Infinite Lists

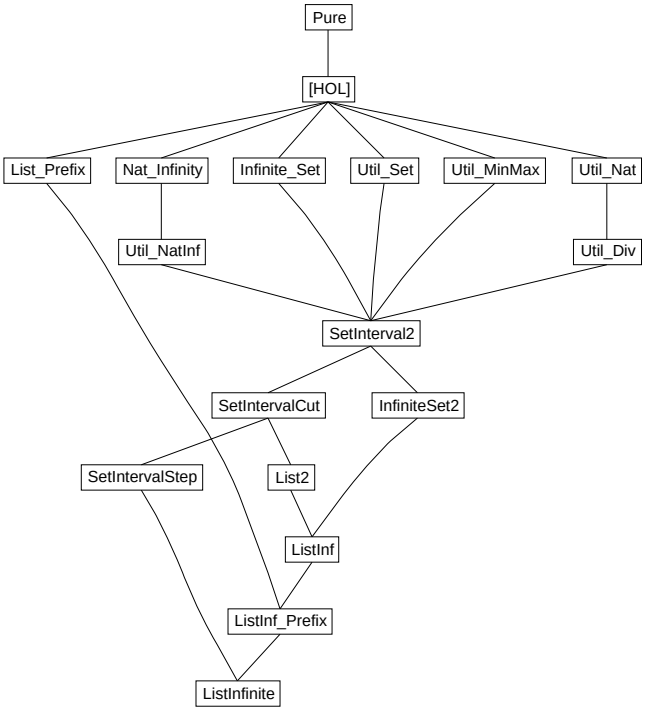
David Trachtenherz

February 24, 2011

Abstract

We introduce a theory of infinite lists in HOL formalized as functions over naturals (folder ListInf, theories ListInf and ListInf_Prefix). It also provides additional results for finite lists (theory ListInf/List2), natural numbers (folder CommonArith, esp. division/modulo, naturals with infinity), sets (folder CommonSet, esp. cutting/truncating sets, traversing sets of naturals).

Contents



1 Util-Set: Convenience results for set quantifiers

theory *Util-Set*
imports *Fun Set*
begin

1.1 Some auxiliary results for HOL rules

lemma *conj-disj-absorb*: $(P \wedge Q \vee Q) = Q$ **by** *blast*
lemma *disj-eq-distribL*: $((a \vee b) = (a \vee c)) = (a \vee (b = c))$ **by** *blast*
lemma *disj-eq-distribR*: $((a \vee c) = (b \vee c)) = ((a = b) \vee c)$ **by** *blast*

1.1.1 Some auxiliary results for *Let*

lemma *Let-swap*: $f\ (\mathit{let}\ x{=}a\ \mathit{in}\ g\ x) = (\mathit{let}\ x{=}a\ \mathit{in}\ f\ (g\ x))$ **by** *simp*

1.1.2 Some auxiliary *if*-rules

thm *if-P*
lemma *if-P'*: $[\![\ P;\ x = z\ ]\!] \Longrightarrow (\mathit{if}\ P\ \mathit{then}\ x\ \mathit{else}\ y) = z$ **by** *simp*
thm *if-not-P*
lemma *if-not-P'*: $[\![\ \neg\ P;\ y = z\ ]\!] \Longrightarrow (\mathit{if}\ P\ \mathit{then}\ x\ \mathit{else}\ y) = z$ **by** *simp*

lemma *if-P-both*: $[\![\ Q\ x;\ Q\ y\ ]\!] \Longrightarrow Q\ (\mathit{if}\ P\ \mathit{then}\ x\ \mathit{else}\ y)$ **by** *simp*
lemma *if-P-both-in-set*: $[\![\ x \in s;\ y \in s\ ]\!] \Longrightarrow (\mathit{if}\ P\ \mathit{then}\ x\ \mathit{else}\ y) \in s$ **by** *simp*

1.1.3 Some auxiliary rules for function composition

lemma *comp2-conv*: $f1 \circ f2 = (\lambda x.\ f1\ (f2\ x))$ **by** (*simp add*: *comp-def*)
lemma *comp3-conv*: $f1 \circ f2 \circ f3 = (\lambda x.\ f1\ (f2\ (f3\ x)))$ **by** (*simp add*: *comp-def*)

1.2 Some auxiliary lemmata for quantifiers

1.2.1 Auxiliary results for universal and existential quantifiers

lemma *ball-cong2*:
$[\![\ I \subseteq A;\ \forall x{\in}A.\ f\,x = g\,x\ ]\!] \Longrightarrow (\forall x{\in}I.\ P\ (f\,x)) = (\forall x{\in}I.\ P\ (g\,x))$ **by** *fastsimp*
lemma *bex-cong2*:
$[\![\ I \subseteq A;\ \forall x{\in}I.\ f\,x = g\,x\ ]\!] \Longrightarrow (\exists x{\in}I.\ P\ (f\,x)) = (\exists x{\in}I.\ P\ (g\,x))$ **by** *simp*
lemma *ball-all-cong*:
$\forall x.\ f\,x = g\,x \Longrightarrow (\forall x{\in}I.\ P\ (f\,x)) = (\forall x{\in}I.\ P\ (g\,x))$ **by** *simp*
lemma *bex-all-cong*:
$\forall x.\ f\,x = g\,x \Longrightarrow (\exists x{\in}I.\ P\ (f\,x)) = (\exists x{\in}I.\ P\ (g\,x))$ **by** *simp*
lemma *all-cong*:
$\forall x.\ f\,x = g\,x \Longrightarrow (\forall x.\ P\ (f\,x)) = (\forall x.\ P\ (g\,x))$ **by** *simp*
lemma *ex-cong*:
$\forall x.\ f\,x = g\,x \Longrightarrow (\exists x.\ P\ (f\,x)) = (\exists x.\ P\ (g\,x))$ **by** *simp*

lemmas *all-eqI* = *iff-allI*

lemmas *ex-eqI* = *iff-exI*

lemma *all-imp-eqI*:
$[\![P = P'; \bigwedge x.\ P\ x \Longrightarrow Q\ x = Q'\ x\ ]\!] \Longrightarrow$
$(\forall x.\ P\ x \longrightarrow Q\ x) = (\forall x.\ P'\ x \longrightarrow Q'\ x)$
by *blast*
lemma *ex-imp-eqI*:
$[\![P = P'; \bigwedge x.\ P\ x \Longrightarrow Q\ x = Q'\ x\ ]\!] \Longrightarrow$
$(\exists x.\ P\ x \wedge Q\ x) = (\exists x.\ P'\ x \wedge Q'\ x)$
by *blast*

1.2.2 Auxiliary results for *empty* sets

lemma *empty-imp-not-in*: $x \notin \{\}$ **by** *blast*
lemma *ex-imp-not-empty*: $\exists x.\ x \in A \Longrightarrow A \neq \{\}$ **by** *blast*
lemma *in-imp-not-empty*: $x \in A \Longrightarrow A \neq \{\}$ **by** *blast*
lemma *not-empty-imp-ex*: $A \neq \{\} \Longrightarrow \exists x.\ x \in A$ **by** *blast*
lemma *not-ex-in-conv*: $(\neg\ (\exists x.\ x \in A)) = (A = \{\})$ **by** *blast*

1.2.3 Some auxiliary results for subset and membership relation

lemma *bex-subset-imp-bex*: $[\![\exists x{\in}A.\ P\ x;\ A \subseteq B\ ]\!] \Longrightarrow \exists x{\in}B.\ P\ x$ **by** *blast*
lemma *bex-imp-ex*: $\exists x{\in}A.\ P\ x \Longrightarrow \exists x.\ P\ x$ **by** *blast*
lemma *ball-subset-imp-ball*: $[\![\forall x{\in}B.\ P\ x;\ A \subseteq B\ ]\!] \Longrightarrow \forall x{\in}A.\ P\ x$ **by** *blast*
thm
ball-subset-imp-ball
ball-subset-imp-ball[*rule-format*]
lemma *all-imp-ball*: $\forall x.\ P\ x \Longrightarrow \forall x{\in}A.\ P\ x$ **by** *blast*

thm *mem-Collect-eq*
lemma *mem-Collect-eq-not*: $(a \notin \{x.\ P\ x\}) = (\neg\ P\ a)$ **by** *blast*
lemma *Collect-not-in-imp-not*: $a \notin \{x.\ P\ x\} \Longrightarrow \neg\ P\ a$ **by** *blast*
lemma *Collect-not-imp-not-in*: $\neg\ P\ a \Longrightarrow a \notin \{x.\ P\ x\}$ **by** *blast*
lemma *Collect-is-subset*: $\{x \in A.\ P\ x\} \subseteq A$ **by** *blast*

end

2 Util-MinMax: Order and linear order: min and max

theory *Util-MinMax*
imports *Lattices*
begin

2.1 Additional lemmata about *min* and *max*

thm *min-less-iff-conj*

lemma *min-less-imp-conj*: $(z::'a::linorder) < min\ x\ y \Longrightarrow z < x \wedge z < y$ **by** *simp*
lemma *conj-less-imp-min*: $[\![\ z < x;\ z < y\ ]\!] \Longrightarrow (z::'a::linorder) < min\ x\ y$ **by** *simp*

lemmas *min-le-iff-conj* = *min-max.le-inf-iff*
lemma *min-le-imp-conj*: $(z::'a::linorder) \leq min\ x\ y \Longrightarrow z \leq x \wedge z \leq y$ **by** *simp*

lemmas *conj-le-imp-min* = *min-max.le-infI*

thm *min-max.inf-absorb1*

lemmas *min-eqL* = *min-max.inf-absorb1*

lemmas *min-eqR* = *min-max.inf-absorb2*
lemmas *min-eq* = *min-eqL min-eqR*
thm *min-eq*

thm *max-less-iff-conj*
lemma *max-less-imp-conj*:$max\ x\ y < b \Longrightarrow x < (b::('a::linorder)) \wedge y < b$ **by** *simp*
lemma *conj-less-imp-max*:$[\![\ x < (b::('a::linorder));\ y < b\ ]\!] \Longrightarrow max\ x\ y < b$ **by** *simp*

lemmas *max-le-iff-conj* = *min-max.le-sup-iff*
lemma *max-le-imp-conj*:$max\ x\ y \leq b \Longrightarrow x \leq (b::('a::linorder)) \wedge y \leq b$ **by** *simp*

lemmas *conj-le-imp-max* = *min-max.le-supI*

thm *min-max.sup-absorb1*

lemmas *max-eqL* = *min-max.sup-absorb1*

lemmas *max-eqR* = *min-max.sup-absorb2*
lemmas *max-eq* = *max-eqL max-eqR*
thm *max-eq*

thm
le-maxI1
le-maxI2

lemmas *le-minI1* = *min-max.inf-le1*

lemmas *le-minI2* = *min-max.inf-le2*

lemma
min-le-monoR: $(a::'a::linorder) \leq b \Longrightarrow min\ x\ a \leq min\ x\ b$ **and**
min-le-monoL: $(a::'a::linorder) \leq b \Longrightarrow min\ a\ x \leq min\ b\ x$
by (*fastsimp simp*: *min-max.inf-mono min-def*)+
lemma
max-le-monoR: $(a::'a::linorder) \leq b \Longrightarrow max\ x\ a \leq max\ x\ b$ **and**
max-le-monoL: $(a::'a::linorder) \leq b \Longrightarrow max\ a\ x \leq max\ b\ x$
by (*fastsimp simp*: *min-max.sup-mono max-def*)+

end

3 Util-NatInf: Results for natural arithmetics with infinity

theory *Util-NatInf*
imports $ISABELLE-HOME/src/HOL/Library/Nat-Infinity
begin

3.1 Arithmetic operations with *inat*

3.1.1 Additional definitions

thm *one-inat-def*
thm *plus-inat-def*
thm *times-inat-def*

instantiation *inat* :: {*minus*, *Divides.div*}
begin

definition
diff-inat-def [*code del*]:
$a - b \equiv (case\ a\ of$
$(Fin\ x) \Rightarrow (case\ b\ of\ (Fin\ y) \Rightarrow Fin\ (x - y) \mid \infty \Rightarrow 0) \mid$
$\infty \Rightarrow \infty)$

definition
div-inat-def [*code del*]:
$a\ div\ b \equiv (case\ a\ of$
$(Fin\ x) \Rightarrow (case\ b\ of\ (Fin\ y) \Rightarrow Fin\ (x\ div\ y) \mid \infty \Rightarrow 0) \mid$
$\infty \Rightarrow (case\ b\ of\ (Fin\ y) \Rightarrow ((case\ y\ of\ 0 \Rightarrow 0 \mid Suc\ n \Rightarrow \infty)) \mid \infty \Rightarrow \infty\ ))$
definition
mod-inat-def [*code del*]:
$a\ mod\ b \equiv (case\ a\ of$
$(Fin\ x) \Rightarrow (case\ b\ of\ (Fin\ y) \Rightarrow Fin\ (x\ mod\ y) \mid \infty \Rightarrow a) \mid$
$\infty \Rightarrow \infty)$

instance ..

end

lemmas *inat-arith-defs* =
zero-inat-def one-inat-def
plus-inat-def diff-inat-def times-inat-def div-inat-def mod-inat-def
declare *zero-inat-def*[*simp*]

primrec *the-Fin* :: *inat* ⇒ *nat* **where**
the-Fin (*Fin n*) = *n*

lemma *Fin-the-Fin*: $n \neq \infty \Longrightarrow$ *Fin* (*the-Fin n*) = *n* **by** *fastsimp*

3.1.2 Results for comparison operators

lemma *Fin-ile-mono*: (*Fin m* ≤ *Fin n*) = (*m* ≤ *n*) **by** *simp*
lemma *Fin-iless-mono*: (*Fin m* < *Fin n*) = (*m* < *n*) **by** *simp*
lemma *Infty-ub*: *n* ≤ ∞ **by** *simp*

thm *less-not-sym*
lemma *iless-not-sym*: (*x*::*inat*) < *y* ⟹ ¬ *y* < *x* **by** *simp*
thm *less-not-refl*
lemma *iless-not-refl*: ¬ (*n*::*inat*) < *n* **by** *simp*
thm *le-refl*
lemma *ile-refl*: (*n*::*inat*) ≤ *n* **by** *simp*
lemma *eq-imp-ile*: (*m*::*inat*) = *n* ⟹ *m* ≤ *n* **by** *simp*

lemma *not-Infty-iless*: ¬ ∞ < *n* **by** *simp*
lemma *Fin-iless-Infty*: *Fin n* < ∞ **by** *simp*
lemma *not-Infty-ile-Fin*: ¬ ∞ ≤ *Fin n* **by** *simp*

lemmas *Fin-ile-Infty* = *Infty-ub*

lemmas *neq-Infty-Fin-conv* = *not-Infty-eq*

lemma *Infty-le-eq*: (∞ ≤ *n*) = (*n* = ∞) **by** *simp*

lemma *ile-trans*: ⟦ (*i*::*inat*) ≤ *j*; *j* ≤ *k* ⟧ ⟹ *i* ≤ *k* **by** *simp*
lemma *ile-anti-sym*: ⟦ (*m*::*inat*) ≤ *n*; *n* ≤ *m* ⟧ ⟹ *m* = *n* **by** *simp*
lemma *iless-ile*: ((*m*::*inat*) < *n*) = (*m* ≤ *n* ∧ *m* ≠ *n*) **by** (*safe*, *simp-all*)
lemma *ile-linear*: (*m*::*inat*) ≤ *n* ∨ *n* ≤ *m* **by** (*safe*, *simp-all*)

thm *neq0-conv*
lemma *ineq0-conv*: $(n \neq (0::inat)) = (0 < n)$
by (*case-tac n*, *simp-all*)
lemmas *ineq0-conv-Fin*[*simp*] = *ineq0-conv*[*unfolded zero-inat-def*]

lemmas *not-iless0* = *not-ilessi0*
lemma *iless-iSuc0*: $(n < iSuc\ 0) = (n = 0)$
by (*case-tac n*, *simp-all*)
lemmas *iless-iSuc0-Fin*[*simp*] = *iless-iSuc0*[*unfolded zero-inat-def*]

lemmas *ile-0-eq* = *i0-neq*

lemma *neq-Infty-imp-ex-Fin*: $n \neq \infty \Longrightarrow \exists\, nat.\ n = Fin\ nat$
by (*case-tac n*, *simp-all*)
corollary *less-Infty-imp-ex-Fin*: $n < \infty \Longrightarrow \exists\, nat.\ n = Fin\ nat$
thm *neq-Infty-imp-ex-Fin*[*OF less-imp-neq*]
by (*rule neq-Infty-imp-ex-Fin*[*OF less-imp-neq*])

3.1.3 Addition and difference

lemma *iadd-Fin-Fin*: $Fin\ a + Fin\ b = Fin\ (a + b)$ **by** *simp*
lemma *iadd-Infty*: $\infty + n = \infty$ **by** *simp*
lemma *iadd-Infty-right*: $n + \infty = \infty$ **by** *simp*

lemma *idiff-Fin-Fin*[*simp*,*code*]: $Fin\ a - Fin\ b = Fin\ (a - b)$
unfolding *diff-inat-def* **by** *simp*
lemma *idiff-Infty*[*simp*,*code*]: $\infty - n = \infty$
unfolding *diff-inat-def* **by** *simp*
lemma *idiff-Infty-right*[*simp*,*code*]: $Fin\ a - \infty = 0$
unfolding *diff-inat-def* **by** *simp*

lemma *idiff-0*: $(n::inat) - 0 = n$
by (*case-tac n*, *simp-all*)
lemmas *idiff-0-Fin*[*simp*,*code*] = *idiff-0*[*unfolded zero-inat-def*]
lemma *idiff-0-eq-0*: $(0::inat) - n = 0$
by (*case-tac n*, *simp-all*)
lemmas *idiff-0-eq-0-Fin*[*simp*,*code*] = *idiff-0-eq-0*[*unfolded zero-inat-def*]

lemma *diff-eq-conv-nat*: $(x - y = (z::nat)) = (if\ y < x\ then\ x = y + z\ else\ z = 0)$
by *auto*
lemma *idiff-eq-conv*:

$(x - y = (z::inat)) =$
$(if\ y < x\ then\ x = y + z\ else\ if\ x \neq \infty\ then\ z = 0\ else\ z = \infty)$
by (*case-tac x*, *case-tac y*, *case-tac z*, *auto*, *case-tac z*, *auto*)
lemmas *idiff-eq-conv-Fin* = *idiff-eq-conv*[*unfolded zero-inat-def*]

lemma *idiff-self-eq-0*: $n \neq \infty \Longrightarrow (n::inat) - n = 0$
by *fastsimp*
lemmas *idiff-self-eq-0-Fin* = *idiff-self-eq-0*[*unfolded zero-inat-def*]

lemma *less-eq-idiff-eq-sum*: $y \leq (x::inat) \Longrightarrow (z \leq x - y) = (z + y \leq x)$
by (*case-tac x*, *case-tac y*, *case-tac z*, *fastsimp*+)

lemma *iSuc-pred*: $0 < n \Longrightarrow iSuc\ (n - iSuc\ 0) = n$
apply (*case-tac n*)
apply (*simp add*: *iSuc-Fin*)+
done
lemmas *iSuc-pred-Fin* = *iSuc-pred*[*unfolded zero-inat-def*]
lemma *iadd-0*: $(0::inat) + n = n$
by (*rule monoid-add-class.add-0-left*)
lemmas *iadd-0-Fin*[*simp*, *code*] = *iadd-0*[*unfolded zero-inat-def*]
lemma *iadd-0-right*: $n + (0::inat) = n$
by (*rule monoid-add-class.add-0-right*)
lemmas *iadd-0-right-Fin*[*simp*, *code*] = *iadd-0-right*[*unfolded zero-inat-def*]

lemma *iadd-commute*: $(a::inat) + b = b + a$
by (*rule ab-semigroup-add-class.add-commute*)
thm *add-assoc*
lemma *iadd-assoc*: $(a::inat) + b + c = a + (b + c)$
by (*rule semigroup-add-class.add.assoc*)

thm *add-Suc*
lemma *iadd-Suc*: $iSuc\ m + n = iSuc\ (m + n)$
apply (*case-tac m*, *case-tac n*)
apply (*simp add*: *iSuc-Fin*)+
done

thm *add-Suc-right*
lemma *iadd-Suc-right*: $m + iSuc\ n = iSuc\ (m + n)$
by (*simp only*: *iadd-commute*[*of m*] *iadd-Suc*)

lemma *mono-iSuc*: *mono iSuc*
unfolding *mono-def* **by** *simp*

thm *add-is-0*[*no-vars*]
lemma *iadd-is-0*: $(m + n = (0::inat)) = (m = 0 \wedge n = 0)$
by (*case-tac m*, *case-tac n*, *simp-all*)
lemmas *iadd-is-0-Fin* = *iadd-is-0*[*unfolded zero-inat-def*]

lemma *ile-add1*: $(n::inat) \leq n + m$
by (*case-tac m*, *case-tac n*, *simp-all*)
lemma *ile-add2*: $(n::inat) \leq m + n$
by (*simp only*: *iadd-commute*[*of m*] *ile-add1*)

lemma *iadd-ile-mono*: $[\![(i::inat) \leq j; k \leq l]\!] \Longrightarrow i + k \leq j + l$
by (*rule add-mono*)
lemma *iadd-ile-mono1*: $(i::inat) \leq j \Longrightarrow i + k \leq j + k$
by (*rule add-right-mono*)
lemma *iadd-ile-mono2*: $(i::inat) \leq j \Longrightarrow k + i \leq k + j$
by (*rule add-left-mono*)

lemma *iadd-iless-mono*: $[\![(i::inat) < j; k < l]\!] \Longrightarrow i + k < j + l$
by (*case-tac i*, *case-tac k*, *case-tac j*, *case-tac l*, *simp-all*)

lemma *trans-ile-iadd1*: $i \leq (j::inat) \Longrightarrow i \leq j + m$
by (*rule order-trans*[*OF - ile-add1*])
lemma *trans-ile-iadd2*: $i \leq (j::inat) \Longrightarrow i \leq m + j$
by (*rule order-trans*[*OF - ile-add2*])

lemma *trans-iless-iadd1*: $i < (j::inat) \Longrightarrow i < j + m$
by (*rule order-less-le-trans*[*OF - ile-add1*])
lemma *trans-iless-iadd2*: $i < (j::inat) \Longrightarrow i < m + j$
by (*rule order-less-le-trans*[*OF - ile-add2*])

thm *add-leD1*[*no-vars*]
lemma *iadd-ileD1*: $m + k \leq (n::inat) \Longrightarrow m \leq n$
by (*case-tac m*, *case-tac n*, *case-tac k*, *simp-all*)
lemma *iadd-ileD2*: $m + k \leq (n::inat) \Longrightarrow k \leq n$
by (*rule iadd-ileD1*, *simp only*: *iadd-commute*[*of m*])

thm *diff-le-mono*
lemma *idiff-ile-mono*: $m \leq (n::inat) \Longrightarrow m - l \leq n - l$
by (*case-tac m*, *case-tac n*, *case-tac l*, *simp-all*)
thm *diff-le-mono2*
lemma *idiff-ile-mono2*: $m \leq (n::inat) \Longrightarrow l - n \leq l - m$
by (*case-tac m*, *case-tac n*, *case-tac l*, *simp-all*, *case-tac l*, *simp-all*)

thm *diff-less-mono*
lemma *idiff-iless-mono*: $[\![m < (n::inat); l \leq m]\!] \Longrightarrow m - l < n - l$
by (*case-tac m*, *case-tac n*, *case-tac l*, *simp-all*, *case-tac l*, *simp-all*)
thm *diff-less-mono2*
lemma *idiff-iless-mono2*: $[\![m < (n::inat); m < l]\!] \Longrightarrow l - n \leq l - m$
by (*case-tac m*, *case-tac n*, *case-tac l*, *simp-all*, *case-tac l*, *simp-all*)

3.1.4 Multiplication and division

lemma *imult-Fin-Fin*: $Fin\ a * Fin\ b = Fin\ (a * b)$ **by** *simp*
lemma *imult-Infty*: $(0::inat) < n \Longrightarrow \infty * n = \infty$
by (*case-tac n*, *simp-all*)
lemmas *imult-Infty-Fin*[*simp*] = *imult-Infty*[*unfolded zero-inat-def*]
lemma *imult-Infty-right*: $(0::inat) < n \Longrightarrow n * \infty = \infty$
by (*case-tac n*, *simp-all*)
lemmas *imult-Infty-right-Fin*[*simp*] = *imult-Infty-right*[*unfolded zero-inat-def*]

lemma *idiv-Fin-Fin*[*simp*, *code*]: $Fin\ a\ div\ Fin\ b = Fin\ (a\ div\ b)$
unfolding *div-inat-def* **by** *simp*
lemma *idiv-Infty*: $0 < n \Longrightarrow \infty\ div\ n = \infty$
unfolding *div-inat-def*
apply (*case-tac n*, *simp-all*)
apply (*rename-tac n1*, *case-tac n1*, *simp-all*)
done
lemmas *idiv-Infty-Fin*[*simp*] = *idiv-Infty*[*unfolded zero-inat-def*]

lemma *idiv-Infty-right*[*simp*]: $n \neq \infty \Longrightarrow n\ div\ \infty = 0$
unfolding *div-inat-def* **by** (*case-tac n*, *simp-all*)

lemma *idiv-Infty-if*: $n\ div\ \infty = (if\ n = \infty\ then\ \infty\ else\ 0)$
unfolding *div-inat-def*
by (*case-tac n*, *simp-all*)

lemmas *idiv-Infty-if-Fin* = *idiv-Infty-if*[*unfolded zero-inat-def*]

lemma *imult-0*: $0 * (m::inat) = 0$
by (*case-tac m*, *simp-all*)
lemmas *imult-0-Fin*[*simp*, *code*] = *imult-0*[*unfolded zero-inat-def*]
lemma *imult-0-right*: $(m::inat) * 0 = 0$
by (*case-tac m*, *simp-all*)
lemmas *imult-0-right-Fin*[*simp*, *code*] = *imult-0-right*[*unfolded zero-inat-def*]

lemma *imult-is-0*: $((m::inat) * n = 0) = (m = 0 \vee n = 0)$
apply (*case-tac m*, *case-tac n*, *simp-all*)
apply (*case-tac n*, *simp-all*)
done
lemma *inat-0-less-mult-iff*: $(0 < (m::inat) * n) = (0 < m \wedge 0 < n)$
by (*simp only*: *ineq0-conv*[*symmetric*] *imult-is-0*, *simp*)
lemmas *imult-is-0-Fin* = *imult-is-0*[*unfolded zero-inat-def*]
lemmas *inat-0-less-mult-iff-Fin* = *inat-0-less-mult-iff*[*unfolded zero-inat-def*]

lemma *imult-commute*: $(a::inat) * b = b * a$
by (*rule ab-semigroup-mult-class.mult-commute*)

lemma *imult-Infty-if*: $\infty * n = (if\ n = 0\ then\ 0\ else\ \infty)$
by (*case-tac n*, *simp-all*)

lemma *imult-Infty-right-if*: $n * \infty = (\mathit{if}\ n = 0\ \mathit{then}\ 0\ \mathit{else}\ \infty)$
by (*case-tac n*, *simp-all*)
lemmas *imult-Infty-if-Fin* = *imult-Infty-if*[*unfolded zero-inat-def*]
lemmas *imult-Infty-right-if-Fin* = *imult-Infty-right-if*[*unfolded zero-inat-def*]

lemma *imult-is-Infty*: $((a::inat) * b = \infty) = (a = \infty \wedge b \neq 0 \vee b = \infty \wedge a \neq 0)$
apply (*case-tac a*, *case-tac b*, *simp-all*)
apply (*case-tac b*, *simp-all*)
done
lemmas *imult-is-Infty-Fin* = *imult-is-Infty*[*unfolded zero-inat-def*]

lemma *imult-assoc*: $(a::inat) * b * c = a * (b * c)$
by (*rule semigroup-mult-class.mult.assoc*)

lemma *idiv-by-0*: $(a::inat)\ \mathit{div}\ 0 = 0$
unfolding *div-inat-def* **by** (*case-tac a*, *simp-all*)
lemmas *idiv-by-0-Fin*[*simp*, *code*] = *idiv-by-0*[*unfolded zero-inat-def*]

lemma *idiv-0*: $0\ \mathit{div}\ (a::inat) = 0$
unfolding *div-inat-def* **by** (*case-tac a*, *simp-all*)
lemmas *idiv-0-Fin*[*simp*, *code*] = *idiv-0*[*unfolded zero-inat-def*]

thm *mod-by-0*
lemma *imod-by-0*: $(a::inat)\ \mathit{mod}\ 0 = a$
unfolding *mod-inat-def* **by** (*case-tac a*, *simp-all*)
lemmas *imod-by-0-Fin*[*simp*, *code*] = *imod-by-0*[*unfolded zero-inat-def*]

lemma *imod-0*: $0\ \mathit{mod}\ (a::inat) = 0$
unfolding *mod-inat-def* **by** (*case-tac a*, *simp-all*)
lemmas *imod-0-Fin*[*simp*, *code*] = *imod-0*[*unfolded zero-inat-def*]

lemma *imod-Fin-Fin*[*simp*, *code*]: $\mathit{Fin}\ a\ \mathit{mod}\ \mathit{Fin}\ b = \mathit{Fin}\ (a\ \mathit{mod}\ b)$
unfolding *mod-inat-def* **by** *simp*
lemma *imod-Infty*[*simp*, *code*]: $\infty\ \mathit{mod}\ n = \infty$
unfolding *mod-inat-def* **by** *simp*
lemma *imod-Infty-right*[*simp*, *code*]: $n\ \mathit{mod}\ \infty = n$
unfolding *mod-inat-def* **by** (*case-tac n*) *simp-all*

lemma *idiv-self*: $[\![\ 0 < (n::inat);\ n \neq \infty\ ]\!] \Longrightarrow n\ \mathit{div}\ n = 1$
by (*case-tac n*, *simp-all add*: *one-inat-def*)
lemma *imod-self*: $n \neq \infty \Longrightarrow (n::inat)\ \mathit{mod}\ n = 0$
by (*case-tac n*, *simp-all*)

lemma *idiv-iless*: $m < (n::inat) \Longrightarrow m\ \mathit{div}\ n = 0$
by (*case-tac m*, *simp-all*) (*case-tac n*, *simp-all*)
lemma *imod-iless*: $m < (n::inat) \Longrightarrow m\ \mathit{mod}\ n = m$
by (*case-tac m*, *simp-all*) (*case-tac n*, *simp-all*)

lemma *imod-iless-divisor*: $[\![\ 0 < (n::inat);\ m \neq \infty\ ]\!] \Longrightarrow m\ \mathit{mod}\ n < n$

by (*case-tac m*, *simp-all*) (*case-tac n*, *simp-all*)
lemma *imod-ile-dividend*: (*m*::*inat*) *mod n* ≤ *m*
by (*case-tac m*, *simp-all*) (*case-tac n*, *simp-all*)
lemma *idiv-ile-dividend*: (*m*::*inat*) *div n* ≤ *m*
by (*case-tac m*, *simp-all*) (*case-tac n*, *simp-all*)

thm *div-mult2-eq*
lemma *idiv-imult2-eq*: (*a*::*inat*) *div* (*b* ∗ *c*) = *a div b div c*
apply (*case-tac a*, *case-tac b*, *case-tac c*, *simp add*: *div-mult2-eq*)
apply (*simp add*: *imult-Infty-right-if idiv-Infty-right*)
apply (*simp add*: *imult-Infty-if idiv-Infty-right idiv-0*[*unfolded zero-inat-def*])
apply (*case-tac b* = *0*, *simp*)
apply (*case-tac c* = *0*, *simp*)
thm *idiv-Infty*
thm *idiv-Infty*[*OF inat-0-less-mult-iff*[*THEN iffD2*]]
apply (*simp add*: *idiv-Infty*[*OF inat-0-less-mult-iff*[*THEN iffD2*]])
done

thm *add-mult-distrib*
lemma *iadd-imult-distrib*: ((*m*::*inat*) + *n*) ∗ *k* = *m* ∗ *k* + *n* ∗ *k*
by (*rule left-distrib*)

thm *add-mult-distrib2*
lemma *iadd-imult-distrib2*: *k* ∗ ((*m*::*inat*) + *n*) = *k* ∗ *m* + *k* ∗ *n*
by (*simp only*: *imult-commute*[*of k*] *iadd-imult-distrib*)

thm *mult-le-mono*
lemma *imult-ile-mono*: ⟦ (*i*::*inat*) ≤ *j*; *k* ≤ *l* ⟧ ⟹ *i* ∗ *k* ≤ *j* ∗ *l*
apply (*case-tac i*, *case-tac j*, *case-tac k*, *case-tac l*, *simp-all add*: *mult-le-mono*)
apply (*case-tac k*, *case-tac l*, *simp-all*)
apply (*case-tac k*, *case-tac l*, *simp-all*)
done

lemma *imult-ile-mono1*: (*i*::*inat*) ≤ *j* ⟹ *i* ∗ *k* ≤ *j* ∗ *k*
by (*rule imult-ile-mono*[*OF - ile-refl*])
thm *mult-le-mono2*
lemma *imult-ile-mono2*: (*i*::*inat*) ≤ *j* ⟹ *k* ∗ *i* ≤ *k* ∗ *j*
by (*rule imult-ile-mono*[*OF ile-refl*])

lemma *imult-iless-mono1*: ⟦ (*i*::*inat*) < *j*; *0* < *k*; *k* ≠ ∞ ⟧ ⟹ *i* ∗ *k* ≤ *j* ∗ *k*
by (*case-tac i*, *case-tac j*, *case-tac k*, *simp-all*)
lemma *imult-iless-mono2*: ⟦ (*i*::*inat*) < *j*; *0* < *k*; *k* ≠ ∞ ⟧ ⟹ *k* ∗ *i* ≤ *k* ∗ *j*
by (*simp only*: *imult-commute*[*of k*], *rule imult-iless-mono1*)

lemma *imod-1*: (*Fin m*) *mod iSuc 0* = *0*
by (*simp add*: *iSuc-Fin*)

lemmas *imod-1-Fin*[*simp*, *code*] = *imod-1*[*unfolded zero-inat-def*]

lemma *imod-iadd-self2*: (*m* + *Fin n*) *mod* (*Fin n*) = *m mod* (*Fin n*)
by (*case-tac m*, *simp-all*)

lemma *imod-iadd-self1*: (*Fin n* + *m*) *mod* (*Fin n*) = *m mod* (*Fin n*)
by (*simp only*: *iadd-commute*[*of* - *m*] *imod-iadd-self2*)

lemma *idiv-imod-equality*: (*m*::*inat*) *div n* ∗ *n* + *m mod n* + *k* = *m* + *k*
by (*case-tac m*, *simp-all*) (*case-tac n*, *simp-all*)
lemma *imod-idiv-equality*: (*m*::*inat*) *div n* ∗ *n* + *m mod n* = *m*
by (*insert idiv-imod-equality*[*of m n 0*], *simp*)

lemma *idiv-ile-mono*: *m* ≤ (*n*::*inat*) ⟹ *m div k* ≤ *n div k*
apply (*case-tac k* = *0*, *simp*)
apply (*case-tac m*, *case-tac k*, *simp-all*)
apply (*case-tac n*)
apply (*simp add*: *div-le-mono*)
apply (*simp add*: *idiv-Infty*)
apply (*simp add*: *i0-lb*[*unfolded zero-inat-def*])
done
lemma *idiv-ile-mono2*: ⟦ *0* < *m*; *m* ≤ (*n*::*inat*) ⟧ ⟹ *k div n* ≤ *k div m*
apply (*case-tac n* = *0*, *simp*)
apply (*case-tac m*, *case-tac k*, *simp-all*)
apply (*case-tac n*)
apply (*simp add*: *div-le-mono2*)
apply *simp*
done

end

4 Util-Nat: Results for natural arithmetics

theory *Util-Nat*
imports *Nat*
begin

4.1 Some convenience arithmetic lemmata

thm *Nat.add-Suc-right*
lemma *add-1-Suc-conv*: *m* + *1* = *Suc m* **by** *simp*
lemma *sub-Suc0-sub-Suc-conv*: *b* − *a* − *Suc 0* = *b* − *Suc a* **by** *simp*
thm *Nat.Suc-pred*
lemma *Suc-diff-Suc*: *m* < *n* ⟹ *Suc* (*n* − *Suc m*) = *n* − *m*
apply (*rule subst*[*OF sub-Suc0-sub-Suc-conv*])
apply (*rule Suc-pred*)
apply (*simp only*: *zero-less-diff*)

done

lemma *nat-grSuc0-conv*: $(Suc\ 0 < n) = (n \neq 0 \wedge n \neq Suc\ 0)$
by *fastsimp*
lemma *nat-geSucSuc0-conv*: $(Suc\ (Suc\ 0) \leq n) = (n \neq 0 \wedge n \neq Suc\ 0)$
by *fastsimp*

lemma *nat-lessSucSuc0-conv*: $(n < Suc\ (Suc\ 0)) = (n = 0 \vee n = Suc\ 0)$
by *fastsimp*
lemma *nat-leSuc0-conv*: $(n \leq Suc\ 0) = (n = 0 \vee n = Suc\ 0)$
by *fastsimp*

thm *Nat.mult-Suc*
lemma *mult-pred*: $(m - Suc\ 0) * n = m * n - n$
by (*simp add*: *diff-mult-distrib*)
thm *Nat.mult-Suc-right*
lemma *mult-pred-right*: $m * (n - Suc\ 0) = m * n - m$
by (*simp add*: *diff-mult-distrib2*)

lemma *gr-implies-gr0*: $m < (n::nat) \Longrightarrow 0 < n$ **by** *simp*

thm
Nat.mult-cancel1
Nat.mult-cancel1
corollary *mult-cancel1-gr0*:
$(0::nat) < k \Longrightarrow (k * m = k * n) = (m = n)$ **by** *simp*
corollary *mult-cancel2-gr0*:
$(0::nat) < k \Longrightarrow (m * k = n * k) = (m = n)$ **by** *simp*

thm
Nat.mult-le-cancel1
Nat.mult-le-cancel2
corollary *mult-le-cancel1-gr0*:
$(0::nat) < k \Longrightarrow (k * m \leq k * n) = (m \leq n)$ **by** *simp*
corollary *mult-le-cancel2-gr0*:
$(0::nat) < k \Longrightarrow (m * k \leq n * k) = (m \leq n)$ **by** *simp*

thm *mult-le-mono*
lemma *gr0-imp-self-le-mult1*: $0 < (k::nat) \Longrightarrow m \leq m * k$
by (*drule Suc-leI*, *drule mult-le-mono*[*OF order-refl*], *simp*)

lemma *gr0-imp-self-le-mult2*: $0 < (k::nat) \Longrightarrow m \leq k * m$
by (*subst mult-commute*, *rule gr0-imp-self-le-mult1*)

lemma *less-imp-Suc-mult-le*: $m < n \Longrightarrow Suc\ m * k \leq n * k$
by (*rule mult-le-mono1*, *simp*)

lemma *less-imp-Suc-mult-pred-less*: $[\![\ m < n;\ 0 < k\ ]\!] \Longrightarrow Suc\ m * k - Suc\ 0 < n * k$
apply (*rule Suc-le-lessD*)
apply (*simp only*: *Suc-pred*[*OF nat-0-less-mult-iff*[*THEN iffD2*, *OF conjI*, *OF zero-less-Suc*]])
apply (*rule less-imp-Suc-mult-le*, *assumption*)
done

thm *Nat.zero-less-diff*
lemma *ord-zero-less-diff*: $(0 < (b::'a::ordered\text{-}ab\text{-}group\text{-}add) - a) = (a < b)$
by (*simp add*: *less-diff-eq*)

lemma *ord-zero-le-diff*: $(0 \leq (b::'a::ordered\text{-}ab\text{-}group\text{-}add) - a) = (a \leq b)$
by (*simp add*: *le-diff-eq*)

diff-diff-right in rule format
lemmas *diff-diff-right* = *Nat.diff-diff-right*[*rule-format*]

thm *Nat.le-add1 Nat.le-add2*
lemma *less-add1*: $(0::nat) < j \Longrightarrow i < i + j$ **by** *simp*
lemma *less-add2*: $(0::nat) < j \Longrightarrow i < j + i$ **by** *simp*

thm *Nat.add-leD1 Nat.add-leD2*
thm *Nat.add-lessD1*
lemma *add-lessD2*: $i + j < (k::nat) \Longrightarrow j < k$ **by** *simp*

thm *Nat.add-le-mono1*
lemma *add-le-mono2*: $i \leq (j::nat) \Longrightarrow k + i \leq k + j$ **by** *simp*
thm *Nat.add-less-mono1*
lemma *add-less-mono2*: $i < (j::nat) \Longrightarrow k + i < k + j$ **by** *simp*

thm *Nat.diff-le-self*
lemma *diff-less-self*: $[\![\ (0::nat) < i;\ \ 0 < j\ ]\!] \Longrightarrow i - j < i$ **by** *simp*

lemma
  ge-less-neq-conv: $((a::'a::linorder) \leq n) = (\forall x.\ x < a \longrightarrow n \neq x)$ **and**
  le-greater-neq-conv: $(n \leq (a::'a::linorder)) = (\forall x.\ a < x \longrightarrow n \neq x)$
by (*subst linorder-not-less*[*symmetric*], *blast*)+
lemma
  greater-le-neq-conv: $((a::'a::linorder) < n) = (\forall x.\ x \leq a \longrightarrow n \neq x)$ **and**
  less-ge-neq-conv: $(n < (a::'a::linorder)) = (\forall x.\ a \leq x \longrightarrow n \neq x)$
by (*subst linorder-not-le*[*symmetric*], *blast*)+

Lemmas for @termabs function

lemma *leq-pos-imp-abs-leq*: ⟦ $0 \leq (a::'a::ordered\text{-}ab\text{-}group\text{-}add\text{-}abs)$; $a \leq b$ ⟧ ⟹ $|a| \leq |b|$
by *simp*
lemma *leq-neg-imp-abs-geq*: ⟦ $(a::'a::ordered\text{-}ab\text{-}group\text{-}add\text{-}abs) \leq 0$; $b \leq a$ ⟧ ⟹ $|a| \leq |b|$
by *simp*
lemma *abs-range*: ⟦ $0 \leq (a::'a::\{ordered\text{-}ab\text{-}group\text{-}add\text{-}abs,abs\text{-}if\})$; $-a \leq x$; $x \leq a$ ⟧ ⟹ $|x| \leq a$
apply (*clarsimp simp*: *abs-if*)
thm *neg-le-iff-le*[*THEN iffD1*]
apply (*rule neg-le-iff-le*[*THEN iffD1*], *simp*)
done

Lemmas for @termsgn function

lemma *sgn-abs*:$(x::'a::linordered\text{-}idom) \neq 0 \Longrightarrow |sgn\ x| = 1$
by (*case-tac* $x < 0$, *simp+*)
lemma *sgn-mult-abs*:$|x| * |sgn\ (a::'a::linordered\text{-}idom)| = |x * sgn\ a|$
by (*fastsimp simp add*: *sgn-if abs-if*)
lemma *abs-imp-sgn-abs*: $|a| = |b| \Longrightarrow |sgn\ (a::'a::linordered\text{-}idom)| = |sgn\ b|$
by (*fastsimp simp add*: *abs-if*)
lemma *sgn-mono*: $a \leq b \Longrightarrow sgn\ (a::'a::\{linordered\text{-}idom,linordered\text{-}semidom\}) \leq sgn\ b$
by (*simp add*: *sgn-if*, *safe*, *simp-all*)

4.2 Additional facts about inequalities

thm *Nat.le-add-diff*
lemma *add-diff-le*: $k \leq n \Longrightarrow m + k - n \leq (m::nat)$
by (*case-tac* $m + k < n$, *simp-all*)
thm
Nat.le-add-diff
add-diff-le

lemma *less-add-diff*: $k < (n::nat) \Longrightarrow m < n + m - k$
thm *add-less-imp-less-right*[*of - k*]
by (*rule add-less-imp-less-right*[*of - k*], *simp*)
thm *add-diff-le*
lemma *add-diff-less*: ⟦ $k < n$; $0 < m$ ⟧ ⟹ $m + k - n < (m::nat)$
by (*case-tac* $m + k < n$, *simp-all*)
thm
Nat.le-add-diff
add-diff-le
less-add-diff
add-diff-less

thm *Nat.less-diff-conv*
lemma *add-le-imp-le-diff1*: $i + k \leq j \Longrightarrow i \leq j - (k::nat)$

by (*case-tac* $k \leq j$, *simp-all*)
lemma *add-le-imp-le-diff2*: $k + i \leq j \Longrightarrow i \leq j - (k::nat)$ **by** *simp*
thm
Nat.less-diff-conv[*symmetric*]
Nat.le-diff-conv2[*symmetric*]
add-le-imp-le-diff1
add-le-imp-le-diff2

thm
Nat.le-diff-conv Nat.le-diff-conv2
Nat.less-diff-conv
lemma *diff-less-imp-less-add*: $j - (k::nat) < i \Longrightarrow j < i + k$ **by** *simp*
thm *Nat.le-diff-conv*
lemma *diff-less-conv*: $0 < i \Longrightarrow (j - (k::nat) < i) = (j < i + k)$
by (*safe*, *simp-all*)

lemma *le-diff-swap*: $[\![i \leq (k::nat); j \leq k]\!] \Longrightarrow (k - j \leq i) = (k - i \leq j)$
by (*safe*, *simp-all*)

lemma *diff-less-imp-swap*: $[\![0 < (i::nat); k - i < j]\!] \Longrightarrow (k - j < i)$ **by** *simp*
lemma *diff-less-swap*: $[\![0 < (i::nat); 0 < j]\!] \Longrightarrow (k - j < i) = (k - i < j)$
by (*blast intro*: *diff-less-imp-swap*)

lemma *less-diff-imp-less*: $(i::nat) < j - m \Longrightarrow i < j$ **by** *simp*
lemma *le-diff-imp-le*: $(i::nat) \leq j - m \Longrightarrow i \leq j$ **by** *simp*

lemma *less-diff-le-imp-less*: $[\![(i::nat) < j - m; n \leq m]\!] \Longrightarrow i < j - n$ **by** *simp*
lemma *le-diff-le-imp-le*: $[\![(i::nat) \leq j - m; n \leq m]\!] \Longrightarrow i \leq j - n$ **by** *simp*

thm *Nat.less-imp-diff-less*
lemma *le-imp-diff-le*: $(j::nat) \leq k \Longrightarrow j - n \leq k$ **by** *simp*

4.3 Inequalities for Suc and pred

thm *Nat.less-Suc-eq-le*
corollary *less-eq-le-pred*: $0 < (n::nat) \Longrightarrow (m < n) = (m \leq n - Suc\ 0)$
by (*safe*, *simp-all*)
corollary *less-imp-le-pred*: $m < n \Longrightarrow m \leq n - Suc\ 0$ **by** *simp*
corollary *le-pred-imp-less*: $[\![0 < n; m \leq n - Suc\ 0]\!] \Longrightarrow m < n$ **by** *simp*

thm *Nat.Suc-le-eq*
corollary *pred-less-eq-le*: $0 < m \Longrightarrow (m - Suc\ 0 < n) = (m \leq n)$
by (*safe*, *simp-all*)
corollary *pred-less-imp-le*: $m - Suc\ 0 < n \Longrightarrow m \leq n$ **by** *simp*
corollary *le-imp-pred-less*: $[\![0 < m; m \leq n]\!] \Longrightarrow m - Suc\ 0 < n$ **by** *simp*

thm *Nat.diff-add-inverse*
lemma *diff-add-inverse-Suc*: $n < m \Longrightarrow n + (m - Suc\ n) = m - Suc\ 0$ **by** *simp*

thm *Nat.Suc-mono*
lemma *pred-mono*: $[\![\ m < n;\ 0 < m\ ]\!] \Longrightarrow m - Suc\ 0 < n - Suc\ 0$ **by** *simp*
corollary *pred-Suc-mono*: $[\![\ m < Suc\ n;\ 0 < m\ ]\!] \Longrightarrow m - Suc\ 0 < n$ **by** *simp*

lemma *Suc-less-pred-conv*: $(Suc\ m < n) = (m < n - Suc\ 0)$ **by** (*safe*, *simp-all*)
lemma *Suc-le-pred-conv*: $0 < n \Longrightarrow (Suc\ m \leq n) = (m \leq n - Suc\ 0)$ **by** (*safe*, *simp-all*)
lemma *Suc-le-imp-le-pred*: $Suc\ m \leq n \Longrightarrow m \leq n - Suc\ 0$ **by** *simp*

4.4 Additional facts about cancellation in (in-)equalities

lemma *diff-cancel-imp-eq*: $[\![\ 0 < (n::nat);\ \ n + i - j = n\ ]\!] \Longrightarrow i = j$ **by** *simp*

thm
Nat.nat-add-left-cancel-less
Nat.nat-add-left-cancel-le
Nat.nat-add-right-cancel
Nat.nat-add-left-cancel
Nat.diff-diff-eq
Nat.eq-diff-iff
Nat.less-diff-iff
Nat.le-diff-iff
lemma *nat-diff-left-cancel-less*: $k - m < k - (n::nat) \Longrightarrow n < m$ **by** *simp*
lemma *nat-diff-right-cancel-less*: $n - k < (m::nat) - k \Longrightarrow n < m$ **by** *simp*

lemma *nat-diff-left-cancel-le1*: $[\![\ k - m \leq k - (n::nat);\ m < k\ ]\!] \Longrightarrow n \leq m$ **by** *simp*
lemma *nat-diff-left-cancel-le2*: $[\![\ k - m \leq k - (n::nat);\ n \leq k\ ]\!] \Longrightarrow n \leq m$ **by** *simp*

lemma *nat-diff-right-cancel-le1*: $[\![\ m - k \leq n - (k::nat);\ k < m\ ]\!] \Longrightarrow m \leq n$ **by** *simp*
lemma *nat-diff-right-cancel-le2*: $[\![\ m - k \leq n - (k::nat);\ k \leq n\ ]\!] \Longrightarrow m \leq n$ **by** *simp*

lemma *nat-diff-left-cancel-eq1*: $[\![\ k - m = k - (n::nat);\ m < k\ ]\!] \Longrightarrow m = n$ **by** *simp*
lemma *nat-diff-left-cancel-eq2*: $[\![\ k - m = k - (n::nat);\ n < k\ ]\!] \Longrightarrow m = n$ **by** *simp*

lemma *nat-diff-right-cancel-eq1*: $[\![\ m - k = n - (k::nat);\ k < m\ ]\!] \Longrightarrow m = n$ **by** *simp*
lemma *nat-diff-right-cancel-eq2*: $[\![\ m - k = n - (k::nat);\ k < n\ ]\!] \Longrightarrow m = n$ **by** *simp*

thm *eq-diff-iff*
lemma *eq-diff-left-iff*: $[\![(m::nat) \leq k; n \leq k]\!] \Longrightarrow (k - m = k - n) = (m = n)$
by (*safe*, *simp-all*)

thm *Nat.nat-add-right-cancel Nat.nat-add-left-cancel*
thm *Nat.diff-le-mono*
lemma *eq-imp-diff-eq*: $m = (n::nat) \Longrightarrow m - k = n - k$ **by** *simp*

List of definitions and lemmas

thm
Nat.add-Suc-right
add-1-Suc-conv
sub-Suc0-sub-Suc-conv

thm
Nat.mult-cancel1
Nat.mult-cancel2
mult-cancel1-gr0
mult-cancel2-gr0

thm
Nat.add-lessD1
add-lessD2

thm
Nat.zero-less-diff
ord-zero-less-diff
ord-zero-le-diff

thm
Nat.le-add-diff
add-diff-le
less-add-diff
add-diff-less

thm
Nat.le-diff-conv Nat.le-diff-conv2
Nat.less-diff-conv
diff-less-imp-less-add
diff-less-conv

thm
le-diff-swap
diff-less-imp-swap
diff-less-swap

thm
less-diff-imp-less
le-diff-imp-le

thm
less-diff-le-imp-less
le-diff-le-imp-le

thm
Nat.less-imp-diff-less
le-imp-diff-le

thm
Nat.less-Suc-eq-le
less-eq-le-pred
less-imp-le-pred
le-pred-imp-less

thm
Nat.Suc-le-eq
pred-less-eq-le
pred-less-imp-le
le-imp-pred-less

thm
diff-cancel-imp-eq
thm
diff-add-inverse-Suc
thm
Nat.nat-add-left-cancel-less
Nat.nat-add-left-cancel-le
Nat.nat-add-right-cancel
Nat.nat-add-left-cancel
Nat.eq-diff-iff
Nat.less-diff-iff
Nat.le-diff-iff
thm
nat-diff-left-cancel-less
nat-diff-right-cancel-less
thm
nat-diff-left-cancel-le1
nat-diff-left-cancel-le2
nat-diff-right-cancel-le1
nat-diff-right-cancel-le2
thm
nat-diff-left-cancel-eq1
nat-diff-left-cancel-eq2
nat-diff-right-cancel-eq1
nat-diff-right-cancel-eq2

thm
Nat.eq-diff-iff

eq-diff-left-iff

thm
Nat.nat-add-right-cancel Nat.nat-add-left-cancel
Nat.diff-le-mono
eq-imp-diff-eq

end

5 Util-Div: Results for division and modulo operators on integers

theory *Util-Div*
imports *Util-Nat Presburger Sledgehammer*
begin

5.1 Additional (in-)equalities with *div* and *mod*

thm *Divides.mod-less-divisor*
corollary *Suc-mod-le-divisor*: $0 < m \Longrightarrow Suc\ (n\ mod\ m) \leq m$
by (*rule Suc-leI*, *rule mod-less-divisor*)

thm *Divides.mod-less*
thm *Divides.mod-less-divisor*
lemma *mod-less-dividend*: $[\![\ 0 < m;\ m \leq n\ ]\!] \Longrightarrow n\ mod\ m < (n::nat)$
thm *mod-less-divisor*[*of m n*]
thm *less-le-trans*[*OF mod-less-divisor*[*of m n*]]
by (*rule less-le-trans*[*OF mod-less-divisor*])
thm *Divides.mod-le-divisor*

lemmas *mod-le-dividend* = *mod-less-eq-dividend*

lemma *diff-mod-le*: $(t - r)\ mod\ m \leq (t::nat)$
thm *le-trans*[*OF mod-le-dividend*, *OF diff-le-self*]
by (*rule le-trans*[*OF mod-le-dividend*, *OF diff-le-self*])

thm *Divides.mult-div-cancel*

lemmas *div-mult-cancel* = *div-mod-equality'*

lemma *mod-0-div-mult-cancel*: $(n\ mod\ (m::nat) = 0) = (n\ div\ m * m = n)$
apply (*insert eq-diff-left-iff*[*OF mod-le-dividend le0*, *of n m*])
apply (*simp add*: *mult-commute mult-div-cancel*)
done

thm *Divides.mult-div-cancel*
lemma *div-mult-le*: $(n::nat)\ div\ m * m \leq n$
by (*simp add*: *mult-commute mult-div-cancel*)
lemma *less-div-Suc-mult*: $0 < (m::nat) \Longrightarrow n < Suc\ (n\ div\ m) * m$
apply (*simp add*: *mult-commute mult-div-cancel*)
apply (*rule less-add-diff*)
by (*rule mod-less-divisor*)

lemma *nat-ge2-conv*: $((2::nat) \leq n) = (n \neq 0 \wedge n \neq 1)$
by *fastsimp*

lemma *Suc0-mod*: $m \neq Suc\ 0 \Longrightarrow Suc\ 0\ mod\ m = Suc\ 0$
by (*case-tac m*, *simp-all*)
corollary *Suc0-mod-subst*:
$[\![\ m \neq Suc\ 0;\ P\ (Suc\ 0)\ ]\!] \Longrightarrow P\ (Suc\ 0\ mod\ m)$
thm *subst*[*OF Suc0-mod*[*symmetric*]]
by (*blast intro*: *subst*[*OF Suc0-mod*[*symmetric*]])
corollary *Suc0-mod-cong*:
$m \neq Suc\ 0 \Longrightarrow f\ (Suc\ 0\ mod\ m) = f\ (Suc\ 0)$
thm *arg-cong*[*OF Suc0-mod*]
by (*blast intro*: *arg-cong*[*OF Suc0-mod*])

5.2 Additional results for addition and subtraction with *mod*

thm *Divides.mod-Suc*
lemma *mod-Suc-conv*:
$((Suc\ a)\ mod\ m = (Suc\ b)\ mod\ m) = (a\ mod\ m = b\ mod\ m)$
by (*simp add*: *mod-Suc*)
thm *mod-Suc-conv*

thm *mod-Suc*
lemma *mod-Suc'*:
$0 < n \Longrightarrow Suc\ m\ mod\ n = (if\ m\ mod\ n < n - Suc\ 0\ then\ Suc\ (m\ mod\ n)\ else\ 0)$
apply (*simp add*: *mod-Suc*)
apply (*intro conjI impI*)
apply *simp*
apply (*insert le-neq-trans*[*OF mod-less-divisor*[*THEN Suc-leI*, *of n m*]], *simp*)
done

lemma *mod-add*:
$((a + k)\ mod\ m = (b + k)\ mod\ m) =$
$((a::nat)\ mod\ m = b\ mod\ m)$
by (*induct k*, *simp-all add*: *mod-Suc-conv*)

corollary *mod-sub-add*:
$k \leq (a::nat) \Longrightarrow$
$((a - k)\ mod\ m = b\ mod\ m) = (a\ mod\ m = (b + k)\ mod\ m)$
thm *mod-add*[**where** $m{=}m$ **and** $a{=}a{-}k$ **and** $b{=}b$ **and** $k{=}k$, *symmetric*]

by (*simp add*: *mod-add*[**where** *m*=*m* **and** *a*=*a*−*k* **and** *b*=*b* **and** *k*=*k*, *symmetric*])

thm
mod-Suc-conv
mod-add
mod-sub-add

lemma *mod-sub-eq-mod-0-conv*:
$a + b \leq (n::nat) \Longrightarrow$
$((n - a)\ mod\ m = b\ mod\ m) = ((n - (a + b))\ mod\ m = 0)$
thm *mod-add*[*of* *n*−(*a*+*b*) *b m 0*]
by (*insert mod-add*[*of* *n*−(*a*+*b*) *b m 0*], *simp*)
lemma *mod-sub-eq-mod-swap*:
$[\![\ a \leq (n::nat);\ b \leq n\ ]\!] \Longrightarrow$
$((n - a)\ mod\ m = b\ mod\ m) = ((n - b)\ mod\ m = a\ mod\ m)$
thm *mod-sub-add*
by (*simp add*: *mod-sub-add add-commute*)

lemma *le-mod-greater-imp-div-less*:
$[\![\ a \leq (b::nat);\ a\ mod\ m > b\ mod\ m\ ]\!] \Longrightarrow a\ div\ m < b\ div\ m$
apply (*rule ccontr*, *simp add*: *linorder-not-less*)
thm *mult-le-mono1*[*of* *b div m a div m m*]
apply (*drule mult-le-mono1*[*of* *b div m* - *m*])
thm *add-less-le-mono*[*of* *b mod m a mod m b div m* ∗ *m a div m* ∗ *m*]
apply (*drule add-less-le-mono*[*of* *b mod m a mod m b div m* ∗ *m a div m* ∗ *m*])
apply *simp-all*
done

lemma *less-mod-ge-imp-div-less*: $[\![\ a < (b::nat);\ a\ mod\ m \geq b\ mod\ m\ ]\!] \Longrightarrow a\ div\ m < b\ div\ m$
apply (*case-tac* *m* = *0*, *simp*)
thm *mult-less-cancel2*[*THEN iffD1*, *THEN conjunct2*]
apply (*rule mult-less-cancel1*[*of* *m*, *THEN iffD1*, *THEN conjunct2*])
apply (*simp add*: *mult-div-cancel*)
apply (*rule order-less-le-trans*[*of* - *b* − *a mod m*])
apply (*rule diff-less-mono*)
apply *simp*+
done
corollary *less-mod-0-imp-div-less*: $[\![\ a < (b::nat);\ b\ mod\ m = 0\ ]\!] \Longrightarrow a\ div\ m < b\ div\ m$
by (*simp add*: *less-mod-ge-imp-div-less*)

lemma *mod-diff-right-eq*:
$(a::nat) \leq b \Longrightarrow (b - a)\ mod\ m = (b - a\ mod\ m)\ mod\ m$
proof −
assume *a-as*:$a \leq b$
have $(b - a)\ mod\ m = (b - a + a\ div\ m * m)\ mod\ m$ **by** *simp*

also have ... = $(b + a\ div\ m * m - a)\ mod\ m$ **using** *a-as* **by** *simp*
also have ... = $(b + a\ div\ m * m - (a\ div\ m * m + a\ mod\ m))\ mod\ m$ **by** *simp*
also have ... = $(b + a\ div\ m * m - a\ div\ m * m - a\ mod\ m)\ mod\ m$
thm *diff-diff-left*[*symmetric*]
by (*simp only*: *diff-diff-left*[*symmetric*])
also have ... = $(b - a\ mod\ m)\ mod\ m$ **by** *simp*
finally show *?thesis* **.**
qed
corollary *mod-eq-imp-diff-mod-eq*:
$[\![\ x\ mod\ m = y\ mod\ m;\ x \leq (t::nat);\ y \leq t\ ]\!] \Longrightarrow$
$(t - x)\ mod\ m = (t - y)\ mod\ m$
thm *mod-diff-right-eq*
by (*simp only*: *mod-diff-right-eq*)
lemma *mod-eq-imp-diff-mod-eq2*:
$[\![\ x\ mod\ m = y\ mod\ m;\ (t::nat) \leq x;\ t \leq y\ ]\!] \Longrightarrow$
$(x - t)\ mod\ m = (y - t)\ mod\ m$
apply (*case-tac* $m = 0$, *simp+*)
thm *mod-mult-self2*[*of* $x - t\ m\ t$]
apply (*subst mod-mult-self2*[*of* $x - t\ m\ t$, *symmetric*])
apply (*subst mod-mult-self2*[*of* $y - t\ m\ t$, *symmetric*])
apply (*simp only*: *add-diff-assoc2 diff-add-assoc gr0-imp-self-le-mult2*)
apply (*simp only*: *mod-add*)
done
thm
mod-diff-right-eq
mod-eq-imp-diff-mod-eq
mod-eq-imp-diff-mod-eq2

lemma *divisor-add-diff-mod-if*:
$(m + b\ mod\ m - a\ mod\ m)\ mod\ (m::nat) =$ (
if $a\ mod\ m \leq b\ mod\ m$
then $(b\ mod\ m - a\ mod\ m)$
else $(m + b\ mod\ m - a\ mod\ m)$)
apply (*case-tac* $m = 0$, *simp*)
apply *clarsimp*
apply (*subst diff-add-assoc*, *assumption*)
apply (*simp only*: *mod-add-self1*)
apply (*rule mod-less*)
thm *less-imp-diff-less*
apply (*simp add*: *less-imp-diff-less*)
done
corollary *divisor-add-diff-mod-eq1*:
$a\ mod\ m \leq b\ mod\ m \Longrightarrow$
$(m + b\ mod\ m - a\ mod\ m)\ mod\ (m::nat) = b\ mod\ m - a\ mod\ m$
by (*simp add*: *divisor-add-diff-mod-if*)
corollary *divisor-add-diff-mod-eq2*:
$b\ mod\ m < a\ mod\ m \Longrightarrow$
$(m + b\ mod\ m - a\ mod\ m)\ mod\ (m::nat) = m + b\ mod\ m - a\ mod\ m$

by (*simp add*: *divisor-add-diff-mod-if*)

lemma *mod-add-mod-if*:
(*a mod m + b mod m*) *mod* (*m*::*nat*)= (
if a mod m + b mod m < m
then a mod m + b mod m
else a mod m + b mod m − m)
apply (*case-tac m = 0*, *simp-all*)
apply (*clarsimp simp*: *linorder-not-less*)
apply (*simp add*: *mod-if*[*of a mod m + b mod m*])
apply (*rule mod-less*)
thm *diff-less-conv*[*THEN iffD2*]
apply (*rule diff-less-conv*[*THEN iffD2*], *assumption*)
apply (*simp add*: *add-less-mono*)
done
corollary *mod-add-mod-eq1*:
a mod m + b mod m < m ⟹
(*a mod m + b mod m*) *mod* (*m*::*nat*) = *a mod m + b mod m*
by (*simp add*: *mod-add-mod-if*)
corollary *mod-add-mod-eq2*:
m ≤ *a mod m + b mod m*⟹
(*a mod m + b mod m*) *mod* (*m*::*nat*) = *a mod m + b mod m − m*
by (*simp add*: *mod-add-mod-if*)

thm *Divides.mod-add-eq*
lemma *mod-add1-eq-if*:
(*a + b*) *mod* (*m*::*nat*) = (
if (*a mod m + b mod m < m*) *then a mod m + b mod m*
else a mod m + b mod m − m)
by (*simp add*: *mod-add-eq*[*of a b*] *mod-add-mod-if*)

lemma *mod-add-eq-mod-conv*: *0* < (*m*::*nat*) ⟹
((*x + a*) *mod m* = *b mod m*) =
(*x mod m* = (*m + b mod m − a mod m*) *mod m*)
apply (*simp only*: *mod-add-eq*[*of x a*])
apply (*rule iffI*)
apply (*drule sym*)
thm *mod-add-mod-if*[*of x m a*]
apply (*simp add*: *mod-add-mod-if*)
thm *mod-add-left-eq*[*symmetric*]
thm *le-add-diff-inverse2*[*OF trans-le-add1*[*OF mod-le-divisor*], *of m*]
apply (*simp add*: *mod-add-left-eq*[*symmetric*] *le-add-diff-inverse2*[*OF trans-le-add1*[*OF mod-le-divisor*]])
done

lemma *mod-diff1-eq*:

(*a*::*nat*) ≤ *b* ⟹ (*b* − *a*) *mod m* = (*m* + *b mod m* − *a mod m*) *mod m*
apply (*case-tac m* = *0*, *simp*)
apply *simp*
proof −
assume *a-as*:*a* ≤ *b*
and *m-as*: *0* < *m*
have *a-mod-le-b-s*: *a mod m* ≤ *b*
by (*rule le-trans*[*of* - *a*], *simp only*: *mod-le-dividend*, *simp only*: *a-as*)
have (*b* − *a*) *mod m* = (*b* − *a mod m*) *mod m*
using *a-as* **by** (*simp only*: *mod-diff-right-eq*)
also have ... = (*b* − *a mod m* + *m*) *mod m*
by *simp*
thm *diff-add-assoc2*[*of a mod m b m*, *symmetric*]
also have ... = (*b* + *m* − *a mod m*) *mod m*
using *a-mod-le-b-s* **by** *simp*
thm *mod-div-equality*
also have ... = (*b div m* ∗ *m* + *b mod m* + *m* − *a mod m*) *mod m*
by *simp*
thm *diff-add-assoc*[*of a mod m b mod m* + *m*]
also have ... = (*b div m* ∗ *m* + (*b mod m* + *m* − *a mod m*)) *mod m*
thm *diff-add-assoc*[*OF mod-le-divisor*, *OF m-as*]
by (*simp add*: *diff-add-assoc*[*OF mod-le-divisor*, *OF m-as*])
also have ... = ((*b mod m* + *m* − *a mod m*) + *b div m* ∗ *m*) *mod m*
by *simp*
also have ... = (*b mod m* + *m* − *a mod m*) *mod m*
by *simp*
also have ... = (*m* + *b mod m* − *a mod m*) *mod m*
by (*simp only*: *add-commute*)
finally show *?thesis* **.**
qed
thm *divisor-add-diff-mod-if*
corollary *mod-diff1-eq-if*:
(*a*::*nat*) ≤ *b* ⟹ (*b* − *a*) *mod m* = (
if a mod m ≤ *b mod m then b mod m* − *a mod m*
else m + *b mod m* − *a mod m*)
by (*simp only*: *mod-diff1-eq divisor-add-diff-mod-if*)
corollary *mod-diff1-eq1*:
⟦ (*a*::*nat*) ≤ *b*; *a mod m* ≤ *b mod m* ⟧
⟹ (*b* − *a*) *mod m* = *b mod m* − *a mod m*
by (*simp add*: *mod-diff1-eq-if*)
corollary *mod-diff1-eq2*:
⟦ (*a*::*nat*) ≤ *b*; *b mod m* < *a mod m*⟧
⟹ (*b* − *a*) *mod m* = *m* + *b mod m* − *a mod m*
by (*simp add*: *mod-diff1-eq-if*)

5.2.1 Divisor subtraction with *div* and *mod*

thm
Divides.mod-add-self1

Divides.mod-add-self2
Divides.mod-mult-self1
Divides.mod-mult-self2
thm *mod-diff1-eq2*
lemma *mod-diff-self1*:
$0 < (n::nat) \Longrightarrow (m - n)\ mod\ m = m - n$
by (*case-tac* $m = 0$, *simp-all*)
lemma *mod-diff-self2*:
$m \leq (n::nat) \Longrightarrow (n - m)\ mod\ m = n\ mod\ m$
thm *mod-diff-right-eq*[*of m n m*]
by (*simp add*: *mod-diff-right-eq*)
lemma *mod-diff-mult-self1*:
$k * m \leq (n::nat) \Longrightarrow (n - k * m)\ mod\ m = n\ mod\ m$
thm *mod-diff-right-eq*[*of m n m*]
by (*simp add*: *mod-diff-right-eq*)
lemma *mod-diff-mult-self2*:
$m * k \leq (n::nat) \Longrightarrow (n - m * k)\ mod\ m = n\ mod\ m$
by (*simp only*: *mult-commute*[*of m k*] *mod-diff-mult-self1*)

thm
Divides.div-add-self1
Divides.div-add-self2
Divides.div-mult-self1
Divides.div-mult-self2
thm *div-0*
lemma *div-diff-self1*: $0 < (n::nat) \Longrightarrow (m - n)\ div\ m = 0$
by (*case-tac* $m = 0$, *simp-all*)
lemma *div-diff-self2*: $(n - m)\ div\ m = n\ div\ m - Suc\ 0$
apply (*case-tac* $m = 0$, *simp*)
apply (*case-tac* $n < m$, *simp*)
apply (*case-tac* $n = m$, *simp*)
thm *div-if*
apply (*simp add*: *div-if*)
done

lemma *div-diff-mult-self1*:
$(n - k * m)\ div\ m = n\ div\ m - (k::nat)$
apply (*case-tac* $m = 0$, *simp*)
apply (*case-tac* $n < k * m$)
apply *simp*
thm *div-le-mono*[*OF less-imp-le*]
apply (*drule div-le-mono*[*OF less-imp-le*, *of n - m*])
apply *simp*
apply (*simp add*: *linorder-not-less*)
thm *iffD1*[*OF mult-cancel1-gr0*[**where** k=m]]
apply (*rule iffD1*[*OF mult-cancel1-gr0*[**where** k=m]], *assumption*)
thm *diff-mult-distrib2*
apply (*subst diff-mult-distrib2*)
thm *mult-div-cancel*[*of m n - k * m*]

apply (*simp only*: *mult-div-cancel*)
apply (*simp only*: *diff-commute*[*of* - *k*∗*m*])
apply (*simp only*: *mult-commute*[*of m*])
thm *mod-diff-mult-self1*
apply (*simp only*: *mod-diff-mult-self1*)
done
lemma *div-diff-mult-self2*:
(*n* − *m* ∗ *k*) *div m* = *n div m* − (*k*::*nat*)
by (*simp only*: *mult-commute div-diff-mult-self1*)

5.2.2 Modulo equality and modulo of difference

lemma *mod-eq-imp-diff-mod-0*:
(*a*::*nat*) *mod m* = *b mod m* ⟹ (*b* − *a*) *mod m* = *0*
(**is** *?P* ⟹ *?Q*)
proof −
assume *as1*: *?P*
thm *mod-div-equality*
have *b* − *a* = *b div m* ∗ *m* + *b mod m* − (*a div m* ∗ *m* + *a mod m*)
by *simp*
also have ... = *b div m* ∗ *m* + *b mod m* − (*a mod m* + *a div m* ∗ *m*)
by *simp*
also have ... = *b div m* ∗ *m* + *b mod m* − *a mod m* − *a div m* ∗ *m*
by *simp*
also have ... = *b div m* ∗ *m* + *b mod m* − *b mod m* − *a div m* ∗ *m*
using *as1* **by** *simp*
also have ... = *b div m* ∗ *m* − *a div m* ∗ *m*
by (*simp only*: *diff-add-inverse2*)
also have ... = (*b div m* − *a div m*) ∗ *m*
by (*simp only*: *diff-mult-distrib*)
finally have *b* − *a* = (*b div m* − *a div m*) ∗ *m* **.**
hence (*b* − *a*) *mod m* = (*b div m* − *a div m*) ∗ *m mod m*
by (*rule arg-cong*)
thus *?thesis* **by** (*simp only*: *mod-mult-self2-is-0*)
qed
corollary *mod-eq-imp-diff-dvd*:
(*a*::*nat*) *mod m* = *b mod m* ⟹ *m dvd b* − *a*
by (*rule dvd-eq-mod-eq-0*[*THEN iffD2*, *OF mod-eq-imp-diff-mod-0*])

thm *mod-diff1-eq*
lemma *mod-neq-imp-diff-mod-neq0*:
⟦ (*a*::*nat*) *mod m* ≠ *b mod m*; *a* ≤ *b* ⟧ ⟹ *0* < (*b* − *a*) *mod m*
apply (*case-tac m* = *0*, *simp*)
apply (*drule le-imp-less-or-eq*, *erule disjE*)
prefer *2*
apply *simp*
apply (*drule neq-iff*[*THEN iffD1*], *erule disjE*)
thm *mod-diff1-eq1*
apply (*simp add*: *mod-diff1-eq1*)

thm *mod-diff1-eq2*[*OF less-imp-le*]
thm *trans-less-add1*[*OF mod-less-divisor*]
apply (*simp add*: *mod-diff1-eq2*[*OF less-imp-le*] *trans-less-add1*[*OF mod-less-divisor*])
done
corollary *mod-neq-imp-diff-not-dvd*:
⟦ (*a*::*nat*) *mod m* ≠ *b mod m*; *a* ≤ *b* ⟧ ⟹ ¬ *m dvd b* − *a*
thm *dvd-eq-mod-eq-0*
by (*simp add*: *dvd-eq-mod-eq-0 mod-neq-imp-diff-mod-neq0*)

lemma *diff-mod-0-imp-mod-eq*:
⟦ (*b* − *a*) *mod m* = *0*; *a* ≤ *b* ⟧ ⟹ (*a*::*nat*) *mod m* = *b mod m*
apply (*rule ccontr*)
thm *mod-neq-imp-diff-mod-neq0*
apply (*drule mod-neq-imp-diff-mod-neq0*)
apply *simp-all*
done
corollary *diff-dvd-imp-mod-eq*:
⟦ *m dvd b* − *a*; *a* ≤ *b* ⟧ ⟹ (*a*::*nat*) *mod m* = *b mod m*
thm *dvd-eq-mod-eq-0*[*THEN iffD1*, *THEN diff-mod-0-imp-mod-eq*]
by (*rule dvd-eq-mod-eq-0*[*THEN iffD1*, *THEN diff-mod-0-imp-mod-eq*])

thm
mod-eq-imp-diff-mod-0
diff-mod-0-imp-mod-eq
lemma *mod-eq-diff-mod-0-conv*:
a ≤ (*b*::*nat*) ⟹ (*a mod m* = *b mod m*) = ((*b* − *a*) *mod m* = *0*)
apply (*rule iffI*)
apply (*rule mod-eq-imp-diff-mod-0*, *assumption*)
apply (*rule diff-mod-0-imp-mod-eq*, *assumption*+)
done
corollary *mod-eq-diff-dvd-conv*:
a ≤ (*b*::*nat*) ⟹ (*a mod m* = *b mod m*) = (*m dvd b* − *a*)
thm *dvd-eq-mod-eq-0*[*symmetric*, *THEN subst*]
by (*rule dvd-eq-mod-eq-0*[*symmetric*, *THEN subst*], *rule mod-eq-diff-mod-0-conv*)

