unat* (*x* * *y* :: 'a :: *len word*) = *unat x* * *unat y*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *unat-mod*: *unat* ((*x* :: 'a :: *len word*) *mod y*) = *unat x mod unat y*
⟨*proof*⟩

lemma *uint-div*: *uint* ((*x* :: 'a :: *len word*) *div y*) = *uint x div uint y*
⟨*proof*⟩

lemma *uint-mod*: *uint* ((*x* :: 'a :: *len word*) *mod y*) = *uint x mod uint y*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*))
⟨*proof*⟩

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*⟩

lemma *no-plus-overflow-unat-size*:
((*x* :: *'a* :: *len word*) <= *x* + *y*) = (*unat x* + *unat y* < *2* ^ *size x*)
⟨*proof*⟩

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* ⟹ *unat x* * *unat y* < *2* ^ *len-of TYPE*(*'a*) ⟹
x * *y div y* = *x*
⟨*proof*⟩

lemma *div-lt'*: (*i* :: *'a* :: *len word*) <= *k div x* ⟹
unat i * *unat x* < *2* ^ *len-of TYPE*(*'a*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *div-le-mult*:
(*i* :: *'a* :: *len word*) <= *k div x* ⟹ *0* < *x* ⟹ *i* * *x* <= *k*
⟨*proof*⟩

lemma *div-lt-uint'*:
(*i* :: *'a* :: *len word*) <= *k div x* ⟹ *uint i* * *uint x* < *2* ^ *len-of TYPE*(*'a*)
⟨*proof*⟩

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*))

⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *word-div-mult-le*: *a div b* * *b* <= (*a*::*'a*::*len word*)
⟨*proof*⟩

lemma *word-mod-less-divisor*: *0* < *n* ⟹ *m mod n* < (*n* :: *'a* :: *len word*)
⟨*proof*⟩

lemma *word-of-int-power-hom*:
word-of-int a ^ *n* = (*word-of-int* (*a* ^ *n*) :: *'a* :: *len word*)
⟨*proof*⟩

lemma *word-arith-power-alt*:
a ^ *n* = (*word-of-int* (*uint a* ^ *n*) :: *'a* :: *len word*)
⟨*proof*⟩

lemma *of-bl-length-less*:
length x = *k* ⟹ *k* < *len-of TYPE*(*'a*) ⟹ (*of-bl x* :: *'a* :: *len word*) < *2* ^ *k*
⟨*proof*⟩

127.23 Cardinality, finiteness of set of words

instance *word* :: (*len0*) *finite*
⟨*proof*⟩

lemma *card-word*: *CARD*(*'a*::*len0 word*) = *2* ^ *len-of TYPE*(*'a*)

⟨*proof*⟩

lemma *card-word-size*:
card (*UNIV* :: 'a :: *len0 word set*) = (2 ^ *size* (*x* :: 'a *word*))
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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)
⟨*proof*⟩

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
⟨*proof*⟩

lemma *uint-or*: *uint (x OR y) = (uint x) OR (uint y)*
⟨*proof*⟩

lemma *uint-and*: *uint (x AND y) = (uint x) AND (uint y)*
⟨*proof*⟩

lemma *test-bit-wi* [*simp*]:
(word-of-int x::'a::len0 word) !! n ⟷ n < len-of TYPE('a) ∧ bin-nth x n
⟨*proof*⟩

lemma *word-test-bit-transfer* [*transfer-rule*]:
(rel-fun pcr-word (rel-fun op = op =))
(λx n. n < len-of TYPE('a) ∧ bin-nth x n) (test-bit :: 'a::len0 word ⇒ -)
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *word-ao-nth*:
fixes *x :: 'a::len0 word*
shows *(x OR y) !! n = (x !! n | y !! n) &*

$(x\ AND\ y)\ !!\ n = (x\ !!\ n\ \&\ y\ !!\ n)$
⟨*proof*⟩

lemma *test-bit-numeral* [*simp*]:
$(numeral\ w :: 'a{::}len0\ word)\ !!\ n \longleftrightarrow$
$n < len\text{-}of\ TYPE('a) \wedge bin\text{-}nth\ (numeral\ w)\ n$
⟨*proof*⟩

lemma *test-bit-neg-numeral* [*simp*]:
$(-\ numeral\ w :: 'a{::}len0\ word)\ !!\ n \longleftrightarrow$
$n < len\text{-}of\ TYPE('a) \wedge bin\text{-}nth\ (-\ numeral\ w)\ n$
⟨*proof*⟩

lemma *test-bit-1* [*simp*]: $(1{::}'a{::}len\ word)\ !!\ n \longleftrightarrow n = 0$
⟨*proof*⟩

lemma *nth-0* [*simp*]: $\sim (0{::}'a{::}len0\ word)\ !!\ n$
⟨*proof*⟩

lemma *nth-minus1* [*simp*]: $(-1{::}'a{::}len0\ word)\ !!\ n \longleftrightarrow n < len\text{-}of\ TYPE('a)$
⟨*proof*⟩

lemmas *bwsimps* =
wi-hom-add
word-wi-log-defs

lemma *word-bw-assocs*:
fixes $x :: 'a{::}len0\ word$
shows
$(x\ AND\ y)\ AND\ z = x\ AND\ y\ AND\ z$
$(x\ OR\ y)\ OR\ z = x\ OR\ y\ OR\ z$
$(x\ XOR\ y)\ XOR\ z = x\ XOR\ y\ XOR\ z$
⟨*proof*⟩

lemma *word-bw-comms*:
fixes $x :: 'a{::}len0\ word$
shows
$x\ AND\ y = y\ AND\ x$
$x\ OR\ y = y\ OR\ x$
$x\ XOR\ y = y\ XOR\ x$
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

lemma *word-bw-same*:
fixes *x* :: *'a*::*len0 word*
shows
x AND x = x
x OR x = x
x XOR x = 0
⟨*proof*⟩

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
⟨*proof*⟩

lemma *word-not-not* [*simp*]:

NOT NOT (*x*::'*a*::*len0 word*) = *x*
⟨*proof*⟩

lemma *word-ao-dist*:
fixes *x* :: '*a*::*len0 word*
shows (*x OR y*) *AND z* = *x AND z OR y AND z*
⟨*proof*⟩

lemma *word-oa-dist*:
fixes *x* :: '*a*::*len0 word*
shows *x AND y OR z* = (*x OR z*) *AND* (*y OR z*)
⟨*proof*⟩

lemma *word-add-not* [*simp*]:
fixes *x* :: '*a*::*len0 word*
shows *x* + *NOT x* = −*1*
⟨*proof*⟩

lemma *word-plus-and-or* [*simp*]:
fixes *x* :: '*a*::*len0 word*
shows (*x AND y*) + (*x OR y*) = *x* + *y*
⟨*proof*⟩

lemma *leoa*:
fixes *x* :: '*a*::*len0 word*
shows (*w* = (*x OR y*)) ⟹ (*y* = (*w AND y*)) ⟨*proof*⟩
lemma *leao*:
fixes *x'* :: '*a*::*len0 word*
shows (*w'* = (*x' AND y'*)) ⟹ (*x'* = (*x' OR w'*)) ⟨*proof*⟩

lemma *word-ao-equiv*:
fixes *w w'* :: '*a*::*len0 word*
shows (*w* = *w OR w'*) = (*w'* = *w AND w'*)
⟨*proof*⟩

lemma *le-word-or2*: *x* <= *x OR* (*y*::'*a*::*len0 word*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *bl-word-xor*: *to-bl* (*v XOR w*) = *map2 op* ~= (*to-bl v*) (*to-bl w*)
⟨*proof*⟩

lemma *bl-word-or*: *to-bl* (*v OR w*) = *map2 op* | (*to-bl v*) (*to-bl w*)

