Infinite Lists

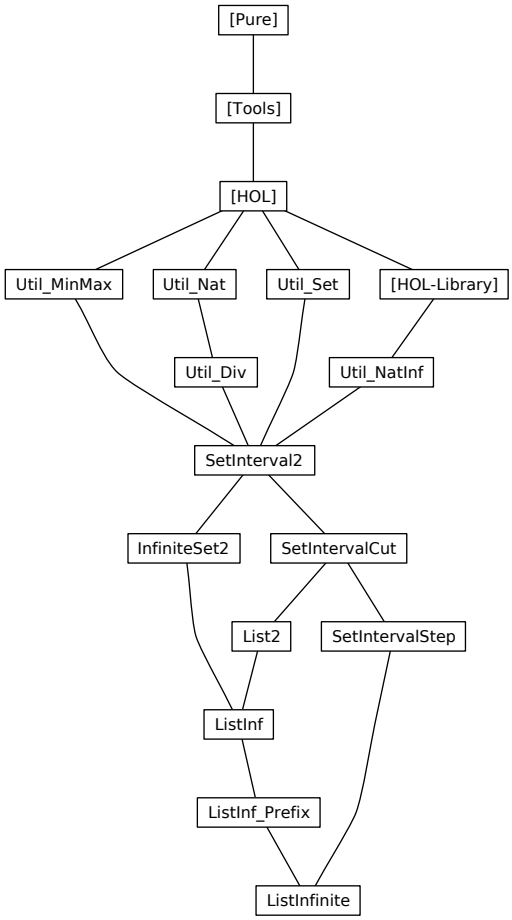
David Trachtenherz

March 17, 2025

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 Convenience results for set quantifiers

theory *Util-Set*
imports *Main*
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

lemma *if-P'*: $[\![\ P;\ x = z\ ]\!] \Longrightarrow (\mathit{if}\ P\ \mathit{then}\ x\ \mathit{else}\ y) = z$ **by** *simp*
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*: $\mathit{f1} \circ \mathit{f2} = (\lambda x.\ \mathit{f1}\ (\mathit{f2}\ x))$ **by** (*simp add*: *comp-def*)
lemma *comp3-conv*: $\mathit{f1} \circ \mathit{f2} \circ \mathit{f3} = (\lambda x.\ \mathit{f1}\ (\mathit{f2}\ (\mathit{f3}\ x)))$ **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** *fastforce*
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*
lemma *all-imp-ball*: $\forall x.\ P\ x \Longrightarrow \forall x{\in}A.\ P\ x$ **by** *blast*

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 Order and linear order: min and max

theory *Util-MinMax*
imports *Main*
begin

2.1 Additional lemmata about *min* and *max*

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.bounded-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.boundedI*

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

lemmas *min-eqR* = *min.absorb2*
lemmas *min-eq* = *min-eqL min-eqR*

lemma *max-less-imp-conj*:*max x y* < *b* ⟹ *x* < (*b*::('*a*::*linorder*)) ∧ *y* < *b* **by** *simp*
lemma *conj-less-imp-max*:⟦ *x* < (*b*::('*a*::*linorder*)); *y* < *b* ⟧ ⟹ *max x y* < *b* **by** *simp*

lemmas *max-le-iff-conj* = *max.bounded-iff*
lemma *max-le-imp-conj*:*max x y* ≤ *b* ⟹ *x* ≤ (*b*::('*a*::*linorder*)) ∧ *y* ≤ *b* **by** *simp*

lemmas *conj-le-imp-max* = *max.boundedI*

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

lemmas *max-eqR* = *max.absorb2*
lemmas *max-eq* = *max-eqL max-eqR*

lemmas *le-minI1* = *min.cobounded1*

lemmas *le-minI2* = *min.cobounded2*

lemma
min-le-monoR: (*a*::'*a*::*linorder*) ≤ *b* ⟹ *min x a* ≤ *min x b* **and**
min-le-monoL: (*a*::'*a*::*linorder*) ≤ *b* ⟹ *min a x* ≤ *min b x*
by (*fastforce simp*: *min.mono min-def*)+
lemma
max-le-monoR: (*a*::'*a*::*linorder*) ≤ *b* ⟹ *max x a* ≤ *max x b* **and**
max-le-monoL: (*a*::'*a*::*linorder*) ≤ *b* ⟹ *max a x* ≤ *max b x*
by (*fastforce simp*: *max.mono max-def*)+

end

3 Results for natural arithmetics with infinity

theory *Util-NatInf*
imports *HOL−Library.Extended-Nat*
begin

3.1 Arithmetic operations with *enat*

3.1.1 Additional definitions

instantiation *enat* :: *modulo*
begin

definition
div-enat-def [*code del*]:
a div b ≡ (*case a of*
(*enat x*) ⇒ (*case b of* (*enat y*) ⇒ *enat* (*x div y*) | ∞ ⇒ *0*) |
∞ ⇒ (*case b of* (*enat y*) ⇒ ((*case y of 0* ⇒ *0* | *Suc n* ⇒ ∞)) | ∞ ⇒ ∞))
definition
mod-enat-def [*code del*]:
a mod b ≡ (*case a of*
(*enat x*) ⇒ (*case b of* (*enat y*) ⇒ *enat* (*x mod y*) | ∞ ⇒ *a*) |
∞ ⇒ ∞)

instance ..

end

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

lemmas *ineq0-conv-enat*[*simp*] = *i0-less*[*symmetric*, *unfolded zero-enat-def*]

lemmas *iless-eSuc0-enat*[*simp*] = *iless-eSuc0*[*unfolded zero-enat-def*]

3.1.2 Addition, difference, order

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*::*enat*)) =
(*if y* < *x then x* = *y* + *z else if x* ≠ ∞ *then z* = *0 else z* = ∞)
by (*case-tac x*, *case-tac y*, *case-tac z*, *auto*, *case-tac z*, *auto*)
lemmas *idiff-eq-conv-enat* = *idiff-eq-conv*[*unfolded zero-enat-def*]

lemma *less-eq-idiff-eq-sum*: *y* ≤ (*x*::*enat*) ⟹ (*z* ≤ *x* − *y*) = (*z* + *y* ≤ *x*)
by (*case-tac x*, *case-tac y*, *case-tac z*, *fastforce*+)

lemma *eSuc-pred*: *0* < *n* ⟹ *eSuc* (*n* − *eSuc 0*) = *n*
apply (*case-tac n*)

apply (*simp add*: *eSuc-enat*)+
done
lemmas *eSuc-pred-enat* = *eSuc-pred*[*unfolded zero-enat-def*]
lemmas *iadd-0-enat*[*simp*] = *add-0-left*[**where** *'a* = *enat*, *unfolded zero-enat-def*]
lemmas *iadd-0-right-enat*[*simp*] = *add-0-right*[**where** *'a*=*enat*, *unfolded zero-enat-def*]

lemma *ile-add1*: (*n*::*enat*) ≤ *n* + *m*
by (*case-tac m*, *case-tac n*, *simp-all*)
lemma *ile-add2*: (*n*::*enat*) ≤ *m* + *n*
by (*simp only*: *add.commute*[*of m*] *ile-add1*)

lemma *iadd-iless-mono*: ⟦ (*i*::*enat*) < *j*; *k* < *l* ⟧ ⟹ *i* + *k* < *j* + *l*
by (*case-tac i*, *case-tac k*, *case-tac j*, *case-tac l*, *simp-all*)

lemma *trans-ile-iadd1*: *i* ≤ (*j*::*enat*) ⟹ *i* ≤ *j* + *m*
by (*rule order-trans*[*OF - ile-add1*])
lemma *trans-ile-iadd2*: *i* ≤ (*j*::*enat*) ⟹ *i* ≤ *m* + *j*
by (*rule order-trans*[*OF - ile-add2*])

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

lemma *iadd-ileD1*: *m* + *k* ≤ (*n*::*enat*) ⟹ *m* ≤ *n*
by (*case-tac m*, *case-tac n*, *case-tac k*, *simp-all*)

lemma *iadd-ileD2*: *m* + *k* ≤ (*n*::*enat*) ⟹ *k* ≤ *n*
by (*rule iadd-ileD1*, *simp only*: *add.commute*[*of m*])

lemma *idiff-ile-mono*: *m* ≤ (*n*::*enat*) ⟹ *m* − *l* ≤ *n* − *l*
by (*case-tac m*, *case-tac n*, *case-tac l*, *simp-all*)

lemma *idiff-ile-mono2*: *m* ≤ (*n*::*enat*) ⟹ *l* − *n* ≤ *l* − *m*
by (*case-tac m*, *case-tac n*, *case-tac l*, *simp-all*, *case-tac l*, *simp-all*)

lemma *idiff-iless-mono*: ⟦ *m* < (*n*::*enat*); *l* ≤ *m* ⟧ ⟹ *m* − *l* < *n* − *l*
by (*case-tac m*, *case-tac n*, *case-tac l*, *simp-all*, *case-tac l*, *simp-all*)

lemma *idiff-iless-mono2*: ⟦ *m* < (*n*::*enat*); *m* < *l* ⟧ ⟹ *l* − *n* ≤ *l* − *m*
by (*case-tac m*, *case-tac n*, *case-tac l*, *simp-all*, *case-tac l*, *simp-all*)

3.1.3 Multiplication and division

lemmas *imult-infinity-enat*[*simp*] = *imult-infinity*[*unfolded zero-enat-def*]
lemmas *imult-infinity-right-enat*[*simp*] = *imult-infinity-right*[*unfolded zero-enat-def*]

lemma *idiv-enat-enat*[*simp*, *code*]: *enat a div enat b* = *enat* (*a div b*)

unfolding *div-enat-def* **by** *simp*

lemma *idiv-infinity*: $0 < n \Longrightarrow (\infty::enat)\ div\ n = \infty$
unfolding *div-enat-def*
apply (*case-tac n*, *simp-all*)
apply (*rename-tac n1*, *case-tac n1*, *simp-all*)
done

lemmas *idiv-infinity-enat*[*simp*] = *idiv-infinity*[*unfolded zero-enat-def*]

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

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

lemmas *idiv-infinity-if-enat* = *idiv-infinity-if*[*unfolded zero-enat-def*]

lemmas *imult-0-enat*[*simp*] = *mult-zero-left*[**where** *'a=enat*,*unfolded zero-enat-def*]
lemmas *imult-0-right-enat*[*simp*] = *mult-zero-right*[**where** *'a=enat*,*unfolded zero-enat-def*]

lemmas *imult-is-0-enat* = *imult-is-0*[*unfolded zero-enat-def*]
lemmas *enat-0-less-mult-iff-enat* = *enat-0-less-mult-iff*[*unfolded zero-enat-def*]

lemma *imult-infinity-if*: $\infty * n = (if\ n = 0\ then\ 0\ else\ \infty::enat)$
by (*case-tac n*, *simp-all*)
lemma *imult-infinity-right-if*: $n * \infty = (if\ n = 0\ then\ 0\ else\ \infty::enat)$
by (*case-tac n*, *simp-all*)
lemmas *imult-infinity-if-enat* = *imult-infinity-if*[*unfolded zero-enat-def*]
lemmas *imult-infinity-right-if-enat* = *imult-infinity-right-if*[*unfolded zero-enat-def*]

lemmas *imult-is-infinity-enat* = *imult-is-infinity*[*unfolded zero-enat-def*]

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

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

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

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

lemma *imod-enat-enat*[*simp*, *code*]: *enat a mod enat b = enat* (*a mod b*)
unfolding *mod-enat-def* **by** *simp*
lemma *imod-infinity*[*simp*, *code*]: ∞ *mod n* = (∞::*enat*)
unfolding *mod-enat-def* **by** *simp*
lemma *imod-infinity-right*[*simp*, *code*]: *n mod* (∞::*enat*) = *n*
unfolding *mod-enat-def* **by** (*case-tac n*) *simp-all*

lemma *idiv-self*: ⟦ *0* < (*n*::*enat*); *n* ≠ ∞ ⟧ ⟹ *n div n* = *1*
by (*case-tac n*, *simp-all add*: *one-enat-def*)
lemma *imod-self*: *n* ≠ ∞ ⟹ (*n*::*enat*) *mod n* = *0*
by (*case-tac n*, *simp-all*)

lemma *idiv-iless*: *m* < (*n*::*enat*) ⟹ *m div n* = *0*
by (*case-tac m*, *simp-all*) (*case-tac n*, *simp-all*)
lemma *imod-iless*: *m* < (*n*::*enat*) ⟹ *m mod n* = *m*
by (*case-tac m*, *simp-all*) (*case-tac n*, *simp-all*)

lemma *imod-iless-divisor*: ⟦ *0* < (*n*::*enat*); *m* ≠ ∞ ⟧ ⟹ *m mod n* < *n*
by (*case-tac m*, *simp-all*) (*case-tac n*, *simp-all*)
lemma *imod-ile-dividend*: (*m*::*enat*) *mod n* ≤ *m*
by (*case-tac m*, *simp-all*) (*case-tac n*, *simp-all*)
lemma *idiv-ile-dividend*: (*m*::*enat*) *div n* ≤ *m*
by (*case-tac m*, *simp-all*) (*case-tac n*, *simp-all*)

lemma *idiv-imult2-eq*: (*a*::*enat*) *div* (*b* * *c*) = *a div b div c*
apply (*case-tac a*, *case-tac b*, *case-tac c*, *simp-all add*: *div-mult2-eq*)
apply (*simp add*: *imult-infinity-if*)
apply (*case-tac b* = *0*, *simp*)
apply (*case-tac c* = *0*, *simp*)
apply (*simp add*: *idiv-infinity*[*OF enat-0-less-mult-iff*[*THEN iffD2*]])
done

lemma *imult-ile-mono*: ⟦ (*i*::*enat*) ≤ *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*::*enat*) ≤ *j* ⟹ *i* * *k* ≤ *j* * *k*
by (*rule imult-ile-mono*[*OF - order-refl*])

lemma *imult-ile-mono2*: (*i*::*enat*) ≤ *j* ⟹ *k* * *i* ≤ *k* * *j*
by (*rule imult-ile-mono*[*OF order-refl*])

lemma *imult-iless-mono1*: ⟦ (*i*::*enat*) < *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*::*enat*) < *j*; *0* < *k*; *k* ≠ ∞ ⟧ ⟹ *k* * *i* ≤ *k* * *j*
by (*simp only*: *mult.commute*[*of k*], *rule imult-iless-mono1*)

lemma *imod-1*: (*enat m*) *mod eSuc 0 = 0*
by (*simp add*: *eSuc-enat*)
lemmas *imod-1-enat*[*simp*, *code*] = *imod-1*[*unfolded zero-enat-def*]

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

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

lemma *idiv-imod-equality*: (*m*::*enat*) *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*::*enat*) *div n * n + m mod n = m*
by (*insert idiv-imod-equality*[*of m n 0*], *simp*)

lemma *idiv-ile-mono*: *m* ≤ (*n*::*enat*) ⟹ *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-infinity*)
apply (*simp add*: *i0-lb*[*unfolded zero-enat-def*])
done
lemma *idiv-ile-mono2*: ⟦ *0 < m*; *m* ≤ (*n*::*enat*) ⟧ ⟹ *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 Results for natural arithmetics

theory *Util-Nat*
imports *Main*
begin

4.1 Some convenience arithmetic lemmata

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*

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 *fastforce*

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

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

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

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

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*

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*

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*

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

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*]

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

lemma *add-lessD2*: $i + j < (k::nat) \Longrightarrow j < k$ **by** *simp*

lemma *add-le-mono2*: $i \leq (j::nat) \Longrightarrow k + i \leq k + j$ **by** *simp*

lemma *add-less-mono2*: $i < (j::nat) \Longrightarrow k + i < k + j$ **by** *simp*

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*: $[\![0 \leq (a::'a::ordered\text{-}ab\text{-}group\text{-}add\text{-}abs);\ a \leq b]\!] \Longrightarrow |a| \leq |b|$
by *simp*
lemma *leq-neg-imp-abs-geq*: $[\![(a::'a::ordered\text{-}ab\text{-}group\text{-}add\text{-}abs) \leq 0;\ b \leq a]\!] \Longrightarrow |a| \leq |b|$
by *simp*
lemma *abs-range*: $[\![0 \leq (a::'a::\{ordered\text{-}ab\text{-}group\text{-}add\text{-}abs,abs\text{-}if\});\ -a \leq x;\ x \leq a]\!] \Longrightarrow |x| \leq a$
apply (*clarsimp simp*: *abs-if*)
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 (*fastforce simp add*: *sgn-if abs-if*)

lemma *abs-imp-sgn-abs*: $|a| = |b| \Longrightarrow |sgn\ (a::'a::linordered\text{-}idom)| = |sgn\ b|$
by (*fastforce simp add*: *abs-if*)
lemma *sgn-mono*: $a \leq b \Longrightarrow sgn\ (a::'a::\{linordered\text{-}idom, linordered\text{-}semidom\}) \leq sgn\ b$
by (*auto simp add*: *sgn-if*)

4.2 Additional facts about inequalities

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

lemma *less-add-diff*: $k < (n::nat) \Longrightarrow m < n + m - k$

by (*rule add-less-imp-less-right*[*of* - k], *simp*)

lemma *add-diff-less*: $[\![\ k < n;\ 0 < m\ ]\!] \Longrightarrow m + k - n < (m::nat)$
by (*case-tac* $m + k < n$, *simp-all*)

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*

lemma *diff-less-imp-less-add*: $j - (k::nat) < i \Longrightarrow j < i + k$ **by** *simp*

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*

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

4.3 Inequalities for Suc and pred

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*

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*

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

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*

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*

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

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
le-add-diff
add-diff-le
less-add-diff
add-diff-less

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

end

5 Results for division and modulo operators on integers

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

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

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

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

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

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

lemmas *div-mult-cancel* = *minus-mod-eq-div-mult* [*symmetric*]

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 minus-mod-eq-mult-div* [*symmetric*])
done

lemma *div-mult-le*: $(n::nat)\ div\ m * m \leq n$
by (*simp add*: *mult.commute minus-mod-eq-mult-div* [*symmetric*])
lemma *less-div-Suc-mult*: $0 < (m::nat) \Longrightarrow n < Suc\ (n\ div\ m) * m$
apply (*simp add*: *mult.commute minus-mod-eq-mult-div* [*symmetric*])
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 *fastforce*

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

5.2 Additional results for addition and subtraction with *mod*

lemma *mod-Suc-conv*:
(*(Suc a) mod m = (Suc b) mod m*) = (*a mod m = b mod m*)
by (*simp add*: *mod-Suc*)

lemma *mod-Suc'*:
0 < n ⟹ 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 ≤ (*a*::*nat*) *⟹*
(*(a − k) mod m = b mod m*) = (*a mod m = (b + k) mod m*)
by (*simp add*: *mod-add*[**where** *m*=*m* **and** *a*=*a*−*k* **and** *b*=*b* **and** *k*=*k*, *symmetric*])

lemma *mod-sub-eq-mod-0-conv*:
a + b ≤ (*n*::*nat*) *⟹*
(*(n − a) mod m = b mod m*) = (*(n − (a + b)) mod m = 0*)
by (*insert mod-add*[*of n*−(*a*+*b*) *b m 0*], *simp*)
lemma *mod-sub-eq-mod-swap*:
⟦ a ≤ (*n*::*nat*); *b ≤ n ⟧ ⟹*
(*(n − a) mod m = b mod m*) = (*(n − b) mod m = a mod m*)
by (*simp add*: *mod-sub-add add.commute*)

lemma *le-mod-greater-imp-div-less*:
⟦ a ≤ (*b*::*nat*); *a mod m > b mod m ⟧ ⟹ a div m < b div m*
apply (*rule ccontr*, *simp add*: *linorder-not-less*)
apply (*drule mult-le-mono1*[*of b 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 ≥ b mod m ⟧ ⟹ a div m < b div m*
apply (*case-tac m = 0*, *simp*)
apply (*rule mult-less-cancel1*[*of m*, *THEN iffD1*, *THEN conjunct2*])
apply (*simp add*: *minus-mod-eq-mult-div* [*symmetric*])
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* ⟧ ⟹ *a div m* < *b div m*
by (*simp add*: *less-mod-ge-imp-div-less*)

lemma *mod-diff-right-eq*:
  (*a*::*nat*) ≤ *b* ⟹ (*b* − *a*) *mod m* = (*b* − *a mod m*) *mod m*
proof −
  assume *a-as*:*a* ≤ *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*
    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* ≤ (*t*::*nat*); *y* ≤ *t* ⟧ ⟹
  (*t* − *x*) *mod m* = (*t* − *y*) *mod m*
by (*simp only*: *mod-diff-right-eq*)
lemma *mod-eq-imp-diff-mod-eq2*:
  ⟦ *x mod m* = *y mod m*; (*t*::*nat*) ≤ *x*; *t* ≤ *y* ⟧ ⟹
  (*x* − *t*) *mod m* = (*y* − *t*) *mod m*
apply (*case-tac m* = *0*, *simp+*)
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

lemma *divisor-add-diff-mod-if*:
  (*m* + *b mod m* − *a mod m*) *mod* (*m*::*nat*)= (
  if a mod m ≤ *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*)
apply (*simp add*: *less-imp-diff-less*)
done
corollary *divisor-add-diff-mod-eq1*:
  a mod m ≤ *b mod m* ⟹
  (*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 ⟹
(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*)
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*)

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*[*symmetric*, *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*[*symmetric*, *of x a*])
apply (*rule iffI*)
apply (*drule sym*)
apply (*simp add*: *mod-add-mod-if*)
apply (*simp add*: *mod-add-left-eq 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*:$a \leq b$
and *m-as*: $0 < m$
have *a-mod-le-b-s*: $a \bmod m \leq b$
by (*rule le-trans*[*of* - *a*], *simp only*: *mod-le-dividend*, *simp only*: *a-as*)
have $(b - a) \bmod m = (b - a \bmod m) \bmod m$
using *a-as* **by** (*simp only*: *mod-diff-right-eq*)
also have $\ldots = (b - a \bmod m + m) \bmod m$
by *simp*
also have $\ldots = (b + m - a \bmod m) \bmod m$
using *a-mod-le-b-s* **by** *simp*
also have $\ldots = (b \operatorname{div} m * m + b \bmod m + m - a \bmod m) \bmod m$
by *simp*
also have $\ldots = (b \operatorname{div} m * m + (b \bmod m + m - a \bmod m)) \bmod m$
by (*simp add*: *diff-add-assoc*[*OF mod-le-divisor*, *OF m-as*])
also have $\ldots = ((b \bmod m + m - a \bmod m) + b \operatorname{div} m * m) \bmod m$
by *simp*
also have $\ldots = (b \bmod m + m - a \bmod m) \bmod m$
by *simp*
also have $\ldots = (m + b \bmod m - a \bmod m) \bmod m$
by (*simp only*: *add.commute*)
finally show *?thesis* **.**
qed
corollary *mod-diff1-eq-if*:
$(a::nat) \leq b \Longrightarrow (b - a) \bmod m = ($
if $a \bmod m \leq b \bmod m$ *then* $b \bmod m - a \bmod m$
else $m + b \bmod m - a \bmod m$)
by (*simp only*: *mod-diff1-eq divisor-add-diff-mod-if*)
corollary *mod-diff1-eq1*:
$[\![(a::nat) \leq b; a \bmod m \leq b \bmod m]\!]$
$\Longrightarrow (b - a) \bmod m = b \bmod m - a \bmod m$
by (*simp add*: *mod-diff1-eq-if*)
corollary *mod-diff1-eq2*:
$[\![(a::nat) \leq b; b \bmod m < a \bmod m]\!]$
$\Longrightarrow (b - a) \bmod m = m + b \bmod m - a \bmod m$
by (*simp add*: *mod-diff1-eq-if*)

5.2.1 Divisor subtraction with *div* and *mod*

lemma *mod-diff-self1*:
$0 < (n::nat) \Longrightarrow (m - n) \bmod m = m - n$
by (*case-tac* $m = 0$, *simp-all*)
lemma *mod-diff-self2*:
$m \leq (n::nat) \Longrightarrow (n - m) \bmod m = n \bmod m$
by (*simp add*: *mod-diff-right-eq*)
lemma *mod-diff-mult-self1*:
$k * m \leq (n::nat) \Longrightarrow (n - k * m) \bmod m = n \bmod m$
by (*simp add*: *mod-diff-right-eq*)
lemma *mod-diff-mult-self2*:
$m * k \leq (n::nat) \Longrightarrow (n - m * k) \bmod m = n \bmod m$

by (*simp only*: *mult.commute*[*of m k*] *mod-diff-mult-self1*)

lemma *div-diff-self1*: *0* < (*n*::*nat*) ⟹ (*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*)
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*
apply (*drule div-le-mono*[*OF less-imp-le*, *of n - m*])
apply *simp*
apply (*simp add*: *linorder-not-less*)
apply (*rule iffD1*[*OF mult-cancel1-gr0*[**where** *k*=*m*]], *assumption*)
apply (*subst diff-mult-distrib2*)
apply (*simp only*: *minus-mod-eq-mult-div* [*symmetric*])
apply (*simp only*: *diff-commute*[*of - k*∗*m*])
apply (*simp only*: *mult.commute*[*of m*])
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*
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*:
$(a::nat)\ mod\ m = b\ mod\ m \Longrightarrow m\ dvd\ b - a$
by (*rule dvd-eq-mod-eq-0*[*THEN iffD2*, *OF mod-eq-imp-diff-mod-0*])

lemma *mod-neq-imp-diff-mod-neq0*:
$[\![(a::nat)\ mod\ m \neq b\ mod\ m;\ a \leq b]\!] \Longrightarrow 0 < (b - a)\ mod\ m$
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*)
apply (*simp add*: *mod-diff1-eq1*)
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*:
$[\![(a::nat)\ mod\ m \neq b\ mod\ m;\ a \leq b]\!] \Longrightarrow \neg\ m\ dvd\ b - a$
by (*simp add*: *dvd-eq-mod-eq-0 mod-neq-imp-diff-mod-neq0*)

lemma *diff-mod-0-imp-mod-eq*:
$[\![(b - a)\ mod\ m = 0;\ a \leq b]\!] \Longrightarrow (a::nat)\ mod\ m = b\ mod\ m$
apply (*rule ccontr*)
apply (*drule mod-neq-imp-diff-mod-neq0*)
apply *simp-all*
done
corollary *diff-dvd-imp-mod-eq*:
$[\![m\ dvd\ b - a;\ a \leq b]\!] \Longrightarrow (a::nat)\ mod\ m = b\ mod\ m$
by (*rule dvd-eq-mod-eq-0*[*THEN iffD1*, *THEN diff-mod-0-imp-mod-eq*])

lemma *mod-eq-diff-mod-0-conv*:
$a \leq (b::nat) \Longrightarrow (a\ mod\ m = b\ mod\ m) = ((b - a)\ mod\ m = 0)$
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*:
$a \leq (b::nat) \Longrightarrow (a\ mod\ m = b\ mod\ m) = (m\ dvd\ b - a)$
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*:
$a\ mod\ m = b\ mod\ m \Longrightarrow (b - a)\ mod\ m = 0$ **for** $a\ b\ m :: int$

by (*simp add*: *mod-diff-cong*)