5.3 Some additional lemmata about integer *div* and *mod*

lemma *zmod-eq-imp-diff-mod-0*:
(*a*::*int*) *mod m* = *b mod m* ⟹ (*b* − *a*) *mod m* = *0*
apply (*simp only*: *diff-minus*)
thm *mod-add-eq*[*of b* −*a m*]
apply (*simp only*: *mod-add-eq*[*of b* −*a m*])
thm *zmod-zminus1-eq-if*
apply (*simp add*: *zmod-zminus1-eq-if*)
done

thm

zmod-eq-imp-diff-mod-0[*of* a::*int* m b]
mod-eq-imp-diff-mod-0[*of* a::*nat* m b]

thm *Divides.nat-mod-distrib*
lemma ⟦ 0 ≤ (n::*int*); 0 ≤ m ⟧ ⟹ *nat* (n *mod* m) = *nat* n *mod nat* m
by (*blast intro*: *nat-mod-distrib*)

lemmas *int-mod-distrib* = *zmod-int*

lemma *zdiff-mod-0-imp-mod-eq--pos*:
⟦ (b − a) *mod* m = 0; 0 < (m::*int*) ⟧ ⟹ a *mod* m = b *mod* m
(**is** ⟦ *?P*; *?Pm* ⟧ ⟹ *?Q*)
proof −
assume *as1*: *?P*
and *as2*: 0 < m

obtain *r1* **where** *a-r1*:*r1* = a *mod* m **by** *blast*
obtain *r2* **where** *b-r2*:*r2* = b *mod* m **by** *blast*

obtain *q1* **where** *a-q1*: *q1* = a *div* m **by** *blast*
obtain *q2* **where** *b-q2*: *q2* = b *div* m **by** *blast*

have *a-r1-q1*: a = m ∗ *q1* + *r1*
using *a-r1 a-q1* **by** *simp*
have *b-r2-q2*: b = m ∗ *q2* + *r2*
using *b-r2 b-q2* **by** *simp*

thm *divmod-int-rel-div-mod*[*of* m a]
thm *Divides.divmod-int-rel-def*

have b − a = m ∗ *q2* + *r2* − (m ∗ *q1* + *r1*)
using *a-r1-q1 b-r2-q2* **by** *simp*
also have ... = m ∗ *q2* + *r2* − m ∗ *q1* − *r1*
by *simp*
also have ... = m ∗ *q2* − m ∗ *q1* + *r2* − *r1*
by *simp*
finally have b − a = m ∗ (*q2* − *q1*) + (*r2* − *r1*)
by (*simp add*: *zdiff-zmult-distrib2*)
hence (b − a) *mod* m = (*r2* − *r1*) *mod* m
by (*simp add*: *mod-add-eq*)
hence *r2-r1-mod-m-0*:(*r2* − *r1*) *mod* m = 0 (**is** *?R1*)
by (*simp only*: *as1*)

have *r1* = *r2*
thm *notI*[*of r1* ≠ *r2*, *simplified*]
proof (*rule notI*[*of r1* ≠ *r2*, *simplified*])
assume *as1'*: *r1* ≠ *r2*
have *diff-le-s*: ⋀a b (m::*int*). ⟦ 0 ≤ a; b < m ⟧ ⟹ b − a < m

```
    by simp
  have s-r1:0 ≤ r1 ∧ r1 < m and s-r2:0 ≤ r2 ∧ r2 < m
    thm pos-mod-conj
    by (simp add: as2 a-r1 b-r2 pos-mod-conj)+
  have mr2r1:−m < r2 − r1 and r2r1m:r2 − r1 < m
    thm minus-less-iff[of m]
    thm diff-le-s
    by (simp add: minus-less-iff[of m] s-r1 s-r2 diff-le-s)+
  have 0 ≤ r2 − r1 ⟹ (r2 − r1) mod m = (r2 − r1)
    thm mod-pos-pos-trivial[of r2−r1 m]
    using r2r1m by (blast intro: mod-pos-pos-trivial)
  hence s1-pos: 0 ≤ r2 − r1 ⟹ r2 − r1 = 0
    using r2-r1-mod-m-0 by simp

  have (r2−r1) mod −m = 0
    thm zmod-zminus2-eq-if[of r2−r1 m, simplified]
    by (simp add: zmod-zminus2-eq-if[of r2−r1 m, simplified] r2-r1-mod-m-0)
  moreover
  have r2 − r1 ≤ 0 ⟹ (r2 − r1) mod −m = r2 − r1
    using mr2r1
    thm mod-neg-neg-trivial[of r2−r1 −m]
    by (simp add: mod-neg-neg-trivial)
  ultimately have s1-neg:r2 − r1 ≤ 0 ⟹ r2 − r1 = 0
    by simp

  have r2 − r1 = 0
    using s1-pos s1-neg zle-linear by blast
  hence r1 = r2 by simp
  thus False
    using as1' by blast
 qed
 thus ?thesis
  using a-r1 b-r2 by blast
qed

lemma zmod-zminus-eq-conv-pos:
  0 < (m::int) ⟹ (a mod − m = b mod − m) = (a mod m = b mod m)
apply (simp only: zmod-zminus2 neg-equal-iff-equal)
thm zmod-zminus1-eq-if[of - m]
apply (simp only: zmod-zminus1-eq-if)
apply (split split-if)+
apply (safe, simp-all)
thm pos-mod-bound[of m]
apply (insert pos-mod-bound[of m a] pos-mod-bound[of m b], simp-all)
done
lemma zmod-zminus-eq-conv:
  ((a::int) mod − m = b mod − m) = (a mod m = b mod m)
apply (insert zless-linear[of 0 m], elim disjE)
apply (blast dest: zmod-zminus-eq-conv-pos)
```

apply *simp*
thm *zmod-zminus-eq-conv-pos*[*of* $-m$, *symmetric*]
apply (*simp add*: *zmod-zminus-eq-conv-pos*[*of* $-m$, *symmetric*])
done

lemma *zdiff-mod-0-imp-mod-eq*:
$(b - a) \bmod m = 0 \Longrightarrow (a::int) \bmod m = b \bmod m$
by (*metis dvd-eq-mod-eq-0 zmod-eq-dvd-iff*)
thm *zdiff-mod-0-imp-mod-eq*

thm
zmod-eq-imp-diff-mod-0
zdiff-mod-0-imp-mod-eq
lemma *zmod-eq-diff-mod-0-conv*:
$((a::int) \bmod m = b \bmod m) = ((b - a) \bmod m = 0)$
apply (*rule iffI*)
apply (*rule zmod-eq-imp-diff-mod-0*, *assumption*)
apply (*rule zdiff-mod-0-imp-mod-eq*, *assumption*)
done

lemma $\neg(\exists (a::int)\ b\ m.\ (b - a) \bmod m = 0 \wedge a \bmod m \neq b \bmod m)$
by (*simp add*: *zmod-eq-diff-mod-0-conv*)
lemma $\exists (a::nat)\ b\ m.\ (b - a) \bmod m = 0 \wedge a \bmod m \neq b \bmod m$
apply (*rule-tac x=1* **in** *exI*)
apply (*rule-tac x=0* **in** *exI*)
apply (*rule-tac x=2* **in** *exI*)
apply *simp*
done

thm
zdiff-mod-0-imp-mod-eq[*of* $b::int$ a m]
diff-mod-0-imp-mod-eq[*of* $b::nat$ a m]

lemma *zmult-div-leq-mono*:
$[\![(0::int) \leq x;\ a \leq b;\ 0 < d]\!] \Longrightarrow x * a \ div\ d \leq x * b \ div\ d$
by (*metis mult-right-mono zdiv-mono1 zmult-commute*)

lemma *zmult-div-leq-mono-neg*:
$[\![x \leq (0::int);\ a \leq b;\ 0 < d]\!] \Longrightarrow x * b \ div\ d \leq x * a \ div\ d$
by (*metis mult-left-mono-neg zdiv-mono1*)

lemma *zmult-div-pos-le*:
$[\![(0::int) \leq a;\ 0 \leq b;\ b \leq c]\!] \Longrightarrow a * b \ div\ c \leq a$
apply (*case-tac* $b = 0$, *simp*)
apply (*subgoal-tac* $b * a \leq c * a$)
prefer *2*

apply (*simp only*: *mult-right-mono*)
apply (*simp only*: *zmult-commute*)
apply (*subgoal-tac* $a * b \ div \ c \leq a * c \ div \ c$)
prefer *2*
thm *zdiv-mono1*
apply (*simp only*: *zdiv-mono1*)
apply *simp*
done

lemma *zmult-div-neg-le*:
$[\![\ a \leq (0::int); \ 0 < c; \ c \leq b \ ]\!] \Longrightarrow a * b \ div \ c \leq a$
apply (*subgoal-tac* $b * a \leq c * a$)
prefer *2*
apply (*simp only*: *mult-right-mono-neg*)
apply (*simp only*:*zmult-commute*)
apply (*subgoal-tac* $a * b \ div \ c \leq a * c \ div \ c$)
prefer *2*
thm *zdiv-mono1*
apply (*simp only*: *zdiv-mono1*)
apply *simp*
done

lemma *zmult-div-ge-0*:$[\![\ (0::int) \leq x; \ 0 \leq a; \ 0 < c \ ]\!] \Longrightarrow 0 \leq a * x \ div \ c$
by (*metis pos-imp-zdiv-nonneg-iff split-mult-pos-le*)

corollary *zmult-div-plus-ge-0*:
$[\![\ (0::int) \leq x; \ 0 \leq a; \ 0 \leq b; \ 0 < c]\!] \Longrightarrow 0 \leq a * x \ div \ c + b$
by (*insert zmult-div-ge-0*[*of x a c*], *simp*)

lemma *zmult-div-abs-ge*:
$[\![\ (0::int) \leq b; \ b \leq b'; \ 0 \leq a; \ 0 < c]\!] \Longrightarrow$
$|a * b \ div \ c| \leq |a * b' \ div \ c|$
thm *zmult-div-ge-0*[*of b a c*]
apply (*insert zmult-div-ge-0*[*of b a c*] *zmult-div-ge-0*[*of b' a c*], *simp*)
by (*metis zmult-div-leq-mono*)

lemma *zmult-div-plus-abs-ge*:
$[\![\ (0::int) \leq b; \ b \leq b'; \ 0 \leq a; \ 0 < c \ ]\!] \Longrightarrow$
$|a * b \ div \ c + a| \leq |a * b' \ div \ c + a|$
thm *zmult-div-plus-ge-0*
apply (*insert zmult-div-plus-ge-0*[*of b a a c*] *zmult-div-plus-ge-0*[*of b' a a c*], *simp*)
by (*metis zmult-div-leq-mono*)
thm *zmult-div-plus-abs-ge*

5.4 Some further (in-)equality results for *div* and *mod*

lemma *less-mod-eq-imp-add-divisor-le*:
⟦ (*x*::*nat*) < *y*; *x mod m* = *y mod m* ⟧ ⟹ *x* + *m* ≤ *y*
apply (*case-tac m* = *0*)
apply *simp*
thm *contrapos-pp*[*of x mod m* = *y mod m*]
apply (*rule contrapos-pp*[*of x mod m* = *y mod m*])
apply *blast*
apply (*rule ccontr*, *simp only*: *not-not*, *clarify*)
proof −
assume *m-greater-0*: *0* < *m*
assume *x-less-y*:*x* < *y*
hence *y-x-greater-0*:*0* < *y* − *x*
by *simp*
assume *x mod m* = *y mod m*
hence *y-x-mod-m*: (*y* − *x*) *mod m* = *0*
thm *mod-eq-imp-diff-mod-0*
by (*simp only*: *mod-eq-imp-diff-mod-0*)
assume ¬ *x* + *m* ≤ *y*
hence *y* < *x* + *m* **by** *simp*
hence *y* − *x* < *x* + *m* − *x*
by (*simp add*: *diff-add-inverse diff-less-conv m-greater-0*)
hence *y-x-less-m*: *y* − *x* < *m*
by *simp*
thm *y-x-greater-0 y-x-less-m*
thm *mod-less*[*of y* − *x m*]
have (*y* − *x*) *mod m* = *y* − *x*
using *y-x-less-m* **by** *simp*
hence *y* − *x* = *0*
using *y-x-mod-m* **by** *simp*
thus *False*
using *y-x-greater-0* **by** *simp*
qed

lemma *less-div-imp-mult-add-divisor-le*:
(*x*::*nat*) < *n div m* ⟹ *x* ∗ *m* + *m* ≤ *n*
apply (*case-tac m* = *0*, *simp*)
apply (*case-tac n* < *m*, *simp*)
apply (*simp add*: *linorder-not-less*)
apply (*subgoal-tac m* ≤ *n* − *n mod m*)
prefer *2*
apply (*drule div-le-mono*[*of m - m*])
apply (*simp only*: *div-self*)
apply (*drule mult-le-mono2*[*of 1 - m*])
apply (*simp only*: *mult-1-right mult-div-cancel*)
apply (*drule less-imp-le-pred*[*of x*])
apply (*drule mult-le-mono2*[*of x - m*])
apply (*simp add*: *diff-mult-distrib2 mult-div-cancel del*: *diff-diff-left*)

thm *le-diff-conv2*[*of m*]
apply (*simp only*: *le-diff-conv2*[*of m*])
thm *le-diff-imp-le*[*of m * x + m*]
apply (*drule le-diff-imp-le*[*of m * x + m*])
apply (*simp only*: *mult-commute*[*of - m*])
done

lemma *mod-add-eq-imp-mod-0*:
  ((*n + k*) *mod* (*m*::*nat*) = *n mod m*) = (*k mod m = 0*)
by (*metis add-eq-if mod-add mod-add-self1 mod-self nat-add-commute*)

lemma *between-imp-mod-between*:
  ⟦ *b* < (*m*::*nat*); *m * k + a* ≤ *n*; *n* ≤ *m * k + b* ⟧ ⟹
  a ≤ *n mod m* ∧ *n mod m* ≤ *b*
apply (*case-tac m = 0*, *simp-all*)
apply (*frule gr-implies-gr0*)
apply (*subgoal-tac k = n div m*)
 prefer *2*
 apply (*rule split-div-lemma*[*THEN iffD1*], *assumption*)
 apply *simp*
apply *clarify*
apply (*rule conjI*)
apply (*rule add-le-imp-le-left*[**where** *c*=*m * (n div m)*], *simp*)+
done

corollary *between-imp-mod-le*:
  ⟦ *b* < (*m*::*nat*); *m * k* ≤ *n*; *n* ≤ *m * k + b* ⟧ ⟹ *n mod m* ≤ *b*
by (*insert between-imp-mod-between*[*of b m k 0 n*], *simp*)
corollary *between-imp-mod-gr0*:
  ⟦ (*m*::*nat*) * *k* < *n*; *n* < *m * k + m* ⟧ ⟹ *0* < *n mod m*
apply (*case-tac m = 0*, *simp-all*)
apply (*rule Suc-le-lessD*)
apply (*rule between-imp-mod-between*[*THEN conjunct1*, *of m − Suc 0 m k Suc 0 n*])
apply *simp-all*
done

Some variations of *split-div-lemma*

corollary *le-less-div-conv*:
  0 < *m* ⟹ (*k * m* ≤ *n* ∧ *n* < *Suc k * m*) = (*n div m = k*)
by (*metis div-mult-le nat-mult-commute split-div-lemma*)
lemma *le-less-imp-div*:
  ⟦ *k * m* ≤ *n*; *n* < *Suc k * m* ⟧ ⟹ *n div m = k*
by (*metis gr-implies-not0 mult-eq-if nat-mult-commute neq0-conv split-div-lemma*)
lemma *div-imp-le-less*:
  ⟦ *n div m = k*; *0* < *m* ⟧ ⟹ *k * m* ≤ *n* ∧ *n* < *Suc k * m*
thm *le-less-div-conv*[*THEN iffD2*]
by (*rule le-less-div-conv*[*THEN iffD2*])

lemma *div-le-mod-le-imp-le*:
⟦ (*a*::*nat*) *div m* ≤ *b div m*; *a mod m* ≤ *b mod m* ⟧ ⟹ *a* ≤ *b*
apply (*rule subst*[*OF mod-div-equality2*[*of m a*]])
apply (*rule subst*[*OF mod-div-equality2*[*of m b*]])
apply (*rule add-le-mono*)
apply (*rule mult-le-mono2*)
apply *assumption*+
done

lemma *le-mod-add-eq-imp-add-mod-le*:
⟦ *a* ≤ *b*; (*a* + *k*) *mod m* = (*b*::*nat*) *mod m* ⟧ ⟹ *a* + *k mod m* ≤ *b*
by (*metis add-le-mono2 diff-add-inverse le-add1 le-add-diff-inverse mod-diff1-eq mod-less-eq-dividend*)

corollary *mult-divisor-le-mod-ge-imp-ge*:
⟦ (*m*::*nat*) * *k* ≤ *n*; *r* ≤ *n mod m* ⟧ ⟹ *m* * *k* + *r* ≤ *n*
apply (*insert le-mod-add-eq-imp-add-mod-le*[*of m * k n n mod m m*])
apply (*simp add*: *add-commute*[*of m * k*])
done

5.5 Additional multiplication results for *mod* and *div*

lemma *mod-0-imp-mod-mult-right-0*:
n mod m = (*0*::*nat*) ⟹ *n* * *k mod m* = *0*
by *fastsimp*
lemma *mod-0-imp-mod-mult-left-0*:
n mod m = (*0*::*nat*) ⟹ *k* * *n mod m* = *0*
by *fastsimp*

lemma *mod-0-imp-div-mult-left-eq*:
n mod m = (*0*::*nat*) ⟹ *k* * *n div m* = *k* * (*n div m*)
by *fastsimp*
lemma *mod-0-imp-div-mult-right-eq*:
n mod m = (*0*::*nat*) ⟹ *n* * *k div m* = *k* * (*n div m*)
by *fastsimp*

thm *Rings.dvd-mult-left*
lemma *mod-0-imp-mod-factor-0-left*:
n mod (*m* * *m'*) = (*0*::*nat*) ⟹ *n mod m* = *0*
by *fastsimp*
lemma *mod-0-imp-mod-factor-0-right*:
n mod (*m* * *m'*) = (*0*::*nat*) ⟹ *n mod m'* = *0*
by *fastsimp*

5.6 Some factor distribution facts for *mod*

lemma *mod-eq-mult-distrib*:
(*a*::*nat*) *mod m* = *b mod m* ⟹
a ∗ *k mod* (*m* ∗ *k*) = *b* ∗ *k mod* (*m* ∗ *k*)
by *simp*

lemma *mod-mult-eq-imp-mod-eq*:
(*a*::*nat*) *mod* (*m* ∗ *k*) = *b mod* (*m* ∗ *k*) ⟹ *a mod m* = *b mod m*
thm *mod-mult2-eq*[*of a m k*]
apply (*simp only*: *mod-mult2-eq*)
thm *arg-cong*[**where** *f*=λ*x*. *x mod m*]
apply (*drule-tac arg-cong*[**where** *f*=λ*x*. *x mod m*])
apply (*simp add*: *add-commute*)
done
corollary *mod-eq-mod-0-imp-mod-eq*:
⟦ (*a*::*nat*) *mod m*′ = *b mod m*′; *m*′ *mod m* = *0* ⟧
⟹ *a mod m* = *b mod m*
by (*clarify*, *drule mod-mult-eq-imp-mod-eq*)

lemma *mod-factor-imp-mod-0*:
⟦(*x*::*nat*) *mod* (*m* ∗ *k*) = *y* ∗ *k mod* (*m* ∗ *k*)⟧ ⟹ *x mod k* = *0*
(**is** ⟦ *?P1* ⟧ ⟹ *?Q*)
proof −
assume *as1*: *?P1*
thm *mod-mult-distrib*[**where** *m*=*y* **and** *n*=*m* **and** *k*=*k*]
have *y* ∗ *k mod* (*m* ∗ *k*) = *y mod m* ∗ *k*
by *simp*
hence *x mod* (*m* ∗ *k*) = *y mod m* ∗ *k*
using *as1* **by** *simp*
hence *y mod m* ∗ *k* = *k* ∗ (*x div k mod m*) + *x mod k* (**is** *?l1* = *?r1*)
by (*simp only*: *mult-ac mod-mult2-eq*)
hence (*y mod m* ∗ *k*) *mod k* = *?r1 mod k*
by *simp*
hence *0* = *?r1 mod k*
by *simp*
thus *x mod k* = *0*
thm *mod-add-eq*
by (*simp add*: *mod-add-eq*)
qed
corollary *mod-factor-div*:
⟦(*x*::*nat*) *mod* (*m* ∗ *k*) = *y* ∗ *k mod* (*m* ∗ *k*)⟧ ⟹ *x div k* ∗ *k* = *x*
thm *mod-factor-imp-mod-0*[*THEN mod-0-div-mult-cancel*[*THEN iffD1*]]
by (*blast intro*: *mod-factor-imp-mod-0*[*THEN mod-0-div-mult-cancel*[*THEN iffD1*]])

lemma *mod-factor-div-mod*:
⟦ (*x*::*nat*) *mod* (*m* ∗ *k*) = *y* ∗ *k mod* (*m* ∗ *k*); *0* < *k* ⟧
⟹ *x div k mod m* = *y mod m*
(**is** ⟦ *?P1*; *?P2* ⟧ ⟹ *?L* = *?R*)
proof −

assume *as1*: *?P1*
assume *as2*: *?P2*
have *x-mod-k-0*: *x mod k = 0*
using *as1* **by** (*blast intro*: *mod-factor-imp-mod-0*)
thm *mod-mult2-eq*[**where** *a*=*x* **and** *b*=*k* **and** *c*=*m*]
have *?L * k + x mod k = x mod (k * m)*
by (*simp only*: *mod-mult2-eq mult-commute*[*of* - *k*])
hence *?L * k = x mod (k * m)*
using *x-mod-k-0* **by** *simp*
hence *?L * k = y * k mod (m * k)*
using *as1* **by** (*simp only*: *mult-ac*)
hence *?L * k = y mod m * k*
thm *mod-mult-distrib*
by (*simp only*: *mod-mult-distrib*)
thus *?thesis*
using *as2* **by** *simp*
qed

thm
Divides.mod-mult-distrib
Divides.mod-mult-distrib2
thm
mod-eq-mult-distrib
thm
mod-factor-imp-mod-0
mod-factor-div
mod-factor-div-mod

5.7 More results about quotient *div* with addition and subtraction

thm *Divides.div-add1-eq*
thm *mod-add1-eq-if*
lemma *div-add1-eq-if*: *0 < m ⟹*
(a + b) div (m::nat) = a div m + b div m + (
if a mod m + b mod m < m then 0 else Suc 0)
apply (*simp only*: *div-add1-eq*[*of a b*])
thm *arg-cong*[*of (a mod m + b mod m) div m*]
apply (*rule arg-cong*[*of (a mod m + b mod m) div m*])
apply (*clarsimp simp*: *linorder-not-less*)
thm *le-less-imp-div*[*of Suc 0 m a mod m + b mod m*]
apply (*rule le-less-imp-div*[*of Suc 0 m a mod m + b mod m*], *simp*)
apply *simp*
apply (*simp only*: *add-less-mono*[*OF mod-less-divisor mod-less-divisor*])
done
corollary *div-add1-eq1*:
a mod m + b mod m < (m::nat) ⟹
(a + b) div (m::nat) = a div m + b div m
apply (*case-tac m = 0*, *simp*)

apply (*simp add*: *div-add1-eq-if*)
done
corollary *div-add1-eq1-mod-0-left*:
a mod m = 0 ⟹ (a + b) div (m::nat) = a div m + b div m
apply (*case-tac m = 0*, *simp*)
apply (*simp add*: *div-add1-eq1*)
done
corollary *div-add1-eq1-mod-0-right*:
b mod m = 0 ⟹ (a + b) div (m::nat) = a div m + b div m
by (*fastsimp simp*: *div-add1-eq1-mod-0-left*)
corollary *div-add1-eq2*:
⟦ 0 < m; *(m::nat) ≤ a mod m + b mod m ⟧ ⟹*
(a + b) div (m::nat) = Suc (a div m + b div m)
by (*simp add*: *div-add1-eq-if*)

lemma *div-Suc*:
0 < n ⟹ Suc m div n = (if Suc (m mod n) = n then Suc (m div n) else m div n)
apply (*drule Suc-leI*, *drule le-imp-less-or-eq*)
apply (*case-tac n = Suc 0*, *simp*)
apply (*split split-if*, *intro conjI impI*)
apply (*rule-tac t=Suc m* **and** *s=m + 1* **in** *subst*, *simp*)
apply (*subst div-add1-eq2*, *simp+*)
thm *le-neq-trans*[*OF mod-less-divisor*[*THEN Suc-leI*]]
apply (*insert le-neq-trans*[*OF mod-less-divisor*[*THEN Suc-leI*, *of n m*]], *simp*)
apply (*rule-tac t=Suc m* **and** *s=m + 1* **in** *subst*, *simp*)
apply (*subst div-add1-eq1*, *simp+*)
done
lemma *div-Suc′*:
0 < n ⟹ Suc m div n = (if m mod n < n − Suc 0 then m div n else Suc (m div n))
apply (*simp add*: *div-Suc*)
apply (*intro conjI impI*)
apply *simp*
apply (*insert le-neq-trans*[*OF mod-less-divisor*[*THEN Suc-leI*, *of n m*]], *simp*)
done

lemma *div-diff1-eq-if*:
(b − a) div (m::nat) =
b div m − a div m − (if a mod m ≤ b mod m then 0 else Suc 0)
apply (*case-tac m = 0*, *simp*)
apply (*case-tac b < a*)
apply (*frule less-imp-le*[*of b*])
apply (*frule div-le-mono*[*of - - m*])
apply *simp*
apply (*simp only*: *linorder-not-less neq0-conv*)
proof −
assume *le-as*: *a ≤ b*
and *m-as*: *0 < m*

have *div-le*:$a\ div\ m \leq b\ div\ m$
using *le-as* **by** (*simp only*: *div-le-mono*)
thm *mod-div-equality*[*of b m*]
have $b - a = b\ div\ m * m + b\ mod\ m - (a\ div\ m * m + a\ mod\ m)$
by *simp*
also have $\ldots = b\ div\ m * m + b\ mod\ m - a\ div\ m * m - a\ mod\ m$
by *simp*
also have $\ldots = b\ div\ m * m - a\ div\ m * m + b\ mod\ m - a\ mod\ m$
thm *mult-le-mono1*[*OF div-le*]
thm *diff-add-assoc2*[*OF mult-le-mono1*[*OF div-le*]]
by (*simp only*: *diff-add-assoc2*[*OF mult-le-mono1*[*OF div-le*]])
finally have *b-a-s1*: $b - a = (b\ div\ m - a\ div\ m) * m + b\ mod\ m - a\ mod\ m$
(**is** *?b-a* = *?b-a1*)
by (*simp only*: *diff-mult-distrib*)
thm *b-a-s1*
hence *b-a-div-s*: $(b - a)\ div\ m =$
$((b\ div\ m - a\ div\ m) * m + b\ mod\ m - a\ mod\ m)\ div\ m$
by (*rule arg-cong*)

show *?thesis*
proof (*cases* $a\ mod\ m \leq b\ mod\ m$)
case *True*
hence *as'*: $a\ mod\ m \leq b\ mod\ m$ **.**

have $(b - a)\ div\ m =$ *?b-a1* $div\ m$
using *b-a-div-s* **.**
also have $\ldots = ((b\ div\ m - a\ div\ m) * m + (b\ mod\ m - a\ mod\ m))\ div\ m$
using *as'* **by** *simp*
thm *div-add1-eq*
also have $\ldots = b\ div\ m - a\ div\ m + (b\ mod\ m - a\ mod\ m)\ div\ m$
thm *div-mult-self1*
apply (*simp only*: *add-commute*)
thm *less-imp-neq*[*OF m-as*, *THEN not-sym*]
thm *div-mult-self1*[*OF less-imp-neq*[*OF m-as*, *THEN not-sym*]]
by (*simp only*: *div-mult-self1*[*OF less-imp-neq*[*OF m-as*, *THEN not-sym*]])
finally have *b-a-div-s'*: $(b - a)\ div\ m = \ldots$ **.**
have $(b\ mod\ m - a\ mod\ m)\ div\ m = 0$
by (*rule div-less*, *rule less-imp-diff-less*,
rule mod-less-divisor, *rule m-as*)
thus *?thesis*
using *b-a-div-s' as'*
by *simp*
next
case *False*
hence *as1'*: $\neg\ a\ mod\ m \leq b\ mod\ m$ **.**
hence *as'*: $b\ mod\ m < a\ mod\ m$ **by** *simp*

have *a-div-less*: $a\ div\ m < b\ div\ m$
using *le-as as'*

thm *le-mod-greater-imp-div-less*
by (*blast intro*: *le-mod-greater-imp-div-less*)

have *b div m − a div m = b div m − a div m − (Suc 0 − Suc 0)*
by *simp*
also have *... = b div m − a div m + Suc 0 − Suc 0*
by *simp*
also have *... = b div m − a div m − Suc 0 + Suc 0*
thm *a-div-less*
thm *a-div-less*[*THEN zero-less-diff*[*THEN iffD2*]]
thm *a-div-less*[*THEN zero-less-diff*[*THEN iffD2*],
THEN Suc-le-eq[*THEN iffD2*]]
thm *diff-add-assoc2*
by (*simp only*: *diff-add-assoc2*
a-div-less[*THEN zero-less-diff*[*THEN iffD2*], *THEN Suc-le-eq*[*THEN iffD2*]])
finally have *b-a-div-s'*: *b div m − a div m = ...* **.**

have *(b − a) div m = ?b-a1 div m*
using *b-a-div-s* **.**
also have *... = ((b div m − a div m − Suc 0 + Suc 0) * m*
+ b mod m − a mod m) div m
using *b-a-div-s'* **by** (*rule arg-cong*)
also have *... = ((b div m − a div m − Suc 0) * m*
*+ Suc 0 * m + b mod m − a mod m) div m*
by (*simp only*: *add-mult-distrib*)
also have *... = ((b div m − a div m − Suc 0) * m*
+ m + b mod m − a mod m) div m
by *simp*
also have *... = ((b div m − a div m − Suc 0) * m*
+ (m + b mod m − a mod m)) div m
thm *diff-add-assoc*[*of a mod m m + b mod m*]
thm *trans-le-add1*[*of a mod m m*, *OF mod-le-divisor*]
by (*simp only*: *add-assoc m-as*
diff-add-assoc[*of a mod m m + b mod m*]
trans-le-add1[*of a mod m m*, *OF mod-le-divisor*])
also have *... = b div m − a div m − Suc 0*
+ (m + b mod m − a mod m) div m
thm *div-mult-self1*
thm *div-mult-self1*[*OF less-imp-neq*[*OF m-as*, *THEN not-sym*]]
by (*simp only*: *add-commute div-mult-self1*[*OF less-imp-neq*[*OF m-as*, *THEN not-sym*]])
finally have *b-a-div-s'*: *(b − a) div m = ...* **.**

have *div-0-s*: *(m + b mod m − a mod m) div m = 0*
thm *add-diff-less*
by (*rule div-less*, *simp only*: *add-diff-less m-as as'*)
show *?thesis*
by (*simp add*: *as1' b-a-div-s' div-0-s*)
qed

qed

thm *div-diff1-eq-if*
corollary *div-diff1-eq*:
  $(b - a)\ div\ (m::nat) =$
  $b\ div\ m - a\ div\ m - (m + a\ mod\ m - Suc\ (b\ mod\ m))\ div\ m$
apply (*case-tac* $m = 0$, *simp*)
apply (*simp only*: *neq0-conv*)
thm *div-diff1-eq-if*
thm *subst*[*of*
  if $a\ mod\ m \leq b\ mod\ m$ *then* 0 *else Suc* 0
  $(m + a\ mod\ m - Suc(b\ mod\ m))\ div\ m$]
apply (*rule subst*[*of*
  if $a\ mod\ m \leq b\ mod\ m$ *then* 0 *else Suc* 0
  $(m + a\ mod\ m - Suc(b\ mod\ m))\ div\ m$])
prefer 2 **apply** (*rule div-diff1-eq-if*)
apply (*split split-if*, *rule conjI*)
apply *simp*
apply (*clarsimp simp*: *linorder-not-le*)
apply (*rule sym*)

thm *Suc-le-eq*[*of b mod m*, *THEN iffD2*]
apply (*drule Suc-le-eq*[*of b mod m*, *THEN iffD2*])
apply (*simp only*: *diff-add-assoc*)
apply (*simp only*: *div-add-self1*)
apply (*simp add*: *less-imp-diff-less*)
done

corollary *div-diff1-eq1*:
  $a\ mod\ m \leq b\ mod\ m \Longrightarrow$
  $(b - a)\ div\ (m::nat) = b\ div\ m - a\ div\ m$
by (*simp add*: *div-diff1-eq-if*)
corollary *div-diff1-eq1-mod-0*:
  $a\ mod\ m = 0 \Longrightarrow$
  $(b - a)\ div\ (m::nat) = b\ div\ m - a\ div\ m$
by (*simp add*: *div-diff1-eq1*)
corollary *div-diff1-eq2*:
  $b\ mod\ m < a\ mod\ m \Longrightarrow$
  $(b - a)\ div\ (m::nat) = b\ div\ m - Suc\ (a\ div\ m)$
by (*simp add*: *div-diff1-eq-if*)

5.8 Further results about *div* and *mod*

5.8.1 Some auxiliary facts about *mod*

lemma *diff-less-divisor-imp-sub-mod-eq*:
  $[\![(x::nat) \leq y;\ y - x < m]\!] \Longrightarrow x = y - (y - x)\ mod\ m$
by *simp*
lemma *diff-ge-divisor-imp-sub-mod-less*:
  $[\![(x::nat) \leq y;\ m \leq y - x;\ 0 < m]\!] \Longrightarrow x < y - (y - x)\ mod\ m$

apply (*simp only*: *less-diff-conv*)
apply (*simp only*: *le-diff-conv2 add-commute*[*of m*])
apply (*rule less-le-trans*[*of - x + m*])
apply *simp-all*
done
thm
diff-less-divisor-imp-sub-mod-eq
diff-ge-divisor-imp-sub-mod-less

lemma *le-imp-sub-mod-le*:
(*x*::*nat*) ≤ *y* ⟹ *x* ≤ *y* − (*y* − *x*) *mod m*
apply (*case-tac m* = *0*, *simp-all*)
thm
diff-less-divisor-imp-sub-mod-eq
diff-ge-divisor-imp-sub-mod-less
apply (*case-tac m* ≤ *y* − *x*)
thm *diff-ge-divisor-imp-sub-mod-less*
apply (*drule diff-ge-divisor-imp-sub-mod-less*[*of x y m*])
apply *simp-all*
done

thm *mod-diff1-eq*
lemma *mod-less-diff-mod*:
⟦ *n mod m* < *r*; *r* ≤ *m*; *r* ≤ (*n*::*nat*) ⟧ ⟹
(*n* − *r*) *mod m* = *m* + *n mod m* − *r*
apply (*case-tac r* = *m*)
thm *mod-diff-self2*
apply (*simp add*: *mod-diff-self2*)
apply (*simp add*: *mod-diff1-eq*[*of r n m*])
done

lemma *mod-0-imp-mod-pred*:
⟦ *0* < (*n*::*nat*); *n mod m* = *0* ⟧ ⟹
(*n* − *Suc 0*) *mod m* = *m* − *Suc 0*
apply (*case-tac m* = *0*, *simp-all*)
thm *mod-less-diff-mod*[*of n m Suc 0*]
apply (*simp only*: *Suc-le-eq*[*symmetric*])
thm *mod-diff1-eq*[*of Suc 0 n m*]
apply (*simp only*: *mod-diff1-eq*)
apply (*case-tac m* = *Suc 0*)
apply *simp-all*
done

lemma *mod-pred*:
0 < *n* ⟹
(*n* − *Suc 0*) *mod m* = (
if n mod m = *0 then m* − *Suc 0 else n mod m* − *Suc 0*)
apply (*split split-if*, *rule conjI*)
apply (*simp add*: *mod-0-imp-mod-pred*)

apply *clarsimp*
apply (*case-tac m = Suc 0*, *simp*)
thm *subst*[*OF Suc0-mod*[*symmetric*], **where** *P*=λ*x*. *x* ≤ *n mod m*]
apply (*frule subst*[*OF Suc0-mod*[*symmetric*], **where** *P*=λ*x*. *x* ≤ *n mod m*], *simp*)
thm *mod-diff1-eq1*[*of Suc 0 n m*]
apply (*simp only*: *mod-diff1-eq1*)
thm *Suc0-mod*
apply (*simp add*: *Suc0-mod*)
done
corollary *mod-pred-Suc-mod*:
0 < n ⟹ *Suc* ((*n − Suc 0*) *mod m*) *mod m* = *n mod m*
apply (*case-tac m = 0*, *simp*)
apply (*simp add*: *mod-pred*)
done
corollary *diff-mod-pred*:
a < b ⟹
(*b − Suc a*) *mod m* = (
if a mod m = b mod m then m − Suc 0 else (*b − a*) *mod m − Suc 0*)
apply (*rule-tac t*=*b − Suc a* **and** *s*=*b − a − Suc 0* **in** *subst*, *simp*)
apply (*subst mod-pred*, *simp*)
apply (*simp add*: *mod-eq-diff-mod-0-conv*)
done
corollary *diff-mod-pred-Suc-mod*:
a < b ⟹ *Suc* ((*b − Suc a*) *mod m*) *mod m* = (*b − a*) *mod m*
apply (*case-tac m = 0*, *simp*)
apply (*simp add*: *diff-mod-pred mod-eq-diff-mod-0-conv*)
done

lemma *mod-eq-imp-diff-mod-eq-divisor*:
⟦ *a < b*; *0 < m*; *a mod m* = *b mod m* ⟧ ⟹
Suc ((*b − Suc a*) *mod m*) = *m*
thm *mod-eq-imp-diff-mod-0*
apply (*drule mod-eq-imp-diff-mod-0*[*of a*])
thm *iffD2*[*OF zero-less-diff*]
apply (*frule iffD2*[*OF zero-less-diff*])
thm *mod-0-imp-mod-pred*[*of b − a m*]
apply (*drule mod-0-imp-mod-pred*[*of b−a m*], *assumption*)
apply *simp*
done

lemma *sub-diff-mod-eq*:
r ≤ *t* ⟹ (*t* − (*t* − *r*) *mod m*) *mod* (*m*::*nat*) = *r mod m*
by (*metis mod-diff-right-eq diff-diff-cancel diff-le-self*)

thm *diff-mod-le*
lemma *sub-diff-mod-eq'*:
r ≤ *t* ⟹ (*k* ∗ *m* + *t* − (*t* − *r*) *mod m*) *mod* (*m*::*nat*) = *r mod m*
thm *diff-mod-le*[*of t r m*]

apply (*simp only*: *diff-mod-le*[*of t r m*, *THEN add-diff-assoc*, *symmetric*])
apply (*simp add*: *sub-diff-mod-eq*)
done

lemma *mod-eq-Suc-0-conv*: *Suc 0 < k* $\Longrightarrow$ ((*x + k − Suc 0*) *mod k = 0*) = (*x mod k = Suc 0*)
apply (*simp only*: *mod-pred*)
apply (*case-tac x mod k = Suc 0*)
apply *simp-all*
done

lemma *mod-eq-divisor-minus-Suc-0-conv*: *Suc 0 < k* $\Longrightarrow$ (*x mod k = k − Suc 0*) = (*Suc x mod k = 0*)
by (*simp only*: *mod-Suc*, *split split-if*, *fastsimp*)

5.8.2 Some auxiliary facts about *div*

lemma *sub-mod-div-eq-div*: ((*n*::*nat*) *− n mod m*) *div m = n div m*
apply (*case-tac m = 0*, *simp*)
thm *mult-div-cancel*[*of m n*, *symmetric*]
apply (*simp add*: *mult-div-cancel*[*symmetric*])
done

thm *split-div-lemma*
lemma *mod-less-imp-diff-div-conv*:
⟦ *n mod m < r*; *r ≤ m + n mod m*⟧ $\Longrightarrow$ (*n − r*) *div m = n div m − Suc 0*
apply (*case-tac m = 0*, *simp*)
apply (*simp only*: *neq0-conv*)
apply (*case-tac n < m*, *simp*)
apply (*simp only*: *linorder-not-less*)
thm *split-div-lemma*[*of m n div m − Suc 0 n−r*]
thm *iffD1*[*OF split-div-lemma*[*of m n div m − Suc 0 n−r*], *symmetric*]
apply (*rule iffD1*[*OF split-div-lemma*, *symmetric*], *assumption*)
apply (*rule conjI*)
apply (*simp-all add*: *diff-mult-distrib2 mult-div-cancel*)
done

corollary *mod-0-le-imp-diff-div-conv*:
⟦ *n mod m = 0*; *0 < r*; *r ≤ m* ⟧ $\Longrightarrow$ (*n − r*) *div m = n div m − Suc 0*
thm *mod-less-imp-diff-div-conv*[*of n m r*]
by (*simp add*: *mod-less-imp-diff-div-conv*)
corollary *mod-0-less-imp-diff-Suc-div-conv*:
⟦ *n mod m = 0*; *r < m* ⟧ $\Longrightarrow$ (*n − Suc r*) *div m = n div m − Suc 0*
by (*drule mod-0-le-imp-diff-div-conv*[**where** *r=Suc r*], *simp-all*)
corollary *mod-0-imp-diff-Suc-div-conv*:
(*n − r*) *mod m = 0* $\Longrightarrow$ (*n − Suc r*) *div m* = (*n − r*) *div m − Suc 0*
apply (*case-tac m = 0*, *simp*)
apply (*rule-tac t=n − Suc r* **and** *s=n − r − Suc 0* **in** *subst*, *simp*)
apply (*rule mod-0-le-imp-diff-div-conv*, *simp+*)

done
corollary *mod-0-imp-sub-1-div-conv*:
  $n\ mod\ m = 0 \Longrightarrow (n - Suc\ 0)\ div\ m = n\ div\ m - Suc\ 0$
thm *mod-0-less-imp-diff-Suc-div-conv*[**where** *r=0*]
apply (*case-tac* $m = 0$, *simp*)
apply (*simp add*: *mod-0-less-imp-diff-Suc-div-conv*)
done
corollary *sub-Suc-mod-div-conv*:
  $(n - Suc\ (n\ mod\ m))\ div\ m = n\ div\ m - Suc\ 0$
apply (*case-tac* $m = 0$, *simp*)
apply (*simp add*: *mod-less-imp-diff-div-conv*)
done

thm *Divides.div-le-mono*
lemma *div-le-conv*: $0 < m \Longrightarrow n\ div\ m \leq k = (n \leq Suc\ k * m - Suc\ 0)$
apply (*rule iffI*)
 apply (*drule mult-le-mono1*[*of* - - *m*])
 apply (*simp only*: *mult-commute*[*of* - *m*] *mult-div-cancel*)
 apply (*drule le-diff-conv*[*THEN iffD1*])
 apply (*rule le-trans*[*of* - $m * k + n\ mod\ m$], *assumption*)
 apply (*simp add*: *add-commute*[*of m*])
 thm *diff-add-assoc*[*OF Suc-leI*]
 apply (*simp only*: *diff-add-assoc*[*OF Suc-leI*])
 thm *add-le-mono*[*OF le-refl*]
 apply (*rule add-le-mono*[*OF le-refl*])
 apply (*rule less-imp-le-pred*)
 apply (*rule mod-less-divisor*, *assumption*)
apply (*drule div-le-mono*[*of* - - *m*])
thm *mod-0-imp-sub-1-div-conv*
apply (*simp add*: *mod-0-imp-sub-1-div-conv*)
done

lemma *le-div-conv*: $0 < (m::nat) \Longrightarrow (n \leq k\ div\ m) = (n * m \leq k)$
apply (*rule iffI*)
 apply (*drule mult-le-mono1*[*of* - - *m*])
 apply (*simp add*: *div-mult-cancel*)
apply (*drule div-le-mono*[*of* - - *m*])
apply *simp*
done

lemma *less-mult-imp-div-less*: $n < k * m \Longrightarrow n\ div\ m < (k::nat)$
apply (*case-tac* $k = 0$, *simp*)
apply (*case-tac* $m = 0$, *simp*)
apply *simp*
apply (*drule less-imp-le-pred*[*of n*])

apply (*drule div-le-mono*[*of* - - *m*])
thm *mod-0-imp-sub-1-div-conv*
apply (*simp add*: *mod-0-imp-sub-1-div-conv*)
done

lemma *div-less-imp-less-mult*: ⟦ *0* < (*m*::*nat*); *n div m* < *k* ⟧ ⟹ *n* < *k* ∗ *m*
apply (*rule ccontr*, *simp only*: *linorder-not-less*)
apply (*drule div-le-mono*[*of* - - *m*])
apply *simp*
done

lemma *div-less-conv*: *0* < (*m*::*nat*) ⟹ (*n div m* < *k*) = (*n* < *k* ∗ *m*)
apply (*rule iffI*)
apply (*rule div-less-imp-less-mult*, *assumption*+)
apply (*rule less-mult-imp-div-less*, *assumption*)
done

lemma *div-eq-0-conv*: (*n div* (*m*::*nat*) = *0*) = (*m* = *0* ∨ *n* < *m*)
apply (*rule iffI*)
 apply (*case-tac m* = *0*, *simp*)
 apply (*rule ccontr*)
 apply (*simp add*: *linorder-not-less*)
 apply (*drule div-le-mono*[*of* - - *m*])
 apply *simp*
apply *fastsimp*
done
lemma *div-eq-0-conv'*: *0* < *m* ⟹ (*n div* (*m*::*nat*) = *0*) = (*n* < *m*)
by (*simp add*: *div-eq-0-conv*)
corollary *div-gr-imp-gr-divisor*: *x* < *n div* (*m*::*nat*) ⟹ *m* ≤ *n*
apply (*drule gr-implies-gr0*, *drule neq0-conv*[*THEN iffD2*])
apply (*simp add*: *div-eq-0-conv*)
done

lemma *mod-0-less-div-conv*:
  n mod (*m*::*nat*) = *0* ⟹ (*k* ∗ *m* < *n*) = (*k* < *n div m*)
apply (*case-tac m* = *0*, *simp*)
apply *fastsimp*
done

lemma *add-le-divisor-imp-le-Suc-div*:
  ⟦ *x div m* ≤ *n*; *y* ≤ *m* ⟧ ⟹ (*x* + *y*) *div m* ≤ *Suc n*
apply (*case-tac m* = *0*, *simp*)
apply (*simp only*: *div-add1-eq-if*[*of* - *x*])
apply (*drule order-le-less*[*of y*, *THEN iffD1*], *fastsimp*)
done

List of definitions and lemmas

thm
  Divides.mod-less

Divides.mod-less-divisor
Divides.mod-le-divisor
mod-less-dividend
mod-le-dividend

thm
Divides.mult-div-cancel
mod-0-div-mult-cancel
div-mult-le
less-div-Suc-mult
thm
Suc0-mod
Suc0-mod-subst
Suc0-mod-cong

thm
Divides.mod-Suc
thm
mod-Suc-conv

thm
mod-add
mod-sub-add

thm
mod-sub-eq-mod-0-conv
mod-sub-eq-mod-swap

thm
le-mod-greater-imp-div-less
thm
mod-diff-right-eq
mod-eq-imp-diff-mod-eq

thm
divisor-add-diff-mod-if
divisor-add-diff-mod-eq1
divisor-add-diff-mod-eq2

thm
mod-add-eq
mod-add1-eq-if
thm
mod-diff1-eq-if
mod-diff1-eq
mod-diff1-eq1
mod-diff1-eq2

thm

Divides.nat-mod-distrib
int-mod-distrib

thm
zmod-zminus-eq-conv

thm
mod-eq-imp-diff-mod-0
zmod-eq-imp-diff-mod-0

thm
mod-neq-imp-diff-mod-neq0
diff-mod-0-imp-mod-eq
zdiff-mod-0-imp-mod-eq

thm
zmod-eq-diff-mod-0-conv
mod-eq-diff-mod-0-conv

thm
less-mod-eq-imp-add-divisor-le
thm
mod-add-eq-imp-mod-0
thm
mod-eq-mult-distrib
mod-factor-imp-mod-0
mod-factor-div
mod-factor-div-mod

thm
Divides.mod-add-self1
Divides.mod-add-self2
Divides.mod-mult-self1
Divides.mod-mult-self2

mod-diff-self1
mod-diff-self2
mod-diff-mult-self1
mod-diff-mult-self2

thm
Divides.div-add-self1
Divides.div-add-self2
Divides.div-mult-self1
Divides.div-mult-self2

div-diff-self1
div-diff-self2

div-diff-mult-self1
div-diff-mult-self2

thm
le-less-imp-div
div-imp-le-less
thm
le-less-div-conv

thm
diff-less-divisor-imp-sub-mod-eq
diff-ge-divisor-imp-sub-mod-less
le-imp-sub-mod-le

thm
sub-mod-div-eq-div

thm
mod-less-imp-diff-div-conv
mod-0-le-imp-diff-div-conv
mod-0-less-imp-diff-Suc-div-conv
mod-0-imp-sub-1-div-conv

thm
sub-Suc-mod-div-conv

thm
mod-less-diff-mod
mod-0-imp-mod-pred

thm
mod-pred
mod-pred-Suc-mod

thm
mod-eq-imp-diff-mod-eq-divisor

thm
diff-mod-le
sub-diff-mod-eq
sub-diff-mod-eq'

thm
Divides.div-add1-eq
div-add1-eq-if
div-add1-eq1
div-add1-eq2
thm

div-diff1-eq-if
div-diff1-eq
div-diff1-eq1
div-diff1-eq2

thm
div-le-conv

end

6 SetInterval2: Sets of natural numbers

theory *SetInterval2*
imports
$*ISABELLE-HOME*/*src*/*HOL*/*Library*/*Infinite-Set*
Util-Set
../*CommonArith*/*Util-MinMax*
../*CommonArith*/*Util-NatInf*
../*CommonArith*/*Util-Div*
begin

6.1 Auxiliary results for monotonic, injective and surjective functions over sets

6.1.1 Monotonicity

thm *Orderings.strict-mono-def*

thm *mono-def*
definition *mono-on* :: ('*a*::*order* ⇒ '*b*::*order*) ⇒ '*a set* ⇒ *bool* **where**
mono-on f A ≡ ∀ *a*∈*A*. ∀ *b*∈*A*. *a* ≤ *b* ⟶ *f a* ≤ *f b*
definition *strict-mono-on* :: ('*a*::*order* ⇒ '*b*::*order*) ⇒ '*a set* ⇒ *bool* **where**
strict-mono-on f A ≡ ∀ *a*∈*A*. ∀ *b*∈*A*. *a* < *b* ⟶ *f a* < *f b*

lemma *mono-on-subset*: ⟦ *mono-on f A* ; *B* ⊆ *A* ⟧ ⟹ *mono-on f B*
unfolding *mono-on-def* **by** *blast*
lemma *strict-mono-on-subset*: ⟦ *strict-mono-on f A* ; *B* ⊆ *A* ⟧ ⟹ *strict-mono-on f B*
unfolding *strict-mono-on-def* **by** *blast*

lemma *mono-on-UNIV-mono-conv*: *mono-on f UNIV* = *mono f*
unfolding *mono-on-def mono-def* **by** *blast*
lemma *strict-mono-on-UNIV-strict-mono-conv*:
strict-mono-on f UNIV = *strict-mono f*
unfolding *strict-mono-on-def strict-mono-def* **by** *blast*

lemma *mono-imp-mono-on*: *mono f* $\Longrightarrow$ *mono-on f A*
unfolding *mono-on-def mono-def* **by** *blast*
lemma *strict-mono-imp-strict-mono-on*: *strict-mono f* $\Longrightarrow$ *strict-mono-on f A*
unfolding *strict-mono-on-def strict-mono-def* **by** *blast*

lemma *strict-mono-on-imp-mono-on*: *strict-mono-on f A* $\Longrightarrow$ *mono-on f A*
apply (*unfold strict-mono-on-def mono-on-def*)
apply (*fastsimp simp*: *order-le-less*)
done

6.1.2 Injectivity

lemma *inj-imp-inj-on*: *inj f* $\Longrightarrow$ *inj-on f A*
unfolding *inj-on-def* **by** *blast*

lemma *strict-mono-on-imp-inj-on*:
strict-mono-on f (*A*::*'a*::*linorder set*) $\Longrightarrow$ *inj-on f A*
apply (*unfold strict-mono-on-def inj-on-def*, *clarify*)
apply (*rule ccontr*)
apply (*fastsimp simp add*: *linorder-neq-iff*)
done

lemma *strict-mono-imp-inj*: *strict-mono* (*f*::(*'a*::*linorder* $\Rightarrow$ *'b*::*order*)) $\Longrightarrow$ *inj f*
by (*rule strict-mono-imp-inj-on*)

lemma *strict-mono-on-mono-on-conv*:
strict-mono-on f (*A*::*'a*::*linorder set*) = (*mono-on f A* $\wedge$ *inj-on f A*)
apply (*rule iffI*)
apply (*frule strict-mono-on-imp-mono-on*)
apply (*frule strict-mono-on-imp-inj-on*)
apply *blast*
apply (*erule conjE*)
apply (*unfold inj-on-def mono-on-def strict-mono-on-def*, *clarify*)
apply *fastsimp*
done

corollary *strict-mono-mono-conv*:
strict-mono (*f*::(*'a*::*linorder* $\Rightarrow$ *'b*::*order*)) = (*mono f* $\wedge$ *inj f*)
by (*simp only*: *strict-mono-on-UNIV-strict-mono-conv*[*symmetric*]
mono-on-UNIV-mono-conv[*symmetric*] *strict-mono-on-mono-on-conv*)

thm *inj-image-mem-iff*

lemma *inj-on-image-mem-iff*:
⟦ *inj-on f A*; *B* ⊆ *A*; *a* ∈ *A* ⟧ ⟹ (*f a* ∈ *f* ‘ *B*) = (*a* ∈ *B*)
unfolding *inj-on-def* **by** *blast*

thm *Set.image-Un*
thm *Fun.inj-on-def*

thm *image-Int*
thm *inj-on-image-Int*
corollary *inj-on-union-image-Int*:
inj-on f (*A* ∪ *B*) ⟹ *f* ‘ (*A* ∩ *B*) = *f* ‘ *A* ∩ *f* ‘ *B*
thm *inj-on-image-Int*[*OF - Un-upper1 Un-upper2*]
by (*rule inj-on-image-Int*[*OF - Un-upper1 Un-upper2*])

6.1.3 Surjectivity

thm *surj-def*

definition
surj-on :: ('*a* ⇒ '*b*) ⇒ '*a set* ⇒ '*b set* ⇒ *bool*
where
surj-on f A B ≡ ∀ *b*∈*B*. ∃ *a*∈*A*. *b* = *f a*
thm *surj-on-def*

lemma *surj-on-conv*: (*surj-on f A B*) = (∀ *b*∈*B*. ∃ *a*∈*A*. *b* = *f a*)
unfolding *surj-on-def* **..**

lemma *surj-on-image-conv*: (*surj-on f A B*) = (*B* ⊆ *f* ‘ *A*)
unfolding *surj-on-conv* **by** *blast*

lemma *surj-on-id*: *surj-on id A A*
unfolding *id-def surj-on-conv* **by** *blast*

lemma
surj-onI: ⟦ ∀ *b* ∈ *B*. ∃ *a*∈*A*. *b* = *f a* ⟧ ⟹ *surj-on f A B* **and**
surj-onD2: *surj-on f A B* ⟹ ∀ *b* ∈ *B*. ∃ *a*∈*A*. *b* = *f a* **and**
surj-onD: ⟦ *surj-on f A B*; *b* ∈ *B* ⟧ ⟹ ∃ *a*∈*A*. *b* = *f a*
unfolding *surj-on-conv*
by *blast*+

thm *comp-surj*
lemma *comp-surj-on*:
⟦ *surj-on f A B*; *surj-on g B C* ⟧ ⟹ *surj-on* (*g* ∘ *f*) *A C*
unfolding *comp-def surj-on-image-conv* **by** *blast*

thm
inj-on-Un
inj-on-diff
inj-on-empty
inj-on-imageI
inj-on-insert
subset-inj-on
lemma *surj-on-Un-right*: *surj-on f A* (*B1* ∪ *B2*) = (*surj-on f A B1* ∧ *surj-on f A B2*)
unfolding *surj-on-image-conv*
by *blast*
lemma *surj-on-Un-left*:
surj-on f (*A1* ∪ *A2*) *B* =
(∃ *B1*. ∃ *B2*. *B* ⊆ *B1* ∪ *B2* ∧ *surj-on f A1 B1* ∧ *surj-on f A2 B2*)
unfolding *surj-on-image-conv*
apply (*rule iffI*)
apply (*rule-tac x*=*f* ‘ *A1* **in** *exI*)
apply (*rule-tac x*=*f* ‘ *A2* **in** *exI*)
apply *blast*
apply *blast*
done
lemma *surj-on-diff-right*: *surj-on f A B* ⟹ *surj-on f A* (*B* − *B'*)
unfolding *surj-on-conv* **by** *blast*
lemma *surj-on-empty-right*: *surj-on f A* {}
unfolding *surj-on-conv* **by** *blast*
lemma *surj-on-empty-left*: *surj-on f* {} *B* = (*B* = {})
unfolding *surj-on-conv* **by** *blast*
lemma *surj-on-imageI*: *surj-on* (*g* ∘ *f*) *A B* ⟹ *surj-on g* (*f* ‘ *A*) *B*
unfolding *surj-on-conv* **by** *fastsimp*
lemma *surj-on-insert-right*: *surj-on f A* (*insert b B*) = (*surj-on f A B* ∧ *surj-on f A* {*b*})
unfolding *surj-on-conv* **by** *blast*
lemma *surj-on-insert-left*: *surj-on f* (*insert a A*) *B* = (*surj-on f A* (*B* − {*f a*}))
unfolding *surj-on-conv* **by** *blast*
lemma *surj-on-subset-right*: ⟦ *surj-on f A B*; *B'* ⊆ *B* ⟧ ⟹ *surj-on f A B'*
unfolding *surj-on-conv* **by** *blast*
lemma *surj-on-subset-left*: ⟦ *surj-on f A B*; *A* ⊆ *A'* ⟧ ⟹ *surj-on f A' B*
unfolding *surj-on-conv* **by** *blast*

lemma *bij-betw-imp-surj-on*: *bij-betw f A B* ⟹ *surj-on f A B*
unfolding *bij-betw-def surj-on-image-conv* **by** *simp*

lemma *bij-betw-inj-on-surj-on-conv*:
bij-betw f A B = (*inj-on f A* ∧ *surj-on f A B* ∧ *f* ‘ *A* ⊆ *B*)
unfolding *bij-betw-def surj-on-image-conv* **by** *blast*

6.1.4 Induction over natural sets

lemma *image-nat-induct*:

⟦ *P* (*f* 0); ⋀*n*. *P* (*f n*) ⟹ *P* (*f* (*Suc n*)); *surj-on f UNIV I*; *a* ∈ *I* ⟧ ⟹ *P a*
proof −
assume *as-P0*: *P* (*f* 0)
and *as-IA*: ⋀*n*. *P* (*f n*) ⟹ *P* (*f* (*Suc n*))
and *as-surj-f*: *surj-on f UNIV I*
and *as-a*: *a* ∈ *I*
have *P-n*:⋀*n*. *P* (*f n*)
apply (*induct-tac n*)
apply (*simp only*: *as-P0*)
apply (*simp only*: *as-IA*)
done
have ∀ *x*∈*I*. ∃ *n*. *x* = *f n*
using *as-surj-f*
by (*unfold surj-on-conv*, *blast*)
hence ∃ *n*. *a* = *f n*
using *as-a* **by** *blast*
thus *P a*
using *P-n* **by** *blast*
qed
thm *image-nat-induct*

thm *nat-induct*
lemma *nat-induct'*[*rule-format*]:
⟦ *P n0*; ⋀*n*. ⟦ *n0* ≤ *n*; *P n* ⟧ ⟹ *P* (*Suc n*); *n0* ≤ *n* ⟧ ⟹ *P n*
thm *nat-induct*
thm *nat-induct*[**where** *n*=*n*−*n0* **and** *P*=λ*n*. *P* (*n0*+*n*)]
by (*insert nat-induct*[**where** *n*=*n*−*n0* **and** *P*=λ*n*. *P* (*n0*+*n*)], *simp*)

thm
nat-induct'
nat-induct'[**where** *?n0.0*=*0*, *simplified*]
nat-induct

lemma *inat-induct*:
⟦ *P* 0; *P* ∞; ⋀*n*. *P n* ⟹ *P* (*iSuc n*)⟧ ⟹ *P n*
apply (*case-tac n*)
prefer *2*
apply *simp*
apply (*simp only*: *inat-defs*)
apply (*rename-tac n1*)
apply (*induct-tac n1*)
apply (*simp add*: *inat-splits*)+
done

lemma *iSuc-imp-Suc-aux-0*:
  $[\![\bigwedge n.\ P\ n \Longrightarrow P\ (iSuc\ n);\ n0' \leq n';\ P\ (Fin\ n')]\!] \Longrightarrow P\ (Fin\ (Suc\ n'))$
by (*simp only*: *inat-defs inat-splits*)
lemma *iSuc-imp-Suc-aux-n0*:
  $[\![\bigwedge n.\ [\![Fin\ n0' \leq n;\ P\ n]\!] \Longrightarrow P\ (iSuc\ n);\ n0' \leq n';\ P\ (Fin\ n')]\!] \Longrightarrow P\ (Fin\ (Suc\ n'))$
thm *inat-defs*
proof −
  assume *IA*: $\bigwedge n.\ [\![Fin\ n0' \leq n;\ P\ n]\!] \Longrightarrow P\ (iSuc\ n)$
    and *n0-n*: $n0' \leq n'$
    and *Pn*: $P\ (Fin\ n')$
  from *n0-n*
  have $(Fin\ n0' \leq Fin\ n')$ **by** *simp*
  with *Pn IA*
  have $P\ (iSuc\ (Fin\ n'))$ **by** *blast*
  thus $P\ (Fin\ (Suc\ n'))$ **by** (*simp only*: *iSuc-Fin*)
qed

lemma *inat-induct'*:
  $[\![P\ (n0::inat);\ P\ \infty;\ \bigwedge n.\ [\![n0 \leq n;\ \ P\ n]\!] \Longrightarrow P\ (iSuc\ n);\ n0 \leq n]\!] \Longrightarrow P\ n$
apply (*case-tac n*)
 prefer *2* **apply** *simp*
apply (*case-tac n0*)
 prefer *2* **apply** (*simp add*: *inat-defs*)
apply (*rename-tac n' n0'*, *simp*)

thm *nat-induct'*[**where** *?n0.0=n0'* **and** *n=n'* **and** $P=\lambda n.\ P\ (Fin\ n)$]
apply (*rule-tac ?n0.0=n0'* **and** *n=n'* **and** $P=\lambda n.\ P\ (Fin\ n)$ **in** *nat-induct'*)
  apply *simp*
 apply (*simp add*: *iSuc-Fin*[*symmetric*])
apply *simp*
done

thm
  inat-induct'
  inat-induct'[**where** *?n0.0=0*, *simplified*]
  inat-induct
thm *inat-induct'*

thm
  nat-induct
  nat-induct'
thm
  inat-induct
  inat-induct'

thm *wellorder-class.intro*
thm *wf-def*

thm
wf-less
wf-subset

lemma *wf-less-interval*:
wf { (*x*,*y*). *x* ∈ (*I*::*nat set*) ∧ *y* ∈ *I* ∧ *x* < *y* }
thm *wf-subset*
thm *wf-subset*[**where**
p={ (*x*,*y*). *x* ∈ *I* ∧ *y* ∈ *I* ∧ *x* < *y* } **and**
r={(*x*,*y*). *x* < *y*}]
apply (*rule wf-subset*[**where**
p={ (*x*,*y*). *x* ∈ *I* ∧ *y* ∈ *I* ∧ *x* < *y* } **and**
r={(*x*,*y*). *x* < *y*}])
thm *wf-less*
apply (*rule wf-less*)
apply *blast*
done
thm *wf-less-interval*

thm *wf-induct*
lemma *interval-induct*:
⟦ ⋀*x*. ∀ *y*. (*x*∈(*I*::*nat set*) ∧ *y* ∈ *I* ∧ *y* < *x* ⟶ *P y*) ⟹ *P x* ⟧
⟹ *P a*
(**is** ⟦ ⋀*x*. ∀ *y*. *?IA x y* ⟹ *P x* ⟧ ⟹ *P a*)
thm *wf-induct*
thm *wf-induct*[**where** *r*={ (*x*,*y*). *x* ∈ *I* ∧ *y* ∈ *I* ∧ *x* < *y* }]
apply (*rule-tac r*={ (*x*,*y*). *x* ∈ *I* ∧ *y* ∈ *I* ∧ *x* < *y* } **in** *wf-induct*)
apply (*rule wf-less-interval*)
apply *blast*
done
corollary *interval-induct-rule*:
⟦ ⋀*x*. (⋀*y*. (*x*∈(*I*::*nat set*) ∧ *y* ∈ *I* ∧ *y* < *x* ⟹ *P y*)) ⟹ *P x* ⟧
⟹ *P a*
by (*blast intro*: *interval-induct*)

thm
wf-induct
wf-induct-rule
interval-induct
interval-induct-rule

6.1.5 Monotonicity and injectivity of artithmetic operators

lemma *add-left-inj*: *inj* (λ*x*. *n* + (*x*::′*a*::*cancel-ab-semigroup-add*))
by (*simp add*: *inj-on-def*)
lemma *add-right-inj*: *inj* (λ*x*. *x* + (*n*::′*a*::*cancel-ab-semigroup-add*))
by (*simp add*: *inj-on-def*)
thm
add-left-inj

add-right-inj

lemma *mult-left-inj*: $0 < n \Longrightarrow inj\ (\lambda x.\ (n::nat) * x)$
by (*simp add*: *inj-on-def*)
lemma *mult-right-inj*: $0 < n \Longrightarrow inj\ (\lambda x.\ x * (n::nat))$
by (*simp add*: *inj-on-def*)
thm
mult-left-inj
mult-right-inj

lemma *sub-left-inj-on*: *inj-on* $(\lambda x.\ (x::nat) - k)\ \{k..\}$
by (*rule inj-onI*, *simp*)
lemma *sub-right-inj-on*: *inj-on* $(\lambda x.\ k - (x::nat))\ \{..k\}$
by (*rule inj-onI*, *simp*)

lemma *add-left-strict-mono*: *strict-mono* $(\lambda x.\ n + (x::'a::ordered\text{-}cancel\text{-}ab\text{-}semigroup\text{-}add))$
by (*unfold strict-mono-def*, *clarify*, *rule add-strict-left-mono*)
lemma *add-right-strict-mono*: *strict-mono* $(\lambda x.\ x + (n::'a::ordered\text{-}cancel\text{-}ab\text{-}semigroup\text{-}add))$
by (*unfold strict-mono-def*, *clarify*, *rule add-strict-right-mono*)

lemma *mult-left-strict-mono*: $0 < n \Longrightarrow strict\text{-}mono\ (\lambda x.\ n * (x::nat))$
by (*unfold strict-mono-def*, *clarify*, *simp*)
lemma *mult-right-strict-mono*: $0 < n \Longrightarrow strict\text{-}mono\ (\lambda x.\ x * (n::nat))$
by (*unfold strict-mono-def*, *clarify*, *simp*)
lemma *sub-left-strict-mono-on*: *strict-mono-on* $(\lambda x.\ (x::nat) - k)\ \{k..\}$
apply (*rule strict-mono-on-mono-on-conv*[*THEN iffD2*], *rule conjI*)
apply (*unfold mono-on-def*, *clarify*, *simp*)
apply (*rule sub-left-inj-on*)
done

lemma *div-right-strict-mono-on*:
$[\![\ 0 < (k::nat);\ \forall x{\in}I.\ \forall y{\in}I.\ x < y \longrightarrow x + k \leq y\ ]\!] \Longrightarrow$
strict-mono-on $(\lambda x.\ x\ div\ k)\ I$
apply (*unfold strict-mono-on-def*, *clarify*)
apply (*fastsimp dest*: *div-le-mono*[*of* - - *k*])
done

lemma *mod-eq-div-right-strict-mono-on*:
$[\![\ 0 < (k::nat);\ \forall x{\in}I.\ \forall y{\in}I.\ x\ mod\ k = y\ mod\ k\ ]\!] \Longrightarrow$
strict-mono-on $(\lambda x.\ x\ div\ k)\ I$
apply (*rule div-right-strict-mono-on*, *simp*)
thm *less-mod-eq-imp-add-divisor-le*
apply (*blast intro*: *less-mod-eq-imp-add-divisor-le*)
done

corollary *div-right-inj-on*:

$[\![\ 0 < (k::nat);\ \forall x{\in}I.\ \forall y{\in}I.\ x < y \longrightarrow x + k \leq y\ ]\!] \Longrightarrow$
$inj\text{-}on\ (\lambda x.\ x\ div\ k)\ I$
by (*rule strict-mono-on-imp-inj-on*[*OF div-right-strict-mono-on*])
corollary *mod-eq-imp-div-right-inj-on*:
$[\![\ 0 < (k::nat);\ \forall x{\in}I.\ \forall y{\in}I.\ x\ mod\ k = y\ mod\ k\ ]\!] \Longrightarrow$
$inj\text{-}on\ (\lambda x.\ x\ div\ k)\ I$
by (*rule strict-mono-on-imp-inj-on*[*OF mod-eq-div-right-strict-mono-on*])

6.2 *Min* and *Max* elements of a set

A special minimum operator is required for dealing with infinite wellordered sets because the standard operator *Min* is usable only with finite sets.

thm *Least-def*
definition
$iMin :: 'a::wellorder\ set \Rightarrow 'a$
where
$iMin\ I \equiv LEAST\ x.\ x \in I$

thm
Least-def
Nat.Least-Suc
Set.Least-mono
Orderings.not-less-Least
Orderings.LeastI2-ex
Orderings.LeastI2
Orderings.LeastI-ex
Orderings.Least-le
Orderings.LeastI
Orderings.wellorder-Least-lemma
Orderings.LeastI2-order
Orderings.Least-equality