⟨*proof*⟩

lemma *bl-word-and*: *to-bl* (*v AND w*) = *map2 op* & (*to-bl v*) (*to-bl w*)
⟨*proof*⟩

lemma *word-lsb-alt*: *lsb* (*w*::*'a*::*len0 word*) = *test-bit w 0*
⟨*proof*⟩

lemma *word-lsb-1-0* [*simp*]: *lsb* (*1*::*'a*::*len word*) & ~ *lsb* (*0*::*'b*::*len0 word*)
⟨*proof*⟩

lemma *word-lsb-last*: *lsb* (*w*::*'a*::*len word*) = *last* (*to-bl w*)
⟨*proof*⟩

lemma *word-lsb-int*: *lsb w* = (*uint w mod 2* = *1*)
⟨*proof*⟩

lemma *word-msb-sint*: *msb w* = (*sint w* < *0*)
⟨*proof*⟩

lemma *msb-word-of-int*:
msb (*word-of-int x*::*'a*::*len word*) = *bin-nth x* (*len-of TYPE*(*'a*) − *1*)
⟨*proof*⟩

lemma *word-msb-numeral* [*simp*]:
msb (*numeral w*::*'a*::*len word*) = *bin-nth* (*numeral w*) (*len-of TYPE*(*'a*) − *1*)
⟨*proof*⟩

lemma *word-msb-neg-numeral* [*simp*]:
msb (− *numeral w*::*'a*::*len word*) = *bin-nth* (− *numeral w*) (*len-of TYPE*(*'a*) − *1*)
⟨*proof*⟩

lemma *word-msb-0* [*simp*]: ¬ *msb* (*0*::*'a*::*len word*)
⟨*proof*⟩

lemma *word-msb-1* [*simp*]: *msb* (*1*::*'a*::*len word*) ⟷ *len-of TYPE*(*'a*) = *1*
⟨*proof*⟩

lemma *word-msb-nth*:
msb (*w*::*'a*::*len word*) = *bin-nth* (*uint w*) (*len-of TYPE*(*'a*) − *1*)
⟨*proof*⟩

lemma *word-msb-alt*: *msb* (*w*::*'a*::*len word*) = *hd* (*to-bl w*)
⟨*proof*⟩

lemma *word-set-nth* [*simp*]:
set-bit w n (*test-bit w n*) = (*w*::*'a*::*len0 word*)
⟨*proof*⟩

lemma *bin-nth-uint'*:
bin-nth (*uint w*) *n* = (*rev* (*bin-to-bl* (*size w*) (*uint w*)) ! *n* & *n* < *size w*)
⟨*proof*⟩

lemmas *bin-nth-uint* = *bin-nth-uint'* [*unfolded word-size*]

lemma *test-bit-bl*: *w* !! *n* = (*rev* (*to-bl w*) ! *n* & *n* < *size w*)
⟨*proof*⟩

lemma *to-bl-nth*: *n* < *size w* ⟹ *to-bl w* ! *n* = *w* !! (*size w* − *Suc n*)
⟨*proof*⟩

lemma *test-bit-set*:
fixes *w* :: 'a::*len0 word*
shows (*set-bit w n x*) !! *n* = (*n* < *size w* & *x*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *of-bl-rep-False*: *of-bl* (*replicate n False* @ *bs*) = *of-bl bs*
⟨*proof*⟩

lemma *msb-nth*:
fixes *w* :: 'a::*len word*
shows *msb w* = *w* !! (*len-of TYPE*('a) − *1*)
⟨*proof*⟩

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*)
⟨*proof*⟩

interpretation *test-bit*:
td-ext op !! :: 'a::*len0 word* => *nat* => *bool*
set-bits
{*f*. ∀ *i*. *f i* ⟶ *i* < *len-of TYPE*('a::*len0*)}

$(\lambda h\ i.\ h\ i \wedge i < \textit{len-of}\ \textit{TYPE}('a::\textit{len0}))$
⟨*proof*⟩

lemmas *td-nth* = *test-bit.td-thm*

lemma *word-set-set-same* [*simp*]:
fixes $w :: 'a::\textit{len0}\ \textit{word}$
shows $\textit{set-bit}\ (\textit{set-bit}\ w\ n\ x)\ n\ y = \textit{set-bit}\ w\ n\ y$
⟨*proof*⟩

lemma *word-set-set-diff*:
fixes $w :: 'a::\textit{len0}\ \textit{word}$
assumes $m \neq n$
shows $\textit{set-bit}\ (\textit{set-bit}\ w\ m\ x)\ n\ y = \textit{set-bit}\ (\textit{set-bit}\ w\ n\ y)\ m\ x$
⟨*proof*⟩

lemma *nth-sint*:
fixes $w :: 'a::\textit{len}\ \textit{word}$
defines $l \equiv \textit{len-of}\ \textit{TYPE}\ ('a)$
shows $\textit{bin-nth}\ (\textit{sint}\ w)\ n = (\textit{if}\ n < l - 1\ \textit{then}\ w\ !!\ n\ \textit{else}\ w\ !!\ (l - 1))$
⟨*proof*⟩

lemma *word-lsb-numeral* [*simp*]:
$\textit{lsb}\ (\textit{numeral bin} :: 'a :: \textit{len word}) \longleftrightarrow \textit{bin-last}\ (\textit{numeral bin})$
⟨*proof*⟩

lemma *word-lsb-neg-numeral* [*simp*]:
$\textit{lsb}\ (-\ \textit{numeral bin} :: 'a :: \textit{len word}) \longleftrightarrow \textit{bin-last}\ (-\ \textit{numeral bin})$
⟨*proof*⟩

lemma *set-bit-word-of-int*:
$\textit{set-bit}\ (\textit{word-of-int}\ x)\ n\ b = \textit{word-of-int}\ (\textit{bin-sc}\ n\ b\ x)$
⟨*proof*⟩

lemma *word-set-numeral* [*simp*]:
$\textit{set-bit}\ (\textit{numeral bin}::'a::\textit{len0 word})\ n\ b =$
$\quad \textit{word-of-int}\ (\textit{bin-sc}\ n\ b\ (\textit{numeral bin}))$
⟨*proof*⟩

lemma *word-set-neg-numeral* [*simp*]:
$\textit{set-bit}\ (-\ \textit{numeral bin}::'a::\textit{len0 word})\ n\ b =$
$\quad \textit{word-of-int}\ (\textit{bin-sc}\ n\ b\ (-\ \textit{numeral bin}))$
⟨*proof*⟩

lemma *word-set-bit-0* [*simp*]:
$\textit{set-bit}\ 0\ n\ b = \textit{word-of-int}\ (\textit{bin-sc}\ n\ b\ 0)$
⟨*proof*⟩

lemma *word-set-bit-1* [*simp*]:

set-bit 1 n b = word-of-int (bin-sc n b 1)
⟨*proof*⟩

lemma *setBit-no* [*simp*]:
setBit (numeral bin) n = word-of-int (bin-sc n True (numeral bin))
⟨*proof*⟩