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*

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*: *right-diff-distrib*)
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*
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*
from *as2 a-r1 b-r2* **have** *s-r1*:*0* ≤ *r1* ∧ *r1* < *m* **and** *s-r2*:*0* ≤ *r2* ∧ *r2* < *m*
by *simp-all*
have *mr2r1*:−*m* < *r2* − *r1* **and** *r2r1m*:*r2* − *r1* < *m*
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*)
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
      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
      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 linorder-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: mod-minus-right neg-equal-iff-equal)
apply (simp only: zmod-zminus1-eq-if)
apply (split if-split)+
apply (safe, simp-all)
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 linorder-less-linear[of 0 m], elim disjE)
apply (blast dest: zmod-zminus-eq-conv-pos)
apply simp
apply (simp add: zmod-zminus-eq-conv-pos[of −m, symmetric])
done

lemma zdiff-mod-0-imp-mod-eq:
  (b − a) mod m = 0 ⟹ (a::int) mod m = b mod m
by (metis dvd-eq-mod-eq-0 mod-eq-dvd-iff)

lemma zmod-eq-diff-mod-0-conv:
  ((a::int) mod m = b mod m) = ((b − a) mod 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 ¬(∃(a::int) b m. (b − a) mod m = 0 ∧ a mod m ≠ b mod m)
by (simp add: zmod-eq-diff-mod-0-conv)
```

lemma $\exists (a::nat)\ b\ m.\ (b - a)\ mod\ m = 0 \wedge a\ mod\ m \neq b\ mod\ m$
apply (*rule-tac x=1* **in** *exI*)
apply (*rule-tac x=0* **in** *exI*)
apply (*rule-tac x=2* **in** *exI*)
apply *simp*
done

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 mult.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 ≤ c * a*)
prefer *2*
apply (*simp only*: *mult-right-mono*)
apply (*simp only*: *mult.commute*)
apply (*subgoal-tac a * b div c ≤ a * c div c*)
prefer *2*
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 ≤ c * a*)
prefer *2*
apply (*simp only*: *mult-right-mono-neg*)
apply (*simp only*: *mult.commute*)
apply (*subgoal-tac a * b div c ≤ a * c div c*)
prefer *2*
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*) ≤ *b*; *b* ≤ *b'*; *0* ≤ *a*; *0* < *c*⟧ ⟹
|*a* ∗ *b* *div* *c*| ≤ |*a* ∗ *b'* *div* *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*) ≤ *b*; *b* ≤ *b'*; *0* ≤ *a*; *0* < *c* ⟧ ⟹
|*a* ∗ *b* *div* *c* + *a*| ≤ |*a* ∗ *b'* *div* *c* + *a*|
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*)

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*
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*
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*
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* $m \leq n - n \bmod m$)
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 minus-mod-eq-mult-div* [*symmetric*])
apply (*drule less-imp-le-pred*[*of x*])
apply (*drule mult-le-mono2*[*of x - m*])
apply (*simp add*: *diff-mult-distrib2 minus-mod-eq-mult-div* [*symmetric*] *del*: *diff-diff-left*)
apply (*simp only*: *le-diff-conv2*[*of 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*:
$((n + k) \bmod (m::nat) = n \bmod m) = (k \bmod m = 0)$
by (*metis add-eq-if mod-add mod-add-self1 mod-self add.commute*)

lemma *between-imp-mod-between*:
$[\![\ b < (m::nat);\ m * k + a \leq n;\ n \leq m * k + b\ ]\!] \Longrightarrow$
$a \leq n \bmod m \wedge n \bmod m \leq b$
apply (*case-tac* $m = 0$, *simp-all*)
apply (*frule gr-implies-gr0*)
apply (*subgoal-tac* $k = n \ div\ m$)
prefer *2*
apply (*rule sym*, *rule div-nat-eqI*) **apply** *simp*
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*:
$[\![\ b < (m::nat);\ m * k \leq n;\ n \leq m * k + b\ ]\!] \Longrightarrow n \bmod m \leq b$
by (*insert between-imp-mod-between*[*of b m k 0 n*], *simp*)
corollary *between-imp-mod-gr0*:
$[\![\ (m::nat) * k < n;\ n < m * k + m\ ]\!] \Longrightarrow 0 < n \bmod m$
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

corollary *le-less-div-conv*:
$0 < m \Longrightarrow (k * m \leq n \wedge n < Suc\ k * m) = (n\ div\ m = k)$
by (*auto simp add*: *ac-simps intro*: *div-nat-eqI dividend-less-times-div*)

lemma *le-less-imp-div*:
⟦ $k * m \leq n$; $n < Suc\ k * m$ ⟧ ⟹ $n\ div\ m = k$
by (*auto simp add*: *ac-simps intro*: *div-nat-eqI*)

lemma *div-imp-le-less*:
⟦ $n\ div\ m = k$; $0 < m$ ⟧ ⟹ $k * m \leq n \wedge n < Suc\ k * m$
by (*auto simp add*: *ac-simps intro*: *dividend-less-times-div*)

lemma *div-le-mod-le-imp-le*:
⟦ $(a::nat)\ div\ m \leq b\ div\ m$; $a\ mod\ m \leq b\ mod\ m$ ⟧ ⟹ $a \leq b$
apply (*rule subst*[*OF mult-div-mod-eq*[*of m a*]])
apply (*rule subst*[*OF mult-div-mod-eq*[*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*:
⟦ $a \leq b$; $(a + k)\ mod\ m = (b::nat)\ mod\ m$ ⟧ ⟹ $a + k\ mod\ m \leq b$
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*:
⟦ $(m::nat) * k \leq n$; $r \leq n\ mod\ m$ ⟧ ⟹ $m * k + r \leq n$
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*:
$n\ mod\ m = (0::nat) \Longrightarrow n * k\ mod\ m = 0$
by *fastforce*
lemma *mod-0-imp-mod-mult-left-0*:
$n\ mod\ m = (0::nat) \Longrightarrow k * n\ mod\ m = 0$
by *fastforce*

lemma *mod-0-imp-div-mult-left-eq*:
$n\ mod\ m = (0::nat) \Longrightarrow k * n\ div\ m = k * (n\ div\ m)$
by *fastforce*
lemma *mod-0-imp-div-mult-right-eq*:
$n\ mod\ m = (0::nat) \Longrightarrow n * k\ div\ m = k * (n\ div\ m)$
by *fastforce*

lemma *mod-0-imp-mod-factor-0-left*:
$n\ mod\ (m * m') = (0::nat) \Longrightarrow n\ mod\ m = 0$
by *fastforce*
lemma *mod-0-imp-mod-factor-0-right*:

*n mod (m * m') = (0::nat) ⟹ n mod m' = 0*
by *fastforce*

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*
apply (*simp only*: *mod-mult2-eq*)
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
using *mod-mod-cancel* [*of m m' a*] *mod-mod-cancel* [*of m m' b*] **by** *auto*

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*
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*: *ac-simps 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*
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*
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*)
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*: *ac-simps*)
hence *?L * k = y mod m * k*
by (*simp only*: *mult-mod-left*)
thus *?thesis*
using *as2* **by** *simp*
qed

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

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*])
apply (*rule arg-cong*[*of* (*a mod m + b mod m*) *div m*])
apply (*clarsimp simp*: *linorder-not-less*)
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 (*fastforce 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 if-split*, *intro conjI impI*)
apply (*rule-tac t=Suc m* **and** *s=m + 1* **in** *subst*, *simp*)
apply (*subst div-add1-eq2*, *simp+*)
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 ≤ b div m*
using *le-as* **by** (*simp only*: *div-le-mono*)
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 div m * m − a mod m*
by *simp*
also have *... = b div m * m − a div m * m + b mod m − a mod m*
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*)
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 ≤ b mod m*)
case *True*

```
  hence as': a mod m ≤ b mod m .

  have (b − a) div m = ?b-a1 div m
    using b-a-div-s .
  also have ... = ((b div m − a div m) * m + (b mod m − a mod m)) div m
    using as' by simp
  also have ... = b div m − a div m + (b mod m − a mod m) div m
    apply (simp only: add.commute)
    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 = ... .
  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': ¬ a mod m ≤ 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'
    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
    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
    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
     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
     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

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)
apply (rule subst[of
  if a mod m ≤ 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 if-split, rule conjI)
 apply simp
apply (clarsimp simp: linorder-not-le)
apply (rule sym)
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 ≤ b mod m ⟹
  (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 ⟹
  (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 ⟹
  (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*) ≤ *y*; *y* − *x* < *m* ⟧ ⟹ *x* = *y* − (*y* − *x*) *mod m*
by *simp*
lemma *diff-ge-divisor-imp-sub-mod-less*:
⟦ (*x*::*nat*) ≤ *y*; *m* ≤ *y* − *x*; 0 < *m* ⟧ ⟹ *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

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

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*)
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*)
apply (*simp only*: *Suc-le-eq*[*symmetric*])
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 if-split*, *rule conjI*)
apply (*simp add*: *mod-0-imp-mod-pred*)
apply *clarsimp*
apply (*case-tac m* = *Suc* 0, *simp*)
apply (*frule subst*[*OF Suc0-mod*[*symmetric*], **where** *P*=λ*x*. *x* ≤ *n mod m*], *simp*)

apply (*simp only*: *mod-diff1-eq1*)
apply (*simp add*: *Suc0-mod*)
done
corollary *mod-pred-Suc-mod*:
  $0 < n \Longrightarrow Suc\ ((n - Suc\ 0)\ mod\ m)\ mod\ m = n\ mod\ m$
apply (*case-tac* $m = 0$, *simp*)
apply (*simp add*: *mod-pred*)
done
corollary *diff-mod-pred*:
  $a < b \Longrightarrow$
  $(b - Suc\ a)\ mod\ m =$ (
    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*:
  $a < b \Longrightarrow Suc\ ((b - Suc\ a)\ mod\ m)\ mod\ m = (b - a)\ mod\ m$
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*:
  $[\![\ a < b;\ 0 < m;\ a\ mod\ m = b\ mod\ m\ ]\!] \Longrightarrow$
  $Suc\ ((b - Suc\ a)\ mod\ m) = m$
apply (*drule mod-eq-imp-diff-mod-0*[*of a*])
apply (*frule iffD2*[*OF zero-less-diff*])
apply (*drule mod-0-imp-mod-pred*[*of b−a m*], *assumption*)
apply *simp*
done

lemma *sub-diff-mod-eq*:
  $r \leq t \Longrightarrow (t - (t - r)\ mod\ m)\ mod\ (m{::}nat) = r\ mod\ m$
by (*metis mod-diff-right-eq diff-diff-cancel diff-le-self*)

lemma *sub-diff-mod-eq′*:
  $r \leq t \Longrightarrow (k * m + t - (t - r)\ mod\ m)\ mod\ (m{::}nat) = r\ mod\ m$
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 ⟹ (x mod k = k − Suc 0) = (Suc x mod k = 0)*
by (*simp only*: *mod-Suc*, *split if-split*, *fastforce*)

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*)
apply (*simp add*: *minus-mod-eq-mult-div*)
done

lemma *mod-less-imp-diff-div-conv*:
⟦ n mod m < r; r ≤ m + n mod m⟧ ⟹ (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*)
apply (*rule div-nat-eqI*)
apply (*simp-all add*: *algebra-simps minus-mod-eq-mult-div* [*symmetric*])
done

corollary *mod-0-le-imp-diff-div-conv*:
⟦ n mod m = 0; 0 < r; r ≤ m ⟧ ⟹ (n − r) div m = n div m − Suc 0
by (*simp add*: *mod-less-imp-diff-div-conv*)
corollary *mod-0-less-imp-diff-Suc-div-conv*:
⟦ n mod m = 0; r < m ⟧ ⟹ (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 ⟹ (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 ⟹ (n − Suc 0) div m = n div m − Suc 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

lemma *div-le-conv*: *0 < m ⟹ n div m ≤ k = (n ≤ Suc k * m − Suc 0)*
apply (*rule iffI*)
apply (*drule mult-le-mono1*[*of - - m*])
apply (*simp only*: *mult.commute*[*of - m*] *minus-mod-eq-mult-div* [*symmetric*])

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*])
apply (*simp only*: *diff-add-assoc*[*OF Suc-leI*])
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*])
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*])
apply (*simp add*: *mod-0-imp-sub-1-div-conv*)
done

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

lemma *div-less-conv*: $0 < (m::nat) \Longrightarrow (n \ div \ m < k) = (n < k * m)$
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*: $(n \ div \ (m::nat) = 0) = (m = 0 \vee n < m)$
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 *fastforce*
done

lemma *div-eq-0-conv'*: $0 < m \Longrightarrow (n\ div\ (m::nat) = 0) = (n < m)$
by (*simp add*: *div-eq-0-conv*)
corollary *div-gr-imp-gr-divisor*: $x < n\ div\ (m::nat) \Longrightarrow m \leq n$
apply (*drule gr-implies-gr0*, *drule neq0-conv*[*THEN iffD2*])
apply (*simp add*: *div-eq-0-conv*)
done

lemma *mod-0-less-div-conv*:
$n\ mod\ (m::nat) = 0 \Longrightarrow (k * m < n) = (k < n\ div\ m)$
apply (*case-tac* $m = 0$, *simp*)
apply *fastforce*
done

lemma *add-le-divisor-imp-le-Suc-div*:
$[\![\ x\ div\ m \leq n;\ y \leq m\ ]\!] \Longrightarrow (x + y)\ div\ m \leq Suc\ n$
apply (*case-tac* $m = 0$, *simp*)
apply (*simp only*: *div-add1-eq-if*[*of* - *x*])
apply (*drule order-le-less*[*of y*, *THEN iffD1*], *fastforce*)
done

List of definitions and lemmas

thm
minus-mod-eq-mult-div [*symmetric*]
mod-0-div-mult-cancel
div-mult-le
less-div-Suc-mult
thm
Suc0-mod
Suc0-mod-subst
Suc0-mod-cong

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

thm
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
div-diff1-eq-if

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

thm
div-le-conv

end

6 Sets of natural numbers

theory *SetInterval2*
imports
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

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* ⟹ *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 (*fastforce 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 (*fastforce 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 *fastforce*
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*)

lemma *inj-on-image-mem-iff*:
$[\![$ *inj-on f A*; *B* $\subseteq$ *A*; *a* $\in$ *A* $]\!] \Longrightarrow$ (*f a* $\in$ *f* ‘ *B*) = (*a* $\in$ *B*)
unfolding *inj-on-def* **by** *blast*

corollary *inj-on-union-image-Int*:
inj-on f (*A* $\cup$ *B*) $\Longrightarrow$ *f* ‘ (*A* $\cap$ *B*) = *f* ‘ *A* $\cap$ *f* ‘ *B*
by (*rule inj-on-image-Int*[*OF - Un-upper1 Un-upper2*])

6.1.3 Surjectivity

definition *surj-on* :: $('a \Rightarrow 'b) \Rightarrow 'a\ set \Rightarrow 'b\ set \Rightarrow bool$
  where *surj-on f A B* $\equiv \forall\, b{\in}B.\ \exists\, a{\in}A.\ b = f\ a$

lemma *surj-on-conv*: $(surj\text{-}on\ f\ A\ B) = (\forall\, b{\in}B.\ \exists\, a{\in}A.\ b = f\ a)$
unfolding *surj-on-def* **..**

lemma *surj-on-image-conv*: $(surj\text{-}on\ f\ A\ B) = (B \subseteq f\ `\ A)$
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: $[\![\ \forall\, b \in B.\ \exists\, a{\in}A.\ b = f\ a\ ]\!] \Longrightarrow surj\text{-}on\ f\ A\ B$ **and**
  surj-onD2: $surj\text{-}on\ f\ A\ B \Longrightarrow \forall\, b \in B.\ \exists\, a{\in}A.\ b = f\ a$ **and**
  surj-onD: $[\![\ surj\text{-}on\ f\ A\ B;\ b \in B\ ]\!] \Longrightarrow \exists\, a{\in}A.\ b = f\ a$
unfolding *surj-on-conv*
by *blast+*

lemma *comp-surj-on*:
  $[\![\ surj\text{-}on\ f\ A\ B;\ surj\text{-}on\ g\ B\ C\ ]\!] \Longrightarrow surj\text{-}on\ (g \circ f)\ A\ C$
unfolding *comp-def surj-on-image-conv* **by** *blast*

lemma *surj-on-Un-right*: $surj\text{-}on\ f\ A\ (B1 \cup B2) = (surj\text{-}on\ f\ A\ B1 \wedge surj\text{-}on\ f\ A\ B2)$
unfolding *surj-on-image-conv*
by *blast*

lemma *surj-on-Un-left*:
  $surj\text{-}on\ f\ (A1 \cup A2)\ B =$
  $(\exists\, B1.\ \exists\, B2.\ B \subseteq B1 \cup B2 \wedge surj\text{-}on\ f\ A1\ B1 \wedge surj\text{-}on\ f\ A2\ B2)$
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*: $surj\text{-}on\ f\ A\ B \Longrightarrow surj\text{-}on\ f\ A\ (B - B')$
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*: $surj\text{-}on\ f\ \{\}\ B = (B = \{\})$
unfolding *surj-on-conv* **by** *blast*

lemma *surj-on-imageI*: *surj-on* $(g \circ f)$ A $B \Longrightarrow$ *surj-on* g $(f$ ‘ $A)$ B
unfolding *surj-on-conv* **by** *fastforce*

lemma *surj-on-insert-right*: *surj-on* f A $(insert$ b $B) = (surj\text{-}on$ f A $B \wedge surj\text{-}on$ f A $\{b\})$
unfolding *surj-on-conv* **by** *blast*

lemma *surj-on-insert-left*: *surj-on* f $(insert$ a $A)$ $B = (surj\text{-}on$ f A $(B - \{f$ $a\}))$
unfolding *surj-on-conv* **by** *blast*

lemma *surj-on-subset-right*: $[\![$ *surj-on* f A B; $B' \subseteq B$ $]\!] \Longrightarrow$ *surj-on* f A B'
unfolding *surj-on-conv* **by** *blast*

lemma *surj-on-subset-left*: $[\![$ *surj-on* f A B; $A \subseteq A'$ $]\!] \Longrightarrow$ *surj-on* f A' B
unfolding *surj-on-conv* **by** *blast*

lemma *bij-betw-imp-surj-on*: *bij-betw* f A $B \Longrightarrow$ *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 $B = (inj\text{-}on$ f $A \wedge surj\text{-}on$ f A $B \wedge f$ ‘ $A \subseteq B)$
unfolding *bij-betw-def surj-on-image-conv* **by** *blast*

6.1.4 Induction over natural sets

lemma *image-nat-induct*:
$[\![$ P $(f$ $0)$; $\bigwedge n.$ P $(f$ $n) \Longrightarrow P$ $(f$ $(Suc$ $n))$; *surj-on* f *UNIV* I; $a \in I$ $]\!] \Longrightarrow P$ a
proof −
assume *as-P0*: P $(f$ $0)$
and *as-IA*: $\bigwedge n.$ P $(f$ $n) \Longrightarrow P$ $(f$ $(Suc$ $n))$
and *as-surj-f*: *surj-on* f *UNIV* I
and *as-a*: $a \in I$
have *P-n*:$\bigwedge n.$ P $(f$ $n)$
apply $(induct\text{-}tac$ $n)$
apply $(simp$ *only*: *as-P0*$)$
apply $(simp$ *only*: *as-IA*$)$
done
have $\forall x{\in}I.$ $\exists n.$ $x = f$ n
using *as-surj-f*
by $(unfold$ *surj-on-conv*, *blast*$)$
hence $\exists n.$ $a = f$ n
using *as-a* **by** *blast*
thus P a
using *P-n* **by** *blast*
qed

lemma *nat-induct'*[*rule-format*]:
$[\![$ P *n0*; $\bigwedge n.$ $[\![$ *n0* $\leq n$; P n $]\!] \Longrightarrow P$ $(Suc$ $n)$; *n0* $\leq n$ $]\!] \Longrightarrow P$ n
by $(insert$ *nat-induct*[**where** $n{=}n{-}$*n0* **and** $P{=}\lambda n.$ P $(n0{+}n)]$, *simp*$)$

lemma *enat-induct*:
  $[\![P\ 0;\ P\ \infty;\ \bigwedge n.\ P\ n \Longrightarrow P\ (eSuc\ n)]\!] \Longrightarrow P\ n$
apply (*case-tac n*)
 prefer *2*
 apply *simp*
apply (*simp only*: *enat-defs*)
apply (*rename-tac n1*)
apply (*induct-tac n1*)
apply (*simp add*: *enat.splits*)+
done

lemma *eSuc-imp-Suc-aux-0*:
  $[\![\bigwedge n.\ P\ n \Longrightarrow P\ (eSuc\ n);\ n0' \leq n';\ P\ (enat\ n')]\!] \Longrightarrow P\ (enat\ (Suc\ n'))$
by (*simp only*: *enat-defs enat.splits*)

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

lemma *enat-induct'*:
  $[\![P\ (n0::enat);\ P\ \infty;\ \bigwedge n.\ [\![n0 \leq n;\ \ P\ n]\!] \Longrightarrow P\ (eSuc\ n);\ n0 \leq n]\!] \Longrightarrow P\ n$
apply (*case-tac n*)
 prefer *2* **apply** *simp*
apply (*case-tac n0*)
 prefer *2* **apply** *simp*
apply (*rename-tac n' n0'*, *simp*)
apply (*rule-tac ?n0.0=n0'* **and** *n=n'* **and** $P{=}\lambda n.\ P\ (enat\ n)$ **in** *nat-induct'*)
  apply *simp*
 apply (*simp add*: *eSuc-enat*[*symmetric*])
apply *simp*
done

lemma *wf-less-interval*:
  wf { $(x,y).\ x \in (I::nat\ set) \wedge y \in I \wedge x < y$ }
apply (*rule wf-subset*[**where**
  $p{=}\{ (x,y).\ x \in I \wedge y \in I \wedge x < y \}$ **and**
  $r{=}\{(x,y).\ x < y\}$])

apply (*rule wf-less*)
apply *blast*
done

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*)
apply (*rule-tac r*={ (*x*,*y*). *x* ∈ *I* ∧ *y* ∈ *I* ∧ *x* < *y* } **in** *wf-induct*)
apply (*rule wf-less-interval*)
apply *simp*
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*)

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*)

lemma *mult-left-inj*: *0* < *n* ⟹ *inj* (λ*x*. (*n*::*nat*) ∗ *x*)
by (*simp add*: *inj-on-def*)

lemma *mult-right-inj*: *0* < *n* ⟹ *inj* (λ*x*. *x* ∗ (*n*::*nat*))
by (*simp add*: *inj-on-def*)

lemma *sub-left-inj-on*: *inj-on* (λ*x*. (*x*::*nat*) − *k*) {*k*..}
by (*rule inj-onI*, *simp*)

lemma *sub-right-inj-on*: *inj-on* (λ*x*. *k* − (*x*::*nat*)) {..*k*}
by (*rule inj-onI*, *simp*)

lemma *add-left-strict-mono*: *strict-mono* (λ*x*. *n* + (*x*::*'a*::*ordered-cancel-ab-semigroup-add*))
by (*unfold strict-mono-def*, *clarify*, *rule add-strict-left-mono*)

lemma *add-right-strict-mono*: *strict-mono* (λ*x*. *x* + (*n*::*'a*::*ordered-cancel-ab-semigroup-add*))
by (*unfold strict-mono-def*, *clarify*, *rule add-strict-right-mono*)

lemma *mult-left-strict-mono*: *0* < *n* ⟹ *strict-mono* (λ*x*. *n* ∗ (*x*::*nat*))
by (*unfold strict-mono-def*, *clarify*, *simp*)

lemma *mult-right-strict-mono*: *0* < *n* ⟹ *strict-mono* (λ*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 (*fastforce 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*)
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-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-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.

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

6.2.1 Basic results, as for *Least*

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

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*)

lemma *iMinI2*: $[\![\ k \in I;\ \bigwedge x.\ x \in I \Longrightarrow P\ x\ ]\!] \Longrightarrow P\ (iMin\ I)$
by (*blast intro*: *iMinI*)

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*)

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*])

lemma *iMin-mono*:
$[\![\ mono\ f;\ \exists\, x.\ x \in I\ ]\!] \Longrightarrow iMin\ (f\ `\ I) = f\ (iMin\ I)$
apply (*unfold iMin-def*)
apply (*rule Least-mono*[*of f I*], *simp*)
apply (*rule-tac x*=*iMin I* **in** *bexI*)
apply (*simp add*: *iMin-le*)
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*)

lemma *not-less-iMin*: $k < iMin\ I \Longrightarrow k \notin I$
unfolding *iMin-def*
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*: $[\![\ k \in I;\ P\ (k::('a::wellorder))\ ]\!] \Longrightarrow \exists\, x{\in}I.\ P\ x \wedge (\forall\, y{\in}I.\ y < x \longrightarrow \neg\ P\ y)$
apply (*rule-tac x*=*iMin* $\{y \in I.\ P\ y\}$ **in** *bexI*)
prefer *2*
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*)

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

lemma *iMin-equality*:
$[\![\ k \in I;\ \bigwedge x.\ x \in I \Longrightarrow k \leq x\ ]\!] \Longrightarrow iMin\ I = k$
unfolding *iMin-def*
by (*blast intro*: *Least-equality*)

lemma *iMin-mono-on*:
$[\![\ mono\text{-}on\ f\ I;\ \exists\, x.\ x \in I\ ]\!] \Longrightarrow iMin\ (f\ `\ I) = f\ (iMin\ I)$
apply (*unfold mono-on-def*)
apply (*rule iMin-equality*)
apply (*blast intro*: *iMinI-ex*)+
done

lemma *iMin-mono-on2*:
$[\![\ mono\text{-}on\ f\ I;\ I \neq \{\}\ ]\!] \Longrightarrow iMin\ (f\ `\ I) = f\ (iMin\ I)$
by (*rule iMin-mono-on*, *blast*+)

lemma *iMinI2-order*:
$[\![\ k \in I;\ \bigwedge y.\ y \in I \Longrightarrow k \leq y;$
$\bigwedge x.\ [\![\ x \in I;\ \forall\, y{\in}I.\ x \leq y\ ]\!] \Longrightarrow P\ x\ ]\!] \Longrightarrow$
$P\ (iMin\ I)$
by (*simp add*: *iMin-def LeastI2-order*)

lemma *wellorder-iMin-lemma*:
$k \in I \Longrightarrow iMin\ I \in I \wedge iMin\ I \leq k$
by (*blast intro*: *iMinI iMin-le*)

lemma *iMin-Min-conv*:
$[\![\ finite\ I;\ I \neq \{\}\ ]\!] \Longrightarrow iMin\ I = Min\ I$
apply (*rule order-antisym*)
apply (*rule Min-ge-iff*[*THEN iffD2*], *assumption*+)
apply *blast*

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*⇒*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*:
⟦ ∃ $x.\ P\ x$; ∃ $x.\ Q\ x$ ⟧ ⟹
($LEAST\ (x::'a::wellorder).\ (P\ x \vee Q\ x)) = min\ (LEAST\ x.\ P\ x)\ (LEAST\ x.\ Q\ x)$
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 if-split*, *safe*)
apply *blast*
apply (*subst disj-commute*)
apply (*fastforce simp*: *linorder-not-le*)
done

lemma *Least-imp-le*:
⟦ ∃ $x.\ P\ x$; $\bigwedge x.\ P\ x \Longrightarrow Q\ x$ ⟧ ⟹
($LEAST\ (x::'a::wellorder).\ Q\ x) \leq (LEAST\ x.\ P\ x)$
by (*blast intro*: *Least-le LeastI2-ex*)

lemma *Least-imp-disj-eq*:
⟦ ∃ $x.\ P\ x$; $\bigwedge x.\ P\ x \Longrightarrow Q\ x$ ⟧ ⟹
($LEAST\ (x::'a::wellorder).\ P\ x \vee Q\ x) = (LEAST\ x.\ Q\ x)$
apply (*subst Least-disj*, *assumption*, *blast*)
apply (*subst min.commute*)
apply (*rule min.absorb-iff1*[*THEN iffD1*])
apply (*rule Least-imp-le*, *assumption*, *blast*)

done

lemma *Least-le-imp-le*:
$[\![\ \exists x.\ P\ x;\ \exists x.\ Q\ x;\ \bigwedge x\ y.\ [\![\ P\ x;\ Q\ y\ ]\!] \Longrightarrow x \leq y\ ]\!] \Longrightarrow$
$(LEAST\ (x::'a::wellorder).\ P\ x) \leq (LEAST\ (x::'a::wellorder).\ Q\ x)$
by (*blast intro*: *LeastI*)
lemma *Least-le-imp-le-disj*:
$[\![\ \exists x.\ P\ x;\ \bigwedge x\ y.\ [\![\ P\ x;\ Q\ y\ ]\!] \Longrightarrow x \leq y\ ]\!] \Longrightarrow$
$(LEAST\ (x::'a::wellorder).\ P\ x \vee Q\ x) = (LEAST\ (x::'a::wellorder).\ P\ x)$
apply (*case-tac* $\exists x.\ Q\ x$)
apply (*simp only*: *Least-disj min.absorb-iff1*[*THEN iffD1*, *OF Least-le-imp-le*])
apply *simp*
done

lemma *Max-equality*: $[\![\ (k::'a::linorder) \in A;\ finite\ A;\ \bigwedge x.\ x \in A \Longrightarrow x \leq k\ ]\!] \Longrightarrow$
$Max\ A = k$
by (*rule Max-eqI*)

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

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

lemma *Collect-Max-ge*: $[\![\ finite\ \{x.\ P\ x\};\ P\ k\ ]\!] \Longrightarrow k \leq Max\ \{x.\ P\ x\}$
by (*rule Max-ge*, *assumption*, *rule CollectI*)

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

lemma *MaxI-ex*: $[\![\ \exists x.\ x \in A;\ finite\ A\ ]\!] \Longrightarrow Max\ A \in A$
by (*blast intro*: *MaxI*)

lemma *MaxI-ex2*: $[\![\ A \neq \{\};\ finite\ A\ ]\!] \Longrightarrow Max\ A \in A$
by (*blast intro*: *MaxI*)

lemma *MaxI2*: $[\![\ k \in A;\ \bigwedge x.\ x \in A \Longrightarrow P\ x;\ finite\ A\ ]\!] \Longrightarrow P\ (Max\ A)$
by (*drule Max-in*, *blast+*)

lemma *MaxI2-ex*:$[\![\ \exists x.\ x \in A;\ \bigwedge x.\ x \in A \Longrightarrow P\ x;\ finite\ A\ ]\!] \Longrightarrow P\ (Max\ A)$
by (*blast intro*: *MaxI2*)

lemma *MaxI2-ex2*:$[\![\ A \neq \{\};\ \bigwedge x.\ x \in A \Longrightarrow P\ x;\ finite\ A\ ]\!] \Longrightarrow P\ (Max\ A)$
by (*blast intro*: *MaxI2*)