6.2.1 Basic results, as for *Least*

thm *LeastI*
lemma *iMinI*: $k \in I \Longrightarrow iMin\ I \in I$
unfolding *iMin-def*
thm *LeastI*
by (*rule-tac k=k* **in** *LeastI*)

thm *LeastI-ex*
lemma *iMinI-ex*: $\exists x.\ x \in I \Longrightarrow iMin\ I \in I$
by (*blast intro*: *iMinI*)
corollary *iMinI-ex2*: $I \neq \{\} \Longrightarrow iMin\ I \in I$
by (*blast intro*: *iMinI-ex*)

thm *LeastI2*
lemma *iMinI2*: $[\![\ k \in I;\ \bigwedge x.\ x \in I \Longrightarrow P\ x\ ]\!] \Longrightarrow P\ (iMin\ I)$
thm *iMinI*
by (*blast intro*: *iMinI*)
thm *LeastI2-ex*
lemma *iMinI2-ex*: $[\![\ \exists x.\ x \in I;\ \bigwedge x.\ x \in I \Longrightarrow P\ x\ ]\!] \Longrightarrow P\ (iMin\ I)$
by (*blast intro*: *iMinI-ex*)
lemma *iMinI2-ex2*: $[\![\ I \neq \{\}\ ;\ \bigwedge x.\ x \in I \Longrightarrow P\ x\ ]\!] \Longrightarrow P\ (iMin\ I)$
by (*blast intro*: *iMinI-ex2*)

thm *Least-le*
lemma *iMin-le*[*dest*]: $k \in I \Longrightarrow iMin\ I \leq k$
by (*simp only*: *iMin-def Least-le*)

lemma *iMin-neq-imp-greater*[*dest*]: $[\![\ k \in I;\ k \neq iMin\ I\ ]\!] \Longrightarrow iMin\ I < k$
by (*rule order-neq-le-trans*[*OF not-sym iMin-le*])

thm *Least-mono*
lemma *iMin-mono*:
$[\![\ mono\ f;\ \exists x.\ x \in I\ ]\!] \Longrightarrow iMin\ (f\ `\ I) = f\ (iMin\ I)$
apply (*unfold iMin-def*)
thm *Least-mono*
thm *Least-mono*[*of f I*]
apply (*rule Least-mono*[*of f I*], *simp*)
apply (*rule-tac x*=*iMin I* **in** *bexI*)
thm *iMin-le*
apply (*simp add*: *iMin-le*)
thm *iMinI-ex*
apply (*simp add*: *iMinI-ex*)
done
corollary *iMin-mono2*:
$[\![\ mono\ f;\ I \neq \{\}\ ]\!] \Longrightarrow iMin\ (f\ `\ I) = f\ (iMin\ I)$
by (*blast intro*: *iMin-mono*)

thm *not-less-Least*
lemma *not-less-iMin*: $k < iMin\ I \Longrightarrow k \notin I$
unfolding *iMin-def*
thm *not-less-Least*
by (*rule not-less-Least*)

lemma *Collect-not-less-iMin*: $k < iMin\ \{x.\ P\ x\} \Longrightarrow \neg\ P\ k$
by (*drule not-less-iMin*, *blast*)
lemma *Collect-iMin-le*: $P\ k \Longrightarrow iMin\ \{x.\ P\ x\} \leq k$

by (*rule iMin-le*, *blast*)

lemma *Collect-minI*: ⟦ $k \in I$; $P\ (k::('a::wellorder))$ ⟧ ⟹ $\exists x{\in}I.\ P\ x \wedge (\forall y{\in}I.\ y < x \longrightarrow \neg\ P\ y)$
apply (*rule-tac x=iMin* $\{y \in I.\ P\ y\}$ **in** *bexI*)
prefer *2*
thm *iMinI*
thm *subsetD*
thm *Collect-is-subset*
thm *subsetD*[*OF - iMinI*, *OF Collect-is-subset*]
apply (*rule subsetD*[*OF - iMinI*, *OF Collect-is-subset*], *blast*)
apply (*rule conjI*)
apply (*blast intro*: *iMinI2*)
apply (*blast dest*: *not-less-iMin*)
done
corollary *Collect-minI-ex*: $\exists k{\in}I.\ P\ (k::('a::wellorder)) \Longrightarrow \exists x{\in}I.\ P\ x \wedge (\forall y{\in}I.\ y < x \longrightarrow \neg\ P\ y)$
by (*erule bexE*, *rule Collect-minI*)
corollary *Collect-minI-ex2*: $\{k{\in}I.\ P\ (k::('a::wellorder))\} \neq \{\} \Longrightarrow \exists x{\in}I.\ P\ x \wedge (\forall y{\in}I.\ y < x \longrightarrow \neg\ P\ y)$
by (*drule ex-in-conv*[*THEN iffD2*], *blast intro*: *Collect-minI*)

thm
Orderings.wellorder-Least-lemma
Orderings.Least-equality
Orderings.LeastI2-order
Least-def

thm *Least-def*
lemma *iMin-the*: *iMin I* = (*THE x*. $x \in I \wedge (\forall y.\ y \in I \longrightarrow x \leq y)$)
by (*simp add*: *iMin-def Least-def*)
lemma *iMin-the2*: *iMin I* = (*THE x*. $x \in I \wedge (\forall y{\in}I.\ x \leq y)$)
apply (*simp add*: *iMin-the*)
apply (*subgoal-tac* $\bigwedge x.\ (\forall y \in I.\ x \leq y) = (\forall y.\ y \in I \longrightarrow x \leq y)$)
prefer *2* **apply** *blast*
apply *simp*
done

thm *Least-equality*
lemma *iMin-equality*:
⟦ $k \in I$; $\bigwedge x.\ x \in I \Longrightarrow k \leq x$ ⟧ ⟹ *iMin I* = k
unfolding *iMin-def*
thm *Least-equality*
by (*blast intro*: *Least-equality*)

lemma *iMin-mono-on*:
⟦ *mono-on f I*; ∃ *x*. *x* ∈ *I* ⟧ ⟹ *iMin* (*f* ‘ *I*) = *f* (*iMin I*)
apply (*unfold mono-on-def*)
apply (*rule iMin-equality*)
thm *iMinI-ex*
apply (*blast intro*: *iMinI-ex*)+
done
lemma *iMin-mono-on2*:
⟦ *mono-on f I*; *I* ≠ {} ⟧ ⟹ *iMin* (*f* ‘ *I*) = *f* (*iMin I*)
by (*rule iMin-mono-on*, *blast*+)

thm *LeastI2-order*
lemma *iMinI2-order*:
⟦ *k* ∈ *I*; ⋀*y*. *y* ∈ *I* ⟹ *k* ≤ *y*;
⋀*x*. ⟦ *x* ∈ *I*; ∀ *y*∈*I*. *x* ≤ *y* ⟧ ⟹ *P x* ⟧ ⟹
P (*iMin I*)
thm *LeastI2-order*
thm *LeastI2-order*[*of* λ*x*. *x* ∈ *i k P*]
by (*simp add*: *iMin-def LeastI2-order*)

thm
iMinI2-order
thm
iMinI2
iMinI2-ex
iMinI2-ex2

thm *wellorder-Least-lemma*
lemma *wellorder-iMin-lemma*:
k ∈ *I* ⟹ *iMin I* ∈ *I* ∧ *iMin I* ≤ *k*
thm
iMinI
iMin-le
by (*blast intro*: *iMinI iMin-le*)

thm
iMin-the *iMin-the2*
thm
iMin-mono *iMin-mono2*
thm
iMin-le
not-less-iMin

thm
iMinI
iMinI-ex *iMinI-ex2*
thm
iMinI2
iMinI2-ex *iMinI2-ex2*
thm
wellorder-iMin-lemma
thm
iMin-equality
thm
iMinI2-order

thm *iMinI*

lemma *iMin-Min-conv*:
⟦ *finite I*; $I \neq \{\}$ ⟧ ⟹ $iMin\ I = Min\ I$
apply (*rule order-antisym*)

thm *Min-ge-iff*[*THEN iffD2*]
apply (*rule Min-ge-iff*[*THEN iffD2*], *assumption*+)
apply *blast*
thm *Min-le-iff*[*THEN iffD2*]
apply (*rule Min-le-iff*[*THEN iffD2*], *assumption*+)
apply (*blast intro*: *iMinI-ex2*)
done

lemma *Min-neq-imp-greater*[*dest*]: ⟦ *finite I*; $k \in I$; $k \neq Min\ I$ ⟧ ⟹ $Min\ I < k$
by (*rule order-neq-le-trans*[*OF not-sym Min-le*])
lemma *Max-neq-imp-less*[*dest*]: ⟦ *finite I*; $k \in I$; $k \neq Max\ I$ ⟧ ⟹ $k < Max\ I$
by (*rule order-neq-le-trans*[*OF - Max-ge*])

lemma *nat-Least-mono*:
⟦ $A \neq \{\}$; *mono* (f::($nat \Rightarrow nat$)) ⟧ ⟹
($LEAST\ x.\ x \in f\ `\ A) = f\ (LEAST\ x.\ x \in A)$
unfolding *iMin-def*[*symmetric*]
by (*blast intro*: *iMin-mono2*)

lemma *Least-disj*:
⟦ $\exists x.\ P\ x$; $\exists x.\ Q\ x$ ⟧ ⟹
($LEAST\ (x::'a::wellorder).\ (P\ x \vee Q\ x)) = min\ (LEAST\ x.\ P\ x)\ (LEAST\ x.\ Q\ x)$
apply (*elim exE*, *rename-tac x1 x2*)
apply (*subgoal-tac* $\bigwedge x1\ x2\ P\ Q.$ ⟦$P\ x1$; $Q\ x2$; $Least\ P \leq Least\ Q$⟧ ⟹ ($LEAST\ x.\ P\ x \vee Q\ x) = Least\ P$)
prefer *2*
apply (*rule Least-equality*)

apply (*blast intro*: *LeastI Least-le*)
apply (*erule disjE*)
apply (*rule Least-le*, *assumption*)
apply (*rule order-trans*, *assumption*)
apply (*rule Least-le*, *assumption*)
apply (*unfold min-def*, *split split-if*, *safe*)
apply *blast*
apply (*subst disj-commute*)
apply (*fastsimp simp*: *linorder-not-le*)
done

lemma *Least-imp-le*:
⟦ ∃ *x*. *P x*; ⋀*x*. *P x* ⟹ *Q x* ⟧ ⟹
(*LEAST* (*x*::′*a*::*wellorder*). *Q x*) ≤ (*LEAST x*. *P x*)
thm *Least-le LeastI2-ex*
by (*blast intro*: *Least-le LeastI2-ex*)

lemma *Least-imp-disj-eq*:
⟦ ∃ *x*. *P x*; ⋀*x*. *P x* ⟹ *Q x* ⟧ ⟹
(*LEAST* (*x*::′*a*::*wellorder*). *P x* ∨ *Q x*) = (*LEAST x*. *Q x*)
apply (*subst Least-disj*, *assumption*, *blast*)
apply (*subst min-max.inf-commute*)
apply (*rule min-max.le-iff-inf*[*THEN iffD1*])
apply (*rule Least-imp-le*, *assumption*, *blast*)
done

lemma *Least-le-imp-le*:
⟦ ∃ *x*. *P x*; ∃ *x*. *Q x*; ⋀*x y*. ⟦ *P x*; *Q y* ⟧ ⟹ *x* ≤ *y* ⟧ ⟹
(*LEAST* (*x*::′*a*::*wellorder*). *P x*) ≤ (*LEAST* (*x*::′*a*::*wellorder*). *Q x*)
by (*blast intro*: *LeastI*)
lemma *Least-le-imp-le-disj*:
⟦ ∃ *x*. *P x*; ⋀*x y*. ⟦ *P x*; *Q y* ⟧ ⟹ *x* ≤ *y* ⟧ ⟹
(*LEAST* (*x*::′*a*::*wellorder*). *P x* ∨ *Q x*) = (*LEAST* (*x*::′*a*::*wellorder*). *P x*)
thm *Least-imp-disj-eq*
apply (*case-tac* ∃ *x*. *Q x*)
apply (*simp only*: *Least-disj min-max.le-iff-inf*[*THEN iffD1*, *OF Least-le-imp-le*])
apply *simp*
done

thm *Max-le-iff*
thm *Max-less-iff*

thm *iMin-equality*
lemma *Max-equality*: ⟦ (*k*::′*a*::*linorder*) ∈ *A*; *finite A*; ⋀*x*. *x* ∈ *A* ⟹ *x* ≤ *k* ⟧ ⟹

Max A = *k*
by (*rule Max-eqI*)

thm
iMin-le
Max-ge

thm *not-less-iMin*
lemma *not-greater-Max*: $[\![\ \mathit{finite}\ A;\ \mathit{Max}\ A < k\ ]\!] \Longrightarrow k \notin A$
apply (*rule ccontr*, *simp*)
thm *Max-ge*[*of I k*]
apply (*frule Max-ge*[*of A k*], *blast*)
thm *order-le-less-trans*[*of k Max A k*]
apply (*frule order-le-less-trans*[*of - - k*], *blast*)
apply *blast*
done

lemma *Collect-not-greater-Max*: $[\![\ \mathit{finite}\ \{x.\ P\ x\};\ \mathit{Max}\ \{x.\ P\ x\} < k\ ]\!] \Longrightarrow \neg\ P\ k$
by (*drule not-greater-Max*, *assumption*, *drule Collect-not-in-imp-not*)
lemma *Collect-Max-ge*: $[\![\ \mathit{finite}\ \{x.\ P\ x\};\ P\ k\ ]\!] \Longrightarrow k \leq \mathit{Max}\ \{x.\ P\ x\}$
by (*rule Max-ge*, *assumption*, *rule CollectI*)

thm
iMinI-ex2
Max-in

thm *iMinI*
lemma *MaxI*: $[\![\ k \in A;\ \mathit{finite}\ A\ ]\!] \Longrightarrow \mathit{Max}\ A \in A$
by (*case-tac* $A = \{\}$, *simp-all*)

thm *iMinI-ex*
lemma *MaxI-ex*: $[\![\ \exists\, x.\ x \in A;\ \mathit{finite}\ A\ ]\!] \Longrightarrow \mathit{Max}\ A \in A$
by (*blast intro*: *MaxI*)
thm *iMinI-ex2*
lemma *MaxI-ex2*: $[\![\ A \neq \{\};\ \mathit{finite}\ A\ ]\!] \Longrightarrow \mathit{Max}\ A \in A$
by (*blast intro*: *MaxI*)

thm *iMinI2*
lemma *MaxI2*: $[\![\ k \in A;\ \bigwedge x.\ x \in A \Longrightarrow P\ x;\ \mathit{finite}\ A\ ]\!] \Longrightarrow P\ (\mathit{Max}\ A)$
thm *Max-in*
by (*drule Max-in*, *blast+*)
thm *iMinI2-ex*
lemma *MaxI2-ex*:$[\![\ \exists\, x.\ x \in A;\ \bigwedge x.\ x \in A \Longrightarrow P\ x;\ \mathit{finite}\ A\ ]\!] \Longrightarrow P\ (\mathit{Max}\ A)$
by (*blast intro*: *MaxI2*)
thm *iMinI2-ex2*
lemma *MaxI2-ex2*:$[\![\ A \neq \{\};\ \bigwedge x.\ x \in A \Longrightarrow P\ x;\ \mathit{finite}\ A\ ]\!] \Longrightarrow P\ (\mathit{Max}\ A)$
by (*blast intro*: *MaxI2*)

thm *iMin-mono*
lemma *Max-mono*: ⟦ *mono f*; ∃ *x*. *x* ∈ *A*; *finite A* ⟧ ⟹ *Max* (*f* ‘ *A*) = *f* (*Max A*)
apply (*unfold mono-def*)
apply *clarify*
apply (*frule Max-in*, *blast*)
thm *Max-equality*
apply (*rule Max-equality*, *clarsimp*+)
done

thm *iMin-mono2*
lemma *Max-mono2*:⟦ *mono f*; *A* ≠ {}; *finite A* ⟧ ⟹ *Max* (*f* ‘ *A*) = *f* (*Max A*)
by (*blast intro*: *Max-mono*)

thm *iMin-mono-on*
lemma *Max-mono-on*: ⟦ *mono-on f A*; ∃ *x*. *x* ∈ *A*; *finite A* ⟧ ⟹ *Max* (*f* ‘ *A*) = *f* (*Max A*)
apply (*unfold mono-on-def*)
apply (*rule Max-equality*)
apply (*blast intro*: *Max-in*)
apply (*rule finite-imageI*, *assumption*)
apply (*blast intro*: *Max-in Max-ge*)
done

lemma *Max-mono-on2*:
⟦ *mono-on f A*; *A* ≠ {}; *finite A* ⟧ ⟹ *Max* (*f* ‘ *A*) = *f* (*Max A*)
by (*rule Max-mono-on*, *blast*+)

thm *iMin-the*
lemma *Max-the*:
⟦ *A* ≠ {}; *finite A* ⟧ ⟹
Max A = (*THE x*. *x* ∈ *A* ∧ (∀ *y*. *y* ∈ *A* ⟶ *y* ≤ *x*))
thm *iffD1*[*OF eq-commute*]
apply (*rule iffD1*[*OF eq-commute*])
thm *the-equality*
apply (*rule the-equality*, *simp*)
thm *Max-equality*
apply (*rule sym*)
apply (*rule Max-equality*)
apply *blast*+
done

thm *iMin-the2*
lemma *Max-the2*: $[\![\ A \neq \{\};\ \mathit{finite}\ A\ ]\!] \Longrightarrow$
  $\mathit{Max}\ A = (\mathit{THE}\ x.\ x \in A \wedge (\forall\, y{\in}A.\ y \leq x))$
apply (*simp add*: *Max-the*)
apply (*subgoal-tac* $\bigwedge x.\ (\forall\, y{\in}A.\ y \leq x) = (\forall\, y.\ y \in A \longrightarrow y \leq x)$)
 prefer *2*
 apply *blast*
apply *simp*
done

thm *wellorder-iMin-lemma*
lemma *wellorder-Max-lemma*: $[\![\ k \in A;\ \mathit{finite}\ A\ ]\!] \Longrightarrow \mathit{Max}\ A \in A \wedge k \leq \mathit{Max}\ A$
by (*case-tac* $A = \{\}$, *simp-all*)

thm *iMinI2-order*
lemma *MaxI2-order*: $[\![\ k \in A;\ \mathit{finite}\ A;\ \bigwedge y.\ y \in A \Longrightarrow y \leq k;$
  $\bigwedge x.\ [\![\ x \in A;\ \forall\, y{\in}A.\ y \leq x\ ]\!] \Longrightarrow P\ x\ ]\!]$
  $\Longrightarrow P\ (\mathit{Max}\ A)$
thm *Max-equality*
by (*simp add*: *Max-equality*)

thm
  iMin-equality
  Max-equality
thm
  iMin-the iMin-the2
  Max-the Max-the2
thm
  iMin-mono iMin-mono2
  Max-mono Max-mono2
thm
  iMin-le
  Max-ge
thm
  not-less-iMin
  not-greater-Max
thm
  iMinI
  MaxI
thm
  iMinI-ex iMinI-ex2
  MaxI-ex MaxI-ex2
thm
  iMinI2
  MaxI2
thm
  iMinI2-ex iMinI2-ex2
  MaxI2-ex MaxI2-ex2
thm

wellorder-iMin-lemma
wellorder-Max-lemma
thm
iMinI2-order
MaxI2-order

lemma *Min-le-Max*: ⟦ *finite A*; $A \neq \{\}$ ⟧ ⟹ $Min\ A \leq Max\ A$
by (*rule Max-ge*[*OF - Min-in*])
lemma *iMin-le-Max*: ⟦ *finite A*; $A \neq \{\}$ ⟧ ⟹ $iMin\ A \leq Max\ A$
thm *subst*[*OF iMin-Min-conv*]
by (*rule ssubst*[*OF iMin-Min-conv*], *assumption+*, *rule Min-le-Max*)

6.2.2 *Max* for sets over *inat*

definition
$iMax :: nat\ set \Rightarrow inat$
where
$iMax\ i \equiv if\ (finite\ i)\ then\ (Fin\ (Max\ i))\ else\ \infty$

lemma *iMax-finite-conv*: $finite\ I = (iMax\ I \neq \infty)$
by (*simp add*: *iMax-def*)
lemma *iMax-infinite-conv*: $infinite\ I = (iMax\ I = \infty)$
by (*simp add*: *iMax-def*)

thm *lattice.inf-sup-aci*
thm *lattice-class.inf-sup-aci*
thm *semilattice-inf-class.inf-aci*
thm *semilattice-sup-class.sup-aci*

thm *lattice-class-def*
thm *lattice-class.axioms*

thm *distrib-lattice-class-def*

print-locale *distrib-lattice*
print-locale *lattice*
print-locale *distrib-lattice*
print-locale! *distrib-lattice*

thm *distrib-lattice-class.axioms*
interpretation *min-max2*:
$distrib\text{-}lattice\ op \leq :: 'a{::}linorder \Rightarrow 'a \Rightarrow bool\ op <\ min\ max$
..
print-theorems

term *distrib-lattice-class*
lemma *class.distrib-lattice* (*op* ≤) (*op* <) (*min*::('*a*::*linorder* ⇒ '*a* ⇒ '*a*)) *max*
print-locale *distrib-lattice*
thm *distrib-lattice-class.intro*
apply (*subgoal-tac class.order* (*op* ≤) (*op* <))
prefer *2*
apply (*rule class.order.intro*)
apply (*rule class.preorder.intro*)
apply (*rule less-le-not-le*)
apply (*rule order-refl*)
apply (*rule order-trans*, *assumption*+)
apply (*rule class.order-axioms.intro*)
apply (*rule order-antisym*, *assumption*+)
apply (*subgoal-tac class.linorder* (*op* ≤) (*op* <))
prefer *2*
apply (*rule class.linorder.intro*, *assumption*)
apply (*rule class.linorder-axioms.intro*)
apply (*rule linorder-class.linear*)
apply (*rule class.distrib-lattice.intro*)
apply (*rule class.lattice.intro*)
thm *class.semilattice-inf.intro*
apply (*rule class.semilattice-inf.intro*, *assumption*)
apply (*rule class.semilattice-inf-axioms.intro*)
apply (*rule le-minI1*)
apply (*rule le-minI2*)
apply (*rule conj-le-imp-min*, *assumption*+)
apply (*rule class.semilattice-sup.intro*, *assumption*)
apply (*rule class.semilattice-sup-axioms.intro*)
apply (*rule le-maxI1*)
apply (*rule le-maxI2*)
apply (*rule conj-le-imp-max*, *assumption*+)
apply (*rule class.distrib-lattice-axioms.intro*)
apply (*rule min-max.sup-inf-distrib1*)
done

lemma *class.distrib-lattice* (*op* ≥) (*op* >) (*max*::('*a*::*linorder* ⇒ '*a* ⇒ '*a*)) *min*
thm *linorder-class.min-max.dual-distrib-lattice*
by (*rule linorder-class.min-max.dual-distrib-lattice*)

print-locale *Lattices.distrib-lattice*

thm *Big-Operators.distrib-lattice.sup-Inf1-distrib*
lemma *max-Min-eq-Min-max*[*rule-format*]:
finite A ⟹
A ≠ {} ⟶
max x (*Min A*) = *Min* {*max x a* |*a*. *a* ∈ *A*}
thm *finite.induct*[*of A*]
apply (*rule finite.induct*[*of A*], *simp-all del*: *insert-iff*)
apply (*rename-tac A1 a*)

apply (*case-tac A1 = {}*, *simp*)
apply (*rule-tac*
  t={*max x b* |*b*. *b* ∈ *insert a A1*} **and**
  s=*insert* (*max x a*) {*max x b* |*b*. *b* ∈ *A1*}
  in *subst*)
 apply *blast*
apply (*subst Min-insert*, *simp-all*)
apply (*case-tac a* ≤ *Min A1*)
 apply (*frule max-le-monoR*[**where** *x*=*x*])
 apply (*simp only*: *min-eqL*)
apply (*simp only*: *linorder-not-le*)
thm *max-le-monoR*[*OF less-imp-le*]
apply (*frule max-le-monoR*[**where** *x*=*x*, *OF less-imp-le*])
apply (*simp only*: *min-eqR*)
done
thm *max-Min-eq-Min-max*

lemma *max-iMin-eq-iMin-max*:
  ⟦ *finite A*; *A* ≠ {} ⟧ ⟹
  max x (*iMin A*) = *iMin* {*max x a* |*a*. *a* ∈ *A*}
thm *iMin-Min-conv*
apply (*simp add*: *iMin-Min-conv*)
thm *iMin-Min-conv*[*of* {*max x a* |*a*. *a* ∈ *A*}]
apply (*insert iMin-Min-conv*[*of* {*max x a* |*a*. *a* ∈ *A*}], *simp*)
apply (*subgoal-tac finite* {*max x a* |*a*. *a* ∈ *A*})
 prefer *2*
 apply *simp*
thm *max-Min-eq-Min-max*
apply (*simp add*: *max-Min-eq-Min-max*)
done

lemma ⟦ *finite A*; *A* ≠{} ⟧ ⟹ ∀ *x*∈*A*. *x* ≤ *Max A*
by *simp*

6.2.3 *Min* and *Max* for set operations

lemma *iMin-subset*: ⟦ *A* ≠ {}; *A* ⊆ *B* ⟧ ⟹ *iMin B* ≤ *iMin A*
thm *iMin-le*[*of iMin A j*]
thm *iMinI-ex2*[*of A*]
by (*blast intro*: *iMin-le iMinI-ex2*)

lemma *Max-subset*: ⟦ *A* ≠ {}; *A* ⊆ *B*; *finite B* ⟧ ⟹ *Max A* ≤ *Max B*
by (*rule linorder-class.Max-mono*)
lemma *Min-subset*: ⟦ *A* ≠ {}; *A* ⊆ *B*; *finite B* ⟧ ⟹ *Min B* ≤ *Min A*
by (*rule linorder-class.Min-antimono*)

lemma *iMin-Int-ge1*: (*A* ∩ *B*) ≠ {} ⟹ *iMin A* ≤ *iMin* (*A* ∩ *B*)

thm *iMin-subset*[*OF - Int-lower1*]
by (*rule iMin-subset*[*OF - Int-lower1*])
lemma *iMin-Int-ge2*: $(A \cap B) \neq \{\} \Longrightarrow iMin\ B \leq iMin\ (A \cap B)$
by (*rule iMin-subset*[*OF - Int-lower2*])
lemma *iMin-Int-ge*: $(A \cap B) \neq \{\} \Longrightarrow max\ (iMin\ A)\ (iMin\ B) \leq iMin\ (A \cap B)$
apply (*rule conj-le-imp-max*)
apply (*rule iMin-Int-ge1*, *assumption*)
apply (*rule iMin-Int-ge2*, *assumption*)
done

lemma *Max-Int-le1*: $[\![\ (A \cap B) \neq \{\};\ finite\ A\ ]\!] \Longrightarrow Max\ (A \cap B) \leq Max\ A$
thm *Max-subset*[*OF - Int-lower1*]
by (*rule Max-subset*[*OF - Int-lower1*])
lemma *Max-Int-le2*: $[\![\ (A \cap B) \neq \{\};\ finite\ B\ ]\!] \Longrightarrow Max\ (A \cap B) \leq Max\ B$
by (*rule Max-subset*[*OF - Int-lower2*])
lemma *Max-Int-le*: $[\![\ (A \cap B) \neq \{\};\ finite\ A;\ finite\ B\ ]\!] \Longrightarrow$
$Max\ (A \cap B) \leq min\ (Max\ A)\ (Max\ B)$
apply (*rule conj-le-imp-min*)
apply (*rule Max-Int-le1*, *assumption*+)
apply (*rule Max-Int-le2*, *assumption*+)
done

lemma *iMin-Un*[*rule-format*]:
$[\![\ A \neq \{\};\ B \neq \{\}\ ]\!] \Longrightarrow$
$iMin\ (A \cup B) = min\ (iMin\ A)\ (iMin\ B)$
apply (*rule order-antisym*)
apply *simp*
apply (*blast intro*: *iMin-subset*)
thm *min-le-iff-disj*
apply (*simp add*: *min-le-iff-disj*)
thm *iMinI-ex2*[*of* $A \cup B$]
apply (*insert iMinI-ex2*[*of* $A \cup B$])
apply (*blast intro*: *iMin-le*)
done
thm *Min-Un*
thm *Max-Un*

thm *linorder-class.Min-singleton linorder-class.Max-singleton*
thm *singletonI*[*THEN iMinI*, *THEN singletonD*]
lemma *iMin-singleton*[*simp*]: $iMin\ \{a\} = a$
by (*rule singletonI*[*THEN iMinI*, *THEN singletonD*])
lemma *iMax-singleton*[*simp*]: $iMax\ \{a\} = Fin\ a$
by (*simp add*: *iMax-def*)

lemma *Max-le-Min-imp-singleton*:

⟦ *finite A*; *A* ≠ {}; *Max A* ≤ *Min A* ⟧ ⟹ *A* = {*Min A*}
thm *Max-ge*
thm *Max-le-iff*[*THEN iffD1*]
apply (*frule Max-le-iff*[*THEN iffD1*, *of* - *Min A*], *assumption*+)
apply (*frule Min-ge-iff*[*THEN iffD1*, *of* - *Min A*], *assumption*, *simp*)
apply (*rule set-eqI*)
apply (*unfold Ball-def*)
apply (*erule-tac x*=*x* **in** *allE*, *erule-tac x*=*x* **in** *allE*)
apply (*blast intro*: *order-antisym Min-in*)
done
lemma *Max-le-Min-conv-singleton*:
⟦ *finite A*; *A* ≠ {} ⟧ ⟹ (*Max A* ≤ *Min A*) = (∃ *x*. *A* = {*x*})
apply (*rule iffI*)
apply (*rule-tac x*=*Min A* **in** *exI*)
apply (*rule Max-le-Min-imp-singleton*, *assumption*+)
apply *fastsimp*
done

lemma *Max-le-iMin-imp-le*:
⟦ *finite A*; *Max A* ≤ *iMin B*; *a* ∈ *A*; *b* ∈ *B* ⟧ ⟹ *a* ≤ *b*
by (*blast dest*: *Max-ge intro*: *order-trans*)

lemma *le-imp-Max-le-iMin*:
⟦ *finite A*; *A* ≠ {}; *B* ≠ {}; ∀ *a*∈*A*. ∀ *b*∈*B*. *a* ≤ *b* ⟧ ⟹ *Max A* ≤ *iMin B*
by (*blast intro*: *Max-in iMinI-ex2*)

lemma *Max-le-iMin-conv-le*:
⟦ *finite A*; *A* ≠ {}; *B* ≠ {} ⟧ ⟹ (*Max A* ≤ *iMin B*) = (∀ *a*∈*A*. ∀ *b*∈*B*. *a* ≤ *b*)
by (*blast intro*: *Max-le-iMin-imp-le le-imp-Max-le-iMin*)

lemma *Max-less-iMin-imp-less*:
⟦ *finite A*; *Max A* < *iMin B*; *a* ∈ *A*; *b* ∈ *B* ⟧ ⟹ *a* < *b*
thm *Max-less-iff*[*THEN iffD1*, *of* - *iMin B*]
by (*blast dest*: *Max-less-iff intro*: *order-less-le-trans iMin-le*)

lemma *less-imp-Max-less-iMin*:
⟦ *finite A*; *A* ≠ {}; *B* ≠ {}; ∀ *a*∈*A*. ∀ *b*∈*B*. *a* < *b* ⟧ ⟹ *Max A* < *iMin B*
by (*blast intro*: *Max-in iMinI-ex2*)

lemma *Max-less-iMin-conv-less*:
⟦ *finite A*; *A* ≠ {}; *B* ≠ {} ⟧ ⟹ (*Max A* < *iMin B*) = (∀ *a*∈*A*. ∀ *b*∈*B*. *a* < *b*)
by (*blast intro*: *Max-less-iMin-imp-less less-imp-Max-less-iMin*)

lemma *Max-less-iMin-imp-disjoint*:
⟦ *finite A*; *Max A* < *iMin B* ⟧ ⟹ *A* ∩ *B* = {}
apply (*case-tac A* = {}, *simp*)
apply (*case-tac B* = {}, *simp*)

apply (*rule disjoint-iff-not-equal*[*THEN iffD2*])
apply (*intro ballI*)
apply (*rule less-imp-neq*)
by (*rule Max-less-iMin-imp-less*)

thm *Min.in-idem*
lemma *iMin-in-idem*: $n \in I \Longrightarrow min\ n\ (iMin\ I) = iMin\ I$
by (*simp add*: *iMin-le min-eqR*)

thm *Min-insert*
lemma *iMin-insert*: $I \neq \{\} \Longrightarrow iMin\ (insert\ n\ I) = min\ n\ (iMin\ I)$
apply (*subst insert-is-Un*)
apply (*subst iMin-Un*)
apply *simp-all*
done

thm *Min.insert-remove*
lemma *iMin-insert-remove*:
$iMin\ (insert\ n\ I) =$
$(if\ I - \{n\} = \{\}\ then\ n\ else\ min\ n\ (iMin\ (I - \{n\})))$
by (*metis iMin-insert iMin-singleton insert-Diff-single*)

thm *Min.remove*
lemma *iMin-remove*: $n \in I \Longrightarrow iMin\ I = (if\ I - \{n\} = \{\}\ then\ n\ else\ min\ n\ (iMin\ (I - \{n\})))$
by (*metis iMin-insert-remove insert-absorb*)

thm *Min.subset-idem*
lemma *iMin-subset-idem*: $[\![\ B \neq \{\};\ B \subseteq A\ ]\!] \Longrightarrow min\ (iMin\ B)\ (iMin\ A) = iMin\ A$
by (*metis iMin-subset min-max.inf-absorb2*)

thm *Min.union-inter*
lemma *iMin-union-inter*: $A \cap B \neq \{\} \Longrightarrow min\ (iMin\ (A \cup B))\ (iMin\ (A \cap B)) = min\ (iMin\ A)\ (iMin\ B)$
by (*metis Int-empty-left Int-lower2 Un-absorb2 Un-assoc Un-empty iMin-Un*)

thm *Min-ge-iff*
lemma *iMin-ge-iff*: $I \neq \{\} \Longrightarrow (n \leq iMin\ I) = (\forall\, a{\in}I.\ n \leq a)$
by (*metis Collect-iMin-le Collect-mem-eq iMinI-ex2 order-trans*)

thm *Min-gr-iff*
lemma *iMin-gr-iff*: $I \neq \{\} \Longrightarrow (n < iMin\ I) = (\forall\, a{\in}I.\ n < a)$
by (*metis iMinI-ex2 iMin-neq-imp-greater order-less-trans*)

thm *Min-le-iff*
lemma *iMin-le-iff*: $I \neq \{\} \Longrightarrow (iMin\ I \leq n) = (\exists\, a{\in}I.\ a \leq n)$
by (*metis Collect-iMin-le Collect-mem-eq iMinI-ex2 order-trans*)

thm *Min-less-iff*
lemma *iMin-less-iff*: $I \neq \{\} \Longrightarrow (iMin\ I < n) = (\exists\, a{\in}I.\ a < n)$
by (*metis iMinI-ex2 iMin-neq-imp-greater order-less-trans*)

thm *hom-Min-commute*
lemma *hom-iMin-commute*: $[\![\bigwedge x\ y.\ h\ (min\ x\ y) = min\ (h\ x)\ (h\ y);\ I \neq \{\}\ ]\!] \Longrightarrow$
$iMin\ (h\ `\ I) = h\ (iMin\ I)$
apply (*rule iMin-equality*)
apply (*blast intro*: *iMinI-ex2*)
apply (*rename-tac y*)
apply (*drule-tac x=iMin I* **in** *meta-spec*)
apply (*clarsimp simp*: *image-iff*, *rename-tac x*)
apply (*drule-tac x=x* **in** *meta-spec*)
apply (*rule-tac t=h* (*iMin I*) **and** *s=min* (*h* (*iMin I*)) (*h x*) **in** *subst*)
thm *min-eqL*[*OF iMin-le*]
apply (*simp add*: *min-eqL*[*OF iMin-le*])
apply *simp*
done

Synonyms for similarity with theorem names for *Min*"

thm *Min-eqI iMin-equality*
lemmas *iMin-eqI* = *iMin-equality*
thm *Min-in iMinI-ex2*
lemmas *iMin-in* = *iMinI-ex2*

6.3 Some auxiliary results for set operations

6.3.1 Some additional abbreviations for relations

Abbreviations for *refl*, *sym*, *equiv*, *refl* similar to *transP* from HOL/Predicate.

abbreviation *reflP* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$ **where**
reflP r ≡ *refl* $\{(x,\ y).\ r\ x\ y\}$

abbreviation *symP* :: $('a => 'a => bool) => bool$ **where**
symP r == *sym* $\{(x,\ y).\ r\ x\ y\}$

abbreviation *equivP* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$ **where**
equivP r ≡ *reflP r* ∧ *symP r* ∧ *transP r*

abbreviation *irreflP* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$ **where**
irreflP r ≡ *irrefl* $\{(x,\ y).\ r\ x\ y\}$

Example for *reflP*

lemma *reflP* $((op \leq)::('a::preorder \Rightarrow 'a \Rightarrow bool))$
by (*simp add*: *refl-on-def*)

Example for *symP*

lemma *symP* (*op* =)

by (*simp add*: *sym-def*)

Example for *equivP*

lemma *equivP* (*op* =)
by (*simp add*: *trans-def refl-on-def sym-def*)

Example for *irreflP*

lemma *irreflP* ((*op* <)::('*a*::*preorder* ⇒ '*a* ⇒ *bool*))
by (*simp add*: *irrefl-def*)

6.3.2 Auxiliary results for *singletons*

lemma *singleton-not-empty*: $\{a\} \neq \{\}$ **by** *blast*
lemma *singleton-finite*: *finite* $\{a\}$ **by** *blast*

lemma *singleton-ball*: $(\forall x{\in}\{a\}.\ P\ x) = P\ a$ **by** *blast*
lemma *singleton-bex*: $(\exists x{\in}\{a\}.\ P\ x) = P\ a$ **by** *blast*

thm *Set.subset-singletonD*
lemma *subset-singleton-conv*: $(A \subseteq \{a\}) = (A = \{\} \vee A = \{a\})$ **by** *blast*
lemma *singleton-subset-conv*: $(\{a\} \subseteq A) = (a \in A)$ **by** *blast*
thm *Set.singleton-inject*
lemma *singleton-eq-conv*: $(\{a\} = \{b\}) = (a = b)$ **by** *blast*
lemma *singleton-subset-singleton-conv*: $(\{a\} \subseteq \{b\}) = (a = b)$ **by** *blast*

lemma *singleton-Int1-if*: $\{a\} \cap A = (\mathit{if}\ a \in A\ \mathit{then}\ \{a\}\ \mathit{else}\ \{\})$
by (*split split-if*, *blast*)
lemma *singleton-Int2-if*: $A \cap \{a\} = (\mathit{if}\ a \in A\ \mathit{then}\ \{a\}\ \mathit{else}\ \{\})$
by (*split split-if*, *blast*)

lemma *singleton-image*: $f\ `\ \{a\} = \{f\ a\}$ **by** *blast*
lemma *inj-on-singleton*: *inj-on f* $\{a\}$ **by** *blast*
lemma *strict-mono-on-singleton*: *strict-mono-on f* $\{a\}$
unfolding *strict-mono-on-def* **by** *blast*

Auxiliary results for *empty* sets

thm *empty-imp-not-in*
thm *ex-imp-not-empty*
thm *in-imp-not-empty*

6.3.3 Auxiliary results for *finite* and *infinite* sets

thm *infinite-imp-nonempty*
lemma *infinite-imp-not-singleton*: *infinite* $A \Longrightarrow \neg\ (\exists\, a.\ A = \{a\})$ **by** *blast*

lemma *infinite-insert*: *infinite* (*insert a A*) = *infinite A*
by *simp*
lemma *infinite-Diff-insert*: *infinite* $(A - \mathit{insert}\ a\ B) = \mathit{infinite}\ (A - B)$
by *simp*

lemma *subset-finite-imp-finite*: ⟦ *finite* A; $B \subseteq A$ ⟧ ⟹ *finite* B
by (*rule finite-subset*)
lemma *infinite-not-subset-finite*: ⟦ *infinite* A; *finite* B ⟧ ⟹ ¬ $A \subseteq B$
by (*blast intro*: *subset-finite-imp-finite*)

thm *Un-infinite*
lemma *Un-infinite-right*: *infinite* T ⟹ *infinite* $(S \cup T)$ **by** *blast*
lemma *Un-infinite-iff*: *infinite* $(S \cup T)$ = (*infinite* S ∨ *infinite* T) **by** *blast*

thm *transfer-nat-int-set-relations*

Give own name to the lemma about finiteness of the integer image of a nat set

corollary *finite-A-int-A-conv*: *finite* A = *finite* (*int* ‘ A)
by (*rule transfer-nat-int-set-relations*)

Corresponding fact fo infinite sets

corollary *infinite-A-int-A-conv*: *infinite* A = *infinite* (*int* ‘ A)
by (*simp only*: *finite-A-int-A-conv*)

lemma *cartesian-product-infiniteL-imp-infinite*: ⟦ *infinite* A; $B \neq \{\}$ ⟧ ⟹ *infinite* $(A \times B)$
by (*blast dest*: *finite-cartesian-productD1*)
lemma *cartesian-product-infiniteR-imp-infinite*: ⟦ *infinite* B; $A \neq \{\}$ ⟧ ⟹ *infinite* $(A \times B)$
by (*blast dest*: *finite-cartesian-productD2*)

thm *finite-nat-iff-bounded*
lemma *finite-nat-iff-bounded2*:
  finite S = $(\exists (k::nat).\forall n{\in}S.\ n < k)$
by (*simp only*: *finite-nat-iff-bounded*, *blast*)
thm *finite-nat-iff-bounded-le*
lemma *finite-nat-iff-bounded-le2*:
  finite S = $(\exists (k::nat).\forall n{\in}S.\ n \leq k)$
by (*simp only*: *finite-nat-iff-bounded-le*, *blast*)

lemma *nat-asc-chain-imp-unbounded*:
  ⟦ $S \neq \{\}$; $(\forall m{\in}S.\ \exists n{\in}S.\ m < (n::nat))$ ⟧ ⟹ $\forall m.\ \exists n{\in}S.\ m \leq n$
apply (*rule allI*)
apply (*induct-tac* m)
 apply *blast*
apply (*erule bexE*)
apply (*rename-tac n1*)
apply (*erule-tac* x=*n1* **in** *ballE*)
  prefer *2*
  apply *simp*
apply (*clarify*, *rename-tac* x, *rule-tac* x=x **in** *bexI*)

apply *simp-all*
done

thm *infinite-nat-iff-unbounded-le*
lemma *infinite-nat-iff-asc-chain*:
  $S \neq \{\} \Longrightarrow$ *infinite* $S = (\forall\, m{\in}S.\ \exists\, n{\in}S.\ m < (n{::}nat))$
by (*metis Max-in infinite-nat-iff-unbounded not-greater-Max*)
lemma *infinite-imp-asc-chain*: *infinite* $S \Longrightarrow \forall\, m{\in}S.\ \exists\, n{\in}S.\ m < (n{::}nat)$
thm *infinite-nat-iff-asc-chain*[*THEN iffD1*, *OF infinite-imp-nonempty*]
by (*rule infinite-nat-iff-asc-chain*[*THEN iffD1*, *OF infinite-imp-nonempty*])

lemma *infinite-image-imp-infinite*: *infinite* $(f\ `\ A) \Longrightarrow$ *infinite* A
by *fastsimp*

lemma *inj-on-imp-infinite-image*: $[\![$ *infinite* A; *inj-on* $f\ A$ $]\!] \Longrightarrow$ *infinite* $(f\ `\ A)$
apply (*frule card-image*)
apply (*fastsimp simp*: *card-eq-0-iff*)
done

lemma *inj-on-infinite-image-iff*: *inj-on* $f\ A \Longrightarrow$ *infinite* $(f\ `\ A) =$ *infinite* A
apply (*rule iffI*)
 apply (*rule ccontr*, *simp*)
apply (*rule inj-on-imp-infinite-image*, *assumption*+)
done

lemma *inj-on-finite-image-iff*: *inj-on* $f\ A \Longrightarrow$ *finite* $(f\ `\ A) =$ *finite* A
by (*drule inj-on-infinite-image-iff*, *simp*)

lemma *nat-ex-greater-finite-Max-conv*:
  $A \neq \{\} \Longrightarrow (\exists\, x{\in}A.\ (n{::}nat) < x) = (\mathit{finite}\ A \longrightarrow n < \mathit{Max}\ A)$
apply (*rule iffI*)
 apply (*blast intro*: *order-less-le-trans Max-ge*)
apply (*case-tac finite A*)
 apply (*blast intro*: *Max-in*)
thm *infinite-nat-iff-unbounded*[*THEN iffD1*, *rule-format*]
apply (*drule infinite-nat-iff-unbounded*[*THEN iffD1*, *rule-format*, *of* - *n*])
apply *blast*
done
corollary *nat-ex-greater-infinite-finite-Max-conv'*:
  $(\exists\, x{\in}A.\ (n{::}nat) < x) = (\mathit{finite}\ A \wedge A \neq \{\} \wedge n < \mathit{Max}\ A \vee \mathit{infinite}\ A)$
apply (*case-tac* $A = \{\}$, *blast*)
apply (*drule nat-ex-greater-finite-Max-conv*[*of* - *n*])
apply *blast*
done

6.3.4 Some auxiliary results for disjoint sets

thm *Set.disjoint-iff-not-equal*
lemma *disjoint-iff-in-not-in1*: $(A \cap B = \{\}) = (\forall x{\in}A.\ x \notin B)$ **by** *blast*
lemma *disjoint-iff-in-not-in2*: $(A \cap B = \{\}) = (\forall x{\in}B.\ x \notin A)$ **by** *blast*

lemma *disjoint-in-Un*:
$[\![A \cap B = \{\};\ x \in A \cup B]\!] \Longrightarrow x \notin A \vee x \notin B$
thm *disjoint-iff-in-not-in1*[*THEN iffD1*, *rule-format*]
by (*blast intro*: *disjoint-iff-in-not-in1*[*THEN iffD1*])+

lemma *partition-Union*: $A \subseteq \bigcup C \Longrightarrow (\bigcup c{\in}C.\ A \cap c) = A$ **by** *blast*

lemma *disjoint-partition-Int*:
$\forall\, c1{\in}C.\ \forall\, c2{\in}C.\ c1 \neq c2 \longrightarrow c1 \cap c2 = \{\} \Longrightarrow$
$\forall\, a1{\in}\{A \cap c|\ c.\ c \in C\}.\ \forall\, a2{\in}\{A \cap c|\ c.\ c \in C\}.$
$a1 \neq a2 \longrightarrow a1 \cap a2 = \{\}$
by *blast*
thm
image-Union
image-eq-UN
image-def
image-Collect
lemma $\{f\ x\ |x.\ x \in A\} = (\bigcup x{\in}A.\ \{f\ x\})$
by *fastsimp*

thm *Finite-Set.card-partition*

This lemma version drops the superfluous precondition *finite* $(\bigcup C)$ (and turns the resulting equation to the sensible order *card* .. $= k * card$ ..).

lemma *card-partition*:
$[\![$ *finite* C; $\bigwedge c.\ c \in C \Longrightarrow card\ c = k$; $\bigwedge c1\ c2.\ [\![c1 \in C;\ c2 \in C;\ c1 \neq c2]\!] \Longrightarrow c1 \cap c2 = \{\}\ ]\!] \Longrightarrow$
card $(\bigcup C) = k * card\ C$
by (*metis card-infinite card-partition finite-Union mult-eq-if*)

6.3.5 Some auxiliary results for subset relation

thm *bex-subset-imp-bex*
thm *bex-imp-ex*
thm
ball-subset-imp-ball
ball-subset-imp-ball[*rule-format*]
thm
all-imp-ball
all-imp-ball[*rule-format*]

thm *image-mono*

lemma *subset-image-Int*: $A \subseteq B \Longrightarrow f\ `\ (A \cap B) = f\ `\ A \cap f\ `\ B$
by (*simp only*: *Int-absorb2 image-mono*)

lemma *image-diff-disjoint-image-Int*:
$[\![f\ `\ (A - B) \cap f\ `\ B = \{\}]\!] \Longrightarrow$
$f\ `\ (A \cap B) = f\ `\ A \cap f\ `\ B$
apply (*rule set-eqI*)
apply (*simp add*: *image-iff*)
apply *blast*
done

lemma *subset-imp-Int-subset1*: $A \subseteq C \Longrightarrow A \cap B \subseteq C$
thm *subset-trans*[*of* - $C \cap B$]
thm *subset-trans*[*of* - $C \cap B$, *OF Int-mono*]
thm *subset-trans*[*of* - $C \cap B$, *OF Int-mono*, *OF* - *subset-refl Int-lower1*]
by (*rule subset-trans*[*of* - $C \cap B$, *OF Int-mono*, *OF* - *subset-refl Int-lower1*])
lemma *subset-imp-Int-subset2*: $B \subseteq C \Longrightarrow A \cap B \subseteq C$
by (*simp only*: *Int-commute*[*of A*], *rule subset-imp-Int-subset1*)

6.3.6 Auxiliary results for intervals from *SetInterval*

lemmas *set-interval-defs* =
lessThan-def atMost-def
greaterThan-def atLeast-def
greaterThanLessThan-def atLeastLessThan-def
greaterThanAtMost-def atLeastAtMost-def
thm *set-interval-defs*

thm *image-add-atLeastAtMost*
lemma *image-add-atLeast*:
$(\lambda n{::}nat.\ n{+}k)\ `\ \{i..\} = \{i{+}k..\}$ (**is** *?A* = *?B*)
proof
show *?A* $\subseteq$ *?B* **by** *fastsimp*
next
show *?B* $\subseteq$ *?A*
proof
fix n **assume** *a*: n : *?B*
hence $n - k \in \{i..\}$ **by** *simp*
moreover have $n = (n - k) + k$ **using** *a* **by** *fastsimp*
ultimately show $n \in$ *?A* **by** *blast*
qed
qed

lemma *image-add-atMost*:
$(\lambda n{::}nat.\ n{+}k)\ `\ \{..i\} = \{k..i{+}k\}$ (**is** *?A* = *?B*)
proof −
have *s1*: $\{..i\} = \{0..i\}$
by (*simp add*: *set-interval-defs*)

show *?A = ?B*
by (*simp add*: *s1 image-add-atLeastAtMost*)
qed

thm *image-add-atLeastAtMost*
thm *image-add-atLeast*
thm *image-add-atMost*

thm *image-Suc-atLeastAtMost*

thm *image-add-atLeast*
corollary *image-Suc-atLeast*: *Suc ‘ {i..} = {Suc i..}*
by (*insert image-add-atLeast*[*of Suc 0*], *simp*)

thm *image-add-atMost*
corollary *image-Suc-atMost*: *Suc ‘ {..i} = {Suc 0..Suc i}*
thm *image-add-atMost*[*of Suc 0*]
by (*insert image-add-atMost*[*of Suc 0*], *simp*)

lemmas *image-add-lemmas* =
image-add-atLeastAtMost
image-add-atLeast
image-add-atMost
lemmas *image-Suc-lemmas* =
image-Suc-atLeastAtMost
image-Suc-atLeast
image-Suc-atMost

lemma *atMost-atLeastAtMost-0-conv*: $\{..i::nat\} = \{0..i\}$
by (*simp add*: *set-interval-defs*)

lemma *atLeastAtMost-subset-atMost*: $(k::'a::order) \leq k' \Longrightarrow \{l..k\} \subseteq \{..k'\}$
by (*clarsimp*, *blast intro*: *order-trans*)

thm *image-add-lemmas*
thm *image-Suc-lemmas*
thm *atMost-atLeastAtMost-0-conv*

lemma *lessThan-insert*: *insert* $(n::'a::order)$ $\{..<n\} = \{..n\}$
apply (*rule set-eqI*)
apply (*clarsimp simp add*: *order-le-less*)
apply *blast*
done
lemma *greaterThan-insert*: *insert* $(n::'a::order)$ $\{n<..\} = \{n..\}$
apply (*rule set-eqI*)
apply (*clarsimp simp add*: *order-le-less*)
apply *blast*
done

lemma *atMost-remove*: $\{..n\} - \{(n::'a::order)\} = \{..<n\}$
apply (*simp only*: *lessThan-insert*[*symmetric*])
apply (*rule Diff-insert-absorb*)
apply *simp*
done
lemma *atLeast-remove*: $\{n..\} - \{(n::'a::order)\} = \{n<..\}$
apply (*simp only*: *greaterThan-insert*[*symmetric*])
apply (*rule Diff-insert-absorb*)
apply *simp*
done

thm *atMost-def lessThan-def*
lemma *atMost-lessThan-conv*: $\{..n\} = \{..<Suc\ n\}$
by (*simp only*: *atMost-def lessThan-def less-Suc-eq-le*)

thm *atLeastAtMost-def atLeastLessThan-def*
lemma *atLeastAtMost-atLeastLessThan-conv*: $\{l..u\} = \{l..<Suc\ u\}$
by (*simp only*: *atLeastAtMost-def atLeastLessThan-def atMost-lessThan-conv*)

lemma *atLeast-greaterThan-conv*: $\{Suc\ n..\} = \{n<..\}$
by (*simp only*: *atLeast-def greaterThan-def Suc-le-eq*)

lemma *atLeastAtMost-greaterThanAtMost-conv*: $\{Suc\ l..u\} = \{l<..u\}$
by (*simp only*: *greaterThanAtMost-def atLeastAtMost-def atLeast-greaterThan-conv*)

lemma *finite-subset-atLeastAtMost*: *finite* $A \Longrightarrow A \subseteq \{Min\ A..Max\ A\}$
by (*simp add*: *subset-eq*)

lemma *Max-le-imp-subset-atMost*:
$[\![$ *finite* A; $Max\ A \leq n$ $]\!] \Longrightarrow A \subseteq \{..n\}$
thm *subset-trans*[*OF finite-subset-atLeastAtMost atLeastAtMost-subset-atMost*]
by (*rule subset-trans*[*OF finite-subset-atLeastAtMost atLeastAtMost-subset-atMost*])

lemma *subset-atMost-imp-Max-le*:
$[\![$ *finite* A; $A \neq \{\}$; $A \subseteq \{..n\}$ $]\!] \Longrightarrow Max\ A \leq n$
by (*simp add*: *subset-iff*)
lemma *subset-atMost-Max-le-conv*:
$[\![$ *finite* A; $A \neq \{\}$ $]\!] \Longrightarrow (A \subseteq \{..n\}) = (Max\ A \leq n)$
apply (*rule iffI*)
apply (*blast intro*: *subset-atMost-imp-Max-le*)
apply (*rule Max-le-imp-subset-atMost*, *assumption*+)
done

lemma *iMin-atLeast*: $iMin\ \{n..\} = n$
by (*rule iMin-equality*, *simp-all*)
lemma *iMin-greaterThan*: $iMin\ \{n<..\} = Suc\ n$
by (*simp only*: *atLeast-Suc-greaterThan*[*symmetric*] *iMin-atLeast*)
lemma *iMin-atMost*: $iMin\ \{..(n::nat)\} = 0$
by (*rule iMin-equality*, *simp-all*)
lemma *iMin-lessThan*: $0 < n \Longrightarrow iMin\ \{..<(n::nat)\} = 0$
by (*rule iMin-equality*, *simp-all*)

lemma *Max-atMost*: $Max\ \{..(n::nat)\} = n$
by (*rule Max-equality*[*OF - finite-atMost*], *simp-all*)
lemma *Max-lessThan*: $0 < n \Longrightarrow Max\ \{..<n\} = n - Suc\ 0$
by (*rule Max-equality*[*OF - finite-lessThan*], *simp-all*)
lemma *iMin-atLeastLessThan*: $m < n \Longrightarrow iMin\ \{m..<n\} = m$
by (*rule iMin-equality*, *simp-all*)
lemma *Max-atLeastLessThan*: $m < n \Longrightarrow Max\ \{m..<n\} = n - Suc\ 0$
by (*rule Max-equality*[*OF - finite-atLeastLessThan*], *simp-all add*: *less-imp-le-pred*)
lemma *iMin-greaterThanLessThan*: $Suc\ m < n \Longrightarrow iMin\ \{m<..<n\} = Suc\ m$
by (*simp only*: *atLeastSucLessThan-greaterThanLessThan*[*symmetric*] *iMin-atLeastLessThan*)
lemma *Max-greaterThanLessThan*: $Suc\ m < n \Longrightarrow Max\ \{m<..<n\} = n - Suc\ 0$
by (*simp only*: *atLeastSucLessThan-greaterThanLessThan*[*symmetric*] *Max-atLeastLessThan*)
lemma *iMin-greaterThanAtMost*: $m < n \Longrightarrow iMin\ \{m<..n\} = Suc\ m$
by (*simp only*: *atLeastSucAtMost-greaterThanAtMost*[*symmetric*] *atLeastLessThanSuc-atLeastAtMost*[*symmet*
iMin-atLeastLessThan)
lemma *Max-greaterThanAtMost*: $m < n \Longrightarrow Max\ \{m<..(n::nat)\} = n$
by (*simp add*: *atLeastSucAtMost-greaterThanAtMost*[*symmetric*] *atLeastLessThanSuc-atLeastAtMost*[*symmetr*
Max-atLeastLessThan)
lemma *iMin-atLeastAtMost*: $m \leq n \Longrightarrow iMin\ \{m..n\} = m$
by (*rule iMin-equality*, *simp-all*)
lemma *Max-atLeastAtMost*: $m \leq n \Longrightarrow Max\ \{m..(n::nat)\} = n$
by (*rule Max-equality*[*OF - finite-atLeastAtMost*], *simp-all*)

lemma *infinite-atLeast*: *infinite* $\{(n::nat)..\}$
by (*rule unbounded-k-infinite*[*of n*], *fastsimp*)
lemma *infinite-greaterThan*: *infinite* $\{(n::nat)<..\}$
by (*simp add*: *atLeast-Suc-greaterThan*[*symmetric*] *infinite-atLeast*)

lemma *infinite-atLeast-int*: *infinite* $\{(n::int)..\}$
apply (*rule-tac f*=$\lambda x.\ nat\ (x - n)$ **in** *inj-on-infinite-image-iff*[*THEN iffD1*, *rule-format*])
apply (*fastsimp simp*: *inj-on-def*)
apply (*rule-tac t*=$((\lambda x.\ nat\ (x - n))\ `\ \{n..\})$ **and** *s*=$\{0..\}$ **in** *subst*)
apply (*simp add*: *set-eq-iff image-iff Bex-def*)
apply (*clarify*, *rename-tac n1*)
apply (*rule-tac x*=$n + int\ n1$ **in** *exI*)
apply *simp-all*
done

lemma *infinite-greaterThan-int*: *infinite* $\{(n::int)<..\}$
thm *atLeast-remove*

apply (*simp only*: *atLeast-remove*[*symmetric*])
apply (*rule Diff-infinite-finite*[*OF singleton-finite*])
apply (*rule infinite-atLeast-int*)
done

lemma *infinite-atMost-int*: *infinite* {..(*n*::*int*)}
apply (*rule-tac f*=λ*x*. *n* − *x* **in** *inj-on-infinite-image-iff* [*THEN iffD1*, *rule-format*])
apply (*simp add*: *inj-on-def*)
apply (*rule-tac t*=(*op* − *n* ‘ {..*n*}) **and** *s*={*0*..} **in** *subst*)
apply (*simp add*: *set-eq-iff image-iff Bex-def*)
apply (*rule allI*, *rename-tac n1*)
apply (*rule iffI*)
apply (*rule-tac x*=*n* − *n1* **in** *exI*, *simp*)
apply *fastsimp*
apply (*rule infinite-atLeast-int*)
done

lemma *infinite-lessThan-int*: *infinite* {..<(*n*::*int*)}
thm *atMost-remove*
apply (*simp only*: *atMost-remove*[*symmetric*])
apply (*rule Diff-infinite-finite*[*OF singleton-finite*])
apply (*rule infinite-atMost-int*)
done

6.3.7 Auxiliary results for *card*

lemma *setsum-singleton*: (∑ *x*∈{*a*}. *f x*) = *f a*
by *simp*

lemma *card-singleton*: *card* {*a*} = *Suc 0*
by *simp*

thm *card-cartesian-product-singleton*
lemma *card-cartesian-product-singleton-right*: *card* (*A* × {*x*}) = *card A*
by (*simp add*: *card-cartesian-product*)

lemma *card-1-imp-singleton*: *card A* = *Suc 0* ⟹ (∃ *a*. *A* = {*a*})
by (*metis card-eq-SucD*)

lemma *card-1-singleton-conv*: (*card A* = *Suc 0*) = (∃ *a*. *A* = {*a*})
apply (*rule iffI*)
apply (*simp add*: *card-1-imp-singleton*)
apply *fastsimp*
done

lemma *card-gr0-imp-finite*: *0* < *card A* ⟹ *finite A*
by (*rule ccontr*, *simp*)
lemma *card-gr0-imp-not-empty*: (*0* < *card A*) ⟹ *A* ≠ {}
by (*rule ccontr*, *simp*)

lemma *not-empty-card-gr0-conv*: *finite A* $\Longrightarrow$ $(A \neq \{\}) = (0 < card\ A)$
by *fastsimp*

lemma *nat-card-le-Max*: *card* $(A::nat\ set) \leq Suc\ (Max\ A)$
apply (*case-tac finite A*)
prefer *2*
apply *simp*
thm *card-mono*[*OF finite-atMost*, *of A Max A*]
apply (*cut-tac card-mono*[*OF finite-atMost*, *of A Max A*])
apply *simp*
apply *fastsimp*
done

lemma *Int-card1*: *finite A* $\Longrightarrow$ *card* $(A \cap B) \leq card\ A$
by (*rule card-mono*, *simp-all*)
lemma *Int-card2*: *finite B* $\Longrightarrow$ *card* $(A \cap B) \leq card\ B$
by (*simp only*: *Int-commute*[*of A*], *rule Int-card1*)
lemma *Un-card1*: $[\![$ *finite A*; *finite B* $]\!] \Longrightarrow$ *card* $A \leq card\ (A \cup B)$
by (*rule card-mono*, *simp-all*)
lemma *Un-card2*: $[\![$ *finite A*; *finite B* $]\!] \Longrightarrow$ *card* $B \leq card\ (A \cup B)$
by (*simp only*: *Un-commute*[*of A*], *rule Un-card1*)

thm *Finite-Set.card-Un-Int*
lemma *card-Un-conv*:
$[\![$ *finite A*; *finite B* $]\!] \Longrightarrow$
card $(A \cup B) = card\ A + card\ B - card\ (A \cap B)$
by (*simp only*: *card-Un-Int diff-add-inverse2*)
lemma *card-Int-conv*:
$[\![$ *finite A*; *finite B* $]\!] \Longrightarrow$
card $(A \cap B) = card\ A + card\ B - card\ (A \cup B)$
by (*simp only*: *card-Un-Int diff-add-inverse*)

Pigeonhole principle, dirichlet's box principle

lemma *pigeonhole-principle*[*rule-format*]:
card $(f\ `\ A) < card\ A \longrightarrow (\exists\, x{\in}A.\ \exists\, y{\in}A.\ x \neq y \wedge f\ x = f\ y)$
apply (*case-tac finite A*)
prefer *2*
apply *simp*
thm *finite.induct*[*of A*]
apply (*rule finite.induct*[*of A*])
apply *simp-all*

apply (*clarsimp*, *rename-tac A1 a*)
apply (*case-tac a* $\in$ *A1*, *force simp*: *insert-absorb*)
apply (*case-tac f a* $\in$ *f* ` *A1*, *fastsimp*+)
done

thm *pigeonhole-principle*
corollary *pigeonhole-principle-linorder*[*rule-format*]:

card (*f* ‘ *A*) < *card* (*A*::*'a*::*linorder set*) ⟹ (∃ *x*∈*A*. ∃ *y*∈*A*. *x* < *y* ∧ *f x* = *f y*)
apply (*drule pigeonhole-principle*, *clarify*)
apply (*drule neq-iff*[*THEN iffD1*])
apply *fastsimp*
done

corollary *pigeonhole-mod*:
⟦ *0* < *m*; *m* < *card A* ⟧ ⟹ ∃ *x*∈*A*. ∃ *y*∈*A*. *x* < *y* ∧ *x mod m* = *y mod m*
apply (*rule pigeonhole-principle-linorder*)
apply (*rule le-less-trans*[*of* - *card* {..<*m*}])
apply (*rule card-mono*)
apply *fastsimp*+
done
corollary *pigeonhole-mod2*:
⟦ (*0*::*nat*) < *m*; *m* ≤ *c*; *inj-on f* {..*c*} ⟧ ⟹ ∃ *x*≤*c*. ∃ *y*≤*c*. *x* < *y* ∧ *f x mod m* = *f y mod m*
thm *pigeonhole-mod*[*of m f* ‘ {..*c*}]
apply (*insert pigeonhole-mod*[*of m f* ‘ {..*c*}])
apply (*clarsimp simp add*: *card-image*, *rename-tac x y*)
apply (*subgoal-tac x* ≠ *y*)
prefer *2*
apply *blast*
apply (*drule neq-iff*[*THEN iffD1*], *safe*)
apply *blast*
apply (*blast intro*: *eq-commute*[*THEN iffD1*])
done

end

7 SetIntervalCut: Cutting linearly ordered and natural sets

theory *SetIntervalCut*
imports *SetInterval2*
begin

7.1 Set restriction

A set to set function f is a *set restriction*, if there exists a predicate P, so that for every set s the function result $f\,s$ contains all its elements fulfilling P

definition
set-restriction :: (*'a set* ⇒ *'a set*) ⇒ *bool*
where
set-restriction f ≡ ∃ *P*. ∀ *A*. *f A* = {*x* ∈ *A*. *P x*}

lemma *set-restrictionD*: *set-restriction f* ⟹ ∃ *P*. ∀ *A*. *f A* = {*x* ∈ *A*. *P x*}

unfolding *set-restriction-def* **by** *blast*
lemma *set-restrictionD-spec*: *set-restriction f* $\Longrightarrow \exists P.\ f\ A = \{x \in A.\ P\ x\}$
unfolding *set-restriction-def* **by** *blast*
lemma *set-restrictionI*: $f = (\lambda A.\ \{x \in A.\ P\ x\}) \Longrightarrow$ *set-restriction f*
unfolding *set-restriction-def* **by** *blast*

lemma *set-restriction-comp*:
$[\![$ *set-restriction f*; *set-restriction g* $]\!] \Longrightarrow$ *set-restriction* $(f \circ g)$
apply (*unfold set-restriction-def*)
apply (*elim exE*, *rename-tac P1 P2*)
apply (*rule-tac* $x{=}\lambda x.\ P1\ x \wedge P2\ x$ **in** *exI*)
apply *fastsimp*
done
lemma *set-restriction-commute*:
$[\![$ *set-restriction f*; *set-restriction g* $]\!] \Longrightarrow f\ (g\ I) = g\ (f\ I)$
unfolding *set-restriction-def* **by** *fastsimp*

Constructs a set restriction function with the given restriction predicate

definition
set-restriction-fun :: $('a \Rightarrow bool) \Rightarrow ('a\ set \Rightarrow 'a\ set)$
where
set-restriction-fun $P \equiv \lambda A.\ \{x \in A.\ P\ x\}$

lemma *set-restriction-fun-is-set-restriction*:
set-restriction (*set-restriction-fun P*)
unfolding *set-restriction-def set-restriction-fun-def* **by** *blast*

lemma *set-restriction-Int-conv*:
set-restriction f $= (\exists B.\ \forall A.\ f\ A = A \cap B)$
apply (*unfold set-restriction-def*)
apply (*rule iffI*)
apply (*erule exE*, *rule-tac* $x{=}Collect\ P$ **in** *exI*, *blast*)
apply (*erule exE*, *rule-tac* $x{=}\lambda x.\ x \in B$ **in** *exI*, *blast*)
done

lemma *set-restriction-Un*:
set-restriction f $\Longrightarrow f\ (A \cup B) = f\ A \cup f\ B$
unfolding *set-restriction-def* **by** *fastsimp*
lemma *set-restriction-Int*:
set-restriction f $\Longrightarrow f\ (A \cap B) = f\ A \cap f\ B$
unfolding *set-restriction-def* **by** *fastsimp*
lemma *set-restriction-Diff*:
set-restriction f $\Longrightarrow f\ (A - B) = f\ A - f\ B$
unfolding *set-restriction-def* **by** *fastsimp*
lemma *set-restriction-mono*:
$[\![$ *set-restriction f*; $A \subseteq B$ $]\!] \Longrightarrow f\ A \subseteq f\ B$
unfolding *set-restriction-def* **by** *fastsimp*
lemma *set-restriction-absorb*:
set-restriction f $\Longrightarrow f\ (f\ A) = f\ A$

unfolding *set-restriction-def* **by** *fastsimp*
lemma *set-restriction-empty*:
  set-restriction f $\Longrightarrow$ *f* {} = {}
unfolding *set-restriction-def* **by** *blast*
lemma *set-restriction-non-empty-imp*:
  ⟦ *set-restriction f*; *f A* ≠ {} ⟧ $\Longrightarrow$ *A* ≠ {}
unfolding *set-restriction-def* **by** *blast*
lemma *set-restriction-subset*:
  set-restriction f $\Longrightarrow$ *f A* ⊆ *A*
unfolding *set-restriction-def* **by** *blast*
lemma *set-restriction-finite*:
  ⟦ *set-restriction f*; *finite A* ⟧ $\Longrightarrow$ *finite* (*f A*)
unfolding *set-restriction-def* **by** *fastsimp*
lemma *set-restriction-card*:
  ⟦ *set-restriction f*; *finite A* ⟧ $\Longrightarrow$
  card (*f A*) = *card A* − *card* {*a* ∈ *A*. *f* {*a*} = {}}
apply (*unfold set-restriction-def*)
thm *card-Diff-subset*[*symmetric*]
apply (*subgoal-tac* {*a* ∈ *A*. *f* {*a*} = {}} ⊆ *A*)
 prefer *2*
 apply *blast*
apply (*frule finite-subset*, *simp*)
thm *card-Diff-subset*[*symmetric*]
apply (*simp only*: *card-Diff-subset*[*symmetric*])
apply (*rule arg-cong*[**where** *f*=*card*])
apply *fastsimp*
done
lemma *set-restriction-card-le*:
  ⟦ *set-restriction f*; *finite A* ⟧ $\Longrightarrow$ *card* (*f A*) ≤ *card A*
by (*simp add*: *set-restriction-card*)

lemma *set-restriction-not-in-imp*:
  ⟦ *set-restriction f*; *x* ∉ *A* ⟧ $\Longrightarrow$ *x* ∉ *f A*
unfolding *set-restriction-def* **by** *blast*
lemma *set-restriction-in-imp*:
  ⟦ *set-restriction f*; *x* ∈ *f A* ⟧ $\Longrightarrow$ *x* ∈ *A*
unfolding *set-restriction-def* **by** *blast*

lemma *set-restriction-fun-singleton*:
  set-restriction-fun P {*a*} = (*if P a then* {*a*} *else* {})
unfolding *set-restriction-fun-def* **by** *force*
lemma *set-restriction-fun-all-conv*:
  ((*set-restriction-fun P*) *A* = *A*) = (∀ *x*∈*A*. *P x*)
unfolding *set-restriction-fun-def* **by** *blast*
lemma *set-restriction-fun-empty-conv*:
  ((*set-restriction-fun P*) *A* = {}) = (∀ *x*∈*A*. ¬ *P x*)
unfolding *set-restriction-fun-def* **by** *blast*

7.2 Cut operators for sets/intervals

7.2.1 Definitions and basic lemmata for cut operators

definition
  cut-le :: *'a::linorder set* ⇒ *'a* ⇒ *'a set* (**infixl** ↓≤ *100*)
where
  $I \downarrow\leq t \equiv \{ x \in I.\ x \leq t \}$
definition
  cut-less :: *'a::linorder set* ⇒ *'a* ⇒ *'a set* (**infixl** ↓< *100*)
where
  $I \downarrow< t \equiv \{ x \in I.\ x < t \}$

definition
  cut-ge :: *'a::linorder set* ⇒ *'a* ⇒ *'a set* (**infixl** ↓≥ *100*)
where
  $I \downarrow\geq t \equiv \{ x \in I.\ t \leq x \}$
definition
  cut-greater :: *'a::linorder set* ⇒ *'a* ⇒ *'a set* (**infixl** ↓> *100*)
where
  $I \downarrow> t \equiv \{ x \in I.\ t < x \}$

lemmas *i-cut-defs* =
  cut-le-def cut-less-def
  cut-ge-def cut-greater-def
thm *i-cut-defs*

lemma *cut-le-mem-iff*: $x \in I \downarrow\leq t = (x \in I \wedge x \leq t)$
by (*unfold cut-le-def*, *blast*)
lemma *cut-less-mem-iff*: $x \in I \downarrow< t = (x \in I \wedge x < t)$
by (*unfold cut-less-def*, *blast*)
lemma *cut-ge-mem-iff*: $x \in I \downarrow\geq t = (x \in I \wedge t \leq x)$
by (*unfold cut-ge-def*, *blast*)
lemma *cut-greater-mem-iff*: $x \in I \downarrow> t = (x \in I \wedge t < x)$
by (*unfold cut-greater-def*, *blast*)

lemmas *i-cut-mem-iff* =
  cut-le-mem-iff cut-less-mem-iff
  cut-ge-mem-iff cut-greater-mem-iff
thm *i-cut-mem-iff*

lemma
  cut-leI [*intro*!]: $x \in I \Longrightarrow x \leq t \Longrightarrow x \in I \downarrow\leq t$ **and**
  cut-lessI [*intro*!]: $x \in I \Longrightarrow x < t \Longrightarrow x \in I \downarrow< t$ **and**
  cut-geI [*intro*!]: $x \in I \Longrightarrow x \geq t \Longrightarrow x \in I \downarrow\geq t$ **and**
  cut-greaterI [*intro*!]: $x \in I \Longrightarrow x > t \Longrightarrow x \in I \downarrow> t$
by (*simp-all add*: *i-cut-mem-iff*)

lemma
  cut-leE [*elim*!]: $x \in I \downarrow\leq t \Longrightarrow (x \in I \Longrightarrow x \leq t \Longrightarrow P) \Longrightarrow P$ **and**

cut-lessE [*elim*!]: $x \in I \downarrow< t \Longrightarrow (x \in I \Longrightarrow x < t \Longrightarrow P) \Longrightarrow P$ **and**
cut-geE [*elim*!]: $x \in I \downarrow\geq t \Longrightarrow (x \in I \Longrightarrow x \geq t \Longrightarrow P) \Longrightarrow P$ **and**
cut-greaterE [*elim*!]: $x \in I \downarrow> t \Longrightarrow (x \in I \Longrightarrow x > t \Longrightarrow P) \Longrightarrow P$
by (*simp-all add*: *i-cut-mem-iff*)

lemma
cut-less-bound: $n \in I \downarrow< t \Longrightarrow n < t$ **and**
cut-le-bound: $n \in I \downarrow\leq t \Longrightarrow n \leq t$ **and**
cut-greater-bound: $n \in i \downarrow> t \Longrightarrow t < n$ **and**
cut-ge-bound: $n \in i \downarrow\geq t \Longrightarrow t \leq n$
unfolding *i-cut-defs* **by** *simp-all*
lemmas *i-cut-bound* =
cut-less-bound cut-le-bound
cut-greater-bound cut-ge-bound

lemma
cut-le-Int-conv: $I \downarrow\leq t = I \cap \{..t\}$ **and**
cut-less-Int-conv: $I \downarrow< t = I \cap \{..<t\}$ **and**
cut-ge-Int-conv: $I \downarrow\geq t = I \cap \{t..\}$ **and**
cut-greater-Int-conv: $I \downarrow> t = I \cap \{t<..\}$
unfolding *i-cut-defs* **by** *blast+*

lemmas *i-cut-Int-conv* =
cut-le-Int-conv cut-less-Int-conv
cut-ge-Int-conv cut-greater-Int-conv
thm *i-cut-Int-conv*

lemma
cut-le-Diff-conv: $I \downarrow\leq t = I - \{t<..\}$ **and**
cut-less-Diff-conv: $I \downarrow< t = I - \{t..\}$ **and**
cut-ge-Diff-conv: $I \downarrow\geq t = I - \{..<t\}$ **and**
cut-greater-Diff-conv: $I \downarrow> t = I - \{..t\}$
by (*fastsimp simp*: *i-cut-defs*)+
lemmas *i-cut-Diff-conv* =
cut-le-Diff-conv cut-less-Diff-conv
cut-ge-Diff-conv cut-greater-Diff-conv
thm *i-cut-Diff-conv*