lemma *clearBit-no* [*simp*]:
clearBit (numeral bin) n = word-of-int (bin-sc n False (numeral bin))
⟨*proof*⟩

lemma *to-bl-n1*:
to-bl (−1::'a::len0 word) = replicate (len-of TYPE ('a)) True
⟨*proof*⟩

lemma *word-msb-n1* [*simp*]: *msb (−1::'a::len word)*
⟨*proof*⟩

lemma *word-set-nth-iff*:
(set-bit w n b = w) = (w !! n = b | n >= size (w::'a::len0 word))
⟨*proof*⟩

lemma *test-bit-2p*:
(word-of-int (2 ^ n)::'a::len word) !! m ⟷ m = n ∧ m < len-of TYPE('a)
⟨*proof*⟩

lemma *nth-w2p*:
((2::'a::len word) ^ n) !! m ⟷ m = n ∧ m < len-of TYPE('a::len)
⟨*proof*⟩

lemma *uint-2p*:
(0::'a::len word) < 2 ^ n ⟹ uint (2 ^ n::'a::len word) = 2 ^ n
⟨*proof*⟩

lemma *word-of-int-2p*: *(word-of-int (2 ^ n) :: 'a :: len word) = 2 ^ n*
⟨*proof*⟩

lemma *bang-is-le*: *x !! m ⟹ 2 ^ m <= (x :: 'a :: len word)*
⟨*proof*⟩

lemma *word-clr-le*:
fixes *w :: 'a::len0 word*
shows *w >= set-bit w n False*
⟨*proof*⟩

lemma *word-set-ge*:
fixes *w :: 'a::len word*
shows *w <= set-bit w n True*
⟨*proof*⟩

127.25 Shifting, Rotating, and Splitting Words

lemma *shiftl1-wi* [*simp*]: *shiftl1* (*word-of-int w*) = *word-of-int* (*w BIT False*)
⟨*proof*⟩

lemma *shiftl1-numeral* [*simp*]:
shiftl1 (*numeral w*) = *numeral* (*Num.Bit0 w*)
⟨*proof*⟩

lemma *shiftl1-neg-numeral* [*simp*]:
shiftl1 (− *numeral w*) = − *numeral* (*Num.Bit0 w*)
⟨*proof*⟩

lemma *shiftl1-0* [*simp*] : *shiftl1 0* = *0*
⟨*proof*⟩

lemma *shiftl1-def-u*: *shiftl1 w* = *word-of-int* (*uint w BIT False*)
⟨*proof*⟩

lemma *shiftl1-def-s*: *shiftl1 w* = *word-of-int* (*sint w BIT False*)
⟨*proof*⟩

lemma *shiftr1-0* [*simp*]: *shiftr1 0* = *0*
⟨*proof*⟩

lemma *sshiftr1-0* [*simp*]: *sshiftr1 0* = *0*
⟨*proof*⟩

lemma *sshiftr1-n1* [*simp*] : *sshiftr1* (− *1*) = − *1*
⟨*proof*⟩

lemma *shiftl-0* [*simp*] : (*0*::*'a*::*len0 word*) << *n* = *0*
⟨*proof*⟩

lemma *shiftr-0* [*simp*] : (*0*::*'a*::*len0 word*) >> *n* = *0*
⟨*proof*⟩

lemma *sshiftr-0* [*simp*] : *0* >>> *n* = *0*
⟨*proof*⟩

lemma *sshiftr-n1* [*simp*] : *−1* >>> *n* = *−1*
⟨*proof*⟩

lemma *nth-shiftl1*: *shiftl1 w* !! *n* = (*n* < *size w* & *n* > *0* & *w* !! (*n* − *1*))
⟨*proof*⟩

lemma *nth-shiftl'* [*rule-format*]:
ALL n. ((*w*::*'a*::*len0 word*) << *m*) !! *n* = (*n* < *size w* & *n* >= *m* & *w* !! (*n* − *m*))
⟨*proof*⟩

lemmas *nth-shiftl* = *nth-shiftl'* [*unfolded word-size*]

lemma *nth-shiftr1*: *shiftr1 w* !! *n* = *w* !! *Suc n*
⟨*proof*⟩

lemma *nth-shiftr*:
⋀*n*. ((*w*::'*a*::*len0 word*) >> *m*) !! *n* = *w* !! (*n* + *m*)
⟨*proof*⟩

lemma *uint-shiftr1*: *uint* (*shiftr1 w*) = *bin-rest* (*uint w*)
⟨*proof*⟩

lemma *nth-sshiftr1*:
sshiftr1 w !! *n* = (*if n* = *size w* − *1 then w* !! *n else w* !! *Suc n*)
⟨*proof*⟩

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*)))
⟨*proof*⟩

lemma *shiftr1-div-2*: *uint* (*shiftr1 w*) = *uint w div 2*
⟨*proof*⟩

lemma *sshiftr1-div-2*: *sint* (*sshiftr1 w*) = *sint w div 2*
⟨*proof*⟩

lemma *shiftr-div-2n*: *uint* (*shiftr w n*) = *uint w div 2* ^ *n*
⟨*proof*⟩

lemma *sshiftr-div-2n*: *sint* (*sshiftr w n*) = *sint w div 2* ^ *n*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *shiftl1-of-bl*: *shiftl1* (*of-bl bl*) = *of-bl* (*bl* @ [*False*])
⟨*proof*⟩

lemma *shiftl1-bl*: *shiftl1* (*w*::'*a*::*len0 word*) = *of-bl* (*to-bl w* @ [*False*])
⟨*proof*⟩

lemma *bl-shiftl1*:
to-bl (*shiftl1* (*w* :: *'a* :: *len word*)) = *tl* (*to-bl w*) @ [*False*]
⟨*proof*⟩

lemma *bl-shiftl1'*:
to-bl (*shiftl1 w*) = *tl* (*to-bl w* @ [*False*])
⟨*proof*⟩

lemma *shiftr1-bl*: *shiftr1 w* = *of-bl* (*butlast* (*to-bl w*))
⟨*proof*⟩

lemma *bl-shiftr1*:
to-bl (*shiftr1* (*w* :: *'a* :: *len word*)) = *False* # *butlast* (*to-bl w*)
⟨*proof*⟩

lemma *bl-shiftr1'*:
to-bl (*shiftr1 w*) = *butlast* (*False* # *to-bl w*)
⟨*proof*⟩

lemma *shiftl1-rev*:
shiftl1 w = *word-reverse* (*shiftr1* (*word-reverse w*))
⟨*proof*⟩

lemma *shiftl-rev*:
shiftl w n = *word-reverse* (*shiftr* (*word-reverse w*) *n*)
⟨*proof*⟩

lemma *rev-shiftl*: *word-reverse w* << *n* = *word-reverse* (*w* >> *n*)
⟨*proof*⟩

lemma *shiftr-rev*: *w* >> *n* = *word-reverse* (*word-reverse w* << *n*)
⟨*proof*⟩

lemma *rev-shiftr*: *word-reverse w* >> *n* = *word-reverse* (*w* << *n*)
⟨*proof*⟩

lemma *bl-sshiftr1*:
to-bl (*sshiftr1* (*w* :: *'a* :: *len word*)) = *hd* (*to-bl w*) # *butlast* (*to-bl w*)
⟨*proof*⟩

lemma *drop-shiftr*:
drop n (*to-bl* ((*w* :: *'a* :: *len word*) >> *n*)) = *take* (*size w* − *n*) (*to-bl w*)
⟨*proof*⟩

lemma *drop-sshiftr*:
drop n (*to-bl* ((*w* :: *'a* :: *len word*) >>> *n*)) = *take* (*size w* − *n*) (*to-bl w*)