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*)
apply (*rule Max-equality*, *clarsimp*+)
done

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

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*+)

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

lemma *Max-the2*: ⟦ *A* ≠ {}; *finite A* ⟧ ⟹
Max A = (*THE x*. *x* ∈ *A* ∧ (∀ *y*∈*A*. *y* ≤ *x*))
apply (*simp add*: *Max-the*)
apply (*subgoal-tac* ⋀*x*. (∀ *y*∈*A*. *y* ≤ *x*) = (∀ *y*. *y* ∈ *A* ⟶ *y* ≤ *x*))
prefer *2*
apply *blast*
apply *simp*
done

lemma *wellorder-Max-lemma*: ⟦ *k* ∈ *A*; *finite A* ⟧ ⟹ *Max A* ∈ *A* ∧ *k* ≤ *Max A*
by (*case-tac A* = {}, *simp-all*)

lemma *MaxI2-order*: ⟦ *k* ∈ *A*; *finite A*; ⋀*y*. *y* ∈ *A* ⟹ *y* ≤ *k*;
⋀*x*. ⟦ *x* ∈ *A*; ∀ *y*∈*A*. *y* ≤ *x* ⟧ ⟹ *P x* ⟧
⟹ *P* (*Max A*)

by (*simp add*: *Max-equality*)

lemma *Min-le-Max*: ⟦ *finite A*; *A* ≠ {} ⟧ ⟹ *Min A* ≤ *Max A*
by (*rule Max-ge*[*OF - Min-in*])

lemma *iMin-le-Max*: ⟦ *finite A*; *A* ≠ {} ⟧ ⟹ *iMin A* ≤ *Max A*
by (*rule ssubst*[*OF iMin-Min-conv*], *assumption*+, *rule Min-le-Max*)

6.2.2 *Max* for sets over *enat*

definition *iMax* :: *nat set* ⇒ *enat*
  where *iMax i* ≡ *if* (*finite i*) *then* (*enat* (*Max i*)) *else* ∞

lemma *iMax-finite-conv*: *finite I* = (*iMax I* ≠ ∞)
by (*simp add*: *iMax-def*)

lemma *iMax-infinite-conv*: *infinite I* = (*iMax I* = ∞)
by (*simp add*: *iMax-def*)

lemma *class.distrib-lattice* (*min*::('*a*::*linorder* ⇒ '*a* ⇒ '*a*)) (≤) (<) *max*
apply (*subgoal-tac class.order* (≤) (<))
 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* (≤) (<))
 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*)
  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 max.cobounded1*)
  apply (*rule max.cobounded2*)
 apply (*rule conj-le-imp-max*, *assumption*+)
apply (*rule class.distrib-lattice-axioms.intro*)
apply (*rule max-min-distrib2*)
done

print-locale *Lattices.distrib-lattice*

lemma *max-Min-eq-Min-max*[*rule-format*]:
  finite A $\Longrightarrow$
  $A \neq \{\} \longrightarrow$
  max x (*Min A*) = *Min* $\{max\ x\ a\ |a.\ a \in A\}$
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=$\{max\ x\ b\ |b.\ b \in insert\ a\ A1\}$ **and**
  s=*insert* (*max x a*) $\{max\ x\ b\ |b.\ b \in A1\}$
  in *subst*)
 apply *blast*
apply (*subst Min-insert*, *simp-all*)
apply (*case-tac a* $\leq$ *Min A1*)
 apply (*frule max-le-monoR*[**where** *x*=*x*])
 apply (*simp only*: *min-eqL*)
apply (*simp only*: *linorder-not-le*)
apply (*frule max-le-monoR*[**where** *x*=*x*, *OF less-imp-le*])
apply (*simp only*: *min-eqR*)
done

lemma *max-iMin-eq-iMin-max*:
  $[\![$ *finite A*; $A \neq \{\}$ $]\!] \Longrightarrow$
  max x (*iMin A*) = *iMin* $\{max\ x\ a\ |a.\ a \in A\}$
apply (*simp add*: *iMin-Min-conv*)
apply (*insert iMin-Min-conv*[*of* $\{max\ x\ a\ |a.\ a \in A\}$], *simp*)
apply (*subgoal-tac finite* $\{max\ x\ a\ |a.\ a \in A\}$)
 prefer *2*
 apply *simp*
apply (*simp add*: *max-Min-eq-Min-max*)
done

lemma $[\![$ *finite A*; $A \neq \{\}$ $]\!] \Longrightarrow \forall x \in A.\ x \leq Max\ A$
by *simp*

6.2.3 *Min* and *Max* for set operations

lemma *iMin-subset*: $[\![\ A \neq \{\};\ A \subseteq B\ ]\!] \Longrightarrow iMin\ B \leq iMin\ A$
by (*blast intro*: *iMin-le iMinI-ex2*)

lemma *Max-subset*: $[\![\ A \neq \{\};\ A \subseteq B;$ *finite B* $]\!] \Longrightarrow Max\ A \leq Max\ B$
by (*rule linorder-class.Max-mono*)

lemma *Min-subset*: $[\![\ A \neq \{\};\ A \subseteq B;$ *finite B* $]\!] \Longrightarrow Min\ B \leq Min\ A$
by (*rule linorder-class.Min-antimono*)

lemma *iMin-Int-ge1*: $(A \cap B) \neq \{\} \Longrightarrow iMin\ A \leq iMin\ (A \cap B)$
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*: $[\![\ (A \cap B) \neq \{\};\ finite\ A\ ]\!] \Longrightarrow Max\ (A \cap B) \leq Max\ A$
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*)
apply (*simp add*: *min-le-iff-disj*)
apply (*insert iMinI-ex2*[*of A∪B*])
apply (*blast intro*: *iMin-le*)
done

lemma *iMin-singleton*[*simp*]: $iMin\ \{a\} = a$
by (*rule singletonI*[*THEN iMinI*, *THEN singletonD*])

lemma *iMax-singleton*[*simp*]: $iMax\ \{a\} = enat\ a$
by (*simp add*: *iMax-def*)

lemma *Max-le-Min-imp-singleton*:
$[\![\ finite\ A;\ A \neq \{\};\ Max\ A \leq Min\ A\ ]\!] \Longrightarrow A = \{Min\ A\}$
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*; $A \neq \{\}$ ⟧ ⟹ $(Max\ A \leq Min\ A) = (\exists x.\ A = \{x\})$
apply (*rule iffI*)
apply (*rule-tac x=Min A* **in** *exI*)
apply (*rule Max-le-Min-imp-singleton*, *assumption*+)
apply *fastforce*
done

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

lemma *le-imp-Max-le-iMin*:
⟦ *finite A*; $A \neq \{\}$; $B \neq \{\}$; $\forall a{\in}A.\ \forall b{\in}B.\ a \leq b$ ⟧ ⟹ $Max\ A \leq iMin\ B$
by (*blast intro*: *Max-in iMinI-ex2*)

lemma *Max-le-iMin-conv-le*:
⟦ *finite A*; $A \neq \{\}$; $B \neq \{\}$ ⟧ ⟹ $(Max\ A \leq iMin\ B) = (\forall a{\in}A.\ \forall b{\in}B.\ a \leq b)$
by (*blast intro*: *Max-le-iMin-imp-le le-imp-Max-le-iMin*)

lemma *Max-less-iMin-imp-less*:
⟦ *finite A*; $Max\ A < iMin\ B$; $a \in A$; $b \in B$ ⟧ ⟹ $a < b$
by (*blast dest*: *Max-less-iff intro*: *order-less-le-trans iMin-le*)

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

lemma *Max-less-iMin-conv-less*:
⟦ *finite A*; $A \neq \{\}$; $B \neq \{\}$ ⟧ ⟹ $(Max\ A < iMin\ B) = (\forall a{\in}A.\ \forall b{\in}B.\ a < b)$
by (*blast intro*: *Max-less-iMin-imp-less less-imp-Max-less-iMin*)

lemma *Max-less-iMin-imp-disjoint*:
⟦ *finite A*; $Max\ A < iMin\ B$ ⟧ ⟹ $A \cap B = \{\}$
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*)

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

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

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*)

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*)

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

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*)

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*)

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*)

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*)

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*)

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*)
apply (*simp add*: *min-eqL*[*OF iMin-le*])

apply *simp*
done

Synonyms for similarity with theorem names for *Min*"

lemmas *iMin-eqI* = *iMin-equality*

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*, *trans*

abbreviation (*input*) *reflP* :: ('*a* ⇒ '*a* ⇒ *bool*) ⇒ *bool* **where**
reflP r ≡ *refl* {(*x*, *y*). *r x y*}

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

abbreviation (*input*) *transP* :: ('*a* => '*a* => *bool*) => *bool* **where**
transP r == *trans* {(*x*, *y*). *r x y*}

abbreviation (*input*) *equivP* :: ('*a* ⇒ '*a* ⇒ *bool*) ⇒ *bool* **where**
equivP r ≡ *reflP r* ∧ *symP r* ∧ *transp r*

abbreviation (*input*) *irreflP* :: ('*a* ⇒ '*a* ⇒ *bool*) ⇒ *bool* **where**
irreflP r ≡ *irrefl* {(*x*, *y*). *r x y*}

Example for *reflP*

lemma *reflP* ((≤)::('*a*::*preorder* ⇒ '*a* ⇒ *bool*))
by (*simp add*: *refl-on-def*)

Example for *symP*

lemma *symP* (=)
by (*simp add*: *sym-def*)

Example for *equivP*

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

Example for *irreflP*

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

6.3.2 Auxiliary results for *singletons*

lemma *singleton-not-empty*: {*a*} ≠ {} **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*

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*

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*: $\{a\} \cap A = (\textit{if } a \in A \textit{ then } \{a\} \textit{ else } \{\})$
by (*split if-split*, *blast*)

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

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

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

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

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

lemma *subset-finite-imp-finite*: $[\![\ \textit{finite } A;\ B \subseteq A\ ]\!] \Longrightarrow \textit{finite } B$
by (*rule finite-subset*)

lemma *infinite-not-subset-finite*: $[\![\ \textit{infinite } A;\ \textit{finite } B\ ]\!] \Longrightarrow \neg\ A \subseteq B$
by (*blast intro*: *subset-finite-imp-finite*)

lemma *Un-infinite-right*: *infinite* $T \Longrightarrow \textit{infinite } (S \cup T)$ **by** *blast*
lemma *Un-infinite-iff*: *infinite* $(S \cup T) = (\textit{infinite } S \vee \textit{infinite } T)$ **by** *blast*

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

corollary *finite-A-int-A-conv*: *finite* $A = \textit{finite } (\textit{int } `\ A)$
proof −
  have *inj-on int A*
    by (*auto intro*: *inj-onI*)
  then show *?thesis*
    by (*simp add*: *finite-image-iff*)
qed

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* ≠ {} ⟧ ⟹ *infinite* (*A* × *B*)
by (*blast dest*: *finite-cartesian-productD1*)

lemma *cartesian-product-infiniteR-imp-infinite*: ⟦ *infinite B*; *A* ≠ {} ⟧ ⟹ *infinite* (*A* × *B*)
by (*blast dest*: *finite-cartesian-productD2*)

lemma *finite-nat-iff-bounded2*:
finite S = (∃ (*k*::*nat*).∀ *n*∈*S*. *n* < *k*)
by (*simp only*: *finite-nat-iff-bounded*, *blast*)

lemma *finite-nat-iff-bounded-le2*:
finite S = (∃ (*k*::*nat*).∀ *n*∈*S*. *n* ≤ *k*)
by (*simp only*: *finite-nat-iff-bounded-le*, *blast*)

lemma *nat-asc-chain-imp-unbounded*:
⟦ *S* ≠ {}; (∀ *m*∈*S*. ∃ *n*∈*S*. *m* < (*n*::*nat*)) ⟧ ⟹ ∀ *m*. ∃ *n*∈*S*. *m* ≤ *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

lemma *infinite-nat-iff-asc-chain*:
S ≠ {} ⟹ *infinite S* = (∀ *m*∈*S*. ∃ *n*∈*S*. *m* < (*n*::*nat*))
by (*metis Max-in infinite-nat-iff-unbounded not-greater-Max*)

lemma *infinite-imp-asc-chain*: *infinite S* ⟹ ∀ *m*∈*S*. ∃ *n*∈*S*. *m* < (*n*::*nat*)
by (*rule infinite-nat-iff-asc-chain*[*THEN iffD1*, *OF infinite-imp-nonempty*])

lemma *infinite-image-imp-infinite*: *infinite* (*f* ‘ *A*) ⟹ *infinite A*
by *fastforce*

lemma *inj-on-imp-infinite-image*: ⟦ *infinite A*; *inj-on f A* ⟧ ⟹ *infinite* (*f* ‘ *A*)
apply (*frule card-image*)
apply (*fastforce 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*) = (*finite A* $\longrightarrow$ *n* < *Max A*)
apply (*rule iffI*)
apply (*blast intro*: *order-less-le-trans Max-ge*)
apply (*case-tac finite A*)
apply (*blast intro*: *Max-in*)
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*) = (*finite A* $\wedge$ *A* $\neq$ {} $\wedge$ *n* < *Max A* $\vee$ *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

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*
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*

lemma {*f x* |*x*. *x* $\in$ *A*} = ($\bigcup$ *x*$\in$*A*. {*f x*})
by *fastforce*

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*; ⋀*c*. *c* ∈ *C* ⟹ *card c* = *k*; ⋀*c1 c2*. ⟦*c1* ∈ *C*; *c2* ∈ *C*; *c1* ≠ *c2*⟧ ⟹ *c1* ∩ *c2* = {} ⟧ ⟹
  card (⋃ *C*) = *k* * *card C*
by (*metis card.infinite card-partition finite-Union mult-eq-if*)

6.3.5 Some auxiliary results for subset relation

lemma *subset-image-Int*: *A* ⊆ *B* ⟹ *f* ‘ (*A* ∩ *B*) = *f* ‘ *A* ∩ *f* ‘ *B*
by (*simp only*: *Int-absorb2 image-mono*)

lemma *image-diff-disjoint-image-Int*:
  ⟦ *f* ‘ (*A* − *B*) ∩ *f* ‘ *B* = {} ⟧ ⟹
  f ‘ (*A* ∩ *B*) = *f* ‘ *A* ∩ *f* ‘ *B*
apply (*rule set-eqI*)
apply (*simp add*: *image-iff*)
apply *blast*
done

lemma *subset-imp-Int-subset1*: *A* ⊆ *C* ⟹ *A* ∩ *B* ⊆ *C*
by (*rule subset-trans*[*of* - *C* ∩ *B*, *OF Int-mono*, *OF* - *subset-refl Int-lower1*])

lemma *subset-imp-Int-subset2*: *B* ⊆ *C* ⟹ *A* ∩ *B* ⊆ *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

lemma *image-add-atLeast*:
  (λ*n*::*nat*. *n*+*k*) ‘ {*i*..} = {*i*+*k*..} (**is** *?A* = *?B*)
proof
  show *?A* ⊆ *?B* **by** *fastforce*
next
  show *?B* ⊆ *?A*
  proof
    fix *n* **assume** *a*: *n* : *?B*
    hence *n* − *k* ∈ {*i*..} **by** *simp*
    moreover have *n* = (*n* − *k*) + *k* **using** *a* **by** *fastforce*
    ultimately show *n* ∈ *?A* **by** *blast*
  qed
qed

lemma *image-add-atMost*:

(λn::nat. n+k) ‘ {..i} = {k..i+k} (**is** *?A = ?B*)

proof −
have *s1*: {..i} = {*0*..i}
by (*simp add*: *set-interval-defs*)
then show *?A = ?B*
by *simp*
qed

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

corollary *image-Suc-atMost*: *Suc* ‘ {..i} = {*Suc 0*..*Suc* i}
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*) ≤ k′ ⟹ {l..k} ⊆ {..k′}
by (*simp*)

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

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

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; $Max\ A \leq n$ $]\!] \Longrightarrow A \subseteq \{..n\}$
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* ⟹ *Max* {..<*n*} = *n* − *Suc 0*
by (*rule Max-equality*[*OF* - *finite-lessThan*], *simp-all*)

lemma *iMin-atLeastLessThan*: *m* < *n* ⟹ *iMin* {*m*..<*n*} = *m*
by (*rule iMin-equality*, *simp-all*)

lemma *Max-atLeastLessThan*: *m* < *n* ⟹ *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* ⟹ *iMin* {*m*<..<*n*} = *Suc m*
by (*simp only*: *atLeastSucLessThan-greaterThanLessThan*[*symmetric*] *iMin-atLeastLessThan*)

lemma *Max-greaterThanLessThan*: *Suc m* < *n* ⟹ *Max* {*m*<..<*n*} = *n* − *Suc 0*
by (*simp only*: *atLeastSucLessThan-greaterThanLessThan*[*symmetric*] *Max-atLeastLessThan*)

lemma *iMin-greaterThanAtMost*: *m* < *n* ⟹ *iMin* {*m*<..*n*} = *Suc m*
by (*simp only*: *atLeastSucAtMost-greaterThanAtMost*[*symmetric*] *atLeastLessThanSuc-atLeastAtMost*[*symmetr*
iMin-atLeastLessThan)

lemma *Max-greaterThanAtMost*: *m* < *n* ⟹ *Max* {*m*<..(*n*::*nat*)} = *n*
by (*simp add*: *atLeastSucAtMost-greaterThanAtMost*[*symmetric*] *atLeastLessThanSuc-atLeastAtMost*[*symmetr*
Max-atLeastLessThan)

lemma *iMin-atLeastAtMost*: *m* ≤ *n* ⟹ *iMin* {*m*..*n*} = *m*
by (*rule iMin-equality*, *simp-all*)

lemma *Max-atLeastAtMost*: *m* ≤ *n* ⟹ *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*], *fastforce*)

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*=λ*x*. *nat* (*x* − *n*) **in** *inj-on-infinite-image-iff*[*THEN iffD1*, *rule-format*])
apply (*fastforce simp*: *inj-on-def*)
apply (*rule-tac t*=((λ*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*)<..}
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*=((−) *n* ' {..*n*}) **and** *s*={*0*..} **in** *subst*)
apply (*simp add*: *set-eq-iff image-iff Bex-def*)
apply (*rule infinite-atLeast-int*)
done

lemma *infinite-lessThan-int*: *infinite* {..<(*n*::*int*)}
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 *sum-singleton*: (∑ *x*∈{*a*}. *f x*) = *f a*
by *simp*

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

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 *fastforce*
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* ⟹ (*A* ≠ {}) = (*0* < *card A*)
by *fastforce*

lemma *nat-card-le-Max*: *card* (*A*::*nat set*) ≤ *Suc* (*Max A*)

apply (*case-tac finite A*)
prefer *2*
apply *simp*
apply (*cut-tac card-mono*[*OF finite-atMost*, *of A Max A*])
apply *simp*
apply *fastforce*
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*)

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 $(f \text{ ` } A) <$ *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*
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*, *fastforce*+)
done

corollary *pigeonhole-principle-linorder*[*rule-format*]:
card $(f \text{ ` } A) <$ *card* $(A{::}'a{::}$*linorder set*$) \Longrightarrow (\exists\, x{\in}A.\ \exists\, y{\in}A.\ x < y \wedge f\, x = f\, y)$
apply (*drule pigeonhole-principle*, *clarify*)
apply (*drule neq-iff*[*THEN iffD1*])
apply *fastforce*
done

corollary *pigeonhole-mod*:
$[\![\ 0 < m;\ m <$ *card A* $]\!] \Longrightarrow \exists\, x{\in}A.\ \exists\, y{\in}A.\ x < y \wedge x \bmod m = y \bmod m$
apply (*rule pigeonhole-principle-linorder*)

apply (*rule le-less-trans*[*of* - *card* {..<*m*}])
apply (*rule card-mono*)
apply *fastforce*+
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*
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 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* ⟹ ∃ *P*. *f A* = {*x* ∈ *A*. *P x*}
unfolding *set-restriction-def* **by** *blast*
lemma *set-restrictionI*: *f* = (λ*A*. {*x* ∈ *A*. *P x*}) ⟹ *set-restriction f*
unfolding *set-restriction-def* **by** *blast*

lemma *set-restriction-comp*:
⟦ *set-restriction f*; *set-restriction g* ⟧ ⟹ *set-restriction* (*f* ∘ *g*)
apply (*unfold set-restriction-def*)
apply (*elim exE*, *rename-tac P1 P2*)
apply (*rule-tac x*=λ*x*. *P1 x* ∧ *P2 x* **in** *exI*)
apply *fastforce*
done

lemma *set-restriction-commute*:
⟦ *set-restriction f*; *set-restriction g* ⟧ ⟹ *f* (*g I*) = *g* (*f I*)
unfolding *set-restriction-def* **by** *fastforce*

Constructs a set restriction function with the given restriction predicate

definition
set-restriction-fun :: ('*a* ⇒ *bool*) ⇒ ('*a set* ⇒ '*a set*)
where
set-restriction-fun P ≡ λ*A*. {*x* ∈ *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 = (∃ *B*. ∀ *A*. *f A* = *A* ∩ *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*=λ*x*. *x* ∈ *B* **in** *exI*, *blast*)
done

lemma *set-restriction-Un*:
set-restriction f ⟹ *f* (*A* ∪ *B*) = *f A* ∪ *f B*
unfolding *set-restriction-def* **by** *fastforce*
lemma *set-restriction-Int*:
set-restriction f ⟹ *f* (*A* ∩ *B*) = *f A* ∩ *f B*
unfolding *set-restriction-def* **by** *fastforce*
lemma *set-restriction-Diff*:
set-restriction f ⟹ *f* (*A* − *B*) = *f A* − *f B*
unfolding *set-restriction-def* **by** *fastforce*
lemma *set-restriction-mono*:
⟦ *set-restriction f*; *A* ⊆ *B* ⟧ ⟹ *f A* ⊆ *f B*
unfolding *set-restriction-def* **by** *fastforce*
lemma *set-restriction-absorb*:
set-restriction f ⟹ *f* (*f A*) = *f A*
unfolding *set-restriction-def* **by** *fastforce*
lemma *set-restriction-empty*:
set-restriction f ⟹ *f* {} = {}
unfolding *set-restriction-def* **by** *blast*
lemma *set-restriction-non-empty-imp*:
⟦ *set-restriction f*; *f A* ≠ {} ⟧ ⟹ *A* ≠ {}
unfolding *set-restriction-def* **by** *blast*
lemma *set-restriction-subset*:
set-restriction f ⟹ *f A* ⊆ *A*
unfolding *set-restriction-def* **by** *blast*
lemma *set-restriction-finite*:
⟦ *set-restriction f*; *finite A* ⟧ ⟹ *finite* (*f A*)
unfolding *set-restriction-def* **by** *fastforce*

lemma *set-restriction-card*:
⟦ *set-restriction f*; *finite A* ⟧ ⟹
card (*f A*) = *card A* − *card* {*a* ∈ *A*. *f* {*a*} = {}}
apply (*unfold set-restriction-def*)
apply (*subgoal-tac* {*a* ∈ *A*. *f* {*a*} = {}} ⊆ *A*)
prefer *2*
apply *blast*
apply (*frule finite-subset*, *simp*)
apply (*simp only*: *card-Diff-subset*[*symmetric*])
apply (*rule arg-cong*[**where** *f*=*card*])
apply *fastforce*
done

lemma *set-restriction-card-le*:
⟦ *set-restriction f*; *finite A* ⟧ ⟹ *card* (*f A*) ≤ *card A*
by (*simp add*: *set-restriction-card*)

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

lemma *set-restriction-in-imp*:
⟦ *set-restriction f*; *x* ∈ *f A* ⟧ ⟹ *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* ↓≤ *t* ≡ {*x*∈*I*. *x* ≤ *t*}

definition *cut-less* :: *'a*::*linorder set* ⇒ *'a* ⇒ *'a set* (**infixl** ‹↓<› *100*)
where *I* ↓< *t* ≡ {*x*∈*I*. *x* < *t*}

definition *cut-ge* :: *'a*::*linorder set* ⇒ *'a* ⇒ *'a set* (**infixl** ‹↓≥› *100*)
where *I* ↓≥ *t* ≡ {*x*∈*I*. *t* ≤ *x*}

definition *cut-greater* :: *'a*::*linorder set* ⇒ *'a* ⇒ *'a set* (**infixl** ‹↓>› *100*)
where *I* ↓> *t* ≡ {*x*∈*I*. *t* < *x*}

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

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

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

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 (*fastforce 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

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*:
$[\![$ *cut-op* $= (\downarrow<) \vee$ *cut-op* $= (\downarrow\leq) \vee$
cut-op $= (\downarrow>) \vee$ *cut-op* $= (\downarrow\geq)$;
$f = (\lambda I.$ *cut-op* $I\ t)\ ]\!] \Longrightarrow$ *set-restriction* f
apply *safe*

apply (*simp-all only*: *i-cut-eq-set-restriction-fun set-restriction-fun-is-set-restriction*)
done

corollary
  i-cut-less-set-restriction: *set-restriction* ($\lambda I.\ I \downarrow< t$) **and**
  i-cut-le-set-restriction: *set-restriction* ($\lambda I.\ I \downarrow\leq t$) **and**
  i-cut-greater-set-restriction: *set-restriction* ($\lambda I.\ I \downarrow> t$) **and**
  i-cut-ge-set-restriction: *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*: ⟦
  cut-op1 = ($\downarrow<$) ∨ *cut-op1* = ($\downarrow\leq$) ∨
  cut-op1 = ($\downarrow>$) ∨ *cut-op1* = ($\downarrow\geq$);
  cut-op2 = ($\downarrow<$) ∨ *cut-op2* = ($\downarrow\leq$) ∨
  cut-op2 = ($\downarrow>$) ∨ *cut-op2* = ($\downarrow\geq$) ⟧ ⟹
  cut-op2 (*cut-op1 I t1*) *t2* = *cut-op1* (*cut-op2 I t2*) *t1*
apply (*rule set-restriction-commute*)
apply (*simp-all only*: *i-cut-set-restriction-disj*)
done

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

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

lemma
  cut-less-singleton: {*a*} $\downarrow<$ *t* = (*if a* < *t then* {*a*} *else* {}) **and**
  cut-le-singleton: {*a*} $\downarrow\leq$ *t* = (*if a* ≤ *t then* {*a*} *else* {}) **and**
  cut-greater-singleton: {*a*} $\downarrow>$ *t* = (*if a* > *t then* {*a*} *else* {}) **and**
  cut-ge-singleton: {*a*} $\downarrow\geq$ *t* = (*if a* ≥ *t then* {*a*} *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: $I \downarrow\geq t \subseteq I$
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

lemma *i-cut-Un-disj*:
$[\![$ *cut-op* $= (\downarrow<) \vee$ *cut-op* $= (\downarrow\leq) \vee$
cut-op $= (\downarrow>) \vee$ *cut-op* $= (\downarrow\geq)$ $]\!]$
$\Longrightarrow$ *cut-op* $(A \cup B)$ t = *cut-op* A $t \cup$ *cut-op* B t
apply (*drule i-cut-set-restriction-disj*[**where** f=$\lambda I.$ *cut-op* I t], *simp*)
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*:
$[\![$ *cut-op* $= (\downarrow<) \vee$ *cut-op* $= (\downarrow\leq) \vee$
cut-op $= (\downarrow>) \vee$ *cut-op* $= (\downarrow\geq)$ $]\!]$
$\Longrightarrow$ *cut-op* $(A \cap B)$ t = *cut-op* A $t \cap$ *cut-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* $= (\downarrow<) \vee$ *cut-op* $= (\downarrow\leq) \vee$
cut-op $= (\downarrow>) \vee$ *cut-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

lemma *i-cut-subset-mono-disj*:
$[\![$ *cut-op* $= (\downarrow<) \vee$ *cut-op* $= (\downarrow\leq) \vee$
cut-op $= (\downarrow>) \vee$ *cut-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*)
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* $= (\downarrow<) \vee$ *cut-op* $= (\downarrow\leq) \vee$
cut-op $= (\downarrow>) \vee$ *cut-op* $= (\downarrow\geq)\ ]\!]$
$\Longrightarrow$ *cut-op* (*cut-op I t*) *t* = *cut-op I t*
apply (*drule i-cut-set-restriction-disj*[**where** $f{=}\lambda I.$ *cut-op I t*], *blast*)
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: $I \downarrow> t \downarrow> t = I \downarrow> t$
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** *fastforce*+

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

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*)

