

Infinite Lists

David Trachtenherz

September 13, 2023

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).

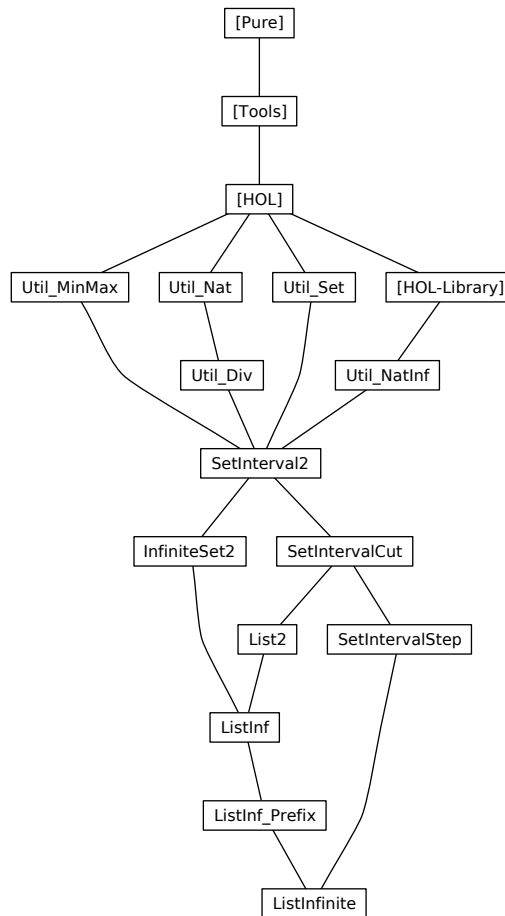
Contents

1	Convenience results for set quantifiers	3
1.1	Some auxiliary results for HOL rules	3
1.1.1	Some auxiliary results for <i>Let</i>	3
1.1.2	Some auxiliary <i>if</i> -rules	3
1.1.3	Some auxiliary rules for function composition	3
1.2	Some auxiliary lemmata for quantifiers	3
1.2.1	Auxiliary results for universal and existential quantifiers	3
1.2.2	Auxiliary results for <i>empty</i> sets	4
1.2.3	Some auxiliary results for subset and membership relation	4
2	Order and linear order: min and max	4
2.1	Additional lemmata about <i>min</i> and <i>max</i>	4
3	Results for natural arithmetics with infinity	5
3.1	Arithmetic operations with <i>enat</i>	6
3.1.1	Additional definitions	6
3.1.2	Addition, difference, order	6
3.1.3	Multiplication and division	7
4	Results for natural arithmetics	10
4.1	Some convenience arithmetic lemmata	10
4.2	Additional facts about inequalities	13
4.3	Inequalities for <code>Suc</code> and <code>pred</code>	13
4.4	Additional facts about cancellation in (in-)equalities	14

5	Results for division and modulo operators on integers	17
5.1	Additional (in-)equalities with <i>div</i> and <i>mod</i>	17
5.2	Additional results for addition and subtraction with <i>mod</i>	18
5.2.1	Divisor subtraction with <i>div</i> and <i>mod</i>	21
5.2.2	Modulo equality and modulo of difference	22
5.3	Some additional lemmata about integer <i>div</i> and <i>mod</i>	23
5.4	Some further (in-)equality results for <i>div</i> and <i>mod</i>	27
5.5	Additional multiplication results for <i>mod</i> and <i>div</i>	29
5.6	Some factor distribution facts for <i>mod</i>	30
5.7	More results about quotient <i>div</i> with addition and subtraction	31
5.8	Further results about <i>div</i> and <i>mod</i>	35
5.8.1	Some auxiliary facts about <i>mod</i>	35
5.8.2	Some auxiliary facts about <i>div</i>	37
6	Sets of natural numbers	42
6.1	Auxiliary results for monotonic, injective and surjective functions over sets	42
6.1.1	Monotonicity	42
6.1.2	Injectivity	43
6.1.3	Surjectivity	44
6.1.4	Induction over natural sets	45
6.1.5	Monotonicity and injectivity of arithmetic operators	47
6.2	<i>Min</i> and <i>Max</i> elements of a set	48
6.2.1	Basic results, as for <i>Least</i>	48
6.2.2	<i>Max</i> for sets over <i>enat</i>	54
6.2.3	<i>Min</i> and <i>Max</i> for set operations	55
6.3	Some auxiliary results for set operations	59
6.3.1	Some additional abbreviations for relations	59
6.3.2	Auxiliary results for <i>singletons</i>	59
6.3.3	Auxiliary results for <i>finite</i> and <i>infinite</i> sets	60
6.3.4	Some auxiliary results for disjoint sets	62
6.3.5	Some auxiliary results for subset relation	63
6.3.6	Auxiliary results for intervals from <i>SetInterval</i>	63
6.3.7	Auxiliary results for <i>card</i>	67
7	Cutting linearly ordered and natural sets	69
7.1	Set restriction	69
7.2	Cut operators for sets/intervals	71
7.2.1	Definitions and basic lemmata for cut operators	71
7.2.2	Basic results for cut operators	73
7.2.3	Relations between cut operators	80
7.2.4	Function images with cut operators	81
7.2.5	Finiteness and cardinality with cut operators	82
7.2.6	Cutting a set at <i>Min</i> or <i>Max</i> element	83