⟨*proof*⟩

lemma *take-shiftr*:
n ≤ *size w* ⟹ *take n* (*to-bl* (*w* >> *n*)) = *replicate n False*
⟨*proof*⟩

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*))
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *shiftl-bl*:
(*w*::'*a*::*len0 word*) << (*n*::*nat*) = *of-bl* (*to-bl w* @ *replicate n False*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *shiftl-zero-size*:
fixes *x* :: '*a*::*len0 word*
shows *size x* <= *n* ⟹ *x* << *n* = *0*
⟨*proof*⟩

lemma *shiftl1-2t*: *shiftl1* (*w* :: 'a :: *len word*) = *2* ∗ *w*
⟨*proof*⟩

lemma *shiftl1-p*: *shiftl1* (*w* :: 'a :: *len word*) = *w* + *w*
⟨*proof*⟩

lemma *shiftl-t2n*: *shiftl* (*w* :: 'a :: *len word*) *n* = *2* ^ *n* ∗ *w*
⟨*proof*⟩

lemma *shiftr1-bintr* [*simp*]:
(*shiftr1* (*numeral w*) :: *'a* :: *len0 word*) =
word-of-int (*bin-rest* (*bintrunc* (*len-of TYPE* (*'a*)) (*numeral w*)))
⟨*proof*⟩

lemma *sshiftr1-sbintr* [*simp*]:
(*sshiftr1* (*numeral w*) :: *'a* :: *len word*) =
word-of-int (*bin-rest* (*sbintrunc* (*len-of TYPE* (*'a*) − *1*) (*numeral w*)))
⟨*proof*⟩

lemma *shiftr-no* [*simp*]:

(*numeral w*::*'a*::*len0 word*) >> *n* = *word-of-int*
((*bin-rest* ^^ *n*) (*bintrunc* (*len-of TYPE*(*'a*)) (*numeral w*)))
⟨*proof*⟩

lemma *sshiftr-no* [*simp*]:

(*numeral w*::*'a*::*len word*) >>> *n* = *word-of-int*
((*bin-rest* ^^ *n*) (*sbintrunc* (*len-of TYPE*(*'a*) − *1*) (*numeral w*)))
⟨*proof*⟩

lemma *shiftr1-bl-of*:
length bl ≤ *len-of TYPE*(*'a*) ⟹
shiftr1 (*of-bl bl*::*'a*::*len0 word*) = *of-bl* (*butlast bl*)
⟨*proof*⟩

lemma *shiftr-bl-of*:
length bl ≤ *len-of TYPE*(*'a*) ⟹
(*of-bl bl*::*'a*::*len0 word*) >> *n* = *of-bl* (*take* (*length bl* − *n*) *bl*)
⟨*proof*⟩

lemma *shiftr-bl*:
(*x*::*'a*::*len0 word*) >> *n* ≡ *of-bl* (*take* (*len-of TYPE*(*'a*) − *n*) (*to-bl x*))
⟨*proof*⟩

lemma *msb-shift*:
msb (*w*::*'a*::*len word*) ⟷ (*w* >> (*len-of TYPE*(*'a*) − *1*)) ≠ *0*
⟨*proof*⟩

lemma *zip-replicate*:
n ≥ *length ys* ⟹ *zip* (*replicate n x*) *ys* = *map* (λ*y*. (*x*, *y*)) *ys*
⟨*proof*⟩

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*
⟨*proof*⟩

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
⟨*proof*⟩

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)
⟨*proof*⟩

127.25.2 Mask

lemma *nth-mask* [*OF refl*, *simp*]:
m = mask n ⟹ test-bit m i = (i < n & i < size m)
⟨*proof*⟩

lemma *mask-bl*: *mask n = of-bl (replicate n True)*
⟨*proof*⟩

lemma *mask-bin*: *mask n = word-of-int (bintrunc n (− 1))*
⟨*proof*⟩

lemma *and-mask-bintr*: *w AND mask n = word-of-int (bintrunc n (uint w))*
⟨*proof*⟩

lemma *and-mask-wi*: *word-of-int i AND mask n = word-of-int (bintrunc n i)*
⟨*proof*⟩

lemma *and-mask-no*: *numeral i AND mask n = word-of-int (bintrunc n (numeral i))*
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *and-mask-mod-2p*: *w AND mask n = word-of-int (uint w mod 2 ^ n)*
⟨*proof*⟩

lemma *and-mask-lt-2p*: *uint (w AND mask n) < 2 ^ n*
⟨*proof*⟩

lemma *eq-mod-iff*: *0 < (n::int) ⟹ b = b mod n ⟷ 0 ≤ b ∧ b < n*
⟨*proof*⟩

lemma *mask-eq-iff*: $(w\ AND\ mask\ n) = w \longleftrightarrow uint\ w < 2\ \hat{}\ n$
⟨*proof*⟩

lemma *and-mask-dvd*: $2\ \hat{}\ n\ dvd\ uint\ w = (w\ AND\ mask\ n = 0)$
⟨*proof*⟩

lemma *and-mask-dvd-nat*: $2\ \hat{}\ n\ dvd\ unat\ w = (w\ AND\ mask\ n = 0)$
⟨*proof*⟩

lemma *word-2p-lem*:
$n < size\ w \Longrightarrow w < 2\ \hat{}\ n = (uint\ (w :: 'a :: len\ word) < 2\ \hat{}\ n)$
⟨*proof*⟩

lemma *less-mask-eq*: $x < 2\ \hat{}\ n \Longrightarrow x\ AND\ mask\ n = (x :: 'a :: len\ word)$
⟨*proof*⟩

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*: $n < size\ x \Longrightarrow x\ AND\ mask\ n < 2\hat{}n$
⟨*proof*⟩

lemma *word-mod-2p-is-mask* [*OF refl*]:
$c = 2\ \hat{}\ n \Longrightarrow c > 0 \Longrightarrow x\ mod\ c = (x :: 'a :: len\ word)\ AND\ mask\ n$
⟨*proof*⟩

lemma *mask-eqs*:
$(a\ AND\ mask\ n) + b\ AND\ mask\ n = a + b\ AND\ mask\ n$
$a + (b\ AND\ mask\ n)\ AND\ mask\ n = a + b\ AND\ mask\ n$
$(a\ AND\ mask\ n) - b\ AND\ mask\ n = a - b\ AND\ mask\ n$
$a - (b\ AND\ mask\ n)\ AND\ mask\ n = a - b\ AND\ mask\ n$
$a * (b\ AND\ mask\ n)\ AND\ mask\ n = a * b\ AND\ mask\ n$
$(b\ AND\ mask\ n) * a\ AND\ mask\ n = b * a\ AND\ mask\ n$
$(a\ AND\ mask\ n) + (b\ AND\ mask\ n)\ AND\ mask\ n = a + b\ AND\ mask\ n$
$(a\ AND\ mask\ n) - (b\ AND\ mask\ n)\ AND\ mask\ n = a - b\ AND\ mask\ n$
$(a\ AND\ mask\ n) * (b\ AND\ mask\ n)\ AND\ mask\ n = a * b\ AND\ mask\ n$
$-(a\ AND\ mask\ n)\ AND\ mask\ n = -a\ AND\ mask\ n$
$word\text{-}succ\ (a\ AND\ mask\ n)\ AND\ mask\ n = word\text{-}succ\ a\ AND\ mask\ n$
$word\text{-}pred\ (a\ AND\ mask\ n)\ AND\ mask\ n = word\text{-}pred\ a\ AND\ mask\ n$
⟨*proof*⟩

lemma *mask-power-eq*:
$(x\ AND\ mask\ n)\ \hat{}\ k\ AND\ mask\ n = x\ \hat{}\ k\ AND\ mask\ n$
⟨*proof*⟩