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: $x \notin I \Longrightarrow x \notin I \downarrow> t$
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

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: $x \in I \downarrow> t \Longrightarrow x \in I$
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 *fastforce*
lemma *cut-less-cut-ge-ident*: $t2 \leq t1 \Longrightarrow I \downarrow< t1 \cup I \downarrow\geq t2 = I$
by *fastforce*
lemma *cut-le-cut-ge-ident*: $t2 \leq t1 \Longrightarrow I \downarrow\leq t1 \cup I \downarrow\geq t2 = I$
by *fastforce*

lemma *cut-less-cut-greater-ident*:
⟦ *t2* ≤ *t1*; *I* ∩ {*t1*..*t2*} = {} ⟧ ⟹ *I* ↓< *t1* ∪ *I* ↓> *t2* = *I*
by *fastforce*
corollary *cut-less-cut-greater-ident'*:
t ∉ *I* ⟹ *I* ↓< *t* ∪ *I* ↓> *t* = *I*
by (*simp add*: *cut-less-cut-greater-ident*)

lemma *insert-eq-cut-less-cut-greater*: *insert n I* = *I* ↓< *n* ∪ {*n*} ∪ *I* ↓> *n*
by *fastforce*

7.2.3 Relations between cut operators

lemma *insert-Int-conv-if*: *A* ∩ (*insert x B*) = (
if x ∈ *A then insert x* (*A* ∩ *B*) *else A* ∩ *B*)
by *simp*

lemma *cut-le-less-conv-if*: *I* ↓≤ *t* = (
if t ∈ *I then insert t* (*I* ↓< *t*) *else* (*I* ↓< *t*))
by (*simp add*: *i-cut-Int-conv lessThan-insert*[*symmetric*] *insert-Int-conv-if*)

lemma *cut-le-less-conv*: *I* ↓≤ *t* = ({*t*} ∩ *I*) ∪ (*I* ↓< *t*)
by *fastforce*

lemma *cut-less-le-conv*: *I* ↓< *t* = (*I* ↓≤ *t*) − {*t*}
by *fastforce*
lemma *cut-less-le-conv-if*: *I* ↓< *t* = (
if t ∈ *I then* (*I* ↓≤ *t*) − {*t*} *else* (*I* ↓≤ *t*))
by (*simp only*: *cut-less-le-conv*, *force*)

lemma *cut-ge-greater-conv-if*: *I* ↓≥ *t* = (
if t ∈ *I then insert t* (*I* ↓> *t*) *else* (*I* ↓> *t*))
by (*simp add*: *i-cut-Int-conv greaterThan-insert*[*symmetric*] *insert-Int-conv-if*)
lemma *cut-ge-greater-conv*: *I* ↓≥ *t* = ({*t*} ∩ *I*) ∪ (*I* ↓> *t*)
apply (*simp only*: *cut-ge-greater-conv-if*)
apply (*case-tac t* ∈ *I*)
apply *simp-all*
done
lemma *cut-greater-ge-conv*: *I* ↓> *t* = (*I* ↓≥ *t*) − {*t*}
by *fastforce*
lemma *cut-greater-ge-conv-if*: *I* ↓> *t* = (
if t ∈ *I then* (*I* ↓≥ *t*) − {*t*} *else* (*I* ↓≥ *t*))
by (*simp only*: *cut-greater-ge-conv*, *force*)

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

7.2.4 Function images with cut operators

lemma *cut-less-image*:
$[\![$ *strict-mono-on f A*; $I \subseteq A$; $n \in A$ $]\!] \Longrightarrow$
$(f\ `\ I) \downarrow< f\ n = f\ `\ (I \downarrow< n)$
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 rev-subsetD*)
apply *blast*
done

lemma *cut-le-image*:
$[\![$ *strict-mono-on f A*; $I \subseteq A$; $n \in A$ $]\!] \Longrightarrow$
$(f\ `\ I) \downarrow\leq f\ n = f\ `\ (I \downarrow\leq n)$
apply (*frule strict-mono-on-imp-inj-on*)
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*; $I \subseteq A$; $n \in A$ $]\!] \Longrightarrow$
$(f\ `\ I) \downarrow> f\ n = f\ `\ (I \downarrow> n)$
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 rev-subsetD*)
apply *blast*
done

lemma *cut-ge-image*:
$[\![$ *strict-mono-on f A*; $I \subseteq A$; $n \in A$ $]\!] \Longrightarrow$
$(f\ `\ I) \downarrow\geq f\ n = f\ `\ (I \downarrow\geq n)$
apply (*frule strict-mono-on-imp-inj-on*)
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

7.2.5 Finiteness and cardinality with cut operators

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

lemma *nat-cut-le-finite*: *finite* $(I \downarrow\leq (t::nat))$
by (*fastforce simp*: *finite-nat-iff-bounded-le2 cut-le-def*)

lemma *nat-cut-less-finite*: *finite* $(I \downarrow< (t::nat))$
by (*fastforce simp*: *finite-nat-iff-bounded2 cut-less-def*)

lemma *nat-cut-ge-finite-iff*: *finite* $(I \downarrow\geq (t::nat)) =$ *finite* I
apply (*rule iffI*)
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*: *finite* $(I \downarrow> (t::nat)) =$ *finite* I
by (*simp only*: *nat-cut-ge-greater-conv*[*symmetric*] *nat-cut-ge-finite-iff*)

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

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

lemma *nat-cut-ge-card*: *card* $(I \downarrow\geq (t::nat)) \leq$ *card* I
apply (*case-tac finite I*)
apply (*simp add*: *cut-ge-card*)
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*: $I \downarrow> (iMin\ I) = I - \{iMin\ I\}$
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 (*fastforce simp*: *i-cut-defs*)
lemma *cut-less-Min-empty*: $t \leq iMin\ I \Longrightarrow I \downarrow< t = \{\}$
by (*fastforce 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*)
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*
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*
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

lemma *cut-ge-Max-empty*: *finite* $I \Longrightarrow Max\ I < t \Longrightarrow I \downarrow\geq t = \{\}$

by (*fastforce simp*: *i-cut-defs*)

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

lemma *cut-less-Max-all*: *finite* $I \Longrightarrow Max\ I < t \Longrightarrow I \downarrow< t = I$
by (*fastforce 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

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

lemma
greaterThanLessThan-cut-le: $\{m<..<n\} \downarrow\leq t = (\textit{if } n \leq t \textit{ then } \{m<..<n\} \textit{ else } \{m<..t\})$ **and**
greaterThanLessThan-cut-less: $\{m<..<n\} \downarrow< t = (\textit{if } n \leq t \textit{ then } \{m<..<n\} \textit{ else } \{m<..<t\})$ **and**
greaterThanLessThan-cut-ge: $\{m<..<n\} \downarrow\geq t = (\textit{if } t \leq m \textit{ then } \{m<..<n\} \textit{ else } \{t..<n\})$ **and**
greaterThanLessThan-cut-greater: $\{m<..<n\} \downarrow> t = (\textit{if } t \leq m \textit{ then } \{m<..<n\} \textit{ else } \{t<..<n\})$ **and**
atLeastLessThan-cut-le: $\{m..<n\} \downarrow\leq t = (\textit{if } n \leq t \textit{ then } \{m..<n\} \textit{ else } \{m..t\})$ **and**
atLeastLessThan-cut-less: $\{m..<n\} \downarrow< t = (\textit{if } n \leq t \textit{ then } \{m..<n\} \textit{ else } \{m..<t\})$ **and**
atLeastLessThan-cut-ge: $\{m..<n\} \downarrow\geq t = (\textit{if } t \leq m \textit{ then } \{m..<n\} \textit{ else } \{t..<n\})$ **and**
atLeastLessThan-cut-greater: $\{m..<n\} \downarrow> t = (\textit{if } t < m \textit{ then } \{m..<n\} \textit{ else } \{t<..<n\})$ **and**
greaterThanAtMost-cut-le: $\{m<..n\} \downarrow\leq t = (\textit{if } n \leq t \textit{ then } \{m<..n\} \textit{ else } \{m<..t\})$ **and**
greaterThanAtMost-cut-less: $\{m<..n\} \downarrow< t = (\textit{if } n < t \textit{ then } \{m<..n\} \textit{ else } \{m<..<t\})$ **and**
greaterThanAtMost-cut-ge: $\{m<..n\} \downarrow\geq t = (\textit{if } t \leq m \textit{ then } \{m<..n\} \textit{ else } \{t..n\})$ **and**
greaterThanAtMost-cut-greater: $\{m<..n\} \downarrow> t = (\textit{if } t \leq m \textit{ then } \{m<..n\} \textit{ else } \{t<..n\})$ **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 if-split linorder-not-le linorder-not-less*)

apply *fastforce+*
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* ⇒ *nat* ⇒ *nat* ⇒ *nat*
  where *nat-mirror n l r* ≡ *l* + *r* − *n*

lemma *nat-mirror-commute*: *nat-mirror n l r* = *nat-mirror n r l*
unfolding *nat-mirror-def* **by** *simp*

lemma *nat-mirror-inj-on*: *inj-on* (λ*x*. *nat-mirror x l r*) {..*l* + *r*}
unfolding *inj-on-def nat-mirror-def* **by** *fastforce*

lemma *nat-mirror-nat-mirror-ident*:
  n ≤ *l* + *r* ⟹ *nat-mirror* (*nat-mirror n l r*) *l r* = *n*
unfolding *nat-mirror-def* **by** *simp*

lemma *nat-mirror-add*:
  nat-mirror (*n* + *k*) *l r* = (*nat-mirror n l r*) − *k*
unfolding *nat-mirror-def* **by** *simp*
lemma *nat-mirror-diff*:
  ⟦ *k* ≤ *n*; *n* ≤ *l* + *r* ⟧ ⟹
  nat-mirror (*n* − *k*) *l r* = (*nat-mirror n l r*) + *k*
unfolding *nat-mirror-def* **by** *simp*

lemma *nat-mirror-le*: *a* ≤ *b* ⟹ *nat-mirror b l r* ≤ *nat-mirror a l r*
unfolding *nat-mirror-def* **by** *simp*
lemma *nat-mirror-le-conv*:
  a ≤ *l* + *r* ⟹ (*nat-mirror b l r* ≤ *nat-mirror a l r*) = (*a* ≤ *b*)
unfolding *nat-mirror-def* **by** *fastforce*

lemma *nat-mirror-less*:
  ⟦ *a* < *b*; *a* < *l* + *r* ⟧ ⟹
  nat-mirror b l r < *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** *fastforce*

lemma *nat-mirror-eq-conv*:

⟦ $a \leq l + r$; $b \leq l + r$ ⟧ ⟹
($nat\text{-}mirror\ a\ l\ r = nat\text{-}mirror\ b\ l\ r$) = ($a = b$)
unfolding *nat-mirror-def* **by** *fastforce*

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* (λ*x*. *mirror-elem x I*) *I*
unfolding *mirror-elem-def*
by (*metis Max-le-imp-subset-atMost nat-mirror-inj-on not-add-less2 not-le-imp-less 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*

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*)

lemma *mirror-elem-eq-conv*:
⟦ $a \leq iMin\ I + Max\ I$; $b \leq iMin\ I + Max\ I$ ⟧ ⟹
($mirror\text{-}elem\ a\ I = mirror\text{-}elem\ b\ I$) = ($a = b$)
by (*simp add*: *mirror-elem-def nat-mirror-eq-conv*)
lemma *mirror-elem-eq-conv'*:
⟦ *finite I*; $a \in I$; $b \in I$ ⟧ ⟹ ($mirror\text{-}elem\ a\ I = mirror\text{-}elem\ b\ I$) = ($a = b$)
apply (*rule mirror-elem-eq-conv*)
apply (*simp-all add*: *trans-le-add2*)
done

definition
imirror-bounds :: *nat set* ⇒ *nat* ⇒ *nat* ⇒ *nat set*
where
$imirror\text{-}bounds\ I\ l\ r \equiv (\lambda x.\ nat\text{-}mirror\ x\ l\ r)\ `\ I$

Mirroring all elements between the interval boundaries of I

definition
imirror :: *nat set* ⇒ *nat set*
where
$imirror\ I \equiv (\lambda x.\ iMin\ I + Max\ I - x)\ `\ I$

lemma *imirror-eq-nat-mirror-image*:
$imirror\ I = (\lambda x.\ nat\text{-}mirror\ x\ (iMin\ I)\ (Max\ I))\ `\ I$
unfolding *imirror-def nat-mirror-def* **by** *simp*
lemma *imirror-eq-mirror-elem-image*:
$imirror\ I = (\lambda x.\ mirror\text{-}elem\ x\ I)\ `\ I$
by (*simp add*: *mirror-elem-def imirror-eq-nat-mirror-image*)

lemma *imirror-eq-imirror-bounds*:
$imirror\ I = imirror\text{-}bounds\ I\ (iMin\ I)\ (Max\ I)$
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* ⟹ *finite* (*imirror I*)

unfolding *imirror-def* **by** *simp*

lemma *imirror-bounds-iMin*:
⟦ *finite I*; $I \neq \{\}$; *iMin I* $\leq l + r$ ⟧ ⟹
iMin (*imirror-bounds I l r*) = $l + r -$ *Max I*
apply (*unfold imirror-bounds-def nat-mirror-def*)
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* $\leq l + r$ ⟧ ⟹
Max (*imirror-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* ⟹ *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* ⟹ *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* ⟧ ⟹ *iMin I* $\leq n \wedge n \leq$ *Max I*
apply (*frule Max-ge*[*OF imirror-finite*, *of - n*], *assumption*)
apply (*fastforce simp*: *imirror-iMin imirror-Max*)
done

lemma *imirror-bounds-iff*:
($n \in$ *imirror-bounds I l r*) = ($\exists\, x{\in}I.\ n = l + r - x$)
by (*simp add*: *imirror-bounds-def nat-mirror-def image-iff*)

lemma *imirror-iff*: ($n \in$ *imirror I*) = ($\exists\, x{\in}I.\ n =$ *iMin I* + *Max I* $- x$)
by (*simp add*: *imirror-def image-iff*)

lemma *imirror-bounds-imirror-bounds-ident*:
⟦ *finite I*; *Max I* $\leq l + r$ ⟧ ⟹
imirror-bounds (*imirror-bounds I l r*) *l r* = *I*
apply (*rule set-eqI*)
apply (*simp add*: *imirror-bounds-def image-image image-iff*)
apply (*rule iffI*)

apply (*fastforce simp*: *nat-mirror-nat-mirror-ident*)
apply (*rule-tac x*=*x* **in** *bexI*)
apply (*fastforce simp*: *nat-mirror-nat-mirror-ident*)+
done

lemma *imirror-imirror-ident*: *finite I* $\Longrightarrow$ *imirror* (*imirror I*) = *I*
apply (*case-tac I* = {}, *simp add*: *imirror-empty*)
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 $\Longrightarrow$ *mirror-elem t* (*imirror I*) = *mirror-elem t I*
by (*simp add*: *mirror-elem-def imirror-iMin imirror-Max*)

lemma *imirror-card*: *finite I* $\Longrightarrow$ *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* ⟧ $\Longrightarrow$
((*nat-mirror n l r*) ∈ *imirror-bounds I l r*) = (*n* ∈ *I*)
apply (*unfold imirror-bounds-def*)
apply (*rule inj-on-image-mem-iff*)
apply (*rule nat-mirror-inj-on*)
apply *fastforce*
apply *simp*
done

lemma *imirror-mem-conv*:
⟦ *finite I*; *n* ≤ *iMin I* + *Max I* ⟧ $\Longrightarrow$ ((*mirror-elem n I*) ∈ *imirror I*) = (*n* ∈ *I*)
by (*simp add*: *mirror-elem-def imirror-eq-imirror-bounds imirror-bounds-elem-conv*)

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

lemma *imirror-cut-less*:
finite I $\Longrightarrow$
imirror I ↓< (*mirror-elem t I*) =
imirror-bounds (*I* ↓> *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*)

apply (*bestsimp intro*: *nat-mirror-less-imp-less*)
apply (*bestsimp simp add*: *nat-mirror-less*)
done

lemma *imirror-cut-le*:
⟦ *finite I*; *t* ≤ *iMin I* + *Max I* ⟧ ⟹
imirror I ↓≤ (*mirror-elem t I*) =
imirror-bounds (*I* ↓≥ *t*) (*iMin I*) (*Max I*)
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*])
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 ↓≥ (*mirror-elem t I*) =
imirror-bounds (*I* ↓≤ *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*])
apply (*simp add*: *imirror-imirror-ident mirror-elem-imirror mirror-elem-mirror-elem-ident*)
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*)
apply (*simp add*: *nat-cut-ge-greater-conv*[*symmetric*])
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 (*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

corollary *imirror-cut-le′*:
⟦ *finite I*; *t* ∈ *I* ⟧ ⟹
imirror I ↓≤ *mirror-elem t I* =
imirror-bounds (*I* ↓≥ *t*) (*iMin I*) (*Max I*)
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′

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*)
apply (*rule inj-on-image-Int*[*OF - Un-upper1 Un-upper2*])
apply (*rule subset-inj-on*[*OF nat-mirror-inj-on*])
apply (*rule Un-least*[*of A - B*], *assumption*+)
done

end

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

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

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** (*fastforce dest*: *Max-ge*)
lemma *iprev-iMin*: *iprev* (*iMin I*) *I* = *iMin I*
unfolding *iprev-def cut-less-def* **by** *fastforce*

lemma *inext-ge-Max*: ⟦ *finite I*; *Max I* ≤ *n* ⟧ ⟹ *inext n I* = *n*

unfolding *inext-def cut-greater-def* **by** (*fastforce dest*: *Max-ge*)

lemma *iprev-le-iMin*: $n \leq iMin\ I \Longrightarrow iprev\ n\ I = n$
unfolding *iprev-def cut-less-def* **by** *fastforce*

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

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

lemma *inext-closed*: $n \in I \Longrightarrow inext\ n\ I \in I$
apply (*clarsimp simp*: *inext-def*)
apply (*rule subsetD*[*OF cut-greater-subset*])
apply (*rule iMinI-ex2*, *assumption*)
done

lemma *iprev-closed*: $n \in I \Longrightarrow iprev\ n\ I \in I$
apply (*clarsimp simp*: *iprev-def*)
apply (*rule subsetD*[*of* $I\ {\downarrow_<}\ n$], *fastforce*)
by (*rule Max-in*[*OF nat-cut-less-finite*])

lemma *inext-in-imp-in*: $inext\ n\ I \in I \Longrightarrow n \in I$
by (*case-tac* $n \in I$, *simp-all add*: *not-in-inext-fix*)

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

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

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* ∈ *I*) = (*n* ∈ *I*)
apply (*rule iffI*)
apply (*rule iprev-in-imp-in*, *assumption*)
apply (*rule iprev-closed*, *assumption*)
done

lemma *subset-iprev-closed*: ⟦ *n* ∈ *B*; *A* ⊆ *B* ⟧ ⟹ *iprev n A* ∈ *B*
apply (*case-tac n* ∈ *A*)
apply (*fastforce simp*: *iprev-closed*)
apply (*simp add*: *not-in-iprev-fix*)
done

lemma *subset-iprev-in-imp-in*: ⟦ *iprev n A* ∈ *B*; *A* ⊆ *B* ⟧ ⟹ *n* ∈ *B*
apply (*case-tac n* ∈ *A*)
apply *fastforce*
apply (*simp add*: *not-in-iprev-fix*)
done

lemma *subset-iprev-in-iff*: *A* ⊆ *B* ⟹ (*iprev n A* ∈ *B*) = (*n* ∈ *B*)
apply (*rule iffI*)
apply (*rule subset-iprev-in-imp-in*, *assumption*+)
apply (*rule subset-iprev-closed*, *assumption*+)
done

lemma *inext-mono*: *n* ≤ *inext n I*
by (*simp add*: *inext-def i-cut-defs iMin-ge-iff*)

corollary *inext-neq-imp-less*: *n* ≠ *inext n I* ⟹ *n* < *inext n I*
by (*insert inext-mono*[*of n I*], *simp*)

lemma *inext-mono2*: ⟦ *n* ∈ *I*; ∃ *x*∈*I*. *n* < *x* ⟧ ⟹ *n* < *inext n I*
by (*fastforce simp add*: *inext-def i-cut-defs iMin-gr-iff*)

lemma *inext-mono2-infin*: ⟦ *n* ∈ *I*; *infinite I* ⟧ ⟹ *n* < *inext n I*
apply (*simp add*: *inext-def i-cut-defs iMin-gr-iff*)
apply (*fastforce simp*: *infinite-nat-iff-unbounded*)
done

lemma *inext-mono2-fin*: ⟦ *n* ∈ *I*; *finite I*; *n* ≠ *Max I* ⟧ ⟹ *n* < *inext n I*
apply (*simp add*: *inext-def i-cut-defs iMin-gr-iff*)
apply (*blast intro*: *Max-ge Max-in*)
done

lemma *inext-mono2-infin-fin*:
⟦ *n* ∈ *I*; *n* ≠ *Max I* ∨ *infinite I* ⟧ ⟹ *n* < *inext n I*
by (*blast intro*: *inext-mono2-infin inext-mono2-fin*)

lemma *inext-neq-iMin*: $\exists\, x{\in}I.\ n < x \Longrightarrow inext\ n\ I \neq iMin\ I$
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*)
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*)
apply (*blast dest*: *infinite-nat-iff-unbounded*[*THEN iffD1*])
done

lemma *Max-le-iMin-imp-singleton*: $[\![\ finite\ I;\ I \neq \{\};\ Max\ I \leq iMin\ I\ ]\!] \Longrightarrow I = \{iMin\ I\}$
by (*simp add*: *iMin-Min-conv Max-le-Min-imp-singleton*)

lemma *inext-neq-iMin-not-singleton*:
$[\![\ I \neq \{\};\ \neg(\exists\, a.\ I = \{a\})\ ]\!] \Longrightarrow inext\ n\ I \neq iMin\ I$
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*:
$[\![\ I \neq \{\};\ card\ I \neq Suc\ 0\ ]\!] \Longrightarrow inext\ n\ I \neq iMin\ I$
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*: $[\![\ n < k;\ k < inext\ n\ I\ ]\!] \Longrightarrow k \notin I$
apply (*case-tac* $n \in I$)
prefer *2*
apply (*simp add*: *inext-def*)

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

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

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*)
apply (*blast dest*: *inext-min-step*)
by (*rule inext-mono2*)

lemma *inext-less-mono-fin*:
⟦ $a < b$; $a \in I$; $b \in I$; *finite* I; $b \neq$ *Max* I ⟧ ⟹ *inext* a I < *inext* b I
by (*blast intro*: *inext-less-mono Max-in*)

lemma *inext-less-mono-infin*:
⟦ $a < b$; $a \in I$; $b \in I$; *infinite* I ⟧ ⟹ *inext* a I < *inext* b I
apply (*rule inext-less-mono*, *assumption*+)
apply (*blast dest*: *infinite-imp-asc-chain*)
done

lemma *inext-less-mono-infin-fin*:
⟦ $a < b$; $a \in I$; $b \in I$; $b \neq$ *Max* $I \vee$ *infinite* I ⟧ ⟹ *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 $I < x$ ⟧ ⟹ $a \leq b$
apply (*rule ccontr*, *simp add*: *linorder-not-le*)
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⟧ ⟹ $a \leq b$
by (*metis inext-in-iff inext-le-mono-rev inext-mono2-infin-fin*)

lemma *inext-le-mono-infin-rev*:
⟦ *inext* a $I \leq$ *inext* b I; $a \in I$; $b \in I$; *infinite* I ⟧ ⟹ $a \leq b$
by (*metis inext-in-iff inext-le-mono-rev inext-mono2-infin-fin*)

lemma *inext-le-mono-infin-fin-rev*:
⟦ *inext* a $I \leq$ *inext* b I; $a \in I$; $b \in I$; *inext* a $I \neq$ *Max* $I \vee$ *infinite* I ⟧ ⟹ $a \leq b$
by (*blast intro*: *inext-le-mono-infin-rev inext-le-mono-fin-rev*)

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

lemma *less-imp-inext-le*: ⟦ $a < b$; $a \in I$; $b \in I$ ⟧ ⟹ *inext* a $I \leq b$
by (*metis inext-min-step not-le*)

lemma *iprev-mono*: *iprev* n $I \leq n$
unfolding *iprev-def i-cut-defs* **by** *simp*
corollary *iprev-neq-imp-greater*: $n \neq$ *iprev* n $I \Longrightarrow$ *iprev* n $I < n$
by (*insert iprev-mono*[*of n I*], *simp*)

lemma *iprev-mono2*: ⟦ $n \in I$; $\exists x{\in}I.$ $x < n$⟧ ⟹ *iprev* n $I < n$
apply (*unfold iprev-def i-cut-defs*, *clarsimp*)

apply (*blast intro*: *finite-nat-iff-bounded*)+
done

lemma *iprev-mono2-if-neq-iMin*: ⟦ $n \in I$; *iMin* $I \neq n$⟧ ⟹ *iprev* n $I < n$
by (*blast intro*: *iMinI iprev-mono2*)

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

lemma *iprev-neq-Max-not-singleton*:
⟦ *finite* I; $I \neq \{\}$; ¬(∃ a. $I = \{a\}$) ⟧ ⟹ *iprev* n $I \neq$ *Max* I
apply (*case-tac* $n \in I$)
prefer *2*
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; $I \neq \{\}$; *card* $I \neq$ *Suc 0* ⟧ ⟹ *iprev* n $I \neq$ *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 $I \neq n$ ⟹ *iMin* $I < n$
by (*rule ccontr*, *clarsimp simp*: *iprev-le-iMin*)

lemma *iprev-greater-conv*: ($n \in I \wedge$ *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*: ($n \notin I \vee$ (*finite* $I \wedge$ *Max* $I = n$)) = (*inext* n $I = n$)
apply (*case-tac* $n \notin I$, *simp add*: *not-in-inext-fix*)
by (*metis inext-Max inext-min-step2 inext-mono2-infin-fin*)

lemma *iprev-fix-iff*: ($n \notin I \vee$ *iMin* $I = n$) = (*iprev* n $I = n$)
apply (*case-tac* $n \notin I$, *simp add*: *not-in-iprev-fix*)
by (*metis iprev-iMin iprev-mono2-if-neq-iMin less-not-refl3*)

lemma *iprev-min-step*: $[\![\ \mathit{iprev}\ n\ I < k;\ k < n\ ]\!] \Longrightarrow k \notin I$
apply (*case-tac* $n \in I$)
prefer *2*
apply (*simp add*: *iprev-def*)
apply (*rule contrapos-pn*[*of iprev n I* $<$ *k k* $\in$ *I*], *simp*)
apply (*unfold iprev-def i-cut-defs*, *simp*)
apply (*split if-split-asm*)
apply (*cut-tac Max-ge*[*of* $\{x \in I.\ x < n\}$ *k*])
apply *fastforce+*
done

corollary *iprev-min-step2*: $\neg(\exists\, x{\in}I.\ \mathit{iprev}\ n\ I < x \wedge x < n)$
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 $= x$
apply (*rule ccontr*)
apply (*simp add*: *nat-neq-iff*, *elim disjE*)
apply (*simp add*: *iprev-min-step*)
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*:
$[\![\ n \in A;\ \mathit{iprev}\ n\ A < n;\ n \in B;\ \mathit{iprev}\ n\ A \in B;\ B \downarrow> (\mathit{iprev}\ n\ A) \downarrow< n = \{\}\ ]\!]$
$\Longrightarrow$
iprev n B $=$ *iprev n A*
by (*blast intro*: *min-step-iprev2*)

lemma *iprev-le-mono*: $[\![\ a \leq b;\ a \in I;\ b \in I\ ]\!] \Longrightarrow \mathit{iprev}\ a\ I \leq \mathit{iprev}\ b\ I$
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*)
by (*blast dest*: *iprev-min-step*)

lemma *iprev-less-mono*:
$[\![\ a < b;\ a \in I;\ b \in I;\ \exists\, x{\in}I.\ x < a\ ]\!] \Longrightarrow \mathit{iprev}\ a\ I < \mathit{iprev}\ b\ I$
apply (*rule less-le-trans*[*of - a*])
apply (*blast intro*: *iprev-mono2*)

apply (*rule ccontr*, *simp add*: *linorder-not-le*)
by (*blast dest*: *iprev-min-step*)

lemma *iprev-less-mono-if-neq-iMin*:
⟦ $a < b$; $a \in I$; $b \in I$; $iMin\ I \neq a$ ⟧ ⟹ $iprev\ a\ I < iprev\ b\ I$
by (*metis iprev-in-iff iprev-less-mono iprev-mono2-if-neq-iMin*)

lemma *iprev-le-mono-rev*:
⟦ $iprev\ a\ I \leq iprev\ b\ I$; $a \in I$; $b \in I$; $iMin\ I \neq iprev\ b\ I$ ⟧ ⟹ $a \leq b$
apply (*rule ccontr*, *simp add*: *linorder-not-le*)
by (*metis iprev-fix-iff iprev-less-mono-if-neq-iMin less-le-not-le*)

lemma *iprev-less-mono-rev*:
⟦ $iprev\ a\ I < iprev\ b\ I$; $a \in I$; $b \in I$ ⟧ ⟹ $a < b$
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*; $n \in$ *interval-fun* I; $inext\ n\ I \in$ *interval-fun* I ⟧ ⟹
$inext\ n\ (interval\text{-}fun\ I) = inext\ n\ I$
apply (*subgoal-tac* $n \in I$)
prefer *2*
apply (*blast intro*: *set-restriction-in-imp*)
apply (*case-tac* $inext\ n\ I = n$)
apply *simp*
apply (*frule inext-fix-iff*[*THEN iffD2*], *clarsimp*)
apply (*frule set-restriction-finite*, *assumption*)
apply (*subgoal-tac Max* $(interval\text{-}fun\ I) = Max\ I$)
prefer *2*
apply (*blast intro*: *Max-equality Max-ge set-restriction-in-imp*)
apply (*blast intro*: *inext-fix-iff*[*THEN iffD1*])
apply (*drule le-neq-implies-less*[*OF inext-mono*, *OF not-sym*])
apply (*rule between-empty-imp-inext-eq*, *assumption*+)
apply (*simp add*: *not-ex-in-conv*[*symmetric*] *i-cut-mem-iff*)
by (*metis inext-min-step2 set-restriction-in-imp*)

lemma *set-restriction-inext-singleton-eq*:
⟦ *set-restriction interval-fun*; $n \in$ *interval-fun* I; $inext\ n\ I \in$ *interval-fun* I ⟧ ⟹
$\{inext\ n\ (interval\text{-}fun\ I)\} = interval\text{-}fun\ \{inext\ n\ I\}$
apply (*case-tac* $n \notin I$)
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*)
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*
apply (*drule arg-cong*[**where** *f*=λx. *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*=λx. *iprev x I*])
apply (*simp add*: *iprev-inext-fin*)
done