7.2.2 Basic results for cut operators

lemma
cut-less-eq-set-restriction-fun': $(\lambda I.\ I \downarrow< t) = \textit{set-restriction-fun}\ (\lambda x.\ x < t)$
and
cut-le-eq-set-restriction-fun': $(\lambda I.\ I \downarrow\leq t) = \textit{set-restriction-fun}\ (\lambda x.\ x \leq t)$
and
cut-greater-eq-set-restriction-fun': $(\lambda I.\ I \downarrow> t) = \textit{set-restriction-fun}\ (\lambda x.\ x > t)$
and

cut-ge-eq-set-restriction-fun': $(\lambda I.\ I \downarrow\geq t) = \textit{set-restriction-fun}\ (\lambda x.\ x \geq t)$
unfolding *set-restriction-fun-def i-cut-defs* **by** *blast+*
lemmas *i-cut-eq-set-restriction-fun' =*
cut-less-eq-set-restriction-fun' cut-le-eq-set-restriction-fun'
cut-greater-eq-set-restriction-fun' cut-ge-eq-set-restriction-fun'

lemma
cut-less-eq-set-restriction-fun: $I \downarrow< t = \textit{set-restriction-fun}\ (\lambda x.\ x < t)\ I$ **and**
cut-le-eq-set-restriction-fun: $I \downarrow\leq t = \textit{set-restriction-fun}\ (\lambda x.\ x \leq t)\ I$ **and**
cut-greater-eq-set-restriction-fun: $I \downarrow> t = \textit{set-restriction-fun}\ (\lambda x.\ x > t)\ I$ **and**
cut-ge-eq-set-restriction-fun: $I \downarrow\geq t = \textit{set-restriction-fun}\ (\lambda x.\ x \geq t)\ I$
by (*simp-all only*: *i-cut-eq-set-restriction-fun'*[*symmetric*])
lemmas *i-cut-eq-set-restriction-fun =*
cut-less-eq-set-restriction-fun cut-le-eq-set-restriction-fun
cut-greater-eq-set-restriction-fun cut-ge-eq-set-restriction-fun

lemma *i-cut-set-restriction-disj*:
$[\![\ \textit{cut-op} = \textit{op}\downarrow< \vee\ \textit{cut-op} = \textit{op}\downarrow\leq \vee$
$\textit{cut-op} = \textit{op}\downarrow> \vee\ \textit{cut-op} = \textit{op}\downarrow\geq;$
$f = (\lambda I.\ \textit{cut-op}\ I\ t)\ ]\!] \Longrightarrow \textit{set-restriction}\ f$
apply *safe*
apply (*simp-all only*: *i-cut-eq-set-restriction-fun set-restriction-fun-is-set-restriction*)
done
thm *set-restriction-def*
corollary
i-cut-less-set-restriction: $\textit{set-restriction}\ (\lambda I.\ I \downarrow< t)$ **and**
i-cut-le-set-restriction: $\textit{set-restriction}\ (\lambda I.\ I \downarrow\leq t)$ **and**
i-cut-greater-set-restriction: $\textit{set-restriction}\ (\lambda I.\ I \downarrow> t)$ **and**
i-cut-ge-set-restriction: $\textit{set-restriction}\ (\lambda I.\ I \downarrow\geq t)$
by (*simp-all only*: *i-cut-eq-set-restriction-fun set-restriction-fun-is-set-restriction*)
lemmas *i-cut-set-restriction =*
i-cut-le-set-restriction i-cut-less-set-restriction
i-cut-ge-set-restriction i-cut-greater-set-restriction
lemma *i-cut-commute-disj*: $[\![$
$\textit{cut-op1} = \textit{op}\downarrow< \vee\ \textit{cut-op1} = \textit{op}\downarrow\leq \vee$
$\textit{cut-op1} = \textit{op}\downarrow> \vee\ \textit{cut-op1} = \textit{op}\downarrow\geq;$
$\textit{cut-op2} = \textit{op}\downarrow< \vee\ \textit{cut-op2} = \textit{op}\downarrow\leq \vee$
$\textit{cut-op2} = \textit{op}\downarrow> \vee\ \textit{cut-op2} = \textit{op}\downarrow\geq\ ]\!] \Longrightarrow$
$\textit{cut-op2}\ (\textit{cut-op1}\ I\ t1)\ t2 = \textit{cut-op1}\ (\textit{cut-op2}\ I\ t2)\ t1$
thm
set-restriction-commute[*of* $\lambda I.\ \textit{cut-op1}\ I\ t$]
i-cut-set-restriction-disj
apply (*rule set-restriction-commute*)
apply (*simp-all only*: *i-cut-set-restriction-disj*)
done
thm *i-cut-commute-disj*
thm *i-cut-commute-disj*[*of op* $\downarrow<$ *op* $\downarrow<$, *simplified*]
thm *i-cut-commute-disj*[*of op* $\downarrow<$ *op* $\downarrow>$, *simplified*]
thm *i-cut-commute-disj*[*of op* $\downarrow\leq$ *op* $\downarrow>$, *simplified*]

thm *i-cut-commute-disj*[*of op* $\downarrow\geq$ *op* $\downarrow>$, *simplified*]

lemma
  cut-less-empty: $\{\} \downarrow< t = \{\}$ **and**
  cut-le-empty: $\{\} \downarrow\leq t = \{\}$ **and**
  cut-greater-empty: $\{\} \downarrow> t = \{\}$ **and**
  cut-ge-empty: $\{\} \downarrow\geq t = \{\}$
by *blast*+
lemmas *i-cut-empty* =
  cut-less-empty cut-le-empty
  cut-greater-empty cut-ge-empty
thm *i-cut-empty*

lemma
  cut-less-not-empty-imp: $I \downarrow< t \neq \{\} \Longrightarrow I \neq \{\}$ **and**
  cut-le-not-empty-imp: $I \downarrow\leq t \neq \{\} \Longrightarrow I \neq \{\}$ **and**
  cut-greater-not-empty-imp: $I \downarrow> t \neq \{\} \Longrightarrow I \neq \{\}$ **and**
  cut-ge-not-empty-imp: $I \downarrow\geq t \neq \{\} \Longrightarrow I \neq \{\}$
by *blast*+

lemma
  cut-less-singleton: $\{a\} \downarrow< t = (\textit{if } a < t \textit{ then } \{a\} \textit{ else } \{\})$ **and**
  cut-le-singleton: $\{a\} \downarrow\leq t = (\textit{if } a \leq t \textit{ then } \{a\} \textit{ else } \{\})$ **and**
  cut-greater-singleton: $\{a\} \downarrow> t = (\textit{if } a > t \textit{ then } \{a\} \textit{ else } \{\})$ **and**
  cut-ge-singleton: $\{a\} \downarrow\geq t = (\textit{if } a \geq t \textit{ then } \{a\} \textit{ else } \{\})$
by (*rule i-cut-eq-set-restriction-fun*[*THEN ssubst*], *simp only*: *set-restriction-fun-singleton*)+
lemmas *i-cut-singleton* =
  cut-le-singleton cut-less-singleton
  cut-ge-singleton cut-greater-singleton

lemma
  cut-less-subset: $I \downarrow< t \subseteq I$ **and**
  cut-le-subset: $I \downarrow\leq t \subseteq I$ **and**
  cut-greater-subset: $I \downarrow> t \subseteq I$ **and**
  cut-ge-subset: $I \downarrow\geq t \subseteq I$
thm *i-cut-set-restriction*[*THEN set-restriction-subset*]
by (*simp-all only*: *i-cut-set-restriction*[*THEN set-restriction-subset*])
lemmas *i-cut-subset* =
  cut-less-subset cut-le-subset
  cut-greater-subset cut-ge-subset
thm *i-cut-subset*

thm *set-restriction-Un*
lemma *i-cut-Un-disj*:
  $[\![$ *cut-op* = *op* $\downarrow< \vee$ *cut-op* = *op* $\downarrow\leq \vee$

$\quad cut\text{-}op = op \downarrow> \vee\ cut\text{-}op = op \downarrow\geq \rrbracket$
$\Longrightarrow cut\text{-}op\ (A \cup B)\ t = cut\text{-}op\ A\ t \cup cut\text{-}op\ B\ t$
thm *i-cut-set-restriction-disj*[*of cut-op* $\lambda I.$ *cut-op I t t*]
apply (*drule i-cut-set-restriction-disj*[**where** $f{=}\lambda I.$ *cut-op I t*], *simp*)
thm *set-restriction-Un*
by (*rule set-restriction-Un*)

corollary
cut-less-Un: $(A \cup B) \downarrow< t = A \downarrow< t \cup B \downarrow< t$ **and**
cut-le-Un: $(A \cup B) \downarrow\leq t = A \downarrow\leq t \cup B \downarrow\leq t$ **and**
cut-greater-Un: $(A \cup B) \downarrow> t = A \downarrow> t \cup B \downarrow> t$ **and**
cut-ge-Un: $(A \cup B) \downarrow\geq t = A \downarrow\geq t \cup B \downarrow\geq t$
by (*rule i-cut-Un-disj*, *blast*)+
lemmas *i-cut-Un* =
cut-less-Un cut-le-Un
cut-greater-Un cut-ge-Un

lemma *i-cut-Int-disj*:
$\llbracket\ cut\text{-}op = op \downarrow< \vee\ cut\text{-}op = op \downarrow\leq \vee$
$\quad cut\text{-}op = op \downarrow> \vee\ cut\text{-}op = op \downarrow\geq \rrbracket$
$\Longrightarrow cut\text{-}op\ (A \cap B)\ t = cut\text{-}op\ A\ t \cap cut\text{-}op\ B\ t$
apply (*drule i-cut-set-restriction-disj*[**where** $f{=}\lambda I.$ *cut-op I t*], *simp*)
by (*rule set-restriction-Int*)

lemma
cut-less-Int: $(A \cap B) \downarrow< t = A \downarrow< t \cap B \downarrow< t$ **and**
cut-le-Int: $(A \cap B) \downarrow\leq t = A \downarrow\leq t \cap B \downarrow\leq t$ **and**
cut-greater-Int: $(A \cap B) \downarrow> t = A \downarrow> t \cap B \downarrow> t$ **and**
cut-ge-Int: $(A \cap B) \downarrow\geq t = A \downarrow\geq t \cap B \downarrow\geq t$
by *blast*+
lemmas *i-cut-Int* =
cut-less-Int cut-le-Int
cut-greater-Int cut-ge-Int

lemma
cut-less-Int-left: $(A \cap B) \downarrow< t = A \downarrow< t \cap B$ **and**
cut-le-Int-left: $(A \cap B) \downarrow\leq t = A \downarrow\leq t \cap B$ **and**
cut-greater-Int-left: $(A \cap B) \downarrow> t = A \downarrow> t \cap B$ **and**
cut-ge-Int-left: $(A \cap B) \downarrow\geq t = A \downarrow\geq t \cap B$
by *blast*+
lemmas *i-cut-Int-left* =
cut-less-Int-left cut-le-Int-left
cut-greater-Int-left cut-ge-Int-left

lemma

cut-less-Int-right: $(A \cap B) \downarrow< t = A \cap B \downarrow< t$ **and**
cut-le-Int-right: $(A \cap B) \downarrow\leq t = A \cap B \downarrow\leq t$ **and**
cut-greater-Int-right: $(A \cap B) \downarrow> t = A \cap B \downarrow> t$ **and**
cut-ge-Int-right: $(A \cap B) \downarrow\geq t = A \cap B \downarrow\geq t$
by *blast+*
lemmas *i-cut-Int-right* =
cut-less-Int-right cut-le-Int-right
cut-greater-Int-right cut-ge-Int-right

lemma *i-cut-Diff-disj*:
$[\![$ *cut-op* $=$ *op* $\downarrow< \vee$ *cut-op* $=$ *op* $\downarrow\leq \vee$
cut-op $=$ *op* $\downarrow> \vee$ *cut-op* $=$ *op* $\downarrow\geq]\!]$
$\Longrightarrow$ *cut-op* $(A - B)$ t $=$ *cut-op* A $t -$ *cut-op* B t
apply (*drule i-cut-set-restriction-disj*[**where** $f{=}\lambda I.$ *cut-op* I t], *simp*)
by (*rule set-restriction-Diff*)
corollary
cut-less-Diff: $(A - B) \downarrow< t = A \downarrow< t - B \downarrow< t$ **and**
cut-le-Diff: $(A - B) \downarrow\leq t = A \downarrow\leq t - B \downarrow\leq t$ **and**
cut-greater-Diff: $(A - B) \downarrow> t = A \downarrow> t - B \downarrow> t$ **and**
cut-ge-Diff: $(A - B) \downarrow\geq t = A \downarrow\geq t - B \downarrow\geq t$
by (*rule i-cut-Diff-disj*, *blast*)+
lemmas *i-cut-Diff* =
cut-less-Diff cut-le-Diff
cut-greater-Diff cut-ge-Diff

thm *set-restriction-mono*
lemma *i-cut-subset-mono-disj*:
$[\![$ *cut-op* $=$ *op* $\downarrow< \vee$ *cut-op* $=$ *op* $\downarrow\leq \vee$
cut-op $=$ *op* $\downarrow> \vee$ *cut-op* $=$ *op* $\downarrow\geq$; $A \subseteq B]\!]$
$\Longrightarrow$ *cut-op* A $t \subseteq$ *cut-op* B t
apply (*drule i-cut-set-restriction-disj*[**where** $f{=}\lambda I.$ *cut-op* I t], *simp*)
thm *set-restriction-mono*[**where** $f{=}\lambda I.$ *cut-op* I t]
by (*rule set-restriction-mono*[**where** $f{=}\lambda I.$ *cut-op* I t])
corollary
cut-less-subset-mono: $A \subseteq B \Longrightarrow A \downarrow< t \subseteq B \downarrow< t$ **and**
cut-le-subset-mono: $A \subseteq B \Longrightarrow A \downarrow\leq t \subseteq B \downarrow\leq t$ **and**
cut-greater-subset-mono: $A \subseteq B \Longrightarrow A \downarrow> t \subseteq B \downarrow> t$ **and**
cut-ge-subset-mono: $A \subseteq B \Longrightarrow A \downarrow\geq t \subseteq B \downarrow\geq t$
by (*rule i-cut-subset-mono-disj*[*of* - A], *simp+*)+
lemmas *i-cut-subset-mono* =
cut-less-subset-mono cut-le-subset-mono
cut-greater-subset-mono cut-ge-subset-mono

lemma

cut-less-mono: $t \leq t' \Longrightarrow I \downarrow< t \subseteq I \downarrow< t'$ **and**
cut-greater-mono: $t' \leq t \Longrightarrow I \downarrow> t \subseteq I \downarrow> t'$ **and**
cut-le-mono: $t \leq t' \Longrightarrow I \downarrow\leq t \subseteq I \downarrow\leq t'$ **and**
cut-ge-mono: $t' \leq t \Longrightarrow I \downarrow\geq t \subseteq I \downarrow\geq t'$
by (*unfold i-cut-defs*, *safe*, *simp-all*)
lemmas *i-cut-mono* =
cut-le-mono cut-less-mono
cut-ge-mono cut-greater-mono

lemma
cut-cut-le: $i \downarrow\leq a \downarrow\leq b = i \downarrow\leq min\ a\ b$ **and**
cut-cut-less: $i \downarrow< a \downarrow< b = i \downarrow< min\ a\ b$ **and**
cut-cut-ge: $i \downarrow\geq a \downarrow\geq b = i \downarrow\geq max\ a\ b$ **and**
cut-cut-greater: $i \downarrow> a \downarrow> b = i \downarrow> max\ a\ b$
unfolding *i-cut-defs* **by** *simp-all*
lemmas *i-cut-cut* =
cut-cut-le cut-cut-less
cut-cut-ge cut-cut-greater

lemma *i-cut-absorb-disj*:
$[\![$ *cut-op* = *op* $\downarrow< \vee$ *cut-op* = *op* $\downarrow\leq \vee$
cut-op = *op* $\downarrow> \vee$ *cut-op* = *op* $\downarrow\geq]\!]$
$\Longrightarrow$ *cut-op* (*cut-op I t*) *t* = *cut-op I t*
thm *i-cut-set-restriction-disj*[**where** $f{=}\lambda I.$ *cut-op I t*]
apply (*drule i-cut-set-restriction-disj*[**where** $f{=}\lambda I.$ *cut-op I t*], *blast*)
thm *set-restriction-absorb*
apply (*blast dest*: *set-restriction-absorb*)
done
corollary
cut-le-absorb: $I \downarrow\leq t \downarrow\leq t = I \downarrow\leq t$ **and**
cut-less-absorb: $I \downarrow< t \downarrow< t = I \downarrow< t$ **and**
cut-ge-absorb: $I \downarrow\geq t \downarrow\geq t = I \downarrow\geq t$ **and**
cut-greater-absorb: $I \downarrow> t \downarrow> t = I \downarrow> t$
thm *i-cut-absorb-disj*
by (*rule i-cut-absorb-disj*, *blast*)+
lemmas *i-cut-absorb* =
cut-le-absorb cut-less-absorb
cut-ge-absorb cut-greater-absorb

lemma
cut-less-0-empty: $I \downarrow< (0{::}nat) = \{\}$ **and**
cut-ge-0-all: $I \downarrow\geq (0{::}nat) = I$
unfolding *i-cut-defs* **by** *blast*+

lemma
cut-le-all-iff: $(I \downarrow\leq t = I) = (\forall x{\in}I.\ x \leq t)$ **and**

cut-less-all-iff: $(I \downarrow< t = I) = (\forall x{\in}I.\ x < t)$ **and**
cut-ge-all-iff: $(I \downarrow\geq t = I) = (\forall x{\in}I.\ x \geq t)$ **and**
cut-greater-all-iff: $(I \downarrow> t = I) = (\forall x{\in}I.\ x > t)$
by *blast+*
lemmas *i-cut-all-iff* =
cut-le-all-iff cut-less-all-iff
cut-ge-all-iff cut-greater-all-iff

lemma
cut-le-empty-iff: $(I \downarrow\leq t = \{\}) = (\forall x{\in}I.\ t < x)$ **and**
cut-less-empty-iff: $(I \downarrow< t = \{\}) = (\forall x{\in}I.\ t \leq x)$ **and**
cut-ge-empty-iff: $(I \downarrow\geq t = \{\}) = (\forall x{\in}I.\ x < t)$ **and**
cut-greater-empty-iff: $(I \downarrow> t = \{\}) = (\forall x{\in}I.\ x \leq t)$
unfolding *i-cut-defs* **by** *fastsimp+*
lemmas *i-cut-empty-iff* =
cut-le-empty-iff cut-less-empty-iff
cut-ge-empty-iff cut-greater-empty-iff

lemma
cut-le-not-empty-iff: $(I \downarrow\leq t \neq \{\}) = (\exists x{\in}I.\ x \leq t)$ **and**
cut-less-not-empty-iff: $(I \downarrow< t \neq \{\}) = (\exists x{\in}I.\ x < t)$ **and**
cut-ge-not-empty-iff: $(I \downarrow\geq t \neq \{\}) = (\exists x{\in}I.\ t \leq x)$ **and**
cut-greater-not-empty-iff: $(I \downarrow> t \neq \{\}) = (\exists x{\in}I.\ t < x)$
unfolding *i-cut-defs* **by** *blast+*

lemmas *i-cut-not-empty-iff* =
cut-le-not-empty-iff cut-less-not-empty-iff
cut-ge-not-empty-iff cut-greater-not-empty-iff
thm *i-cut-not-empty-iff*

lemma *nat-cut-ge-infinite-not-empty*: *infinite* $I \Longrightarrow I \downarrow\geq (t{::}nat) \neq \{\}$
by (*drule infinite-nat-iff-unbounded-le*[*THEN iffD1*], *blast*)
lemma *nat-cut-greater-infinite-not-empty*: *infinite* $I \Longrightarrow I \downarrow> (t{::}nat) \neq \{\}$
by (*drule infinite-nat-iff-unbounded*[*THEN iffD1*], *blast*)

thm *set-restriction-not-in-imp*
corollary
cut-le-not-in-imp: $x \notin I \Longrightarrow x \notin I \downarrow\leq t$ **and**
cut-less-not-in-imp: $x \notin I \Longrightarrow x \notin I \downarrow< t$ **and**
cut-ge-not-in-imp: $x \notin I \Longrightarrow x \notin I \downarrow\geq t$ **and**
cut-greater-not-in-imp: $x \notin I \Longrightarrow x \notin I \downarrow> t$
thm *i-cut-set-restriction*[*THEN set-restriction-not-in-imp*]
by (*rule i-cut-set-restriction*[*THEN set-restriction-not-in-imp*], *assumption*)+
lemmas *i-cut-not-in-imp* =
cut-le-not-in-imp cut-less-not-in-imp
cut-ge-not-in-imp cut-greater-not-in-imp

thm *set-restriction-in-imp*
corollary
cut-le-in-imp: $x \in I \downarrow\leq t \Longrightarrow x \in I$ **and**
cut-less-in-imp: $x \in I \downarrow< t \Longrightarrow x \in I$ **and**
cut-ge-in-imp: $x \in I \downarrow\geq t \Longrightarrow x \in I$ **and**
cut-greater-in-imp: $x \in I \downarrow> t \Longrightarrow x \in I$
thm *i-cut-set-restriction*[*THEN set-restriction-in-imp*]
by (*rule i-cut-set-restriction*[*THEN set-restriction-in-imp*], *assumption*)+
lemmas *i-cut-in-imp* =
cut-le-in-imp cut-less-in-imp
cut-ge-in-imp cut-greater-in-imp

lemma *Collect-minI-cut*: $[\![k \in I; P\ (k::('a::wellorder))]\!] \Longrightarrow \exists x{\in}I.\ P\ x \wedge (\forall y{\in}(I \downarrow< x).\ \neg\ P\ y)$
by (*drule Collect-minI*, *assumption*, *blast*)
corollary *Collect-minI-ex-cut*: $\exists k{\in}I.\ P\ (k::('a::wellorder)) \Longrightarrow \exists x{\in}I.\ P\ x \wedge (\forall y{\in}(I \downarrow< x).\ \neg\ P\ y)$
by (*drule Collect-minI-ex*, *blast*)
corollary *Collect-minI-ex2-cut*: $\{k{\in}I.\ P\ (k::('a::wellorder))\} \neq \{\} \Longrightarrow \exists x{\in}I.\ P\ x \wedge (\forall y{\in}(I \downarrow< x).\ \neg\ P\ y)$
by (*drule Collect-minI-ex2*, *blast*)

lemma *cut-le-cut-greater-ident*: $t2 \leq t1 \Longrightarrow I \downarrow\leq t1 \cup I \downarrow> t2 = I$
by *fastsimp*
lemma *cut-less-cut-ge-ident*: $t2 \leq t1 \Longrightarrow I \downarrow< t1 \cup I \downarrow\geq t2 = I$
by *fastsimp*
lemma *cut-le-cut-ge-ident*: $t2 \leq t1 \Longrightarrow I \downarrow\leq t1 \cup I \downarrow\geq t2 = I$
by *fastsimp*

lemma *cut-less-cut-greater-ident*:
$[\![t2 \leq t1; I \cap \{t1..t2\} = \{\}]\!] \Longrightarrow I \downarrow< t1 \cup I \downarrow> t2 = I$
by *fastsimp*
corollary *cut-less-cut-greater-ident'*:
$t \notin I \Longrightarrow I \downarrow< t \cup I \downarrow> t = I$
by (*simp add*: *cut-less-cut-greater-ident*)

lemma *insert-eq-cut-less-cut-greater*: *insert n I* $= I \downarrow< n \cup \{n\} \cup I \downarrow> n$
by *fastsimp*

7.2.3 Relations between cut operators

lemma *insert-Int-conv-if*: $A \cap (insert\ x\ B) = (\ if\ x \in A\ then\ insert\ x\ (A \cap B)\ else\ A \cap B)$
by *simp*

lemma *cut-le-less-conv-if*: $I\ \downarrow\leq t = (\ if\ t \in I\ then\ insert\ t\ (I\ \downarrow< t)\ else\ (I\ \downarrow< t))$
by (*simp add*: *i-cut-Int-conv lessThan-insert*[*symmetric*] *insert-Int-conv-if*)

lemma *cut-le-less-conv*: $I\ \downarrow\leq t = (\{t\} \cap I) \cup (I\ \downarrow< t)$
by *fastsimp*

lemma *cut-less-le-conv*: $I\ \downarrow< t = (I\ \downarrow\leq t) - \{t\}$
by *fastsimp*
lemma *cut-less-le-conv-if*: $I\ \downarrow< t = (\ if\ t \in I\ then\ (I\ \downarrow\leq t) - \{t\}\ else\ (I\ \downarrow\leq t))$
by (*simp only*: *cut-less-le-conv*, *force*)

lemma *cut-ge-greater-conv-if*: $I\ \downarrow\geq t = (\ if\ t \in I\ then\ insert\ t\ (I\ \downarrow> t)\ else\ (I\ \downarrow> t))$
by (*simp add*: *i-cut-Int-conv greaterThan-insert*[*symmetric*] *insert-Int-conv-if*)
lemma *cut-ge-greater-conv*: $I\ \downarrow\geq t = (\{t\} \cap I) \cup (I\ \downarrow> t)$
apply (*simp only*: *cut-ge-greater-conv-if*)
apply (*case-tac* $t \in I$)
apply *simp-all*
done
lemma *cut-greater-ge-conv*: $I\ \downarrow> t = (I\ \downarrow\geq t) - \{t\}$
by *fastsimp*
lemma *cut-greater-ge-conv-if*: $I\ \downarrow> t = (\ if\ t \in I\ then\ (I\ \downarrow\geq t) - \{t\}\ else\ (I\ \downarrow\geq t))$
by (*simp only*: *cut-greater-ge-conv*, *force*)

lemma *nat-cut-le-less-conv*: $I\ \downarrow\leq t = I\ \downarrow< Suc\ t$
by *fastsimp*
lemma *nat-cut-less-le-conv*: $0 < t \Longrightarrow I\ \downarrow< t = I\ \downarrow\leq (t - Suc\ 0)$
by *fastsimp*
lemma *nat-cut-ge-greater-conv*: $I\ \downarrow\geq Suc\ t = I\ \downarrow> t$
by *fastsimp*
lemma *nat-cut-greater-ge-conv*: $0 < t \Longrightarrow I\ \downarrow> (t - Suc\ 0) = I\ \downarrow\geq t$
by *fastsimp*

7.2.4 Function images with cut operators

lemma *cut-less-image*:
⟦ *strict-mono-on f A*; $I \subseteq A$; $n \in A$ ⟧ ⟹
$(f \mathbin{`} I) \downarrow< f\ n = f \mathbin{`} (I \downarrow< n)$
apply (*rule set-eqI*)
apply (*simp add*: *image-iff Bex-def cut-less-mem-iff*)
apply (*unfold strict-mono-on-def*)
apply (*rule iffI*)
apply (*metis not-less-iff-gr-or-eq set-rev-mp*)
apply *blast*
done

lemma *cut-le-image*:
⟦ *strict-mono-on f A*; $I \subseteq A$; $n \in A$ ⟧ ⟹
$(f \mathbin{`} I) \downarrow\leq f\ n = f \mathbin{`} (I \downarrow\leq n)$
apply (*frule strict-mono-on-imp-inj-on*)
thm *cut-le-less-conv-if*
apply (*clarsimp simp*: *cut-le-less-conv-if cut-less-image inj-on-def*)
apply *blast*
done

lemma *cut-greater-image*:
⟦ *strict-mono-on f A*; $I \subseteq A$; $n \in A$ ⟧ ⟹
$(f \mathbin{`} I) \downarrow> f\ n = f \mathbin{`} (I \downarrow> n)$
apply (*rule set-eqI*)
apply (*simp add*: *image-iff Bex-def cut-greater-mem-iff*)
apply (*unfold strict-mono-on-def*)
apply (*rule iffI*)
apply (*metis not-less-iff-gr-or-eq set-rev-mp*)
apply *blast*
done

lemma *cut-ge-image*:
⟦ *strict-mono-on f A*; $I \subseteq A$; $n \in A$ ⟧ ⟹
$(f \mathbin{`} I) \downarrow\geq f\ n = f \mathbin{`} (I \downarrow\geq n)$
apply (*frule strict-mono-on-imp-inj-on*)
thm *cut-ge-greater-conv-if*
apply (*clarsimp simp*: *cut-ge-greater-conv-if cut-greater-image inj-on-def*)
apply *blast*
done

lemmas *i-cut-image* =
cut-le-image cut-less-image
cut-ge-image cut-greater-image
thm *i-cut-image*
thm *i-cut-image*[*OF - subset-refl*]

7.2.5 Finiteness and cardinality with cut operators

thm *set-restriction-finite*
lemma
  cut-le-finite: $\mathit{finite}\ I \Longrightarrow \mathit{finite}\ (I\ {\downarrow}{\leq}\ t)$ **and**
  cut-less-finite: $\mathit{finite}\ I \Longrightarrow \mathit{finite}\ (I\ {\downarrow}{<}\ t)$ **and**
  cut-ge-finite: $\mathit{finite}\ I \Longrightarrow \mathit{finite}\ (I\ {\downarrow}{\geq}\ t)$ **and**
  cut-greater-finite: $\mathit{finite}\ I \Longrightarrow \mathit{finite}\ (I\ {\downarrow}{>}\ t)$
thm *finite-subset*
by (*rule finite-subset*[*of* - *I*], *rule i-cut-subset*, *assumption*+)+

lemma *nat-cut-le-finite*: $\mathit{finite}\ (I\ {\downarrow}{\leq}\ (t{::}\mathit{nat}))$
by (*fastsimp simp*: *finite-nat-iff-bounded-le2 cut-le-def*)
lemma *nat-cut-less-finite*: $\mathit{finite}\ (I\ {\downarrow}{<}\ (t{::}\mathit{nat}))$
by (*fastsimp simp*: *finite-nat-iff-bounded2 cut-less-def*)
lemma *nat-cut-ge-finite-iff*: $\mathit{finite}\ (I\ {\downarrow}{\geq}\ (t{::}\mathit{nat})) = \mathit{finite}\ I$
apply (*rule iffI*)
 thm *cut-less-cut-ge-ident*[*OF order-refl*]
 apply (*subst cut-less-cut-ge-ident*[*of t*, *OF order-refl*, *symmetric*])
 apply (*simp add*: *nat-cut-less-finite*)
apply (*simp add*: *cut-ge-finite*)
done
lemma *nat-cut-greater-finite-iff*: $\mathit{finite}\ (I\ {\downarrow}{>}\ (t{::}\mathit{nat})) = \mathit{finite}\ I$
thm *cut-ge-greater-conv*
by (*simp only*: *nat-cut-ge-greater-conv*[*symmetric*] *nat-cut-ge-finite-iff*)

lemma
  cut-le-card: $\mathit{finite}\ I \Longrightarrow \mathit{card}\ (I\ {\downarrow}{\leq}\ t) \leq \mathit{card}\ I$ **and**
  cut-less-card: $\mathit{finite}\ I \Longrightarrow \mathit{card}\ (I\ {\downarrow}{<}\ t) \leq \mathit{card}\ I$ **and**
  cut-ge-card: $\mathit{finite}\ I \Longrightarrow \mathit{card}\ (I\ {\downarrow}{\geq}\ t) \leq \mathit{card}\ I$ **and**
  cut-greater-card: $\mathit{finite}\ I \Longrightarrow \mathit{card}\ (I\ {\downarrow}{>}\ t) \leq \mathit{card}\ I$
by (*rule card-mono*, *assumption*, *rule i-cut-subset*)+

lemma *nat-cut-greater-card*: $\mathit{card}\ (I\ {\downarrow}{>}\ (t{::}\mathit{nat})) \leq \mathit{card}\ I$
apply (*case-tac finite I*)
 apply (*simp add*: *cut-greater-card*)
thm *nat-cut-greater-finite-iff*
apply (*simp add*: *nat-cut-greater-finite-iff*)
done
lemma *nat-cut-ge-card*: $\mathit{card}\ (I\ {\downarrow}{\geq}\ (t{::}\mathit{nat})) \leq \mathit{card}\ I$
apply (*case-tac finite I*)
 apply (*simp add*: *cut-ge-card*)
thm *nat-cut-ge-finite-iff*
apply (*simp add*: *nat-cut-ge-finite-iff*)
done

7.2.6 Cutting a set at *Min* or *Max* element

lemma *cut-greater-Min-eq-Diff*: $I\ {\downarrow}{>}\ (\mathit{iMin}\ I) = I - \{\mathit{iMin}\ I\}$

by *blast*
lemma *cut-less-Max-eq-Diff*: *finite* $I \Longrightarrow I \downarrow< (Max\ I) = I - \{Max\ I\}$
by *blast*

lemma *cut-le-Min-empty*: $t < iMin\ I \Longrightarrow I \downarrow\leq t = \{\}$
by (*fastsimp simp*: *i-cut-defs*)
lemma *cut-less-Min-empty*: $t \leq iMin\ I \Longrightarrow I \downarrow< t = \{\}$
by (*fastsimp simp*: *i-cut-defs*)

lemma *cut-le-Min-not-empty*: $[\![\ I \neq \{\};\ iMin\ I \leq t\ ]\!] \Longrightarrow I \downarrow\leq t \neq \{\}$
apply (*simp add*: *i-cut-defs*)
apply (*rule-tac x=iMin I* **in** *exI*)
thm *iMinI-ex2*
apply (*simp add*: *iMinI-ex2*)
done
lemma *cut-less-Min-not-empty*: $[\![\ I \neq \{\};\ iMin\ I < t\ ]\!] \Longrightarrow I \downarrow< t \neq \{\}$
apply (*simp add*: *i-cut-defs*)
apply (*rule-tac x=iMin I* **in** *exI*)
apply (*simp add*: *iMinI-ex2*)
done

lemma *cut-ge-Min-all*: $t \leq iMin\ I \Longrightarrow I \downarrow\geq t = I$
apply (*simp add*: *i-cut-defs*)
apply *safe*
thm *iMin-le*
apply (*drule iMin-le*, *simp*)
done

lemma *cut-greater-Min-all*: $t < iMin\ I \Longrightarrow I \downarrow> t = I$
apply (*simp add*: *i-cut-defs*)
apply *safe*
thm *iMin-le*
apply (*drule iMin-le*, *simp*)
done

lemmas *i-cut-min-empty* =
  cut-le-Min-empty
  cut-less-Min-empty
  cut-le-Min-not-empty
  cut-less-Min-not-empty
lemmas *i-cut-min-all* =
  cut-ge-Min-all
  cut-greater-Min-all

thm
  i-cut-min-empty
  i-cut-min-all

lemma *cut-ge-Max-empty*: *finite* $I \Longrightarrow Max\ I < t \Longrightarrow I \downarrow\geq t = \{\}$
by (*fastsimp simp*: *i-cut-defs*)

lemma *cut-greater-Max-empty*: *finite* $I \Longrightarrow Max\ I \leq t \Longrightarrow I \downarrow> t = \{\}$
by (*fastsimp simp*: *i-cut-defs*)

lemma *cut-ge-Max-not-empty*: $[\![\ I \neq \{\}$; *finite* I; $t \leq Max\ I\ ]\!] \Longrightarrow I \downarrow\geq t \neq \{\}$
apply (*simp add*: *i-cut-defs*)
apply (*rule-tac x=Max I* **in** *exI*)
thm *MaxI-ex2*
apply (*simp add*: *MaxI-ex2*)
done

lemma *cut-greater-Max-not-empty*: $[\![\ I \neq \{\}$; *finite* I; $t < Max\ I\ ]\!] \Longrightarrow I \downarrow> t \neq \{\}$
apply (*simp add*: *i-cut-defs*)
apply (*rule-tac x=Max I* **in** *exI*)
apply (*simp add*: *MaxI-ex2*)
done

lemma *cut-le-Max-all*: *finite* $I \Longrightarrow Max\ I \leq t \Longrightarrow I \downarrow\leq t = I$
by (*fastsimp simp*: *i-cut-defs*)

lemma *cut-less-Max-all*: *finite* $I \Longrightarrow Max\ I < t \Longrightarrow I \downarrow< t = I$
by (*fastsimp simp*: *i-cut-defs*)

lemmas *i-cut-max-empty* =
cut-ge-Max-empty
cut-greater-Max-empty
cut-ge-Max-not-empty
cut-greater-Max-not-empty
lemmas *i-cut-max-all* =
cut-le-Max-all
cut-less-Max-all

thm
i-cut-max-empty
i-cut-max-all

lemma *cut-less-Max-less*:
$[\![$ *finite* $(I \downarrow< t)$; $I \downarrow< t \neq \{\}\ ]\!] \Longrightarrow Max\ (I \downarrow< t) < t$
by (*rule cut-less-bound*[*OF Max-in*])
lemma *cut-le-Max-le*:
$[\![$ *finite* $(I \downarrow\leq t)$; $I \downarrow\leq t \neq \{\}\ ]\!] \Longrightarrow Max\ (I \downarrow\leq t) \leq t$
by (*rule cut-le-bound*[*OF Max-in*])
lemma *nat-cut-less-Max-less*:
$I \downarrow< t \neq \{\} \Longrightarrow Max\ (I \downarrow< t) < (t::nat)$
by (*rule cut-less-bound*[*OF Max-in*[*OF nat-cut-less-finite*]])

lemma *nat-cut-le-Max-le*:
  $I \downarrow\leq t \neq \{\} \Longrightarrow Max\ (I \downarrow\leq t) \leq (t::nat)$
by (*rule cut-le-bound*[*OF Max-in*[*OF nat-cut-le-finite*]])
lemma *cut-greater-Min-greater*:
  $I \downarrow> t \neq \{\} \Longrightarrow iMin\ (I \downarrow> t) > t$
by (*rule cut-greater-bound*[*OF iMinI-ex2*])
lemma *cut-ge-Min-greater*:
  $I \downarrow\geq t \neq \{\} \Longrightarrow iMin\ (I \downarrow\geq t) \geq t$
by (*rule cut-ge-bound*[*OF iMinI-ex2*])

lemma *cut-less-Min-eq*: $I \downarrow< t \neq \{\} \Longrightarrow iMin\ (I \downarrow< t) = iMin\ I$
apply (*drule cut-less-not-empty-iff*[*THEN iffD1*])
apply (*rule iMin-equality*)
 apply (*fastsimp intro*: *iMinI*)
apply *blast*
done

lemma *cut-le-Min-eq*: $I \downarrow\leq t \neq \{\} \Longrightarrow iMin\ (I \downarrow\leq t) = iMin\ I$
apply (*drule cut-le-not-empty-iff*[*THEN iffD1*])
apply (*rule iMin-equality*)
 apply (*fastsimp intro*: *iMinI*)
apply *blast*
done

lemma *cut-ge-Max-eq*: $[\![\ finite\ I;\ I \downarrow\geq t \neq \{\}\ ]\!] \Longrightarrow Max\ (I \downarrow\geq t) = Max\ I$
apply (*drule cut-ge-not-empty-iff*[*THEN iffD1*])
apply (*rule Max-equality*)
  apply (*fastsimp intro*: *MaxI*)
 apply (*simp add*: *cut-ge-finite*)
apply *fastsimp*
done

lemma *cut-greater-Max-eq*: $[\![\ finite\ I;\ I \downarrow> t \neq \{\}\ ]\!] \Longrightarrow Max\ (I \downarrow> t) = Max\ I$
apply (*drule cut-greater-not-empty-iff*[*THEN iffD1*])
apply (*rule Max-equality*)
  apply (*fastsimp intro*: *MaxI*)
 apply (*simp add*: *cut-greater-finite*)
apply *fastsimp*
done

7.2.7 Cut operators with intervals from SetInterval

lemma
  UNIV-cut-le: $UNIV \downarrow\leq t = \{..t\}$ **and**
  UNIV-cut-less: $UNIV \downarrow< t = \{..<t\}$ **and**
  UNIV-cut-ge: $UNIV \downarrow\geq t = \{t..\}$ **and**

UNIV-cut-greater: $UNIV \downarrow> t = \{t<..\}$
by *blast+*

lemma
lessThan-cut-le: $\{..<n\} \downarrow\leq t = (if\ n \leq t\ then\ \{..<n\}\ else\ \{..t\})$ **and**
lessThan-cut-less: $\{..<n\} \downarrow< t = (if\ n \leq t\ then\ \{..<n\}\ else\ \{..<t\})$ **and**
lessThan-cut-ge: $\{..<n\} \downarrow\geq t = \{t..<n\}$ **and**
lessThan-cut-greater: $\{..<n\} \downarrow> t = \{t<..<n\}$ **and**
atMost-cut-le: $\{..n\} \downarrow\leq t = (if\ n \leq t\ then\ \{..n\}\ else\ \{..t\})$ **and**
atMost-cut-less: $\{..n\} \downarrow< t = (if\ n < t\ then\ \{..n\}\ else\ \{..<t\})$ **and**
atMost-cut-ge: $\{..n\} \downarrow\geq t = \{t..n\}$ **and**
atMost-cut-greager: $\{..n\} \downarrow> t = \{t<..n\}$ **and**
greaterThan-cut-le: $\{n<..\} \downarrow\leq t = \{n<..t\}$ **and**
greaterThan-cut-less: $\{n<..\} \downarrow< t = \{n<..<t\}$ **and**
greaterThan-cut-ge: $\{n<..\} \downarrow\geq t = (if\ t \leq n\ then\ \{n<..\}\ else\ \{t..\})$ **and**
greaterThan-cut-greater: $\{n<..\} \downarrow> t = (if\ t \leq n\ then\ \{n<..\}\ else\ \{t<..\})$ **and**
atLeast-cut-le: $\{n..\} \downarrow\leq t = \{n..t\}$ **and**
atLeast-cut-less: $\{n..\} \downarrow< t = \{n..<t\}$ **and**
atLeast-cut-ge: $\{n..\} \downarrow\geq t = (if\ t \leq n\ then\ \{n..\}\ else\ \{t..\})$ **and**
atLeast-cut-greater: $\{n..\} \downarrow\geq t = (if\ t \leq n\ then\ \{n..\}\ else\ \{t..\})$
apply (*simp-all add*: *set-eq-iff i-cut-mem-iff linorder-not-le linorder-not-less*)
apply *fastsimp+*
done

lemma
greaterThanLessThan-cut-le: $\{m<..<n\} \downarrow\leq t = (if\ n \leq t\ then\ \{m<..<n\}\ else\ \{m<..t\})$ **and**
greaterThanLessThan-cut-less: $\{m<..<n\} \downarrow< t = (if\ n \leq t\ then\ \{m<..<n\}\ else\ \{m<..<t\})$ **and**
greaterThanLessThan-cut-ge: $\{m<..<n\} \downarrow\geq t = (if\ t \leq m\ then\ \{m<..<n\}\ else\ \{t..<n\})$ **and**
greaterThanLessThan-cut-greater: $\{m<..<n\} \downarrow> t = (if\ t \leq m\ then\ \{m<..<n\}\ else\ \{t<..<n\})$ **and**
atLeastLessThan-cut-le: $\{m..<n\} \downarrow\leq t = (if\ n \leq t\ then\ \{m..<n\}\ else\ \{m..t\})$ **and**
atLeastLessThan-cut-less: $\{m..<n\} \downarrow< t = (if\ n \leq t\ then\ \{m..<n\}\ else\ \{m..<t\})$ **and**
atLeastLessThan-cut-ge: $\{m..<n\} \downarrow\geq t = (if\ t \leq m\ then\ \{m..<n\}\ else\ \{t..<n\})$ **and**
atLeastLessThan-cut-greater: $\{m..<n\} \downarrow> t = (if\ t < m\ then\ \{m..<n\}\ else\ \{t<..<n\})$ **and**
greaterThanAtMost-cut-le: $\{m<..n\} \downarrow\leq t = (if\ n \leq t\ then\ \{m<..n\}\ else\ \{m<..t\})$ **and**
greaterThanAtMost-cut-less: $\{m<..n\} \downarrow< t = (if\ n < t\ then\ \{m<..n\}\ else\ \{m<..<t\})$ **and**
greaterThanAtMost-cut-ge: $\{m<..n\} \downarrow\geq t = (if\ t \leq m\ then\ \{m<..n\}\ else\ \{t..n\})$ **and**
greaterThanAtMost-cut-greater: $\{m<..n\} \downarrow> t = (if\ t \leq m\ then\ \{m<..n\}\ else$

$\{t<..n\}$) **and**
  atLeastAtMost-cut-le: $\{m..n\} \downarrow\leq t = (\textit{if } n \leq t \textit{ then } \{m..n\} \textit{ else } \{m..t\})$ **and**
  atLeastAtMost-cut-less: $\{m..n\} \downarrow< t = (\textit{if } n < t \textit{ then } \{m..n\} \textit{ else } \{m..<t\})$
and
  atLeastAtMost-cut-ge: $\{m..n\} \downarrow\geq t = (\textit{if } t \leq m \textit{ then } \{m..n\} \textit{ else } \{t..n\})$ **and**
  atLeastAtMost-cut-greater: $\{m..n\} \downarrow> t = (\textit{if } t < m \textit{ then } \{m..n\} \textit{ else } \{t<..n\})$
apply (*simp-all add*: *set-eq-iff i-cut-mem-iff split-if linorder-not-le linorder-not-less*)
apply *fastsimp+*
done

7.2.8 Mirroring finite natural sets between their *Min* and *Max* element

Mirroring a number at the middle of the interval min l r..max l r

Mirroring a single element n between the interval boundaries l and r

definition
  nat-mirror :: $nat \Rightarrow nat \Rightarrow nat \Rightarrow nat$
where
  $\textit{nat-mirror } n\ l\ r \equiv l + r - n$

lemma *nat-mirror-commute*: $\textit{nat-mirror } n\ l\ r = \textit{nat-mirror } n\ r\ l$
unfolding *nat-mirror-def* **by** *simp*

lemma *nat-mirror-inj-on*: $\textit{inj-on } (\lambda x.\ \textit{nat-mirror } x\ l\ r)\ \{..l + r\}$
unfolding *inj-on-def nat-mirror-def* **by** *fastsimp*

lemma *nat-mirror-nat-mirror-ident*:
  $n \leq l + r \Longrightarrow \textit{nat-mirror } (\textit{nat-mirror } n\ l\ r)\ l\ r = n$
unfolding *nat-mirror-def* **by** *simp*

lemma *nat-mirror-add*:
  $\textit{nat-mirror } (n + k)\ l\ r = (\textit{nat-mirror } n\ l\ r) - k$
unfolding *nat-mirror-def* **by** *simp*
lemma *nat-mirror-diff*:
  $[\![\ k \leq n;\ n \leq l + r\ ]\!] \Longrightarrow$
  $\textit{nat-mirror } (n - k)\ l\ r = (\textit{nat-mirror } n\ l\ r) + k$
unfolding *nat-mirror-def* **by** *simp*

lemma *nat-mirror-le*: $a \leq b \Longrightarrow \textit{nat-mirror } b\ l\ r \leq \textit{nat-mirror } a\ l\ r$
unfolding *nat-mirror-def* **by** *simp*
lemma *nat-mirror-le-conv*:
  $a \leq l + r \Longrightarrow (\textit{nat-mirror } b\ l\ r \leq \textit{nat-mirror } a\ l\ r) = (a \leq b)$
unfolding *nat-mirror-def* **by** *fastsimp*

lemma *nat-mirror-less*:
  $[\![\ a < b;\ a < l + r\ ]\!] \Longrightarrow$
  $\textit{nat-mirror } b\ l\ r < \textit{nat-mirror } a\ l\ r$

unfolding *nat-mirror-def* **by** *simp*
lemma *nat-mirror-less-imp-less*:
nat-mirror b l r < nat-mirror a l r ⟹ a < b
unfolding *nat-mirror-def* **by** *simp*
lemma *nat-mirror-less-conv*:
a < l + r ⟹ (nat-mirror b l r < nat-mirror a l r) = (a < b)
unfolding *nat-mirror-def* **by** *fastsimp*

lemma *nat-mirror-eq-conv*:
⟦ a ≤ l + r; b ≤ l + r ⟧ ⟹
(nat-mirror a l r = nat-mirror b l r) = (a = b)
unfolding *nat-mirror-def* **by** *fastsimp*

Mirroring a single element n between the interval boundaries of I

definition
mirror-elem :: nat ⇒ nat set ⇒ nat
where
mirror-elem n I ≡ nat-mirror n (iMin I) (Max I)

lemma *mirror-elem-inj-on*: *finite I ⟹ inj-on (λx. mirror-elem x I) I*
unfolding *mirror-elem-def*
by (*metis Max-le-imp-subset-atMost nat-mirror-inj-on not-add-less2 not-leE subset-inj-on*)

lemma *mirror-elem-add*:
finite I ⟹ mirror-elem (n + k) I = mirror-elem n I − k
unfolding *mirror-elem-def* **by** (*rule nat-mirror-add*)
lemma *mirror-elem-diff*:
⟦ finite I; k ≤ n; n ∈ I ⟧ ⟹ mirror-elem (n − k) I = mirror-elem n I + k
apply (*unfold mirror-elem-def*)
apply (*rule nat-mirror-diff*, *assumption*)
apply (*simp add*: *trans-le-add2*)
done
lemma *mirror-elem-Min*:
⟦ finite I; I ≠ {} ⟧ ⟹ mirror-elem (iMin I) I = Max I
unfolding *mirror-elem-def nat-mirror-def* **by** *simp*
lemma *mirror-elem-Max*:
⟦ finite I; I ≠ {} ⟧ ⟹ mirror-elem (Max I) I = iMin I
unfolding *mirror-elem-def nat-mirror-def* **by** *simp*

lemma *mirror-elem-mirror-elem-ident*:
⟦ finite I; n ≤ iMin I + Max I ⟧ ⟹ mirror-elem (mirror-elem n I) I = n
unfolding *mirror-elem-def nat-mirror-def* **by** *simp*

thm *nat-mirror-le-conv*
lemma *mirror-elem-le-conv*:
⟦ finite I; a ∈ I; b ∈ I ⟧ ⟹
(mirror-elem b I ≤ mirror-elem a I) = (a ≤ b)

apply (*unfold mirror-elem-def*)
apply (*rule nat-mirror-le-conv*)
apply (*simp add*: *trans-le-add2*)
done
lemma *mirror-elem-less-conv*:
⟦ *finite I*; *a* ∈ *I*; *b* ∈ *I* ⟧ ⟹
(*mirror-elem b I* < *mirror-elem a I*) = (*a* < *b*)
unfolding *mirror-elem-def nat-mirror-def*
by (*metis diff-less-mono2 nat-diff-left-cancel-less nat-ex-greater-infinite-finite-Max-conv'* *trans-less-add2*)

thm *nat-mirror-eq-conv*
lemma *mirror-elem-eq-conv*:
⟦ *a* ≤ *iMin I* + *Max I*; *b* ≤ *iMin I* + *Max I* ⟧ ⟹
(*mirror-elem a I* = *mirror-elem b I*) = (*a* = *b*)
by (*simp add*: *mirror-elem-def nat-mirror-eq-conv*)
lemma *mirror-elem-eq-conv'*:
⟦ *finite I*; *a* ∈ *I*; *b* ∈ *I* ⟧ ⟹ (*mirror-elem a I* = *mirror-elem b I*) = (*a* = *b*)
apply (*rule mirror-elem-eq-conv*)
apply (*simp-all add*: *trans-le-add2*)
done

definition
imirror-bounds :: *nat set* ⇒ *nat* ⇒ *nat* ⇒ *nat set*
where
imirror-bounds I l r ≡ (λ*x*. *nat-mirror x l r*) ‘ *I*

Mirroring all elements between the interval boundaries of I

definition
imirror :: *nat set* ⇒ *nat set*
where
imirror I ≡ (λ*x*. *iMin I* + *Max I* − *x*) ‘ *I*

lemma *imirror-eq-nat-mirror-image*:
imirror I = (λ*x*. *nat-mirror x* (*iMin I*) (*Max I*)) ‘ *I*
unfolding *imirror-def nat-mirror-def* **by** *simp*
lemma *imirror-eq-mirror-elem-image*:
imirror I = (λ*x*. *mirror-elem x I*) ‘ *I*
by (*simp add*: *mirror-elem-def imirror-eq-nat-mirror-image*)

lemma *imirror-eq-imirror-bounds*:
imirror I = *imirror-bounds I* (*iMin I*) (*Max I*)
unfolding *imirror-def imirror-bounds-def nat-mirror-def* **by** *simp*

lemma *imirror-empty*: *imirror* {} = {}

unfolding *imirror-def* **by** *simp*
lemma *imirror-is-empty*: $(imirror\ I = \{\}) = (I = \{\})$
unfolding *imirror-def* **by** *simp*
lemma *imirror-not-empty*: $I \neq \{\} \Longrightarrow imirror\ I \neq \{\}$
unfolding *imirror-def* **by** *simp*

lemma *imirror-singleton*: $imirror\ \{a\} = \{a\}$
unfolding *imirror-def* **by** *simp*

lemma *imirror-finite*: $finite\ I \Longrightarrow finite\ (imirror\ I)$
unfolding *imirror-def* **by** *simp*

lemma *imirror-bounds-iMin*:
$[\![\ finite\ I;\ I \neq \{\};\ iMin\ I \leq l + r\ ]\!] \Longrightarrow$
$iMin\ (imirror\text{-}bounds\ I\ l\ r) = l + r - Max\ I$
apply (*unfold imirror-bounds-def nat-mirror-def*)
thm *iMin-equality*
apply (*rule iMin-equality*)
apply (*blast intro*: *Max-in*)
apply (*blast intro*: *Max-ge diff-le-mono2*)
done

lemma *imirror-bounds-Max*:
$[\![\ finite\ I;\ I \neq \{\};\ Max\ I \leq l + r\ ]\!] \Longrightarrow$
$Max\ (imirror\text{-}bounds\ I\ l\ r) = l + r - iMin\ I$
apply (*unfold imirror-bounds-def nat-mirror-def*)
apply (*rule Max-equality*)
apply (*blast intro*: *iMinI*)
apply *simp*
apply (*blast intro*: *iMin-le diff-le-mono2*)
done

lemma *imirror-iMin*: $finite\ I \Longrightarrow iMin\ (imirror\ I) = iMin\ I$
apply (*case-tac* $I = \{\}$, *simp add*: *imirror-empty*)
apply (*simp add*: *imirror-eq-imirror-bounds imirror-bounds-iMin le-add1*)
done

lemma *imirror-Max*: $finite\ I \Longrightarrow Max\ (imirror\ I) = Max\ I$
apply (*case-tac* $I = \{\}$, *simp add*: *imirror-empty*)
apply (*simp add*: *imirror-eq-imirror-bounds imirror-bounds-Max trans-le-add2*)
done
corollary *imirror-iMin-Max*: $[\![\ finite\ I;\ n \in imirror\ I\ ]\!] \Longrightarrow iMin\ I \leq n \wedge n \leq Max\ I$
apply (*frule Max-ge*[*OF imirror-finite*, *of - n*], *assumption*)
thm *imirror-iMin imirror-Max*
apply (*fastsimp simp*: *imirror-iMin imirror-Max*)
done

thm *image-iff*
lemma *imirror-bounds-iff*:
(*n* ∈ *imirror-bounds I l r*) = (∃ *x*∈*I*. *n* = *l* + *r* − *x*)
by (*simp add*: *imirror-bounds-def nat-mirror-def image-iff*)
lemma *imirror-iff*: (*n* ∈ *imirror I*) = (∃ *x*∈*I*. *n* = *iMin I* + *Max I* − *x*)
by (*simp add*: *imirror-def image-iff*)

lemma *imirror-bounds-imirror-bounds-ident*:
⟦ *finite I*; *Max I* ≤ *l* + *r* ⟧ ⟹
imirror-bounds (*imirror-bounds I l r*) *l r* = *I*
apply (*rule set-eqI*)
apply (*simp add*: *imirror-bounds-def image-image image-iff*)
thm *nat-mirror-nat-mirror-ident*
apply (*rule iffI*)
apply (*fastsimp simp*: *nat-mirror-nat-mirror-ident*)
apply (*rule-tac x*=*x* **in** *bexI*)
apply (*fastsimp simp*: *nat-mirror-nat-mirror-ident*)+
done

lemma *imirror-imirror-ident*: *finite I* ⟹ *imirror* (*imirror I*) = *I*
apply (*case-tac I* = {}, *simp add*: *imirror-empty*)
thm *imirror-eq-imirror-bounds*
thm *imirror-bounds-imirror-bounds-ident*
apply (*simp add*: *imirror-eq-imirror-bounds imirror-bounds-iMin imirror-bounds-Max*
le-add1 trans-le-add2 imirror-bounds-imirror-bounds-ident)
done
lemma *mirror-elem-imirror*:
finite I ⟹ *mirror-elem t* (*imirror I*) = *mirror-elem t I*
by (*simp add*: *mirror-elem-def imirror-iMin imirror-Max*)

lemma *imirror-card*: *finite I* ⟹ *card* (*imirror I*) = *card I*
apply (*simp only*: *imirror-eq-mirror-elem-image*)
apply (*rule inj-on-iff-eq-card*[*THEN iffD1*], *assumption*)
apply (*rule mirror-elem-inj-on*, *assumption*)
done

lemma *imirror-bounds-elem-conv*:
⟦ *finite I*; *n* ≤ *l* + *r*; *Max I* ≤ *l* + *r* ⟧ ⟹
((*nat-mirror n l r*) ∈ *imirror-bounds I l r*) = (*n* ∈ *I*)
apply (*unfold imirror-bounds-def*)
thm *inj-on-image-mem-iff*[**where** *A*={..*l* + *r*}]
apply (*rule inj-on-image-mem-iff*)
apply (*rule nat-mirror-inj-on*)
apply *fastsimp*
apply *simp*
done

lemma *imirror-mem-conv*:

⟦ *finite I*; $n \leq iMin\ I + Max\ I$ ⟧ ⟹ ((*mirror-elem n I*) ∈ *imirror I*) = ($n \in I$)
thm *imirror-bounds-elem-conv*
by (*simp add*: *mirror-elem-def imirror-eq-imirror-bounds imirror-bounds-elem-conv*)

corollary *in-imp-mirror-elem-in*:
⟦ *finite I*; $n \in I$ ⟧ ⟹ (*mirror-elem n I*) ∈ *imirror I*
thm *imirror-mem-conv*[*OF - trans-le-add2*[*OF Max-ge*], *THEN iffD2*]
by (*rule imirror-mem-conv*[*OF - trans-le-add2*[*OF Max-ge*], *THEN iffD2*])

lemma *imirror-cut-less*:
finite I ⟹
imirror I $\downarrow<$ (*mirror-elem t I*) =
imirror-bounds (*I* $\downarrow>$ *t*) (*iMin I*) (*Max I*)
apply (*simp only*: *imirror-eq-imirror-bounds*)
apply (*unfold imirror-def imirror-bounds-def mirror-elem-def*)
apply (*rule set-eqI*)
apply (*simp add*: *Bex-def i-cut-mem-iff image-iff*)
apply (*rule iffI*)
thm *nat-mirror-less-imp-less*
apply (*bestsimp intro*: *nat-mirror-less-imp-less*)
thm *nat-mirror-less*
apply (*bestsimp simp add*: *nat-mirror-less*)
done

lemma *imirror-cut-le*:
⟦ *finite I*; $t \leq iMin\ I + Max\ I$ ⟧ ⟹
imirror I $\downarrow\leq$ (*mirror-elem t I*) =
imirror-bounds (*I* $\downarrow\geq$ *t*) (*iMin I*) (*Max I*)
thm *nat-cut-le-less-conv nat-cut-greater-ge-conv*
apply (*simp only*: *nat-cut-le-less-conv*)
apply (*case-tac t = 0*)
apply (*simp add*: *cut-ge-0-all i-cut-empty*)
apply (*simp only*: *imirror-eq-imirror-bounds*[*symmetric*])
apply (*rule cut-less-Max-all*)
apply (*rule imirror-finite*, *assumption*)
apply (*simp add*: *mirror-elem-def nat-mirror-def imirror-Max*)
apply (*simp add*: *nat-cut-greater-ge-conv*[*symmetric*])
thm *imirror-cut-less*
apply (*rule subst*[*of mirror-elem* (*t − Suc 0*) *I Suc* (*mirror-elem t I*)])
apply (*simp add*: *mirror-elem-def nat-mirror-diff*)
apply (*rule imirror-cut-less*, *assumption*)
done

lemma *imirror-cut-ge*:
finite I ⟹
imirror I $\downarrow\geq$ (*mirror-elem t I*) =
imirror-bounds (*I* $\downarrow\leq$ *t*) (*iMin I*) (*Max I*)
(**is** *?P* ⟹ *?lhs I* = *?rhs I t*)
apply (*case-tac iMin I + Max I < t*)

apply (*simp add*: *mirror-elem-def nat-mirror-def cut-ge-0-all cut-le-Max-all imirror-eq-imirror-bounds*)
apply (*case-tac t < iMin I*)
apply (*simp add*: *cut-le-Min-empty imirror-bounds-def mirror-elem-def nat-mirror-def cut-ge-Max-empty imirror-Max imirror-finite*)
apply (*simp add*: *linorder-not-le linorder-not-less*)
apply (*rule subst*[*of imirror* (*imirror I*) ↓≤ *mirror-elem* (*mirror-elem t I*) (*imirror I*) *I* ↓≤ *t*])
thm *mirror-elem-mirror-elem-ident*
apply (*simp add*: *imirror-imirror-ident mirror-elem-imirror mirror-elem-mirror-elem-ident*)
thm *imirror-cut-le*[*of imirror I mirror-elem t I*, *OF imirror-finite*]
apply (*subgoal-tac mirror-elem t I* ≤ *Max* (*imirror I*))
prefer *2*
apply (*simp add*: *imirror-Max mirror-elem-def nat-mirror-def*)
apply (*simp add*: *imirror-cut-le imirror-finite*)
by (*metis cut-ge-Max-eq cut-ge-Max-not-empty imirror-Max imirror-bounds-imirror-bounds-ident imirror-finite imirror-iMin le-add2 nat-cut-ge-finite-iff*)

lemma *imirror-cut-greater*: ⟦ *finite I*; *t* ≤ *iMin I* + *Max I* ⟧ ⟹
imirror I ↓> (*mirror-elem t I*) =
imirror-bounds (*I* ↓< *t*) (*iMin I*) (*Max I*)
apply (*case-tac t* = *0*)
apply (*simp add*: *cut-less-0-empty imirror-bounds-def*)
apply (*rule cut-greater-Max-empty*)
apply (*rule imirror-finite*, *assumption*)
apply (*simp add*: *imirror-Max mirror-elem-def nat-mirror-def*)
thm *nat-cut-ge-greater-conv*
apply (*simp add*: *nat-cut-ge-greater-conv*[*symmetric*])
thm *subst*[*of mirror-elem* (*t* − *Suc 0*) *I Suc* (*mirror-elem t I*)]
apply (*rule subst*[*of mirror-elem* (*t* − *Suc 0*) *I Suc* (*mirror-elem t I*)])
thm *nat-mirror-diff*
apply (*simp add*: *mirror-elem-def nat-mirror-diff*)
thm *imirror-cut-ge nat-cut-less-le-conv*
apply (*simp add*: *imirror-cut-ge nat-cut-less-le-conv*)
done

lemmas *imirror-cut* =
imirror-cut-less imirror-cut-ge
imirror-cut-le imirror-cut-greater
thm *imirror-cut*

corollary *imirror-cut-le'*:
⟦ *finite I*; *t* ∈ *I* ⟧ ⟹
imirror I ↓≤ *mirror-elem t I* =
imirror-bounds (*I* ↓≥ *t*) (*iMin I*) (*Max I*)
thm *imirror-cut-le*[*OF - trans-le-add2*[*OF Max-ge*]]
by (*rule imirror-cut-le*[*OF - trans-le-add2*[*OF Max-ge*]])
corollary *imirror-cut-greater'*:
⟦ *finite I*; *t* ∈ *I* ⟧ ⟹
imirror I ↓> *mirror-elem t I* =

imirror-bounds (*I* ↓< *t*) (*iMin I*) (*Max I*)
by (*rule imirror-cut-greater*[*OF - trans-le-add2*[*OF Max-ge*]])
lemmas *imirror-cut'* =
imirror-cut-le' imirror-cut-greater'
thm
imirror-cut
imirror-cut'

lemma *imirror-bounds-Un*:
imirror-bounds (*A* ∪ *B*) *l r* =
imirror-bounds A l r ∪ *imirror-bounds B l r*
by (*simp add*: *imirror-bounds-def image-Un*)
lemma *imirror-bounds-Int*:
⟦ *A* ⊆ {..*l* + *r*}; *B* ⊆ {..*l* + *r*} ⟧ ⟹
imirror-bounds (*A* ∩ *B*) *l r* =
imirror-bounds A l r ∩ *imirror-bounds B l r*
apply (*unfold imirror-bounds-def*)
thm *inj-on-image-Int*[*OF - Un-upper1 Un-upper2*]
apply (*rule inj-on-image-Int*[*OF - Un-upper1 Un-upper2*])
thm *nat-mirror-inj-on*
thm *subset-inj-on*[*OF nat-mirror-inj-on*]
apply (*rule subset-inj-on*[*OF nat-mirror-inj-on*])
apply (*rule Un-least*[*of A - B*], *assumption*+)
done

end

8 SetIntervalStep: Stepping through sets of natural numbers

theory *SetIntervalStep*
imports *SetIntervalCut*
begin

8.1 Function *inext* and *iprev* for stepping through natural sets

definition
inext :: *nat* ⇒ *nat set* ⇒ *nat*
where
inext n I ≡ (
if (*n* ∈ *I* ∧ (*I* ↓> *n* ≠ {}))
then iMin (*I* ↓> *n*)
else n)

definition
iprev :: *nat* ⇒ *nat set* ⇒ *nat*

where
iprev n I ≡ (
if (*n* ∈ *I* ∧ (*I* ↓< *n* ≠ {}))
then Max (*I* ↓< *n*)
else n)

inext and *iprev* can be viewed as generalisations of *Suc* and *prev*

lemma *inext-UNIV*: *inext n UNIV* = *Suc n*
apply (*simp add*: *inext-def cut-greater-def*, *safe*)
apply (*rule iMin-equality*)
apply *fastsimp*+
done
lemma *iprev-UNIV*: *iprev n UNIV* = *n* − *Suc 0*
apply (*simp add*: *iprev-def cut-less-def*, *safe*)
apply (*rule Max-equality*)
apply *fastsimp*+
done

lemma *inext-empty*: *inext n* {} = *n*
unfolding *inext-def* **by** *simp*
lemma *iprev-empty*: *iprev n* {} = *n*
unfolding *iprev-def* **by** *simp*

thm
finite-nat-iff-bounded-le
finite-nat-iff-bounded-le2

lemma *not-in-inext-fix*: *n* ∉ *I* ⟹ *inext n I* = *n*
unfolding *inext-def* **by** *simp*
lemma *not-in-iprev-fix*: *n* ∉ *I* ⟹ *iprev n I* = *n*
unfolding *iprev-def* **by** *simp*

lemma *inext-all-le-fix*: ∀ *x*∈*I*. *x* ≤ *n* ⟹ *inext n I* = *n*
unfolding *inext-def* **by** *force*
lemma *iprev-all-ge-fix*: ∀ *x*∈*I*. *n* ≤ *x* ⟹ *iprev n I* = *n*
unfolding *iprev-def* **by** *force*

lemma *inext-Max*: *finite I* ⟹ *inext* (*Max I*) *I* = *Max I*
unfolding *inext-def cut-greater-def* **by** (*fastsimp dest*: *Max-ge*)
lemma *iprev-iMin*: *iprev* (*iMin I*) *I* = *iMin I*
unfolding *iprev-def cut-less-def* **by** *fastsimp*

lemma *inext-ge-Max*: ⟦ *finite I*; *Max I* ≤ *n* ⟧ ⟹ *inext n I* = *n*
unfolding *inext-def cut-greater-def* **by** (*fastsimp dest*: *Max-ge*)
thm *iprev-iMin*
lemma *iprev-le-iMin*: *n* ≤ *iMin I* ⟹ *iprev n I* = *n*
unfolding *iprev-def cut-less-def* **by** *fastsimp*

lemma *inext-singleton*: *inext n* $\{a\}$ = *n*
unfolding *inext-def* **by** *fastsimp*

lemma *iprev-singleton*: *iprev n* $\{a\}$ = *n*
unfolding *iprev-def* **by** *fastsimp*

lemma *inext-closed*: *n* ∈ *I* ⟹ *inext n I* ∈ *I*
apply (*clarsimp simp*: *inext-def*)
thm *subsetD*[*of I* ↓> *n I*]
apply (*rule subsetD*[*OF cut-greater-subset*])
apply (*rule iMinI-ex2*, *assumption*)
done

lemma *iprev-closed*: *n* ∈ *I* ⟹ *iprev n I* ∈ *I*
apply (*clarsimp simp*: *iprev-def*)
thm *subsetD*[*of I* ↓< *n I*]
apply (*rule subsetD*[*of I* ↓< *n*], *fastsimp*)
thm *Max-in*[*OF nat-cut-less-finite*]
by (*rule Max-in*[*OF nat-cut-less-finite*])

thm *inext-closed*
lemma *inext-in-imp-in*: *inext n I* ∈ *I* ⟹ *n* ∈ *I*
by (*case-tac n* ∈ *I*, *simp-all add*: *not-in-inext-fix*)

lemma *inext-in-iff*: (*inext n I* ∈ *I*) = (*n* ∈ *I*)
apply (*rule iffI*)
apply (*rule inext-in-imp-in*, *assumption*)
apply (*rule inext-closed*, *assumption*)
done

lemma *subset-inext-closed*: ⟦ *n* ∈ *B*; *A* ⊆ *B* ⟧ ⟹ *inext n A* ∈ *B*
apply (*case-tac n* ∈ *A*)
apply (*fastsimp simp*: *inext-closed*)
apply (*simp add*: *not-in-inext-fix*)
done
lemma *subset-inext-in-imp-in*: ⟦ *inext n A* ∈ *B*; *A* ⊆ *B* ⟧ ⟹ *n* ∈ *B*
apply (*case-tac n* ∈ *A*)
apply *fastsimp*
apply (*simp add*: *not-in-inext-fix*)
done
lemma *subset-inext-in-iff*: *A* ⊆ *B* ⟹ (*inext n A* ∈ *B*) = (*n* ∈ *B*)
apply (*rule iffI*)
apply (*rule subset-inext-in-imp-in*, *assumption*+)
apply (*rule subset-inext-closed*, *assumption*+)
done

thm *iprev-closed*
lemma *iprev-in-imp-in*: *iprev n I* $\in$ *I* $\Longrightarrow$ *n* $\in$ *I*
apply (*case-tac n* $\in$ *I*)
apply (*simp-all add*: *not-in-iprev-fix*)
done
lemma *iprev-in-iff*: (*iprev n I* $\in$ *I*) = (*n* $\in$ *I*)
apply (*rule iffI*)
apply (*rule iprev-in-imp-in*, *assumption*)
apply (*rule iprev-closed*, *assumption*)
done

lemma *subset-iprev-closed*: $[\![$ *n* $\in$ *B*; *A* $\subseteq$ *B* $]\!]$ $\Longrightarrow$ *iprev n A* $\in$ *B*
apply (*case-tac n* $\in$ *A*)
apply (*fastsimp simp*: *iprev-closed*)
apply (*simp add*: *not-in-iprev-fix*)
done
lemma *subset-iprev-in-imp-in*: $[\![$ *iprev n A* $\in$ *B*; *A* $\subseteq$ *B* $]\!]$ $\Longrightarrow$ *n* $\in$ *B*
apply (*case-tac n* $\in$ *A*)
apply *fastsimp*
apply (*simp add*: *not-in-iprev-fix*)
done
lemma *subset-iprev-in-iff*: *A* $\subseteq$ *B* $\Longrightarrow$ (*iprev n A* $\in$ *B*) = (*n* $\in$ *B*)
apply (*rule iffI*)
apply (*rule subset-iprev-in-imp-in*, *assumption*+)
apply (*rule subset-iprev-closed*, *assumption*+)
done

lemma *inext-mono*: *n* $\leq$ *inext n I*
by (*simp add*: *inext-def i-cut-defs iMin-ge-iff*)
corollary *inext-neq-imp-less*: *n* $\neq$ *inext n I* $\Longrightarrow$ *n* $<$ *inext n I*
by (*insert inext-mono*[*of n I*], *simp*)

lemma *inext-mono2*: $[\![$ *n* $\in$ *I*; $\exists$ *x*$\in$*I*. *n* $<$ *x* $]\!]$ $\Longrightarrow$ *n* $<$ *inext n I*
by (*fastsimp simp add*: *inext-def i-cut-defs iMin-gr-iff*)

lemma *inext-mono2-infin*: $[\![$ *n* $\in$ *I*; *infinite I* $]\!]$ $\Longrightarrow$ *n* $<$ *inext n I*
apply (*simp add*: *inext-def i-cut-defs iMin-gr-iff*)
apply (*fastsimp simp*: *infinite-nat-iff-unbounded*)
done

lemma *inext-mono2-fin*: $[\![$ *n* $\in$ *I*; *finite I*; *n* $\neq$ *Max I* $]\!]$ $\Longrightarrow$ *n* $<$ *inext n I*
apply (*simp add*: *inext-def i-cut-defs iMin-gr-iff*)
apply (*blast intro*: *Max-ge Max-in*)
done