7.2.7	Cut operators with intervals from <code>SetInterval</code>	85
7.2.8	Mirroring finite natural sets between their <i>Min</i> and <i>Max</i> element	87
8	Stepping through sets of natural numbers	93
8.1	Function <i>inext</i> and <i>iprev</i> for stepping through natural sets . . .	94
8.2	<i>inext-nth</i> and <i>iprev-nth</i> – nth element of a natural set	118
8.3	Induction over arbitrary natural sets using the functions <i>inext</i> and <i>iprev</i>	128
8.4	Natural intervals with <i>inext</i> and <i>iprev</i>	132
8.5	Further result for <i>inext-nth</i> and <i>iprev-nth</i>	134
9	Additional definitions and results for lists	136
9.1	Additional definitions and results for lists	136
9.1.1	Additional lemmata about list emptiness	137
9.1.2	Additional lemmata about <i>take</i> , <i>drop</i> , <i>hd</i> , <i>last</i> , <i>nth</i> and <i>filter</i>	137
9.1.3	Ordered lists	143
9.1.4	Additional definitions and results for sublists	148
9.1.5	Natural set images with lists	153
9.1.6	Mapping lists of functions to lists	155
9.1.7	Mapping functions with two arguments to lists	157
10	Set operations with results of type <code>enat</code>	161
10.1	Set operations with <i>enat</i>	161
10.1.1	Basic definitions	161
10.2	Results for <i>icard</i>	161
11	Additional definitions and results for lists	167
11.1	Infinite lists	167
11.1.1	Appending a functions to a list	167
11.1.2	<i>take</i> and <i>drop</i> for infinite lists	173
11.1.3	<i>zip</i> for infinite lists	179
11.1.4	Mapping functions with two arguments to infinite lists	180
11.2	Generalised lists as combination of finite and infinite lists . . .	182
11.2.1	Basic definitions	182
11.2.2	<i>glength</i>	183
11.2.3	@ _{<i>g</i>} – <i>gappend</i>	184
11.2.4	<i>gmap</i>	185
11.2.5	<i>gset</i>	185
11.2.6	! _{<i>g</i>} – <i>gnth</i>	186
11.2.7	<i>gtake</i> and <i>gdrop</i>	186

12 Prefices on finite and infinite lists	188
12.1 Additional list prefix results	188
12.2 Counting equal pairs	190
12.3 Prefix length	192
12.4 Prefix infimum	194
12.5 Prefices for infinite lists	196



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*: $f (\text{let } x=a \text{ in } g \ x) = (\text{let } x=a \text{ in } f (g \ x))$ **by** *simp*

1.1.2 Some auxiliary *if*-rules

lemma *if-P'*: $\llbracket P; x = z \rrbracket \Longrightarrow (\text{if } P \text{ then } x \text{ else } y) = z$ **by** *simp*

lemma *if-not-P'*: $\llbracket \neg P; y = z \rrbracket \Longrightarrow (\text{if } P \text{ then } x \text{ else } y) = z$ **by** *simp*

lemma *if-P-both*: $\llbracket Q \ x; Q \ y \rrbracket \Longrightarrow Q (\text{if } P \text{ then } x \text{ else } y)$ **by** *simp*

lemma *if-P-both-in-set*: $\llbracket x \in s; y \in s \rrbracket \Longrightarrow (\text{if } P \text{ then } x \text{ else } y) \in s$ **by** *simp*