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*
by (*blast intro*: *inext-eq-fin inext-eq-infin*)+

lemma *inext-eq*:
⟦ *inext a I* = *inext b I*; $\exists$ *x*$\in$*I*. *a* < *x*; $\exists$ *x*$\in$*I*. *b* < *x* ⟧ $\Longrightarrow$ *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* $\neq$ *a*; *iMin I* $\neq$ *b* ⟧ $\Longrightarrow$ *a* = *b*
apply (*drule arg-cong*[**where** *f*=λx. *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*
apply (*frule le-less-trans*[*OF inext-mono*])
apply (*case-tac n* ∈ *I*)
 apply (*simp add*: *inext-def*)
 apply (*simp add*: *i-cut-commute-disj*[*of* (↓<) (↓>)] *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*
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*)
 apply (*frule cut-greater-mem-iff*[*THEN iffD2*, *OF conjI*], *simp*)
 apply (*simp add*: *inext-def i-cut-commute-disj*[*of* (↓>) (↓>)] *cut-cut-greater max-def*)
apply (*simp add*: *not-in-inext-fix cut-greater-not-in-imp*)
done

lemma *inext-cut-ge-conv*: *t* ≤ *n* ⟹ *inext n* (*I* ↓≥ *t*) = *inext n I*
apply (*case-tac t* = *0*)
 apply (*simp add*: *cut-ge-0-all*)
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* ⟹ *iprev n* (*I* ↓> *t*) = *iprev n I*
apply (*frule less-le-trans*[*OF* - *iprev-mono*])
apply (*case-tac n* ∈ *I*)

apply (*simp add*: *iprev-def*)
apply (*simp add*: *i-cut-commute-disj*[*of* (↓>) (↓<)] *cut-greater-mem-iff*)
apply (*case-tac I* ↓< *n* ≠ {})
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* ≤ *iprev n I* ⟹ *iprev n* (*I* ↓≥ *t*) = *iprev n I*
apply (*case-tac t* = *0*)
apply (*simp add*: *cut-ge-0-all*)
apply (*simp add*: *nat-cut-greater-ge-conv*[*symmetric*] *iprev-cut-greater-conv*)
done

lemma *iprev-cut-less-conv*: *n* < *t* ⟹ *iprev n* (*I* ↓< *t*) = *iprev n I*
apply (*case-tac n* ∈ *I*)
apply (*frule cut-less-mem-iff*[*THEN iffD2*, *OF conjI*], *simp*)
apply (*simp add*: *iprev-def i-cut-commute-disj*[*of* (↓<) (↓<)] *cut-cut-less min-def*)
apply (*simp add*: *not-in-iprev-fix cut-less-not-in-imp*)
done

lemma *iprev-cut-le-conv*: *n* ≤ *t* ⟹ *iprev n* (*I* ↓≤ *t*) = *iprev n I*
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

lemma *inext-cut-less-fix*: *t* ≤ *inext n I* ⟹ *inext n* (*I* ↓< *t*) = *n*
apply (*case-tac n* ∈ *I*)
prefer *2*
apply (*frule contra-subsetD*[*OF cut-less-subset*[*of* - *t*]])
apply (*simp add*: *not-in-inext-fix*)
apply (*case-tac t* ≤ *n*)
apply (*metis cut-less-mem-iff not-in-inext-fix not-le*)
apply (*rule-tac t*=*n* **and** *s*=*Max* (*I* ↓< *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*)
apply (*simp add*: *inext-min-step*)
apply (*blast intro*: *inext-Max nat-cut-less-finite*)
done

lemma *inext-cut-le-fix*: *t* < *inext n I* ⟹ *inext n* (*I* ↓≤ *t*) = *n*
by (*simp add*: *nat-cut-le-less-conv inext-cut-less-fix*)

lemma *iprev-cut-greater-fix*: *iprev n I* ≤ *t* ⟹ *iprev n* (*I* ↓> *t*) = *n*

apply (*case-tac* $n \in I$)
prefer *2*
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-le-imp-less order-le-less-trans*)
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*)
apply (*simp add*: *nat-cut-greater-ge-conv*[*symmetric*] *iprev-cut-greater-fix*)
done

definition
$CommuteWithIntervalCut4 :: ((\,'a{::}linorder)\ set \Rightarrow\ 'a\ set) \Rightarrow bool$
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 $CommuteWithIntervalCut2 :: ((\,'a{::}linorder)\ set \Rightarrow\ 'a\ set) \Rightarrow bool$
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* $(\lambda I.\ I \downarrow\geq t)$
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

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

lemma *iprev-image*:
$[\![\ n \in I;\ \textit{strict-mono-on}\ f\ I\ ]\!] \Longrightarrow \textit{iprev}\ (f\ n)\ (f\ `\ I) = f\ (\textit{iprev}\ n\ I)$
apply (*case-tac* $\exists\, x{\in}I.\ x < n$)
apply (*frule iprev-mono2*, *assumption*)
apply (*frule cut-less-not-empty-iff*[*THEN iffD2*])
apply (*simp add*: *iprev-def image-iff*)
apply (*subgoal-tac* $\exists\, x{\in}I.\ f\ n = f\ x$)
prefer *2*
apply *blast*
apply (*simp add*: *cut-less-image*)
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*)
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*)
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*
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*)
apply (*simp add*: *cut-less-Min-empty mirror-elem-Min*)
apply (*subst imirror-Max*[*symmetric*], *assumption*)
apply (*simp add*: *cut-greater-Max-empty imirror-finite*)
apply (*frule iMin-le*[*of n I*])
apply (*intro conjI impI*)
apply (*simp add*: *imirror-cut-greater′*)
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*)
apply (*simp add*: *cut-less-Min-not-empty*)
done

corollary *inext-imirror-iprev-conv′*:
⟦ *finite I*; *n* $\in$ *I* ⟧ $\Longrightarrow$
inext (*mirror-elem n I*) (*imirror I*) = *mirror-elem* (*iprev n I*) *I*
by (*simp add*: *inext-imirror-iprev-conv trans-le-add2*)

lemma *iprev-imirror-inext-conv*:
⟦ *finite I*; *n* $\leq$ *iMin I* + *Max I* ⟧ $\Longrightarrow$
iprev (*mirror-elem n I*) (*imirror I*) = *mirror-elem* (*inext n I*) *I*
apply (*case-tac n* $\in$ *I*)

prefer *2*
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*)
apply (*simp add*: *cut-greater-Max-empty mirror-elem-Max*)
apply (*subst imirror-iMin*[*symmetric*], *assumption*)
apply (*simp add*: *cut-less-Min-empty imirror-finite*)
apply (*frule Max-ge*[*of I n*], *assumption*)
apply (*drule le-neq-trans*, *assumption*)
apply (*intro conjI impI*)
apply (*simp add*: *imirror-cut-less*)
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*)
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*)

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*)
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*)
apply (*drule le-neq-trans*, *simp*)
apply (*rule min-step-iprev2*)
apply (*simp-all add*: *iMin-Min-conv*, *blast?*)
done

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 if-split-asm*)
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*:
⟦ $t \in I$; $t \neq Max\ I \vee infinite\ I$ ⟧ ⟹ $inext\ t\ I \neq iMin\ I$
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*)
apply (*blast dest*: *inext-closed*)
apply *simp*
done

lemma *cut-le-less-inext-conv*:
⟦ $t \in I$; $t \neq Max\ I \vee infinite\ I$⟧ ⟹ $I \downarrow\leq t = I \downarrow< (inext\ t\ I)$
apply (*cut-tac cut-less-le-iprev-conv*[*of inext t I 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*, *fastforce*)
apply (*split if-split-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*:
⟦ $t \in I$; $t \neq Max\ I \vee infinite\ I$ ⟧ ⟹ $I \downarrow> t = I \downarrow\geq (inext\ t\ I)$

apply (*cut-tac cut-ge-greater-iprev-conv*[*of inext t I 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

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 *fastforce*
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 *fastforce*
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 *fastforce*
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*)
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*)
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)
  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 ∉ (I ↓> a ↓< b))
 prefer 2
 apply blast
apply (subgoal-tac (I ↓≥ a ↓< b) = insert a (I ↓> a ↓< b))
 prefer 2
 apply (simp add:
   i-cut-commute-disj[of (↓≥) (↓<)] i-cut-commute-disj[of (↓>) (↓<)])
 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 ↓> a ↓< 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 ≤ b; a ∈ I; b ∈ I; ¬ P a; P b ⟧ ⟹
  ∃ n∈I. a ≤ n ∧ n < b ∧ ¬ P n ∧ P (inext n I)
apply (drule order-le-less[THEN iffD1], erule disjE)
 prefer 2
 apply blast
apply (drule inext-predicate-change-exists-aux[OF refl], assumption+)
apply blast
done

lemma iprev-predicate-change-exists:
  ⟦ a ≤ b; a ∈ I; b ∈ I; ¬ P b; P a ⟧ ⟹
  ∃ n∈I. a < n ∧ n ≤ b ∧ ¬ P n ∧ P (iprev n I)
apply (frule inext-predicate-change-exists[of a b I λx. ¬ 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*:
⟦ $a \leq b$; $\neg\ P\ a$; $P\ b$ ⟧ $\Longrightarrow$ $\exists\, n{\geq}a.\ n < b \wedge \neg\ P\ n \wedge P\ (Suc\ n)$
apply (*drule inext-predicate-change-exists*[*OF - UNIV-I UNIV-I*], *assumption+*)
apply (*simp add*: *inext-UNIV*)
done

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

lemma *inext-predicate-change-exists2-all*:
⟦ $(a{::}nat) \leq b$; $a \in I$; $b \in I$; $\neg\ P\ a$; $\forall\, k \in I\ {\downarrow}{\geq}\ b.\ P\ k$ ⟧ $\Longrightarrow$
$\exists\, n{\in}I.\ a \leq n \wedge n < b \wedge \neg\ P\ n \wedge (\forall\, k \in I\ {\downarrow}{>}\ n.\ P\ k)$
apply (*drule order-le-less*[*THEN iffD1*], *erule disjE*)
prefer *2*
apply *blast*
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 if-split-asm*)
apply (*subgoal-tac* $I\ {\downarrow}{>}\ n = \{\}$, *simp+*)
apply (*drule not-sym*)
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*)
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*
apply (*simp add*: *cut-greater-ge-conv-if*)
apply *blast*
done

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

lemma *iprev-predicate-change-exists2-all*:
$[\![(a::nat) \leq b; a \in I; b \in I; \neg\ P\ b; \forall\, k{\in}I{\downarrow}{\leq}a.\ P\ k]\!] \Longrightarrow$
$\exists\, n{\in}I.\ a < n \wedge n \leq b \wedge \neg\ P\ n \wedge (\forall\, k{\in}I{\downarrow}{<}n.\ P\ k)$
apply (*drule order-le-less*[*THEN iffD1*], *erule disjE*)
prefer *2*
apply *blast*
apply (*frule iprev-predicate-change-exists*[*OF less-imp-le*,
of a b I $\lambda n.$ *if* $(n = b)$ *then P n else* $(\forall\, k{\in}I{\downarrow}{\leq}n.\ P\ k)$])

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 if-split-asm*)
apply *clarsimp*
apply (*split if-split-asm*)
apply *simp*
apply (*simp add*: *cut-less-le-iprev-conv*[*symmetric*])
apply *blast*
apply (*split if-split-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

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*)
apply (*frule iprev-predicate-change-exists2-all*[*of a b I* ↓≥ *a*])
apply (*simp add*: *i-cut-mem-iff*)+
apply *fastforce*
apply *blast*
done

corollary *nat-pred-predicate-change-exists2-all*:
⟦ (*a*::*nat*) ≤ *b*; ¬ *P b*; ∀ *k*≤*a*. *P k* ⟧ ⟹
∃ *n*>*a*. *n* ≤ *b* ∧ ¬ *P n* ∧ (∀ *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

corollary *nat-pred-predicate-change-exists2*:
⟦ (*a*::*nat*) ≤ *b*; ¬ *P b*; *P a* ⟧ ⟹
∃ *n*>*a*. *n* ≤ *b* ∧ ¬ *P n* ∧ (∀ *k*≥*a*. *k* < *n* ⟶ *P k*)
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

primrec *inext-nth* :: *nat set* ⇒ *nat* ⇒ *nat* (‹(- → -)› [*100*, *100*] *60*)
where
  I → *0* = *iMin I*
| *I* → *Suc n* = *inext* (*inext-nth I n*) *I*

lemma *inext-nth-closed*: *I* ≠ {} ⟹ *I* → *n* ∈ *I*
apply (*induct n*)
 apply (*simp add*: *iMinI-ex2*)
apply (*simp add*: *inext-closed*)
done

lemma *inext-nth-image*:
  ⟦ *I* ≠ {}; *strict-mono-on f I* ⟧ ⟹ (*f* ‘ *I*) → *n* = *f* (*I* → *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

lemma *inext-nth-Suc-mono*: *I* → *n* ≤ *I* → *Suc n*
by (*simp add*: *inext-mono*)

lemma *inext-nth-mono*: *a* ≤ *b* ⟹ *I* → *a* ≤ *I* → *b*
apply (*induct b*)
 apply *simp*
apply (*drule le-Suc-eq*[*THEN iffD1*], *erule disjE*)
apply (*rule-tac y*=*I* → *b* **in** *order-trans*)
 apply *simp*
 apply (*rule inext-nth-Suc-mono*)
apply *simp*
done

lemma *inext-nth-Suc-mono2*: ∃ *x*∈*I*. *I* → *n* < *x* ⟹ *I* → *n* < *I* → *Suc n*
apply *simp*
apply (*rule inext-mono2*)
apply (*blast intro*: *inext-nth-closed inext-mono2*)+
done

lemma *inext-nth-mono2*: ∃ *x*∈*I*. *I* → *a* < *x* ⟹ (*I* → *a* < *I* → *b*) = (*a* < *b*)
apply (*subgoal-tac I* ≠ {})
 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 $\Longrightarrow$ $(I \to a < I \to b) = (a < b)$
apply (*drule infinite-nat-iff-unbounded*[*THEN iffD1*])
apply (*rule inext-nth-mono2*)
apply *blast*
done

lemma *inext-nth-Max-fix*:
$[\![$ *finite I*; $I \neq \{\}$; $I \to a = Max\ I$; $a \leq b$ $]\!] \Longrightarrow I \to b = Max\ I$
apply (*induct b*)
apply *simp*
apply (*drule le-Suc-eq*[*THEN iffD1*], *erule disjE*)
apply (*simp add*: *inext-Max*)
apply *blast*
done

lemma *inext-nth-cut-less-conv*:
$\bigwedge I.\ I \to n < t \Longrightarrow (I \downarrow< t) \to n = I \to n$
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*
apply (*frule order-le-less-trans*[*OF inext-mono*])
apply (*simp add*: *inext-cut-less-conv*)
done

lemma *remove-Min-inext-nth-Suc-conv*: $\bigwedge I.$
Suc 0 $<$ *card I* $\vee$ *infinite I* $\Longrightarrow$
$(I - \{iMin\ I\}) \to n = I \to Suc\ n$

apply (*subgoal-tac* $I \neq \{\}$)
prefer *2*
apply (*blast dest*: *card-gr0-imp-not-empty*[*OF gr-implies-gr0*])
apply (*subgoal-tac* $I - \{iMin\ I\} \neq \{\}$)
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*)
apply (*simp add*: *cut-greater-Min-eq-Diff*[*symmetric*] *inext-def iMinI-ex2*)
apply *simp*
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*)
apply (*simp add*: *inext-neq-iMin-not-card-1 inext-neq-iMin-infin*)+
apply (*subgoal-tac iMin I* < (*I* → *Suc n*))
prefer *2*
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*
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*)
apply (*drule Max-le-Min-conv-singleton*[*THEN iffD1*], *assumption*+)
apply *clarsimp*
apply (*drule-tac x*=*I* − {*iMin I*} **in** *meta-spec*)
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*)
apply (*frule Max-le-iMin-conv-singleton*[*THEN iffD1*], *clarsimp*+)
apply (*simp add*: *remove-Max-eq Max-le-iMin-conv-singleton*)
done

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*
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*)
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*
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*=$I \downarrow< Max\ I$ **and** *k*=*k* **in** *not-greater-Max*, *assumption*)
apply (*simp add*: *cut-less-mem-iff*)
done

lemma *inext-nth-card-Max-aux'*:
$\bigwedge I.\ [\![\ finite\ I;\ I \neq \{\}\ ]\!] \Longrightarrow I \rightarrow (card\ I - Suc\ 0) = Max\ I$
by (*simp add*: *inext-nth-card-Max-aux not-empty-card-gr0-conv*)

lemma *inext-nth-card-Max*:
$[\![\ finite\ I;\ I \neq \{\};\ card\ I \leq Suc\ n\ ]\!] \Longrightarrow I \rightarrow n = Max\ I$
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'*:
$[\![\ finite\ I;\ I \neq \{\};\ card\ I - Suc\ 0 \leq n\ ]\!] \Longrightarrow I \rightarrow n = Max\ I$
by (*simp add*: *inext-nth-card-Max*)

lemma *inext-nth-singleton*: $\{a\} \rightarrow n = a$
by (*simp add*: *inext-nth-Max-fix*[*OF singleton-finite singleton-not-empty* - *le0*])

lemma *inext-nth-eq-Min-conv*:
$I \neq \{\} \Longrightarrow (I \rightarrow n = iMin\ I) = (n = 0 \vee (\exists\, a.\ I = \{a\}))$
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* ≠ $I \rightarrow n$ *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*: $\bigwedge I\ b.$
$I \neq \{\} \Longrightarrow I \downarrow\geq (I \to a) \to b = I \to (a + b)$
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 ∨ 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*)
apply (*drule-tac x=I − {iMin I}* **in** *meta-spec*)
apply (*drule-tac x=b* **in** *meta-spec*)
apply (*drule meta-mp*, *blast*)
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 → 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

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*)
apply (*blast intro*: *order-less-imp-le order-le-less-trans iMin-le-Max*)
apply (*frule-tac n=Suc n* **in** *Max-ge*[*OF - inext-nth-closed*, *rule-format*], *assumption*)
apply (*drule order-le-neq-trans*, *simp*+)
apply (*drule order-le-less-trans*[*OF inext-mono*])
apply (*simp add*: *inext-append-eq1 inext-nth-closed*)
done

lemma *inext-nth-card-append-eq1*:
$\bigwedge A\ B.$⟦ *Max A < iMin B*; *n < card A* ⟧ ⟹
(*A ∪ B*) *→ n = A → n*
apply (*case-tac B = {}*, *simp*)
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*)
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*)
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* $\rightarrow$ *Suc n*)
prefer *2*
apply (*subgoal-tac A* $\rightarrow$ *Suc n* $=$ *Max A*)
prefer *2*
apply (*rule inext-nth-card-Max*)
apply *simp*+
apply (*rule-tac n*=*A* $\rightarrow$ *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

lemma *inext-nth-card-append-eq3*:
⟦ *finite A*; *B* $\neq$ {}; *Max A* $<$ *iMin B* ⟧ $\Longrightarrow$
(*A* $\cup$ *B*) $\rightarrow$ (*card A*) $=$ *iMin B*
apply (*case-tac A* $=$ {}, *simp*)
apply (*frule not-empty-card-gr0-conv*[*THEN iffD1*], *assumption*)
apply (*rule subst*[*OF Suc-pred*, *of card A*], *assumption*)
apply (*simp only*: *inext-nth.simps*)
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* $\neq$ {}; *B* $\neq$ {}; *Max A* $<$ *iMin B*; *card A* $\leq$ *n* ⟧ $\Longrightarrow$
(*A* $\cup$ *B*) $\rightarrow$ *n* $=$ *B* $\rightarrow$ (*n* $-$ *card A*)
apply (*rule-tac t*=(*A* $\cup$ *B*) $\rightarrow$ *n* **and** *s*=(*A* $\cup$ *B*) $\rightarrow$ (*card A* $+$ (*n* $-$ *card A*)) **in** *subst*, *simp*)
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

lemma *inext-nth-card-append*:
⟦ *finite A*; *A* $\neq$ {}; *B* $\neq$ {}; *Max A* $<$ *iMin B* ⟧ $\Longrightarrow$
(*A* $\cup$ *B*) $\rightarrow$ *n* $=$ (*if n* $<$ *card A then A* $\rightarrow$ *n else B* $\rightarrow$ (*n* $-$ *card A*))

by (*simp add*: *inext-nth-card-append-eq1 inext-nth-card-append-eq2*)

lemma *inext-nth-insert-Suc*:
⟦ $I \neq \{\}$; $a < iMin\ I$ ⟧ ⟹ $(insert\ a\ I) \to Suc\ n = I \to n$
apply (*frule not-less-iMin*)
apply (*rule-tac t*=$I \to n$ **and** *s*=$(insert\ a\ I - \{iMin\ (insert\ a\ I)\}) \to n$ **in** *subst*)
apply (*simp add*: *iMin-insert min-eqL*)
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*:
$n < card\ (I \downarrow< t) \Longrightarrow (I \downarrow< t) \to n = I \to n$
apply (*rule-tac t*=$I \to n$ **and** *s*=$(I \downarrow< t \cup I \downarrow\geq t) \to n$ **in** *subst*)
apply (*simp add*: *cut-less-cut-ge-ident*)
apply (*case-tac* $I \downarrow\geq t = \{\}$, *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*:
$n < card\ (I \downarrow< t) \Longrightarrow I \to n < t$
apply (*case-tac* $I \downarrow< t = \{\}$, *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*:
$I \downarrow\geq t \neq \{\} \Longrightarrow (I \to n < t) = (n < card\ (I \downarrow< t))$
apply (*case-tac* $I = \{\}$, *blast*)
apply (*case-tac* $I \downarrow< t = \{\}$)
apply (*simp add*: *linorder-not-less*)
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* $(I \downarrow< t) < iMin\ (I \downarrow\geq t)$)
prefer *2*
apply (*simp add*: *nat-cut-less-finite iMin-gr-iff Ball-def i-cut-mem-iff*)
apply (*drule ssubst*[*OF cut-less-cut-ge-ident*[*OF order-refl*], *of* $\lambda x.\ x \to n < t$ - *t*])
apply (*drule inext-nth-card-append-eq2*[*OF nat-cut-less-finite*, *of* $I\ t\ I \downarrow\geq t\ n$], *assumption*+)
apply (*simp add*: *inext-nth-card-append-eq2 nat-cut-less-finite*)
apply (*subgoal-tac* $\bigwedge x.\ I \downarrow\geq t \to x \geq t$)
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*:
$n < card\ I \vee infinite\ I \Longrightarrow card\ (I \downarrow< (I \to n)) = n$
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 *fastforce*
apply *simp*
apply (*subgoal-tac* $I \to n \neq Max\ I \vee infinite\ I$)
prefer *2*
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*:
$[\![\ finite\ I;\ card\ I \leq Suc\ n\ ]\!] \Longrightarrow card\ (I \downarrow< (I \to n)) = card\ I - Suc\ 0$
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*:
$card\ (I \downarrow< (I \to n)) = ($
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*:
$n < card\ I \vee infinite\ I \Longrightarrow card\ (I \downarrow\leq (I \to n)) = Suc\ n$
apply (*case-tac* $I = \{\}$, *simp*)
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*:
$[\![\ finite\ I;\ card\ I \leq Suc\ n\ ]\!] \Longrightarrow card\ (I \downarrow\leq (I \to n)) = card\ I$
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*:
$card\ (I \downarrow\leq (I \to n)) = ($

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*)

primrec *iprev-nth* :: *nat set* ⇒ *nat* ⇒ *nat* (‹(- ← -)› [*100*, *100*] *60*)
where
I ← *0* = *Max I*
| *I* ← *Suc n* = *iprev* (*iprev-nth I n*) *I*

lemma *iprev-nth-closed*: ⟦ *finite I*; *I* ≠ {} ⟧ ⟹ *I* ← *n* ∈ *I*
apply (*induct n*)
apply *simp*
apply (*simp add*: *iprev-closed*)
done

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

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*
apply *simp*
apply (*rule iprev-mono2*)
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*:
⟦ $I \neq \{\}$; $I \leftarrow a = iMin\ I$; $a \leq b$ ⟧ ⟹ $I \leftarrow b = iMin\ I$
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*: $\{a\} \leftarrow n = a$
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*:
$\{x \in I.\ \neg(\exists n.\ I \rightarrow n = x)\} = \{\}$
(**is** *?S* = {}
is $\{ x \in I.\ ?P\ x\} = \{\}$)
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* ≠ {}
using *as-S-not-empty* **by** *blast*

have *s-not-ex*: ⋀*x*. ⟦ *x* ∈ *I*; *?P x* ⟧ ⟹ *x* ∈ *S*
using *s-S* **by** *blast*

have *s-subset*:*S* ⊆ *I*
using *s-S* **by** *blast*
have *i-not-empty*: *I* ≠ {}
using *as-S-not-empty* **by** *blast*

have *s-iMin-S*: *iMin S* ∈ *S*
using *S-not-empty* **by** (*simp add*: *iMinI-ex2*)
hence *s-iMin-i*: *iMin S* ∈ *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 ∈ I
    apply (simp add: s-prev)
    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)
    apply (rule inext-iprev)
    apply (insert as, simp)
    done

  have s-prev-min-1: prev < iMin S
    apply (simp only: s-prev)
    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 ∉ S
    by (simp add: not-less-iMin)
  have ∃ n. I → n = 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)
    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

lemma inext-nth-surj-on:surj-on (λn. I → n) UNIV I
apply (simp add: surj-on-conv)
by (insert inext-nth-surj-aux1[of I], blast)

corollary in-imp-ex-inext-nth: x ∈ I ⟹ ∃ n. x = I → n
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
apply (rule-tac f=λn. I → n and I=I in image-nat-induct)
apply (simp add: inext-nth-closed[OF in-imp-not-empty] inext-nth-surj-on)+
done

lemma iprev-nth-surj-aux1:
  finite I ⟹ { x ∈ I. ¬(∃ n. I ← n = x)} = {}
apply (case-tac I = {}, blast)
proof (rule ccontr)
  assume as-finite-i: finite I
  let ?S = {x ∈ I. ¬ (∃ n. I ← n = x)}
  assume as-S-not-empty: ?S ≠ {}

  obtain S where s-S: S = ?S by blast
  hence S-not-empty: S ≠ {}
    using as-S-not-empty by blast

  have s-not-ex: ⋀x. ⟦ x ∈ I; ¬(∃ n. I ← n = x) ⟧ ⟹ x ∈ S
    using s-S by blast

  have s-subset:S ⊆ I
    using s-S by blast
  have i-not-empty: I ≠ {}
    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 ∈ S
```