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-bl (*revcast w* :: *'a* :: *len0 word*) =
takefill False (*len-of TYPE* (*'a*)) (*to-bl w*)
⟨*proof*⟩

lemma *revcast-rev-ucast* [*OF refl refl refl*]:
cs = [*rc*, *uc*] ⟹ *rc* = *revcast* (*word-reverse w*) ⟹ *uc* = *ucast w* ⟹
rc = *word-reverse uc*
⟨*proof*⟩

lemma *revcast-ucast*: *revcast w* = *word-reverse* (*ucast* (*word-reverse w*))
⟨*proof*⟩

lemma *ucast-revcast*: *ucast w* = *word-reverse* (*revcast* (*word-reverse w*))
⟨*proof*⟩

lemma *ucast-rev-revcast*: *ucast* (*word-reverse w*) = *word-reverse* (*revcast w*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *revcast-down-us* [*OF refl*]:
rc = *revcast* ⟹ *source-size rc* = *target-size rc* + *n* ⟹
rc (*w* :: *'a* :: *len word*) = *ucast* (*w* >>> *n*)
⟨*proof*⟩

lemma *revcast-down-su* [*OF refl*]:
rc = *revcast* ⟹ *source-size rc* = *target-size rc* + *n* ⟹
rc (*w* :: *'a* :: *len word*) = *scast* (*w* >> *n*)
⟨*proof*⟩

lemma *revcast-down-ss* [*OF refl*]:
rc = *revcast* ⟹ *source-size rc* = *target-size rc* + *n* ⟹
rc (*w* :: *'a* :: *len word*) = *scast* (*w* >>> *n*)
⟨*proof*⟩

lemma *cast-down-rev*:

uc = ucast ⟹ source-size uc = target-size uc + n ⟹
uc w = revcast ((w :: 'a :: len word) << n)
⟨*proof*⟩

lemma *revcast-up* [*OF refl*]:
rc = revcast ⟹ source-size rc + n = target-size rc ⟹
rc w = (ucast w :: 'a :: len word) << n
⟨*proof*⟩

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)))
⟨*proof*⟩

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)))
⟨*proof*⟩

lemma *slice1-0* [*simp*] : *slice1 n 0 = 0*
⟨*proof*⟩

lemma *slice-0* [*simp*] : *slice n 0 = 0*
⟨*proof*⟩

lemma *slice-take'*: *slice n w = of-bl (take (size w − n) (to-bl w))*
⟨*proof*⟩

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)*
⟨*proof*⟩

lemma *nth-slice*:
(*slice n w* :: *'a* :: *len0 word*) !! *m* =
(*w* !! (*m* + *n*) & *m* < *len-of TYPE* (*'a*))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *ucast-slice*: *ucast w* = *slice 0 w*
⟨*proof*⟩

lemma *slice-id*: *slice 0 t* = *t*
⟨*proof*⟩

lemma *revcast-slice1* [*OF refl*]:
rc = *revcast w* ⟹ *slice1* (*size rc*) *w* = *rc*
⟨*proof*⟩

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*))))
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *rev-slice*:

$n + k + \textit{len-of TYPE}('a::\textit{len0}) = \textit{len-of TYPE}('b::\textit{len0}) \Longrightarrow$
$\textit{slice } n \ (\textit{word-reverse } (w::'b \textit{ word})) = \textit{word-reverse } (\textit{slice } k \ w::'a \textit{ word})$
⟨*proof*⟩

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*:
$(\textit{sint } (x :: 'a :: \textit{len word}) + \textit{sint } y = \textit{sint } (x + y)) =$
$((((x+y) \ \textit{XOR} \ x) \ \textit{AND} \ ((x+y) \ \textit{XOR} \ y)) >> (\textit{size } x - 1) = 0)$
⟨*proof*⟩

127.26 Split and cat

lemmas *word-split-bin'* = *word-split-def*
lemmas *word-cat-bin'* = *word-cat-def*

lemma *word-rsplit-no*:
$(\textit{word-rsplit } (\textit{numeral bin} :: 'b :: \textit{len0 word}) :: 'a \textit{ word list}) =$
$\textit{map word-of-int } (\textit{bin-rsplit } (\textit{len-of TYPE}('a :: \textit{len}))$
$(\textit{len-of TYPE}('b), \textit{bintrunc } (\textit{len-of TYPE}('b)) \ (\textit{numeral bin})))$
⟨*proof*⟩

lemmas *word-rsplit-no-cl* [*simp*] = *word-rsplit-no*
[*unfolded bin-rsplitl-def bin-rsplit-l* [*symmetric*]]

lemma *test-bit-cat*:
$wc = \textit{word-cat } a \ b \Longrightarrow wc \ !! \ n = (n < \textit{size } wc \ \&$
$(\textit{if } n < \textit{size } b \textit{ then } b \ !! \ n \textit{ else } a \ !! \ (n - \textit{size } b)))$
⟨*proof*⟩

lemma *word-cat-bl*: $\textit{word-cat } a \ b = \textit{of-bl } (\textit{to-bl } a \ @ \ \textit{to-bl } b)$
⟨*proof*⟩

lemma *of-bl-append*:
$(\textit{of-bl } (xs \ @ \ ys) :: 'a :: \textit{len word}) = \textit{of-bl } xs * 2\hat{}(\textit{length } ys) + \textit{of-bl } ys$
⟨*proof*⟩

lemma *of-bl-False* [*simp*]:
$\textit{of-bl } (\textit{False}\#xs) = \textit{of-bl } xs$
⟨*proof*⟩

lemma *of-bl-True* [*simp*]:
$(\textit{of-bl } (\textit{True}\#xs)::'a::\textit{len word}) = 2\hat{}\textit{length } xs + \textit{of-bl } xs$
⟨*proof*⟩

lemma *of-bl-Cons*:
of-bl (*x*#*xs*) = *of-bool x* ∗ *2*^*length xs* + *of-bl xs*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)))
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)))
⟨*proof*⟩
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*)))
⟨*proof*⟩

lemma *test-bit-split*:
word-split c = (*a*, *b*) ⟹
(∀ *n*::*nat*. *b* !! *n* ⟷ *n* < *size b* ∧ *c* !! *n*) ∧ (∀ *m*::*nat*. *a* !! *m* ⟷ *m* < *size a*
∧ *c* !! (*m* + *size b*))
⟨*proof*⟩

lemma *test-bit-split-eq*: *word-split c* = (*a*, *b*) ⟷
((*ALL n*::*nat*. *b* !! *n* = (*n* < *size b* & *c* !! *n*)) &
(*ALL m*::*nat*. *a* !! *m* = (*m* < *size a* & *c* !! (*m* + *size b*))))
⟨*proof*⟩

lemma *word-cat-id*: *word-cat a b* = *b*
⟨*proof*⟩
lemma *word-cat-hom*:
len-of TYPE(*'a*::*len0*) <= *len-of TYPE*(*'b*::*len0*) + *len-of TYPE* (*'c*::*len0*)
⟹
(*word-cat* (*word-of-int w* :: *'b word*) (*b* :: *'c word*) :: *'a word*) =
word-of-int (*bin-cat w* (*size b*) (*uint b*))
⟨*proof*⟩

lemma *word-cat-split-alt*:

size w <= size u + size v ⟹ word-split w = (u, v) ⟹ word-cat u v = w
⟨*proof*⟩

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
⟨*proof*⟩

lemma *slice-cat1* [*OF refl*]:
wc = word-cat a b ⟹ size wc >= size a + size b ⟹ slice (size b) wc = a
⟨*proof*⟩