thm *inext-mono2*
lemma *inext-mono2-infin-fin*:
⟦ $n \in I$; $n \neq Max\ I \vee infinite\ I$ ⟧ ⟹ $n < inext\ n\ I$
by (*blast intro*: *inext-mono2-infin inext-mono2-fin*)

thm *Nat.zero-less-Suc*
lemma *inext-neq-iMin*: ∃ $x{\in}I.\ n < x \Longrightarrow inext\ n\ I \neq iMin\ I$
apply (*case-tac* $n \in I$)
prefer *2*
apply (*simp add*: *not-in-inext-fix*)
apply (*blast dest*: *iMinI*)
apply (*rule not-sym*, *rule less-imp-neq*)
thm *le-less-trans*[*OF iMin-le*[*of n*], *OF - inext-mono2*]
by (*rule le-less-trans*[*OF iMin-le*[*of n*], *OF - inext-mono2*])

lemma *inext-neq-iMin-infin*: *infinite* $I \Longrightarrow inext\ n\ I \neq iMin\ I$
apply (*rule inext-neq-iMin*)
thm *infinite-nat-iff-unbounded*[*THEN iffD1*]
apply (*blast dest*: *infinite-nat-iff-unbounded*[*THEN iffD1*])
done

thm *Max-le-Min-imp-singleton*
lemma *Max-le-iMin-imp-singleton*: ⟦ *finite* I; $I \neq \{\}$; $Max\ I \leq iMin\ I$ ⟧ ⟹ $I = \{iMin\ I\}$
by (*simp add*: *iMin-Min-conv Max-le-Min-imp-singleton*)

lemma *inext-neq-iMin-not-singleton*:
⟦ $I \neq \{\}$; ¬(∃ $a.\ I = \{a\}$) ⟧ ⟹ $inext\ n\ I \neq iMin\ I$
apply (*case-tac finite* I)
prefer *2*
apply (*simp add*: *inext-neq-iMin-infin*)
apply (*case-tac* $n \in I$)
prefer *2*
apply (*simp add*: *not-in-inext-fix*)
apply (*blast intro*: *iMinI-ex2*)
by (*metis Max-le-iMin-imp-singleton iMin-le-Max inext-Max inext-mono2-infin-fin not-less-iMin*)
corollary *inext-neq-iMin-not-card-1*:
⟦ $I \neq \{\}$; $card\ I \neq Suc\ 0$ ⟧ ⟹ $inext\ n\ I \neq iMin\ I$
by (*simp add*: *inext-neq-iMin-not-singleton card-1-singleton-conv*)

lemma *inext-neq-imp-Max*: $n \neq inext\ n\ I \Longrightarrow n < Max\ I \vee infinite\ I$
by (*rule ccontr*, *clarsimp simp*: *inext-ge-Max*)

lemma *inext-less-conv*: $(n \in I \wedge (n < Max\ I \vee infinite\ I)) = (n < inext\ n\ I)$
apply (*rule iffI*)
apply (*blast intro*: *inext-mono2-infin-fin*)
apply (*rule conjI*)
apply (*rule ccontr*)

apply (*simp add*: *not-in-inext-fix*)
apply (*blast dest*: *inext-neq-imp-Max less-imp-neq*)
done

lemma *inext-min-step*: ⟦ $n < k$; $k <$ *inext* $n\ I$ ⟧ ⟹ $k \notin I$
apply (*case-tac* $n \in I$)
prefer *2*
apply (*simp add*: *inext-def*)
thm *contrapos-pn*[*of* $k <$ *inext* $n\ I\ k \in I$]
apply (*rule contrapos-pn*[*of* $k <$ *inext* $n\ I\ k \in I$], *simp*)
apply (*simp add*: *inext-def i-cut-defs*)
apply (*case-tac* $\exists x.\ x \in I \wedge n < x$)
apply *simp*
thm *not-less-iMin*
thm *not-less-iMin*[*of* k $\{x \in I.\ n < x\}$]
apply (*blast dest*: *not-less-iMin*)
apply *blast*
done
corollary *inext-min-step2*: $\neg(\exists k{\in}I.\ n < k \wedge k <$ *inext* $n\ I)$
by (*clarsimp simp add*: *inext-min-step*)

lemma *min-step-inext*[*rule-format*]:
⟦ $x < y$; $x \in I$; $y \in I$; $\bigwedge k.$ ⟦ $x < k$; $k < y$ ⟧ ⟹ $k \notin I$ ⟧ ⟹
inext $x\ I = y$
apply (*rule ccontr*)
thm *nat-neq-iff*
apply (*simp add*: *nat-neq-iff*, *safe*)
thm *inext-closed*[*of* $x\ I$]
thm *inext-mono2*[*of* $x\ I$]
apply (*blast dest*: *inext-closed inext-mono2*)
thm *inext-min-step*[*of* $x\ y\ I$]
apply (*simp add*: *inext-min-step*)
done

corollary *min-step-inext2*[*rule-format*]:
⟦ $x < y$; $x \in I$; $y \in I$; $\neg(\exists k \in I.\ x < k \wedge k < y)$ ⟧ ⟹
inext $x\ I = y$
by (*blast intro*: *min-step-inext*)
lemma *between-empty-imp-inext-eq*:
⟦ $n \in A$; $n <$ *inext* $n\ A$; $n \in B$; *inext* $n\ A \in B$; $B \downarrow> n \downarrow< (\mathit{inext}\ n\ A) = \{\}$ ⟧
⟹
inext $n\ B =$ *inext* $n\ A$
by (*blast intro*: *min-step-inext2*)

lemma *inext-le-mono*: $[\![\ a \leq b;\ a \in I;\ b \in I\ ]\!] \Longrightarrow inext\ a\ I \leq inext\ b\ I$
apply (*drule order-le-less*[*THEN iffD1*], *erule disjE*)
prefer *2*
apply *simp*
apply (*rule order-trans*[*of* - *b*])
apply (*rule ccontr*, *simp add*: *linorder-not-le*)
thm *inext-min-step*
apply (*blast dest*: *inext-min-step*)
by (*rule inext-mono*)

thm *inext-mono2*
lemma *inext-less-mono*:
$[\![\ a < b;\ a \in I;\ b \in I;\ \exists x{\in}I.\ b < x\ ]\!] \Longrightarrow inext\ a\ I < inext\ b\ I$
apply (*rule le-less-trans*[*of* - *b*])
apply (*rule ccontr*, *simp add*: *linorder-not-le*)
thm *inext-min-step*
apply (*blast dest*: *inext-min-step*)
by (*rule inext-mono2*)

thm *inext-mono2-fin*
lemma *inext-less-mono-fin*:
$[\![\ a < b;\ a \in I;\ b \in I;\ finite\ I;\ b \neq Max\ I\ ]\!] \Longrightarrow inext\ a\ I < inext\ b\ I$
thm *inext-less-mono Max-in*
by (*blast intro*: *inext-less-mono Max-in*)

thm *inext-mono2-infin*
lemma *inext-less-mono-infin*:
$[\![\ a < b;\ a \in I;\ b \in I;\ infinite\ I\ ]\!] \Longrightarrow inext\ a\ I < inext\ b\ I$
apply (*rule inext-less-mono*, *assumption*+)
apply (*blast dest*: *infinite-imp-asc-chain*)
done
thm *inext-mono2-infin-fin*
lemma *inext-less-mono-infin-fin*:
$[\![\ a < b;\ a \in I;\ b \in I;\ b \neq Max\ I \vee infinite\ I\ ]\!] \Longrightarrow inext\ a\ I < inext\ b\ I$
by (*blast intro*: *inext-less-mono-infin inext-less-mono-fin*)

lemma *inext-le-mono-rev*:
$[\![\ inext\ a\ I \leq inext\ b\ I;\ a \in I;\ b \in I;\ \exists x{\in}I.\ inext\ a\ I < x\ ]\!] \Longrightarrow a \leq b$
apply (*rule ccontr*, *simp add*: *linorder-not-le*)
thm *inext-less-mono*
apply (*frule inext-less-mono*, *assumption*+)
apply (*blast intro*: *le-less-trans inext-mono*)
apply *simp*
done
lemma *inext-le-mono-fin-rev*:
$[\![\ inext\ a\ I \leq inext\ b\ I;\ a \in I;\ b \in I;\ finite\ I;\ inext\ a\ I \neq Max\ I]\!] \Longrightarrow a \leq b$
by (*metis inext-in-iff inext-le-mono-rev inext-mono2-infin-fin*)

lemma *inext-le-mono-infin-rev*:
⟦ *inext a I* ≤ *inext b I*; *a* ∈ *I*; *b* ∈ *I*; *infinite I* ⟧ ⟹ *a* ≤ *b*
by (*metis inext-in-iff inext-le-mono-rev inext-mono2-infin-fin*)
lemma *inext-le-mono-infin-fin-rev*:
⟦ *inext a I* ≤ *inext b I*; *a* ∈ *I*; *b* ∈ *I*; *inext a I* ≠ *Max I* ∨ *infinite I* ⟧ ⟹ *a* ≤ *b*
by (*blast intro*: *inext-le-mono-infin-rev inext-le-mono-fin-rev*)

lemma *inext-less-mono-rev*:
⟦ *inext a I* < *inext b I*; *a* ∈ *I*; *b* ∈ *I* ⟧ ⟹ *a* < *b*
by (*metis inext-le-mono not-le*)

lemma *less-imp-inext-le*: ⟦ *a* < *b*; *a* ∈ *I*; *b* ∈ *I* ⟧ ⟹ *inext a I* ≤ *b*
by (*metis inext-min-step not-le*)

lemma *iprev-mono*: *iprev n I* ≤ *n*
unfolding *iprev-def i-cut-defs* **by** *simp*
corollary *iprev-neq-imp-greater*: *n* ≠ *iprev n I* ⟹ *iprev n I* < *n*
by (*insert iprev-mono*[*of n I*], *simp*)

lemma *iprev-mono2*: ⟦ *n* ∈ *I*; ∃ *x*∈*I*. *x* < *n*⟧ ⟹ *iprev n I* < *n*
apply (*unfold iprev-def i-cut-defs*, *clarsimp*)
thm *finite-nat-iff-bounded*
apply (*blast intro*: *finite-nat-iff-bounded*)+
done

thm *inext-mono2-fin*
lemma *iprev-mono2-if-neq-iMin*: ⟦ *n* ∈ *I*; *iMin I* ≠ *n*⟧ ⟹ *iprev n I* < *n*
thm *iMinI*
thm *iprev-mono2*
by (*blast intro*: *iMinI iprev-mono2*)

thm *inext-neq-iMin*
lemma *iprev-neq-Max*: ⟦ *finite I*; ∃ *x*∈*I*. *x* < *n* ⟧ ⟹ *iprev n I* ≠ *Max I*
apply (*case-tac n* ∈ *I*)
prefer *2*
apply (*simp add*: *not-in-iprev-fix*)
apply (*blast dest*: *Max-in*)
apply (*rule less-imp-neq*)
thm *less-le-trans*[*OF iprev-mono2 Max-ge*]
by (*rule less-le-trans*[*OF iprev-mono2 Max-ge*])

thm *inext-neq-iMin-not-singleton*
lemma *iprev-neq-Max-not-singleton*:
⟦ *finite I*; *I* ≠ {}; ¬(∃ *a*. *I* = {*a*}) ⟧ ⟹ *iprev n I* ≠ *Max I*
apply (*case-tac n* ∈ *I*)

prefer *2*
thm *not-in-iprev-fix*
apply (*simp add*: *not-in-iprev-fix*)
apply (*blast intro*: *Max-in*)
apply (*case-tac n = iMin I*)
apply (*metis Max-le-Min-conv-singleton iMin-Min-conv iMin-le-Max iprev-iMin*)
apply (*metis iprev-mono2-if-neq-iMin not-greater-Max*)
done
corollary *iprev-neq-Max-not-card-1*:
⟦ *finite I*; *I* ≠ {}; *card I* ≠ *Suc 0* ⟧ ⟹ *iprev n I* ≠ *Max I*
apply (*rule iprev-neq-Max-not-singleton*, *assumption*+)
apply (*simp add*: *card-1-singleton-conv*)
done

lemma *iprev-neq-imp-iMin*: *iprev n I* ≠ *n* ⟹ *iMin I* < *n*
by (*rule ccontr*, *clarsimp simp*: *iprev-le-iMin*)

lemma *iprev-greater-conv*: (*n* ∈ *I* ∧ *iMin I* < *n*) = (*iprev n I* < *n*)
apply (*rule iffI*)
apply (*blast intro*: *iprev-mono2-if-neq-iMin*)
apply (*rule conjI*)
apply (*rule ccontr*)
apply (*simp add*: *not-in-iprev-fix*)
apply (*blast dest*: *iprev-neq-imp-iMin less-imp-neq*)
done

lemma *inext-fix-iff*: (*n* ∉ *I* ∨ (*finite I* ∧ *Max I* = *n*)) = (*inext n I* = *n*)
apply (*case-tac n* ∉ *I*, *simp add*: *not-in-inext-fix*)
by (*metis inext-Max inext-min-step2 inext-mono2-infin-fin*)
lemma *iprev-fix-iff*: (*n* ∉ *I* ∨ *iMin I* = *n*) = (*iprev n I* = *n*)
apply (*case-tac n* ∉ *I*, *simp add*: *not-in-iprev-fix*)
by (*metis iprev-iMin iprev-mono2-if-neq-iMin less-not-refl3*)

lemma *iprev-min-step*: ⟦ *iprev n I* < *k*; *k* < *n* ⟧ ⟹ *k* ∉ *I*
apply (*case-tac n* ∈ *I*)
prefer *2*
apply (*simp add*: *iprev-def*)
thm *contrapos-pn*[*of iprev n I* < *k k* ∈ *I*]
apply (*rule contrapos-pn*[*of iprev n I* < *k k* ∈ *I*], *simp*)
apply (*unfold iprev-def i-cut-defs*, *simp*)
apply (*split split-if-asm*)
thm *Max-ge*[*of* {*x* ∈ *I*. *x* < *n*} *k*]
apply (*cut-tac Max-ge*[*of* {*x* ∈ *I*. *x* < *n*} *k*])
apply *fastsimp*+
done

corollary *iprev-min-step2*: $\neg(\exists\, x{\in}I.\ iprev\ n\ I < x \wedge x < n)$
by (*clarsimp simp add*: *iprev-min-step*)

lemma *min-step-iprev*:
$[\![\ x < y;\ x \in I;\ y \in I;\ \bigwedge k.\ [\![\ x < k;\ k < y\ ]\!] \Longrightarrow k \notin I\ ]\!] \Longrightarrow$
$iprev\ y\ I = x$
thm *ccontr*
apply (*rule ccontr*)
thm *nat-neq-iff*
apply (*simp add*: *nat-neq-iff*, *elim disjE*)
thm *iprev-min-step*
apply (*simp add*: *iprev-min-step*)
thm *iprev-closed*
thm *iprev-mono2*
apply (*blast dest*: *iprev-closed iprev-mono2 iprev-min-step*)
done
corollary *min-step-iprev2*[*rule-format*]:
$[\![\ x < y;\ x \in I;\ y \in I;\ \neg(\exists\, k \in I.\ x < k \wedge k < y)\ ]\!] \Longrightarrow$
$iprev\ y\ I = x$
by (*blast intro*: *min-step-iprev*)
lemma *between-empty-imp-iprev-eq*:
$[\![\ n \in A;\ iprev\ n\ A < n;\ n \in B;\ iprev\ n\ A \in B;\ B \downarrow> (iprev\ n\ A) \downarrow< n = \{\}\ ]\!]$
$\Longrightarrow$
$iprev\ n\ B = iprev\ n\ A$
by (*blast intro*: *min-step-iprev2*)

lemma *iprev-le-mono*: $[\![\ a \le b;\ a \in I;\ b \in I\ ]\!] \Longrightarrow iprev\ a\ I \le iprev\ b\ I$
apply (*drule order-le-less*[*THEN iffD1*], *erule disjE*)
prefer *2*
apply *simp*
apply (*rule order-trans*[*OF iprev-mono*])
apply (*rule ccontr*, *simp add*: *linorder-not-le*)
thm *iprev-min-step*
by (*blast dest*: *iprev-min-step*)
lemma *iprev-less-mono*:
$[\![\ a < b;\ a \in I;\ b \in I;\ \exists\, x{\in}I.\ x < a\ ]\!] \Longrightarrow iprev\ a\ I < iprev\ b\ I$
apply (*rule less-le-trans*[*of - a*])
apply (*blast intro*: *iprev-mono2*)
apply (*rule ccontr*, *simp add*: *linorder-not-le*)
thm *iprev-min-step*
by (*blast dest*: *iprev-min-step*)

lemma *iprev-less-mono-if-neq-iMin*:
$[\![\ a < b;\ a \in I;\ b \in I;\ iMin\ I \neq a\ ]\!] \Longrightarrow iprev\ a\ I < iprev\ b\ I$
by (*metis iprev-in-iff iprev-less-mono iprev-mono2-if-neq-iMin*)

thm *inext-le-mono-rev*
lemma *iprev-le-mono-rev*:
⟦ *iprev a I* ≤ *iprev b I*; *a* ∈ *I*; *b* ∈ *I*; *iMin I* ≠ *iprev b I* ⟧ ⟹ *a* ≤ *b*
apply (*rule ccontr*, *simp add*: *linorder-not-le*)
by (*metis iprev-fix-iff iprev-less-mono-if-neq-iMin less-le-not-le*)

thm *inext-less-mono-rev*
lemma *iprev-less-mono-rev*:
⟦ *iprev a I* < *iprev b I*; *a* ∈ *I*; *b* ∈ *I* ⟧ ⟹ *a* < *b*
apply (*rule ccontr*, *simp add*: *linorder-not-less*)
by (*metis iprev-le-mono less-le-not-le*)

lemma *set-restriction-inext-eq*:
⟦ *set-restriction interval-fun*; *n* ∈ *interval-fun I*; *inext n I* ∈ *interval-fun I* ⟧ ⟹

inext n (*interval-fun I*) = *inext n I*
apply (*subgoal-tac n* ∈ *I*)
prefer *2*
apply (*blast intro*: *set-restriction-in-imp*)
apply (*case-tac inext n I* = *n*)
apply *simp*
thm *inext-fix-iff*
thm *inext-fix-iff*[*THEN iffD1*]
apply (*frule inext-fix-iff*[*THEN iffD2*], *clarsimp*)
apply (*frule set-restriction-finite*, *assumption*)
apply (*subgoal-tac Max* (*interval-fun I*) = *Max I*)
prefer *2*
apply (*blast intro*: *Max-equality Max-ge set-restriction-in-imp*)
thm *inext-fix-iff*
apply (*blast intro*: *inext-fix-iff*[*THEN iffD1*])
thm *inext-mono*
thm *le-neq-implies-less*[*OF inext-mono*, *OF not-sym*]
apply (*drule le-neq-implies-less*[*OF inext-mono*, *OF not-sym*])
apply (*rule between-empty-imp-inext-eq*, *assumption*+)
thm *not-ex-in-conv*
apply (*simp add*: *not-ex-in-conv*[*symmetric*] *i-cut-mem-iff*)
by (*metis inext-min-step2 set-restriction-in-imp*)

thm *set-restriction-inext-eq*
lemma *set-restriction-inext-singleton-eq*:
⟦ *set-restriction interval-fun*; *n* ∈ *interval-fun I*; *inext n I* ∈ *interval-fun I* ⟧ ⟹

{*inext n* (*interval-fun I*)} = *interval-fun* {*inext n I*}
apply (*case-tac n* ∉ *I*)
apply (*blast dest*: *set-restriction-not-in-imp*)
apply (*frule set-restrictionD*, *erule exE*, *rename-tac P*)

apply (*simp add*: *singleton-iff set-eq-iff*)
by (*metis set-restriction-inext-eq*)

lemma *inext-iprev*: *iMin I* $\neq$ *n* $\Longrightarrow$ *inext* (*iprev n I*) *I* = *n*
apply (*case-tac n* $\notin$ *I*)
apply (*simp add*: *inext-def iprev-def*)
apply *simp*
apply (*frule iMin-neq-imp-greater*[*OF - not-sym*], *assumption*)
thm *iMinI iprev-min-step*
thm *min-step-inext iprev-mono2 iprev-closed*
apply (*blast dest*: *iMinI iprev-min-step intro*: *min-step-inext iprev-mono2 iprev-closed*)
done

lemma *iprev-inext-infin*: *infinite I* $\Longrightarrow$ *iprev* (*inext n I*) *I* = *n*
apply (*case-tac n* $\notin$ *I*)
apply (*simp add*: *inext-def iprev-def*)
apply *simp*
by (*metis inext-in-iff inext-min-step2 inext-mono2-infin-fin min-step-iprev2*)

lemma *iprev-inext-fin*:
⟦ *finite I*; *n* $\neq$ *Max I* ⟧ $\Longrightarrow$ *iprev* (*inext n I*) *I* = *n*
apply (*case-tac n* $\notin$ *I*)
apply (*simp add*: *inext-def iprev-def*)
apply *simp*
by (*metis inext-in-iff inext-min-step2 inext-mono2-infin-fin min-step-iprev2*)

lemma *iprev-inext*:
n $\neq$ *Max I* $\vee$ *infinite I* $\Longrightarrow$ *iprev* (*inext n I*) *I* = *n*
by (*blast intro*: *iprev-inext-infin iprev-inext-fin*)

lemma *inext-eq-infin*:
⟦ *inext a I* = *inext b I*; *infinite I* ⟧ $\Longrightarrow$ *a* = *b*
thm *arg-cong*[**where** *f*=$\lambda x.$ *iprev x I*]
apply (*drule arg-cong*[**where** *f*=$\lambda x.$ *iprev x I*])
apply (*simp add*: *iprev-inext-infin*)
done
lemma *inext-eq-fin*:
⟦ *inext a I* = *inext b I*; *finite I*; *a* $\neq$ *Max I*; *b* $\neq$ *Max I* ⟧ $\Longrightarrow$ *a* = *b*
apply (*drule arg-cong*[**where** *f*=$\lambda x.$ *iprev x I*])
apply (*simp add*: *iprev-inext-fin*)
done
thm *inext-mono2-infin-fin*
lemma *inext-eq-infin-fin*:
⟦ *inext a I* = *inext b I*; *a* $\neq$ *Max I* $\wedge$ *b* $\neq$ *Max I* $\vee$ *infinite I* ⟧ $\Longrightarrow$ *a* = *b*
thm *inext-eq-fin inext-eq-infin*

by (*blast intro*: *inext-eq-fin inext-eq-infin*)+
lemma *inext-eq*:
⟦ *inext a I* = *inext b I*; ∃ *x*∈*I*. *a* < *x*; ∃ *x*∈*I*. *b* < *x* ⟧ ⟹ *a* = *b*
by (*metis iprev-inext not-le wellorder-Max-lemma*)

lemma *iprev-eq-if-neq-iMin*:
⟦ *iprev a I* = *iprev b I*; *iMin I* ≠ *a*; *iMin I* ≠ *b* ⟧ ⟹ *a* = *b*
apply (*drule arg-cong*[**where** *f*=λ*x*. *inext x I*])
apply (*simp add*: *inext-iprev*)
done
lemma *iprev-eq*:
⟦ *iprev a I* = *iprev b I*; ∃ *x*∈*I*. *x* < *a*; ∃ *x*∈*I*. *x* < *b* ⟧ ⟹ *a* = *b*
by (*metis iprev-eq-if-neq-iMin not-less-iMin*)

lemma *greater-imp-iprev-ge*: ⟦ *b* < *a*; *a* ∈ *I*; *b* ∈ *I* ⟧ ⟹ *b* ≤ *iprev a I*
apply (*rule ccontr*, *simp add*: *linorder-not-le*)
apply (*blast dest*: *iprev-min-step*)
done

lemma *inext-cut-less-conv*: *inext n I* < *t* ⟹ *inext n* (*I* ↓< *t*) = *inext n I*
thm *le-less-trans*[*OF inext-mono*]
apply (*frule le-less-trans*[*OF inext-mono*])
apply (*case-tac n* ∈ *I*)
apply (*simp add*: *inext-def*)
thm *i-cut-commute-disj*[*of op* ↓< *op* ↓>, *simplified*]
apply (*simp add*: *i-cut-commute-disj*[*of op* ↓< *op* ↓>] *cut-less-mem-iff*)
apply (*case-tac I* ↓> *n* ≠ {})
apply *simp*
apply (*metis cut-less-Min-eq cut-less-Min-not-empty*)
apply (*simp add*: *i-cut-empty*)
apply (*simp add*: *not-in-inext-fix cut-less-not-in-imp*)
done
lemma *inext-cut-le-conv*: *inext n I* ≤ *t* ⟹ *inext n* (*I* ↓≤ *t*) = *inext n I*
thm *nat-cut-le-less-conv inext-cut-less-conv*
by (*simp add*: *nat-cut-le-less-conv inext-cut-less-conv*)

lemma *inext-cut-greater-conv*: *t* < *n* ⟹ *inext n* (*I* ↓> *t*) = *inext n I*
apply (*case-tac n* ∈ *I*)
thm *cut-greater-mem-iff*[*THEN iffD2*, *OF conjI*]
apply (*frule cut-greater-mem-iff*[*THEN iffD2*, *OF conjI*], *simp*)
thm *i-cut-commute-disj*[*of op* ↓> *op* ↓>, *simplified*]
thm *cut-cut-greater*
apply (*simp add*: *inext-def i-cut-commute-disj*[*of op* ↓> *op* ↓>] *cut-cut-greater max-def*)

apply (*simp add*: *not-in-inext-fix cut-greater-not-in-imp*)
done
lemma *inext-cut-ge-conv*: $t \leq n \Longrightarrow inext\ n\ (I\ {\downarrow}{\geq}\ t) = inext\ n\ I$
apply (*case-tac* $t = 0$)
apply (*simp add*: *cut-ge-0-all*)
thm *nat-cut-greater-ge-conv*[*symmetric*]
apply (*simp add*: *nat-cut-greater-ge-conv*[*symmetric*] *inext-cut-greater-conv*)
done

lemmas *inext-cut-conv* =
inext-cut-less-conv inext-cut-le-conv
inext-cut-greater-conv inext-cut-ge-conv

lemma *iprev-cut-greater-conv*: $t < iprev\ n\ I \Longrightarrow iprev\ n\ (I\ {\downarrow}{>}\ t) = iprev\ n\ I$
thm *less-le-trans*[*OF - iprev-mono*]
apply (*frule less-le-trans*[*OF - iprev-mono*])
apply (*case-tac* $n \in I$)
apply (*simp add*: *iprev-def*)
thm *i-cut-commute-disj*[*of op* $\downarrow>$ *op* $\downarrow<$, *simplified*]
apply (*simp add*: *i-cut-commute-disj*[*of op* $\downarrow>$ *op* $\downarrow<$] *cut-greater-mem-iff*)
apply (*case-tac* $I\ {\downarrow}{<}\ n \neq \{\}$)
apply *simp*
apply (*metis cut-greater-Max-eq cut-greater-Max-not-empty nat-cut-less-finite*)
apply (*simp add*: *i-cut-empty*)
apply (*simp add*: *not-in-iprev-fix cut-greater-not-in-imp*)
done
lemma *iprev-cut-ge-conv*: $t \leq iprev\ n\ I \Longrightarrow iprev\ n\ (I\ {\downarrow}{\geq}\ t) = iprev\ n\ I$
apply (*case-tac* $t = 0$)
apply (*simp add*: *cut-ge-0-all*)
thm *nat-cut-greater-ge-conv*
apply (*simp add*: *nat-cut-greater-ge-conv*[*symmetric*] *iprev-cut-greater-conv*)
done
lemma *iprev-cut-less-conv*: $n < t \Longrightarrow iprev\ n\ (I\ {\downarrow}{<}\ t) = iprev\ n\ I$
apply (*case-tac* $n \in I$)
thm *cut-less-mem-iff*[*THEN iffD2*, *OF conjI*]
apply (*frule cut-less-mem-iff*[*THEN iffD2*, *OF conjI*], *simp*)
thm *i-cut-commute-disj*[*of op* $\downarrow<$ *op* $\downarrow<$, *simplified*]
apply (*simp add*: *iprev-def i-cut-commute-disj*[*of op* $\downarrow<$ *op* $\downarrow<$] *cut-cut-less min-def*)
apply (*simp add*: *not-in-iprev-fix cut-less-not-in-imp*)
done
lemma *iprev-cut-le-conv*: $n \leq t \Longrightarrow iprev\ n\ (I\ {\downarrow}{\leq}\ t) = iprev\ n\ I$
thm *nat-cut-le-less-conv iprev-cut-less-conv*
by (*simp add*: *nat-cut-le-less-conv iprev-cut-less-conv*)

lemmas *iprev-cut-conv* =
iprev-cut-less-conv iprev-cut-le-conv
iprev-cut-greater-conv iprev-cut-ge-conv

thm
inext-cut-conv
iprev-cut-conv

thm *inext-cut-less-conv*
lemma *inext-cut-less-fix*: $t \leq inext\ n\ I \Longrightarrow inext\ n\ (I \downarrow< t) = n$
apply (*case-tac* $n \in I$)
prefer *2*
thm *contra-subsetD*[*OF cut-less-subset*]
apply (*frule contra-subsetD*[*OF cut-less-subset*[*of* - *t*]])
apply (*simp add*: *not-in-inext-fix*)
apply (*case-tac* $t \leq n$)
apply (*metis cut-less-mem-iff not-in-inext-fix not-le*)
apply (*rule-tac* $t{=}n$ **and** $s{=}Max\ (I \downarrow< t)$ **in** *subst*)
apply (*rule Max-equality*[*OF* - *nat-cut-less-finite*])
apply (*simp add*: *cut-less-mem-iff*)
apply (*rule ccontr*)
apply (*clarsimp simp*: *cut-less-mem-iff linorder-not-le*)
thm *inext-min-step*
apply (*simp add*: *inext-min-step*)
thm *inext-Max nat-cut-less-finite*
apply (*blast intro*: *inext-Max nat-cut-less-finite*)
done
lemma *inext-cut-le-fix*: $t < inext\ n\ I \Longrightarrow inext\ n\ (I \downarrow\leq t) = n$
thm *nat-cut-le-less-conv*
by (*simp add*: *nat-cut-le-less-conv inext-cut-less-fix*)

lemma *iprev-cut-greater-fix*: $iprev\ n\ I \leq t \Longrightarrow iprev\ n\ (I \downarrow> t) = n$
apply (*case-tac* $n \in I$)
prefer *2*
thm *contra-subsetD*[*OF cut-greater-subset*]
apply (*frule contra-subsetD*[*OF cut-greater-subset*[*of* - *t*]])
apply (*simp add*: *not-in-iprev-fix*)
apply (*case-tac* $n \leq t$)
apply (*metis cut-greater-mem-iff not-in-iprev-fix not-le*)
apply (*rule-tac* $t{=}n$ **and** $s{=}iMin\ (I \downarrow> t)$ **in** *subst*)
apply (*rule iMin-equality*)
apply (*simp add*: *cut-greater-mem-iff*)
apply (*metis cut-greater-mem-iff iprev-min-step2 not-leE order-le-less-trans*)
thm *iprev-iMin*
apply (*rule iprev-iMin*)
done
lemma *iprev-cut-ge-fix*: $iprev\ n\ I < t \Longrightarrow iprev\ n\ (I \downarrow\geq t) = n$
apply (*case-tac* $t = 0$)
apply (*simp add*: *cut-ge-0-all*)
thm *nat-cut-greater-ge-conv*[*symmetric*] *iprev-cut-greater-fix*
apply (*simp add*: *nat-cut-greater-ge-conv*[*symmetric*] *iprev-cut-greater-fix*)

done

definition
CommuteWithIntervalCut4 :: $((\prime a::linorder)\ set \Rightarrow \prime a\ set) \Rightarrow bool$
where
CommuteWithIntervalCut4 fun $\equiv$
$\forall\, t\ fun2\ I.$
$(fun2 = (\lambda I.\ I \downarrow< t) \vee fun2 = (\lambda I.\ I \downarrow\leq t) \vee fun2 = (\lambda I.\ I \downarrow> t) \vee fun2 = (\lambda I.\ I \downarrow\geq t)\ ) \longrightarrow$
$fun\ (fun2\ I) = fun2\ (fun\ I)$
definition *CommuteWithIntervalCut2* :: $((\prime a::linorder)\ set \Rightarrow \prime a\ set) \Rightarrow bool$
where
CommuteWithIntervalCut2 fun $\equiv$
$\forall\, t\ fun2\ I.$
$(fun2 = (\lambda I.\ I \downarrow< t) \vee fun2 = (\lambda I.\ I \downarrow> t)) \longrightarrow$
$fun\ (fun2\ I) = fun2\ (fun\ I)$

lemma *CommuteWithIntervalCut4-imp-2*: *CommuteWithIntervalCut4 fun* $\Longrightarrow$ *CommuteWithIntervalCut2 fun*
unfolding *CommuteWithIntervalCut2-def CommuteWithIntervalCut4-def* **by** *blast*

lemma *nat-CommuteWithIntervalCut2-4-eq*:
CommuteWithIntervalCut4 $(fun::nat\ set \Rightarrow nat\ set) =$ *CommuteWithIntervalCut2 fun*
apply (*unfold CommuteWithIntervalCut2-def CommuteWithIntervalCut4-def*)
apply (*rule iffI*)
apply *blast*
apply *clarify*
apply (*case-tac fun2* $= (\lambda I.\ I \downarrow< t)$, *simp*)
apply (*case-tac fun2* $= (\lambda I.\ I \downarrow> t)$, *simp*)
apply *simp*
apply (*erule disjE*)
apply (*simp add*: *nat-cut-le-less-conv*)
apply (*case-tac* $t = 0$)
apply (*simp add*: *cut-ge-0-all*)
apply (*simp add*: *nat-cut-greater-ge-conv*[*symmetric*])
done

lemma
cut-less-CommuteWithIntervalCut4: *CommuteWithIntervalCut4* $(\lambda I.\ I \downarrow< t)$
and
cut-le-CommuteWithIntervalCut4: *CommuteWithIntervalCut4* $(\lambda I.\ I \downarrow\leq t)$
and
cut-greater-CommuteWithIntervalCut4: *CommuteWithIntervalCut4* $(\lambda I.\ I \downarrow> t)$
and
cut-ge-CommuteWithIntervalCut4: *CommuteWithIntervalCut4* $(\lambda I.\ I \downarrow\geq t)$
thm *i-cut-commute-disj*
unfolding *CommuteWithIntervalCut4-def* **by** (*simp-all add*: *i-cut-commute-disj*)
lemmas *i-cut-CommuteWithIntervalCut4* =

cut-less-CommuteWithIntervalCut4 cut-le-CommuteWithIntervalCut4
cut-greater-CommuteWithIntervalCut4 cut-ge-CommuteWithIntervalCut4

thm *cut-greater-image*
lemma *inext-image*:
⟦ $n \in I$; *strict-mono-on f I* ⟧ ⟹ *inext* (*f n*) (*f* ‘ *I*) = *f* (*inext n I*)
apply (*case-tac* ∃ $x{\in}I.\ n < x$)
thm *inext-mono2*
apply (*frule inext-mono2*, *assumption*)
thm *cut-greater-not-empty-iff*[*THEN iffD2*]
apply (*frule cut-greater-not-empty-iff*[*THEN iffD2*])
apply (*simp add*: *inext-def image-iff*)
apply (*subgoal-tac* ∃ $x{\in}I.\ f\ n = f\ x$)
prefer *2*
apply *blast*
thm *cut-greater-image*
apply (*simp add*: *cut-greater-image*)
thm *iMin-mono-on2*[*OF strict-mono-on-imp-mono-on*]
thm *strict-mono-on-subset*
apply (*blast intro*: *strict-mono-on-subset iMin-mono-on2 strict-mono-on-imp-mono-on*)
apply (*drule strict-mono-on-imp-mono-on*)
thm *inext-all-le-fix*
apply (*simp add*: *inext-all-le-fix linorder-not-less mono-on-def*)
done

lemma *iprev-image*:
⟦ $n \in I$; *strict-mono-on f I* ⟧ ⟹ *iprev* (*f n*) (*f* ‘ *I*) = *f* (*iprev n I*)
apply (*case-tac* ∃ $x{\in}I.\ x < n$)
thm *iprev-mono2*
apply (*frule iprev-mono2*, *assumption*)
thm *cut-less-not-empty-iff*[*THEN iffD2*]
apply (*frule cut-less-not-empty-iff*[*THEN iffD2*])
apply (*simp add*: *iprev-def image-iff*)
apply (*subgoal-tac* ∃ $x{\in}I.\ f\ n = f\ x$)
prefer *2*
apply *blast*
thm *cut-less-image*
apply (*simp add*: *cut-less-image*)
thm *Max-mono-on2*[*OF strict-mono-on-imp-mono-on*]
thm *strict-mono-on-subset*
thm *nat-cut-less-finite*
apply (*blast intro*: *strict-mono-on-subset Max-mono-on2 strict-mono-on-imp-mono-on nat-cut-less-finite*)

apply (*drule strict-mono-on-imp-mono-on*)
thm *inext-all-le-fix*
apply (*simp add*: *iprev-all-ge-fix linorder-not-less mono-on-def*)
done

lemma *inext-image2*:
strict-mono f $\Longrightarrow$ *inext* (*f n*) (*f* ' *I*) = *f* (*inext n I*)
apply (*case-tac n* $\in$ *I*)
apply (*blast intro*: *strict-mono-imp-strict-mono-on inext-image*)
thm *inj-image-mem-iff strict-mono-imp-inj*
apply (*simp add*: *not-in-inext-fix inj-image-mem-iff strict-mono-imp-inj*)
done
lemma *iprev-image2*:
strict-mono f $\Longrightarrow$ *iprev* (*f n*) (*f* ' *I*) = *f* (*iprev n I*)
apply (*case-tac n* $\in$ *I*)
apply (*blast intro*: *strict-mono-imp-strict-mono-on iprev-image*)
apply (*simp add*: *not-in-iprev-fix inj-image-mem-iff strict-mono-imp-inj*)
done

lemma *inext-imirror-iprev-conv*:
$[\![$ *finite I*; *n* $\leq$ *iMin I* + *Max I* $]\!] \Longrightarrow$
inext (*mirror-elem n I*) (*imirror I*) = *mirror-elem* (*iprev n I*) *I*
apply (*case-tac n* $\in$ *I*)
prefer *2*
thm *imirror-mem-conv*
apply (*simp add*: *not-in-iprev-fix not-in-inext-fix imirror-mem-conv*)
apply (*frule in-imp-not-empty*[*of* - *I*])
apply (*frule in-imp-mirror-elem-in*[*of* - *n*], *assumption*)
apply (*simp add*: *inext-def iprev-def*)
apply (*case-tac n* = *iMin I*)
thm *cut-less-Min-empty*
thm *mirror-elem-Min*
apply (*simp add*: *cut-less-Min-empty mirror-elem-Min*)
thm *imirror-Max*
apply (*subst imirror-Max*[*symmetric*], *assumption*)
thm *cut-greater-Max-empty*[*OF* - *order-refl*]
apply (*simp add*: *cut-greater-Max-empty imirror-finite*)
apply (*frule iMin-le*[*of n I*])
apply (*intro conjI impI*)
thm *imirror-cut-greater'*
apply (*simp add*: *imirror-cut-greater'*)
thm *imirror-bounds-iMin nat-cut-less-finite*
apply (*simp add*: *imirror-bounds-iMin nat-cut-less-finite cut-less-Min-eq*)
apply (*simp add*: *mirror-elem-def nat-mirror-def*)
apply (*simp add*: *imirror-cut-greater'*)

apply (*simp add*: *imirror-bounds-def*)
thm *cut-less-Min-not-empty*
apply (*simp add*: *cut-less-Min-not-empty*)
done
corollary *inext-imirror-iprev-conv'*:
⟦ *finite I*; *n* ∈ *I* ⟧ ⟹
inext (*mirror-elem n I*) (*imirror I*) = *mirror-elem* (*iprev n I*) *I*
thm *inext-imirror-iprev-conv*[*OF - trans-le-add2*[*OF Max-ge*]]
by (*simp add*: *inext-imirror-iprev-conv trans-le-add2*)

lemma *iprev-imirror-inext-conv*:
⟦ *finite I*; *n* ≤ *iMin I* + *Max I* ⟧ ⟹
iprev (*mirror-elem n I*) (*imirror I*) = *mirror-elem* (*inext n I*) *I*
apply (*case-tac n* ∈ *I*)
prefer *2*
thm *imirror-mem-conv*
apply (*simp add*: *not-in-iprev-fix not-in-inext-fix imirror-mem-conv*)
apply (*frule in-imp-not-empty*[*of - I*])
apply (*frule in-imp-mirror-elem-in*[*of - n*], *assumption*)
apply (*simp add*: *inext-def iprev-def*)
apply (*case-tac n* = *Max I*)
thm *cut-greater-Max-empty*
thm *mirror-elem-Max*
apply (*simp add*: *cut-greater-Max-empty mirror-elem-Max*)
thm *imirror-iMin*
apply (*subst imirror-iMin*[*symmetric*], *assumption*)
thm *cut-less-Min-empty*[*OF order-refl*]
apply (*simp add*: *cut-less-Min-empty imirror-finite*)
thm *Max-ge*[*of I n*]
apply (*frule Max-ge*[*of I n*], *assumption*)
apply (*drule le-neq-trans*, *assumption*)
apply (*intro conjI impI*)
thm *imirror-cut-less*
apply (*simp add*: *imirror-cut-less*)
thm *imirror-bounds-Max cut-greater-finite cut-greater-Max-eq*
thm *imirror-bounds-Max*[*OF cut-greater-finite*]
thm *Max-le-iff*
apply (*simp add*: *imirror-bounds-Max cut-greater-finite cut-greater-Max-eq del*: *Max-le-iff*)
apply (*simp add*: *mirror-elem-def nat-mirror-def*)
apply (*simp add*: *imirror-cut-less*)
apply (*simp add*: *imirror-bounds-def*)
thm *cut-greater-Max-not-empty*[*of I n*]
apply (*simp add*: *cut-greater-Max-not-empty*)
done
corollary *iprev-imirror-inext-conv'*:
⟦ *finite I*; *n* ∈ *I* ⟧ ⟹
iprev (*mirror-elem n I*) (*imirror I*) = *mirror-elem* (*inext n I*) *I*
by (*simp add*: *iprev-imirror-inext-conv trans-le-add2*)

thm
inext-imirror-iprev-conv
inext-imirror-iprev-conv'
iprev-imirror-inext-conv
iprev-imirror-inext-conv'

lemma *inext-insert-ge-Max*:
⟦ *finite I*; *I* ≠ {}; *Max I* ≤ *a* ⟧ ⟹ *inext* (*Max I*) (*insert a I*) = *a*
apply (*case-tac a* = *Max I*)
apply (*simp add*: *insert-absorb inext-Max*)
thm *le-neq-trans*
apply (*drule le-neq-trans*, *simp*)
apply (*rule min-step-inext2*)
apply (*simp*, *simp*, *simp*)
apply (*simp-all*, *blast?*)
done

lemma *iprev-insert-le-iMin*:
⟦ *finite I*; *I* ≠ {}; *a* ≤ *iMin I* ⟧ ⟹ *iprev* (*iMin I*) (*insert a I*) = *a*
apply (*case-tac a* = *iMin I*)
apply (*simp add*: *iMinI-ex2 insert-absorb iprev-iMin*)
thm *le-neq-trans*
apply (*drule le-neq-trans*, *simp*)
apply (*rule min-step-iprev2*)
apply (*simp-all add*: *iMin-Min-conv*, *blast?*)
done

thm *cut-le-less-conv*
lemma *cut-less-le-iprev-conv*:
⟦ *t* ∈ *I*; *t* ≠ *iMin I* ⟧ ⟹ *I* ↓< *t* = *I* ↓≤ (*iprev t I*)
apply (*unfold iprev-def*)
apply (*rule set-eqI*, *safe*)
apply (*simp add*: *i-cut-defs*)
apply *simp*
apply (*split split-if-asm*)
thm *Max-ge-iff nat-cut-less-finite*
apply (*simp add*: *Max-ge-iff nat-cut-less-finite*)
apply (*blast intro*: *le-less-trans*)
apply (*frule iMin-neq-imp-greater*, *assumption*)
apply (*blast intro*: *iMin-in*)
done

lemma *neq-Max-imp-inext-neq-iMin*:
⟦ *t* ∈ *I*; *t* ≠ *Max I* ∨ *infinite I* ⟧ ⟹ *inext t I* ≠ *iMin I*
apply (*case-tac finite I*)
apply (*metis inext-mono2-infin-fin not-less-iMin*)

apply (*blast dest*: *inext-neq-iMin-infin*)
done
corollary *neq-Max-imp-inext-gr-iMin*:
⟦ $t \in I$; $t \neq Max\ I \vee infinite\ I$⟧ ⟹ $iMin\ I < inext\ t\ I$
apply (*frule neq-Max-imp-inext-neq-iMin*[*THEN not-sym*], *assumption*)
apply (*drule neq-le-trans*)
thm *inext-closed*
apply (*blast dest*: *inext-closed*)
apply *simp*
done

lemma *cut-le-less-inext-conv*:
⟦ $t \in I$; $t \neq Max\ I \vee infinite\ I$⟧ ⟹ $I\ {\downarrow}{\leq}\ t = I\ {\downarrow}{<}\ (inext\ t\ I)$
thm *cut-less-le-iprev-conv*[*of inext t I I*]
apply (*cut-tac cut-less-le-iprev-conv*[*of inext t I I*])
thm *iprev-inext*[*of t I*]
apply (*cut-tac iprev-inext*[*of t I*], *simp*)
apply *assumption*
apply (*rule inext-closed*, *assumption*)
apply (*rule neq-Max-imp-inext-neq-iMin*, *assumption*+)
done
lemma *cut-ge-greater-iprev-conv*:
⟦ $t \in I$; $t \neq iMin\ I$ ⟧ ⟹ $I\ {\downarrow}{\geq}\ t = I\ {\downarrow}{>}\ (iprev\ t\ I)$
apply (*frule iMin-neq-imp-greater*, *simp*+)
apply (*unfold iprev-def*)
apply (*rule set-eqI*, *safe*)
apply (*simp add*: *i-cut-defs linorder-not-less*)
apply (*drule iMinI*, *fastsimp*)
apply (*split split-if-asm*)
apply (*rule ccontr*)
apply (*simp add*: *nat-cut-less-finite linorder-not-le*)
apply *blast*
apply *simp*
done
lemma *cut-greater-ge-inext-conv*:
⟦ $t \in I$; $t \neq Max\ I \vee infinite\ I$ ⟧ ⟹ $I\ {\downarrow}{>}\ t = I\ {\downarrow}{\geq}\ (inext\ t\ I)$
thm *cut-ge-greater-iprev-conv*[*of inext t I I*]
apply (*cut-tac cut-ge-greater-iprev-conv*[*of inext t I I*])
thm *iprev-inext*[*of t I*]
apply (*cut-tac iprev-inext*[*of t I*], *simp*)
apply *blast*
apply (*rule inext-closed*, *assumption*)
apply (*rule neq-Max-imp-inext-neq-iMin*, *assumption*+)
done

thm
cut-less-le-iprev-conv
cut-le-less-inext-conv
cut-ge-greater-iprev-conv

cut-greater-ge-inext-conv

lemma *inext-append*:
⟦ *finite A*; *A* ≠ {}; *B* ≠ {}; *Max A* < *iMin B* ⟧ ⟹
inext n (*A* ∪ *B*) = (*if n* ∈ *B then inext n B else* (*if n* = *Max A then iMin B else inext n A*))
apply (*case-tac n* ∈ *A* ∪ *B*)
prefer *2*
apply (*simp add*: *not-in-inext-fix*)
apply (*blast dest*: *Max-in*)
apply (*frule Max-less-iMin-imp-disjoint*, *assumption*)
apply (*drule Un-iff*[*THEN iffD1*], *elim disjE*)
apply (*drule disjoint-iff-in-not-in1*[*THEN iffD1*])
apply *simp*
apply (*intro conjI impI*)
apply (*simp add*: *inext-def cut-greater-Un cut-greater-Max-empty cut-greater-Min-all*)
apply (*frule Max-neq-imp-less*[*of A*], *simp*+)
apply (*simp add*: *inext-def cut-greater-Un cut-greater-Min-all*)
apply (*subgoal-tac A* ↓> *n* ≠ {})
prefer *2*
apply (*simp add*: *cut-greater-not-empty-iff*)
apply (*blast intro*: *Max-in*)
apply (*simp add*: *iMin-Un*)
apply (*drule iMin-in*[*THEN cut-greater-in-imp*])
apply (*rule min-eqL*)
apply (*rule less-imp-le*)
apply *blast*
apply (*drule disjoint-iff-in-not-in2*[*THEN iffD1*])
apply *simp*
apply (*subgoal-tac A* ↓> *n* = {})
prefer *2*
apply (*simp add*: *cut-greater-empty-iff*)
apply *fastsimp*
apply (*simp add*: *inext-def cut-greater-Un*)
done
corollary *inext-append-eq1*:
⟦ *finite A*; *A* ≠ {}; *B* ≠ {}; *Max A* < *iMin B*; *n* ∈ *A*; *n* ≠ *Max A* ⟧ ⟹
inext n (*A* ∪ *B*) = *inext n A*
apply (*frule Max-less-iMin-imp-disjoint*, *assumption*)
apply (*drule disjoint-iff-in-not-in1*[*THEN iffD1*])
apply (*simp add*: *inext-append Max-less-iMin-imp-disjoint*)
done
corollary *inext-append-eq2*:

⟦ *finite A*; *A* ≠ {}; *B* ≠ {}; *Max A* < *iMin B*; *n* ∈ *B* ⟧ ⟹
inext n (*A* ∪ *B*) = *inext n B*
by (*simp add*: *inext-append*)
corollary *inext-append-eq3*:
⟦ *finite A*; *A* ≠ {}; *B* ≠ {}; *Max A* < *iMin B* ⟧ ⟹
inext (*Max A*) (*A* ∪ *B*) = *iMin B*
by (*simp add*: *inext-append not-less-iMin*)

lemma *iprev-append*: ⟦ *finite A*; *A* ≠ {}; *B* ≠ {}; *Max A* < *iMin B* ⟧ ⟹
iprev n (*A* ∪ *B*) = (*if n* ∈ *A then iprev n A else* (*if n* = *iMin B then Max A else iprev n B*))
apply (*case-tac n* ∈ *A* ∪ *B*)
prefer *2*
apply (*simp add*: *not-in-iprev-fix*)
apply (*blast intro*: *iMin-in*)
apply (*frule Max-less-iMin-imp-disjoint*, *assumption*)
apply (*drule Un-iff*[*THEN iffD1*], *elim disjE*)
apply (*drule disjoint-iff-in-not-in1*[*THEN iffD1*])
apply *simp*
apply (*subgoal-tac B* ↓< *n* = {})
prefer *2*
apply (*simp add*: *cut-less-empty-iff*)
apply *fastsimp*
apply (*simp add*: *iprev-def cut-less-Un*)
apply (*drule disjoint-iff-in-not-in2*[*THEN iffD1*])
apply *simp*
apply (*intro conjI impI*)
apply (*simp add*: *iprev-def cut-less-Un cut-less-Min-empty cut-less-Max-all*)
apply (*frule iMin-neq-imp-greater*[*of - B*], *simp+*)
apply (*simp add*: *iprev-def cut-less-Un*)
apply (*subgoal-tac A* ↓< *n* = *A*)
prefer *2*
apply (*simp add*: *cut-less-all-iff*)
apply *fastsimp*
apply (*subgoal-tac B* ↓< *n* ≠ {})
prefer *2*
apply (*simp add*: *cut-less-not-empty-iff*)
apply (*blast intro*: *iMin-in*)
apply (*simp add*: *Max-Un nat-cut-less-finite*)
apply (*rule max-eqR*)
apply (*rule less-imp-le*)
thm *Max-in*[*OF nat-cut-less-finite*, *THEN cut-less-in-imp*]
apply (*drule Max-in*[*OF nat-cut-less-finite*, *THEN cut-less-in-imp*])
apply (*blast intro*: *iMin-le Max-in order-less-le-trans*)
done

corollary *iprev-append-eq1*:
⟦ *finite A*; *A* ≠ {}; *B* ≠ {}; *Max A* < *iMin B*; *n* ∈ *A* ⟧ ⟹
iprev n (*A* ∪ *B*) = *iprev n A*

by (*simp add*: *iprev-append*)
corollary *iprev-append-eq2*:
⟦ *finite A*; *A* ≠ {}; *B* ≠ {}; *Max A* < *iMin B*; *n* ∈ *B*; *n* ≠ *iMin B* ⟧ ⟹
iprev n (*A* ∪ *B*) = *iprev n B*
apply (*frule Max-less-iMin-imp-disjoint*, *assumption*)
apply (*drule disjoint-iff-in-not-in2*[*THEN iffD1*])
apply (*simp add*: *iprev-append*)
done
corollary *iprev-append-eq3*:
⟦ *finite A*; *A* ≠ {}; *B* ≠ {}; *Max A* < *iMin B* ⟧ ⟹
iprev (*iMin B*) (*A* ∪ *B*) = *Max A*
by (*simp add*: *iprev-append not-greater-Max*[*of* - *iMin B*])

lemma *inext-predicate-change-exists-aux*: ⋀*a*.
⟦ *c* = *card* (*I* ↓≥ *a* ↓< *b*); *a* < *b*; *a* ∈ *I*; *b* ∈ *I*; ¬ *P a*; *P b* ⟧ ⟹
∃ *n* ∈ (*I* ↓≥ *a* ↓< *b*). ¬ *P n* ∧ *P* (*inext n I*)
apply (*subgoal-tac* 0 < *c*)
prefer 2
apply *clarify*
apply (*rule-tac x*=*a* **in** *not-empty-card-gr0-conv*[*OF nat-cut-less-finite*, *THEN iffD1*, *OF in-imp-not-empty*, *rule-format*])
apply (*simp add*: *i-cut-mem-iff*)
apply (*induct c*)
apply *simp*
apply (*subgoal-tac a* < *inext a I*)
prefer 2
apply (*blast intro*: *inext-mono2*)
apply (*drule-tac x*=*inext a I* **in** *meta-spec*)
thm *less-imp-inext-le*[*of* - *b I*]
apply (*frule less-imp-inext-le*[*of* - *b I*], *assumption*+)
apply (*case-tac inext a I* < *b*)
prefer 2
apply *simp*
apply (*subgoal-tac I* ↓≥ *a* ↓< *b* = {*a*})
prefer 2
apply (*simp add*: *set-eq-iff i-cut-mem-iff*, *clarify*)
apply (*rule iffI*)
prefer 2
apply *simp*
apply *clarify*
apply (*case-tac a* < *x*)
apply (*simp add*: *inext-min-step*)
apply *simp*+
apply (*subgoal-tac I* ↓≥ *inext a I* = *I* ↓> *a*)
prefer 2
apply (*rule cut-greater-ge-inext-conv*[*symmetric*], *assumption*)

apply (*case-tac finite I*)
apply (*simp*, *rule less-imp-neq*)
thm *Max-gr-iff*[*OF - in-imp-not-empty*]
apply (*simp add*: *Max-gr-iff in-imp-not-empty*)
apply (*blast intro*: *inext-closed*)
apply *simp*
apply (*simp add*: *inext-closed*)
apply (*subgoal-tac* $a \notin (I \downarrow> a \downarrow< b)$)
prefer *2*
apply *blast*
apply (*subgoal-tac* $(I \downarrow\geq a \downarrow< b) = insert\ a\ (I \downarrow> a \downarrow< b)$)
prefer *2*
apply (*simp add*:
i-cut-commute-disj[*of op* $\downarrow\geq$ *op* $\downarrow<$] *i-cut-commute-disj*[*of op* $\downarrow>$ *op* $\downarrow<$])
apply (*simp add*: *cut-ge-greater-conv-if i-cut-mem-iff*)
apply (*simp add*: *card-insert-disjoint*[*OF nat-cut-less-finite*])
apply (*case-tac P* (*inext a I*))
apply *blast*
apply (*case-tac card* $(I \downarrow> a \downarrow< b) = 0$)
apply (*drule card-0-eq*[*OF nat-cut-less-finite*, *THEN iffD1*])
apply (*simp add*: *cut-less-empty-iff*)
apply (*drule-tac x*=*inext a I* **in** *bspec*)
apply (*blast intro*: *inext-closed*)
apply *simp*
apply *simp*
done

lemma *inext-predicate-change-exists*:
$[\![\ a \leq b;\ a \in I;\ b \in I;\ \neg\ P\ a;\ P\ b\ ]\!] \Longrightarrow$
$\exists\, n{\in}I.\ a \leq n \wedge n < b \wedge \neg\ P\ n \wedge P\ (inext\ n\ I)$
apply (*drule order-le-less*[*THEN iffD1*], *erule disjE*)
prefer *2*
apply *blast*
thm *inext-predicate-change-exists-aux*[*OF refl*]
apply (*drule inext-predicate-change-exists-aux*[*OF refl*], *assumption*+)
apply *blast*
done

lemma *iprev-predicate-change-exists*:
$[\![\ a \leq b;\ a \in I;\ b \in I;\ \neg\ P\ b;\ P\ a\ ]\!] \Longrightarrow$
$\exists\, n{\in}I.\ a < n \wedge n \leq b \wedge \neg\ P\ n \wedge P\ (iprev\ n\ I)$
apply (*frule inext-predicate-change-exists*[*of a b I* $\lambda x.\ \neg\ P\ x$], *simp*+)
apply *clarify*
apply (*rule-tac x*=*inext n I* **in** *bexI*)
prefer *2*
apply (*blast intro*: *inext-closed*)
apply (*subgoal-tac* $n < inext\ n\ I$)
prefer *2*
apply (*blast intro*: *inext-mono2*)

apply (*frule-tac x=a* **and** *z=inext n I* **in** *le-less-trans*, *assumption*)
apply (*frule less-imp-inext-le*, *assumption+*)
apply (*cut-tac n=n* **and** *I=I* **in** *iprev-inext*)
apply (*case-tac finite I*)
apply *simp*
apply (*rule less-imp-neq*)
apply (*blast intro*: *inext-closed Max-ge order-less-le-trans*)
apply *simp+*
done

corollary *nat-Suc-predicate-change-exists*:
⟦ $a \leq b$; $\neg P\ a$; $P\ b$ ⟧ ⟹ $\exists n{\geq}a.\ n < b \wedge \neg P\ n \wedge P\ (Suc\ n)$
apply (*drule inext-predicate-change-exists*[*OF - UNIV-I UNIV-I*], *assumption+*)
apply (*simp add*: *inext-UNIV*)
done

corollary *nat-pred-predicate-change-exists*:
⟦ $a \leq b$; $\neg P\ b$; $P\ a$ ⟧ ⟹ $\exists n{\leq}b.\ a < n \wedge \neg P\ n \wedge P\ (n - Suc\ 0)$
apply (*drule iprev-predicate-change-exists*[*OF - UNIV-I UNIV-I*], *assumption+*)
apply (*fastsimp simp add*: *iprev-UNIV*)
done

lemma *inext-predicate-change-exists2-all*:
⟦ $(a::nat) \leq b$; $a \in I$; $b \in I$; $\neg P\ a$; $\forall k \in I \downarrow{\geq} b.\ P\ k$ ⟧ ⟹
$\exists n{\in}I.\ a \leq n \wedge n < b \wedge \neg P\ n \wedge (\forall k \in I \downarrow{>} n.\ P\ k)$
apply (*drule order-le-less*[*THEN iffD1*], *erule disjE*)
prefer *2*
apply *blast*
thm *inext-predicate-change-exists*[*of a b I* $\lambda n.$ *if* $(n = a)$ *then P n else* $(\forall k{\in}I{\downarrow}{\geq}n.$ *P k*)]
apply (*frule inext-predicate-change-exists*[*OF less-imp-le*,
of a b I $\lambda n.$ *if* $(n = a)$ *then P n else* $(\forall k{\in}I{\downarrow}{\geq}n.\ P\ k)$])
apply *simp+*
apply *clarify*
apply (*rule-tac x=n* **in** *bexI*)
prefer *2*
apply *assumption*
apply (*case-tac* $a < n$)
prefer *2*
apply *simp*
apply (*split split-if-asm*)
apply (*subgoal-tac* $I \downarrow> n = \{\}$, *simp+*)
apply (*drule not-sym*)
thm *cut-greater-ge-inext-conv*
apply (*rule ssubst*[*OF cut-greater-ge-inext-conv*])
apply *assumption*
apply (*case-tac finite I*)

prefer *2*
apply *simp*
apply *simp*
apply (*rule less-imp-neq*)
thm *inext-neq-imp-less*
apply (*drule inext-neq-imp-less*)
apply (*rule less-le-trans*[*OF - Max-ge*])
apply *assumption*+
apply (*subgoal-tac* $a < inext\ n\ I$)
prefer *2*
apply (*blast intro*: *inext-mono order-less-le-trans*)
apply (*subgoal-tac* $I \downarrow\geq inext\ n\ I = I \downarrow> n$)
prefer *2*
apply (*rule cut-greater-ge-inext-conv*[*symmetric*], *assumption*)
apply (*case-tac finite I*)
apply *simp*
apply (*rule less-imp-neq*)
apply (*blast intro*: *inext-closed Max-ge order-less-le-trans*)
apply *simp*
apply *simp*
thm *cut-greater-ge-conv-if*
apply (*simp add*: *cut-greater-ge-conv-if*)
apply *blast*
done

corollary *inext-predicate-change-exists2*:
$[\![(a::nat) \leq b;\ a \in I;\ b \in I;\ \neg\ P\ a;\ P\ b\ ]\!] \Longrightarrow$
$\exists\, n{\in}I.\ a \leq n \wedge n < b \wedge \neg\ P\ n \wedge (\forall\, k{\in}I.\ n < k \wedge k \leq b \longrightarrow P\ k)$
thm *inext-predicate-change-exists2-all*[*of a b I* $\downarrow\leq$ *b*]
apply (*frule inext-predicate-change-exists2-all*[*of a b I* $\downarrow\leq$ *b*])
apply (*simp add*: *i-cut-mem-iff*)+
apply *fastsimp*
apply *blast*
done

corollary *nat-Suc-predicate-change-exists2-all*:
$[\![(a::nat) \leq b;\ \neg\ P\ a;\ \forall\, k{\geq}b.\ P\ k\ ]\!] \Longrightarrow$
$\exists\, n{\geq}a.\ n < b \wedge \neg\ P\ n \wedge (\forall\, k{>}n.\ P\ k)$
apply (*drule inext-predicate-change-exists2-all*[*rule-format*, *OF - UNIV-I UNIV-I*])
apply (*simp add*: *i-cut-mem-iff Ball-def*)+
done
corollary *nat-Suc-predicate-change-exists2*:
$[\![(a::nat) \leq b;\ \neg\ P\ a;\ P\ b\ ]\!] \Longrightarrow$
$\exists\, n{\geq}a.\ n < b \wedge \neg\ P\ n \wedge (\forall\, k{\leq}b.\ n < k \longrightarrow P\ k)$
thm *inext-predicate-change-exists2*[*of a b UNIV*]
apply (*drule inext-predicate-change-exists2*[*of a b UNIV*])
apply *simp*+
apply *blast*
done

thm *inext-predicate-change-exists2-all*
lemma *iprev-predicate-change-exists2-all*:
⟦ (*a*::*nat*) ≤ *b*; *a* ∈ *I*; *b* ∈ *I*; ¬ *P b*; ∀ *k*∈*I*↓≤*a*. *P k* ⟧ ⟹
∃ *n*∈*I*. *a* < *n* ∧ *n* ≤ *b* ∧ ¬ *P n* ∧ (∀ *k*∈*I*↓<*n*. *P k*)
apply (*drule order-le-less*[*THEN iffD1*], *erule disjE*)
prefer *2*
apply *blast*
thm *iprev-predicate-change-exists*[*of a b I* λ*n*. *if* (*n* = *b*) *then P n else* (∀ *k*∈*I*↓≤*n*. *P k*)]
apply (*frule iprev-predicate-change-exists*[*OF less-imp-le*,
of a b I λ*n*. *if* (*n* = *b*) *then P n else* (∀ *k*∈*I*↓≤*n*. *P k*)])
apply *simp*+
apply *clarify*
apply (*rule-tac x*=*n* **in** *bexI*)
prefer *2*
apply *assumption*
apply (*case-tac a* < *n*)
prefer *2*
apply *simp*
apply *simp*
apply (*subgoal-tac iMin I* < *n*)
prefer *2*
apply (*blast intro*: *order-le-less-trans*)
apply (*split split-if-asm*)
apply *clarsimp*
apply (*split split-if-asm*)
apply *simp*
thm *cut-less-le-iprev-conv*[*symmetric*]
apply (*simp add*: *cut-less-le-iprev-conv*[*symmetric*])
apply *blast*
apply (*split split-if-asm*)
apply *simp*
apply (*simp add*: *cut-less-le-iprev-conv*[*symmetric*])
apply (*clarsimp*, *rename-tac x*)
apply (*case-tac x* < *n*)
apply *blast*
apply *simp*
done

thm *inext-predicate-change-exists2*
corollary *iprev-predicate-change-exists2*:
⟦ (*a*::*nat*) ≤ *b*; *a* ∈ *I*; *b* ∈ *I*; ¬ *P b*; *P a* ⟧ ⟹
∃ *n*∈*I*. *a* < *n* ∧ *n* ≤ *b* ∧ ¬ *P n* ∧ (∀ *k*∈*I*. *a* ≤ *k* ∧ *k* < *n* ⟶ *P k*)
thm *iprev-predicate-change-exists2-all*[*of a b I* ↓≥ *a*]
apply (*frule iprev-predicate-change-exists2-all*[*of a b I* ↓≥ *a*])
apply (*simp add*: *i-cut-mem-iff*)+

apply *fastsimp*
apply *blast*
done

thm *nat-Suc-predicate-change-exists2-all*
corollary *nat-pred-predicate-change-exists2-all*:
$[\![(a::nat) \leq b; \neg\ P\ b; \forall\, k{\leq}a.\ P\ k\ ]\!] \Longrightarrow$
$\exists\, n{>}a.\ n \leq b \wedge \neg\ P\ n \wedge (\forall\, k{<}n.\ P\ k)$
apply (*drule iprev-predicate-change-exists2-all*[*rule-format*, *OF - UNIV-I UNIV-I*])
apply (*simp add*: *i-cut-mem-iff Ball-def*)+
done

thm *nat-Suc-predicate-change-exists2*
corollary *nat-pred-predicate-change-exists2*:
$[\![(a::nat) \leq b; \neg\ P\ b; P\ a\ ]\!] \Longrightarrow$
$\exists\, n{>}a.\ n \leq b \wedge \neg\ P\ n \wedge (\forall\, k{\geq}a.\ k < n \longrightarrow P\ k)$
thm *iprev-predicate-change-exists2*[*of a b UNIV*]
apply (*drule iprev-predicate-change-exists2*[*of a b UNIV*])
apply *simp*+
apply *blast*
done

8.2 *inext-nth* and *iprev-nth* – nth element of a natural set

term *inext*
primrec
inext-nth :: *nat set* ⇒ *nat* ⇒ *nat*
where
inext-nth I 0 = *iMin I*
| *inext-nth I* (*Suc n*) = *inext* (*inext-nth I n*) *I*
notation (*xsymbols*)
inext-nth ((- → -) [*100*, *100*] *60*)
notation (*HTML* **output**)
inext-nth ((- → -) [*100*, *100*] *60*)

term $(I \rightarrow a) + b$
term $(I \rightarrow n) \in I$
term $(A \rightarrow n) + (B \rightarrow n)$

lemma *inext-nth-closed*: $I \neq \{\} \Longrightarrow I \rightarrow n \in I$
apply (*induct n*)
apply (*simp add*: *iMinI-ex2*)
apply (*simp add*: *inext-closed*)
done

thm *inext-image*
lemma *inext-nth-image*:

⟦ $I \neq \{\}$; *strict-mono-on f I* ⟧ $\Longrightarrow$ $(f\ `\ I) \rightarrow n = f\ (I \rightarrow n)$
apply (*induct n*)
apply (*simp add*: *iMin-mono-on2 strict-mono-on-imp-mono-on*)
apply (*simp add*: *inext-image inext-nth-closed*)
done

thm *inext-mono2*
lemma *inext-nth-Suc-mono*: $I \rightarrow n \leq I \rightarrow Suc\ n$
by (*simp add*: *inext-mono*)

lemma *inext-nth-mono*: $a \leq b \Longrightarrow I \rightarrow a \leq I \rightarrow b$
apply (*induct b*)
apply *simp*
apply (*drule le-Suc-eq*[*THEN iffD1*], *erule disjE*)
apply (*rule-tac y*=$I \rightarrow b$ **in** *order-trans*)
apply *simp*
apply (*rule inext-nth-Suc-mono*)
apply *simp*
done

thm *inext-mono2*
lemma *inext-nth-Suc-mono2*: $\exists\, x{\in}I.\ I \rightarrow n < x \Longrightarrow I \rightarrow n < I \rightarrow Suc\ n$
apply *simp*
apply (*rule inext-mono2*)
apply (*blast intro*: *inext-nth-closed inext-mono2*)+
done

lemma *inext-nth-mono2*: $\exists\, x{\in}I.\ I \rightarrow a < x \Longrightarrow (I \rightarrow a < I \rightarrow b) = (a < b)$
apply (*subgoal-tac* $I \neq \{\}$)
prefer *2*
apply *blast*
apply (*rule iffI*)
apply (*rule ccontr*)
apply (*simp add*: *linorder-not-less*)
apply (*drule inext-nth-mono*[*of - - I*])
apply *simp*
apply *clarify*
apply (*induct b*)
apply *blast*
apply (*drule less-Suc-eq*[*THEN iffD1*], *erule disjE*)
apply (*blast intro*: *order-less-le-trans inext-nth-Suc-mono*)
apply (*blast intro*: *inext-nth-Suc-mono2*)
done

lemma *inext-nth-mono2-infin*:
infinite I $\Longrightarrow$ $(I \rightarrow a < I \rightarrow b) = (a < b)$
apply (*drule infinite-nat-iff-unbounded*[*THEN iffD1*])
apply (*rule inext-nth-mono2*)
apply *blast*

done

lemma *inext-nth-Max-fix*:
⟦ *finite I*; *I* ≠ {}; *I* → *a* = *Max I*; *a* ≤ *b* ⟧ ⟹ *I* → *b* = *Max I*
apply (*induct b*)
apply *simp*
apply (*drule le-Suc-eq*[*THEN iffD1*], *erule disjE*)
apply (*simp add*: *inext-Max*)
apply *blast*
done

thm *inext-cut-less-conv*
lemma *inext-nth-cut-less-conv*:
⋀*I*. *I* → *n* < *t* ⟹ (*I* ↓< *t*) → *n* = *I* → *n*
apply (*case-tac I* = {})
apply (*simp add*: *cut-less-empty*)
apply (*induct n*)
apply (*simp add*: *cut-less-Min-eq cut-less-Min-not-empty*)
apply *simp*
thm *order-le-less-trans*[*OF inext-mono*]
apply (*frule order-le-less-trans*[*OF inext-mono*])
thm *inext-cut-less-conv*
apply (*simp add*: *inext-cut-less-conv*)
done
thm
inext-cut-less-conv
inext-nth-cut-less-conv

lemma *remove-Min-inext-nth-Suc-conv*: ⋀*I*.
Suc 0 < *card I* ∨ *infinite I* ⟹
(*I* − {*iMin I*}) → *n* = *I* → *Suc n*

apply (*subgoal-tac I* ≠ {})
prefer *2*
thm *card-gr0-imp-not-empty*[*OF gr-implies-gr0*]
apply (*blast dest*: *card-gr0-imp-not-empty*[*OF gr-implies-gr0*])
apply (*subgoal-tac I* − {*iMin I*} ≠ {})
prefer *2*
apply (*rule ccontr*, *simp*)
apply (*erule disjE*)
apply (*drule card-mono*[*OF singleton-finite*])
apply *simp*
apply (*simp add*: *subset-singleton-conv*)
apply (*blast dest*: *infinite-imp-nonempty infinite-imp-not-singleton*)
apply (*induct n*)
thm *cut-greater-Min-eq-Diff*[*symmetric*]
apply (*simp add*: *cut-greater-Min-eq-Diff*[*symmetric*] *inext-def iMinI-ex2*)

apply *simp*
thm *ssubst*[*OF inext-def*[*THEN meta-eq-to-obj-eq*], *rule-format*]
apply (*rule-tac n*=(*inext* (*I* → *n*) *I*) **in** *ssubst*[*OF inext-def*[*THEN meta-eq-to-obj-eq*], *rule-format*])
apply (*rule-tac n*=(*inext* (*I* → *n*) *I*) **in** *ssubst*[*OF inext-def*[*THEN meta-eq-to-obj-eq*], *rule-format*])
apply (*simp add*: *inext-closed inext-nth-closed*)
apply (*subgoal-tac inext* (*I* → *n*) *I* ≠ *iMin I*)
prefer *2*
apply (*erule disjE*)
thm *inext-neq-iMin-not-card-1*
thm *inext-neq-iMin-infin*
apply (*simp add*: *inext-neq-iMin-not-card-1 inext-neq-iMin-infin*)+
apply (*subgoal-tac iMin I* < (*I* → *Suc n*))
prefer *2*
thm *iMin-le*[*OF inext-nth-closed*, *rule-format*]
apply (*drule-tac n*=*Suc n* **in** *iMin-le*[*OF inext-nth-closed*, *rule-format*])
apply *simp*
apply (*simp add*: *cut-greater-Diff cut-greater-singleton*)
done

corollary *remove-Min-inext-nth-Suc-conv-finite*: *Suc 0* < *card I* ⟹ (*I* − {*iMin I*}) → *n* = *I* → *Suc n*
by (*simp add*: *remove-Min-inext-nth-Suc-conv*)
corollary *remove-Min-inext-nth-Suc-conv-infinite*: *infinite I* ⟹ (*I* − {*iMin I*}) → *n* = *I* → *Suc n*
by (*simp add*: *remove-Min-inext-nth-Suc-conv*)

lemma *remove-Max-eq*: ⟦ *finite I*; *I* ≠ {}; *n* ≠ *Max I* ⟧ ⟹ *Max* (*I* − {*n*}) = *Max I*
by (*rule Max-equality*, *simp*+)
lemma *remove-iMin-eq*: ⟦ *I* ≠ {}; *n* ≠ *iMin I* ⟧ ⟹ *iMin* (*I* − {*n*}) = *iMin I*
by (*rule iMin-equality*, *simp-all add*: *iMinI-ex2 iMin-le*)
lemma *remove-Min-eq*: ⟦ *finite I*; *I* ≠ {}; *n* ≠ *Min I* ⟧ ⟹ *Min* (*I* − {*n*}) = *Min I*
by (*rule Min-eqI*, *simp*+)
lemma *Max-le-iMin-conv-singleton*: ⟦ *finite I*; *I* ≠ {} ⟧ ⟹ (*Max I* ≤ *iMin I*) = (∃ *x*. *I* = {*x*})
by (*simp add*: *iMin-Min-conv Max-le-Min-conv-singleton del*: *Max-le-iff Min-ge-iff*)

lemma *inext-nth-card-less-Max*:
⋀*I*. *Suc n* < *card I* ⟹ *I* → *n* < *Max I*
thm *card-gr0-imp-not-empty*[*OF less-trans*[*OF zero-less-Suc*]]
apply (*frule card-gr0-imp-not-empty*[*OF less-trans*[*OF zero-less-Suc*]])
apply (*frule card-gr0-imp-finite*[*OF less-trans*[*OF zero-less-Suc*]])
apply (*induct n*)
apply (*rule ccontr*)

apply (*simp add*: *linorder-not-less iMin-Min-conv del*: *Max-le-iff Min-ge-iff*)
thm *Max-le-Min-conv-singleton*
apply (*drule Max-le-Min-conv-singleton*[*THEN iffD1*], *assumption*+)
apply *clarsimp*
apply (*drule-tac x*=*I* − {*iMin I*} **in** *meta-spec*)
thm *remove-Min-inext-nth-Suc-conv*
apply (*simp add*: *remove-Min-inext-nth-Suc-conv*)
apply (*subgoal-tac* ¬ *I* ⊆ {*iMin I*})
prefer *2*
apply (*rule ccontr*, *simp*)
apply (*drule card-mono*[*OF singleton-finite*])
apply *simp*
apply (*simp add*: *card-Diff-singleton iMin-in Suc-less-pred-conv*)
apply (*subgoal-tac Max I* ≠ *iMin I*)
prefer *2*
apply (*rule ccontr*, *simp*)
thm *Max-le-iMin-conv-singleton*[*THEN iffD1*]
apply (*frule Max-le-iMin-conv-singleton*[*THEN iffD1*], *clarsimp*+)
thm *remove-Max-eq*
apply (*simp add*: *remove-Max-eq Max-le-iMin-conv-singleton*)
done
thm *inext-nth-card-less-Max*
lemma *inext-nth-card-less-Max'*:
n < *card I* − *Suc 0* ⟹ *I* → *n* < *Max I*
by (*simp add*: *inext-nth-card-less-Max*)

lemma *inext-nth-card-Max-aux*:
⋀*I*. *card I* = *Suc n* ⟹ *I* → *n* = *Max I*
thm *card-gr0-imp-not-empty*[*OF less-le-trans*[*OF zero-less-Suc*, *OF eq-imp-le*[*OF sym*]]]
apply (*frule card-gr0-imp-not-empty*[*OF less-le-trans*[*OF zero-less-Suc*, *OF eq-imp-le*[*OF sym*]]])
apply (*frule card-gr0-imp-finite*[*OF less-le-trans*[*OF zero-less-Suc*, *OF eq-imp-le*[*OF sym*]]])
apply (*induct n*)
apply (*clarsimp simp*: *card-1-singleton-conv*)
apply *simp*
apply (*cut-tac I*=*I* **and** *t*=*Max I* **in** *nat-cut-less-finite*)
apply (*subgoal-tac card* (*I* ↓< *Max I*) = *Suc n*)
prefer *2*
apply (*simp add*: *cut-less-le-conv cut-le-Max-all*)
thm *card-gr0-imp-not-empty*[*OF less-le-trans*[*OF zero-less-Suc*, *OF eq-imp-le*[*OF sym*]], *rule-format*]
apply (*frule-tac n*=*n* **in** *card-gr0-imp-not-empty*[*OF less-le-trans*[*OF zero-less-Suc*, *OF eq-imp-le*[*OF sym*]], *rule-format*])
apply (*subgoal-tac Max* (*I* ↓< *Max I*) < *iMin* {*Max I*})
prefer *2*
apply (*simp*, *blast*)

apply (*subgoal-tac inext-nth I n < Max I*)
prefer *2*
thm *inext-nth-card-less-Max*
apply (*simp add*: *inext-nth-card-less-Max*)
apply (*frule inext-nth-cut-less-conv*[*symmetric*])
apply *simp*
apply (*rule min-step-inext*)
apply *simp*
apply (*rule subsetD*, *rule cut-less-subset*, *rule Max-in*, *assumption*+)
apply *simp*
apply (*frule-tac A=I ↓< Max I* **and** *k=k* **in** *not-greater-Max*, *assumption*)
apply (*simp add*: *cut-less-mem-iff*)
done
lemma *inext-nth-card-Max-aux'*:
⋀*I*. ⟦ *finite I*; *I* ≠ {} ⟧ ⟹ *I* → (*card I* − *Suc 0*) = *Max I*
by (*simp add*: *inext-nth-card-Max-aux not-empty-card-gr0-conv*)

thm
inext-nth-card-Max-aux
inext-nth-card-Max-aux'
thm
inext-nth-card-less-Max
inext-nth-Max-fix
lemma *inext-nth-card-Max*:
⟦ *finite I*; *I* ≠ {}; *card I* ≤ *Suc n* ⟧ ⟹ *I* → *n* = *Max I*
thm *inext-nth-Max-fix*[*of - card I − Suc 0*]
apply (*rule inext-nth-Max-fix*[*of - card I − Suc 0*], *assumption*+)
apply (*simp add*: *inext-nth-card-Max-aux'*)
apply *simp*
done
lemma *inext-nth-card-Max'*:
⟦ *finite I*; *I* ≠ {}; *card I* − *Suc 0* ≤ *n* ⟧ ⟹ *I* → *n* = *Max I*
by (*simp add*: *inext-nth-card-Max*)

thm
inext-nth-card-less-Max
inext-nth-card-Max
inext-nth-card-Max'

lemma *inext-nth-singleton*: {*a*} → *n* = *a*
thm *inext-nth-Max-fix*[*OF singleton-finite singleton-not-empty - le0*]
by (*simp add*: *inext-nth-Max-fix*[*OF singleton-finite singleton-not-empty - le0*])

lemma *inext-nth-eq-Min-conv*:
I ≠ {} ⟹ (*I* → *n* = *iMin I*) = (*n* = *0* ∨ (∃ *a*. *I* = {*a*}))
apply (*rule iffI*)
apply (*case-tac n*, *simp*)

apply (*rename-tac* n')
apply (*rule ccontr*)
apply (*drule-tac* $n{=}I \rightarrow n'$ **in** *inext-neq-iMin-not-singleton*, *simp*)
apply *simp*
apply (*erule disjE*, *simp*)
apply (*clarsimp simp*: *inext-nth-singleton*)
done

lemma *inext-nth-gr-Min-conv*:
$I \neq \{\} \Longrightarrow (iMin\ I < I \rightarrow n) = (0 < n \wedge \neg(\exists\, a.\ I = \{a\}))$
apply (*rule subst*[*of iMin* $I \neq I \rightarrow n$ *iMin* $I < I \rightarrow n$])
apply (*frule iMin-le*[*OF inext-nth-closed*[*of* - n]])
apply (*simp add*: *linorder-neq-iff*)
apply (*subst neq-commute*[*of iMin* I])
apply (*simp add*: *inext-nth-eq-Min-conv*)
done

lemma *inext-nth-gr-Min-conv-infinite*:
infinite $I \Longrightarrow (iMin\ I < I \rightarrow n) = (0 < n)$
by (*simp add*: *inext-nth-gr-Min-conv infinite-imp-nonempty infinite-imp-not-singleton*)

lemma *inext-nth-cut-ge-inext-nth*: $\bigwedge I\ b.$
$I \neq \{\} \Longrightarrow I \downarrow\geq (I \rightarrow a) \rightarrow b = I \rightarrow (a + b)$
apply (*induct a*)
apply (*simp add*: *cut-ge-Min-all*)
apply (*case-tac card* $I = Suc\ 0$)
apply (*drule card-1-imp-singleton*, *clarify*)
apply (*simp add*: *inext-nth-singleton inext-singleton cut-ge-Min-all*)
apply (*subgoal-tac Suc* $0 <$ *card* $I \vee$ *infinite* I)
prefer *2*
apply (*rule ccontr*, *clarsimp simp*: *linorder-not-less not-empty-card-gr0-conv*)
apply (*case-tac* $I - \{iMin\ I\} = \{\}$)
apply (*rule-tac* $t{=}I$ **and** $s{=}\{iMin\ I\}$ **in** *subst*, *blast*)
apply (*simp* (*no-asm*) *add*: *inext-nth-singleton inext-singleton cut-ge-Min-all*)
apply (*simp add*: *subset-singleton-conv*)
thm *remove-Min-inext-nth-Suc-conv*
apply (*drule-tac* $x{=}I - \{iMin\ I\}$ **in** *meta-spec*)
apply (*drule-tac* $x{=}b$ **in** *meta-spec*)
apply (*drule meta-mp*, *blast*)
thm *remove-Min-inext-nth-Suc-conv*
apply (*simp add*: *remove-Min-inext-nth-Suc-conv*)
apply (*simp add*: *cut-ge-Diff cut-ge-singleton*)
apply (*subgoal-tac iMin* $I <$ *inext* $(I \rightarrow a)\ I$, *simp*)
apply (*rule le-neq-trans*[*OF* - *not-sym*])
apply (*simp add*: *iMin-le inext-closed inext-nth-closed*)
apply (*erule disjE*)
apply (*simp add*: *inext-neq-iMin-not-card-1 inext-neq-iMin-infin*)+
done

thm *inext-append-eq1*
lemma *inext-nth-append-eq1*:
⟦ *finite A*; *A* ≠ {}; *Max A* < *iMin B*; *A* → *n* ≠ *Max A* ⟧ ⟹
(*A* ∪ *B*) → *n* = *A* → *n*
apply (*case-tac B* = {}, *simp*)
apply (*induct n*)
apply (*simp add*: *iMin-Un del*: *Max-less-iff*)
apply (*rule min-eq*)
thm *iMin-le-Max*
apply (*blast intro*: *order-less-imp-le order-le-less-trans iMin-le-Max*)
thm *Max-ge*[*OF - inext-nth-closed*]
apply (*frule-tac n*=*Suc n* **in** *Max-ge*[*OF - inext-nth-closed*, *rule-format*], *assumption*)
apply (*drule order-le-neq-trans*, *simp*+)
thm *order-le-less-trans*[*OF inext-mono*]
apply (*drule order-le-less-trans*[*OF inext-mono*])
thm *inext-append-eq1*
apply (*simp add*: *inext-append-eq1 inext-nth-closed*)
done

thm *inext-nth-append-eq1*
thm *inext-append-eq2*
lemma *inext-nth-card-append-eq1*:
⋀*A B*.⟦ *Max A* < *iMin B*; *n* < *card A* ⟧ ⟹
(*A* ∪ *B*) → *n* = *A* → *n*
apply (*case-tac B* = {}, *simp*)
thm *card-gr0-imp-finite*[*OF le-less-trans*[*OF le0*]]
apply (*frule card-gr0-imp-finite*[*OF le-less-trans*[*OF le0*]])
apply (*frule card-gr0-imp-not-empty*[*OF le-less-trans*[*OF le0*]])
apply (*drule Suc-leI*[*of n*], *drule order-le-less*[*THEN iffD1*], *erule disjE*)
apply (*rule inext-nth-append-eq1*, *assumption*+)
apply (*simp add*: *inext-nth-card-less-Max less-imp-neq*)
thm *inext-nth-card-Max*[*OF - - eq-imp-le*[*OF sym*]]
apply (*simp add*: *inext-nth-card-Max*[*OF - - eq-imp-le*[*OF sym*]] *del*: *Max-less-iff*)
apply (*induct n*)
apply (*frule card-1-imp-singleton*[*OF sym*], *erule exE*)
apply (*simp add*: *iMin-insert*)
apply *simp*
apply (*subgoal-tac inext-nth A n* < *Max A*)
prefer *2*
apply (*rule inext-nth-card-less-Max*, *simp*)
thm *inext-nth-append-eq1*
apply (*simp add*: *inext-nth-append-eq1*)
apply (*rule min-step-inext*)
apply (*simp add*: *inext-nth-closed*)+
apply (*rule conjI*)
apply (*subgoal-tac k* < *A* → *Suc n*)

```
  prefer 2
  apply (subgoal-tac A → Suc n = Max A)
   prefer 2
   thm inext-nth-card-Max
   apply (rule inext-nth-card-Max)
   apply simp+
 apply (rule-tac n=A → n and k=k in inext-min-step, simp+)
apply (rule not-less-iMin)
apply (rule-tac y=Max A in order-less-trans)
apply simp+
done

thm inext-append-eq3
lemma inext-nth-card-append-eq3:
  ⟦ finite A; B ≠ {}; Max A < iMin B ⟧ ⟹
  (A ∪ B) → (card A) = iMin B
apply (case-tac A = {}, simp)
apply (frule not-empty-card-gr0-conv[THEN iffD1], assumption)
thm subst[OF Suc-pred]
apply (rule subst[OF Suc-pred, of card A], assumption)
apply (simp only: inext-nth.simps)
thm inext-nth-card-append-eq1 inext-nth-card-Max' inext-append-eq3
apply (simp add: inext-nth-card-append-eq1 inext-nth-card-Max' inext-append-eq3)
done


lemma inext-nth-card-append-eq2:
  ⟦ finite A; A ≠ {}; B ≠ {}; Max A < iMin B; card A ≤ n ⟧ ⟹
  (A ∪ B) → n = B → (n − card A)
thm inext-nth-cut-ge-inext-nth
apply (rule-tac t=(A ∪ B) → n and s=(A ∪ B) → (card A + (n − card A)) in
subst, simp)
thm inext-nth-cut-ge-inext-nth[symmetric]
apply (subst inext-nth-cut-ge-inext-nth[symmetric], simp)
apply (subst inext-nth-card-append-eq3, assumption+)
apply (simp add: cut-ge-Un cut-ge-Max-empty cut-ge-Min-all del: Max-less-iff)
done




thm inext-append
lemma inext-nth-card-append:
  ⟦ finite A; A ≠ {}; B ≠ {}; Max A < iMin B ⟧ ⟹
  (A ∪ B) → n = (if n < card A then A → n else B → (n − card A))
by (simp add: inext-nth-card-append-eq1 inext-nth-card-append-eq2)

lemma inext-nth-insert-Suc:
  ⟦ I ≠ {}; a < iMin I ⟧ ⟹ (insert a I) → Suc n = I → n
```

apply (*frule not-less-iMin*)
apply (*rule-tac t=I → n* **and** *s=(insert a I − {iMin (insert a I)}) → n* **in** *subst*)
apply (*simp add*: *iMin-insert min-eqL*)
thm *remove-Min-inext-nth-Suc-conv*
apply (*subst remove-Min-inext-nth-Suc-conv*)
apply (*case-tac finite I*)
apply (*simp add*: *not-empty-card-gr0-conv*)+
done

lemma *inext-nth-cut-less-eq*:
n < card (I ↓< t) ⟹ (I ↓< t) → n = I → n
apply (*rule-tac t=I → n* **and** *s=(I ↓< t ∪ I ↓≥ t) → n* **in** *subst*)
apply (*simp add*: *cut-less-cut-ge-ident*)
thm *inext-nth-card-append-eq1*
apply (*case-tac I ↓≥ t = {}*, *simp*)
apply (*rule sym*, *rule inext-nth-card-append-eq1*)
apply (*drule card-gt-0-iff*[*THEN iffD1*, *OF gr-implies-gr0*], *clarify*)
apply (*simp add*: *Ball-def i-cut-mem-iff iMin-gr-iff*)
apply *simp*
done

lemma *less-card-cut-less-imp-inext-nth-less*:
n < card (I ↓< t) ⟹ I → n < t
apply (*case-tac I ↓< t = {}*, *simp*)
apply (*rule subst*[*OF inext-nth-cut-less-eq*], *assumption*)
apply (*rule cut-less-bound*[*OF inext-nth-closed*], *assumption*)
done

lemma *inext-nth-less-less-card-conv*:
I ↓≥ t ≠ {} ⟹ (I → n < t) = (n < card (I ↓< t))
apply (*case-tac I = {}*, *blast*)
apply (*case-tac I ↓< t = {}*)
apply (*simp add*: *linorder-not-less*)
thm *cut-less-empty-iff inext-nth-closed*
apply (*simp add*: *cut-less-empty-iff inext-nth-closed*)
apply (*rule iffI*)
apply (*rule ccontr*, *simp add*: *linorder-not-less*)
apply (*subgoal-tac Max (I ↓< t) < iMin (I ↓≥ t)*)
prefer *2*
apply (*simp add*: *nat-cut-less-finite iMin-gr-iff Ball-def i-cut-mem-iff*)
thm *ssubst*[*OF cut-less-cut-ge-ident*[*OF order-refl*], *of* λ*x*. *x → n < t - t*]
apply (*drule ssubst*[*OF cut-less-cut-ge-ident*[*OF order-refl*], *of* λ*x*. *x → n < t - t*])
thm *inext-nth-card-append-eq2*[*OF nat-cut-less-finite*, *of I t I ↓≥ t n*]
apply (*drule inext-nth-card-append-eq2*[*OF nat-cut-less-finite*, *of I t I ↓≥ t n*], *assumption*+)
apply (*simp add*: *inext-nth-card-append-eq2 nat-cut-less-finite*)
apply (*subgoal-tac* ⋀*x*. *I ↓≥ t → x ≥ t*)
prefer *2*

apply (*rule cut-ge-bound*[*OF inext-nth-closed*], *assumption*)
apply (*simp add*: *linorder-not-le*[*symmetric*])
apply (*rule subst*[*OF inext-nth-cut-less-eq*], *assumption*)
apply (*rule cut-less-bound*[*OF inext-nth-closed*], *assumption*)
done

lemma *cut-less-inext-nth-card-eq1*:
$n < card\ I \vee infinite\ I \Longrightarrow card\ (I \downarrow< (I \rightarrow n)) = n$
apply (*case-tac* $I = \{\}$, *simp*)
apply (*induct n*)
apply (*simp add*: *card-eq-0-iff nat-cut-less-finite cut-less-Min-empty*)
apply (*subgoal-tac* $n < card\ I \vee infinite\ I$)
prefer *2*
apply *fastsimp*
apply *simp*
apply (*subgoal-tac* $I \rightarrow n \neq Max\ I \vee infinite\ I$)
prefer *2*
thm *inext-nth-card-less-Max*[*THEN less-imp-neq*]
apply (*blast dest*: *inext-nth-card-less-Max less-imp-neq*)
apply (*rule subst*[*OF cut-le-less-inext-conv*[*OF inext-nth-closed*]], *assumption*+)
apply (*simp add*: *cut-le-less-conv-if inext-nth-closed cut-less-mem-iff card-insert-if nat-cut-less-finite*)
done

lemma *cut-less-inext-nth-card-eq2*:
$[\![finite\ I;\ card\ I \leq Suc\ n\ ]\!] \Longrightarrow card\ (I \downarrow< (I \rightarrow n)) = card\ I - Suc\ 0$
apply (*case-tac* $I = \{\}$, *simp add*: *cut-less-empty*)
apply (*simp add*: *inext-nth-card-Max cut-less-Max-eq-Diff*)
done

lemma *cut-less-inext-nth-card-if*:
$card\ (I \downarrow< (I \rightarrow n)) = (if\ (n < card\ I \vee infinite\ I)\ then\ n\ else\ card\ I - Suc\ 0)$
by (*simp add*: *cut-less-inext-nth-card-eq1 cut-less-inext-nth-card-eq2*)

lemma *cut-le-inext-nth-card-eq1*:
$n < card\ I \vee infinite\ I \Longrightarrow card\ (I \downarrow\leq (I \rightarrow n)) = Suc\ n$
apply (*case-tac* $I = \{\}$, *simp*)
thm *cut-le-less-inext-conv*[*OF inext-nth-closed*]
apply (*simp add*: *cut-le-less-conv-if inext-nth-closed card-insert-if nat-cut-less-finite cut-less-mem-iff cut-less-inext-nth-card-eq1*)
done
lemma *cut-le-inext-nth-card-eq2*:
$[\![finite\ I;\ card\ I \leq Suc\ n\ ]\!] \Longrightarrow card\ (I \downarrow\leq (I \rightarrow n)) = card\ I$
apply (*case-tac* $I = \{\}$, *simp add*: *cut-le-empty*)
apply (*simp add*: *inext-nth-card-Max cut-le-Max-all*)
done

lemma *cut-le-inext-nth-card-if*:
card (*I* ↓≤ (*I* → *n*)) = (
if (*n* < *card I* ∨ *infinite I*) *then Suc n else card I*)
by (*simp add*: *cut-le-inext-nth-card-eq1 cut-le-inext-nth-card-eq2*)

term *iprev*
primrec
iprev-nth :: *nat set* ⇒ *nat* ⇒ *nat*
where
iprev-nth I 0 = *Max I*
| *iprev-nth I* (*Suc n*) = *iprev* (*iprev-nth I n*) *I*
thm *inext-iprev*
notation (*xsymbols*)
iprev-nth ((- ← -) [*100*, *100*] *60*)
notation (*HTML* **output**)
iprev-nth ((- ← -) [*100*, *100*] *60*)

lemma *iprev-nth-closed*: ⟦ *finite I*; *I* ≠ {} ⟧ ⟹ *I* ← *n* ∈ *I*
apply (*induct n*)
apply *simp*
apply (*simp add*: *iprev-closed*)
done

thm *iprev-image*
lemma *iprev-nth-image*:
⟦ *finite I*; *I* ≠ {}; *strict-mono-on f I* ⟧ ⟹ (*f* ' *I*) ← *n* = *f* (*I* ← *n*)
apply (*induct n*)
apply (*simp add*: *Max-mono-on2 strict-mono-on-imp-mono-on*)
apply (*simp add*: *iprev-image iprev-nth-closed*)
done

thm *iprev-mono*
lemma *iprev-nth-Suc-mono*: *I* ← (*Suc n*) ≤ *I* ← *n*
by (*simp add*: *iprev-mono*)
lemma *iprev-nth-mono*: *a* ≤ *b* ⟹ *I* ← *b* ≤ *I* ← *a*
apply (*induct b*)
apply *simp*
apply (*drule le-Suc-eq*[*THEN iffD1*], *erule disjE*)
apply (*rule-tac y*=*iprev-nth I b* **in** *order-trans*)
apply (*rule iprev-nth-Suc-mono*)
apply *simp*
apply *simp*
done
lemma *iprev-nth-Suc-mono2*:
⟦ *finite I*; ∃ *x*∈*I*. *x* < *I* ← *n* ⟧ ⟹ *I* ← (*Suc n*) < *I* ← *n*
thm *iprev-mono2*

apply *simp*
apply (*rule iprev-mono2*)
thm *iprev-nth-closed*
apply (*blast intro*: *iprev-nth-closed*)+
done

lemma *iprev-nth-mono2*:
⟦ *finite I*; ∃ *x*∈*I*. *x* < *I* ← *a* ⟧ ⟹ (*I* ← *b* < *I* ← *a*) = (*a* < *b*)
apply (*subgoal-tac I* ≠ {})
prefer *2*
apply *blast*
apply (*rule iffI*)
apply (*rule ccontr*)
apply (*simp add*: *linorder-not-less*)
apply (*drule iprev-nth-mono*[*of - - I*])
apply *simp*
apply *clarify*
apply (*induct b*)
apply *blast*
apply (*drule less-Suc-eq*[*THEN iffD1*], *erule disjE*)
apply (*blast intro*: *order-le-less-trans iprev-nth-Suc-mono*)
apply (*blast intro*: *iprev-nth-Suc-mono2*)
done

lemma *iprev-nth-iMin-fix*:
⟦ *I* ≠ {}; *I* ← *a* = *iMin I*; *a* ≤ *b* ⟧ ⟹ *I* ← *b* = *iMin I*
apply (*induct b*)
apply *simp*
apply (*drule le-Suc-eq*[*THEN iffD1*], *erule disjE*)
apply (*simp add*: *iprev-iMin*)
apply *blast*
done

lemma *iprev-nth-singleton*: {*a*} ← *n*= *a*
thm *iprev-nth-iMin-fix*[*OF singleton-not-empty - le0*]
by (*simp add*: *iprev-nth-iMin-fix*[*OF singleton-not-empty - le0*])

8.3 Induction over arbitrary natural sets using the functions *inext* and *iprev*

lemma *inext-nth-surj-aux1*:
{*x* ∈ *I*. ¬(∃ *n*. *I* → *n* = *x*)} = {}
(**is** *?S* = {}
is { *x* ∈ *I*. *?P x*} = {})
apply (*case-tac I* = {}, *blast*)
proof (*rule ccontr*)
assume *as-S-not-empty*: *?S* ≠ {}

obtain *S* **where** *s-S*: *S* = *?S* **by** *blast*

hence *S-not-empty*: $S \neq \{\}$
using *as-S-not-empty* **by** *blast*

have *s-not-ex*: $\bigwedge x.\ [\![\ x \in I;\ ?P\ x\ ]\!] \Longrightarrow x \in S$
using *s-S* **by** *blast*

have *s-subset*:$S \subseteq I$
using *s-S* **by** *blast*
have *i-not-empty*: $I \neq \{\}$
using *as-S-not-empty* **by** *blast*

have *s-iMin-S*: $iMin\ S \in S$
thm *iMinI-ex2*
using *S-not-empty* **by** (*simp add*: *iMinI-ex2*)
hence *s-iMin-i*: $iMin\ S \in I$
using *s-subset* **by** *blast*

show *False*
proof *cases*
assume *as*:$iMin\ I < iMin\ S$

obtain *prev* **where** *s-prev*: $prev = iprev\ (iMin\ S)\ I$ **by** *blast*
have *s-prev-in*: $prev \in I$
apply (*simp add*: *s-prev*)
thm *iprev-closed*[*of iMin S i*]
apply (*rule iprev-closed*)
apply (*rule s-iMin-i*)
done

have *s-prev-next-min*: $inext\ prev\ I = iMin\ S$
apply (*simp add*: *s-prev*)
thm *inext-iprev*[*of I iMin S*]
apply (*rule inext-iprev*)
apply (*insert as*, *simp*)
done

have *s-prev-min-1*: $prev < iMin\ S$
apply (*simp only*: *s-prev*)
thm *iprev-mono2*[*of iMin S I*]
apply (*rule iprev-mono2*[*of iMin S*])
apply (*rule s-iMin-i*)
apply (*rule-tac x*=*iMin I* **in** *bexI*)
apply (*rule as*)
apply (*simp add*: *iMinI-ex2 i-not-empty*)
done
hence *prev-not-in-s*: $prev \notin S$
thm *not-less-iMin*
by (*simp add*: *not-less-iMin*)
have $\exists\, n.\ I \rightarrow n = prev$

```
    thm s-not-ex[of prev]
    by (insert prev-not-in-s s-not-ex[of prev] s-prev-in, blast)
  then obtain nPrev where s-nPrev: I → nPrev = prev by blast
  hence I → (Suc nPrev) = inext prev I by simp
  hence I → (Suc nPrev) = iMin S
    using s-prev-next-min by simp
  hence ∃ n. I → n = iMin S by blast
  hence iMin S ∉ S
    using s-iMin-i s-S by blast
  thus False
    using s-iMin-S by blast
next
  assume as:¬(iMin I < iMin S)

  have iMin S = iMin I
    apply (insert s-subset S-not-empty as)
    thm iMin-subset
    apply (frule-tac A=S and B=I in iMin-subset)
    by simp-all
  hence ∃ n. I → n ∈ S
    apply (rule-tac x=0 in exI)
    apply (insert s-iMin-S)
    apply simp
    done
  thus False
    using s-S by blast
qed
qed

term inext-nth
term λn. I → n
lemma inext-nth-surj-on:surj-on (λn. I → n) UNIV I
apply (simp add: surj-on-conv)
thm inext-nth-surj-aux1[of I]
by (insert inext-nth-surj-aux1[of I], blast)
corollary in-imp-ex-inext-nth: x ∈ I ⟹ ∃ n. x = I → n
thm surj-onD
thm surj-onD[where A=UNIV, simplified]
apply (rule surj-onD[where A=UNIV, simplified])
apply (rule inext-nth-surj-on)
apply assumption
done

lemma inext-induct:
  ⟦ P (iMin I); ⋀n. ⟦ n ∈ I; P n ⟧ ⟹ P (inext n I); n ∈ I ⟧ ⟹ P n
thm image-nat-induct
thm image-nat-induct[where P=P and f=λn. I → n and I=I and a=n]
apply (rule-tac f=λn. I → n and I=I in image-nat-induct)
thm inext-nth-closed[OF in-imp-not-empty]
```

thm *inext-nth-surj-on*
apply (*simp add*: *inext-nth-closed*[*OF in-imp-not-empty*] *inext-nth-surj-on*)+
done
thm *inext-induct*

lemma *iprev-nth-surj-aux1*:
  finite $I \Longrightarrow \{ x \in I.\ \neg(\exists\, n.\ I \leftarrow n = x)\} = \{\}$
apply (*case-tac* $I = \{\}$, *blast*)
proof (*rule ccontr*)
  assume *as-finite-i*: *finite I*
  let *?S* $= \{x \in I.\ \neg\ (\exists\, n.\ I \leftarrow n = x)\}$
  assume *as-S-not-empty*: *?S* $\neq \{\}$

  obtain *S* **where** *s-S*: $S =$ *?S* **by** *blast*
  hence *S-not-empty*: $S \neq \{\}$
    using *as-S-not-empty* **by** *blast*

  have *s-not-ex*: $\bigwedge x.\ [\![\ x \in I;\ \neg(\exists\, n.\ I \leftarrow n = x)\ ]\!] \Longrightarrow x \in S$
    using *s-S* **by** *blast*

  have *s-subset*:$S \subseteq I$
    using *s-S* **by** *blast*
  have *i-not-empty*: $I \neq \{\}$
    using *as-S-not-empty* **by** *blast*

  from *as-finite-i*
  have *S-finite*: *finite S*
    using *s-subset* **by** (*blast intro*: *finite-subset*)

  have *s-Max-S*: *Max S* $\in S$
    thm *Max-in*
    using *S-not-empty S-finite* **by** *simp*
  hence *s-Max-i*: *Max S* $\in I$
    using *s-subset* **by** *blast*

  show *False*
  proof *cases*
    assume *as*:*Max S* $<$ *Max I*

    obtain *next'* **where** *s-next*: *next'* $=$ *inext* (*Max S*) *I* **by** *blast*
    have *s-next-in*: *next'* $\in I$
      thm *inext-closed*[*of Max S I*]
      by (*simp add*: *s-next inext-closed s-Max-i*)

    have *s-next-prev-max*: *iprev next' I* $=$ *Max S*
      apply (*simp add*: *s-next*)
      thm *iprev-inext*[*of Max S I*]
      apply (*rule iprev-inext*)

```
    apply (insert as, simp)
    done

  have s-next-max-1: Max S < next'
    apply (simp add: s-next)
    thm inext-mono2[of Max S I]
    apply (rule inext-mono2[of Max S I])
    apply (rule s-Max-i)
    apply (rule-tac x=Max I in bexI)
    apply (rule as)
    apply (simp add: as-finite-i i-not-empty)
    done
  hence next-not-in-s: next' ∉ S
    using S-finite S-not-empty
    apply clarify
    thm Max-ge[of S next']
    apply (drule Max-ge[of - next'])
    apply simp-all
    done
  have ∃ n. I ← n = next'
    thm s-not-ex[of next']
    by (insert next-not-in-s s-not-ex[of next'] s-next-in, blast)
  then obtain nNext where s-nNext: I ← nNext = next' by blast
  hence I ← (Suc nNext) = iprev next' I by simp
  hence I ← (Suc nNext) = Max S
    using s-next-prev-max by simp
  hence ∃ n. I ← n = Max S by blast
  hence Max S ∉ S
    using s-Max-i s-S by blast
  thus False
    using s-Max-S by blast+
 next
  assume as:¬(Max S < Max I)

  have Max S = Max I
    apply (insert s-subset S-not-empty as-finite-i as)
    thm Max-subset[of S I]
    apply (drule Max-subset[of - I])
    by simp-all
  hence ∃ n. I ← n ∈ S
    apply (rule-tac x=0 in exI)
    apply (insert s-Max-S)
    apply simp
    done
  thus False
    using s-S by blast
 qed
qed
```

term *iprev-nth*
term $\lambda n.$ *iprev-nth I n*
lemma *iprev-nth-surj-on*: *finite I* $\Longrightarrow$ *surj-on* $(\lambda n.\ I \leftarrow n)$ *UNIV I*
apply (*simp add*: *surj-on-def*)
thm *iprev-nth-surj-aux1*[*of I*]
by (*insert iprev-nth-surj-aux1*[*of I*], *blast*)
corollary *in-imp-ex-iprev-nth*:
$[\![$ *finite I*; $x \in I\ ]\!] \Longrightarrow \exists n.\ x = I \leftarrow n$
thm *surj-onD*
thm *surj-onD*[*of - UNIV I*, *simplified*]
apply (*rule surj-onD*[*of - UNIV I*, *simplified*])
apply (*rule iprev-nth-surj-on*)
apply *assumption*+
done

lemma *iprev-induct*:
$[\![\ P\ (Max\ I); \bigwedge n.\ [\![\ n \in I; P\ n\ ]\!] \Longrightarrow P\ (iprev\ n\ I);$ *finite I*; $n \in I\ ]\!] \Longrightarrow P\ n$
thm *image-nat-induct*
thm *image-nat-induct*[**where** $P{=}P$ **and** $f{=}\lambda n.\ I \leftarrow n$ **and** $I{=}I$ **and** $a{=}n$]
apply (*rule-tac* $f{=}\lambda n.\ I \leftarrow n$ **and** $I{=}I$ **in** *image-nat-induct*)
thm *iprev-nth-closed*[*OF - in-imp-not-empty*]
apply (*simp add*: *iprev-nth-closed*[*OF - in-imp-not-empty*] *iprev-nth-surj-on*)+
done
thm
inext-induct
iprev-induct

8.4 Natural intervals with *inext* and *iprev*

lemma *inext-atLeast*: $n \leq t \Longrightarrow$ *inext t* $\{n..\} =$ *Suc t*
apply (*unfold inext-def*)
apply (*subgoal-tac Suc t* $\in \{n..\} \downarrow> t$)
prefer *2*
apply (*simp add*: *cut-greater-mem-iff*)
apply (*simp add*: *in-imp-not-empty*)
apply (*rule iMin-equality*, *assumption*)
apply (*simp add*: *cut-greater-mem-iff*)
done

lemma *iprev-atLeast'*: $n \leq t \Longrightarrow$ *iprev* $(Suc\ t)\ \{n..\} = t$
apply (*rule subst*[*OF inext-atLeast*], *assumption*)
apply (*rule iprev-inext-infin*[*OF infinite-atLeast*])
done
lemma *iprev-atLeast*: $n < t \Longrightarrow$ *iprev t* $\{n..\} = t - Suc\ 0$
by (*insert iprev-atLeast'*[*of n t* $-$ *Suc 0*], *simp*)

lemma *inext-atMost*: $t < n \Longrightarrow$ *inext t* $\{..n\} =$ *Suc t*
apply (*unfold inext-def*)
apply (*subgoal-tac Suc t* $\in \{..n\} \downarrow> t$)

prefer *2*
apply (*simp add*: *cut-greater-mem-iff*)
apply (*simp add*: *in-imp-not-empty*)
apply (*rule iMin-equality*, *assumption*)
apply (*simp add*: *cut-greater-mem-iff*)
done
lemma *iprev-atMost*: $t \leq n \Longrightarrow iprev\ t\ \{..n\} = t - Suc\ 0$
apply (*case-tac t*)
apply *simp*
thm *subst*[*OF iMin-atMost*[*of n*]]
apply (*rule subst*[*OF iMin-atMost*[*of n*]])
apply (*rule iprev-iMin*)
apply *simp*
apply (*drule Suc-le-lessD*)
apply (*rule subst*[*OF inext-atMost*], *assumption*)
apply (*simp add*: *Max-atMost iprev-inext-fin*)
done

lemma *inext-lessThan*: $Suc\ t < n \Longrightarrow inext\ t\ \{..<n\} = Suc\ t$
apply (*rule subst*[*OF Suc-pred*, *of n*], *simp*)
apply (*subst lessThan-Suc-atMost*)
apply (*simp add*: *inext-atMost*)
done
lemma *iprev-lessThan*: $t < n \Longrightarrow iprev\ t\ \{..<n\} = t - Suc\ 0$
apply (*case-tac n*, *simp*)
apply (*simp add*: *lessThan-Suc-atMost iprev-atMost*)
done

lemma *inext-atLeastAtMost*: $[\![\ m \leq t;\ t < n\ ]\!] \Longrightarrow inext\ t\ \{m..n\} = Suc\ t$
by (*simp add*: *atLeastAtMost-def cut-le-Int-conv*[*symmetric*] *inext-atLeast inext-cut-le-conv*)
lemma *iprev-atLeastAtMost*: $[\![\ m < t;\ t \leq n\ ]\!] \Longrightarrow iprev\ t\ \{m..n\} = t - Suc\ 0$
by (*simp add*: *atLeastAtMost-def cut-le-Int-conv*[*symmetric*] *iprev-atLeast iprev-cut-le-conv*)
lemma *iprev-atLeastAtMost'*: $[\![\ m \leq t;\ t < n\ ]\!] \Longrightarrow iprev\ (Suc\ t)\ \{m..n\} = t$
by (*simp add*: *iprev-atLeastAtMost*[*of - Suc t*])

lemma *inext-nth-atLeast* : $\{n..\} \rightarrow a = n + a$
apply (*induct a*, *simp add*: *iMin-atLeast*)
apply (*simp add*: *inext-atLeast*)
done
lemma *inext-nth-atLeastAtMost*: $[\![\ a \leq n - m;\ m \leq n\ ]\!] \Longrightarrow \{m..n\} \rightarrow a = m +$
a
apply (*induct a*, *simp add*: *iMin-atLeastAtMost*)
apply (*simp add*: *inext-atLeastAtMost*)
done
lemma *iprev-nth-atLeastAtMost*: $[\![\ a \leq n - m;\ m \leq n\ ]\!] \Longrightarrow \{m..n\} \leftarrow a = n -$
a
apply (*induct a*, *simp add*: *Max-atLeastAtMost*)
apply (*simp add*: *iprev-atLeastAtMost*)
done

lemma *inext-nth-atMost*: $a \leq n \Longrightarrow \{..n\} \rightarrow a = a$
apply (*insert inext-nth-atLeastAtMost*[*of a n 0*])
apply (*simp add*: *atMost-atLeastAtMost-0-conv*)
done
lemma *iprev-nth-atMost*: $a \leq n \Longrightarrow \{..n\} \leftarrow a = n - a$
apply (*insert iprev-nth-atLeastAtMost*[*of a n 0*])
apply (*simp add*: *atMost-atLeastAtMost-0-conv*)
done

lemma *inext-nth-lessThan* : $a < n \Longrightarrow \{..<n\} \rightarrow a = a$
apply (*case-tac n*, *simp*)
apply (*simp add*: *lessThan-Suc-atMost inext-nth-atMost*)
done
lemma *iprev-nth-lessThan*: $a < n \Longrightarrow \{..<n\} \leftarrow a = n - Suc\ a$
apply (*case-tac n*, *simp*)
apply (*simp add*: *lessThan-Suc-atMost iprev-nth-atMost*)
done

lemma *inext-nth-UNIV*: $UNIV \rightarrow a = a$
by (*simp add*: *inext-nth-atLeast del*: *atLeast-0 add*: *atLeast-0*[*symmetric*])

8.5 Further result for *inext-nth* and *iprev-nth*

thm *inext-iprev*
lemma *inext-iprev-nth-Suc*:
$iMin\ I \neq I \leftarrow n \Longrightarrow inext\ (I \leftarrow Suc\ n)\ I = I \leftarrow n$
by (*simp add*: *inext-iprev*)
lemma *inext-iprev-nth-pred*:
$[\![\ finite\ I;\ iMin\ I \neq I \leftarrow (n - Suc\ 0)\ ]\!] \Longrightarrow$
$inext\ (I \leftarrow n)\ I = I \leftarrow (n - Suc\ 0)$
apply (*case-tac n*)
apply (*simp add*: *inext-Max*)
apply (*simp add*: *inext-iprev*)
done

lemma *iprev-inext-nth-Suc*:
$I \rightarrow n \neq Max\ I \vee infinite\ I \Longrightarrow iprev\ (I \rightarrow Suc\ n)\ I = I \rightarrow n$
by (*simp add*: *iprev-inext*)
lemma *iprev-inext-nth-pred*:
$I \rightarrow (n - Suc\ 0) \neq Max\ I \vee infinite\ I \Longrightarrow$
$iprev\ (I \rightarrow n)\ I = I \rightarrow (n - Suc\ 0)$
apply (*case-tac n*)
apply (*simp add*: *iprev-iMin*)
apply (*simp add*: *iprev-inext*)
done

thm *inext-imirror-iprev-conv*
lemma *inext-nth-imirror-iprev-nth-conv*:
$[\![\ finite\ I;\ I \neq \{\}\ ]\!] \Longrightarrow$

(*imirror I*) $\rightarrow$ *n* = *mirror-elem* (*I* $\leftarrow$ *n*) *I*
apply (*induct n*)
apply (*simp add*: *imirror-iMin mirror-elem-Max*)
apply (*simp add*: *inext-imirror-iprev-conv'* *iprev-nth-closed*)
done
corollary *inext-nth-imirror-iprev-nth-conv2*:
⟦ *finite I*; *I* $\neq$ {} ⟧ $\Longrightarrow$
mirror-elem ((*imirror I*) $\leftarrow$ *n*) *I* = *I* $\rightarrow$ *n*
thm *inext-nth-imirror-iprev-nth-conv*[*OF imirror-finite imirror-not-empty*]
apply (*frule inext-nth-imirror-iprev-nth-conv*[*OF imirror-finite imirror-not-empty*, *of - n*], *assumption*)
apply (*simp add*: *imirror-imirror-ident mirror-elem-imirror*)
done

lemma *iprev-nth-imirror-inext-nth-conv*:
⟦ *finite I*; *I* $\neq$ {} ⟧ $\Longrightarrow$
(*imirror I*) $\leftarrow$ *n* = *mirror-elem* (*I* $\rightarrow$ *n*) *I*
apply (*induct n*)
apply (*simp add*: *imirror-Max mirror-elem-Min*)
apply (*simp add*: *iprev-imirror-inext-conv'* *inext-nth-closed*)
done
corollary *iprev-nth-imirror-inext-nth-conv2*:
⟦ *finite I*; *I* $\neq$ {} ⟧ $\Longrightarrow$
mirror-elem ((*imirror I*) $\rightarrow$ *n*) *I* = (*I* $\leftarrow$ *n*)
thm *iprev-nth-imirror-inext-nth-conv*[*OF imirror-finite imirror-not-empty*]
apply (*frule iprev-nth-imirror-inext-nth-conv*[*OF imirror-finite imirror-not-empty*, *of - n*], *assumption*)
apply (*simp add*: *imirror-imirror-ident mirror-elem-imirror*)
done

thm *inext-nth-card-less-Max*
lemma *iprev-nth-card-greater-iMin*: *Suc n* < *card I* $\Longrightarrow$ *iMin I* < *I* $\leftarrow$ *n*
apply (*subgoal-tac I* $\neq$ {} *finite I*)
prefer *2*
apply (*rule card-gr0-imp-finite*, *simp*)
prefer *2*
apply (*rule card-gr0-imp-not-empty*, *simp*)
thm *subst*[*OF iprev-nth-imirror-inext-nth-conv2*]
apply (*subst iprev-nth-imirror-inext-nth-conv2*[*symmetric*], *assumption*+)
thm *subst*[*OF mirror-elem-Max*]
apply (*subst mirror-elem-Max*[*symmetric*], *assumption*+)
thm *subst*[*OF mirror-elem-imirror*, *of I*]
apply (*subst mirror-elem-imirror*[*symmetric*], *assumption*)

apply (*subst mirror-elem-imirror*[*symmetric*], *assumption*)
apply (*frule imirror-finite*, *frule imirror-not-empty*)
thm *mirror-elem-less-conv*
apply (*rule mirror-elem-less-conv*[*THEN iffD2*])
apply *assumption*
apply (*rule inext-nth-closed*, *assumption*)
apply (*rule subst*[*OF imirror-Max*], *assumption*)
apply (*rule Max-in*, *assumption*+)
apply (*rule subst*[*OF imirror-Max*], *assumption*)
thm *inext-nth-card-less-Max*
apply (*simp add*: *inext-nth-card-less-Max imirror-card*)
done

lemma *iprev-nth-card-iMin*:
⟦ *finite I*; *I* ≠ {}; *card I* ≤ *Suc n* ⟧ ⟹ *I* ← *n* = *iMin I*
thm *subst*[*OF iprev-nth-imirror-inext-nth-conv2*]
apply (*subst iprev-nth-imirror-inext-nth-conv2*[*symmetric*], *assumption*+)
thm *subst*[*OF mirror-elem-Max*]
apply (*subst mirror-elem-Max*[*symmetric*], *assumption*+)
thm *subst*[*OF mirror-elem-imirror*, *of I*]
apply (*subst mirror-elem-imirror*[*symmetric*], *assumption*)
apply (*subst mirror-elem-imirror*[*symmetric*], *assumption*)
thm *subst*[*OF imirror-Max*]
apply (*rule subst*[*OF imirror-Max*], *assumption*)
apply (*frule imirror-finite*, *frule imirror-not-empty*)
thm *mirror-elem-eq-conv'*
apply (*simp add*: *mirror-elem-eq-conv' inext-nth-closed inext-nth-card-Max imirror-card*)
done
lemma *iprev-nth-card-iMin'*:
⟦ *finite I*; *I* ≠ {}; *card I* − *Suc 0* ≤ *n* ⟧ ⟹ *I* ← *n* = *iMin I*
by (*simp add*: *iprev-nth-card-iMin*)

end

9 List2: Additional definitions and results for lists

theory *List2*
imports *../CommonSet/SetIntervalCut*
begin

9.1 Additional definitions and results for lists

Infix syntactical abbreviations for operators *take* and *drop*. The abbreviations resemble to the operator symbols used later for take and drop operators on infinite lists in ListInf.

abbreviation (*xsymbols*)
f-take' :: *'a list* ⇒ *nat* ⇒ *'a list* (**infixl** ↓ *100*)

where
  $xs \downarrow n \equiv take\ n\ xs$
abbreviation (*xsymbols*)
  f-drop′ :: *'a list* ⇒ *nat* ⇒ *'a list* (**infixl** ↑ *100*)
where
  $xs \uparrow n \equiv drop\ n\ xs$
syntax (*HTML* **output**)
  f-take′ :: *'a list* ⇒ *nat* ⇒ *'a list* (**infixl** ↓ *100*)
  f-drop′ :: *'a list* ⇒ *nat* ⇒ *'a list* (**infixl** ↑ *100*)

term $xs \downarrow n$
term $xs \uparrow n$

thm *List.append-Cons*
lemma *append-eq-Cons*: $[x] @ xs = x \# xs$
by *simp*

lemma *length-Cons*: $length\ (x \# xs) = Suc\ (length\ xs)$
by *simp*
lemma *length-snoc*: $length\ (xs @ [x]) = Suc\ (length\ xs)$
by *simp*

9.1.1 Additional lemmata about list emptiness

lemma *length-greater-imp-not-empty*:$n < length\ xs \Longrightarrow xs \neq []$
by *fastsimp*

lemma *length-ge-Suc-imp-not-empty*:$Suc\ n \leq length\ xs \Longrightarrow xs \neq []$
by *fastsimp*
thm *length-take*

lemma *length-take-le*: $length\ (xs \downarrow n) \leq length\ xs$
by *simp*

lemma *take-not-empty-conv*:$(xs \downarrow n \neq []) = (0 < n \wedge xs \neq [])$
by *simp*

lemma *drop-not-empty-conv*:$(xs \uparrow n \neq []) = (n < length\ xs)$
by *fastsimp*

lemma *zip-eq-Nil*: $(zip\ xs\ ys = []) = (xs = [] \vee ys = [])$
by (*force simp*: *length-0-conv*[*symmetric*] *min-def simp del*: *length-0-conv*)

lemma *zip-not-empty-conv*: $(zip\ xs\ ys \neq []) = (xs \neq [] \wedge ys \neq [])$
by (*simp add*: *zip-eq-Nil*)

9.1.2 Additional lemmata about *take*, *drop*, *hd*, *last*, *nth* and *filter*

lemma *nth-tl-eq-nth-Suc*:
Suc n ≤ length xs ⟹ (tl xs) ! n = xs ! Suc n
thm *hd-Cons-tl*[*OF length-ge-Suc-imp-not-empty*, *THEN subst*]
by (*rule hd-Cons-tl*[*OF length-ge-Suc-imp-not-empty*, *THEN subst*], *simp+*)
corollary *nth-tl-eq-nth-Suc2*:
n < length xs ⟹ (tl xs) ! n = xs ! Suc n
by (*simp add*: *nth-tl-eq-nth-Suc*)

lemma *hd-eq-first*: *xs ≠ [] ⟹ xs ! 0 = hd xs*
by (*induct xs*, *simp-all*)
corollary *take-first*:*xs ≠ [] ⟹ xs ↓ (Suc 0) = [xs ! 0]*
by (*induct xs*, *simp-all*)
corollary *take-hd*:*xs ≠ [] ⟹ xs ↓ (Suc 0) = [hd xs]*
by (*simp add*: *take-first hd-eq-first*)

thm *last-conv-nth*
theorem *last-nth*: *xs ≠ [] ⟹ last xs = xs ! (length xs − Suc 0)*
by (*simp add*: *last-conv-nth*)

lemma *last-take*: *n < length xs ⟹ last (xs ↓ Suc n) = xs ! n*
by (*simp add*: *last-nth length-greater-imp-not-empty min-eqR*)
corollary *last-take2*:
⟦ 0 < n; n ≤ length xs ⟧ ⟹ last (xs ↓ n) = xs ! (n − Suc 0)
thm *diff-Suc-less*[*THEN order-less-le-trans*]
apply (*frule diff-Suc-less*[*THEN order-less-le-trans*, *of - length xs 0*], *assumption*)
thm *last-take*[*of n − Suc 0 xs*]
apply (*drule last-take*[*of n − Suc 0 xs*])
apply *simp*
done

thm *List.nth-drop*
corollary *nth-0-drop*: *n ≤ length xs ⟹ (xs ↑ n) ! 0 = xs ! n*
by (*cut-tac nth-drop*[*of n 0 xs*], *simp+*)
corollary *hd-drop*: *n < length xs ⟹ hd (xs ↑ n) = xs ! n*
apply (*frule drop-not-empty-conv*[*THEN iffD2*])
apply (*simp add*: *hd-eq-first*[*symmetric*])
done

lemma *drop-eq-tl*: *xs ↑ (Suc 0) = tl xs*
by (*simp add*: *drop-Suc*)

lemma *drop-take-1*:
n < length xs ⟹ xs ↑ n ↓ (Suc 0) = [xs ! n]
thm *take-hd hd-drop*
by (*simp add*: *take-hd hd-drop*)

lemma *upt-append*: *m ≤ n ⟹ [0..<m] @ [m..<n] = [0..<n]*
thm *upt-add-eq-append*[*of 0 m n − m*]

by (*insert upt-add-eq-append*[*of 0 m n − m*], *simp*)

thm *nth-append*
lemma *nth-append1*: $n < length\ xs \Longrightarrow (xs\ @\ ys)\ !\ n = xs\ !\ n$
by (*simp add*: *nth-append*)
lemma *nth-append2*: $length\ xs \leq n \Longrightarrow (xs\ @\ ys)\ !\ n = ys\ !\ (n - length\ xs)$
by (*simp add*: *nth-append*)

lemma *list-all-conv*: $list\text{-}all\ P\ xs = (\forall i{<}length\ xs.\ P\ (xs\ !\ i))$
by (*rule list-all-length*)

thm *Fun.fun-eq-iff*
lemma *expand-list-eq*:
$\bigwedge ys.\ (xs = ys) = (length\ xs = length\ ys \wedge (\forall i{<}length\ xs.\ xs\ !\ i = ys\ !\ i))$
by (*rule list-eq-iff-nth-eq*)
lemmas *list-eq-iff = expand-list-eq*

lemma *list-take-drop-imp-eq*:
$[\![\ xs \downarrow n = ys \downarrow n;\ \ xs \uparrow n = ys \uparrow n\ ]\!] \Longrightarrow xs = ys$
apply (*rule subst*[*OF append-take-drop-id*[*of n xs*]])
apply (*rule subst*[*OF append-take-drop-id*[*of n ys*]])
apply *simp*
done

lemma *list-take-drop-eq-conv*:
$(xs = ys) = (\exists n.\ (xs \downarrow n = ys \downarrow n \wedge xs \uparrow n = ys \uparrow n))$
by (*blast intro*: *list-take-drop-imp-eq*)

lemma *list-take-eq-conv*: $(xs = ys) = (\forall n.\ xs \downarrow n = ys \downarrow n)$
apply (*rule iffI*, *simp*)
apply (*drule-tac x=max* (*length xs*) (*length ys*) **in** *spec*)
apply *simp*
done

lemma *list-drop-eq-conv*: $(xs = ys) = (\forall n.\ xs \uparrow n = ys \uparrow n)$
apply (*rule iffI*, *simp*)
apply (*drule-tac x=0* **in** *spec*)
apply *simp*
done

abbreviation (*xsymbols*)
$replicate'$ :: $'a \Rightarrow nat \Rightarrow 'a\ list$ (-$^{-}$ [*1000*,*65*])
where
$x^n \equiv replicate\ n\ x$

term *length* $x^{(a+b)}$

thm *List.replicate-Suc*
thm *List.replicate-app-Cons-same*
lemma *replicate-snoc*: x^n @ $[x] = x^{Suc\ n}$
by (*simp add*: *replicate-app-Cons-same*)

thm *List.nth-replicate*
lemma *eq-replicate-conv*: $(\forall i{<}length\ xs.\ xs\ !\ i = m) = (xs = m^{length\ xs})$
apply (*rule iffI*)
apply (*simp add*: *expand-list-eq*)
apply *clarsimp*
apply (*rule ssubst*[*of xs replicate* (*length xs*) *m*], *assumption*)
apply (*rule nth-replicate*, *simp*)
done

lemma *replicate-Cons-length*: *length* $(x\ \#\ a^n) = Suc\ n$
by *simp*
lemma *replicate-pred-Cons-length*: $0 < n \Longrightarrow length\ (x\ \#\ a^{n\ -\ Suc\ 0}) = n$
by *simp*

thm *replicate-add*
lemma *replicate-le-diff*: $m \leq n \Longrightarrow x^m$ @ $x^{n\ -\ m} = x^n$
by (*simp add*: *replicate-add*[*symmetric*])
lemma *replicate-le-diff2*: $[\![\ k \leq m;\ m \leq n\ ]\!] \Longrightarrow x^{m\ -\ k}$ @ $x^{n\ -\ m} = x^{n\ -\ k}$
by (*subst replicate-add*[*symmetric*], *simp*)

thm *list.induct*
lemma *append-constant-length-induct-aux*: $\bigwedge xs.$
$[\![\ length\ xs\ div\ k = n;\ \bigwedge ys.\ k = 0 \vee length\ ys < k \Longrightarrow P\ ys;$
$\bigwedge xs\ ys.\ [\![\ length\ xs = k;\ P\ ys\ ]\!] \Longrightarrow P\ (xs\ @\ ys)\ ]\!] \Longrightarrow P\ xs$
apply (*case-tac* $k = 0$, *blast*)
apply *simp*
apply (*induct n*)
apply (*simp add*: *div-eq-0-conv'*)
apply (*subgoal-tac* $k \leq length\ xs$)
prefer *2*
apply (*rule div-gr-imp-gr-divisor*[*of 0*], *simp*)
apply (*simp only*: *atomize-all atomize-imp*, *clarsimp*)
apply (*erule-tac x=drop k xs* **in** *allE*)
apply (*simp add*: *div-diff-self2*)
apply (*erule-tac x=undefined* **in** *allE*)
apply (*erule-tac x=take k xs* **in** *allE*)
apply (*simp add*: *min-eqR*)
apply (*erule-tac x=drop k xs* **in** *allE*)
apply *simp*
done

lemma *append-constant-length-induct*:

$[\![\bigwedge ys.\ k = 0 \vee length\ ys < k \Longrightarrow P\ ys;$
$\quad \bigwedge xs\ ys.\ [\![length\ xs = k;\ P\ ys]\!] \Longrightarrow P\ (xs\ @\ ys)]\!] \Longrightarrow P\ xs$
by (*simp add*: *append-constant-length-induct-aux*[*of* - - *length xs div k*])

lemma *zip-swap*: *map* (λ(*y*,*x*). (*x*,*y*)) (*zip ys xs*) = (*zip xs ys*)
by (*simp add*: *expand-list-eq*)

lemma *zip-takeL*: (*zip xs ys*) ↓ *n* = *zip* (*xs* ↓ *n*) *ys*
by (*simp add*: *expand-list-eq*)

lemma *zip-takeR*: (*zip xs ys*) ↓ *n* = *zip xs* (*ys* ↓ *n*)
thm *zip-swap*[*of ys*]
apply (*subst zip-swap*[*of ys*, *symmetric*])
apply (*subst take-map*)
apply (*subst zip-takeL*)
apply (*simp add*: *zip-swap*)
done

lemma *zip-take*: (*zip xs ys*) ↓ *n* = *zip* (*xs* ↓ *n*) (*ys* ↓ *n*)
by (*rule take-zip*)

thm *nth-zip*
lemma *hd-zip*: [\![*xs* ≠ []; *ys* ≠ []]\!] ⟹ *hd* (*zip xs ys*) = (*hd xs*, *hd ys*)
by (*simp add*: *hd-conv-nth zip-not-empty-conv*)

lemma *map-id*: *map id xs* = *xs*
by (*simp add*: *id-def*)

lemma *map-id-subst*: *P* (*map id xs*) ⟹ *P xs*
by (*subst map-id*[*symmetric*])

lemma *map-one*: *map f* [*x*] = [*f x*]
by *simp*

lemma *map-last*: *xs* ≠ [] ⟹ *last* (*map f xs*) = *f* (*last xs*)
by (*rule last-map*)

lemma *filter-list-all*: *list-all P xs* ⟹ *filter P xs* = *xs*
by (*induct xs*, *simp*+)

lemma *filter-snoc*: *filter P* (*xs* @ [*x*]) = (*if P x then* (*filter P xs*) @ [*x*] *else filter P xs*)
by (*case-tac P x*, *simp*+)

lemma *filter-filter-eq*: *list-all* (λ*x*. *P x* = *Q x*) *xs* ⟹ *filter P xs* = *filter Q xs*
by (*induct xs*, *simp*+)

lemma *filter-nth*: ⋀*n*.

```
  n < length (filter P xs) ⟹
  (filter P xs) ! n =
  xs ! (LEAST k.
    k < length xs ∧
    n < card {i. i ≤ k ∧ i < length xs ∧ P (xs ! i)})
apply (induct xs rule: rev-induct, simp)
apply (rename-tac x xs n)
thm filter-snoc
apply (simp only: filter-snoc)
apply (simp split del: split-if)
apply (case-tac xs = [])
 apply (simp split del: split-if)
 apply (rule-tac
   t = λk i. i = 0 ∧ i ≤ k ∧ P ([x] ! i) and
   s = λk i. i = 0 ∧ P x
   in subst)
  apply (simp add: fun-eq-iff)
  apply fastsimp
 apply (fastsimp simp: Least-def)
apply (rule-tac
  t = λk. card {i. i ≤ k ∧ i < Suc (length xs) ∧ P ((xs @ [x]) ! i)} and
  s = λk. (card {i. i ≤ k ∧ i < length xs ∧ P (xs ! i)} +
           (if k ≥ length xs ∧ P x then Suc 0 else 0))
  in subst)
 apply (clarsimp simp: fun-eq-iff split del: split-if, rename-tac k)
 apply (simp split del: split-if add: less-Suc-eq conj-disj-distribL conj-disj-distribR
Collect-disj-eq)
 apply (subst card-Un-disjoint)
    apply (rule-tac n=length xs in bounded-nat-set-is-finite, blast)
   apply (rule-tac n=Suc (length xs) in bounded-nat-set-is-finite, blast)
  apply blast
 apply (rule-tac
   t = λi. i < length xs ∧ P ((xs @ [x]) ! i) and
   s = λi. i < length xs ∧ P (xs ! i)
   in subst)
  apply (rule fun-eq-iff[THEN iffD2])
  apply (fastsimp simp: nth-append1)
 apply (rule nat-add-left-cancel[THEN iffD2])
 apply (rule-tac
   t = λi. i = length xs ∧ i ≤ k ∧ P ((xs @ [x]) ! i) and
   s = λi. i = length xs ∧ i ≤ k ∧ P x
   in subst)
  apply (rule fun-eq-iff[THEN iffD2])
  apply fastsimp
 apply (case-tac length xs ≤ k)
  apply clarsimp
  apply (rule-tac
    t = λi. i = length xs ∧ i ≤ k and
    s = λi. i = length xs
```

```
    in subst)
   apply (rule fun-eq-iff[THEN iffD2])
   apply fastsimp
  apply simp
 apply simp
apply (simp split del: split-if add: less-Suc-eq conj-disj-distribL conj-disj-distribR)
apply (rule-tac
  t = λk. k < length xs ∧
        n < card {i. i ≤ k ∧ i < length xs ∧ P (xs ! i)} + (if length xs ≤ k ∧ P
x then Suc 0 else 0) and
  s = λk. k < length xs ∧ n < card {i. i ≤ k ∧ i < length xs ∧ P (xs ! i)}
  in subst)
 apply (simp add: fun-eq-iff)
apply (rule-tac
  t = λk. k = length xs ∧
        n < card {i. i ≤ k ∧ i < length xs ∧ P (xs ! i)} + (if length xs ≤ k ∧ P
x then Suc 0 else 0) and
  s = λk. k = length xs ∧
          n < card {i. i ≤ k ∧ i < length xs ∧ P (xs ! i)} + (if P x then Suc 0
else 0)
  in subst)
 apply (simp add: fun-eq-iff)
apply (case-tac n < length (filter P xs))
 apply (rule-tac
   t = (if P x then filter P xs @ [x] else filter P xs) ! n and
   s = (filter P xs) ! n
   in subst)
  apply (simp add: nth-append1)
 apply (simp split del: split-if)
 apply (subgoal-tac ∃k<length xs. n < card {i. i ≤ k ∧ i < length xs ∧ P (xs !
i)})
  prefer 2
  apply (rule-tac x=length xs − Suc 0 in exI)
  apply (simp add: length-filter-conv-card less-eq-le-pred[symmetric])
 apply (subgoal-tac ∃k≤length xs. n < card {i. i ≤ k ∧ i < length xs ∧ P (xs !
i)})
  prefer 2
  apply (blast intro: less-imp-le)
 thm Least-le-imp-le-disj
 apply (subst Least-le-imp-le-disj)
   apply simp
  apply simp
 thm nth-append1
 apply (rule sym, rule nth-append1)
 apply (rule LeastI2-ex, assumption)
 apply blast
apply (simp add: linorder-not-less)
apply (subgoal-tac P x)
 prefer 2
```

apply (*rule ccontr*, *simp*)
apply (*simp add*: *length-snoc*)
apply (*drule less-Suc-eq-le*[*THEN iffD1*], *drule-tac x*=*n* **in** *order-antisym*, *assumption*)
apply (*simp add*: *nth-append2*)
thm *length-filter-conv-card*
apply (*simp add*: *length-filter-conv-card*)
apply (*rule-tac*
t = λ*k*. *card* {*i*. *i* < *length xs* ∧ *P* (*xs* ! *i*)} < *card* {*i*. *i* ≤ *k* ∧ *i* < *length xs* ∧ *P* (*xs* ! *i*)} **and**
s = λ*k*. *False*
in *subst*)
apply (*rule fun-eq-iff*[*THEN iffD2*], *rule allI*, *rename-tac k*)
apply (*simp add*: *linorder-not-less*)
apply (*rule card-mono*)
apply *fastsimp*
apply *blast*
apply *simp*
apply (*rule-tac*
t = (*LEAST k*. *k* = *length xs* ∧
card {*i*. *i* < *length xs* ∧ *P* (*xs* ! *i*)} < *Suc* (*card* {*i*. *i* ≤ *k* ∧ *i* < *length xs* ∧ *P* (*xs* ! *i*)})) **and**
s = *length xs*
in *subst*)
apply (*rule sym*, *rule Least-equality*)
apply *simp*
apply (*rule le-imp-less-Suc*)
apply (*rule card-mono*)
apply *fastsimp*
apply *fastsimp*
apply *simp*
apply *simp*
done

9.1.3 Ordered lists

fun
list-ord :: ('*a* ⇒ '*a* ⇒ *bool*) ⇒ ('*a*::*ord*) *list* ⇒ *bool*
where
list-ord ord (*x1* # *x2* # *xs*) = (*ord x1 x2* ∧ *list-ord ord* (*x2* # *xs*))
| *list-ord ord xs* = *True*

thm *list-ord.simps*
definition *list-asc* :: ('*a*::*ord*) *list* ⇒ *bool* **where**
list-asc xs ≡ *list-ord* (*op* ≤) *xs*
definition *list-strict-asc* :: ('*a*::*ord*) *list* ⇒ *bool* **where**
list-strict-asc xs ≡ *list-ord* (*op* <) *xs*
value *list-asc* [*1*::*nat*, *2*, *2*]
value *list-strict-asc* [*1*::*nat*, *2*, *2*]

definition *list-desc* :: ('*a*::*ord*) *list* ⇒ *bool* **where**
list-desc xs ≡ *list-ord* (*op* ≥) *xs*
definition *list-strict-desc* :: ('*a*::*ord*) *list* ⇒ *bool* **where**
list-strict-desc xs ≡ *list-ord* (*op* >) *xs*

lemma *list-ord-Nil*: *list-ord ord* []
by *simp*
lemma *list-ord-one*: *list-ord ord* [*x*]
by *simp*
lemma *list-ord-Cons*:
list-ord ord (*x* # *xs*) =
(*xs* = [] ∨ (*ord x* (*hd xs*) ∧ *list-ord ord xs*))
by (*induct xs*, *simp*+)
lemma *list-ord-Cons-imp*: ⟦ *list-ord ord xs*; *ord x* (*hd xs*) ⟧ ⟹ *list-ord ord* (*x* # *xs*)
by (*induct xs*, *simp*+)
lemma *list-ord-append*: ⋀*ys*.
list-ord ord (*xs* @ *ys*) =
(*list-ord ord xs* ∧
(*ys* = [] ∨ (*list-ord ord ys* ∧ (*xs* = [] ∨ *ord* (*last xs*) (*hd ys*)))))
apply (*induct xs*, *fastsimp*)
apply (*case-tac xs*, *case-tac ys*, *fastsimp*+)
done
lemma *list-ord-snoc*:
list-ord ord (*xs* @ [*x*]) =
(*xs* = [] ∨ (*ord* (*last xs*) *x* ∧ *list-ord ord xs*))
by (*fastsimp simp*: *list-ord-append*)

lemma *list-ord-all-conv*:
(*list-ord ord xs*) = (∀ *n* < *length xs* − *1*. *ord* (*xs* ! *n*) (*xs* ! *Suc n*))
apply (*rule iffI*)
apply (*induct xs*, *simp*)
apply *clarsimp*
apply (*simp add*: *list-ord-Cons*)
apply (*erule disjE*, *simp*)
apply *clarsimp*
apply (*case-tac n*)
apply (*simp add*: *hd-conv-nth*)
apply *simp*
apply (*induct xs*, *simp*)
apply (*simp add*: *list-ord-Cons*)
apply (*case-tac xs* = [], *simp*)
apply (*drule meta-mp*)
apply (*intro allI impI*, *rename-tac n*)
apply (*drule-tac x*=*Suc n* **in** *spec*, *simp*)
apply (*drule-tac x*=*0* **in** *spec*)
apply (*simp add*: *hd-conv-nth*)
done

lemma *list-ord-imp*:
⟦ ⋀*x y*. *ord x y* ⟹ *ord′ x y*; *list-ord ord xs* ⟧ ⟹
list-ord ord′ xs
apply (*induct xs*, *simp*)
apply (*simp add*: *list-ord-Cons*)
apply *fastsimp*
done
corollary *list-strict-asc-imp-list-asc*:
list-strict-asc (*xs*::′*a*::*preorder list*) ⟹ *list-asc xs*
by (*unfold list-strict-asc-def list-asc-def*, *rule list-ord-imp*[*of op* <], *rule order-less-imp-le*)
corollary *list-strict-desc-imp-list-desc*:
list-strict-desc (*xs*::′*a*::*preorder list*) ⟹ *list-desc xs*
by (*unfold list-strict-desc-def list-desc-def*, *rule list-ord-imp*[*of op* >], *rule order-less-imp-le*)

lemma *list-ord-trans-imp*: ⋀*i*.
⟦ *transP ord*; *list-ord ord xs*; *j* < *length xs*; *i* < *j* ⟧ ⟹
ord (*xs* ! *i*) (*xs* ! *j*)
apply (*simp add*: *list-ord-all-conv*)
apply (*induct j*, *simp*)
apply (*case-tac j* < *i*, *simp*)
apply (*simp add*: *linorder-not-less*)
apply (*case-tac i* = *j*, *simp*)
thm *trans-def*
apply (*drule-tac x*=*i* **in** *meta-spec*, *simp*)
apply (*drule-tac x*=*j* **in** *spec*, *simp add*: *Suc-less-pred-conv*)
apply (*unfold trans-def*)
apply (*drule-tac x*=*xs* ! *i* **in** *spec*, *drule-tac x*=*xs* ! *j* **in** *spec*, *drule-tac x*=*xs* ! *Suc*
j **in** *spec*)
apply *simp*
done

lemma *list-ord-trans*:
transP ord ⟹
(*list-ord ord xs*) =
(∀*j* < *length xs*. ∀*i* < *j*. *ord* (*xs* ! *i*) (*xs* ! *j*))
apply (*rule iffI*)
apply (*simp add*: *list-ord-trans-imp*)
apply (*simp add*: *list-ord-all-conv*)
done

lemma *list-ord-trans-refl-le*:
⟦ *transP ord*; *reflP ord* ⟧ ⟹
(*list-ord ord xs*) =
(∀*j* < *length xs*. ∀*i* ≤ *j*. *ord* (*xs* ! *i*) (*xs* ! *j*))
apply (*subst list-ord-trans*, *simp*)
apply (*rule iffI*)
apply *clarsimp*
apply (*case-tac i* = *j*)
apply (*simp add*: *refl-on-def*)

apply *simp+*
done

lemma *list-ord-trans-refl-le-imp*:
⟦ *transP ord*; ⋀*x y*. *ord x y* ⟹ *ord′ x y*; *reflP ord′*;
list-ord ord xs ⟧ ⟹
(∀*j* < *length xs*. ∀*i* ≤ *j*. *ord′* (*xs* ! *i*) (*xs* ! *j*))
apply *clarify*
apply (*case-tac i* = *j*)
apply (*simp add*: *refl-on-def*)
thm *list-ord-trans-imp*
apply (*simp add*: *list-ord-trans-imp*)
done

corollary
list-asc-trans:
(*list-asc* (*xs*::′*a*::*preorder list*)) =
(∀*j* < *length xs*. ∀*i* < *j*. *xs* ! *i* ≤ *xs* ! *j*) **and**
list-strict-asc-trans:
(*list-strict-asc* (*xs*::′*a*::*preorder list*)) =
(∀*j* < *length xs*. ∀*i* < *j*. *xs* ! *i* < *xs* ! *j*) **and**
list-desc-trans:
(*list-desc* (*xs*::′*a*::*preorder list*)) =
(∀*j* < *length xs*. ∀*i* < *j*. *xs* ! *j* ≤ *xs* ! *i*) **and**
list-strict-desc-trans:
(*list-strict-desc* (*xs*::′*a*::*preorder list*)) =
(∀*j* < *length xs*. ∀*i* < *j*. *xs* ! *j* < *xs* ! *i*)
apply (*unfold list-asc-def list-strict-asc-def list-desc-def list-strict-desc-def*)
apply (*rule list-ord-trans*, *unfold trans-def*, *blast intro*: *order-trans order-less-trans*)+
done

corollary
list-asc-trans-le:
(*list-asc* (*xs*::′*a*::*preorder list*)) =
(∀*j* < *length xs*. ∀*i* ≤ *j*. *xs* ! *i* ≤ *xs* ! *j*) **and**
list-desc-trans-le:
(*list-desc* (*xs*::′*a*::*preorder list*)) =
(∀*j* < *length xs*. ∀*i* ≤ *j*. *xs* ! *j* ≤ *xs* ! *i*)
apply (*unfold list-asc-def list-strict-asc-def list-desc-def list-strict-desc-def*)
apply (*rule list-ord-trans-refl-le*, *unfold trans-def*, *blast intro*: *order-trans*, *simp add*: *refl-on-def*)+
done

corollary
list-strict-asc-trans-le:
(*list-strict-asc* (*xs*::′*a*::*preorder list*)) ⟹
(∀*j* < *length xs*. ∀*i* ≤ *j*. *xs* ! *i* ≤ *xs* ! *j*)
apply (*unfold list-strict-asc-def*)
thm *list-ord-trans-refl-le-imp*

apply (*rule list-ord-trans-refl-le-imp*[**where** *ord*=*op* ≤])
apply (*unfold trans-def*, *blast intro*: *order-trans*)
apply *assumption*
apply (*unfold refl-on-def*, *clarsimp*)
thm *list-ord-imp*
apply (*rule list-ord-imp*[**where** *ord*=*op* <], *simp-all add*: *less-imp-le*)
done

lemma *list-ord-le-sorted-eq*: *list-asc xs* = *sorted xs*
apply (*rule sym*)
apply (*simp add*: *list-asc-def*)
apply (*induct xs*, *simp*)
apply (*rename-tac x xs*)
apply (*simp add*: *list-ord-Cons sorted-Cons*)
apply (*case-tac xs* = [], *simp-all*)
apply (*case-tac list-ord op* ≤ *xs*, *simp-all*)
apply (*rule iffI*)
apply (*drule-tac x*=*hd xs* **in** *bspec*, *simp-all*)
apply *clarify*
apply (*drule in-set-conv-nth*[*THEN iffD1*], *clarsimp*, *rename-tac i1*)
apply (*simp add*: *hd-conv-nth*)
apply (*case-tac i1*, *simp*)
apply (*rename-tac i2*)
apply *simp*
apply (*fold list-asc-def*)
thm *list-asc-trans*
apply (*fastsimp simp*: *list-asc-trans*)
done
corollary *list-asc-upto*: *list-asc* [*m*..*n*]
by (*simp add*: *list-ord-le-sorted-eq*)

lemma *list-strict-asc-upt*: *list-strict-asc* [*m*..<*n*]
by (*simp add*: *list-strict-asc-def list-ord-all-conv*)
thm *list-strict-asc-imp-list-asc*[*OF list-strict-asc-upt*]

lemma *list-ord-distinct-aux*:
⟦ *irrefl* {(*a*, *b*). *ord a b*}; *transP ord*; *list-ord ord xs*;
i < *length xs*; *j* < *length xs*; *i* < *j* ⟧ ⟹
xs ! *i* ≠ *xs* ! *j*
apply (*subgoal-tac* ⋀*x y*. *ord x y* ⟹ *x* ≠ *y*)
prefer *2*
apply (*rule ccontr*)
apply (*simp add*: *irrefl-def*)
thm *list-ord-trans*
apply (*simp add*: *list-ord-trans*)
done
lemma *list-ord-distinct*:
⟦ *irrefl* {(*a*,*b*). *ord a b*}; *transP ord*; *list-ord ord xs* ⟧ ⟹

distinct xs
thm *distinct-conv-nth*
apply (*simp add*: *distinct-conv-nth*, *intro allI impI*, *rename-tac i j*)
apply (*drule neq-iff*[*THEN iffD1*], *erule disjE*)
thm *list-ord-distinct-aux*
apply (*simp add*: *list-ord-distinct-aux*)
thm *list-ord-distinct-aux*[*THEN not-sym*]
apply (*simp add*: *list-ord-distinct-aux*[*THEN not-sym*])
done

lemma *list-strict-asc-distinct*: *list-strict-asc* (*xs*::'*a*::*preorder list*) ⟹ *distinct xs*
apply (*rule-tac ord*=*op* < **in** *list-ord-distinct*)
apply (*unfold irrefl-def list-strict-asc-def trans-def*)
apply (*blast intro*: *less-trans*)+
done
lemma *list-strict-desc-distinct*: *list-strict-desc* (*xs*::'*a*::*preorder list*) ⟹ *distinct xs*
apply (*rule-tac ord*=*op* > **in** *list-ord-distinct*)
apply (*unfold irrefl-def list-strict-desc-def trans-def*)
apply (*blast intro*: *less-trans*)+
done