```
  using S-not-empty S-finite by simp
hence s-Max-i: Max S ∈ 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' ∈ 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)
    apply (rule iprev-inext)
    apply (insert as, simp)
    done

  have s-next-max-1: Max S < next'
    apply (simp add: s-next)
    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
    apply (drule Max-ge[of - next'])
    apply simp-all
    done
  have ∃ n. I ← n = 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)
    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

lemma iprev-nth-surj-on: finite I ⟹ surj-on (λn. I ← n) UNIV I
apply (simp add: surj-on-def)
by (insert iprev-nth-surj-aux1[of I], blast)

corollary in-imp-ex-iprev-nth:
 ⟦ finite I;  x ∈ I ⟧ ⟹ ∃ n. x = I ← n
apply (rule surj-onD[of - UNIV I, simplified])
apply (rule iprev-nth-surj-on)
apply assumption+
done

lemma iprev-induct:
 ⟦ P (Max I); ⋀n. ⟦ n ∈ I; P n ⟧ ⟹ P (iprev n I); finite I; n ∈ I ⟧ ⟹ P n
apply (rule-tac f=λn. I ← n and I=I in image-nat-induct)
apply (simp add: iprev-nth-closed[OF - in-imp-not-empty] iprev-nth-surj-on)+
done
```

8.4 Natural intervals with *inext* and *iprev*

```
lemma inext-atLeast: n ≤ t ⟹ inext t {n..} = Suc t
apply (unfold inext-def)
apply (subgoal-tac Suc t ∈ {n..} ↓> 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 ≤ t ⟹ 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  ⟹ iprev t {n..} = t − Suc 0
by (insert iprev-atLeast'[of n t − Suc 0], simp)