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
⟨*proof*⟩

lemma *word-split-cat-alt*:
w = word-cat u v ⟹ size u + size v <= size w ⟹ word-split w = (u, v)
⟨*proof*⟩

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]*
⟨*proof*⟩

lemma *word-rsplit-empty-iff-size*:
(word-rsplit w = []) = (size w = 0)
⟨*proof*⟩

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))*
⟨*proof*⟩

lemma *word-rcat-bl*: *word-rcat wl = of-bl (concat (map to-bl wl))*
⟨*proof*⟩

lemma *size-rcat-lem'*:
*size (concat (map to-bl wl)) = length wl * size (hd wl)*
⟨*proof*⟩

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))
⟨*proof*⟩

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))
⟨*proof*⟩

lemma *foldl-eq-foldr*:
foldl op + x xs = foldr op + (x # xs) (0 :: 'a :: comm-monoid-add)
⟨*proof*⟩

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
⟨*proof*⟩

lemma *length-word-rsplit-size*:
n = len-of TYPE ('a :: len) ⟹
*(length (word-rsplit w :: 'a word list) <= m) = (size w <= m * n)*
⟨*proof*⟩

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

⟨*proof*⟩

lemma *length-word-rsplit-even-size*:
n = *len-of TYPE* (*'a* :: *len*) ⟹ *size w* = *m* ∗ *n* ⟹
length (*word-rsplit w* :: *'a word list*) = *m*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *rotate1-rl* [*simp*] : *rotater1* (*rotate1 l*) = *l*
⟨*proof*⟩

lemma *rotate1-lr* [*simp*] : *rotate1* (*rotater1 l*) = *l*
⟨*proof*⟩

lemma *rotater1-rev'*: *rotater1* (*rev xs*) = *rev* (*rotate1 xs*)
⟨*proof*⟩

lemma *rotater-rev'*: *rotater n* (*rev xs*) = *rev* (*rotate n xs*)
⟨*proof*⟩

lemma *rotater-rev*: *rotater n ys* = *rev* (*rotate n* (*rev ys*))
⟨*proof*⟩

lemma *rotater-drop-take*:
rotater n xs =
drop (*length xs* − *n mod length xs*) *xs* @
take (*length xs* − *n mod length xs*) *xs*
⟨*proof*⟩

lemma *rotater-Suc* [*simp*] :
rotater (*Suc n*) *xs* = *rotater1* (*rotater n xs*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *rotate-gal'*: (*ys* = *rotater n xs*) = (*xs* = *rotate n ys*)
⟨*proof*⟩

lemma *length-rotater* [*simp*]:
length (*rotater n xs*) = *length xs*
⟨*proof*⟩

lemma *restrict-to-left*:
assumes *x* = *y*
shows (*x* = *z*) = (*y* = *z*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *rotater1-map*: *rotater1* (*map f xs*) = *map f* (*rotater1 xs*)
⟨*proof*⟩

lemma *rotater-map*:
rotater n (*map f xs*) = *map f* (*rotater n xs*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *rotater1-zip*:
length xs = *length ys* ⟹
rotater1 (*zip xs ys*) = *zip* (*rotater1 xs*) (*rotater1 ys*)
⟨*proof*⟩

lemma *rotater1-map2*:
length xs = *length ys* ⟹
rotater1 (*map2 f xs ys*) = *map2 f* (*rotater1 xs*) (*rotater1 ys*)

⟨*proof*⟩

lemmas *lrth* =
box-equals [*OF asm-rl length-rotater* [*symmetric*]
length-rotater [*symmetric*],
THEN rotater1-map2]

lemma *rotater-map2*:
length xs = *length ys* ⟹
rotater n (*map2 f xs ys*) = *map2 f* (*rotater n xs*) (*rotater n ys*)
⟨*proof*⟩

lemma *rotate1-map2*:
length xs = *length ys* ⟹
rotate1 (*map2 f xs ys*) = *map2 f* (*rotate1 xs*) (*rotate1 ys*)
⟨*proof*⟩

lemmas *lth* = *box-equals* [*OF asm-rl length-rotate* [*symmetric*]
length-rotate [*symmetric*], *THEN rotate1-map2*]

lemma *rotate-map2*:
length xs = *length ys* ⟹
rotate n (*map2 f xs ys*) = *map2 f* (*rotate n xs*) (*rotate n ys*)
⟨*proof*⟩

lemma *to-bl-rotl*:
to-bl (*word-rotl n w*) = *rotate n* (*to-bl w*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *word-rotr-rev*:
word-rotr n w = *word-reverse* (*word-rotl n* (*word-reverse w*))
⟨*proof*⟩

lemma *word-roti-0* [*simp*]: *word-roti 0 w* = *w*
⟨*proof*⟩

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*⟩

lemma *word-roti-conv-mod'*: *word-roti n w* = *word-roti* (*n mod int* (*size w*)) *w*
⟨*proof*⟩

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
⟨*proof*⟩
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)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *word-roti-0'* [*simp*] : *word-roti n 0 = 0*
⟨*proof*⟩

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^len-of TYPE('a)*

⟨*proof*⟩

lemma *word-nat-cases* [*cases type*: *word*]:
obtains *n* **where** (*x* ::'*a*::*len word*) = *of-nat n* **and** *n* < *2^len-of TYPE*('*a*)
⟨*proof*⟩

lemma *max-word-eq*: (*max-word*::'*a*::*len word*) = *2^len-of TYPE*('*a*) − *1*
⟨*proof*⟩

lemma *max-word-max* [*simp*,*intro*!]: *n* ≤ *max-word*
⟨*proof*⟩

lemma *word-of-int-2p-len*: *word-of-int* (*2* ^ *len-of TYPE*('*a*)) = (*0*::'*a*::*len0 word*)
⟨*proof*⟩

lemma *word-pow-0*:
(*2*::'*a*::*len word*) ^ *len-of TYPE*('*a*) = *0*
⟨*proof*⟩

lemma *max-word-wrap*: *x* + *1* = *0* ⟹ *x* = *max-word*
⟨*proof*⟩

lemma *max-word-minus*:
max-word = (−*1*::'*a*::*len word*)
⟨*proof*⟩

lemma *max-word-bl* [*simp*]:
to-bl (*max-word*::'*a*::*len word*) = *replicate* (*len-of TYPE*('*a*)) *True*
⟨*proof*⟩