9.1.4 Additional definitions and results for sublists

primrec
sublist-list :: '*a list* ⇒ *nat list* ⇒ '*a list*
where
sublist-list xs [] = []
| *sublist-list xs* (*y* # *ys*) = (*xs* ! *y*) # (*sublist-list xs ys*)

value *sublist-list* [*0*::*int*,*10*::*int*,*20*,*30*,*40*,*50*] [*1*::*nat*,*2*,*3*]
value *sublist-list* [*0*::*int*,*10*::*int*,*20*,*30*,*40*,*50*] [*1*::*nat*,*1*,*2*,*3*]
value *sublist-list* [*0*::*int*,*10*::*int*,*20*,*30*,*40*,*50*] [*1*::*nat*,*1*,*2*,*3*,*10*]

thm *sublist-def*
term *map fst* (*filter* (λ*p*. *snd p* ∈ *A*) (*zip xs* [*0*..<*length xs*]))
term *map fst* ([*p*←(*zip xs* [*0*..<*length xs*]). (*snd p* ∈ *A*)])

lemma *sublist-list-length*: *length* (*sublist-list xs ys*) = *length ys*
by (*induct ys*, *simp-all*)

lemma *sublist-list-append*:
⋀*zs*. *sublist-list xs* (*ys* @ *zs*) = *sublist-list xs ys* @ *sublist-list xs zs*
by (*induct ys*, *simp-all*)

lemma *sublist-list-Nil*: *sublist-list xs* [] =[]
by *simp*

lemma *sublist-list-is-Nil-conv*:

(*sublist-list xs ys* = []) = (*ys* = [])
apply (*rule iffI*)
apply (*rule ccontr*)
apply (*clarsimp simp*: *neq-Nil-conv*)
apply *simp*
done

lemma *sublist-list-eq-imp-length-eq*:
sublist-list xs ys = *sublist-list xs zs* ⟹ *length ys* = *length zs*
by (*drule arg-cong*[**where** *f*=*length*], *simp add*: *sublist-list-length*)

lemma *sublist-list-nth*:
⋀*n*. *n* < *length ys* ⟹ *sublist-list xs ys* ! *n* = *xs* ! (*ys* ! *n*)
apply (*induct ys*, *simp*)
apply (*case-tac n*, *simp-all*)
done

lemma *take-drop-eq-sublist-list*:
m + *n* ≤ *length xs* ⟹ *xs* ↑ *m* ↓ *n* = *sublist-list xs* [*m*..<*m*+*n*]
apply (*insert length-upt*[*of m m*+*n*])
apply (*simp add*: *expand-list-eq*)
apply (*simp add*: *sublist-list-length*)
apply (*frule add-le-imp-le-diff2*)
apply (*simp add*: *min-eqR*)
apply (*clarsimp*, *rename-tac i*)
thm *sublist-list-nth*
apply (*simp add*: *sublist-list-nth*)
done

primrec
sublist-list-if :: *'a list* ⇒ *nat list* ⇒ *'a list*
where
sublist-list-if xs [] = []
| *sublist-list-if xs* (*y* # *ys*) =
(*if y* < *length xs then* (*xs* ! *y*) # (*sublist-list-if xs ys*)
else (*sublist-list-if xs ys*))

value *sublist-list-if* [*0*::*int*,*10*::*int*,*20*,*30*,*40*,*50*] [*1*::*nat*,*2*,*3*]
value *sublist-list-if* [*0*::*int*,*10*::*int*,*20*,*30*,*40*,*50*] [*1*::*nat*,*1*,*2*,*3*]
value *sublist-list-if* [*0*::*int*,*10*::*int*,*20*,*30*,*40*,*50*] [*1*::*nat*,*1*,*2*,*3*,*10*]

lemma *sublist-list-if-sublist-list-filter-conv*: ⋀*xs*.
sublist-list-if xs ys = *sublist-list xs* (*filter* (λ*i*. *i* < *length xs*) *ys*)
by (*induct ys*, *simp*+)
corollary *sublist-list-if-sublist-list-eq*: ⋀*xs*.
list-all (λ*i*. *i* < *length xs*) *ys* ⟹
sublist-list-if xs ys = *sublist-list xs ys*

by (*simp add*: *sublist-list-if-sublist-list-filter-conv filter-list-all*)
corollary *sublist-list-if-sublist-list-eq2*: $\bigwedge xs.$
$\forall n{<}length\ ys.\ ys\ !\ n < length\ xs \Longrightarrow$
sublist-list-if xs ys = sublist-list xs ys
thm *list-all-conv*[*THEN iffD2*]
by (*rule sublist-list-if-sublist-list-eq*, *rule list-all-conv*[*THEN iffD2*])

lemma *sublist-list-if-Nil-left*: *sublist-list-if* [] *ys* = []
by (*induct ys*, *simp+*)
lemma *sublist-list-if-Nil-right*: *sublist-list-if xs* [] = []
by *simp*

lemma *sublist-list-if-length*:
length (*sublist-list-if xs ys*) = *length* (*filter* (λ*i*. *i* < *length xs*) *ys*)
by (*simp add*: *sublist-list-if-sublist-list-filter-conv sublist-list-length*)
lemma *sublist-list-if-append*:
sublist-list-if xs (*ys* @ *zs*) = *sublist-list-if xs ys* @ *sublist-list-if xs zs*
by (*simp add*: *sublist-list-if-sublist-list-filter-conv sublist-list-append*)
lemma *sublist-list-if-snoc*:
sublist-list-if xs (*ys* @ [*y*]) = *sublist-list-if xs ys* @ (*if y* < *length xs then* [*xs* ! *y*]
else [])
by (*simp add*: *sublist-list-if-append*)

lemma *sublist-list-if-is-Nil-conv*:
(*sublist-list-if xs ys* = []) = (*list-all* (λ*i*. *length xs* ≤ *i*) *ys*)
by (*simp add*: *sublist-list-if-sublist-list-filter-conv sublist-list-is-Nil-conv filter-empty-conv list-all-iff linorder-not-less*)

lemma *sublist-list-if-nth*:
n < *length* ((*filter* (λ*i*. *i* < *length xs*) *ys*)) ⟹
sublist-list-if xs ys ! *n* = *xs* ! ((*filter* (λ*i*. *i* < *length xs*) *ys*) ! *n*)
by (*simp add*: *sublist-list-if-sublist-list-filter-conv sublist-list-nth*)

lemma *take-drop-eq-sublist-list-if*:
m + *n* ≤ *length xs* ⟹ *xs* ↑ *m* ↓ *n* = *sublist-list-if xs* [*m*..<*m*+*n*]
thm *take-drop-eq-sublist-list*
by (*simp add*: *sublist-list-if-sublist-list-filter-conv take-drop-eq-sublist-list*)

lemma *sublist-empty-conv*: (*sublist xs I* = []) = (∀ *i*∈*I*. *length xs* ≤ *i*)
by (*fastsimp simp*: *set-empty*[*symmetric*] *set-sublist linorder-not-le*[*symmetric*])

thm *sublist-singleton*
lemma *sublist-singleton2*: *sublist xs* {*y*} = (*if y* < *length xs then* [*xs* ! *y*] *else* [])
apply (*unfold sublist-def*)
apply (*induct xs rule*: *rev-induct*, *simp*)
apply (*simp add*: *nth-append*)
done

lemma *sublist-take-eq*:
⟦ *finite I*; *Max I* < *n* ⟧ ⟹ *sublist* (*xs* ↓ *n*) *I* = *sublist xs I*
apply (*case-tac I* = {}, *simp*)
apply (*case-tac n* < *length xs*)
prefer *2*
apply *simp*
thm *append-take-drop-id*
apply (*rule-tac*
t = *sublist xs I* **and**
s = *sublist* (*xs* ↓ *n* @ *xs* ↑ *n*) *I*
in *subst*)
apply *simp*
apply (*subst sublist-append*)
apply (*simp add*: *min-eqR*)
apply (*rule-tac t*={*j*. *j* + *n* ∈ *I*} **and** *s*={} **in** *subst*)
apply *blast*
apply *simp*
done

lemma *sublist-drop-eq*:
n ≤ *iMin I* ⟹ *sublist* (*xs* ↑ *n*) {*j*. *j* + *n* ∈ *I*} = *sublist xs I*
apply (*case-tac I* = {}, *simp*)
apply (*case-tac n* < *length xs*)
prefer *2*
apply (*simp add*: *sublist-def filter-empty-conv linorder-not-less*)
apply (*clarsimp*, *rename-tac a b*)
thm *set-zip-rightD*
apply (*drule set-zip-rightD*)
apply *fastsimp*
apply (*rule-tac*
t = *sublist xs I* **and**
s = *sublist* (*xs* ↓ *n* @ *xs* ↑ *n*) *I*
in *subst*)
apply *simp*
apply (*subst sublist-append*)
apply (*fastsimp simp*: *sublist-empty-conv min-eqR*)
done

lemma *sublist-cut-less-eq*:
length xs ≤ *n* ⟹ *sublist xs* (*I* ↓< *n*) = *sublist xs I*
apply (*simp add*: *sublist-def cut-less-mem-iff*)
apply (*rule-tac f*=λ*xs*. *map fst xs* **in** *arg-cong*)
thm *filter-filter-eq*
apply (*rule filter-filter-eq*)
apply (*simp add*: *list-all-conv*)
done

lemma *sublist-disjoint-Un*:
⟦ *finite A*; *Max A* < *iMin B* ⟧ ⟹ *sublist xs* (*A* ∪ *B*) = *sublist xs A* @ *sublist xs*

B
apply (*case-tac* $A = \{\}$, *simp*)
apply (*case-tac* $B = \{\}$, *simp*)
apply (*case-tac length xs* $\leq$ *iMin B*)
thm *sublist-cut-less-eq*
apply (*subst sublist-cut-less-eq*[*of xs iMin B*, *symmetric*], *assumption*)
apply (*simp* (*no-asm-simp*) *add*: *cut-less-Un cut-less-Min-empty cut-less-Max-all*)
apply (*simp add*: *sublist-empty-conv iMin-ge-iff*)
apply (*simp add*: *linorder-not-le*)
thm *sublist-append*
apply (*rule-tac*
$t = sublist\ xs\ (A \cup B)$ **and**
$s = sublist\ (xs \downarrow (iMin\ B) @ xs \uparrow (iMin\ B))\ (A \cup B)$
in *subst*)
apply *simp*
apply (*subst sublist-append*)
apply (*simp add*: *min-eqR*)
thm *sublist-cut-less-eq*
apply (*subst sublist-cut-less-eq*[**where** *xs*=*xs* $\downarrow$ *iMin B* **and** *n*=*iMin B*, *symmetric*], *simp*)
apply (*simp add*: *cut-less-Un cut-less-Min-empty cut-less-Max-all*)
thm *sublist-take-eq*
apply (*simp add*: *sublist-take-eq*)
apply (*rule-tac*
$t = \lambda j.\ j + iMin\ B \in A \vee j + iMin\ B \in B$ **and**
$s = \lambda j.\ j + iMin\ B \in B$
in *subst*)
apply (*force simp*: *fun-eq-iff*)
thm *sublist-drop-eq*
apply (*simp add*: *sublist-drop-eq*)
done
corollary *sublist-disjoint-insert-left*:
$[\![\ finite\ I;\ x < iMin\ I\ ]\!] \Longrightarrow sublist\ xs\ (insert\ x\ I) = sublist\ xs\ \{x\} @ sublist\ xs\ I$
apply (*rule-tac t*=*insert x I* **and** *s*=$\{x\} \cup I$ **in** *subst*, *simp*)
apply (*subst sublist-disjoint-Un*)
apply *simp-all*
done
corollary *sublist-disjoint-insert-right*:
$[\![\ finite\ I;\ Max\ I < x\ ]\!] \Longrightarrow sublist\ xs\ (insert\ x\ I) = sublist\ xs\ I @ sublist\ xs\ \{x\}$
apply (*rule-tac t*=*insert x I* **and** *s*=$I \cup \{x\}$ **in** *subst*, *simp*)
apply (*subst sublist-disjoint-Un*)
apply *simp-all*
done

lemma *sublist-all*: $\{..<length\ xs\} \subseteq I \Longrightarrow sublist\ xs\ I = xs$
apply (*case-tac xs* = [], *simp*)
apply (*rule-tac*
$t = I$ **and**
$s = I \downarrow< (length\ xs) \cup I \downarrow\geq (length\ xs)$

in *subst*)
apply (*simp add*: *cut-less-cut-ge-ident*)
apply (*rule-tac*
t = *I* ↓< *length xs* **and**
s = {..<*length xs*}
in *subst*)
apply *blast*
apply (*case-tac I* ↓≥ (*length xs*) = {}, *simp*)
apply (*subst sublist-disjoint-Un*[*OF finite-lessThan*])
apply (*rule less-imp-Max-less-iMin*[*OF finite-lessThan*])
apply *blast*
apply *blast*
apply (*blast intro*: *less-le-trans*)
apply (*fastsimp simp*: *sublist-empty-conv*)
done
corollary *sublist-UNIV*: *sublist xs UNIV* = *xs*
by (*rule sublist-all*[*OF subset-UNIV*])

lemma *sublist-list-sublist-eq*: ⋀*xs*.
list-strict-asc ys ⟹ *sublist-list-if xs ys* = *sublist xs* (*set ys*)
apply (*case-tac xs* = [])
apply (*simp add*: *sublist-list-if-Nil-left*)
apply (*induct ys rule*: *rev-induct*, *simp*)
apply (*rename-tac y ys xs*)
apply (*case-tac ys* = [])
apply (*simp add*: *sublist-singleton2*)
apply (*unfold list-strict-asc-def*)
apply (*simp add*: *sublist-list-if-snoc split del*: *split-if*)
thm *list-ord-append*
apply (*frule list-ord-append*[*THEN iffD1*])
apply (*clarsimp split del*: *split-if*)
apply (*subst sublist-disjoint-insert-right*)
apply *simp*
apply (*clarsimp simp*: *in-set-conv-nth*, *rename-tac i*)
thm *list-strict-asc-trans*[*unfolded list-strict-asc-def*, *THEN iffD1*, *rule-format*]
apply (*drule-tac i*=*i* **and** *j*=*length ys* **in** *list-strict-asc-trans*[*unfolded list-strict-asc-def*, *THEN iffD1*, *rule-format*])
apply (*simp add*: *nth-append split del*: *split-if*)+
apply (*simp add*: *sublist-singleton2*)
done
lemma *set-sublist-list-if*: ⋀*xs*. *set* (*sublist-list-if xs ys*) = {*xs* ! *i* |*i*. *i* < *length xs* ∧ *i* ∈ *set ys*}
apply (*induct ys*, *simp-all*)
apply *blast*
done

lemma *set-sublist-list*:
list-all (λ*i*. *i* < *length xs*) *ys* ⟹
set (*sublist-list xs ys*) = {*xs* ! *i* |*i*. *i* < *length xs* ∧ *i* ∈ *set ys*}

by (*simp add*: *sublist-list-if-sublist-list-eq*[*symmetric*] *set-sublist-list-if*)

lemma *set-sublist-list-if-eq-set-sublist*: *set* (*sublist-list-if xs ys*) = *set* (*sublist xs* (*set ys*))
by (*simp add*: *set-sublist set-sublist-list-if*)
lemma *set-sublist-list-eq-set-sublist*:
list-all (λ*i*. *i* < *length xs*) *ys* ⟹
set (*sublist-list xs ys*) = *set* (*sublist xs* (*set ys*))
by (*simp add*: *sublist-list-if-sublist-list-eq*[*symmetric*] *set-sublist-list-if-eq-set-sublist*)

9.1.5 Natural set images with lists

definition
f-image :: 'a *list* ⇒ *nat set* ⇒ 'a *set* (**infixr** ‘*f* 90)
where
xs ‘*f* *A* ≡ {*y*. ∃ *n*∈*A*. *n* < *length xs* ∧ *y* = *xs* ! *n* }

abbreviation
f-range :: 'a *list* ⇒ 'a *set*
where
f-range xs ≡ *f-image xs UNIV*

thm *Set.image-eqI*
lemma *f-image-eqI*[*simp*, *intro*]:
⟦ *x* = *xs* ! *n*; *n* ∈ *A*; *n* < *length xs* ⟧ ⟹ *x* ∈ *xs* ‘*f* *A*
by (*unfold f-image-def*, *blast*)

thm *Set.imageI*
lemma *f-imageI*: ⟦ *n* ∈ *A*; *n* < *length xs* ⟧ ⟹ *xs* ! *n* ∈ *xs* ‘*f* *A*
by *blast*

thm *Set.rev-image-eqI*
lemma *rev-f-imageI*: ⟦ *n* ∈ *A*; *n* < *length xs*; *x* = *xs* ! *n* ⟧ ⟹ *x* ∈ *xs* ‘*f* *A*
by (*rule f-image-eqI*)

thm *Set.imageE*
lemma *f-imageE*[*elim*!]:
⟦ *x* ∈ *xs* ‘*f* *A*; ⋀*n*. ⟦ *x* = *xs* ! *n*; *n* ∈ *A*; *n* < *length xs* ⟧ ⟹ *P* ⟧ ⟹ *P*
by (*unfold f-image-def*, *blast*)

thm *Set.image-Un*
lemma *f-image-Un*: *xs* ‘*f* (*A* ∪ *B*) = *xs* ‘*f* *A* ∪ *xs* ‘*f* *B*
by *blast*

thm *Set.image-mono*
lemma *f-image-mono*: *A* ⊆ *B* ==> *xs* ‘*f* *A* ⊆ *xs* ‘*f* *B*
by *blast*

thm *Set.image-iff*

lemma *f-image-iff*: $(x \in xs\ `^f\ A) = (\exists n{\in}A.\ n < length\ xs \wedge x = xs\ !\ n)$
by *blast*

thm *Set.image-subset-iff*
lemma *f-image-subset-iff*:
$(xs\ `^f\ A \subseteq B) = (\forall n{\in}A.\ n < length\ xs \longrightarrow xs\ !\ n \in B)$
by *blast*
thm *Set.subset-image-iff*
lemma *subset-f-image-iff*: $(B \subseteq xs\ `^f\ A) = (\exists A'{\subseteq}A.\ B = xs\ `^f\ A')$
apply (*rule iffI*)
apply (*rule-tac x*=$\{\ n.\ n \in A \wedge n < length\ xs \wedge xs\ !\ n \in B\ \}$ **in** *exI*)
apply *blast*
apply (*blast intro*: *f-image-mono*)
done

thm *image-subsetI*
lemma *f-image-subsetI*:
$[\![\ \bigwedge n.\ n \in A \wedge n < length\ xs \Longrightarrow xs\ !\ n \in B\ ]\!] \Longrightarrow xs\ `^f\ A \subseteq B$
by *blast*

thm *Set.image-empty*
lemma *f-image-empty*: $xs\ `^f\ \{\} = \{\}$
by *blast*

thm *Set.image-insert*
lemma *f-image-insert-if*:
$xs\ `^f\ (insert\ n\ A) = ($
if $n < length\ xs$ *then* $insert\ (xs\ !\ n)\ (xs\ `^f\ A)$ *else* $(xs\ `^f\ A))$
by (*split split-if*, *blast*)
lemma *f-image-insert-eq1*:
$n < length\ xs \Longrightarrow xs\ `^f\ (insert\ n\ A) = insert\ (xs\ !\ n)\ (xs\ `^f\ A)$
by (*simp add*: *f-image-insert-if*)
lemma *f-image-insert-eq2*:
$length\ xs \leq n \Longrightarrow xs\ `^f\ (insert\ n\ A) = (xs\ `^f\ A)$
by (*simp add*: *f-image-insert-if*)

thm *Set.insert-image*
lemma *insert-f-image*:
$[\![\ n \in A;\ n < length\ xs\ ]\!] \Longrightarrow insert\ (xs\ !\ n)\ (xs\ `^f\ A) = (xs\ `^f\ A)$
by *blast*
thm *Set.image-is-empty*
lemma *f-image-is-empty*: $(xs\ `^f\ A = \{\}) = (\{x.\ x \in A \wedge x < length\ xs\} = \{\})$
by *blast*

thm *Set.image-Collect*
lemma *f-image-Collect*: $xs\ `^f\ \{n.\ P\ n\} = \{xs\ !\ n\ |n.\ P\ n \wedge n < length\ xs\}$
by *blast*

lemma *f-image-eq-set*: $\forall n{<}length\ xs.\ n \in A \Longrightarrow xs\ `^{f}\ A = set\ xs$
by (*fastsimp simp*: *in-set-conv-nth*)
lemma *f-range-eq-set*: *f-range xs* = *set xs*
by (*simp add*: *f-image-eq-set*)

lemma *f-image-eq-set-sublist*: *xs* `^{f} *A* = *set* (*sublist xs A*)
by (*unfold set-sublist*, *blast*)
lemma *f-image-eq-set-sublist-list-if*: *xs* `^{f} (*set ys*) = *set* (*sublist-list-if xs ys*)
by (*simp add*: *set-sublist-list-if-eq-set-sublist f-image-eq-set-sublist*)
lemma *f-image-eq-set-sublist-list*:
list-all ($\lambda i.\ i < length\ xs$) *ys* $\Longrightarrow$ *xs* `^{f} (*set ys*) = *set* (*sublist-list xs ys*)
by (*simp add*: *sublist-list-if-sublist-list-eq f-image-eq-set-sublist-list-if*)

thm *Set.range-eqI*
lemma *f-range-eqI*: $[\![\ x = xs\ !\ n;\ n < length\ xs\ ]\!] \Longrightarrow x \in$ *f-range xs*
by *blast*
thm *Set.rangeI*
lemma *f-rangeI*: $n < length\ xs \Longrightarrow xs\ !\ n \in$ *f-range xs*
by *blast*
thm *Set.rangeE*
lemma *f-rangeE*[*elim?*]:
$[\![\ x \in$ *f-range xs*; $\bigwedge n.\ [\![\ n < length\ xs; x = xs\ !\ n\ ]\!] \Longrightarrow P\ ]\!] \Longrightarrow P$
by *blast*

9.1.6 Mapping lists of functions to lists

primrec
map-list :: $('a \Rightarrow 'b)\ list \Rightarrow 'a\ list \Rightarrow 'b\ list$
where
map-list [] *xs* = []
| *map-list* (*f* # *fs*) *xs* = *f* (*hd xs*) # *map-list fs* (*tl xs*)

lemma *map-list-Nil*: *map-list* [] *xs* = []
by *simp*
lemma *map-list-Cons-Cons*:
map-list (*f* # *fs*) (*x* # *xs*) =
(*f x*) # *map-list fs xs*
by *simp*

lemma *map-list-length*: $\bigwedge xs.$
length (*map-list fs xs*) = *length fs*
by (*induct fs*, *simp+*)
corollary *map-list-empty-conv*:
(*map-list fs xs* = []) = (*fs* = [])
by (*simp del*: *length-0-conv add*: *length-0-conv*[*symmetric*] *map-list-length*)

corollary *map-list-not-empty-conv*:
(*map-list fs xs* ≠ []) = (*fs* ≠ [])
by (*simp add*: *map-list-empty-conv*)

lemma *map-list-nth*: ⋀*n xs*.
⟦ *n* < *length fs*; *n* < *length xs* ⟧ ⟹
(*map-list fs xs* ! *n*) =
(*fs* ! *n*) (*xs* ! *n*)
apply (*induct fs*, *simp*+)
apply (*case-tac n*)
apply (*simp add*: *hd-conv-nth*)
apply (*simp add*: *nth-tl-eq-nth-Suc2*)
done

lemma *map-list-xs-take*: ⋀*n xs*.
length fs ≤ *n* ⟹
map-list fs (*xs* ↓ *n*) =
map-list fs xs
apply (*induct fs*, *simp*+)
apply (*rename-tac f fs n xs*)
apply (*simp add*: *tl-take*)
thm *arg-cong*
apply (*rule-tac f*=*f* **in** *arg-cong*)
apply (*case-tac xs* = [], *simp*)
apply (*simp add*: *hd-conv-nth*)
done

lemma *map-list-take*: ⋀*n xs*.
(*map-list fs xs*) ↓ *n* =
(*map-list* (*fs* ↓ *n*) *xs*)
apply (*induct fs*, *simp*)
apply (*case-tac n*, *simp*+)
done
lemma *map-list-take-take*: ⋀*n xs*.
(*map-list fs xs*) ↓ *n* =
(*map-list* (*fs* ↓ *n*) (*xs* ↓ *n*))
by (*simp add*: *map-list-take map-list-xs-take*)
lemma *map-list-drop*: ⋀*n xs*.
(*map-list fs xs*) ↑ *n* =
(*map-list* (*fs* ↑ *n*) (*xs* ↑ *n*))
apply (*induct fs*, *simp*)
apply (*case-tac n*)
apply (*simp add*: *drop-Suc*)+
done

lemma *map-list-append-append*: ⋀*xs1* .
length fs1 = *length xs1* ⟹

map-list (*fs1* @ *fs2*) (*xs1* @ *xs2*) =
map-list fs1 xs1 @
map-list fs2 xs2
apply (*induct fs1*, *simp*+)
apply (*case-tac xs1*, *simp*+)
done
lemma *map-list-snoc-snoc*:
length fs = *length xs* ⟹
map-list (*fs* @ [*f*]) (*xs* @ [*x*]) =
map-list fs xs @ [*f x*]
by (*simp add*: *map-list-append-append*)
lemma *map-list-snoc*: ⋀*xs*.
length fs < *length xs* ⟹
map-list (*fs* @ [*f*]) *xs* =
map-list fs xs @ [*f* (*xs* ! (*length fs*))]
apply (*induct fs*)
apply (*simp add*: *hd-conv-nth*)
apply (*simp add*: *nth-tl-eq-nth-Suc2*)
done

lemma *map-list-Cons-if*:
map-list fs (*x* # *xs*) =
(*if* (*fs* = []) *then* [] *else* (
((*hd fs*) *x*) # *map-list* (*tl fs*) *xs*))
by (*case-tac fs*, *simp*+)
lemma *map-list-Cons-not-empty*:
fs ≠ [] ⟹
map-list fs (*x* # *xs*) =
((*hd fs*) *x*) # *map-list* (*tl fs*) *xs*
by (*simp add*: *map-list-Cons-if*)

lemma *map-eq-map-list-take*: ⋀*xs*.
⟦ *length fs* ≤ *length xs*; *list-all* (λ*x*. *x* = *f*) *fs* ⟧ ⟹
map-list fs xs = *map f* (*xs* ↓ *length fs*)
apply (*induct fs*, *simp*+)
apply (*case-tac xs*, *simp*+)
done
lemma *map-eq-map-list-take2*:
⟦ *length fs* = *length xs*; *list-all* (λ*x*. *x* = *f*) *fs* ⟧ ⟹
map-list fs xs = *map f xs*
by (*simp add*: *map-eq-map-list-take*)
lemma *map-eq-map-list-replicate*:
map-list ($f^{length\ xs}$) *xs* = *map f xs*
by (*induct xs*, *simp*+)

9.1.7 Mapping functions with two arguments to lists

primrec *map2* ::

(∗ *Function taking two parameters* ∗)
('*a* ⇒ '*b* ⇒ '*c*) ⇒
(∗ *Lists of parameters* ∗)
'*a list* ⇒ '*b list* ⇒
'*c list*
where
map2 f [] *ys* = []
| *map2 f* (*x* # *xs*) *ys* = *f x* (*hd ys*) # *map2 f xs* (*tl ys*)

lemma *map2-map-list-conv*: ⋀*ys*. *map2 f xs ys* = *map-list* (*map f xs*) *ys*
by (*induct xs*, *simp*+)

lemma *map2-Nil*: *map2 f* [] *ys* = []
by *simp*
lemma *map2-Cons-Cons*:
map2 f (*x* # *xs*) (*y* # *ys*) =
(*f x y*) # *map2 f xs ys*
by *simp*

lemma *map2-length*: ⋀*ys*. *length* (*map2 f xs ys*) = *length xs*
by (*induct xs*, *simp*+)
corollary *map2-empty-conv*:
(*map2 f xs ys* = []) = (*xs* = [])
by (*simp del*: *length-0-conv add*: *length-0-conv*[*symmetric*] *map2-length*)
corollary *map2-not-empty-conv*:
(*map2 f xs ys* ≠ []) = (*xs* ≠ [])
by (*simp add*: *map2-empty-conv*)

lemma *map2-nth*: ⋀*n ys*.
⟦ *n* < *length xs*; *n* < *length ys* ⟧ ⟹
(*map2 f xs ys* ! *n*) =
f (*xs* ! *n*) (*ys* ! *n*)
thm *map-list-nth*
by (*simp add*: *map2-map-list-conv map-list-nth*)

lemma *map2-ys-take*: ⋀*n ys*.
length xs ≤ *n* ⟹
map2 f xs (*ys* ↓ *n*) =
map2 f xs ys
thm *map-list-xs-take*
by (*simp add*: *map2-map-list-conv map-list-xs-take*)

lemma *map2-take*: ⋀*n ys*.
(*map2 f xs ys*) ↓ *n* =
(*map2 f* (*xs* ↓ *n*) *ys*)
thm *map-list-take*
by (*simp add*: *map2-map-list-conv take-map map-list-take*)
lemma *map2-take-take*: ⋀*n ys*.

(*map2 f xs ys*) ↓ *n* =
(*map2 f* (*xs* ↓ *n*) (*ys* ↓ *n*))
by (*simp add*: *map2-take map2-ys-take*)
lemma *map2-drop*: ⋀*n ys*.
(*map2 f xs ys*) ↑ *n* =
(*map2 f* (*xs* ↑ *n*) (*ys* ↑ *n*))
thm *map-list-drop*
by (*simp add*: *map2-map-list-conv map-list-drop drop-map*)

lemma *map2-append-append*: ⋀*ys1* .
length xs1 = *length ys1* ⟹
map2 f (*xs1* @ *xs2*) (*ys1* @ *ys2*) =
map2 f xs1 ys1 @
map2 f xs2 ys2
thm *map-list-append-append*
by (*simp add*: *map2-map-list-conv map-list-append-append*)
lemma *map2-snoc-snoc*:
length xs = *length ys* ⟹
map2 f (*xs* @ [*x*]) (*ys* @ [*y*]) =
map2 f xs ys @
[*f x y*]
by (*simp add*: *map2-append-append*)
lemma *map2-snoc*: ⋀*ys*.
length xs < *length ys* ⟹
map2 f (*xs* @ [*x*]) *ys* =
map2 f xs ys @
[*f x* (*ys* ! (*length xs*))]
thm *map-list-snoc*
by (*simp add*: *map2-map-list-conv map-list-snoc*)

lemma *map2-Cons-if*:
map2 f xs (*y* # *ys*) =
(*if* (*xs* = []) *then* [] *else* (
(*f* (*hd xs*) *y*) # *map2 f* (*tl xs*) *ys*))
by (*case-tac xs*, *simp*+)
lemma *map2-Cons-not-empty*:
xs ≠ [] ⟹
map2 f xs (*y* # *ys*) =
(*f* (*hd xs*) *y*) # *map2 f* (*tl xs*) *ys*
by (*simp add*: *map2-Cons-if*)

lemma *map2-append1-take-drop*:
length xs1 ≤ *length ys* ⟹
map2 f (*xs1* @ *xs2*) *ys* =
map2 f xs1 (*ys* ↓ *length xs1*) @
map2 f xs2 (*ys* ↑ *length xs1*)
thm *map2-append-append*
thm *append-take-drop-id*

apply (*rule-tac*
  t = *map2 f* (*xs1* @ *xs2*) *ys* **and**
  s = *map2 f* (*xs1* @ *xs2*) (*ys* ↓ *length xs1* @ *ys* ↑ *length xs1*)
  in *subst*)
 apply *simp*
apply (*simp add*: *map2-append-append del*: *append-take-drop-id*)
done
lemma *map2-append2-take-drop*:
  length ys1 ≤ *length xs* ⟹
  map2 f xs (*ys1* @ *ys2*) =
  map2 f (*xs* ↓ *length ys1*) *ys1* @
  map2 f (*xs* ↑ *length ys1*) *ys2*
apply (*rule-tac*
  t = *map2 f xs* (*ys1* @ *ys2*) **and**
  s = *map2 f* (*xs* ↓ *length ys1* @ *xs* ↑ *length ys1*) (*ys1* @ *ys2*)
  in *subst*)
 apply *simp*
apply (*simp add*: *map2-append-append del*: *append-take-drop-id*)
done

thm *List.map-cong*
lemma *map2-cong*:
  ⟦ *xs1* = *xs2*; *ys1* = *ys2*; *length xs2* ≤ *length ys2*;
    ⋀*x y*. ⟦ *x* ∈ *set xs2*; *y* ∈ *set ys2* ⟧ ⟹ *f x y* = *g x y* ⟧ ⟹
  map2 f xs1 ys1 = *map2 g xs2 ys2*
by (*simp* (*no-asm-simp*) *add*: *expand-list-eq map2-length map2-nth*)

thm *List.map-eq-conv*
lemma *map2-eq-conv*:
  length xs ≤ *length ys* ⟹
  (*map2 f xs ys* = *map2 g xs ys*) = (∀ *i*<*length xs*. *f* (*xs* ! *i*) (*ys* ! *i*) = *g* (*xs* ! *i*)
(*ys* ! *i*))
by (*simp add*: *expand-list-eq map2-length map2-nth*)

thm *List.map-replicate*
lemma *map2-replicate*: *map2 f* x^n y^n = $(f\ x\ y)^n$
by (*induct n*, *simp*+)

lemma *map2-zip-conv*: ⋀*ys*.
  length xs ≤ *length ys* ⟹
  map2 f xs ys = *map* (λ(*x*,*y*). *f x y*) (*zip xs ys*)
apply (*induct xs*, *simp*)
apply (*case-tac ys*, *simp*+)
done

lemma *map2-rev*: ⋀*ys*.
  length xs = *length ys* ⟹
  rev (*map2 f xs ys*) = *map2 f* (*rev xs*) (*rev ys*)
apply (*induct xs*, *simp*)

apply (*case-tac ys*, *simp*)
apply (*simp add*: *map2-Cons-Cons map2-snoc-snoc*)
done

end

10 InfiniteSet2: Set operations with results of type inat

theory *InfiniteSet2*
imports *SetInterval2*
begin

10.1 Set operations with *inat*

10.1.1 Basic definitions

definition
  icard :: *'a set* ⇒ *inat*
where
  icard A ≡ *if finite A then Fin* (*card A*) *else* ∞

10.2 Results for *icard*

lemma *icard-UNIV-nat*: *icard* (*UNIV*::*nat set*) = ∞
by (*simp add*: *icard-def*)

lemma *icard-finite-conv*: (*icard A* = *Fin* (*card A*)) = *finite A*
by (*case-tac finite A*, *simp-all add*: *icard-def*)
lemma *icard-infinite-conv*: (*icard A* = ∞) = *infinite A*
by (*case-tac finite A*, *simp-all add*: *icard-def*)

corollary *icard-finite*: *finite A* ⟹ *icard A* = *Fin* (*card A*)
by (*rule icard-finite-conv*[*THEN iffD2*])
corollary *icard-infinite*[*simp*]: *infinite A* ⟹ *icard A* = ∞
by (*rule icard-infinite-conv*[*THEN iffD2*])

lemma *icard-eq-Fin-imp*: *icard A* = *Fin n* ⟹ *finite A*
by (*case-tac finite A*, *simp-all*)
lemma *icard-eq-Infty-imp*: *icard A* = ∞ ⟹ *infinite A*
by (*rule icard-infinite-conv*[*THEN iffD1*])

lemma *icard-the-Fin*: *finite A* ⟹ *the-Fin* (*icard A*) = *card A*
by (*simp add*: *icard-def*)

lemma *icard-eq-Fin-imp-card*: *icard A* = *Fin n* ⟹ *card A* = *n*
by (*frule icard-eq-Fin-imp*, *simp add*: *icard-finite*)

lemma *icard-eq-Fin-card-conv*: $0 < n \Longrightarrow (icard\ A = Fin\ n) = (card\ A = n)$
apply (*rule iffI*)
apply (*simp add*: *icard-eq-Fin-imp-card*)
apply (*drule sym*, *simp*)
apply (*frule card-gr0-imp-finite*)
apply (*rule icard-finite*, *assumption*)
done

lemma *icard-empty*[*simp*]: $icard\ \{\} = 0$
by (*simp add*: *icard-finite*[*OF finite.emptyI*])
lemma *icard-empty-iff*: $(icard\ A = 0) = (A = \{\})$
apply (*unfold zero-inat-def*)
apply (*rule iffI*)
apply (*frule icard-eq-Fin-imp*)
apply (*simp add*: *icard-finite*)
apply *simp*
done
lemmas *icard-empty-iff-Fin* = *icard-empty-iff*[*unfolded zero-inat-def*]

lemma *icard-not-empty-iff*: $(0 < icard\ A) = (A \neq \{\})$
by (*simp add*: *icard-empty-iff*[*symmetric*])
lemmas *icard-not-empty-iff-Fin* = *icard-not-empty-iff*[*unfolded zero-inat-def*]

lemma *icard-singleton*: $icard\ \{a\} = iSuc\ 0$
by (*simp add*: *icard-finite iSuc-Fin*)
lemmas *icard-singleton-Fin*[*simp*] = *icard-singleton*[*unfolded zero-inat-def*]
lemma *icard-1-imp-singleton*: $icard\ A = iSuc\ 0 \Longrightarrow \exists\, a.\ A = \{a\}$
apply (*simp add*: *iSuc-Fin*)
apply (*frule icard-eq-Fin-imp*)
apply (*simp add*: *icard-finite card-1-imp-singleton*)
done
lemma *icard-1-singleton-conv*: $(icard\ A = iSuc\ 0) = (\exists\, a.\ A = \{a\})$
apply (*rule iffI*)
apply (*simp add*: *icard-1-imp-singleton*)
apply *fastsimp*
done

thm *Finite-Set.card-insert-disjoint*
lemma *icard-insert-disjoint*: $x \notin A \Longrightarrow icard\ (insert\ x\ A) = iSuc\ (icard\ A)$
apply (*case-tac finite A*)
apply (*simp add*: *icard-finite iSuc-Fin card-insert-disjoint*)
apply (*simp add*: *infinite-insert*)
done
thm *Finite-Set.card-insert-if*
lemma *icard-insert-if*: $icard\ (insert\ x\ A) = (if\ x \in A\ then\ icard\ A\ else\ iSuc\ (icard\ A))$
apply (*case-tac* $x \in A$)

apply (*simp add*: *insert-absorb*)
apply (*simp add*: *icard-insert-disjoint*)
done
thm *Finite-Set.card-0-eq*
lemmas *icard-0-eq* = *icard-empty-iff*

thm *Finite-Set.card-Suc-Diff1*
lemma *icard-Suc-Diff1*: $x \in A \Longrightarrow iSuc\ (icard\ (A - \{x\})) = icard\ A$
apply (*case-tac finite A*)
apply (*simp add*: *icard-finite iSuc-Fin in-imp-not-empty not-empty-card-gr0-conv*[*THEN iffD1*])
apply (*simp add*: *Diff-infinite-finite*[*OF singleton-finite*])
done

thm *Finite-Set.card-Diff-singleton*
lemma *icard-Diff-singleton*: $x \in A \Longrightarrow icard\ (A - \{x\}) = icard\ A - 1$
apply (*rule iSuc-inject*[*THEN iffD1*])
apply (*frule in-imp-not-empty*, *drule icard-not-empty-iff*[*THEN iffD2*])
apply (*simp add*: *icard-Suc-Diff1 iSuc-pred-Fin one-iSuc*)
done

thm *Finite-Set.card-Diff-singleton-if*
lemma *icard-Diff-singleton-if*: $icard\ (A - \{x\}) = (if\ x \in A\ then\ icard\ A - 1\ else\ icard\ A)$
by (*simp add*: *icard-Diff-singleton*)

thm *Finite-Set.card-insert*
lemma *icard-insert*: $icard\ (insert\ x\ A) = iSuc\ (icard\ (A - \{x\}))$
by (*metis icard-Diff-singleton-if icard-Suc-Diff1 icard-insert-disjoint insert-absorb*)

thm *Finite-Set.card-insert-le*
lemma *icard-insert-le*: $icard\ A \leq icard\ (insert\ x\ A)$
by (*simp add*: *icard-insert-if*)

thm *Finite-Set.card-mono*
lemma *icard-mono*: $A \subseteq B \Longrightarrow icard\ A \leq icard\ B$
apply (*case-tac finite B*)
apply (*frule subset-finite-imp-finite*, *simp*)
apply (*simp add*: *icard-finite card-mono*)
apply *simp*
done

thm *Finite-Set.card-seteq*
lemma *not-icard-seteq*: $\exists (A::nat\ set)\ B.\ (A \subseteq B \wedge icard\ B \leq icard\ A \wedge \neg\ A = B)$
apply (*rule-tac x*=$\{1..\}$ **in** *exI*)
apply (*rule-tac x*=$\{0..\}$ **in** *exI*)
apply (*fastsimp simp add*: *infinite-atLeast*)
done

thm *Finite-Set.psubset-card-mono*
lemma *not-psubset-icard-mono*: $\exists (A::nat\ set)\ B.\ A \subset B \wedge \neg\ icard\ A < icard\ B$
apply (*rule-tac* $x=\{1..\}$ **in** *exI*)
apply (*rule-tac* $x=\{0..\}$ **in** *exI*)
apply (*fastsimp simp add*: *infinite-atLeast*)
done

thm *Finite-Set.card-Un-Int*
lemma *icard-Un-Int*: $icard\ A + icard\ B = icard\ (A \cup B) + icard\ (A \cap B)$
apply (*case-tac finite A*, *case-tac finite B*)
thm *card-Un-Int*
apply (*simp add*: *icard-finite card-Un-Int*[*of A*])
apply *simp-all*
done

thm *Finite-Set.card-Un-disjoint*
lemma *icard-Un-disjoint*: $A \cap B = \{\} \Longrightarrow icard\ (A \cup B) = icard\ A + icard\ B$
by (*simp add*: *icard-Un-Int*[*of A*])

thm *Finite-Set.card-Diff-subset*
lemma *not-icard-Diff-subset*: $\exists (A::nat\ set)\ B.\ B \subseteq A \wedge \neg\ icard\ (A - B) = icard\ A - icard\ B$
apply (*rule-tac* $x=\{0..\}$ **in** *exI*)
apply (*rule-tac* $x=\{1..\}$ **in** *exI*)
apply (*simp add*: *set-diff-eq linorder-not-le icard-UNIV-nat iSuc-Fin*)
done

thm
Finite-Set.card-Diff1-less
Finite-Set.card-Diff2-less
lemma *not-icard-Diff1-less*: $\exists (A::nat\ set)x.\ x \in A \wedge \neg\ icard\ (A - \{x\}) < icard\ A$
by (*rule-tac* $x=\{0..\}$ **in** *exI*, *simp*)
lemma *not-icard-Diff2-less*: $\exists (A::nat\ set)x\ y.\ x \in A \wedge y \in A \wedge \neg\ icard\ (A - \{x\} - \{y\}) < icard\ A$
by (*rule-tac* $x=\{0..\}$ **in** *exI*, *simp*)

thm *Finite-Set.card-Diff1-le*
lemma *icard-Diff1-le*: $icard\ (A - \{x\}) \leq icard\ A$
by (*rule icard-mono*, *rule Diff-subset*)

thm *Finite-Set.card-psubset*
lemma *icard-psubset*: $[\![\ A \subseteq B;\ icard\ A < icard\ B\ ]\!] \Longrightarrow A \subset B$
by (*metis iless-ile psubset-eq*)

thm *SetInterval2.card-partition*
lemma *icard-partition*:

$[\![\bigwedge c.\ c \in C \Longrightarrow icard\ c = k;\ \bigwedge c1\ c2.\ [\![c1 \in C;\ c2 \in C;\ c1 \neq c2]\!] \Longrightarrow c1 \cap c2 = \{\}\ ]\!] \Longrightarrow$
$icard\ (\bigcup C) = k * icard\ C$
apply (*case-tac* $C = \{\}$, *simp*)
apply (*case-tac* $k = 0$)
apply (*simp add*: *icard-empty-iff-Fin*)
apply *simp*
apply (*case-tac* k, *rename-tac k1*)
apply (*subgoal-tac* $0 < k1$)
prefer *2*
apply *simp*
apply (*case-tac finite* C)
apply (*simp add*: *icard-finite*)
thm *SetInterval2.card-partition*
thm *icard-eq-Fin-imp-card*
apply (*subgoal-tac* $\bigwedge c.\ c \in C \Longrightarrow card\ c = k1$)
prefer *2*
apply (*rule icard-eq-Fin-imp-card*, *simp*)
thm *SetInterval2.card-partition*
apply (*frule-tac* $C{=}C$ **and** $k{=}k1$ **in** *SetInterval2.card-partition*, *simp*+)
apply (*subgoal-tac finite* $(\bigcup C)$)
prefer *2*
apply (*rule card-gr0-imp-finite*)
apply (*simp add*: *not-empty-card-gr0-conv*)
apply (*simp add*: *icard-finite*)
apply *simp*
apply (*rule icard-infinite*)
thm *finite-UnionD*
apply (*rule ccontr*, *simp*)
apply (*drule finite-UnionD*, *simp*)
apply (*frule icard-not-empty-iff*[*THEN iffD2*])
apply (*simp add*: *icard-infinite-conv*)
apply (*frule not-empty-imp-ex*, *erule exE*, *rename-tac c*)
thm *Union-upper*
apply (*frule Union-upper*)
thm *infinite-super*
apply (*rule infinite-super*, *assumption*)
apply *simp*
done

thm *Big-Operators.setsum-constant*

thm *Big-Operators.setprod-constant*

thm *Big-Operators.setsum-bounded*

thm *Big-Operators.card-UN-disjoint*

thm *Big-Operators.card-Union-disjoint*

thm *Finite-Set.card-image-le*
lemma *icard-image-le*: *icard* (*f* ‘ *A*) ≤ *icard A*
apply (*case-tac finite A*)
apply (*simp add*: *icard-finite card-image-le*)
apply *simp*
done

thm *Finite-Set.card-image*
lemma *icard-image*: *inj-on f A* ⟹ *icard* (*f* ‘ *A*) = *icard A*
apply (*case-tac finite A*)
apply (*simp add*: *icard-finite card-image*)
apply (*simp add*: *icard-infinite-conv inj-on-imp-infinite-image*)
done

thm *Finite-Set.eq-card-imp-inj-on*
lemma *not-eq-icard-imp-inj-on*: ∃(*f*::*nat*⇒*nat*) (*A*::*nat set*). *icard* (*f* ‘ *A*) = *icard A* ∧ ¬ *inj-on f A*
apply (*rule-tac x*=λ*n*. (*if n* = *0 then Suc 0 else n*) **in** *exI*)
apply (*rule-tac x*={*0*..} **in** *exI*)
apply (*rule conjI*)
apply (*rule subst*[*of* {*1*..} ((λ*n*. *if n* = *0 then Suc 0 else n*) ‘ {*0*..})])
apply (*simp add*: *set-eq-iff*)
apply (*rule allI*, *rename-tac n*)
apply (*case-tac n* = *0*, *simp*)
apply *simp*
apply (*simp only*: *icard-infinite*[*OF infinite-atLeast*])
apply (*simp add*: *inj-on-def*)
apply *blast*
done

thm *Finite-Set.inj-on-iff-eq-card*
lemma *not-inj-on-iff-eq-icard*: ∃(*f*::*nat*⇒*nat*) (*A*::*nat set*). ¬ (*inj-on f A* = (*icard* (*f* ‘ *A*) = *icard A*))
by (*insert not-eq-icard-imp-inj-on*, *blast*)

thm *Finite-Set.card-inj-on-le*
lemma *icard-inj-on-le*: ⟦ *inj-on f A*; *f* ‘ *A* ⊆ *B* ⟧ ⟹ *icard A* ≤ *icard B*
apply (*case-tac finite B*)
apply (*metis icard-image icard-mono*)
apply *simp*
done

thm *Finite-Set.card-bij-eq*
thm *Finite-Set.card-bij-eq*[*no-vars*]
lemma *icard-bij-eq*:
⟦ *inj-on f A*; *f* ‘ *A* ⊆ *B*; *inj-on g B*; *g* ‘ *B* ⊆ *A* ⟧ ⟹
icard A = *icard B*

by (*simp add*: *order-eq-iff icard-inj-on-le*)

thm *Big-Operators.card-SigmaI*

thm *Big-Operators.card-cartesian-product*
lemma *icard-cartesian-product*: *icard* $(A \times B) = icard\ A * icard\ B$
apply (*case-tac* $A = \{\} \vee B = \{\}$, *fastsimp*)
apply *clarsimp*
apply (*case-tac finite* $A \wedge$ *finite* B)
apply (*simp add*: *icard-finite*)
apply (*simp only*: *de-Morgan-conj*, *erule disjE*)
apply (*simp-all add*:
icard-not-empty-iff[*symmetric*]
cartesian-product-infiniteL-imp-infinite cartesian-product-infiniteR-imp-infinite)
done

thm *card-cartesian-product-singleton*
lemma *icard-cartesian-product-singleton*: *icard* $(\{x\} \times A) = icard\ A$
by (*simp add*: *icard-cartesian-product mult-iSuc*)

thm *card-cartesian-product-singleton-right*
lemma *icard-cartesian-product-singleton-right*: *icard* $(A \times \{x\}) = icard\ A$
by (*simp add*: *icard-cartesian-product mult-iSuc-right*)

thm *Finite-Set.card-Pow*

thm *Finite-Set.dvd-partition*

thm *Equiv-Relations.equiv-imp-dvd-card*

thm *Equiv-Relations.card-quotient-disjoint*

thm
SetInterval.card-lessThan
SetInterval.card-atMost
SetInterval.card-atLeastLessThan
SetInterval.card-atLeastAtMost
SetInterval.card-greaterThanAtMost
SetInterval.card-greaterThanLessThan
lemma
icard-lessThan: *icard* $\{..<u\} = Fin\ u$ **and**
icard-atMost: *icard* $\{..u\} = Fin\ (Suc\ u)$ **and**
icard-atLeastLessThan: *icard* $\{l..<u\} = Fin\ (u - l)$ **and**
icard-atLeastAtMost: *icard* $\{l..u\} = Fin\ (Suc\ u - l)$ **and**
icard-greaterThanAtMost: *icard* $\{l<..u\} = Fin\ (u - l)$ **and**

icard-greaterThanLessThan: *icard* $\{l<..<u\} = Fin\ (u - Suc\ l)$
by (*simp-all add*: *icard-finite*)

lemma *icard-atLeast*: *icard* $\{(u::nat)..\} = \infty$
by (*simp add*: *infinite-atLeast*)
lemma *icard-greaterThan*: *icard* $\{(u::nat)<..\} = \infty$
by (*simp add*: *infinite-greaterThan*)

thm
SetInterval.card-atLeastZeroLessThan-int
SetInterval.card-atLeastLessThan-int
SetInterval.card-atLeastAtMost-int
SetInterval.card-greaterThanAtMost-int
lemma
icard-atLeastZeroLessThan-int: *icard* $\{0..<u\} = Fin\ (nat\ u)$ **and**
icard-atLeastLessThan-int: *icard* $\{l..<u\} = Fin\ (nat\ (u - l))$ **and**
icard-atLeastAtMost-int: *icard* $\{l..u\} = Fin\ (nat\ (u - l + 1))$ **and**
icard-greaterThanAtMost-int: *icard* $\{l<..u\} = Fin\ (nat\ (u - l))$
by (*simp-all add*: *icard-finite*)

lemma *icard-atLeast-int*: *icard* $\{(u::int)..\} = \infty$
by (*simp add*: *infinite-atLeast-int*)

lemma *icard-greaterThan-int*: *icard* $\{(u::int)<..\} = \infty$
by (*simp add*: *infinite-greaterThan-int*)

lemma *icard-atMost-int*: *icard* $\{..(u::int)\} = \infty$
by (*simp add*: *infinite-atMost-int*)

lemma *icard-lessThan-int*: *icard* $\{..<(u::int)\} = \infty$
by (*simp add*: *infinite-lessThan-int*)

end

11 ListInf: Additional definitions and results for lists

theory *ListInf*
imports *List2* *../CommonSet/InfiniteSet2*
begin

11.1 Infinite lists

We define infinite lists as functions over natural numbers, i. e., we use functions $nat \Rightarrow 'a$ as infinite lists over elements of $'a$. Mapping functions to intervals lists $[m..<n]$ yiels common finite lists.

11.1.1 Appending a functions to a list

types *'a ilist = nat ⇒ 'a*

definition
i-append :: 'a list ⇒ 'a ilist ⇒ 'a ilist (**infixr** *⌢ 65*)
where
xs ⌢ f ≡ λn. if n < length xs then xs ! n else f (n − length xs)

syntax (*HTML* **output**)
i-append :: 'a list ⇒ 'a ilist ⇒ 'a ilist (**infixr** *⌢ 65*)

Synonym for the lemma *Fun.fun-eq-iff* from the HOL library to unify lemma names for finite and infinite lists, providing *list-eq-iff* for finite and *ilist-eq-iff* for infinite lists.

lemmas *expand-ilist-eq = Fun.fun-eq-iff*
lemmas *ilist-eq-iff = expand-ilist-eq*

lemma *i-append-nth*: *(xs ⌢ f) n = (if n < length xs then xs ! n else f (n − length xs))*
by (*simp add*: *i-append-def*)
lemma *i-append-nth1*[*simp*]: *n < length xs ⟹ (xs ⌢ f) n = xs ! n*
by (*simp add*: *i-append-def*)
lemma *i-append-nth2*[*simp*]: *length xs ≤ n ⟹ (xs ⌢ f) n = f (n − length xs)*
by (*simp add*: *i-append-def*)
lemma *i-append-Nil*[*simp*]: *[] ⌢ f = f*
by (*simp add*: *i-append-def*)

lemma *i-append-assoc*[*simp*]: *xs ⌢ (ys ⌢ f) = (xs @ ys) ⌢ f*
apply (*case-tac ys = []*, *simp*)
apply (*fastsimp simp*: *expand-ilist-eq i-append-def nth-append*)
done

thm *append-Cons*
lemma *i-append-Cons*: *(x # xs) ⌢ f = [x] ⌢ (xs ⌢ f)*
by *simp*

thm *List.append-eq-append-conv*
lemma *i-append-eq-i-append-conv*[*simp*]:
length xs = length ys ⟹
(xs ⌢ f = ys ⌢ g) = (xs = ys ∧ f = g)
apply (*rule iffI*)
prefer *2*
apply *simp*
apply (*simp add*: *expand-ilist-eq expand-list-eq i-append-nth*)
apply (*intro conjI impI allI*)
apply (*rename-tac x*)
apply (*drule-tac x=x* **in** *spec*)
apply *simp*

apply (*rename-tac x*)
apply (*drule-tac x=x + length ys* **in** *spec*)
apply *simp*
done

thm *List.append-eq-append-conv2*
lemma *i-append-eq-i-append-conv2-aux*:
⟦ *xs* ⌢ *f* = *ys* ⌢ *g*; *length xs* ≤ *length ys* ⟧ ⟹
∃ *zs*. *xs* @ *zs* = *ys* ∧ *f* = *zs* ⌢ *g*
apply (*simp add*: *expand-ilist-eq expand-list-eq nth-append*)
apply (*rule-tac x=drop* (*length xs*) *ys* **in** *exI*)
apply *simp*
apply (*rule conjI*)
apply (*clarify*, *rename-tac i*)
apply (*drule-tac x=i* **in** *spec*)
apply *simp*
apply (*clarify*, *rename-tac i*)
apply (*drule-tac x=length xs + i* **in** *spec*)
apply (*simp add*: *i-append-nth*)
apply (*case-tac length xs + i < length ys*)
apply *fastsimp*
apply (*fastsimp simp*: *add-commute*[*of - length xs*])
done

lemma *i-append-eq-i-append-conv2*:
(*xs* ⌢ *f* = *ys* ⌢ *g*) =
(∃ *zs*. *xs* = *ys* @ *zs* ∧ *zs* ⌢ *f* = *g* ∨ *xs* @ *zs* = *ys* ∧ *f* = *zs* ⌢ *g*)
apply (*rule iffI*)
apply (*case-tac length xs ≤ length ys*)
apply (*frule i-append-eq-i-append-conv2-aux*, *assumption*)
apply *blast*
apply (*simp add*: *linorder-not-le eq-commute*[*of xs* ⌢ *f*], *drule less-imp-le*)
apply (*frule i-append-eq-i-append-conv2-aux*, *assumption*)
apply *blast*
apply *fastsimp*
done

thm *List.same-append-eq*
lemma *same-i-append-eq*[*iff*]: (*xs* ⌢ *f* = *xs* ⌢ *g*) = (*f* = *g*)
apply (*rule iffI*)
apply (*clarsimp simp*: *expand-ilist-eq*, *rename-tac i*)
apply (*erule-tac x=length xs + i* **in** *allE*)
apply *simp*
apply *simp*
done

thm *List.append-same-eq*
lemma *NOT-i-append-same-eq*:

$\neg(\forall\, xs\ ys\ f.\ (xs \frown (f::(nat \Rightarrow nat))) = ys \frown f) = (xs = ys))$
apply *simp*
apply (*rule-tac x*=[] **in** *exI*)
apply (*rule-tac x*=[*0*] **in** *exI*)
apply (*rule-tac x*=$\lambda n.\ 0$ **in** *exI*)
apply (*simp add*: *expand-ilist-eq i-append-nth*)
done

thm *List.hd-append*
lemma *i-append-hd*: $(xs \frown f)\ 0 = (if\ xs = []\ then\ f\ 0\ else\ hd\ xs)$
by (*simp add*: *hd-eq-first*)

thm *List.hd-append2*
lemma *i-append-hd2*[*simp*]: $xs \neq [] \Longrightarrow (xs \frown f)\ 0 = hd\ xs$
by (*simp add*: *i-append-hd*)

thm *List.eq-Nil-appendI*
lemma *eq-Nil-i-appendI*: $f = g \Longrightarrow f = [] \frown g$
by *simp*

thm *List.append-eq-appendI*
lemma *i-append-eq-i-appendI*:
$[\![\ xs\ @\ xs' = ys;\ f = xs' \frown g\ ]\!] \Longrightarrow xs \frown f = ys \frown g$
by *simp*

thm *List.map-ext*
lemma *o-ext*:
$(\forall\, x.\ (x \in range\ h \longrightarrow f\ x = g\ x)) \Longrightarrow f \circ h = g \circ h$
by (*simp add*: *expand-ilist-eq*)
thm
o-ext
o-ext[*rule-format*]
thm
List.map-ident
Fun.id-o

thm *List.map-append*
lemma *i-append-o*[*simp*]: $g \circ (xs \frown f) = (map\ g\ xs) \frown (g \circ f)$
by (*simp add*: *expand-ilist-eq i-append-nth*)

thm
List.map-map[*of f g h*]
Fun.o-assoc[*of f g h*, *symmetric*]
thm *List.map-eq-conv*
lemma *o-eq-conv*: $(f \circ h = g \circ h) = (\forall\, x{\in}range\ h.\ f\ x = g\ x)$
by (*simp add*: *expand-ilist-eq*)

thm *List.map-cong*
lemma *o-cong*:
⟦ $h = i$; $\bigwedge x.\ x \in range\ i \Longrightarrow f\ x = g\ x$ ⟧ $\Longrightarrow f \circ h = f \circ i$
by *blast*

thm *List.ex-map-conv*
lemma *ex-o-conv*: $(\exists h.\ g = f \circ h) = (\forall y{\in}range\ g.\ \exists x.\ y = f\ x)$
apply (*rule iffI*)
apply *fastsimp*
apply (*simp add*: *expand-ilist-eq*)
apply (*rule-tac x*=$\lambda x.\ (SOME\ y.\ g\ x = f\ y)$ **in** *exI*)
thm *someI-ex*
apply (*fastsimp intro*: *someI-ex*)
done

thm *List.map-inj-on*
lemma *o-inj-on*:
⟦ $f \circ g = f \circ h$; *inj-on f* (*range g* ∪ *range h*) ⟧ $\Longrightarrow g = h$
apply (*rule expand-ilist-eq*[*THEN iffD2*], *clarify*, *rename-tac x*)
apply (*drule-tac x*=*x* **in** *fun-cong*)
apply (*rule inj-onD*)
apply *simp*+
done

thm *List.inj-on-map-eq-map*
lemma *inj-on-o-eq-o*:
inj-on f (*range g* ∪ *range h*) $\Longrightarrow$
$(f \circ g = f \circ h) = (g = h)$
apply (*rule iffI*)
apply (*rule o-inj-on*, *assumption*+)
apply *simp*
done

thm *List.map-injective*
lemma *o-injective*: ⟦ $f \circ g = f \circ h$; *inj f* ⟧ $\Longrightarrow g = h$
by (*simp add*: *expand-ilist-eq inj-on-def*)

thm *List.inj-map-eq-map*
lemma *inj-o-eq-o*: *inj f* $\Longrightarrow (f \circ g = f \circ h) = (g = h)$
apply (*rule iffI*)
apply (*rule o-injective*, *assumption*+)
apply *simp*
done

thm *List.inj-mapI*
lemma *inj-oI*: *inj f* $\Longrightarrow$ *inj* $(\lambda g.\ f \circ g)$
apply (*simp add*: *inj-on-def*)
thm *o-inj-on*[*unfolded inj-on-def*]

apply (*blast intro*: *o-inj-on*[*unfolded inj-on-def*])
done

thm *List.inj-mapD*
lemma *inj-oD*: *inj* (λ*g*. *f* ∘ *g*) ⟹ *inj f*
apply (*clarsimp simp add*: *inj-on-def*, *rename-tac g h*)
apply (*erule-tac x*=λ*n*. *g* **in** *allE*)
apply (*erule-tac x*=λ*n*. *h* **in** *allE*)
apply (*simp add*: *expand-ilist-eq*)
done

thm *List.inj-map*
lemma *inj-o*[*iff*]: *inj* (λ*g*. *f* ∘ *g*) = *inj f*
apply (*rule iffI*)
apply (*rule inj-oD*, *assumption*)
apply (*rule inj-oI*, *assumption*)
done

thm *List.inj-on-mapI*
lemma *inj-on-oI*:
inj-on f (⋃ (λ*f*. *range f*) ` *A*) ⟹ *inj-on* (λ*g*. *f* ∘ *g*) *A*
apply (*rule inj-onI*)
apply (*rule o-inj-on*, *assumption*)
apply (*unfold inj-on-def*)
apply *force*
done

thm *List.map-idI*
lemma *o-idI*: ∀ *x*. *x* ∈ *range g* ⟶ *f x* = *x* ⟹ *f* ∘ *g* = *g*
by (*simp add*: *expand-ilist-eq*)
thm
o-idI
o-idI[*rule-format*]

thm *List.map-fun-upd*
lemma *o-fun-upd*[*simp*]: *y* ∉ *range g* ⟹ *f* (*y* := *x*) ∘ *g* = *f* ∘ *g*
by (*fastsimp simp*: *expand-ilist-eq*)

thm *List.set-append*
lemma *range-i-append*[*simp*]: *range* (*xs* ⌢ *f*) = *set xs* ∪ *range f*
by (*fastsimp simp*: *in-set-conv-nth i-append-nth*)

thm *List.set-subset-Cons*
lemma *set-subset-i-append*: *set xs* ⊆ *range* (*xs* ⌢ *f*)
by *simp*
lemma *range-subset-i-append*: *range f* ⊆ *range* (*xs* ⌢ *f*)
by *simp*

thm *List.set-ConsD*
lemma *range-ConsD*: $y \in range\ ([x] @@ f) \Longrightarrow y = x \vee y \in range\ f$
by *simp*

thm *List.set-map*
lemma *range-o*[*simp*]: $range\ (f \circ g) = f\ `\ range\ g$
by (*rule image-compose*)

thm *List.in-set-conv-decomp*
lemma *in-range-conv-decomp*:
$(x \in range\ f) = (\exists xs\ g.\ f = xs @@ ([x] @@ g))$
apply (*simp add*: *image-iff*)
apply (*rule iffI*)
apply (*clarify*, *rename-tac n*)
apply (*rule-tac x*=*map f* [*0..<n*] **in** *exI*)
apply (*rule-tac x*=$\lambda i.\ f\ (i + Suc\ n)$ **in** *exI*)
apply (*simp add*: *expand-ilist-eq i-append-nth nth-append linorder-not-less less-Suc-eq-le*)
apply (*clarify*, *rename-tac xs g*)
apply (*rule-tac x*=*length xs* **in** *exI*)
apply *simp*
done

nth

thm *List.nth-Cons-0*
lemma *i-append-nth-Cons-0*[*simp*]: $((x \# xs) @@ f)\ 0 = x$
by *simp*

thm *List.nth-Cons-Suc*
lemma *i-append-nth-Cons-Suc*[*simp*]:
$((x \# xs) @@ f)\ (Suc\ n) = (xs @@ f)\ n$
by (*simp add*: *i-append-nth*)

lemma *i-append-nth-Cons*:
$([x] @@ f)\ n = (case\ n\ of\ 0 \Rightarrow x \mid Suc\ k \Rightarrow f\ k)$
by (*case-tac n*, *simp-all add*: *i-append-nth*)

lemma *i-append-nth-Cons'*:
$([x] @@ f)\ n = (if\ n = 0\ then\ x\ else\ f\ (n - Suc\ 0))$
by (*case-tac n*, *simp-all add*: *i-append-nth*)

thm
List.nth-append
thm
i-append-def
i-append-nth1
i-append-nth2

thm *List.nth-append-length*

lemma *i-append-nth-length*[*simp*]: $(xs \frown f)\ (length\ xs) = f\ 0$
by *simp*

thm *List.nth-append-length-plus*
lemma *i-append-nth-length-plus*[*simp*]: $(xs \frown f)\ (length\ xs + n) = f\ n$
by *simp*

thm
List.nth-map
Fun.o-apply

thm
List.set-conv-nth
Set.full-SetCompr-eq

thm *List.in-set-conv-nth*
lemma *range-iff*: $(y \in range\ f) = (\exists x.\ y = f\ x)$
by *blast*

thm *List.list-ball-nth*
lemma *range-ball-nth*: $\forall y{\in}range\ f.\ P\ y \Longrightarrow P\ (f\ x)$
by *blast*

thm
List.nth-mem
Set.rangeI

thm *List.all-nth-imp-all-set*
lemma *all-nth-imp-all-range*: $[\![\ \forall x.\ P\ (f\ x); y \in range\ f\ ]\!] \Longrightarrow P\ y$
by *blast*

thm *List.all-set-conv-all-nth*
lemma *all-range-conv-all-nth*: $(\forall y{\in}range\ f.\ P\ y) = (\forall x.\ P\ (f\ x))$
by *blast*

thm
List.nth-list-update
Fun.fun-upd-def
thm
List.nth-list-update-eq
Fun.fun-upd-same
thm
List.nth-list-update-neq
Fun.fun-upd-other
thm

List.list-update-overwrite
Fun.fun-upd-upd
thm
List.list-update-id
Fun.fun-upd-triv
thm
List.list-update-same-conv
Fun.fun-upd-idem-iff

thm *List.list-update-append1*
lemma *i-append-update1*:
$n < length\ xs \Longrightarrow (xs \frown f)\ (n := x) = xs[n := x] \frown f$
by (*simp add*: *expand-ilist-eq i-append-nth*)
lemma *i-append-update2*:
$length\ xs \leq n \Longrightarrow (xs \frown f)\ (n := x) = xs \frown (f(n - length\ xs := x))$
by (*fastsimp simp*: *expand-ilist-eq i-append-nth*)

thm *List.list-update-append*
lemma *i-append-update*:
$(xs \frown f)\ (n := x) =$
$(if\ n < length\ xs\ then\ xs[n := x] \frown f$
$\ else\ xs \frown (f(n - length\ xs := x)))$
by (*simp add*: *i-append-update1 i-append-update2*)

thm *List.list-update-length*
lemma *i-append-update-length*[*simp*]:
$(xs \frown f)\ (length\ xs := y) = xs \frown (f(0 := y))$
by (*simp add*: *i-append-update2*)

thm *List.set-update-subset-insert*
lemma *range-update-subset-insert*:
$range\ (f(n := x)) \subseteq insert\ x\ (range\ f)$
by *fastsimp*

thm *List.set-update-subsetI*
lemma *range-update-subsetI*:
$[\![\ range\ f \subseteq A;\ x \in A\ ]\!] \Longrightarrow range\ (f(n := x)) \subseteq A$
by *fastsimp*

thm *List.set-update-memI*
lemma *range-update-memI*: $x \in range\ (f(n := x))$
by *fastsimp*

11.1.2 *take* and *drop* for infinite lists

The *i-take* operator takes the first n elements of an infinite list, i.e. *i-take f* $n = [f\ 0, f\ 1, \ldots, f\ (n-1)]$. The *i-drop* operator drops the first n elements of an infinite list, i.e. $(i\text{-}take\ f\ n)\ 0 = f\ n$, $(i\text{-}take\ f\ n)\ 1 = f\ (n + 1)$, ...

definition

i-take :: *nat* ⇒ *'a ilist* ⇒ *'a list*
where
i-take n f ≡ *map f* [*0..<n*]
definition
i-drop :: *nat* ⇒ *'a ilist* ⇒ *'a ilist*
where
i-drop n f ≡ (λ*x*. *f* (*n* + *x*))

abbreviation (*xsymbols*)
i-take' :: *'a ilist* ⇒ *nat* ⇒ *'a list* (**infixl** ⇓ *100*)
where
f ⇓ *n* ≡ *i-take n f*
abbreviation (*xsymbols*)
i-drop' :: *'a ilist* ⇒ *nat* ⇒ *'a ilist* (**infixl** ⇑ *100*)
where
f ⇑ *n* ≡ *i-drop n f*

syntax (*HTML* **output**)
i-take' :: *'a ilist* ⇒ *nat* ⇒ *'a list* (**infixl** ⇓ *100*)
i-drop' :: *'a ilist* ⇒ *nat* ⇒ *'a ilist* (**infixl** ⇑ *100*)

term *f* ⇓ *n*
term *f* ⇑ *n*

lemma *f* ⇓ *n* = *map f* [*0..<n*]
by (*simp add*: *i-take-def*)
lemma *f* ⇑ *n* = (λ*x*. *f* (*n* + *x*))
by (*simp add*: *i-drop-def*)