lemma inext-atMost: t < n ⟹ inext t {..n} = Suc t
```

apply (*unfold inext-def*)
apply (*subgoal-tac Suc t* ∈ {*..n*} ↓> *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* ≤ *n* ⟹ *iprev t* {*..n*} = *t* − *Suc 0*
apply (*case-tac t*)
apply *simp*
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* ⟹ *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* ⟹ *iprev t* {*..*<*n*} = *t* − *Suc 0*
apply (*case-tac n*, *simp*)
apply (*simp add*: *lessThan-Suc-atMost iprev-atMost*)
done

lemma *inext-atLeastAtMost*: ⟦ *m* ≤ *t*; *t* < *n* ⟧ ⟹ *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* ≤ *n* ⟧ ⟹ *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* ≤ *t*; *t* < *n* ⟧ ⟹ *iprev* (*Suc t*) {*m..n*} = *t*
by (*simp add*: *iprev-atLeastAtMost*[*of - Suc t*])

lemma *inext-nth-atLeast* : {*n..*} → *a* = *n* + *a*
apply (*induct a*, *simp add*: *iMin-atLeast*)
apply (*simp add*: *inext-atLeast*)
done
lemma *inext-nth-atLeastAtMost*: ⟦ *a* ≤ *n* − *m*; *m* ≤ *n* ⟧ ⟹ {*m..n*} → *a* = *m* +
a
apply (*induct a*, *simp add*: *iMin-atLeastAtMost*)
apply (*simp add*: *inext-atLeastAtMost*)
done
lemma *iprev-nth-atLeastAtMost*: ⟦ *a* ≤ *n* − *m*; *m* ≤ *n* ⟧ ⟹ {*m..n*} ← *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*

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

lemma *inext-nth-imirror-iprev-nth-conv*:

⟦ *finite I*; *I* ≠ {} ⟧ ⟹
(*imirror I*) → *n* = *mirror-elem* (*I* ← *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* ≠ {} ⟧ ⟹
mirror-elem ((*imirror I*) ← *n*) *I* = *I* → *n*
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* ≠ {} ⟧ ⟹
(*imirror I*) ← *n* = *mirror-elem* (*I* → *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* ≠ {} ⟧ ⟹
mirror-elem ((*imirror I*) → *n*) *I* = (*I* ← *n*)
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

lemma *iprev-nth-card-greater-iMin*: *Suc n* < *card I* ⟹ *iMin I* < *I* ← *n*
apply (*subgoal-tac I* ≠ {} *finite I*)
prefer *2*
apply (*rule card-gr0-imp-finite*, *simp*)
prefer *2*
apply (*rule card-gr0-imp-not-empty*, *simp*)
apply (*subst iprev-nth-imirror-inext-nth-conv2*[*symmetric*], *assumption*+)
apply (*subst mirror-elem-Max*[*symmetric*], *assumption*+)
apply (*subst mirror-elem-imirror*[*symmetric*], *assumption*)
apply (*subst mirror-elem-imirror*[*symmetric*], *assumption*)
apply (*frule imirror-finite*, *frule imirror-not-empty*)
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*)

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*
apply (*subst iprev-nth-imirror-inext-nth-conv2*[*symmetric*], *assumption*+)
apply (*subst mirror-elem-Max*[*symmetric*], *assumption*+)
apply (*subst mirror-elem-imirror*[*symmetric*], *assumption*)
apply (*subst mirror-elem-imirror*[*symmetric*], *assumption*)
apply (*rule subst*[*OF imirror-Max*], *assumption*)
apply (*frule imirror-finite*, *frule imirror-not-empty*)
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 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 *f-take'* :: *'a list* ⇒ *nat* ⇒ *'a list* (**infixl** ‹↓› *100*)
where *xs* ↓ *n* ≡ *take n xs*
abbreviation *f-drop'* :: *'a list* ⇒ *nat* ⇒ *'a list* (**infixl** ‹↑› *100*)
where *xs* ↑ *n* ≡ *drop n xs*

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 *fastforce*

lemma *length-ge-Suc-imp-not-empty*:$Suc\ n \leq length\ xs \Longrightarrow xs \neq []$
by *fastforce*

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 *fastforce*

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*:
$Suc\ n \leq length\ xs \Longrightarrow (tl\ xs)\ !\ n = xs\ !\ Suc\ n$
by (*rule hd-Cons-tl*[*OF length-ge-Suc-imp-not-empty*, *THEN subst*], *simp*+)
corollary *nth-tl-eq-nth-Suc2*:
$n < length\ xs \Longrightarrow (tl\ xs)\ !\ n = xs\ !\ Suc\ n$
by (*simp add*: *nth-tl-eq-nth-Suc*)

lemma *hd-eq-first*: $xs \neq [] \Longrightarrow xs\ !\ 0 = hd\ xs$
by (*induct xs*, *simp-all*)
corollary *take-first*:$xs \neq [] \Longrightarrow xs \downarrow\ (Suc\ 0) = [xs\ !\ 0]$
by (*induct xs*, *simp-all*)
corollary *take-hd*:$xs \neq [] \Longrightarrow xs \downarrow\ (Suc\ 0) = [hd\ xs]$
by (*simp add*: *take-first hd-eq-first*)

theorem *last-nth*: $xs \neq [] \Longrightarrow last\ xs = xs\ !\ (length\ xs - Suc\ 0)$
by (*simp add*: *last-conv-nth*)

lemma *last-take*: $n < length\ xs \Longrightarrow last\ (xs \downarrow Suc\ n) = xs\ !\ n$
by (*simp add*: *last-nth length-greater-imp-not-empty min-eqR*)
corollary *last-take2*:
$[\![\ 0 < n;\ n \leq length\ xs\ ]\!] \Longrightarrow last\ (xs \downarrow n) = xs\ !\ (n - Suc\ 0)$
apply (*frule diff-Suc-less*[*THEN order-less-le-trans*, *of - length xs 0*], *assumption*)
apply (*drule last-take*[*of n − Suc 0 xs*])
apply *simp*
done

corollary *nth-0-drop*: $n \leq length\ xs \Longrightarrow (xs \uparrow n)\ !\ 0 = xs\ !\ n$
by (*cut-tac nth-drop*[*of n xs 0*], *simp+*)

lemma *drop-eq-tl*: $xs \uparrow (Suc\ 0) = tl\ xs$
by (*simp add*: *drop-Suc*)

lemma *drop-take-1*:
$n < length\ xs \Longrightarrow xs \uparrow n \downarrow (Suc\ 0) = [xs\ !\ n]$
by (*simp add*: *take-hd hd-drop-conv-nth*)

lemma *upt-append*: $m \leq n \Longrightarrow [0..<m]\ @\ [m..<n] = [0..<n]$
by (*insert upt-add-eq-append*[*of 0 m n − m*], *simp*)

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-all P xs* = $(\forall\, i{<}length\ xs.\ P\ (xs\ !\ i))$
by (*rule list-all-length*)

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 *replicate'* :: *'a* ⇒ *nat* ⇒ *'a list* (‹-^-› [*1000*,*65*])
  where $x^n \equiv$ *replicate n x*

lemma *replicate-snoc*: x^n @ [*x*] = $x^{Suc\ n}$
by (*simp add*: *replicate-app-Cons-same*)

lemma *eq-replicate-conv*: (∀ *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* ⟹ *length* (*x* # $a^{n\ -\ Suc\ 0}$) = *n*
by *simp*

lemma *replicate-le-diff*: *m* ≤ *n* ⟹ x^m @ $x^{n\ -\ m}$ = x^n
by (*simp add*: *replicate-add*[*symmetric*])
lemma *replicate-le-diff2*: ⟦ *k* ≤ *m*; *m* ≤ *n* ⟧ ⟹ $x^{m\ -\ k}$ @ $x^{n\ -\ m}$ = $x^{n\ -\ k}$
by (*subst replicate-add*[*symmetric*], *simp*)

lemma *append-constant-length-induct-aux*: ⋀*xs*.
  ⟦ *length xs div k* = *n*; ⋀*ys*. *k* = *0* ∨ *length ys* < *k* ⟹ *P ys*;
    ⋀*xs ys*. ⟦ *length xs* = *k*; *P ys* ⟧ ⟹ *P* (*xs* @ *ys*) ⟧ ⟹ *P xs*
apply (*case-tac k* = *0*, *blast*)
apply *simp*
apply (*induct n*)
 apply (*simp add*: *div-eq-0-conv'*)
apply (*subgoal-tac k* ≤ *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*:
  $[\![\bigwedge ys.\ k = 0 \vee length\ ys < k \Longrightarrow P\ ys;$
    $\bigwedge xs\ ys.\ [\![length\ xs = k;\ P\ ys]\!] \Longrightarrow P\ (xs\ @\ ys)]\!] \Longrightarrow P\ xs$
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*) $\downarrow$ *n* = *zip* (*xs* $\downarrow$ *n*) *ys*
by (*simp add*: *expand-list-eq*)

lemma *zip-takeR*: (*zip xs ys*) $\downarrow$ *n* = *zip xs* (*ys* $\downarrow$ *n*)
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*) $\downarrow$ *n* = *zip* (*xs* $\downarrow$ *n*) (*ys* $\downarrow$ *n*)
by (*rule take-zip*)

lemma *hd-zip*: $[\![xs \neq [];\ ys \neq []]\!] \Longrightarrow$ *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*) $\Longrightarrow$ *P xs*
by (*subst map-id*[*symmetric*])

lemma *map-one*: *map f* [*x*] = [*f x*]
by *simp*

lemma *map-last*: *xs* $\neq$ [] $\Longrightarrow$ *last* (*map f xs*) = *f* (*last xs*)
by (*rule last-map*)

lemma *filter-list-all*: *list-all P xs* $\Longrightarrow$ *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* ($\lambda x.$ *P x* = *Q x*) *xs* $\Longrightarrow$ *filter P xs* = *filter Q xs*
by (*induct xs*, *simp*+)

lemma *filter-nth*: $\bigwedge n.$
  n < *length* (*filter P xs*) $\Longrightarrow$

(*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*)
apply (*simp only*: *filter-snoc*)
apply (*simp split del*: *if-split*)
apply (*case-tac xs* = [])
apply (*simp split del*: *if-split*)
apply (*rule-tac*
t = λ*k i*. *i* = *0* ∧ *i* ≤ *k* ∧ *P* ([*x*] ! *i*) **and**
s = λ*k i*. *i* = *0* ∧ *P x*
in *subst*)
apply (*simp add*: *fun-eq-iff*)
apply *fastforce*
apply (*fastforce simp*: *Least-def*)
apply (*rule-tac*
t = λ*k*. *card* {*i*. *i* ≤ *k* ∧ *i* < *Suc* (*length xs*) ∧ *P* ((*xs* @ [*x*]) ! *i*)} **and**
s = λ*k*. (*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*: *if-split*, *rename-tac k*)
apply (*simp split del*: *if-split 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 = λ*i*. *i* < *length xs* ∧ *P* ((*xs* @ [*x*]) ! *i*) **and**
s = λ*i*. *i* < *length xs* ∧ *P* (*xs* ! *i*)
in *subst*)
apply (*rule fun-eq-iff*[*THEN iffD2*])
apply (*fastforce simp*: *nth-append1*)
apply (*rule add-left-cancel*[*THEN iffD2*])
apply (*rule-tac*
t = λ*i*. *i* = *length xs* ∧ *i* ≤ *k* ∧ *P* ((*xs* @ [*x*]) ! *i*) **and**
s = λ*i*. *i* = *length xs* ∧ *i* ≤ *k* ∧ *P x*
in *subst*)
apply (*rule fun-eq-iff*[*THEN iffD2*])
apply *fastforce*
apply (*case-tac length xs* ≤ *k*)
apply *clarsimp*
apply (*rule-tac*
t = λ*i*. *i* = *length xs* ∧ *i* ≤ *k* **and**
s = λ*i*. *i* = *length xs*
in *subst*)
apply (*rule fun-eq-iff*[*THEN iffD2*])

apply *fastforce*
apply *simp*
apply *simp*
apply (*simp split del*: *if-split add*: *less-Suc-eq conj-disj-distribL conj-disj-distribR*)
apply (*rule-tac*
t = λ*k*. *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 = λ*k*. *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 = λ*k*. *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 = λ*k*. *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*: *if-split*)
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*)
apply (*subst Least-le-imp-le-disj*)
apply *simp*
apply *simp*
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*)
apply (*simp add*: *length-filter-conv-card*)
apply (*rule-tac*
  $t = \lambda k.\ card\ \{i.\ i < length\ xs \wedge P\ (xs\ !\ i)\} < card\ \{i.\ i \leq k \wedge i < length\ xs \wedge P\ (xs\ !\ i)\}$ **and**
  $s = \lambda k.\ False$
  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 *fastforce*
 apply *blast*
apply *simp*
apply (*rule-tac*
  $t = (LEAST\ k.\ k = length\ xs \wedge$
    $card\ \{i.\ i < length\ xs \wedge P\ (xs\ !\ i)\} < Suc\ (card\ \{i.\ i \leq k \wedge i < length\ xs \wedge P\ (xs\ !\ i)\}))$ **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 *fastforce*
  apply *fastforce*
 apply *simp*
apply *simp*
done

9.1.3 Ordered lists

fun *list-ord* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a::ord)\ list \Rightarrow bool$
where
  list-ord ord $(x1\ \#\ x2\ \#\ xs) = (ord\ x1\ x2 \wedge list\text{-}ord\ ord\ (x2\ \#\ xs))$
| *list-ord ord xs* = *True*

definition *list-asc* :: $('a::ord)\ list \Rightarrow bool$ **where**
  list-asc xs $\equiv$ *list-ord* $(\leq)$ *xs*
definition *list-strict-asc* :: $('a::ord)\ list \Rightarrow bool$ **where**
  list-strict-asc xs $\equiv$ *list-ord* $(<)$ *xs*
value *list-asc* [*1*::*nat*, *2*, *2*]
value *list-strict-asc* [*1*::*nat*, *2*, *2*]
definition *list-desc* :: $('a::ord)\ list \Rightarrow bool$ **where**
  list-desc xs $\equiv$ *list-ord* $(\geq)$ *xs*
definition *list-strict-desc* :: $('a::ord)\ list \Rightarrow bool$ **where**
  list-strict-desc xs $\equiv$ *list-ord* $(>)$ *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*, *fastforce*)
apply (*case-tac xs*, *case-tac ys*, *fastforce*+)
done
lemma *list-ord-snoc*:
list-ord ord (*xs* @ [*x*]) =
(*xs* = [] ∨ (*ord* (*last xs*) *x* ∧ *list-ord ord xs*))
by (*fastforce 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 *fastforce*
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* (<)], *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* (>)], *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*)
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*)
apply (*simp add*: *list-ord-trans-imp*)
done

corollary
list-asc-trans:
($list\text{-}asc\ (xs::'a::preorder\ list)) =$
$(\forall j < length\ xs.\ \forall i < j.\ xs\ !\ i \leq xs\ !\ j)$ **and**
list-strict-asc-trans:
($list\text{-}strict\text{-}asc\ (xs::'a::preorder\ list)) =$
$(\forall j < length\ xs.\ \forall i < j.\ xs\ !\ i < xs\ !\ j)$ **and**
list-desc-trans:
($list\text{-}desc\ (xs::'a::preorder\ list)) =$
$(\forall j < length\ xs.\ \forall i < j.\ xs\ !\ j \leq xs\ !\ i)$ **and**
list-strict-desc-trans:
($list\text{-}strict\text{-}desc\ (xs::'a::preorder\ list)) =$
$(\forall j < length\ xs.\ \forall i < j.\ xs\ !\ j < xs\ !\ i)$
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:
($list\text{-}asc\ (xs::'a::preorder\ list)) =$
$(\forall j < length\ xs.\ \forall i \leq j.\ xs\ !\ i \leq xs\ !\ j)$ **and**
list-desc-trans-le:
($list\text{-}desc\ (xs::'a::preorder\ list)) =$
$(\forall j < length\ xs.\ \forall i \leq j.\ xs\ !\ j \leq xs\ !\ i)$
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:
($list\text{-}strict\text{-}asc\ (xs::'a::preorder\ list)) \Longrightarrow$
$(\forall j < length\ xs.\ \forall i \leq j.\ xs\ !\ i \leq xs\ !\ j)$
apply (*unfold list-strict-asc-def*)
apply (*rule list-ord-trans-refl-le-imp*[**where** *ord*=$(\leq)$])
apply (*unfold trans-def*, *blast intro*: *order-trans*)
apply *assumption*
apply (*unfold refl-on-def*, *clarsimp*)
apply (*rule list-ord-imp*[**where** *ord*=$(<)$], *simp-all add*: *less-imp-le*)
done

lemma *list-ord-le-sorted-eq*: $list\text{-}asc\ xs = sorted\ xs$
apply (*rule sym*)
apply (*simp add*: *list-asc-def*)

apply (*induct xs*, *simp*)
apply (*rename-tac x xs*)
apply (*simp add*: *list-ord-Cons*)
apply (*case-tac xs* = [], *simp-all*)
apply (*case-tac list-ord* (≤) *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*)
apply (*fastforce 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*)

lemma *list-ord-distinct-aux*:
⟦ *irefl* {(*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*: *irefl-def*)
apply (*simp add*: *list-ord-trans*)
done

lemma *list-ord-distinct*:
⟦ *irefl* {(*a*,*b*). *ord a b*}; *transP ord*; *list-ord ord xs* ⟧ ⟹
distinct xs
apply (*simp add*: *distinct-conv-nth*, *intro allI impI*, *rename-tac i j*)
apply (*drule neq-iff*[*THEN iffD1*], *erule disjE*)
apply (*simp add*: *list-ord-distinct-aux*)
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*=(<) **in** *list-ord-distinct*)
apply (*unfold irefl-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*=(>) **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 [*nbe*] *sublist-list* [*0*::*int*,*10*::*int*,*20*,*30*,*40*,*50*] [*1*::*nat*,*1*,*2*,*3*,*10*]

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 (*clarsimp*, *rename-tac i*)
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*: ⋀*xs*.
∀ *n*<*length ys*. *ys* ! *n* < *length xs* ⟹
sublist-list-if xs ys = *sublist-list xs ys*
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*]
by (*simp add*: *sublist-list-if-sublist-list-filter-conv take-drop-eq-sublist-list*)

lemma *nths-empty-conv*: (*nths xs I* = []) = (∀ *i*∈*I*. *length xs* ≤ *i*)
using [[*hypsubst-thin* = *true*]]
by (*fastforce simp*: *set-empty*[*symmetric*] *set-nths linorder-not-le*[*symmetric*])

lemma *nths-singleton2*: *nths xs* {*y*} = (*if y* < *length xs then* [*xs* ! *y*] *else* [])
apply (*unfold nths-def*)
apply (*induct xs rule*: *rev-induct*, *simp*)
apply (*simp add*: *nth-append*)
done

lemma *nths-take-eq*:
⟦ *finite I*; *Max I* < *n* ⟧ ⟹ *nths* (*xs* ↓ *n*) *I* = *nths xs I*
apply (*case-tac I* = {}, *simp*)
apply (*case-tac n* < *length xs*)
prefer *2*
apply *simp*
apply (*rule-tac*
t = *nths xs I* **and**
s = *nths* (*xs* ↓ *n* @ *xs* ↑ *n*) *I*
in *subst*)
apply *simp*
apply (*subst nths-append*)
apply (*simp add*: *min-eqR*)
apply (*rule-tac t*={*j*. *j* + *n* ∈ *I*} **and** *s*={} **in** *subst*)
apply *blast*
apply *simp*
done

lemma *nths-drop-eq*:
n ≤ *iMin I* ⟹ *nths* (*xs* ↑ *n*) {*j*. *j* + *n* ∈ *I*} = *nths xs I*
apply (*case-tac I* = {}, *simp*)
apply (*case-tac n* < *length xs*)
prefer *2*
apply (*simp add*: *nths-def filter-empty-conv linorder-not-less*)

apply (*clarsimp*, *rename-tac a b*)
apply (*drule set-zip-rightD*)
apply *fastforce*
apply (*rule-tac*
t = *nths xs I* **and**
s = *nths* (*xs* ↓ *n* @ *xs* ↑ *n*) *I*
in *subst*)
apply *simp*
apply (*subst nths-append*)
apply (*fastforce simp*: *nths-empty-conv min-eqR*)
done

lemma *nths-cut-less-eq*:
length xs ≤ *n* ⟹ *nths xs* (*I* ↓< *n*) = *nths xs I*
apply (*simp add*: *nths-def cut-less-mem-iff*)
apply (*rule-tac f*=λ*xs*. *map fst xs* **in** *arg-cong*)
apply (*rule filter-filter-eq*)
apply (*simp add*: *list-all-conv*)
done

lemma *nths-disjoint-Un*:
⟦ *finite A*; *Max A* < *iMin B* ⟧ ⟹ *nths xs* (*A* ∪ *B*) = *nths xs A* @ *nths xs B*
apply (*case-tac A* = {}, *simp*)
apply (*case-tac B* = {}, *simp*)
apply (*case-tac length xs* ≤ *iMin B*)
apply (*subst nths-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*: *nths-empty-conv iMin-ge-iff*)
apply (*simp add*: *linorder-not-le*)
apply (*rule-tac*
t = *nths xs* (*A* ∪ *B*) **and**
s = *nths* (*xs* ↓ (*iMin B*) @ *xs* ↑ (*iMin B*)) (*A* ∪ *B*)
in *subst*)
apply *simp*
apply (*subst nths-append*)
apply (*simp add*: *min-eqR*)
apply (*subst nths-cut-less-eq*[**where** *xs*=*xs* ↓ *iMin B* **and** *n*=*iMin B*, *symmetric*], *simp*)
apply (*simp add*: *cut-less-Un cut-less-Min-empty cut-less-Max-all*)
apply (*simp add*: *nths-take-eq*)
apply (*rule-tac*
t = λ*j*. *j* + *iMin B* ∈ *A* ∨ *j* + *iMin B* ∈ *B* **and**
s = λ*j*. *j* + *iMin B* ∈ *B*
in *subst*)
apply (*force simp*: *fun-eq-iff*)
apply (*simp add*: *nths-drop-eq*)
done

corollary *nths-disjoint-insert-left*:

⟦ *finite I*; *x* < *iMin I* ⟧ ⟹ *nths xs* (*insert x I*) = *nths xs* {*x*} @ *nths xs I*
apply (*rule-tac t*=*insert x I* **and** *s*={*x*} ∪ *I* **in** *subst*, *simp*)
apply (*subst nths-disjoint-Un*)
apply *simp-all*
done

corollary *nths-disjoint-insert-right*:
⟦ *finite I*; *Max I* < *x* ⟧ ⟹ *nths xs* (*insert x I*) = *nths xs I* @ *nths xs* {*x*}
apply (*rule-tac t*=*insert x I* **and** *s*=*I* ∪ {*x*} **in** *subst*, *simp*)
apply (*subst nths-disjoint-Un*)
apply *simp-all*
done

lemma *nths-all*: {..<*length xs*} ⊆ *I* ⟹ *nths xs I* = *xs*
apply (*case-tac xs* = [], *simp*)
apply (*rule-tac*
t = *I* **and**
s = *I* ↓< (*length xs*) ∪ *I* ↓≥ (*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 nths-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 (*fastforce simp*: *nths-empty-conv*)
done

corollary *nths-UNIV*: *nths xs UNIV* = *xs*
by (*rule nths-all*[*OF subset-UNIV*])

lemma *sublist-list-nths-eq*: ⋀*xs*.
list-strict-asc ys ⟹ *sublist-list-if xs ys* = *nths 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*: *nths-singleton2*)
apply (*unfold list-strict-asc-def*)
apply (*simp add*: *sublist-list-if-snoc split del*: *if-split*)
apply (*frule list-ord-append*[*THEN iffD1*])
apply (*clarsimp split del*: *if-split*)

apply (*subst nths-disjoint-insert-right*)
apply *simp*
apply (*clarsimp simp*: *in-set-conv-nth*, *rename-tac i*)
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*: *if-split*)+
apply (*simp add*: *nths-singleton2*)
done

lemma *set-sublist-list-if*: $\bigwedge xs.\ set\ (sublist\text{-}list\text{-}if\ xs\ ys) = \{xs\ !\ i\ |i.\ i < length\ xs \wedge i \in set\ ys\}$
apply (*induct ys*, *simp-all*)
apply *blast*
done

lemma *set-sublist-list*:
$list\text{-}all\ (\lambda i.\ i < length\ xs)\ ys \Longrightarrow$
$set\ (sublist\text{-}list\ xs\ ys) = \{xs\ !\ i\ |i.\ i < length\ xs \wedge i \in set\ ys\}$
by (*simp add*: *sublist-list-if-sublist-list-eq*[*symmetric*] *set-sublist-list-if*)

lemma *set-sublist-list-if-eq-set-sublist*: $set\ (sublist\text{-}list\text{-}if\ xs\ ys) = set\ (nths\ xs\ (set\ ys))$
by (*simp add*: *set-nths set-sublist-list-if*)

lemma *set-sublist-list-eq-set-sublist*:
$list\text{-}all\ (\lambda i.\ i < length\ xs)\ ys \Longrightarrow$
$set\ (sublist\text{-}list\ xs\ ys) = set\ (nths\ xs\ (set\ ys))$
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 $xs\ `f\ A \equiv \{y.\ \exists n{\in}A.\ n < length\ xs \wedge y = xs\ !\ n\}$

abbreviation *f-range* :: *'a list* ⇒ *'a set*
where *f-range xs* ≡ *f-image xs UNIV*

lemma *f-image-eqI*[*simp*, *intro*]:
$[\![\ x = xs\ !\ n;\ n \in A;\ n < length\ xs\ ]\!] \Longrightarrow x \in xs\ `f\ A$
by (*unfold f-image-def*, *blast*)

lemma *f-imageI*: $[\![\ n \in A;\ n < length\ xs\ ]\!] \Longrightarrow xs\ !\ n \in xs\ `f\ A$
by *blast*

lemma *rev-f-imageI*: $[\![\ n \in A;\ n < length\ xs;\ x = xs\ !\ n\ ]\!] \Longrightarrow x \in xs\ `f\ A$
by (*rule f-image-eqI*)

lemma *f-imageE*[*elim*!]:
$[\![\ x \in xs\ `f\ A;\ \bigwedge n.\ [\![\ x = xs\ !\ n;\ n \in A;\ n < length\ xs\ ]\!] \Longrightarrow P\ ]\!] \Longrightarrow P$

by (*unfold f-image-def*, *blast*)

lemma *f-image-Un*: $xs\ `^f\ (A \cup B) = xs\ `^f\ A \cup xs\ `^f\ B$
by *blast*

lemma *f-image-mono*: $A \subseteq B \Longrightarrow xs\ `^f\ A \subseteq xs\ `^f\ B$
by *blast*

lemma *f-image-iff*: $(x \in xs\ `^f\ A) = (\exists\, n{\in}A.\ n < length\ xs \wedge x = xs\ !\ n)$
by *blast*

lemma *f-image-subset-iff*:
$(xs\ `^f\ A \subseteq B) = (\forall\, n{\in}A.\ n < length\ xs \longrightarrow xs\ !\ n \in B)$
by *blast*

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

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*

lemma *f-image-empty*: $xs\ `^f\ \{\} = \{\}$
by *blast*

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 if-split*, *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*)

lemma *insert-f-image*:
$[\![\ n \in A;\ n < length\ xs\ ]\!] \Longrightarrow insert\ (xs\ !\ n)\ (xs\ `^f\ A) = (xs\ `^f\ A)$
by *blast*

lemma *f-image-is-empty*: $(xs\ `^f\ A = \{\}) = (\{x.\ x \in A \wedge x < length\ xs\} = \{\})$
by *blast*

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 (*fastforce 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-nths*: $xs\ {}^{`f}\ A = set\ (nths\ xs\ A)$
by (*unfold set-nths*, *blast*)
lemma *f-image-eq-set-sublist-list-if*: $xs\ {}^{`f}\ (set\ ys) = set\ (sublist\text{-}list\text{-}if\ xs\ ys)$
by (*simp add*: *set-sublist-list-if-eq-set-sublist f-image-eq-set-nths*)
lemma *f-image-eq-set-sublist-list*:
$list\text{-}all\ (\lambda i.\ i < length\ xs)\ ys \Longrightarrow xs\ {}^{`f}\ (set\ ys) = set\ (sublist\text{-}list\ xs\ ys)$
by (*simp add*: *sublist-list-if-sublist-list-eq f-image-eq-set-sublist-list-if*)

lemma *f-range-eqI*: $[\![\ x = xs\ !\ n;\ n < length\ xs\ ]\!] \Longrightarrow x \in f\text{-}range\ xs$
by *blast*

lemma *f-rangeI*: $n < length\ xs \Longrightarrow xs\ !\ n \in f\text{-}range\ xs$
by *blast*

lemma *f-rangeE*[*elim?*]:
$[\![\ x \in f\text{-}range\ xs;\ \bigwedge n.\ [\![\ n < length\ xs;\ x = xs\ !\ n\ ]\!] \Longrightarrow P\ ]\!] \Longrightarrow P$
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
$map\text{-}list\ []\ xs = []$
$|\ map\text{-}list\ (f\ \#\ fs)\ xs = f\ (hd\ xs)\ \#\ map\text{-}list\ fs\ (tl\ xs)$

lemma *map-list-Nil*: *map-list* [] *xs* = []
by *simp*

lemma *map-list-Cons-Cons*:
$map\text{-}list\ (f\ \#\ fs)\ (x\ \#\ xs) =$
$(f\ x)\ \#\ map\text{-}list\ fs\ xs$
by *simp*

lemma *map-list-length*: $\bigwedge xs.$
$length\ (map\text{-}list\ fs\ xs) = length\ fs$
by (*induct fs*, *simp+*)
corollary *map-list-empty-conv*:
$(map\text{-}list\ fs\ xs = []) = (fs = [])$
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 fs n xs*)
apply (*simp add*: *tl-take*)
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)
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
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)
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))
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
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))]
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)
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 $\leq$ *length xs* $\Longrightarrow$
map2 f xs (*ys1* @ *ys2*) =
map2 f (*xs* $\downarrow$ *length ys1*) *ys1* @
map2 f (*xs* $\uparrow$ *length ys1*) *ys2*
apply (*rule-tac*
t = *map2 f xs* (*ys1* @ *ys2*) **and**
s = *map2 f* (*xs* $\downarrow$ *length ys1* @ *xs* $\uparrow$ *length ys1*) (*ys1* @ *ys2*)
in *subst*)
apply *simp*
apply (*simp add*: *map2-append-append del*: *append-take-drop-id*)
done

lemma *map2-cong*:
$[\![$ *xs1* = *xs2*; *ys1* = *ys2*; *length xs2* $\leq$ *length ys2*;
$\bigwedge$*x y*. $[\![$ *x* $\in$ *set xs2*; *y* $\in$ *set ys2* $]\!] \Longrightarrow$ *f x y* = *g x y* $]\!] \Longrightarrow$
map2 f xs1 ys1 = *map2 g xs2 ys2*
by (*simp* (*no-asm-simp*) *add*: *expand-list-eq map2-length map2-nth*)

lemma *map2-eq-conv*:
length xs $\leq$ *length ys* $\Longrightarrow$
(*map2 f xs ys* = *map2 g xs ys*) = ($\forall$ *i*<*length xs*. *f* (*xs* ! *i*) (*ys* ! *i*) = *g* (*xs* ! *i*) (*ys* ! *i*))
by (*simp add*: *expand-list-eq map2-length map2-nth*)

lemma *map2-replicate*: *map2 f* x^n y^n = $(f\ x\ y)^n$
by (*induct n*, *simp*+)

lemma *map2-zip-conv*: $\bigwedge$*ys*.
length xs $\leq$ *length ys* $\Longrightarrow$
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*: $\bigwedge$*ys*.
length xs = *length ys* $\Longrightarrow$
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

hide-const (**open**) *map2*

end

10 Set operations with results of type enat

theory *InfiniteSet2*
imports *SetInterval2*
begin

10.1 Set operations with *enat*

10.1.1 Basic definitions

definition *icard* :: *'a set* ⇒ *enat*
  where *icard A* ≡ *if finite A then enat* (*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* = *enat* (*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* = *enat* (*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-enat-imp*: *icard A* = *enat 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-enat*: *finite A* ⟹ *the-enat* (*icard A*) = *card A*
by (*simp add*: *icard-def*)

lemma *icard-eq-enat-imp-card*: *icard A* = *enat n* ⟹ *card A* = *n*
by (*frule icard-eq-enat-imp*, *simp add*: *icard-finite*)

lemma *icard-eq-enat-card-conv*: *0* < *n* ⟹ (*icard A* = *enat n*) = (*card A* = *n*)
apply (*rule iffI*)
 apply (*simp add*: *icard-eq-enat-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-enat-def*)
apply (*rule iffI*)
apply (*frule icard-eq-enat-imp*)
apply (*simp add*: *icard-finite*)
apply *simp*
done
lemmas *icard-empty-iff-enat* = *icard-empty-iff*[*unfolded zero-enat-def*]

lemma *icard-not-empty-iff*: (*0* < *icard A*) = (*A* ≠ {})
by (*simp add*: *icard-empty-iff*[*symmetric*])
lemmas *icard-not-empty-iff-enat* = *icard-not-empty-iff*[*unfolded zero-enat-def*]

lemma *icard-singleton*: *icard* {*a*} = *eSuc 0*
by (*simp add*: *icard-finite eSuc-enat*)
lemmas *icard-singleton-enat*[*simp*] = *icard-singleton*[*unfolded zero-enat-def*]
lemma *icard-1-imp-singleton*: *icard A* = *eSuc 0* ⟹ ∃ *a*. *A* = {*a*}
apply (*simp add*: *eSuc-enat*)
apply (*frule icard-eq-enat-imp*)
apply (*simp add*: *icard-finite card-1-imp-singleton*)
done
lemma *icard-1-singleton-conv*: (*icard A* = *eSuc 0*) = (∃ *a*. *A* = {*a*})
apply (*rule iffI*)
apply (*simp add*: *icard-1-imp-singleton*)
apply *fastforce*
done

lemma *icard-insert-disjoint*: *x* ∉ *A* ⟹ *icard* (*insert x A*) = *eSuc* (*icard A*)
apply (*case-tac finite A*)
apply (*simp add*: *icard-finite eSuc-enat card-insert-disjoint*)
apply (*simp add*: *infinite-insert*)
done

lemma *icard-insert-if*: *icard* (*insert x A*) = (*if x* ∈ *A then icard A else eSuc* (*icard A*))
apply (*case-tac x* ∈ *A*)
apply (*simp add*: *insert-absorb*)
apply (*simp add*: *icard-insert-disjoint*)
done

lemmas *icard-0-eq* = *icard-empty-iff*

lemma *icard-Suc-Diff1*: *x* ∈ *A* ⟹ *eSuc* (*icard* (*A* − {*x*})) = *icard A*
apply (*case-tac finite A*)
apply (*simp add*: *icard-finite eSuc-enat in-imp-not-empty not-empty-card-gr0-conv*[*THEN iffD1*])
apply (*simp add*: *Diff-infinite-finite*[*OF singleton-finite*])
done

lemma *icard-Diff-singleton*: *x* ∈ *A* ⟹ *icard* (*A* − {*x*}) = *icard A* − *1*

apply (*rule eSuc-inject*[*THEN iffD1*])
apply (*frule in-imp-not-empty*, *drule icard-not-empty-iff*[*THEN iffD2*])
apply (*simp add*: *icard-Suc-Diff1 eSuc-pred-enat one-eSuc*)
done

lemma *icard-Diff-singleton-if*: *icard* (*A* − {*x*}) = (*if x* ∈ *A then icard A* − *1 else icard A*)
by (*simp add*: *icard-Diff-singleton*)

lemma *icard-insert*: *icard* (*insert x A*) = *eSuc* (*icard* (*A* − {*x*}))
by (*metis icard-Diff-singleton-if icard-Suc-Diff1 icard-insert-disjoint insert-absorb*)

lemma *icard-insert-le*: *icard A* ≤ *icard* (*insert x A*)
by (*simp add*: *icard-insert-if*)

lemma *icard-mono*: *A* ⊆ *B* ⟹ *icard A* ≤ *icard B*
apply (*case-tac finite B*)
apply (*frule subset-finite-imp-finite*, *simp*)
apply (*simp add*: *icard-finite card-mono*)
apply *simp*
done

lemma *not-icard-seteq*: ∃(*A*::*nat set*) *B*. (*A* ⊆ *B* ∧ *icard B* ≤ *icard A* ∧ ¬ *A* = *B*)
apply (*rule-tac x*={*1*..} **in** *exI*)
apply (*rule-tac x*={*0*..} **in** *exI*)
apply (*fastforce simp add*: *infinite-atLeast*)
done

lemma *not-psubset-icard-mono*: ∃(*A*::*nat set*) *B*. *A* ⊂ *B* ∧ ¬ *icard A* < *icard B*
apply (*rule-tac x*={*1*..} **in** *exI*)
apply (*rule-tac x*={*0*..} **in** *exI*)
apply (*fastforce simp add*: *infinite-atLeast*)
done

lemma *icard-Un-Int*: *icard A* + *icard B* = *icard* (*A* ∪ *B*) + *icard* (*A* ∩ *B*)
apply (*case-tac finite A*, *case-tac finite B*)
apply (*simp add*: *icard-finite card-Un-Int*[*of A*])
apply *simp-all*
done

lemma *icard-Un-disjoint*: *A* ∩ *B* = {} ⟹ *icard* (*A* ∪ *B*) = *icard A* + *icard B*
by (*simp add*: *icard-Un-Int*[*of A*])

lemma *not-icard-Diff-subset*: ∃(*A*::*nat set*) *B*. *B* ⊆ *A* ∧ ¬ *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 eSuc-enat*)

done

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*)

lemma *icard-Diff1-le*: $icard\ (A - \{x\}) \leq icard\ A$
by (*rule icard-mono*, *rule Diff-subset*)

lemma *icard-psubset*: $[\![\ A \subseteq B;\ icard\ A < icard\ B\ ]\!] \Longrightarrow A \subset B$
by (*metis less-le psubset-eq*)

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-enat*)
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*)
apply (*subgoal-tac* $\bigwedge c.\ c \in C \Longrightarrow card\ c = k1$)
prefer *2*
apply (*rule icard-eq-enat-imp-card*, *simp*)
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*)
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*)
apply (*frule Union-upper*)
apply (*rule infinite-super*, *assumption*)
apply *simp*
done

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

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

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

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*)

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

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*)

lemma *icard-cartesian-product*: *icard* (*A* × *B*) = *icard A* ∗ *icard B*
apply (*case-tac A* = {} ∨ *B* = {}, *fastforce*)
apply *clarsimp*
apply (*case-tac finite A* ∧ *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

lemma *icard-cartesian-product-singleton*: *icard* ($\{x\} \times A$) = *icard A*
by (*simp add*: *icard-cartesian-product mult-eSuc*)

lemma *icard-cartesian-product-singleton-right*: *icard* ($A \times \{x\}$) = *icard A*
by (*simp add*: *icard-cartesian-product mult-eSuc-right*)

lemma
icard-lessThan: *icard* $\{..<u\}$ = *enat u* **and**
icard-atMost: *icard* $\{..u\}$ = *enat* (*Suc u*) **and**
icard-atLeastLessThan: *icard* $\{l..<u\}$ = *enat* ($u - l$) **and**
icard-atLeastAtMost: *icard* $\{l..u\}$ = *enat* (*Suc u* − *l*) **and**
icard-greaterThanAtMost: *icard* $\{l<..u\}$ = *enat* ($u - l$) **and**
icard-greaterThanLessThan: *icard* $\{l<..<u\}$ = *enat* (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*)

lemma
icard-atLeastZeroLessThan-int: *icard* $\{0..<u\}$ = *enat* (*nat u*) **and**
icard-atLeastLessThan-int: *icard* $\{l..<u\}$ = *enat* (*nat* ($u - l$)) **and**
icard-atLeastAtMost-int: *icard* $\{l..u\}$ = *enat* (*nat* ($u - l + 1$)) **and**
icard-greaterThanAtMost-int: *icard* $\{l<..u\}$ = *enat* (*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 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

type-synonym $'a\ ilist = nat \Rightarrow 'a$

definition *i-append* :: $'a\ list \Rightarrow 'a\ ilist \Rightarrow 'a\ ilist$ (**infixr** $\frown$ *65*)
  where $xs \frown f \equiv \lambda n.\ if\ n < length\ xs\ then\ xs\ !\ n\ else\ f\ (n - length\ xs)$

Synonym for the lemma *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-eq-iff*
lemmas *ilist-eq-iff = expand-ilist-eq*

lemma *i-append-nth*: $(xs \frown f)\ n = (if\ n < length\ xs\ then\ xs\ !\ n\ else\ f\ (n - length\ xs))$
by (*simp add*: *i-append-def*)
lemma *i-append-nth1*[*simp*]: $n < length\ xs \Longrightarrow (xs \frown f)\ n = xs\ !\ n$
by (*simp add*: *i-append-def*)
lemma *i-append-nth2*[*simp*]: $length\ xs \leq n \Longrightarrow (xs \frown f)\ n = f\ (n - length\ xs)$
by (*simp add*: *i-append-def*)
lemma *i-append-Nil*[*simp*]: $[] \frown f = f$
by (*simp add*: *i-append-def*)

lemma *i-append-assoc*[*simp*]: $xs \frown (ys \frown f) = (xs\ @\ ys) \frown f$
apply (*case-tac ys = []*, *simp*)
apply (*fastforce simp*: *expand-ilist-eq i-append-def nth-append*)
done

lemma *i-append-Cons*: $(x\ \#\ xs) \frown f = [x] \frown (xs \frown f)$
by *simp*

lemma *i-append-eq-i-append-conv*[*simp*]:
  $length\ xs = length\ ys \Longrightarrow$
  $(xs \frown f = ys \frown g) = (xs = ys \wedge f = g)$
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

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 *fastforce*
apply (*fastforce 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 *fastforce*
done

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

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

lemma *i-append-hd*: $(xs \frown f)\ 0 = (if\ xs = []\ then\ f\ 0\ else\ hd\ xs)$
by (*simp add*: *hd-eq-first*)

lemma *i-append-hd2*[*simp*]: $xs \neq [] \Longrightarrow (xs \frown f)\ 0 = hd\ xs$
by (*simp add*: *i-append-hd*)

lemma *eq-Nil-i-appendI*: $f = g \Longrightarrow f = [] \frown g$
by *simp*

lemma *i-append-eq-i-appendI*:
$[\![\ xs\ @\ xs' = ys;\ f = xs' \frown g\ ]\!] \Longrightarrow xs \frown f = ys \frown g$
by *simp*

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*)

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*)

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*)

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*

lemma *ex-o-conv*: $(\exists\, h.\ g = f \circ h) = (\forall\, y{\in}range\ g.\ \exists\, x.\ y = f\ x)$
apply (*rule iffI*)
apply *fastforce*
apply (*simp add*: *expand-ilist-eq*)
apply (*rule-tac x*=$\lambda x.\ (SOME\ y.\ g\ x = f\ y)$ **in** *exI*)
apply (*fastforce intro*: *someI-ex*)
done

lemma *o-inj-on*:
$[\![\ f \circ g = f \circ h;\ inj\text{-}on\ f\ (range\ g \cup range\ h)\ ]\!] \Longrightarrow g = h$

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

lemma *inj-on-o-eq-o*:
  inj-on f (*range g* ∪ *range h*) ⟹
  (*f* ∘ *g* = *f* ∘ *h*) = (*g* = *h*)
apply (*rule iffI*)
 apply (*rule o-inj-on*, *assumption*+)
apply *simp*
done

lemma *o-injective*: ⟦ *f* ∘ *g* = *f* ∘ *h*; *inj f* ⟧ ⟹ *g* = *h*
by (*simp add*: *expand-ilist-eq inj-on-def*)

lemma *inj-o-eq-o*: *inj f* ⟹ (*f* ∘ *g* = *f* ∘ *h*) = (*g* = *h*)
apply (*rule iffI*)
 apply (*rule o-injective*, *assumption*+)
apply *simp*
done

lemma *inj-oI*: *inj f* ⟹ *inj* (λ*g*. *f* ∘ *g*)
apply (*simp add*: *inj-on-def*)
apply (*blast intro*: *o-inj-on*[*unfolded inj-on-def*])
done

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

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

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

lemma *o-idI*: $\forall x.\ x \in range\ g \longrightarrow f\ x = x \Longrightarrow f \circ g = g$
by (*simp add*: *expand-ilist-eq*)

lemma *o-fun-upd*[*simp*]: $y \notin range\ g \Longrightarrow f\ (y := x) \circ g = f \circ g$
by (*fastforce simp*: *expand-ilist-eq*)

lemma *range-i-append*[*simp*]: $range\ (xs \frown f) = set\ xs \cup range\ f$
by (*fastforce simp*: *in-set-conv-nth i-append-nth*)

lemma *set-subset-i-append*: $set\ xs \subseteq range\ (xs \frown f)$
by *simp*

lemma *range-subset-i-append*: $range\ f \subseteq range\ (xs \frown f)$
by *simp*

lemma *range-ConsD*: $y \in range\ ([x] \frown f) \Longrightarrow y = x \vee y \in range\ f$
by *simp*

lemma *range-o* [*simp*]: $range\ (f \circ g) = f\ `\ range\ g$
by (*simp add*: *image-comp*)

lemma *in-range-conv-decomp*:
$(x \in range\ f) = (\exists\, xs\ g.\ f = xs \frown ([x] \frown g))$
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

lemma *i-append-nth-Cons-0*[*simp*]: $((x \# xs) \frown f)\ 0 = x$
by *simp*

lemma *i-append-nth-Cons-Suc*[*simp*]:
$((x \# xs) \frown f)\ (Suc\ n) = (xs \frown f)\ n$
by (*simp add*: *i-append-nth*)

lemma *i-append-nth-Cons*:
$([x] \frown f)\ n = (case\ n\ of\ 0 \Rightarrow x \mid Suc\ k \Rightarrow f\ k)$
by (*case-tac n*, *simp-all add*: *i-append-nth*)

lemma *i-append-nth-Cons'*:
$([x] \frown f)\ n = (if\ n = 0\ then\ x\ else\ f\ (n - Suc\ 0))$
by (*case-tac n*, *simp-all add*: *i-append-nth*)

lemma *i-append-nth-length*[*simp*]: $(xs \frown f)\ (length\ xs) = f\ 0$
by *simp*

lemma *i-append-nth-length-plus*[*simp*]: $(xs \frown f)\ (length\ xs + n) = f\ n$
by *simp*

lemma *range-iff*: $(y \in range\ f) = (\exists x.\ y = f\ x)$
by *blast*

lemma *range-ball-nth*: $\forall y{\in}range\ f.\ P\ y \Longrightarrow P\ (f\ x)$
by *blast*

lemma *all-nth-imp-all-range*: $[\![\ \forall x.\ P\ (f\ x); y \in range\ f\ ]\!] \Longrightarrow P\ y$
by *blast*

lemma *all-range-conv-all-nth*: $(\forall y{\in}range\ f.\ P\ y) = (\forall x.\ P\ (f\ x))$
by *blast*

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 (*fastforce simp*: *expand-ilist-eq i-append-nth*)

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*)

lemma *i-append-update-length*[*simp*]:
$(xs \frown f)\ (length\ xs := y) = xs \frown (f(0 := y))$
by (*simp add*: *i-append-update2*)

lemma *range-update-subset-insert*:
$range\ (f(n := x)) \subseteq insert\ x\ (range\ f)$
by *fastforce*

lemma *range-update-subsetI*:
$[\![\ range\ f \subseteq A;\ x \in A\ ]\!] \Longrightarrow range\ (f(n := x)) \subseteq A$
by *fastforce*

lemma *range-update-memI*: $x \in range\ (f(n := x))$
by *fastforce*

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. $(i\text{-}take\ f\ n)\ 0 = f\ n$, $(i\text{-}take\ f\ n)\ 1 = f\ (n+1), \ldots$.

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 *i-take'* :: *'a ilist* ⇒ *nat* ⇒ *'a list* (**infixl** ‹⇓› *100*)
  where *f* ⇓ *n* ≡ *i-take n f*
abbreviation *i-drop'* :: *'a ilist* ⇒ *nat* ⇒ *'a ilist* (**infixl** ‹⇑› *100*)
  where *f* ⇑ *n* ≡ *i-drop n f*

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*

lemma *i-take-first*: *f* ⇓ *Suc 0* = [*f 0*]
by (*simp add*: *i-take-def*)

lemma *i-drop-i-take-1*: *f* ⇑ *n* ⇓ *Suc 0* = [*f n*]
by (*simp add*: *i-drop-def i-take-def*)

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*)

lemma *i-drop-nth*[*simp*]: (*s* ⇑ *n*) *x* = *s* (*n* + *x*)
by (*simp add*: *i-drop-def*)

lemma *i-drop-nth-sub*: *n* ≤ *x* ⟹ (*s* ⇑ *n*) (*x* − *n*) = *s x*
by (*simp add*: *i-drop-def*)

theorem *i-take-nth*[*simp*]: *i* < *n* ⟹ (*f* ⇓ *n*) ! *i* = *f i*
by (*simp add*: *i-take-def*)

lemma *i-take-length*[*simp*]: *length* (*f* ⇓ *n*) = *n*
by (*simp add*: *i-take-def*)

lemma *i-take-0*[*simp*]: *f* ⇓ *0* = []

by (*simp add*: *i-take-def*)

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*

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*)

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*]:
$(xs \frown f) \Downarrow n = (xs \downarrow n)\ @\ (f \Downarrow (n - length\ xs))$
by (*simp add*: *i-append-i-take-if*)

lemma *i-append-i-drop-eq1*:
$n \leq length\ xs \Longrightarrow (xs \frown f) \Uparrow n = (xs \uparrow n) \frown f$
by (*simp add*: *expand-ilist-eq i-append-nth less-diff-conv add.commute*[*of - n*])

lemma *i-append-i-drop-eq2*:
$length\ xs \leq n \Longrightarrow (xs \frown f) \Uparrow n = f \Uparrow (n - length\ xs)$
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*)

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 ⇓ *k* = (*THE xs*. *length xs* = *k* ∧ (∃ *g*. *xs* ⌢ *g* = *f*))
apply (*rule the1I2*)
apply (*rule-tac a*=*f* ⇓ *k* **in** *ex1I*)
apply (*fastforce intro*: *i-append-i-take-i-drop-id*)+
done

lemma *i-drop-the-conv*:
f ⇑ *k* = (*THE g*. (∃ *xs*. *length xs* = *k* ∧ *xs* ⌢ *g* = *f*))
apply (*rule sym*, *rule the1-equality*)
apply (*rule-tac a*=*f* ⇑ *k* **in** *ex1I*)

apply (*rule-tac x=f ⇓ k* **in** *exI*, *simp*)
apply *clarsimp*
apply (*rule-tac x=f ⇓ k* **in** *exI*, *simp*)
done

lemma *i-take-Suc-append*[*simp*]:
((*x* # *xs*) ⌢ *f*) ⇓ *Suc n* = *x* # ((*xs* ⌢ *f*) ⇓ *n*)
by (*simp add*: *expand-list-eq*)

corollary *i-take-Suc-Cons*: ([*x*] ⌢ *f*) ⇓ *Suc n* = *x* # (*f* ⇓ *n*)
by *simp*

lemma *i-drop-Suc-append*[*simp*]: ((*x* # *xs*) ⌢ *f*) ⇑ *Suc n* = ((*xs* ⌢ *f*) ⇑ *n*)
by (*simp add*: *expand-list-eq*)

corollary *i-drop-Suc-Cons*: ([*x*] ⌢ *f*) ⇑ *Suc n* = *f* ⇑ *n*
by *simp*

lemma *i-take-Suc*: *f* ⇓ *Suc n* = *f* 0 # (*f* ⇑ *Suc* 0 ⇓ *n*)
by (*simp add*: *expand-list-eq nth-Cons'*)

lemma *i-take-Suc-conv-app-nth*: *f* ⇓ *Suc n* = (*f* ⇓ *n*) @ [*f n*]
by (*simp add*: *i-take-def*)

lemma *i-drop-i-drop*[*simp*]: *s* ⇑ *a* ⇑ *b* = *s* ⇑ (*a* + *b*)
by (*simp add*: *i-drop-def add.assoc*)

corollary *i-drop-Suc*: *f* ⇑ *Suc* 0 ⇑ *n* = *f* ⇑ *Suc n*
by *simp*

lemma *i-take-commute*: *s* ⇓ *a* ↓ *b* = *s* ⇓ *b* ↓ *a*
by (*simp add*: *ac-simps*)

lemma *i-drop-commute*: *s* ⇑ *a* ⇑ *b* = *s* ⇑ *b* ⇑ *a*
by (*simp add*: *add.commute*[*of a*])

corollary *i-drop-tl*: *f* ⇑ *Suc* 0 ⇑ *n* = *f* ⇑ *n* ⇑ *Suc* 0
by *simp*

lemma *nth-via-i-drop*: (*f* ⇑ *n*) 0 = *x* ⟹ *f n* = *x*
by *simp*

lemma *i-drop-Suc-conv-tl*: [*f n*] ⌢ (*f* ⇑ *Suc n*) = *f* ⇑ *n*
by (*simp add*: *expand-ilist-eq i-append-nth*)

lemma *i-drop-Suc-conv-tl'*: ([*f n*] ⌢ *f*) ⇑ *Suc n* = *f* ⇑ *n*
by (*simp add*: *i-drop-Suc-Cons*)

lemma *i-take-i-drop*: *f* ⇑ *m* ⇓ *n* = *f* ⇓ (*n* + *m*) ↑ *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*)

lemma *i-take-drop*: $f \Downarrow n \uparrow m = f \Uparrow m \Downarrow (n - m)$
by (*simp add*: *expand-list-eq*)

lemma *i-take-o*[*simp*]: $(f \circ g) \Downarrow n = map\ f\ (g \Downarrow n)$
by (*simp add*: *expand-list-eq*)

lemma *i-drop-o*[*simp*]: $(f \circ g) \Uparrow n = f \circ (g \Uparrow n)$
by (*simp add*: *expand-ilist-eq*)

lemma *set-i-take-subset*: $set\ (f \Downarrow n) \subseteq range\ f$
by (*fastforce simp*: *in-set-conv-nth*)

lemma *range-i-drop-subset*: $range\ (f \Uparrow n) \subseteq range\ f$
by *fastforce*

lemma *in-set-i-takeD*: $x \in set\ (f \Downarrow n) \Longrightarrow x \in range\ f$
by (*rule subsetD*[*OF set-i-take-subset*])

lemma *in-range-i-takeD*: $x \in range\ (f \Uparrow n) \Longrightarrow x \in range\ f$
by (*rule subsetD*[*OF range-i-drop-subset*])

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

lemma *i-take-add*: $f \Downarrow (i + j) = (f \Downarrow i) @ (f \Uparrow i \Downarrow j)$

by (*simp add*: *expand-list-eq nth-append*)

lemma *i-append-eq-i-append-conv-if-aux*:
  length xs ≤ *length ys* ⟹
  (*xs* ⌢ *f* = *ys* ⌢ *g*) = (*xs* = *ys* ↓ *length xs* ∧ *f* = (*ys* ↑ *length xs*) ⌢ *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* ⌢ *f* = *ys* ⌢ *g*) =
  (*if length xs* ≤ *length ys*
   then xs = *ys* ↓ *length xs* ∧ *f* = (*ys* ↑ *length xs*) ⌢ *g*
   else xs ↓ *length ys* = *ys* ∧ (*xs* ↑ *length ys*) ⌢ *f* = *g*)
apply (*split if-split*, *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

lemma *i-take-hd-i-drop*: (*f* ⇓ *n*) @ [(*f* ⇑ *n*) *0*] = *f* ⇓ *Suc n*
by (*simp add*: *i-take-Suc-conv-app-nth*)

lemma *id-i-take-nth-i-drop*: *f* = (*f* ⇓ *n*) ⌢ (([*f n*] ⌢ *f*) ⇑ *Suc n*)
by (*simp add*: *i-drop-Suc-Cons*)

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*)

theorem *i-take-induct*:
  ⟦ *P* (*f* ⇓ *0*); ⋀*n*. *P* (*f* ⇓ *n*) ⟹ *P* (*f* ⇓ *Suc n*) ⟧ ⟹ *P* (*f* ⇓ *n*)
by (*rule nat.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*)

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*)

lemma *o-eq-i-append-imp*:
  f ∘ *g* = *ys* ⌢ *i* ⟹
  ∃ *xs h*. *g* = *xs* ⌢ *h* ∧ *map f xs* = *ys* ∧ *f* ∘ *h* = *i*
apply (*rule-tac x*=*g* ⇓ (*length ys*) **in** *exI*)
apply (*rule-tac x*=*g* ⇑ (*length ys*) **in** *exI*)
apply (*frule arg-cong*[**where** *f*=λ*x*. *x* ⇓ *length ys*])
apply (*drule arg-cong*[**where** *f*=λ*x*. *x* ⇑ *length ys*])
apply *simp*
done

corollary *o-eq-i-append-conv*:
  (*f* ∘ *g* = *ys* ⌢ *i*) =
  (∃ *xs h*. *g* = *xs* ⌢ *h* ∧ *map f xs* = *ys* ∧ *f* ∘ *h* = *i*)
by (*fastforce simp*: *o-eq-i-append-imp*)
corollary *i-append-eq-o-conv*:
  (*ys* ⌢ *i* = *f* ∘ *g*) =
  (∃ *xs h*. *g* = *xs* ⌢ *h* ∧ *map f xs* = *ys* ∧ *f* ∘ *h* = *i*)
by (*fastforce simp*: *o-eq-i-append-imp*)

11.1.3 *zip* for infinite lists

definition *i-zip* :: *'a ilist* ⇒ *'b ilist* ⇒ (*'a* × *'b*) *ilist*
  where *i-zip f g* ≡ λ*n*. (*f n*, *g n*)

lemma *i-zip-nth*: (*i-zip f g*) *n* = (*f n*, *g n*)
by (*simp add*: *i-zip-def*)

lemma *i-zip-swap*: (λ(*y*, *x*). (*x*, *y*)) ∘ *i-zip g f* = *i-zip f g*
by (*simp add*: *expand-ilist-eq i-zip-nth*)

lemma *i-zip-i-take*: (*i-zip f g*) ⇓ *n* = *zip* (*f* ⇓ *n*) (*g* ⇓ *n*)
by (*simp add*: *expand-list-eq i-zip-nth*)

lemma *i-zip-i-drop*: (*i-zip f g*) ⇑ *n* = *i-zip* (*f* ⇑ *n*) (*g* ⇑ *n*)
by (*simp add*: *expand-ilist-eq i-zip-nth*)

lemma *fst-o-izip*: *fst* ∘ (*i-zip f g*) = *f*
by (*simp add*: *expand-ilist-eq i-zip-nth*)

lemma *snd-o-i-zip*: *snd* ∘ (*i-zip f g*) = *g*
by (*simp add*: *expand-ilist-eq i-zip-nth*)

lemma *update-i-zip*:
(*i-zip f g*)(*n* := *xy*) = *i-zip* (*f*(*n* := *fst xy*)) (*g*(*n* := *snd xy*))
by (*simp add*: *expand-ilist-eq i-zip-nth*)

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*)

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*)

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*)

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*)

lemma *i-zip-range*: *range* (*i-zip f g*) = { (*f n*, *g n*)| *n*. *True* }
by (*fastforce simp*: *i-zip-nth*)

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

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*)

lemma *i-map2-cong*:
⟦ *xs1* = *xs2*; *ys1* = *ys2*;

$\bigwedge x\ y.\ [\![\ x \in \mathit{range}\ \mathit{xs2};\ y \in \mathit{range}\ \mathit{ys2}\ ]\!] \Longrightarrow f\ x\ y = g\ x\ y\ ]\!] \Longrightarrow$
$\mathit{i\text{-}map2}\ f\ \mathit{xs1}\ \mathit{ys1} = \mathit{i\text{-}map2}\ g\ \mathit{xs2}\ \mathit{ys2}$
by (*simp add*: *fun-eq-iff i-map2-nth*)

lemma *i-map2-eq-conv*:
$(\mathit{i\text{-}map2}\ f\ \mathit{xs}\ \mathit{ys} = \mathit{i\text{-}map2}\ g\ \mathit{xs}\ \mathit{ys}) = (\forall i.\ f\ (\mathit{xs}\ i)\ (\mathit{ys}\ i) = g\ (\mathit{xs}\ i)\ (\mathit{ys}\ i))$
by (*simp add*: *fun-eq-iff i-map2-nth*)

lemma *i-map2-replicate*: $\mathit{i\text{-}map2}\ f\ (\lambda n.\ x)\ (\lambda n.\ y) = (\lambda n.\ f\ x\ y)$
by (*simp add*: *fun-eq-iff i-map2-nth*)

lemma *i-map2-i-zip-conv*:
$\mathit{i\text{-}map2}\ f\ \mathit{xs}\ \mathit{ys} = (\lambda(x,y).\ f\ x\ y) \circ (\mathit{i\text{-}zip}\ \mathit{xs}\ \mathit{ys})$
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 (*gset*: $'a$) *glist* = *FL* $'a$ *list* | *IL* $'a$ *ilist* **for** *map*: *gmap*

definition *glength* :: $'a$ *glist* ⇒ *enat*
where
glength a ≡ *case a of*
FL xs ⇒ *enat* (*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 *gtake* :: *enat* ⇒ $'a$ *glist* ⇒ $'a$ *glist*
where
gtake n a ≡ *case n of*
enat m ⇒ *FL* (*case a of*
FL xs ⇒ *xs* ↓ *m* |
IL f ⇒ *f* ⇓ *m*) |
∞ ⇒ *a*

definition *gdrop* :: *enat* ⇒ $'a$ *glist* ⇒ $'a$ *glist*
where