lemma *max-test-bit* [*simp*]:
(*max-word*::'*a*::*len word*) !! *n* = (*n* < *len-of TYPE*('*a*))
⟨*proof*⟩

lemma *word-and-max* [*simp*]:
x AND max-word = *x*
⟨*proof*⟩

lemma *word-or-max* [*simp*]:
x OR max-word = *max-word*
⟨*proof*⟩

lemma *word-ao-dist2*:
x AND (*y OR z*) = *x AND y OR x AND* (*z*::'*a*::*len0 word*)
⟨*proof*⟩

lemma *word-oa-dist2*:
x OR y AND z = (*x OR y*) *AND* (*x OR* (*z*::'*a*::*len0 word*))
⟨*proof*⟩

lemma *word-and-not* [*simp*]:
x AND NOT x = (0::'a::len0 word)
⟨*proof*⟩

lemma *word-or-not* [*simp*]:
x OR NOT x = max-word
⟨*proof*⟩

lemma *word-boolean*:
boolean (op AND) (op OR) bitNOT 0 max-word
⟨*proof*⟩

interpretation *word-bool-alg*:
boolean op AND op OR bitNOT 0 max-word
⟨*proof*⟩

lemma *word-xor-and-or*:
x XOR y = x AND NOT y OR NOT x AND (y::'a::len0 word)
⟨*proof*⟩

interpretation *word-bool-alg*:
boolean-xor op AND op OR bitNOT 0 max-word op XOR
⟨*proof*⟩

lemma *shiftr-x-0* [*iff*]:
(x::'a::len0 word) >> 0 = x
⟨*proof*⟩

lemma *shiftl-x-0* [*simp*]:
(x :: 'a :: len word) << 0 = x
⟨*proof*⟩

lemma *shiftl-1* [*simp*]:
(1::'a::len word) << n = 2^n
⟨*proof*⟩

lemma *uint-lt-0* [*simp*]:
uint x < 0 = False
⟨*proof*⟩

lemma *shiftr1-1* [*simp*]:
shiftr1 (1::'a::len word) = 0
⟨*proof*⟩

lemma *shiftr-1*[*simp*]:
(1::'a::len word) >> n = (if n = 0 then 1 else 0)
⟨*proof*⟩

lemma *word-less-1* [*simp*]:
((x::'a::*len word*) < *1*) = (x = *0*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *map-replicate-True*:
n = *length xs* ⟹
map (λ(x,y). x & y) (*zip xs* (*replicate n True*)) = *xs*
⟨*proof*⟩

lemma *map-replicate-False*:
n = *length xs* ⟹ *map* (λ(x,y). x & y)
(*zip xs* (*replicate n False*)) = *replicate n False*
⟨*proof*⟩

lemma *bl-and-mask*:
fixes w :: 'a::*len word*
fixes n
defines n' ≡ *len-of TYPE*('a) − *n*
shows *to-bl* (w *AND mask n*) = *replicate* n' *False* @ *drop* n' (*to-bl* w)
⟨*proof*⟩

lemma *drop-rev-takefill*:
length xs ≤ *n* ⟹
drop (*n* − *length xs*) (*rev* (*takefill False n* (*rev xs*))) = *xs*
⟨*proof*⟩

lemma *map-nth-0* [*simp*]:
map (*op* !! (*0*::'a::*len0 word*)) *xs* = *replicate* (*length xs*) *False*
⟨*proof*⟩

lemma *uint-plus-if-size*:
uint (x + y) =
(*if uint x* + *uint y* < *2*^*size x then*
uint x + *uint y*
else
uint x + *uint y* − *2*^*size x*)
⟨*proof*⟩

lemma *unat-plus-if-size*:
unat (x + (y::'a::*len word*)) =
(*if unat x* + *unat y* < *2*^*size x then*
unat x + *unat y*
else

unat x + unat y − 2^size x)
⟨*proof*⟩

lemma *word-neq-0-conv*:
fixes *w* :: *'a* :: *len word*
shows *(w ≠ 0) = (0 < w)*
⟨*proof*⟩

lemma *max-lt*:
unat (max a b div c) = unat (max a b) div unat (c:: 'a :: len word)
⟨*proof*⟩

lemma *uint-sub-if-size*:
uint (x − y) =
(if uint y ≤ uint x then
unat x − uint y
else
uint x − uint y + 2^size x)
⟨*proof*⟩

lemma *unat-sub*:
b <= a ⟹ unat (a − b) = unat a − unat b
⟨*proof*⟩

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*⟩

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)
⟨*proof*⟩

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*⟩

lemma *mod-minus-cong*:
assumes *1*: *(b::int) = b'*
and *2*: *x mod b' = x' mod b'*

and *3*: $y \bmod b' = y' \bmod b'$
and *4*: $x' - y' = z'$
shows $(x - y) \bmod b = z' \bmod b'$
⟨*proof*⟩

lemma *word-induct-less*:
$[\![P\ (0::'a::len\ word);\ \bigwedge n.\ [\![n < m;\ P\ n]\!] \Longrightarrow P\ (1 + n)]\!] \Longrightarrow P\ m$
⟨*proof*⟩

lemma *word-induct*:
$[\![P\ (0::'a::len\ word);\ \bigwedge n.\ P\ n \Longrightarrow P\ (1 + n)]\!] \Longrightarrow P\ m$
⟨*proof*⟩

lemma *word-induct2* [*induct type*]:
$[\![P\ 0;\ \bigwedge n.\ [\![1 + n \neq 0;\ P\ n]\!] \Longrightarrow P\ (1 + n)]\!] \Longrightarrow P\ (n::'b::len\ word)$
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *word-rec-Pred*:
$n \neq 0 \Longrightarrow$ *word-rec z s n* = *s* $(n - 1)$ (*word-rec z s* $(n - 1)$)
⟨*proof*⟩

lemma *word-rec-in*:
f (*word-rec z* (λ-. *f*) *n*) = *word-rec* (*f z*) (λ-. *f*) *n*
⟨*proof*⟩

lemma *word-rec-in2*:
f n (*word-rec z f n*) = *word-rec* (*f 0 z*) (*f* ∘ *op* + *1*) *n*
⟨*proof*⟩

lemma *word-rec-twice*:
$m \leq n \Longrightarrow$ *word-rec z f n* = *word-rec* (*word-rec z f* $(n - m)$) (*f* ∘ *op* + $(n - m)$) *m*
⟨*proof*⟩

lemma *word-rec-id*: *word-rec z* (λ-. *id*) *n* = *z*
⟨*proof*⟩

lemma *word-rec-id-eq*: $\forall m < n.\ f\ m = id \Longrightarrow word\text{-}rec\ z\ f\ n = z$
⟨*proof*⟩

lemma *word-rec-max*:
$\forall m{\geq}n.\ m \neq -\ 1 \longrightarrow f\ m = id \Longrightarrow word\text{-}rec\ z\ f\ (-\ 1) = word\text{-}rec\ z\ f\ n$
⟨*proof*⟩

lemma *unatSuc*:
$1 + n \neq (0{::}'a{::}len\ word) \Longrightarrow unat\ (1 + n) = Suc\ (unat\ n)$
⟨*proof*⟩

declare *bin-to-bl-def* [*simp*]

⟨*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*⟩

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* ⇒ *bool* ⇒ *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* ⇒ *bool* ⇒ *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:

- ∀ *x*. *trigger* [[*pat* (*P x*)]] (*weight 2* (*P x*))
- ∀ *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*⟩
named-theorems *old-z3-simp simplification rules for Z3 proof reconstruction*
⟨*ML*⟩

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*]:
$(P \wedge Q) = (\neg(\neg P \vee \neg Q))$
$(P \wedge Q) = (\neg(\neg Q \vee \neg P))$
$(\neg P \wedge Q) = (\neg(P \vee \neg Q))$
$(\neg P \wedge Q) = (\neg(\neg Q \vee P))$
$(P \wedge \neg Q) = (\neg(\neg P \vee Q))$
$(P \wedge \neg Q) = (\neg(Q \vee \neg P))$
$(\neg P \wedge \neg Q) = (\neg(P \vee Q))$
$(\neg P \wedge \neg Q) = (\neg(Q \vee P))$
⟨*proof*⟩

lemma [*old-z3-rule*]:
$(P \longrightarrow Q) = (Q \vee \neg P)$
$(\neg P \longrightarrow Q) = (P \vee Q)$
$(\neg P \longrightarrow Q) = (Q \vee P)$
$(\mathit{True} \longrightarrow P) = P$
$(P \longrightarrow \mathit{True}) = \mathit{True}$
$(\mathit{False} \longrightarrow P) = \mathit{True}$
$(P \longrightarrow P) = \mathit{True}$
⟨*proof*⟩

lemma [*old-z3-rule*]:

$((P = Q) \longrightarrow R) = (R \mid (Q = (\neg P)))$
⟨*proof*⟩

lemma [*old-z3-rule*]:
$(\neg True) = False$
$(\neg False) = True$
$(x = x) = True$
$(P = True) = P$
$(True = P) = P$
$(P = False) = (\neg P)$
$(False = P) = (\neg P)$
$((\neg P) = P) = False$
$(P = (\neg P)) = False$
$((\neg P) = (\neg Q)) = (P = Q)$
$\neg(P = (\neg Q)) = (P = Q)$
$\neg((\neg P) = Q) = (P = Q)$
$(P \neq Q) = (Q = (\neg P))$
$(P = Q) = ((\neg P \vee Q) \wedge (P \vee \neg Q))$
$(P \neq Q) = ((\neg P \vee \neg Q) \wedge (P \vee Q))$
⟨*proof*⟩

lemma [*old-z3-rule*]:
$(if\ P\ then\ P\ else\ \neg P) = True$
$(if\ \neg P\ then\ \neg P\ else\ P) = True$
$(if\ P\ then\ True\ else\ False) = P$
$(if\ P\ then\ False\ else\ True) = (\neg P)$
$(if\ P\ then\ Q\ else\ True) = ((\neg P) \vee Q)$
$(if\ P\ then\ Q\ else\ True) = (Q \vee (\neg P))$
$(if\ P\ then\ Q\ else\ \neg Q) = (P = Q)$
$(if\ P\ then\ Q\ else\ \neg Q) = (Q = P)$
$(if\ P\ then\ \neg Q\ else\ Q) = (P = (\neg Q))$
$(if\ P\ then\ \neg Q\ else\ Q) = ((\neg Q) = P)$
$(if\ \neg P\ then\ x\ else\ y) = (if\ P\ then\ y\ else\ x)$
$(if\ P\ then\ (if\ Q\ then\ x\ else\ y)\ else\ x) = (if\ P \wedge (\neg Q)\ then\ y\ else\ x)$
$(if\ P\ then\ (if\ Q\ then\ x\ else\ y)\ else\ x) = (if\ (\neg Q) \wedge P\ then\ y\ else\ x)$
$(if\ P\ then\ (if\ Q\ then\ x\ else\ y)\ else\ y) = (if\ P \wedge Q\ then\ x\ else\ y)$
$(if\ P\ then\ (if\ Q\ then\ x\ else\ y)\ else\ y) = (if\ Q \wedge P\ then\ x\ else\ y)$
$(if\ P\ then\ x\ else\ if\ P\ then\ y\ else\ z) = (if\ P\ then\ x\ else\ z)$
$(if\ P\ then\ x\ else\ if\ Q\ then\ x\ else\ y) = (if\ P \vee Q\ then\ x\ else\ y)$
$(if\ P\ then\ x\ else\ if\ Q\ then\ x\ else\ y) = (if\ Q \vee P\ then\ x\ else\ y)$
$(if\ P\ then\ x = y\ else\ x = z) = (x = (if\ P\ then\ y\ else\ z))$
$(if\ P\ then\ x = y\ else\ y = z) = (y = (if\ P\ then\ x\ else\ z))$
$(if\ P\ then\ x = y\ else\ z = y) = (y = (if\ P\ then\ x\ else\ z))$
⟨*proof*⟩

lemma [*old-z3-rule*]:
$0 + (x::int) = x$
$x + 0 = x$
$x + x = 2 * x$

$0 * x = 0$
$1 * x = x$
$x + y = y + x$
⟨*proof*⟩

lemma [*old-z3-rule*]:
$P = Q \vee P \vee Q$
$P = Q \vee \neg P \vee \neg Q$
$(\neg P) = Q \vee \neg P \vee Q$
$(\neg P) = Q \vee P \vee \neg Q$
$P = (\neg Q) \vee \neg P \vee Q$
$P = (\neg Q) \vee P \vee \neg Q$
$P \neq Q \vee P \vee \neg Q$
$P \neq Q \vee \neg P \vee Q$
$P \neq (\neg Q) \vee P \vee Q$
$(\neg P) \neq Q \vee P \vee Q$
$P \vee Q \vee P \neq (\neg Q)$
$P \vee Q \vee (\neg P) \neq Q$
$P \vee \neg Q \vee P \neq Q$
$\neg P \vee Q \vee P \neq Q$
$P \vee y = (\mathit{if}\ P\ \mathit{then}\ x\ \mathit{else}\ y)$
$P \vee (\mathit{if}\ P\ \mathit{then}\ x\ \mathit{else}\ y) = y$
$\neg P \vee x = (\mathit{if}\ P\ \mathit{then}\ x\ \mathit{else}\ y)$
$\neg P \vee\ (\mathit{if}\ P\ \mathit{then}\ x\ \mathit{else}\ y) = x$
$P \vee R \vee \neg(\mathit{if}\ P\ \mathit{then}\ Q\ \mathit{else}\ R)$
$\neg P \vee Q \vee \neg(\mathit{if}\ P\ \mathit{then}\ Q\ \mathit{else}\ R)$
$\neg(\mathit{if}\ P\ \mathit{then}\ Q\ \mathit{else}\ R) \vee \neg P \vee Q$
$\neg(\mathit{if}\ P\ \mathit{then}\ Q\ \mathit{else}\ R) \vee P \vee R$
$(\mathit{if}\ P\ \mathit{then}\ Q\ \mathit{else}\ R) \vee \neg P \vee \neg Q$
$(\mathit{if}\ P\ \mathit{then}\ Q\ \mathit{else}\ R) \vee P \vee \neg R$
$(\mathit{if}\ P\ \mathit{then}\ \neg Q\ \mathit{else}\ R) \vee \neg P \vee Q$
$(\mathit{if}\ P\ \mathit{then}\ Q\ \mathit{else}\ \neg R) \vee P \vee R$
⟨*proof*⟩

⟨*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.