Basic results for *i-take* and *i-drop*

thm *take-first*
lemma *i-take-first*: *f* ⇓ *Suc 0* = [*f 0*]
by (*simp add*: *i-take-def*)

thm *drop-take-1*
lemma *i-drop-i-take-1*: *f* ⇑ *n* ⇓ *Suc 0* = [*f n*]
by (*simp add*: *i-drop-def i-take-def*)

thm *List.take-take*
lemma *i-take-take-eq1*: *m* ≤ *n* ⟹ (*f* ⇓ *n*) ↓ *m* = *f* ⇓ *m*
by (*simp add*: *i-take-def take-map*)
lemma *i-take-take-eq2*: *n* ≤ *m* ⟹ (*f* ⇓ *n*) ↓ *m* = *f* ⇓ *n*
by (*simp add*: *i-take-def take-map*)
lemma *i-take-take*[*simp*]: (*f* ⇓ *n*) ↓ *m* = *f* ⇓ *min n m*
by (*simp add*: *min-def i-take-take-eq1 i-take-take-eq2*)

thm *List.nth-drop*
lemma *i-drop-nth*[*simp*]: (*s* ⇑ *n*) *x* = *s* (*n* + *x*)

by (*simp add*: *i-drop-def*)
lemma *i-drop-nth-sub*: $n \leq x \Longrightarrow (s \Uparrow n)\ (x - n) = s\ x$
by (*simp add*: *i-drop-def*)
thm *List.nth-take*
theorem *i-take-nth*[*simp*]: $i < n \Longrightarrow (f \Downarrow n)\ !\ i = f\ i$
by (*simp add*: *i-take-def*)

thm *List.length-take*
lemma *i-take-length*[*simp*]: *length* $(f \Downarrow n) = n$
by (*simp add*: *i-take-def*)

thm *List.take-0*
lemma *i-take-0*[*simp*]: $f \Downarrow 0 = []$
by (*simp add*: *i-take-def*)
thm *List.drop-0*
lemma *i-drop-0*[*simp*]: $f \Uparrow 0 = f$
by (*simp add*: *i-drop-def*)

lemma *i-take-eq-Nil*[*simp*]: $(f \Downarrow n = []) = (n = 0)$
by (*simp add*: *length-0-conv*[*symmetric*] *del*: *length-0-conv*)
lemma *i-take-not-empty-conv*: $(f \Downarrow n \neq []) = (0 < n)$
by *simp*

lemma *last-i-take*: *last* $(f \Downarrow Suc\ n) = f\ n$
by (*simp add*: *last-nth*)

lemma *last-i-take2*: $0 < n \Longrightarrow$ *last* $(f \Downarrow n) = f\ (n - Suc\ 0)$
by (*simp add*: *last-i-take*[*of* - *f*, *symmetric*])

lemma *nth-0-i-drop*: $(f \Uparrow n)\ 0 = f\ n$
by *simp*

thm *take-replicate drop-replicate*
lemma *i-take-const*[*simp*]: $(\lambda n.\ x) \Downarrow i =$ *replicate* $i\ x$
by (*simp add*: *expand-list-eq*)
lemma *i-drop-const*[*simp*]: $(\lambda n.\ x) \Uparrow i = (\lambda n.\ x)$
by (*simp add*: *expand-ilist-eq*)

thm *List.take-append*
lemma *i-append-i-take-eq1*:
$n \leq$ *length* $xs \Longrightarrow (xs \frown f) \Downarrow n = xs \downarrow n$
by (*simp add*: *expand-list-eq*)

lemma *i-append-i-take-eq2*:
length $xs \leq n \Longrightarrow (xs \frown f) \Downarrow n = xs\ @\ (f \Downarrow (n - length\ xs))$
by (*simp add*: *expand-list-eq nth-append*)
lemma *i-append-i-take-if*:
$(xs \frown f) \Downarrow n =$ (*if* $n \leq$ *length* xs *then* $xs \downarrow n$ *else* $xs\ @\ (f \Downarrow (n - length\ xs))$)
by (*simp add*: *i-append-i-take-eq1 i-append-i-take-eq2*)

lemma *i-append-i-take*[*simp*]:
(*xs* ⌢ *f*) ⇓ *n* = (*xs* ↓ *n*) @ (*f* ⇓ (*n* − *length xs*))
by (*simp add*: *i-append-i-take-if*)

thm *List.drop-append*
lemma *i-append-i-drop-eq1*:
n ≤ *length xs* ⟹ (*xs* ⌢ *f*) ⇑ *n* = (*xs* ↑ *n*) ⌢ *f*
by (*simp add*: *expand-ilist-eq i-append-nth less-diff-conv add-commute*[*of* - *n*])
lemma *i-append-i-drop-eq2*:
length xs ≤ *n* ⟹ (*xs* ⌢ *f*) ⇑ *n* = *f* ⇑ (*n* − *length xs*)
by (*simp add*: *expand-ilist-eq i-append-nth*)
lemma *i-append-i-drop-if*:
(*xs* ⌢ *f*) ⇑ *n* = (*if n* < *length xs then* (*xs* ↑ *n*) ⌢ *f else f* ⇑ (*n* − *length xs*))
by (*simp add*: *i-append-i-drop-eq1 i-append-i-drop-eq2*)
lemma *i-append-i-drop*[*simp*]: (*xs* ⌢ *f*) ⇑ *n* = (*xs* ↑ *n*) ⌢ (*f* ⇑ (*n* − *length xs*))
by (*simp add*: *i-append-i-drop-if*)

thm
List.take-append
List.drop-append
i-append-i-take
i-append-i-drop

thm *List.append-take-drop-id*
lemma *i-append-i-take-i-drop-id*[*simp*]: (*f* ⇓ *n*) ⌢ (*f* ⇑ *n*) = *f*
by (*simp add*: *expand-ilist-eq i-append-nth*)

lemma *ilist-i-take-i-drop-imp-eq*:
⟦ *f* ⇓ *n* = *g* ⇓ *n*; *f* ⇑ *n* = *g* ⇑ *n* ⟧ ⟹ *f* = *g*
apply (*subst i-append-i-take-i-drop-id*[*of n f*, *symmetric*])
apply (*subst i-append-i-take-i-drop-id*[*of n g*, *symmetric*])
apply *simp*
done

lemma *ilist-i-take-i-drop-eq-conv*:
(*f* = *g*) = (∃ *n*. (*f* ⇓ *n* = *g* ⇓ *n* ∧ *f* ⇑ *n* = *g* ⇑ *n*))
apply (*rule iffI*, *simp*)
apply (*blast intro*: *ilist-i-take-i-drop-imp-eq*)
done

lemma *ilist-i-take-eq-conv*: (*f* = *g*) = (∀ *n*. *f* ⇓ *n* = *g* ⇓ *n*)
apply (*rule iffI*, *simp*)
apply (*clarsimp simp*: *expand-ilist-eq*, *rename-tac i*)
apply (*drule-tac x*=*Suc i* **in** *spec*)
apply (*drule-tac f*=λ*xs*. *xs* ! *i* **in** *arg-cong*)
apply *simp*
done

lemma *ilist-i-drop-eq-conv*: (*f* = *g*) = (∀ *n*. *f* ⇑ *n* = *g* ⇑ *n*)

apply (*rule iffI*, *simp*)
apply (*drule-tac x=0* **in** *spec*)
apply *simp*
done

lemma *i-take-the-conv*:
  $f \Downarrow k = (THE\ xs.\ length\ xs = k \wedge (\exists g.\ xs \frown g = f))$
thm *the1I2*
apply (*rule the1I2*)
 apply (*rule-tac a=f* $\Downarrow$ *k* **in** *ex1I*)
 apply (*fastsimp intro*: *i-append-i-take-i-drop-id*)+
done

lemma *i-drop-the-conv*:
  $f \Uparrow k = (THE\ g.\ (\exists xs.\ length\ xs = k \wedge xs \frown g = f))$
apply (*rule sym*, *rule the1-equality*)
 apply (*rule-tac a=f* $\Uparrow$ *k* **in** *ex1I*)
  apply (*rule-tac x=f* $\Downarrow$ *k* **in** *exI*, *simp*)
 apply *clarsimp*
apply (*rule-tac x=f* $\Downarrow$ *k* **in** *exI*, *simp*)
done

thm *List.take-Suc-Cons*
lemma *i-take-Suc-append*[*simp*]:
  $((x \# xs) \frown f) \Downarrow Suc\ n = x \# ((xs \frown f) \Downarrow n)$
by (*simp add*: *expand-list-eq*)
corollary *i-take-Suc-Cons*: $([x] \frown f) \Downarrow Suc\ n = x \# (f \Downarrow n)$
by *simp*

lemma *i-drop-Suc-append*[*simp*]: $((x \# xs) \frown f) \Uparrow Suc\ n = ((xs \frown f) \Uparrow n)$
by (*simp add*: *expand-list-eq*)
corollary *i-drop-Suc-Cons*: $([x] \frown f) \Uparrow Suc\ n = f \Uparrow n$
by *simp*

thm *List.take-Suc*
lemma *i-take-Suc*: $f \Downarrow Suc\ n = f\ 0 \# (f \Uparrow Suc\ 0 \Downarrow n)$
by (*simp add*: *expand-list-eq nth-Cons'*)
thm *List.take-Suc-conv-app-nth*
lemma *i-take-Suc-conv-app-nth*: $f \Downarrow Suc\ n = (f \Downarrow n) @ [f\ n]$
by (*simp add*: *i-take-def*)

thm *List.drop-drop*
lemma *i-drop-i-drop*[*simp*]: $s \Uparrow a \Uparrow b = s \Uparrow (a + b)$
by (*simp add*: *i-drop-def add-assoc*)
corollary *i-drop-Suc*: $f \Uparrow Suc\ 0 \Uparrow n = f \Uparrow Suc\ n$
by *simp*

lemma *i-take-commute*: $s \Downarrow a \downarrow b = s \Downarrow b \downarrow a$
by (*simp add*: *min-ac*)

lemma *i-drop-commute*: $s \Uparrow a \Uparrow b = s \Uparrow b \Uparrow a$
by (*simp add*: *add-commute*[*of a*])

thm *List.drop-tl*
corollary *i-drop-tl*: $f \Uparrow Suc\ 0 \Uparrow n = f \Uparrow n \Uparrow Suc\ 0$
by *simp*

thm *List.nth-via-drop*
lemma *nth-via-i-drop*: $(f \Uparrow n)\ 0 = x \Longrightarrow f\ n = x$
by *simp*

thm *List.drop-Suc-conv-tl*
lemma *i-drop-Suc-conv-tl*: $[f\ n] \frown (f \Uparrow Suc\ n) = f \Uparrow n$
by (*simp add*: *expand-ilist-eq i-append-nth*)

lemma *i-drop-Suc-conv-tl'*: $([f\ n] \frown f) \Uparrow Suc\ n = f \Uparrow n$
by (*simp add*: *i-drop-Suc-Cons*)
thm *i-drop-Suc-conv-tl i-drop-Suc-conv-tl'*

thm *List.take-drop*
lemma *i-take-i-drop*: $f \Uparrow m \Downarrow n = f \Downarrow (n + m) \uparrow m$
by (*simp add*: *expand-list-eq*)

Appending an interval of a function

lemma *i-take-int-append*:
$m \leq n \Longrightarrow (f \Downarrow m) @ map\ f\ [m..<n] = f \Downarrow n$
by (*simp add*: *expand-list-eq nth-append*)

lemma *i-take-drop-map-empty-iff*: $(f \Downarrow n \uparrow m = []) = (n \leq m)$
by *simp*

lemma *i-take-drop-map*: $f \Downarrow n \uparrow m = map\ f\ [m..<n]$
by (*simp add*: *expand-list-eq*)

corollary *i-take-drop-append*[*simp*]:
$m \leq n \Longrightarrow (f \Downarrow m) @ (f \Downarrow n \uparrow m) = f \Downarrow n$
by (*simp add*: *i-take-drop-map i-take-int-append*)

thm *List.drop-take*
lemma *i-take-drop*: $f \Downarrow n \uparrow m = f \Uparrow m \Downarrow (n - m)$
by (*simp add*: *expand-list-eq*)

thm *List.take-map*
lemma *i-take-o*[*simp*]: $(f \circ g) \Downarrow n = map\ f\ (g \Downarrow n)$
by (*simp add*: *expand-list-eq*)
thm *List.drop-map*
lemma *i-drop-o*[*simp*]: $(f \circ g) \Uparrow n = f \circ (g \Uparrow n)$
by (*simp add*: *expand-ilist-eq*)

thm *List.set-take-subset*
lemma *set-i-take-subset*: *set* $(f \Downarrow n) \subseteq$ *range* f
by (*fastsimp simp*: *in-set-conv-nth*)

thm *List.set-drop-subset*
lemma *range-i-drop-subset*: *range* $(f \Uparrow n) \subseteq$ *range* f
by *fastsimp*

thm *List.in-set-takeD*
lemma *in-set-i-takeD*: $x \in$ *set* $(f \Downarrow n) \Longrightarrow x \in$ *range* f
by (*rule subsetD*[*OF set-i-take-subset*])
thm *List.in-set-dropD*
lemma *in-range-i-takeD*: $x \in$ *range* $(f \Uparrow n) \Longrightarrow x \in$ *range* f
by (*rule subsetD*[*OF range-i-drop-subset*])

thm *List.append-eq-conv-conj*
lemma *i-append-eq-conv-conj*:
$((xs \frown f) = g) = (xs = g \Downarrow$ *length* $xs \wedge f = g \Uparrow$ *length* $xs)$
apply (*simp add*: *expand-ilist-eq expand-list-eq i-append-nth*)
apply (*rule iffI*)
apply (*clarsimp*, *rename-tac* x)
apply (*drule-tac* x=*length* $xs + x$ **in** *spec*)
apply *simp*
apply *simp*
done

thm *List.take-add*
lemma *i-take-add*: $f \Downarrow (i + j) = (f \Downarrow i)$ @ $(f \Uparrow i \Downarrow j)$
by (*simp add*: *expand-list-eq nth-append*)

thm *List.append-eq-append-conv-if*
lemma *i-append-eq-i-append-conv-if-aux*:
length $xs \leq$ *length* $ys \Longrightarrow$
$(xs \frown f = ys \frown g) = (xs = ys \downarrow$ *length* $xs \wedge f = (ys \uparrow$ *length* $xs) \frown g)$
apply (*simp add*: *expand-list-eq expand-ilist-eq i-append-nth min-eqR*)
apply (*rule iffI*)
apply *simp*
apply (*clarify*, *rename-tac* x)
apply (*drule-tac* x=*length* $xs + x$ **in** *spec*)
apply (*simp add*: *less-diff-conv add-commute*[*of* - *length* xs])
apply *simp*
done
lemma *i-append-eq-i-append-conv-if*:
$(xs \frown f = ys \frown g) =$
(*if length* $xs \leq$ *length* ys
then $xs = ys \downarrow$ *length* $xs \wedge f = (ys \uparrow$ *length* $xs) \frown g$
else $xs \downarrow$ *length* $ys = ys \wedge (xs \uparrow$ *length* $ys) \frown f = g)$

apply (*split split-if*, *intro conjI impI*)
apply (*simp add*: *i-append-eq-i-append-conv-if-aux*)
apply (*force simp*: *eq-commute*[*of xs* ⌢ *f*] *i-append-eq-i-append-conv-if-aux*)
done

thm *List.take-hd-drop*
lemma *i-take-hd-i-drop*: (*f* ⇓ *n*) @ [(*f* ⇑ *n*) *0*] = *f* ⇓ *Suc n*
by (*simp add*: *i-take-Suc-conv-app-nth*)

thm *List.id-take-nth-drop*
lemma *id-i-take-nth-i-drop*: *f* = (*f* ⇓ *n*) ⌢ (([*f n*] ⌢ *f*) ⇑ *Suc n*)
by (*simp add*: *i-drop-Suc-Cons*)

thm *List.upd-conv-take-nth-drop*
lemma *upd-conv-i-take-nth-i-drop*:
f (*n* := *x*) = (*f* ⇓ *n*) ⌢ ([*x*] ⌢ (*f* ⇑ *Suc n*))
by (*simp add*: *expand-ilist-eq nth-append i-append-nth*)

thm *nat.induct*[*of* λ*n*. *P* (*f* ↓ *n*) *n*]
theorem *i-take-induct*:
⟦ *P* (*f* ⇓ *0*); ⋀*n*. *P* (*f* ⇓ *n*) ⟹ *P* (*f* ⇓ *Suc n*) ⟧ ⟹ *P* (*f* ⇓ *n*)
by (*rule nat.induct*)

thm *i-take-induct*
theorem *take-induct*[*rule-format*]:
⟦ *P* (*s* ↓ *0*);
⋀*n*. ⟦ *Suc n* < *length s*; *P* (*s* ↓ *n*) ⟧ ⟹ *P* (*s* ↓ *Suc n*);
i < *length s*⟧
⟹ *P* (*s* ↓ *i*)
by (*induct i*, *simp*+)

theorem *i-drop-induct*:
⟦ *P* (*f* ⇑ *0*); ⋀*n*. *P* (*f* ⇑ *n*) ⟹ *P* (*f* ⇑ *Suc n*) ⟧ ⟹ *P* (*f* ⇑ *n*)
by (*rule nat.induct*)
thm *i-drop-induct*
theorem *f-drop-induct*[*rule-format*]:
⟦ *P* (*s* ↑ *0*);
⋀*n*. ⟦ *Suc n* < *length s*; *P* (*s* ↑ *n*) ⟧ ⟹ *P* (*s* ↑ *Suc n*);
i < *length s*⟧
⟹ *P* (*s* ↑ *i*)
by (*induct i*, *simp*+)

lemma *i-take-drop-eq-map*: *f* ⇑ *m* ⇓ *n* = *map f* [*m*..<*m*+*n*]
by (*simp add*: *expand-list-eq*)

thm *List.map-eq-Cons-conv*
lemma *o-eq-i-append-imp*:
f ∘ *g* = *ys* ⌢ *i* ⟹

$\exists xs\ h.\ g = xs \frown h \wedge map\ f\ xs = ys \wedge f \circ h = i$
apply (*rule-tac* $x{=}g \Downarrow (length\ ys)$ **in** *exI*)
apply (*rule-tac* $x{=}g \Uparrow (length\ ys)$ **in** *exI*)
apply (*frule arg-cong*[**where** $f{=}\lambda x.\ x \Downarrow length\ ys$])
apply (*drule arg-cong*[**where** $f{=}\lambda x.\ x \Uparrow length\ ys$])
apply *simp*
done

corollary *o-eq-i-append-conv*:
$(f \circ g = ys \frown i) =$
$(\exists xs\ h.\ g = xs \frown h \wedge map\ f\ xs = ys \wedge f \circ h = i)$
by (*fastsimp simp*: *o-eq-i-append-imp*)
corollary *i-append-eq-o-conv*:
$(ys \frown i = f \circ g) =$
$(\exists xs\ h.\ g = xs \frown h \wedge map\ f\ xs = ys \wedge f \circ h = i)$
by (*fastsimp simp*: *o-eq-i-append-imp*)

11.1.3 *zip* for infinite lists

term *zip*
definition
$i\text{-}zip :: 'a\ ilist \Rightarrow 'b\ ilist \Rightarrow ('a \times 'b)\ ilist$
where
$i\text{-}zip\ f\ g \equiv \lambda n.\ (f\ n,\ g\ n)$

lemma *i-zip-nth*: $(i\text{-}zip\ f\ g)\ n = (f\ n,\ g\ n)$
by (*simp add*: *i-zip-def*)

thm *zip-swap*
lemma *i-zip-swap*: $(\lambda(y,\ x).\ (x,\ y)) \circ i\text{-}zip\ g\ f = i\text{-}zip\ f\ g$
by (*simp add*: *expand-ilist-eq i-zip-nth*)

lemma *i-zip-i-take*: $(i\text{-}zip\ f\ g) \Downarrow n = zip\ (f \Downarrow n)\ (g \Downarrow n)$
by (*simp add*: *expand-list-eq i-zip-nth*)

lemma *i-zip-i-drop*: $(i\text{-}zip\ f\ g) \Uparrow n = i\text{-}zip\ (f \Uparrow n)\ (g \Uparrow n)$
by (*simp add*: *expand-ilist-eq i-zip-nth*)

thm *List.map-fst-zip*
lemma *fst-o-izip*: $fst \circ (i\text{-}zip\ f\ g) = f$
by (*simp add*: *expand-ilist-eq i-zip-nth*)
lemma *snd-o-i-zip*: $snd \circ (i\text{-}zip\ f\ g) = g$
by (*simp add*: *expand-ilist-eq i-zip-nth*)

thm *List.update-zip*
lemma *update-i-zip*:
$(i\text{-}zip\ f\ g)(n := xy) = i\text{-}zip\ (f(n := fst\ xy))\ (g(n := snd\ xy))$
by (*simp add*: *expand-ilist-eq i-zip-nth*)

thm *List.zip-Cons-Cons*
lemma *i-zip-Cons-Cons*:
i-zip ([*x*] ⌢ *f*) ([*y*] ⌢ *g*) = [(*x*, *y*)] ⌢ (*i-zip f g*)
by (*simp add*: *expand-ilist-eq i-zip-nth i-append-nth*)

thm *List.zip-append1*
lemma *i-zip-i-append1*:
i-zip (*xs* ⌢ *f*) *g* = *zip xs* (*g* ⇓ *length xs*) ⌢ (*i-zip f* (*g* ⇑ *length xs*))
by (*simp add*: *expand-ilist-eq i-zip-nth i-append-nth*)

thm *List.zip-append2*
lemma *i-zip-i-append2*:
i-zip f (*ys* ⌢ *g*) = *zip* (*f* ⇓ *length ys*) *ys* ⌢ (*i-zip* (*f* ⇑ *length ys*) *g*)
by (*simp add*: *expand-ilist-eq i-zip-nth i-append-nth*)

thm *List.zip-append*
lemma *i-zip-append*:
length xs = *length ys* ⟹
i-zip (*xs* ⌢ *f*) (*ys* ⌢ *g*) = *zip xs ys* ⌢ (*i-zip f g*)
by (*simp add*: *expand-ilist-eq i-zip-nth i-append-nth*)

thm *List.set-zip*
lemma *i-zip-range*: *range* (*i-zip f g*) = { (*f n*, *g n*)| *n*. *True* }
by (*fastsimp simp*: *i-zip-nth*)

thm *List.zip-update*
lemma *i-zip-update*:
i-zip (*f*(*n* := *x*)) (*g*(*n* := *y*)) = (*i-zip f g*)(*n* := (*x*, *y*))
by (*simp add*: *update-i-zip*)

lemma *i-zip-const*: *i-zip* (λ*n*. *x*) (λ*n*. *y*) = (λ*n*. (*x*, *y*))
by (*simp add*: *expand-ilist-eq i-zip-nth*)

11.1.4 Mapping functions with two arguments to infinite lists

thm *map2.simps*
definition *i-map2* ::
(∗ *Function taking two parameters* ∗)
('*a* ⇒ '*b* ⇒ '*c*) ⇒
(∗ *Lists of parameters* ∗)
'*a ilist* ⇒ '*b ilist* ⇒
'*c ilist*
where
i-map2 f xs ys ≡ λ*n*. *f* (*xs n*) (*ys n*)

lemma *i-map2-nth*: (*i-map2 f xs ys*) *n* = *f* (*xs n*) (*ys n*)
by (*simp add*: *i-map2-def*)

lemma *i-map2-Cons-Cons*:
i-map2 f (*[x]* ⌢ *xs*) (*[y]* ⌢ *ys*) =
[f x y] ⌢ (*i-map2 f xs ys*)
by (*simp add*: *fun-eq-iff i-map2-nth i-append-nth-Cons'*)

lemma *i-map2-take-ge*:
n ≤ *n1* ⟹
i-map2 f xs ys ⇓ *n* =
map2 f (*xs* ⇓ *n*) (*ys* ⇓ *n1*)
by (*simp add*: *expand-list-eq map2-length i-map2-nth map2-nth*)
lemma *i-map2-take-take*:
i-map2 f xs ys ⇓ *n* =
map2 f (*xs* ⇓ *n*) (*ys* ⇓ *n*)
by (*rule i-map2-take-ge*[*OF le-refl*])

lemma *i-map2-drop*:
(*i-map2 f xs ys*) ⇑ *n* =
(*i-map2 f* (*xs* ⇑ *n*) (*ys* ⇑ *n*))
by (*simp add*: *fun-eq-iff i-map2-nth*)

lemma *i-map2-append-append*:
length xs1 = *length ys1* ⟹
i-map2 f (*xs1* ⌢ *xs*) (*ys1* ⌢ *ys*) =
map2 f xs1 ys1 ⌢ *i-map2 f xs ys*
by (*simp add*: *fun-eq-iff i-map2-nth i-append-nth map2-length map2-nth*)

lemma *i-map2-Cons-left*:
i-map2 f (*[x]* ⌢ *xs*) *ys* =
[f x (*ys 0*)*]* ⌢ *i-map2 f xs* (*ys* ⇑ *Suc 0*)
by (*simp add*: *fun-eq-iff i-map2-nth i-append-nth-Cons'*)
lemma *i-map2-Cons-right*:
i-map2 f xs (*[y]* ⌢ *ys*) =
[f (*xs 0*) *y]* ⌢ *i-map2 f* (*xs* ⇑ *Suc 0*) *ys*
by (*simp add*: *fun-eq-iff i-map2-nth i-append-nth-Cons'*)

lemma *i-map2-append-take-drop-left*:
i-map2 f (*xs1* ⌢ *xs*) *ys* =
map2 f xs1 (*ys* ⇓ *length xs1*) ⌢
i-map2 f xs (*ys* ⇑ *length xs1*)
by (*simp add*: *fun-eq-iff map2-nth i-map2-nth i-append-nth map2-length*)
lemma *i-map2-append-take-drop-right*:
i-map2 f xs (*ys1* ⌢ *ys*) =
map2 f (*xs* ⇓ *length ys1*) *ys1* ⌢
i-map2 f (*xs* ⇑ *length ys1*) *ys*
by (*simp add*: *fun-eq-iff map2-nth i-map2-nth i-append-nth map2-length*)

thm *o-cong*
lemma *i-map2-cong*:

⟦ *xs1* = *xs2*; *ys1* = *ys2*;
⋀*x y*. ⟦ *x* ∈ *range xs2*; *y* ∈ *range ys2* ⟧ ⟹ *f x y* = *g x y* ⟧ ⟹
i-map2 f xs1 ys1 = *i-map2 g xs2 ys2*
by (*simp add*: *fun-eq-iff i-map2-nth*)

thm *o-eq-conv*
lemma *i-map2-eq-conv*:
(*i-map2 f xs ys* = *i-map2 g xs ys*) = (∀ *i*. *f* (*xs i*) (*ys i*) = *g* (*xs i*) (*ys i*))
by (*simp add*: *fun-eq-iff i-map2-nth*)

lemma *i-map2-replicate*: *i-map2 f* (λ*n*. *x*) (λ*n*. *y*) = (λ*n*. *f x y*)
by (*simp add*: *fun-eq-iff i-map2-nth*)

lemma *i-map2-i-zip-conv*:
i-map2 f xs ys = (λ(*x*,*y*). *f x y*) ∘ (*i-zip xs ys*)
by (*simp add*: *fun-eq-iff i-map2-nth i-zip-nth*)

11.2 Generalised lists as combination of finite and infinite lists

11.2.1 Basic definitions

datatype *'a glist* = *FL 'a list* | *IL 'a ilist*

thm *list.simps*
term *nth*

definition
glength :: *'a glist* ⇒ *inat*
where
glength a ≡ *case a of*
FL xs ⇒ *Fin* (*length xs*) |
IL f ⇒ ∞
definition
gCons :: *'a* ⇒ *'a glist* ⇒ *'a glist* (**infixr** #$_g$ *65*)
where
x #$_g$ *a* ≡ *case a of*
FL xs ⇒ *FL* (*x* # *xs*) |
IL g ⇒ *IL* ([*x*] ⌢ *g*)
definition
gappend :: *'a glist* ⇒ *'a glist* ⇒ *'a glist* (**infixr** @$_g$ *65*)
where
gappend a b ≡ *case a of*
FL xs ⇒ (*case b of FL ys* ⇒ *FL* (*xs* @ *ys*) | *IL f* ⇒ *IL* (*xs* ⌢ *f*)) |
IL f ⇒ *IL f*
definition
gmap :: (*'a* ⇒ *'b*) ⇒ *'a glist* ⇒ *'b glist*
where
gmap f a ≡ *case a of*
FL xs ⇒ *FL* (*map f xs*) |

```
    IL g  ⇒ IL (f ∘ g)
definition
  gtake   :: inat ⇒ 'a glist ⇒ 'a glist
where
  gtake n a ≡ case n of
    Fin m ⇒ FL (case a of
      FL xs ⇒ xs ↓ m |
      IL f  ⇒ f ⇓ m) |
    ∞ ⇒ a
definition
  gdrop   :: inat ⇒ 'a glist ⇒ 'a glist
where
  gdrop n a ≡ case n of
    Fin m ⇒ (case a of
      FL xs ⇒ FL (xs ↑ m) |
      IL f  ⇒ IL (f ⇑ m)) |
    ∞ ⇒ FL []
definition
  gset    :: 'a glist ⇒ 'a set
where
  gset a ≡ case a of
    FL xs ⇒ set xs |
    IL f  ⇒ range f
definition
  gnth    :: 'a glist ⇒ nat ⇒ 'a              (infixl !_g 100)
where
  a !_g n ≡ case a of
    FL xs ⇒ xs ! n |
    IL f  ⇒ f n

abbreviation (xsymbols)
  g-take' :: 'a glist ⇒ inat ⇒ 'a glist (infixl ↓_g 100)
where
  a ↓_g n ≡ gtake n a
abbreviation (xsymbols)
  g-drop' :: 'a glist ⇒ inat ⇒ 'a glist (infixl ↑_g 100)
where
  a ↑_g n ≡ gdrop n a
syntax (HTML output)
  g-take' :: 'a glist ⇒ inat ⇒ 'a glist   (infixl ↓_g 100)
  g-drop' :: 'a glist ⇒ inat ⇒ 'a glist   (infixl ↑_g 100)
```

11.2.2 *glength*

```
lemma glength-fin[simp]: glength (FL xs) = Fin (length xs)
by (simp add: glength-def)

lemma glength-infin[simp]: glength (IL f) = ∞
by (simp add: glength-def)
```

lemma *gappend-glength*[*simp*]: *glength* (*a* @$_g$ *b*) = *glength a* + *glength b*
by (*unfold gappend-def*, *case-tac a*, *case-tac b*, *simp*+)

lemma *gmap-glength*[*simp*]: *glength* (*gmap f a*) = *glength a*
by (*unfold gmap-def*, *case-tac a*, *simp*+)

lemma *glength-0-conv*[*simp*]: (*glength a* = *0*) = (*a* = *FL* [])
by (*unfold glength-def*, *case-tac a*, *simp*+)

lemma *glength-greater-0-conv*[*simp*]: (*0* < *glength a*) = (*a* ≠ *FL* [])
by (*simp add*: *glength-0-conv*[*symmetric*])

lemma *glength-gSuc-conv*:
(*glength a* = *iSuc n*) =
(∃ *x b*. *a* = *x* #$_g$ *b* ∧ *glength b* = *n*)
apply (*unfold glength-def gCons-def*, *rule iffI*)
apply (*case-tac a*, *rename-tac a'*)
apply (*case-tac n*, *rename-tac n'*)
apply (*rule-tac x*=*hd a'* **in** *exI*)
apply (*rule-tac x*=*FL* (*tl a'*) **in** *exI*)
apply (*simp add*: *iSuc-Fin*)
apply (*subgoal-tac a'* ≠ [])
prefer *2*
apply (*rule ccontr*, *simp*)
apply *simp*
apply *simp*
apply (*rename-tac f*)
apply (*case-tac n*, *simp add*: *iSuc-Fin*)
apply (*rule-tac x*=*f 0* **in** *exI*)
apply (*rule-tac x*=*IL* (*f* ⇑ *Suc 0*) **in** *exI*)
thm *i-take-first*
apply (*simp add*: *i-take-first*[*symmetric*])
apply (*clarsimp*, *rename-tac x b*)
apply (*case-tac a*)
apply (*case-tac b*)
apply (*simp add*: *iSuc-Fin*)+
apply (*case-tac b*)
apply (*simp add*: *iSuc-Fin*)+
done

lemma *gSuc-glength-conv*:
(*iSuc n* = *glength a*) =
(∃ *x b*. *a* = *x* #$_g$ *b* ∧ *glength b* = *n*)
by (*simp add*: *eq-commute*[*of - glength a*] *glength-gSuc-conv*)

11.2.3 @$_g$ – gappend

thm *append-Nil*

lemma *gappend-Nil*[*simp*]: $(FL\ []) \ @_g\ a = a$
by (*unfold gappend-def*, *case-tac a*, *simp*+)

lemma *gappend-Nil2*[*simp*]: $a \ @_g\ (FL\ [])= a$
by (*unfold gappend-def*, *case-tac a*, *simp*+)

lemma *gappend-is-Nil-conv*[*simp*]: $(a \ @_g\ b = FL\ []) = (a = FL\ [] \wedge b = FL\ [])$
by (*unfold gappend-def*, *case-tac a*, *case-tac b*, *simp*+)

lemma *Nil-is-gappend-conv*[*simp*]: $(FL\ [] = a \ @_g\ b) = (a = FL\ [] \wedge b = FL\ [])$
by (*simp add*: *eq-commute*[*of FL* []])

lemma *gappend-assoc*[*simp*]: $(a \ @_g\ b) \ @_g\ c = a \ @_g\ b \ @_g\ c$
by (*unfold gappend-def*, *case-tac a*, *case-tac b*, *case-tac c*, *simp*+)

lemma *gappend-infin*[*simp*]: $IL\ f \ @_g\ b = IL\ f$
by (*simp add*: *gappend-def*)

lemma *same-gappend-eq-disj*[*simp*]: $(a \ @_g\ b = a \ @_g\ c) = (glength\ a = \infty \vee b = c)$
apply (*case-tac a*)
apply *simp*
apply (*case-tac b*, *case-tac c*)
apply (*simp add*: *gappend-def*)+
apply (*case-tac c*)
apply *simp*+
done
lemma *same-gappend-eq*:
$glength\ a < \infty \Longrightarrow (a \ @_g\ b = a \ @_g\ c) = (b = c)$
by *fastsimp*

11.2.4 *gmap*

lemma *gmap-gappend*[*simp*]: $gmap\ f\ (a \ @_g\ b) = gmap\ f\ a \ @_g\ gmap\ f\ b$
by (*unfold gappend-def gmap-def*, *induct a*, *induct b*, *simp*+)

thm *map-map*
lemma *gmap-gmap*[*simp*]: $gmap\ f\ (gmap\ g\ a) = gmap\ (f \circ g)\ a$
apply (*case-tac a*)
apply (*simp add*: *gmap-def expand-ilist-eq*)+
done

thm *map-eq-conv*
lemma *gmap-eq-conv*[*simp*]: $(gmap\ f\ a = gmap\ g\ a) = (\forall x{\in}gset\ a.\ f\ x = g\ x)$
apply (*case-tac a*)
apply (*simp add*: *gmap-def gset-def o-eq-conv*)+
done

thm *map-cong*

lemma *gmap-cong*:
$[\![\ a = b;\ \bigwedge x.\ x \in gset\ b \Longrightarrow f\ x = g\ x\ ]\!] \Longrightarrow gmap\ f\ a = gmap\ g\ b$
by *simp*

thm *map-is-Nil-conv*
lemma *gmap-is-Nil-conv*: $(gmap\ f\ a = FL\ []) = (a = FL\ [])$
by (*simp add*: *glength-0-conv*[*symmetric*])

lemma *gmap-eq-imp-glength-eq*:
$gmap\ f\ a = gmap\ f\ b \Longrightarrow glength\ a = glength\ b$
by (*drule arg-cong*[**where** *f*=*glength*], *simp*)

11.2.5 *gset*

thm *set-append*
lemma *gset-gappend*[*simp*]:
$gset\ (a\ @_g\ b) =$
$(case\ a\ of\ FL\ a' \Rightarrow set\ a' \cup gset\ b \mid IL\ a' \Rightarrow range\ a')$
by (*unfold gset-def gappend-def*, *case-tac a*, *case-tac b*, *simp*+)
lemma *gset-gappend-if*:
$gset\ (a\ @_g\ b) =$
$(if\ glength\ a < \infty\ then\ gset\ a \cup gset\ b\ else\ gset\ a)$
by (*unfold gset-def gappend-def*, *case-tac a*, *case-tac b*, *simp*+)

thm *set-empty*
lemma *gset-empty*[*simp*]: $(gset\ a = \{\}) = (a = FL\ [])$
by (*unfold gset-def*, *case-tac a*, *simp*+)

thm *set-map*
lemma *gset-gmap*[*simp*]: $gset\ (gmap\ f\ a) = f\ `\ gset\ a$
by (*unfold gset-def gmap-def*, *case-tac a*, *simp*+)

thm *card-length*
lemma *icard-glength*: $icard\ (gset\ a) \leq glength\ a$
apply (*unfold icard-def gset-def glength-def*)
apply (*case-tac a*)
apply (*simp add*: *card-length*)+
done

11.2.6 $!_g$ – gnth

thm *nth-Cons-0*
lemma *gnth-gCons-0*[*simp*]: $(x\ \#_g\ a)\ !_g\ 0 = x$
by (*unfold gCons-def gnth-def*, *case-tac a*, *simp*+)

thm *nth-Cons-Suc*
lemma *gnth-gCons-Suc*[*simp*]: $(x\ \#_g\ a)\ !_g\ Suc\ n = a\ !_g\ n$
by (*unfold gCons-def gnth-def*, *case-tac a*, *simp*+)

thm *nth-append*

lemma *gnth-gappend*:
(*a* @$_g$ *b*) !$_g$ *n* =
(*if Fin n* < *glength a then a* !$_g$ *n*
else b !$_g$ (*n* − *the-Fin* (*glength a*)))
apply (*unfold glength-def gappend-def gCons-def gnth-def*)
apply (*case-tac a*, *case-tac b*)
apply (*simp add*: *nth-append*)+
done

thm *nth-append-length-plus*
lemma *gnth-gappend-length-plus*[*simp*]: (*FL xs* @$_g$ *b*) !$_g$ (*length xs* + *n*) = *b* !$_g$ *n*
by (*simp add*: *gnth-gappend*)

thm *nth-map*
lemma *gmap-gnth*[*simp*]: *Fin n* < *glength a* ⟹ *gmap f a* !$_g$ *n* = *f* (*a* !$_g$ *n*)
by (*unfold gmap-def gnth-def*, *case-tac a*, *simp*+)

thm *in-set-conv-nth*
lemma *in-gset-cong-gnth*: (*x* ∈ *gset a*) = (∃ *i*. *Fin i* < *glength a* ∧ *a* !$_g$ *i* = *x*)
apply (*unfold gset-def gnth-def*, *case-tac a*)
apply (*fastsimp simp*: *in-set-conv-nth*)+
done

11.2.7 *gtake* and *gdrop*

thm *take-0*
lemma *gtake-0*[*simp*]: *a* ↓$_g$ *0* = *FL* []
by (*unfold gtake-def*, *case-tac a*, *simp*+)

thm *drop-0*
lemma *gdrop-0*[*simp*]: *a* ↑$_g$ *0* = *a*
by (*unfold gdrop-def*, *case-tac a*, *simp*+)

lemma *gtake-Infty*[*simp*]: *a* ↓$_g$ ∞ = *a*
by (*unfold gtake-def*, *case-tac a*, *simp*+)

lemma *gdrop-Infty*[*simp*]: *a* ↑$_g$ ∞ = *FL* []
by (*unfold gdrop-def*, *case-tac a*, *simp*+)

thm *take-all*
lemma *gtake-all*[*simp*]: *glength a* ≤ *n* ⟹ *a* ↓$_g$ *n* = *a*
by (*unfold gtake-def*, *case-tac a*, *case-tac n*, *simp*+)

thm *drop-all*
lemma *gdrop-all*[*simp*]: *glength a* ≤ *n* ⟹ *a* ↑$_g$ *n* = *FL* []
by (*unfold gdrop-def*, *case-tac a*, *case-tac n*, *simp*+)

thm *take-Suc-Cons*
lemma *gtake-iSuc-gCons*[*simp*]: (*x* #$_g$ *a*) ↓$_g$ (*iSuc n*) = *x* #$_g$ *a* ↓$_g$ *n*

by (*unfold gtake-def gCons-def*, *case-tac n*, *case-tac a*, *simp-all add*: *iSuc-Fin*)

thm *drop-Suc-Cons*
lemma *gdrop-iSuc-gCons*[*simp*]: $(x \#_g a) \uparrow_g (iSuc\ n) = a \uparrow_g n$
by (*unfold gdrop-def gCons-def*, *case-tac n*, *case-tac a*, *simp-all add*: *iSuc-Fin*)

thm *take-Suc*
lemma *gtake-iSuc*: $a \neq FL\ [] \Longrightarrow a \downarrow_g (iSuc\ n) = a\ !_g\ 0\ \#_g\ (a \uparrow_g (iSuc\ 0) \downarrow_g n)$
apply (*unfold gtake-def gdrop-def gnth-def gCons-def*)
apply (*case-tac n*)
apply (*case-tac a*)
apply (*simp add*: *iSuc-Fin take-Suc hd-eq-first take-drop i-take-Suc*)+
apply (*case-tac a*)
apply (*simp add*: *hd-eq-first drop-eq-tl i-drop-Suc-conv-tl*)+
done

thm *drop-Suc*
lemma *gdrop-iSuc*: $a \uparrow_g (iSuc\ n) = a \uparrow_g (iSuc\ 0) \uparrow_g n$
by (*unfold gtake-def gdrop-def gnth-def gCons-def*, *case-tac n*, *case-tac a*, *simp-all add*: *iSuc-Fin*)

thm *nth-via-drop*
lemma *gnth-via-grop*: $a \uparrow_g (Fin\ n) = x \#_g b \Longrightarrow a\ !_g\ n = x$
apply (*unfold gdrop-def gnth-def gCons-def*)
apply (*case-tac a*, *case-tac b*)
apply (*simp add*: *nth-via-drop*)+
apply (*case-tac b*)
apply (*fastsimp intro*: *nth-via-i-drop*)+
done

thm *take-Suc-conv-app-nth*[*no-vars*]
thm *i-take-Suc-conv-app-nth*
lemma *gtake-iSuc-conv-gapp-gnth*:
$Fin\ n < glength\ a \Longrightarrow a \downarrow_g Fin\ (Suc\ n) = a \downarrow_g (Fin\ n)\ @_g\ FL\ [a\ !_g\ n]$
apply (*unfold glength-def gtake-def gappend-def gnth-def*)
apply (*case-tac a*)
apply (*simp add*: *take-Suc-conv-app-nth i-take-Suc-conv-app-nth*)+
done

thm *drop-Suc-conv-tl*
lemma *gdrop-iSuc-conv-tl*:
$Fin\ n < glength\ a \Longrightarrow a\ !_g\ n\ \#_g\ a \uparrow_g Fin\ (Suc\ n) = a \uparrow_g Fin\ n$
apply (*unfold glength-def gdrop-def gappend-def gnth-def gCons-def*)
apply (*case-tac a*)
apply (*simp add*: *drop-Suc-conv-tl i-drop-Suc-conv-tl*)+
done

thm *length-take*

lemma *glength-gtake*[*simp*]: *glength* $(a \downarrow_g n) = min\ (glength\ a)\ n$
by (*unfold glength-def gtake-def*, *case-tac n*, *case-tac a*, *simp+*)

thm *length-drop*
lemma *glength-drop*[*simp*]: *glength* $(a \uparrow_g (Fin\ n)) = glength\ a - (Fin\ n)$
by (*unfold glength-def gdrop-def*, *case-tac a*, *case-tac n*, *simp+*)

end

12 ListInf-Prefix: Prefices on finite and infinite lists

theory *ListInf-Prefix*
imports $*ISABELLE-HOME*/src/HOL/Library/List-Prefix\ ListInf$
begin

12.1 Additional list prefix results

lemma *prefix-eq-prefix-take-ex*: $(xs \leq ys) = (\exists n.\ ys \downarrow n = xs)$
apply (*unfold prefix-def*, *safe*)
apply (*rule-tac x*=*length xs* **in** *exI*, *simp*)
apply (*rule-tac x*=*ys* $\uparrow$ *n* **in** *exI*, *simp*)
done

lemma *prefix-take-eq-prefix-take-ex*: $(ys \downarrow (length\ xs) = xs) = (\exists n.\ ys \downarrow n = xs)$
by (*fastsimp simp*: *min-def*)

lemma *prefix-eq-prefix-take*: $(xs \leq ys) = (ys \downarrow (length\ xs) = xs)$
by (*simp only*: *prefix-eq-prefix-take-ex prefix-take-eq-prefix-take-ex*)

lemma *strict-prefix-take-eq-strict-prefix-take-ex*:
$(ys \downarrow (length\ xs) = xs \wedge xs \neq ys) =$
$((\exists n.\ ys \downarrow n = xs) \wedge xs \neq ys)$
by (*simp add*: *prefix-take-eq-prefix-take-ex*)

lemma *strict-prefix-eq-strict-prefix-take-ex*: $(xs < ys) = ((\exists n.\ ys \downarrow n = xs) \wedge xs \neq ys)$
by (*simp add*: *strict-prefix-def prefix-eq-prefix-take-ex*)
lemma *strict-prefix-eq-strict-prefix-take*: $(xs < ys) = (ys \downarrow (length\ xs) = xs \wedge xs \neq ys)$
by (*simp only*: *strict-prefix-eq-strict-prefix-take-ex strict-prefix-take-eq-strict-prefix-take-ex*)

lemma *take-imp-prefix*: $xs \downarrow n \leq xs$
by (*rule take-is-prefix*)

lemma *eq-imp-prefix*: $xs = (ys::'a\ list) \Longrightarrow xs \leq ys$
by *simp*

lemma *le-take-imp-prefix*: $a \leq b \Longrightarrow xs \downarrow a \leq xs \downarrow b$
by (*fastsimp simp*: *prefix-eq-prefix-take-ex min-def*)

lemma *take-prefix-imp-le*:
$[\![\ a \leq length\ xs;\ xs \downarrow a \leq xs \downarrow b\ ]\!] \Longrightarrow a \leq b$
by (*drule prefix-length-le*, *simp*)

lemma *take-prefix-le-conv*:
$a \leq length\ xs \Longrightarrow (xs \downarrow a \leq xs \downarrow b) = (a \leq b)$
apply (*rule iffI*)
apply (*rule take-prefix-imp-le*, *assumption*+)
apply (*rule le-take-imp-prefix*, *assumption*+)
done
lemma *append-imp-prefix*[*simp*, *intro*]: $a \leq a\ @\ b$
by (*unfold prefix-def*, *blast*)

lemma *prefix-imp-take-eq*:
$[\![\ n \leq length\ xs;\ xs \leq ys\ ]\!] \Longrightarrow xs \downarrow n = ys \downarrow n$
by (*clarsimp simp*: *prefix-def*)

lemma *prefix-length-le-eq-conv*: $(xs \leq ys \wedge length\ ys \leq length\ xs) = (xs = ys)$
apply (*rule iffI*)
apply (*erule conjE*)
apply (*frule prefix-length-le*)
apply (*simp add*: *prefix-eq-prefix-take*)
apply *simp*
done

lemma *take-length-prefix-conv*:
$length\ xs \leq length\ ys \Longrightarrow (ys \downarrow length\ xs \leq xs) = (xs \leq ys)$
by (*fastsimp simp*: *prefix-eq-prefix-take*)

lemma *append-eq-imp-take*:
$[\![\ k \leq length\ xs;\ length\ r1 = k;\ r1\ @\ r2 = xs\ ]\!] \Longrightarrow r1 = xs \downarrow k$
by *fastsimp*

lemma *take-the-conv*:
$xs \downarrow k = (if\ length\ xs \leq k\ then\ xs\ else\ (THE\ r.\ length\ r = k \wedge (\exists\, r2.\ r\ @\ r2 = xs)))$
apply (*clarsimp simp*: *linorder-not-le*)
apply (*rule the1I2*)
apply (*simp add*: *Ex1-def*)
apply (*rule-tac x=xs* $\downarrow$ *k* **in** *exI*)
apply (*intro conjI*)
apply (*simp add*: *min-eqR*)

apply (*rule-tac x=xs ↑ k* **in** *exI*, *simp*)
apply *fastsimp*
apply *fastsimp*
done

lemma *prefix-refl*: $xs \leq (xs::'a\ list)$
by (*rule order-refl*)

lemma *prefix-trans*: $[\![xs \leq ys; (ys::'a\ list) \leq zs]\!] \Longrightarrow xs \leq zs$
by (*rule order-trans*)

lemma *prefix-antisym*: $[\![xs \leq ys; (ys::'a\ list) \leq xs]\!] \Longrightarrow xs = ys$
by (*rule order-antisym*)

12.2 Counting equal pairs

Counting number of equal elements in two lists

definition
mirror-pair :: $('a \times 'b) \Rightarrow ('b \times 'a)$
where
mirror-pair p ≡ (*snd p*, *fst p*)

lemma *zip-mirror*[*rule-format*]:
$[\![i < min\ (length\ xs)\ (length\ ys);$
p1 = (*zip xs ys*) ! *i*; *p2* = (*zip ys xs*) ! *i* $]\!] \Longrightarrow$
mirror-pair p1 = *p2*
by (*simp add*: *mirror-pair-def*)

definition
equal-pair :: $('a \times 'a) \Rightarrow bool$
where
equal-pair p ≡ (*fst p* = *snd p*)

lemma *mirror-pair-equal*: *equal-pair* (*mirror-pair p*) = (*equal-pair p*)
by (*fastsimp simp*: *mirror-pair-def equal-pair-def*)

primrec
equal-pair-count :: $('a \times 'a)\ list \Rightarrow nat$
where
equal-pair-count [] = *0*
| *equal-pair-count* (*p* # *ps*) = (
if (*fst p* = *snd p*)
then Suc (*equal-pair-count ps*)
else 0)

lemma *equal-pair-count-le*: *equal-pair-count xs* ≤ *length xs*
by (*induct xs*, *simp-all*)

lemma *equal-pair-count-0*:
fst (*hd ps*) ≠ *snd* (*hd ps*) ⟹ *equal-pair-count ps* = *0*
by (*case-tac ps*, *simp-all*)

lemma *equal-pair-count-Suc*:
equal-pair-count ((*a*, *a*) # *ps*) = *Suc* (*equal-pair-count ps*)
by *simp*

lemma *equal-pair-count-eq-pairwise*[*rule-format*]:
⟦ *length ps1* = *length ps2*;
∀ *i*<*length ps2*. *equal-pair* (*ps1* ! *i*) = *equal-pair*(*ps2* ! *i*) ⟧
⟹ *equal-pair-count ps1* = *equal-pair-count ps2*
apply (*induct rule*: *list-induct2*)
apply *simp*
apply (*fastsimp simp add*: *equal-pair-def*)
done

lemma *equal-pair-count-mirror-pairwise*[*rule-format*]:
⟦ *length ps1* = *length ps2*;
∀ *i*<*length ps2*. *ps1* ! *i* = *mirror-pair* (*ps2* ! *i*) ⟧
⟹ *equal-pair-count ps1* = *equal-pair-count ps2*
apply (*rule equal-pair-count-eq-pairwise*, *assumption*)
thm *mirror-pair-equal*
apply (*simp add*: *mirror-pair-equal*)
done

lemma *equal-pair-count-correct*:
⋀*i*. *i* < *equal-pair-count ps* ⟹ *equal-pair* (*ps* ! *i*)
apply (*induct ps*)
apply *simp*
apply *simp*
apply (*split split-if-asm*)
apply (*case-tac i*)
apply (*simp add*: *equal-pair-def*)+
done
thm *equal-pair-count-correct*

lemma *equal-pair-count-maximality-aux*[*rule-format*]: ⋀*i*.
i = *equal-pair-count ps* ⟹ *length ps* = *i* ∨ ¬ *equal-pair* (*ps* ! *i*)
apply (*induct ps*)
apply *simp*
apply (*simp add*: *equal-pair-def*)
done
thm *equal-pair-count-maximality-aux*
corollary *equal-pair-count-maximality1a*[*rule-format*]:
equal-pair-count ps = *length ps* ∨ ¬ *equal-pair* (*ps*!*equal-pair-count ps*)

thm *equal-pair-count-maximality-aux*[*of equal-pair-count ps ps*, *simplified*]
apply (*insert equal-pair-count-maximality-aux*[*of equal-pair-count ps ps*])
apply *clarsimp*
done

corollary *equal-pair-count-maximality1b*[*rule-format*]:
equal-pair-count ps ≠ *length ps* ⟹
¬ *equal-pair* (*ps*!*equal-pair-count ps*)
by (*insert equal-pair-count-maximality1a*[*of ps*], *simp*)

lemma *equal-pair-count-maximality2a*[*rule-format*]:
equal-pair-count ps = *length ps* ∨ (∗ *either all pairs are equal* ∗)
(∀ *i*≥*equal-pair-count ps*.(∃ *j*≤*i*. ¬*equal-pair* (*ps* ! *j*)))
apply *clarsimp*
apply (*rule-tac x*=*equal-pair-count ps* **in** *exI*)
thm *equal-pair-count-maximality1b equal-pair-count-le*
apply (*simp add*: *equal-pair-count-maximality1b equal-pair-count-le*)
done
corollary *equal-pair-count-maximality2b*[*rule-format*]:
equal-pair-count ps ≠ *length ps* ⟹
∀ *i*≥*equal-pair-count ps*.(∃ *j*≤*i*. ¬*equal-pair* (*ps*!*j*))
by (*insert equal-pair-count-maximality2a*[*of ps*], *simp*)

lemmas *equal-pair-count-maximality* =
equal-pair-count-maximality1a equal-pair-count-maximality1b
equal-pair-count-maximality2a equal-pair-count-maximality2b
thm
equal-pair-count-correct[*no-vars*]
equal-pair-count-maximality[*no-vars*]

12.3 Prefix length

Length of the prefix infimum

definition
inf-prefix-length :: *'a list* ⇒ *'a list* ⇒ *nat*
where
inf-prefix-length xs ys ≡ *equal-pair-count* (*zip xs ys*)

value *int* (*inf-prefix-length* [*1*::*int*,*2*,*3*,*4*,*7*,*8*,*15*] [*1*::*int*,*2*,*3*,*4*,*7*,*15*])
value *int* (*inf-prefix-length* [*1*::*int*,*2*,*3*,*4*] [*1*::*int*,*2*,*3*,*4*,*7*,*15*])
value *int* (*inf-prefix-length* [] [*1*::*int*,*2*,*3*,*4*,*7*,*15*])
value *int* (*inf-prefix-length* [*1*::*int*,*2*,*3*,*4*,*5*] [*1*::*int*,*2*,*3*,*4*,*5*])

lemma *inf-prefix-length-commute*[*rule-format*]:
inf-prefix-length xs ys = *inf-prefix-length ys xs*
apply (*unfold inf-prefix-length-def*)
thm *equal-pair-count-mirror-pairwise*
apply (*insert equal-pair-count-mirror-pairwise*[*of zip xs ys zip ys xs*])
apply (*simp add*: *equal-pair-count-mirror-pairwise*[*of zip xs ys zip ys xs*] *min-ac*

mirror-pair-def)
done

lemma *inf-prefix-length-leL*[*intro*]:
inf-prefix-length xs ys ≤ *length xs*
apply (*unfold inf-prefix-length-def*)
apply (*insert equal-pair-count-le*[*of zip xs ys*])
apply *simp*
done
corollary *inf-prefix-length-leR*[*intro*]:
inf-prefix-length xs ys ≤ *length ys*
thm *inf-prefix-length-commute*[*of xs ys*]
by (*simp add*: *inf-prefix-length-commute*[*of xs*] *inf-prefix-length-leL*)
lemmas *inf-prefix-length-le* =
inf-prefix-length-leL
inf-prefix-length-leR

lemma *inf-prefix-length-le-min*[*rule-format*]:
inf-prefix-length xs ys ≤ *min* (*length xs*) (*length ys*)
by (*simp add*: *inf-prefix-length-le*)

thm *equal-pair-count-0*
lemma *hd-inf-prefix-length-0*:
hd xs ≠ *hd ys* ⟹ *inf-prefix-length xs ys* = *0*
apply (*unfold inf-prefix-length-def*)
apply (*case-tac xs* = [], *simp*)
apply (*case-tac ys* = [], *simp*)
apply (*simp add*: *equal-pair-count-0 hd-zip*)
done

lemma *inf-prefix-length-NilL*[*simp*]: *inf-prefix-length* [] *ys* = *0*
by (*simp add*: *inf-prefix-length-def*)
lemma *inf-prefix-length-NilR*[*simp*]: *inf-prefix-length xs* [] = *0*
by (*simp add*: *inf-prefix-length-def*)

thm *equal-pair-count-Suc*
lemma *inf-prefix-length-Suc*[*simp*]:
inf-prefix-length (*a* # *xs*) (*a* # *ys*) = *Suc* (*inf-prefix-length xs ys*)
by (*simp add*: *inf-prefix-length-def*)

thm *equal-pair-count-correct*
lemma *inf-prefix-length-correct*:
i < *inf-prefix-length xs ys* ⟹ *xs* ! *i* = *ys* ! *i*
thm *order-less-le-trans*[*OF* - *inf-prefix-length-leL*]
apply (*frule order-less-le-trans*[*OF* - *inf-prefix-length-leL*])
apply (*frule order-less-le-trans*[*OF* - *inf-prefix-length-leR*])
apply (*unfold inf-prefix-length-def*)
apply (*drule equal-pair-count-correct*)
apply (*simp add*: *equal-pair-def*)

done

corollary *nth-neq-imp-inf-prefix-length-le*:
xs ! *i* ≠ *ys* ! *i* ⟹ *inf-prefix-length xs ys* ≤ *i*
apply (*rule ccontr*)
apply (*simp add*: *inf-prefix-length-correct*)
done

thm *equal-pair-count-maximality1b*
lemma *inf-prefix-length-maximality1*[*rule-format*]:
inf-prefix-length xs ys ≠ *min* (*length xs*) (*length ys*) ⟹
xs ! (*inf-prefix-length xs ys*) ≠ *ys* ! (*inf-prefix-length xs ys*)
thm *equal-pair-count-maximality1b*[*of zip xs ys*]
thm *equal-pair-count-maximality1b*[*of zip xs ys*, *folded inf-prefix-length-def*]
apply (*insert equal-pair-count-maximality1b*[*of zip xs ys*, *folded inf-prefix-length-def*], *simp*)
apply (*drule neq-le-trans*)
apply (*simp add*: *inf-prefix-length-le*)
apply (*simp add*: *inf-prefix-length-def equal-pair-def*)
done

thm *equal-pair-count-maximality2b*
corollary *inf-prefix-length-maximality2*[*rule-format*]:
⟦ *inf-prefix-length xs ys* ≠ *min* (*length xs*) (*length ys*);
inf-prefix-length xs ys ≤ *i* ⟧ ⟹
∃*j*≤*i*. *xs* ! *j* ≠ *ys* ! *j*
apply (*rule-tac x*=*inf-prefix-length xs ys* **in** *exI*)
apply (*simp add*: *inf-prefix-length-maximality1 inf-prefix-length-le-min*)
done

lemmas *inf-prefix-length-maximality* =
inf-prefix-length-maximality1 inf-prefix-length-maximality2

lemma *inf-prefix-length-append*[*simp*]:
inf-prefix-length (*zs* @ *xs*) (*zs* @ *ys*) =
length zs + *inf-prefix-length xs ys*
apply (*induct zs*)
apply *simp*
apply (*simp add*: *inf-prefix-length-def*)
done

lemma *inf-prefix-length-take-correct*:
n ≤ *inf-prefix-length xs ys* ⟹ *xs* ↓ *n* = *ys* ↓ *n*
apply (*frule order-trans*[*OF - inf-prefix-length-leL*])
apply (*frule order-trans*[*OF - inf-prefix-length-leR*])
apply (*simp add*: *list-eq-iff inf-prefix-length-correct*)
done

lemma *inf-prefix-length-0-imp-hd-neq*:
⟦ *xs* ≠ []; *ys* ≠ []; *inf-prefix-length xs ys* = *0* ⟧ ⟹ *hd xs* ≠ *hd ys*
apply (*rule ccontr*)
thm *inf-prefix-length-maximality2*[*of xs ys 0*]
apply (*insert inf-prefix-length-maximality2*[*of xs ys 0*])
apply (*simp add*: *hd-eq-first*)
done

12.4 Prefix infimum

definition
inf-prefix :: *'a list* ⇒ *'a list* ⇒ *'a list* (**infixl** ⊓ *70*)
where
xs ⊓ *ys* ≡ *xs* ↓ (*inf-prefix-length xs ys*)

lemma *length-inf-prefix*: *length* (*xs* ⊓ *ys*) = *inf-prefix-length xs ys*
by (*simp add*: *inf-prefix-def min-eqR inf-prefix-length-leL*)

lemma *inf-prefix-commute*: *xs* ⊓ *ys* = *ys* ⊓ *xs*
by (*simp add*: *inf-prefix-def inf-prefix-length-commute*[*of ys*] *inf-prefix-length-take-correct*)

lemma *inf-prefix-takeL*: *xs* ⊓ *ys* = *xs* ↓ (*inf-prefix-length xs ys*)
by (*simp add*: *inf-prefix-def*)
lemma *inf-prefix-takeR*: *xs* ⊓ *ys* = *ys* ↓ (*inf-prefix-length xs ys*)
by (*subst inf-prefix-commute*, *subst inf-prefix-length-commute*, *rule inf-prefix-takeL*)

thm *inf-prefix-length-correct*
lemma *inf-prefix-correct*: *i* < *length* (*xs* ⊓ *ys*) ⟹ *xs* ! *i* = *ys* ! *i*
by (*simp add*: *length-inf-prefix inf-prefix-length-correct*)
corollary *inf-prefix-correctL*:
i < *length* (*xs* ⊓ *ys*) ⟹ (*xs* ⊓ *ys*) ! *i* = *xs* ! *i*
by (*simp add*: *inf-prefix-takeL*)
corollary *inf-prefix-correctR*:
i < *length* (*xs* ⊓ *ys*) ⟹ (*xs* ⊓ *ys*) ! *i* = *ys* ! *i*
by (*simp add*: *inf-prefix-takeR*)

thm *inf-prefix-length-take-correct*
lemma *inf-prefix-take-correct*:
n ≤ *length* (*xs* ⊓ *ys*) ⟹ *xs* ↓ *n* = *ys* ↓ *n*
by (*simp add*: *length-inf-prefix inf-prefix-length-take-correct*)

lemma *is-inf-prefix*[*rule-format*]:
⟦ *length zs* = *length* (*xs* ⊓ *ys*);
⋀*i*. *i* < *length* (*xs* ⊓ *ys*) ⟹ *zs* ! *i* = *xs* ! *i* ∧ *zs* ! *i* = *ys* ! *i* ⟧ ⟹
zs = *xs* ⊓ *ys*
by (*simp add*: *list-eq-iff inf-prefix-def*)

thm *hd-inf-prefix-length-0*
lemma *hd-inf-prefix-Nil*: *hd xs* ≠ *hd ys* ⟹ *xs* ⊓ *ys* = []

by (*simp add*: *inf-prefix-def hd-inf-prefix-length-0*)

thm *inf-prefix-length-0-imp-hd-neq*
lemma *inf-prefix-Nil-imp-hd-neq*:
⟦ *xs* ≠ []; *ys* ≠ []; *xs* ⊓ *ys* = [] ⟧ ⟹ *hd xs* ≠ *hd ys*
by (*simp add*: *inf-prefix-def inf-prefix-length-0-imp-hd-neq*)

lemma *length-inf-prefix-append*[*simp*]:
length ((*zs* @ *xs*) ⊓ (*zs* @ *ys*)) =
length zs + *length* (*xs* ⊓ *ys*)
by (*simp add*: *length-inf-prefix*)

lemma *inf-prefix-append*[*simp*]: (*zs* @ *xs*) ⊓ (*zs* @ *ys*) = *zs* @ (*xs* ⊓ *ys*)
apply (*rule is-inf-prefix*[*symmetric*], *simp*)
apply (*clarsimp simp*: *nth-append*)
thm *inf-prefix-correctL inf-prefix-correctR*
apply (*intro conjI inf-prefix-correctL inf-prefix-correctR*, *simp*+)
done

lemma *hd-neq-inf-prefix-append*:
hd xs ≠ *hd ys* ⟹ (*zs* @ *xs*) ⊓ (*zs* @ *ys*) = *zs*
by (*simp add*: *hd-inf-prefix-Nil*)

thm *inf-prefix-length-NilL*
lemma *inf-prefix-NilL*[*simp*]: [] ⊓ *ys* = []
by (*simp del*: *length-0-conv add*: *length-0-conv*[*symmetric*] *length-inf-prefix*)
corollary *inf-prefix-NilR*[*simp*]: *xs* ⊓ [] = []
by (*simp add*: *inf-prefix-commute*)
lemmas *inf-prefix-Nil* = *inf-prefix-NilL inf-prefix-NilR*
thm *inf-prefix-Nil*

lemma *inf-prefix-Cons*[*simp*]: (*a* # *xs*) ⊓ (*a* # *ys*) = *a* # *xs* ⊓ *ys*
by (*insert inf-prefix-append*[*of* [*a*] *xs ys*], *simp*)
corollary *inf-prefix-hd*[*simp*]: *hd* ((*a* # *xs*) ⊓ (*a* # *ys*)) = *a*
by *simp*

lemma *inf-prefix-le1*: *xs* ⊓ *ys* ≤ *xs*
by (*simp add*: *inf-prefix-takeL take-imp-prefix*)
lemma *inf-prefix-le2*: *xs* ⊓ *ys* ≤ *ys*
by (*simp add*: *inf-prefix-takeR take-imp-prefix*)

thm *min-le-iff-conj*
lemma *le-inf-prefix-iff*: (*x* ≤ *y* ⊓ *z*) = (*x* ≤ *y* ∧ *x* ≤ *z*)
apply (*rule iffI*)
apply (*blast intro*: *order-trans inf-prefix-le1 inf-prefix-le2*)
apply (*clarsimp simp*: *prefix-def*)

done
lemma *le-imp-le-inf-prefix*: $[\![x \leq y; x \leq z]\!] \Longrightarrow x \leq y \sqcap z$
thm *le-inf-prefix-iff*[*THEN iffD2*]
by (*rule le-inf-prefix-iff*[*THEN iffD2*], *simp*)

interpretation *prefix*:
semilattice-inf
op $\leq$ :: $'a$ *list* $\Rightarrow$ $'a$ *list* $\Rightarrow$ *bool*
op $<$ :: $'a$ *list* $\Rightarrow$ $'a$ *list* $\Rightarrow$ *bool*
op $\sqcap$ :: $'a$ *list* $\Rightarrow$ $'a$ *list* $\Rightarrow$ $'a$ *list*
apply *intro-locales*
apply (*rule class.semilattice-inf-axioms.intro*)
apply (*rule inf-prefix-le1*)
apply (*rule inf-prefix-le2*)
apply (*rule le-imp-le-inf-prefix*, *assumption*+)
done

12.5 Prefices for infinite lists

thm *prefix-def*
term $\lambda x\ y.\ x \leq (y::'a\ list)$
definition
iprefix :: $'a$ *list* $\Rightarrow$ $'a$ *ilist* $\Rightarrow$ *bool* (**infixl** $\sqsubseteq$ *50*)
where
$xs \sqsubseteq f \equiv \exists g.\ f = xs \frown g$

thm *prefix-eq-prefix-take*
lemma *iprefix-eq-iprefix-take*: $(xs \sqsubseteq f) = (f \Downarrow length\ xs = xs)$
apply (*unfold iprefix-def*)
apply (*rule iffI*)
apply *clarsimp*
apply (*rule-tac x=f* $\Uparrow$ (*length xs*) **in** *exI*)
thm *i-append-i-take-i-drop-id*
apply (*subst i-append-i-take-i-drop-id*[**where** *n=length xs*, *symmetric*], *simp*)
done

thm *prefix-take-eq-prefix-take-ex*
lemma *iprefix-take-eq-iprefix-take-ex*:
$(f \Downarrow length\ xs = xs) = (\exists n.\ f \Downarrow n = xs)$
apply (*rule iffI*)
apply (*rule-tac x=length xs* **in** *exI*, *assumption*)
apply *clarsimp*
done

thm *prefix-eq-prefix-take-ex*
lemma *iprefix-eq-iprefix-take-ex*: $(xs \sqsubseteq f) = (\exists n.\ f \Downarrow n = xs)$
by (*simp add*: *iprefix-eq-iprefix-take iprefix-take-eq-iprefix-take-ex*)

thm *take-imp-prefix*

lemma *i-take-imp-iprefix*[*intro*]: $f \Downarrow n \sqsubseteq f$
by (*simp add*: *iprefix-eq-iprefix-take*)

thm *take-prefix-le-conv*
lemma *i-take-prefix-le-conv*: $(f \Downarrow a \leq f \Downarrow b) = (a \leq b)$
by (*fastsimp simp*: *prefix-eq-prefix-take list-eq-iff*)

thm *append-imp-prefix*
lemma *i-append-imp-iprefix*[*simp*,*intro*]: $xs \sqsubseteq xs \frown f$
by (*simp add*: *iprefix-def*)

thm *prefix-imp-take-eq*
lemma *iprefix-imp-take-eq*:
$[\![n \leq \mathit{length}\ xs;\ xs \sqsubseteq f]\!] \Longrightarrow xs \downarrow n = f \Downarrow n$
by (*clarsimp simp*: *iprefix-eq-iprefix-take-ex min-eqR*)

thm *prefix-refl*
thm *prefix-trans*
thm *prefix-antisym*
lemma *prefix-iprefix-trans*: $[\![xs \leq ys;\ ys \sqsubseteq f]\!] \Longrightarrow xs \sqsubseteq f$
by (*fastsimp simp*: *iprefix-eq-iprefix-take-ex prefix-eq-prefix-take-ex*)

thm *take-length-prefix-conv*
lemma *i-take-length-prefix-conv*: $(f \Downarrow \mathit{length}\ xs \leq xs) = (xs \sqsubseteq f)$
thm *prefix-length-le-eq-conv*
by (*simp add*: *iprefix-eq-iprefix-take prefix-length-le-eq-conv*[*symmetric*])

thm *List-Prefix.prefixI*
lemma *iprefixI*[*intro?*]: $f = xs \frown g \Longrightarrow xs \sqsubseteq f$
by (*unfold iprefix-def*, *simp*)
thm *List-Prefix.prefixE*
lemma *iprefixE*[*elim?*]: $[\![xs \sqsubseteq f;\ \bigwedge g.\ f = xs \frown g \Longrightarrow C]\!] \Longrightarrow C$
by (*unfold iprefix-def*, *blast*)

thm *List-Prefix.Nil-prefix*
lemma *Nil-iprefix*[*iff*]: $[] \sqsubseteq f$
by (*unfold iprefix-def*, *simp*)

thm *List-Prefix.same-prefix-prefix*
lemma *same-prefix-iprefix*[*simp*]: $(xs @ ys \sqsubseteq xs \frown f) = (ys \sqsubseteq f)$
by (*simp add*: *iprefix-eq-iprefix-take*)

thm *List-Prefix.prefix-prefix*
lemma *prefix-iprefix*[*simp*]: $xs \leq ys \Longrightarrow xs \sqsubseteq ys \frown f$
by (*clarsimp simp*: *prefix-def iprefix-def i-append-assoc*[*symmetric*] *simp del*: *i-append-assoc*)

thm *List-Prefix.append-prefixD*
lemma *append-iprefixD*: $xs @ ys \sqsubseteq f \Longrightarrow xs \sqsubseteq f$

by (*clarsimp simp*: *iprefix-def i-append-assoc*[*symmetric*] *simp del*: *i-append-assoc*)

lemma *iprefix-length-le-imp-prefix*:
⟦ *xs* ⊑ *ys* ⌢ *f*; *length xs* ≤ *length ys* ⟧ ⟹ *xs* ≤ *ys*
by (*clarsimp simp*: *iprefix-eq-iprefix-take-ex take-is-prefix*)

thm *List-Prefix.prefix-append*
lemma *iprefix-i-append*:
(*xs* ⊑ *ys* ⌢ *f*) = (*xs* ≤ *ys* ∨ (∃ *zs*. *xs* = *ys* @ *zs* ∧ *zs* ⊑ *f*))
apply (*rule iffI*)
apply (*case-tac length xs* ≤ *length ys*)
apply (*blast intro*: *iprefix-length-le-imp-prefix*)
apply (*rule disjI2*)
apply (*rule-tac x*=*f* ⇓ (*length xs* − *length ys*) **in** *exI*)
apply (*simp add*: *iprefix-eq-iprefix-take*)
apply *fastsimp*
done

lemma *i-append-one-iprefix*:
xs ⊑ *f* ⟹ *xs* @ [*f* (*length xs*)] ⊑ *f*
by (*simp add*: *iprefix-eq-iprefix-take i-take-Suc-conv-app-nth*)

lemma *iprefix-same-length-le*:
⟦ *xs* ⊑ *f*; *ys* ⊑ *f*; *length xs* ≤ *length ys* ⟧ ⟹ *xs* ≤ *ys*
by (*clarsimp simp*: *iprefix-eq-iprefix-take-ex i-take-prefix-le-conv*)

thm *List-Prefix.prefix-same-cases*
lemma *iprefix-same-cases*:
⟦ *xs* ⊑ *f*; *ys* ⊑ *f* ⟧ ⟹ *xs* ≤ *ys* ∨ *ys* ≤ *xs*
apply (*case-tac length xs* ≤ *length ys*)
apply (*simp add*: *iprefix-same-length-le*)+
done

thm *List-Prefix.set-mono-prefix*
lemma *set-mono-iprefix*: *xs* ⊑ *f* ⟹ *set xs* ⊆ *range f*
by (*unfold iprefix-def*, *fastsimp*)

end
theory *ListInfinite*
imports
CommonSet/SetIntervalStep
ListInf/ListInf-Prefix
begin
end