```
  gdrop n a ≡ case n of
    enat m ⇒ (case a of
      FL xs ⇒ FL (xs ↑ m) |
      IL f  ⇒ IL (f ⇑ m)) |
    ∞ ⇒ FL []

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 g-take' :: 'a glist ⇒ enat ⇒ 'a glist  (infixl ‹↓g› 100)
  where a ↓g n ≡ gtake n a

abbreviation g-drop' :: 'a glist ⇒ enat ⇒ 'a glist  (infixl ‹↑g› 100)
  where a ↑g n ≡ gdrop n a
```

11.2.2 *glength*

```
lemma glength-fin[simp]: glength (FL xs) = enat (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 (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 = eSuc 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: eSuc-enat)
   apply (subgoal-tac a' ≠ [])
    prefer 2
```

apply (*rule ccontr*, *simp*)
apply *simp*
apply *simp*
apply (*rename-tac f*)
apply (*case-tac n*, *simp add*: *eSuc-enat*)
apply (*rule-tac x=f 0* **in** *exI*)
apply (*rule-tac x=IL* (*f* ⇑ *Suc 0*) **in** *exI*)
apply (*simp add*: *i-take-first*[*symmetric*])
apply (*clarsimp*, *rename-tac x b*)
apply (*case-tac a*)
apply (*case-tac b*)
apply (*simp add*: *eSuc-enat*)+
apply (*case-tac b*)
apply (*simp add*: *eSuc-enat*)+
done

lemma *gSuc-glength-conv*:
(*eSuc 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

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* [] ∧ *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* [] ∧ *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* = ∞ ∨ *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 < ∞ ⟹ (*a* @$_g$ *b* = *a* @$_g$ *c*) = (*b* = *c*)
by *fastforce*

11.2.4 *gmap*

lemma *gmap-gappend*[*simp*]: *gmap f* (*a* @$_g$ *b*) = *gmap f a* @$_g$ *gmap f b*
by (*unfold gappend-def*, *induct a*, *induct b*, *simp*+)

lemmas *gmap-gmap*[*simp*] = *glist.map-comp*

lemma *gmap-eq-conv*[*simp*]: (*gmap f a* = *gmap g a*) = (∀ *x*∈*gset a*. *f x* = *g x*)
apply (*case-tac a*)
apply (*simp add*: *o-eq-conv*)+
done

lemmas *gmap-cong* = *glist.map-cong*

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* ⟹ *glength a* = *glength b*
by (*drule arg-cong*[**where** *f*=*glength*], *simp*)

11.2.5 *gset*

lemma *gset-gappend*[*simp*]:
gset (*a* @$_g$ *b*) =
(*case a of FL a'* ⇒ *set a'* ∪ *gset b* | *IL a'* ⇒ *range a'*)
by (*unfold gappend-def*, *case-tac a*, *case-tac b*, *simp*+)

lemma *gset-gappend-if*:
gset (*a* @$_g$ *b*) =
(*if glength a* < ∞ *then gset a* ∪ *gset b else gset a*)
by (*unfold gappend-def*, *case-tac a*, *case-tac b*, *simp*+)

lemma *gset-empty*[*simp*]: (*gset a* = {}) = (*a* = *FL* [])
by (*case-tac a*, *simp*+)

lemmas *gset-gmap*[*simp*] = *glist.set-map*

lemma *icard-glength*: *icard* (*gset a*) ≤ *glength a*
apply (*unfold icard-def glength-def*)
apply (*case-tac a*)
apply (*simp add*: *card-length*)+
done

11.2.6 $!_g$ – gnth

lemma *gnth-gCons-0*[*simp*]: $(x \#_g a) !_g 0 = x$
by (*unfold gCons-def gnth-def*, *case-tac a*, *simp+*)

lemma *gnth-gCons-Suc*[*simp*]: $(x \#_g a) !_g Suc\ n = a !_g n$
by (*unfold gCons-def gnth-def*, *case-tac a*, *simp+*)

lemma *gnth-gappend*:
$(a @_g b) !_g n =$
(*if enat n* $<$ *glength a then a* $!_g$ *n*
else b $!_g$ $(n - the\text{-}enat\ (glength\ a)))$
apply (*unfold glength-def gappend-def gCons-def gnth-def*)
apply (*case-tac a*, *case-tac b*)
apply (*simp add*: *nth-append*)+
done

lemma *gnth-gappend-length-plus*[*simp*]: $(FL\ xs @_g b) !_g (length\ xs + n) = b !_g n$
by (*simp add*: *gnth-gappend*)

lemma *gmap-gnth*[*simp*]: $enat\ n < glength\ a \Longrightarrow gmap\ f\ a !_g n = f\ (a !_g n)$
by (*unfold gnth-def*, *case-tac a*, *simp+*)

lemma *in-gset-cong-gnth*: $(x \in gset\ a) = (\exists i.\ enat\ i < glength\ a \wedge a !_g i = x)$
apply (*unfold gnth-def*, *case-tac a*)
apply (*fastforce simp*: *in-set-conv-nth*)+
done

11.2.7 *gtake* and *gdrop*

lemma *gtake-0*[*simp*]: $a \downarrow_g 0 = FL\ []$
by (*unfold gtake-def*, *case-tac a*, *simp+*)

lemma *gdrop-0*[*simp*]: $a \uparrow_g 0 = a$
by (*unfold gdrop-def*, *case-tac a*, *simp+*)

lemma *gtake-Infty*[*simp*]: $a \downarrow_g \infty = a$
by (*unfold gtake-def*, *case-tac a*, *simp+*)

lemma *gdrop-Infty*[*simp*]: $a \uparrow_g \infty = FL\ []$
by (*unfold gdrop-def*, *case-tac a*, *simp+*)

lemma *gtake-all*[*simp*]: $glength\ a \leq n \Longrightarrow a \downarrow_g n = a$
by (*unfold gtake-def*, *case-tac a*, *case-tac n*, *simp+*)

lemma *gdrop-all*[*simp*]: $glength\ a \leq n \Longrightarrow a \uparrow_g n = FL\ []$
by (*unfold gdrop-def*, *case-tac a*, *case-tac n*, *simp+*)

lemma *gtake-eSuc-gCons*[*simp*]: $(x \#_g a) \downarrow_g (eSuc\ n) = x \#_g a \downarrow_g n$
by (*unfold gtake-def gCons-def*, *case-tac n*, *case-tac a*, *simp-all add*: *eSuc-enat*)

lemma *gdrop-eSuc-gCons*[*simp*]: $(x \#_g a) \uparrow_g (eSuc\ n) = a \uparrow_g n$
by (*unfold gdrop-def gCons-def*, *case-tac n*, *case-tac a*, *simp-all add*: *eSuc-enat*)

lemma *gtake-eSuc*: $a \neq FL\ [] \Longrightarrow a \downarrow_g (eSuc\ n) = a\ !_g\ 0 \#_g (a \uparrow_g (eSuc\ 0)) \downarrow_g n$
apply (*unfold gtake-def gdrop-def gnth-def gCons-def*)
apply (*case-tac n*)
apply (*case-tac a*)
apply (*simp add*: *eSuc-enat 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

lemma *gdrop-eSuc*: $a \uparrow_g (eSuc\ n) = a \uparrow_g (eSuc\ 0) \uparrow_g n$
by (*unfold gtake-def gdrop-def gnth-def gCons-def*, *case-tac n*, *case-tac a*, *simp-all add*: *eSuc-enat*)

lemma *gnth-via-grop*: $a \uparrow_g (enat\ n) = x \#_g b \Longrightarrow a\ !_g\ n = x$
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 (*fastforce intro*: *nth-via-i-drop*)+
done

lemma *gtake-eSuc-conv-gapp-gnth*:
$enat\ n < glength\ a \Longrightarrow a \downarrow_g enat\ (Suc\ n) = a \downarrow_g (enat\ n) @_g FL\ [a\ !_g\ n]$
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

lemma *gdrop-eSuc-conv-tl*:
$enat\ n < glength\ a \Longrightarrow a\ !_g\ n \#_g a \uparrow_g enat\ (Suc\ n) = a \uparrow_g enat\ n$
apply (*unfold glength-def gdrop-def gappend-def gnth-def gCons-def*)
apply (*case-tac a*)
apply (*simp add*: *Cons-nth-drop-Suc i-drop-Suc-conv-tl*)+
done

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*+)

lemma *glength-drop*[*simp*]: $glength\ (a \uparrow_g (enat\ n)) = glength\ a - (enat\ n)$
by (*unfold glength-def gdrop-def*, *case-tac a*, *case-tac n*, *simp*+)

end

12 Prefices on finite and infinite lists

theory *ListInf-Prefix*
imports *HOL−Library.Sublist ListInf*
begin

12.1 Additional list prefix results

lemma *prefix-eq-prefix-take-ex*: *prefix xs ys* = (∃ *n*. *ys* ↓ *n* = *xs*)
apply (*unfold prefix-def*, *safe*)
apply (*rule-tac x*=*length xs* **in** *exI*, *simp*)
apply (*rule-tac x*=*ys* ↑ *n* **in** *exI*, *simp*)
done

lemma *prefix-take-eq-prefix-take-ex*: (*ys* ↓ (*length xs*) = *xs*) = (∃ *n*. *ys* ↓ *n* = *xs*)
by (*fastforce simp*: *min-def*)

lemma *prefix-eq-prefix-take*: *prefix xs ys* = (*ys* ↓ (*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* ↓ (*length xs*) = *xs* ∧ *xs* ≠ *ys*) =
((∃ *n*. *ys* ↓ *n* = *xs*) ∧ *xs* ≠ *ys*)
by (*simp add*: *prefix-take-eq-prefix-take-ex*)

lemma *strict-prefix-eq-strict-prefix-take-ex*: *strict-prefix xs ys* = ((∃ *n*. *ys* ↓ *n* = *xs*)
∧ *xs* ≠ *ys*)
by (*simp add*: *strict-prefix-def prefix-eq-prefix-take-ex*)
lemma *strict-prefix-eq-strict-prefix-take*: *strict-prefix xs ys* = (*ys* ↓ (*length xs*) =
xs ∧ *xs* ≠ *ys*)
by (*simp only*: *strict-prefix-eq-strict-prefix-take-ex strict-prefix-take-eq-strict-prefix-take-ex*)

lemma *take-imp-prefix*: *prefix* (*xs* ↓ *n*) *xs*
by (*rule take-is-prefix*)

lemma *eq-imp-prefix*: *xs* = (*ys*::'*a list*) ⟹ *prefix xs ys*
by *simp*

lemma *le-take-imp-prefix*: *a* ≤ *b* ⟹ *prefix* (*xs* ↓ *a*) (*xs* ↓ *b*)
by (*fastforce simp*: *prefix-eq-prefix-take-ex min-def*)

lemma *take-prefix-imp-le*:
⟦ *a* ≤ *length xs*; *prefix* (*xs* ↓ *a*) (*xs* ↓ *b*) ⟧ ⟹ *a* ≤ *b*
by (*drule prefix-length-le*, *simp*)

lemma *take-prefixeq-le-conv*:
a ≤ *length xs* ⟹ *prefix* (*xs* ↓ *a*) (*xs* ↓ *b*) = (*a* ≤ *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*]: *prefix a* (*a* @ *b*)
by (*unfold prefix-def*, *blast*)

lemma *prefix-imp-take-eq*:
⟦ *n* ≤ *length xs*; *prefix xs ys* ⟧ ⟹ *xs* ↓ *n* = *ys* ↓ *n*
by (*clarsimp simp*: *prefix-def*)

lemma *prefix-length-le-eq-conv*: (*prefix xs ys* ∧ *length ys* ≤ *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 ≤ *length ys* ⟹ *prefix* (*ys* ↓ *length xs*) *xs* = *prefix xs ys*
by (*fastforce simp*: *prefix-eq-prefix-take*)

lemma *append-eq-imp-take*:
⟦ *k* ≤ *length xs*; *length r1* = *k*; *r1* @ *r2* = *xs* ⟧ ⟹ *r1* = *xs* ↓ *k*
by *fastforce*

lemma *take-the-conv*:
xs ↓ *k* = (*if length xs* ≤ *k then xs else* (*THE r*. *length r* = *k* ∧ (∃ *r2*. *r* @ *r2* = *xs*)))
apply (*clarsimp simp*: *linorder-not-le*)
apply (*rule the1I2*)
apply (*simp add*: *Ex1-def*)
apply (*rule-tac x*=*xs* ↓ *k* **in** *exI*)
apply (*intro conjI*)
apply (*simp add*: *min-eqR*)
apply (*rule-tac x*=*xs* ↑ *k* **in** *exI*, *simp*)
apply *fastforce*
apply *fastforce*
done

lemma *prefix-refl*: *prefix xs* (*xs*::'*a list*)
by (*rule prefix-order.order-refl*)

lemma *prefix-trans*: ⟦ *prefix xs ys*; *prefix* (*ys*::'*a list*) *zs* ⟧ ⟹ *prefix xs zs*
by (*rule prefix-order.order-trans*)

lemma *prefixeq-antisym*: ⟦ *prefix xs ys*; *prefix* (*ys*::'*a list*) *xs* ⟧ ⟹ *xs* = *ys*
by (*rule prefix-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* $\equiv$ (*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* $\equiv$ (*fst p* = *snd p*)

lemma *mirror-pair-equal*: *equal-pair* (*mirror-pair p*) = (*equal-pair p*)
by (*fastforce 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* $\leq$ *length xs*
by (*induct xs*, *simp-all*)

lemma *equal-pair-count-0*:
  fst (*hd ps*) $\neq$ *snd* (*hd ps*) $\Longrightarrow$ *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*;
    $\forall$ *i*<*length ps2*. *equal-pair* (*ps1* ! *i*) = *equal-pair*(*ps2* ! *i*) $]\!]$
  $\Longrightarrow$ *equal-pair-count ps1* = *equal-pair-count ps2*
apply (*induct rule*: *list-induct2*)
 apply *simp*
apply (*fastforce 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*)
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 if-split-asm*)
apply (*case-tac i*)
apply (*simp add*: *equal-pair-def*)+
done

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

corollary *equal-pair-count-maximality1a*[*rule-format*]:
 equal-pair-count ps = *length ps* ∨ ¬ *equal-pair* (*ps*!*equal-pair-count ps*)
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*)
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

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*)
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*] *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*
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*)

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*)

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*)

lemma *inf-prefix-length-correct*:
i < *inf-prefix-length xs ys* ⟹ *xs* ! *i* = *ys* ! *i*
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

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*)
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

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*:
$n \leq$ *inf-prefix-length xs ys* $\Longrightarrow$ *xs* $\downarrow$ *n* = *ys* $\downarrow$ *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* ⟧ $\Longrightarrow$ *hd xs* ≠ *hd ys*
apply (*rule ccontr*)
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* $\downarrow$ (*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* $\downarrow$ (*inf-prefix-length xs ys*)
by (*simp add*: *inf-prefix-def*)

lemma *inf-prefix-takeR*: *xs* ⊓ *ys* = *ys* $\downarrow$ (*inf-prefix-length xs ys*)
by (*subst inf-prefix-commute*, *subst inf-prefix-length-commute*, *rule inf-prefix-takeL*)

lemma *inf-prefix-correct*: *i* < *length* (*xs* ⊓ *ys*) $\Longrightarrow$ *xs* ! *i* = *ys* ! *i*
by (*simp add*: *length-inf-prefix inf-prefix-length-correct*)

corollary *inf-prefix-correctL*:
i < *length* (*xs* ⊓ *ys*) $\Longrightarrow$ (*xs* ⊓ *ys*) ! *i* = *xs* ! *i*
by (*simp add*: *inf-prefix-takeL*)

corollary *inf-prefix-correctR*:
i < *length* (*xs* ⊓ *ys*) $\Longrightarrow$ (*xs* ⊓ *ys*) ! *i* = *ys* ! *i*
by (*simp add*: *inf-prefix-takeR*)

lemma *inf-prefix-take-correct*:
$n \leq$ *length* (*xs* ⊓ *ys*) $\Longrightarrow$ *xs* $\downarrow$ *n* = *ys* $\downarrow$ *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*)

lemma *hd-inf-prefix-Nil*: *hd xs* ≠ *hd ys* ⟹ *xs* ⊓ *ys* = []
by (*simp add*: *inf-prefix-def hd-inf-prefix-length-0*)

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*)
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*)

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*

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*: *prefix* (*xs* ⊓ *ys*) *xs*
by (*simp add*: *inf-prefix-takeL take-imp-prefix*)

lemma *inf-prefix-le2*: *prefix* (*xs* ⊓ *ys*) *ys*

by (*simp add*: *inf-prefix-takeR take-imp-prefix*)

lemma *le-inf-prefix-iff*: *prefix x* (*y* ⊓ *z*) = (*prefix x y* ∧ *prefix x z*)
apply (*rule iffI*)
apply (*blast intro*: *prefix-order.order-trans inf-prefix-le1 inf-prefix-le2*)
apply (*clarsimp simp*: *prefix-def*)
done

lemma *le-imp-le-inf-prefix*: ⟦ *prefix x y*; *prefix x z* ⟧ ⟹ *prefix x* (*y* ⊓ *z*)
by (*rule le-inf-prefix-iff*[*THEN iffD2*], *simp*)

interpretation *prefix*:
semilattice-inf
(⊓) :: *'a list* ⇒ *'a list* ⇒ *'a list*
prefix
strict-prefix
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

definition *iprefix* :: *'a list* ⇒ *'a ilist* ⇒ *bool* (**infixl** ‹⊑› *50*)
where *xs* ⊑ *f* ≡ ∃ *g*. *f* = *xs* ⌢ *g*

lemma *iprefix-eq-iprefix-take*: (*xs* ⊑ *f*) = (*f* ⇓ *length xs* = *xs*)
apply (*unfold iprefix-def*)
apply (*rule iffI*)
apply *clarsimp*
apply (*rule-tac x*=*f* ⇑ (*length xs*) **in** *exI*)
apply (*subst i-append-i-take-i-drop-id*[**where** *n*=*length xs*, *symmetric*], *simp*)
done

lemma *iprefix-take-eq-iprefix-take-ex*:
(*f* ⇓ *length xs* = *xs*) = (∃ *n*. *f* ⇓ *n* = *xs*)
apply (*rule iffI*)
apply (*rule-tac x*=*length xs* **in** *exI*, *assumption*)
apply *clarsimp*
done

lemma *iprefix-eq-iprefix-take-ex*: (*xs* ⊑ *f*) = (∃ *n*. *f* ⇓ *n* = *xs*)
by (*simp add*: *iprefix-eq-iprefix-take iprefix-take-eq-iprefix-take-ex*)

lemma *i-take-imp-iprefix*[*intro*]: *f* ⇓ *n* ⊑ *f*
by (*simp add*: *iprefix-eq-iprefix-take*)

lemma *i-take-prefix-le-conv*: *prefix* $(f \Downarrow a)$ $(f \Downarrow b) = (a \leq b)$
by (*fastforce simp*: *prefix-eq-prefix-take list-eq-iff*)

lemma *i-append-imp-iprefix*[*simp*,*intro*]: $xs \sqsubseteq xs \frown f$
by (*simp add*: *iprefix-def*)

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*)

lemma *prefixeq-iprefix-trans*: $[\![\mathit{prefix}\ xs\ ys;\ ys \sqsubseteq f]\!] \Longrightarrow xs \sqsubseteq f$
by (*fastforce simp*: *iprefix-eq-iprefix-take-ex prefix-eq-prefix-take-ex*)

lemma *i-take-length-prefix-conv*: *prefix* $(f \Downarrow \mathit{length}\ xs)\ xs = (xs \sqsubseteq f)$
by (*simp add*: *iprefix-eq-iprefix-take prefix-length-le-eq-conv*[*symmetric*])

lemma *iprefixI*[*intro?*]: $f = xs \frown g \Longrightarrow xs \sqsubseteq f$
by (*unfold iprefix-def*, *simp*)

lemma *iprefixE*[*elim?*]: $[\![xs \sqsubseteq f;\ \bigwedge g.\ f = xs \frown g \Longrightarrow C]\!] \Longrightarrow C$
by (*unfold iprefix-def*, *blast*)

lemma *Nil-iprefix*[*iff*]: $[] \sqsubseteq f$
by (*unfold iprefix-def*, *simp*)

lemma *same-prefix-iprefix*[*simp*]: $(xs\ @\ ys \sqsubseteq xs \frown f) = (ys \sqsubseteq f)$
by (*simp add*: *iprefix-eq-iprefix-take*)

lemma *prefix-iprefix*[*simp*]: *prefix xs ys* $\Longrightarrow xs \sqsubseteq ys \frown f$
by (*clarsimp simp*: *prefix-def iprefix-def i-append-assoc*[*symmetric*] *simp del*: *i-append-assoc*)

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*:
$[\![xs \sqsubseteq ys \frown f;\ \mathit{length}\ xs \leq \mathit{length}\ ys]\!] \Longrightarrow \mathit{prefix}\ xs\ ys$
by (*clarsimp simp*: *iprefix-eq-iprefix-take-ex take-is-prefix*)

lemma *iprefix-i-append*:
$(xs \sqsubseteq ys \frown f) = (\mathit{prefix}\ xs\ ys \vee (\exists\, zs.\ xs = ys\ @\ zs \wedge zs \sqsubseteq f))$
apply (*rule iffI*)
apply (*case-tac length xs* $\leq$ *length ys*)
apply (*blast intro*: *iprefix-length-le-imp-prefix*)
apply (*rule disjI2*)
apply (*rule-tac x=f* $\Downarrow$ (*length xs* $-$ *length ys*) **in** *exI*)
apply (*simp add*: *iprefix-eq-iprefix-take*)
apply *fastforce*
done

lemma *i-append-one-iprefix*:
  $xs \sqsubseteq f \Longrightarrow xs \mathbin{@} [f\ (length\ xs)] \sqsubseteq f$
by (*simp add*: *iprefix-eq-iprefix-take i-take-Suc-conv-app-nth*)

lemma *iprefix-same-length-le*:
  $[\![\ xs \sqsubseteq f;\ ys \sqsubseteq f;\ length\ xs \leq length\ ys\ ]\!] \Longrightarrow prefix\ xs\ ys$
by (*clarsimp simp*: *iprefix-eq-iprefix-take-ex i-take-prefix-le-conv*)

lemma *iprefix-same-cases*:
  $[\![\ xs \sqsubseteq f;\ ys \sqsubseteq f\ ]\!] \Longrightarrow prefix\ xs\ ys \vee prefix\ ys\ xs$
apply (*case-tac length xs* $\leq$ *length ys*)
apply (*simp add*: *iprefix-same-length-le*)+
done

lemma *set-mono-iprefix*: $xs \sqsubseteq f \Longrightarrow set\ xs \subseteq range\ f$
by (*unfold iprefix-def*, *fastforce*)

end
theory *ListInfinite*
imports
  CommonSet/SetIntervalStep
  ListInf/ListInf-Prefix
begin
end