1.1.3 Some auxiliary rules for function composition

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

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

1.2 Some auxiliary lemmata for quantifiers

1.2.1 Auxiliary results for universal and existential quantifiers

lemma *ball-cong2*:

$\llbracket I \subseteq A; \forall x \in A. f \ x = g \ x \rrbracket \Longrightarrow (\forall x \in I. P (f \ x)) = (\forall x \in I. P (g \ x))$ **by** *fastforce*

lemma *bex-cong2*:

$\llbracket I \subseteq A; \forall x \in I. f \ x = g \ x \rrbracket \Longrightarrow (\exists x \in I. P (f \ x)) = (\exists x \in I. P (g \ x))$ **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*:

$$\llbracket P = P'; \bigwedge x. P x \implies Q x = Q' x \rrbracket \implies (\forall x. P x \longrightarrow Q x) = (\forall x. P' x \longrightarrow Q' x)$$

by *blast*

lemma *ex-imp-eqI*:

$$\llbracket P = P'; \bigwedge x. P x \implies Q x = Q' x \rrbracket \implies (\exists x. P x \wedge Q x) = (\exists x. P' x \wedge Q' x)$$

by *blast*

1.2.2 Auxiliary results for empty sets

lemma *empty-imp-not-in*: $x \notin \{\}$ **by** *blast*

lemma *ex-imp-not-empty*: $\exists x. x \in A \implies A \neq \{\}$ **by** *blast*

lemma *in-imp-not-empty*: $x \in A \implies A \neq \{\}$ **by** *blast*

lemma *not-empty-imp-ex*: $A \neq \{\} \implies \exists x. x \in A$ **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*: $\llbracket \exists x \in A. P x; A \subseteq B \rrbracket \implies \exists x \in B. P x$ **by** *blast*

lemma *beX-imp-ex*: $\exists x \in A. P x \implies \exists x. P x$ **by** *blast*

lemma *ball-subset-imp-ball*: $\llbracket \forall x \in B. P x; A \subseteq B \rrbracket \implies \forall x \in A. P x$ **by** *blast*

lemma *all-imp-ball*: $\forall x. P x \implies \forall x \in A. P x$ **by** *blast*

lemma *mem-Collect-eq-not*: $(a \notin \{x. P x\}) = (\neg P a)$ **by** *blast*

lemma *Collect-not-in-imp-not*: $a \notin \{x. P x\} \implies \neg P a$ **by** *blast*

lemma *Collect-not-imp-not-in*: $\neg P a \implies a \notin \{x. P x\}$ **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*: $(z::'a::linorder) < \min x y \implies z < x \wedge z < y$ **by** *simp*

lemma *conj-less-imp-min*: $\llbracket z < x; z < y \rrbracket \implies (z::'a::linorder) < \min x y$ **by** *simp*

lemmas *min-le-iff-conj* = *min.bounded-iff*

lemma *min-le-imp-conj*: $(z::'a::linorder) \leq \min x y \implies z \leq x \wedge z \leq y$ **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*: $\max x y < b \implies x < (b::('a::linorder)) \wedge y < b$ **by** *simp*

lemma *conj-less-imp-max*: $\llbracket x < (b::('a::linorder)); y < b \rrbracket \implies \max x y < b$ **by** *simp*

lemmas *max-le-iff-conj* = *max.bounded-iff*

lemma *max-le-imp-conj*: $\max x y \leq b \implies x \leq (b::('a::linorder)) \wedge y \leq b$ **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: $(a::'a::linorder) \leq b \implies \min x a \leq \min x b$ **and**

min-le-monoL: $(a::'a::linorder) \leq b \implies \min a x \leq \min b x$

by (*fastforce simp: min.mono min-def*)+

lemma

max-le-monoR: $(a::'a::linorder) \leq b \implies \max x a \leq \max x b$ **and**

max-le-monoL: $(a::'a::linorder) \leq b \implies \max a x \leq \max b x$

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*]:

$a \text{ div } b \equiv (\text{case } a \text{ of}$
 $(\text{enat } x) \Rightarrow (\text{case } b \text{ of } (\text{enat } y) \Rightarrow \text{enat } (x \text{ div } y) \mid \infty \Rightarrow 0) \mid$
 $\infty \Rightarrow (\text{case } b \text{ of } (\text{enat } y) \Rightarrow ((\text{case } y \text{ of } 0 \Rightarrow 0 \mid \text{Suc } n \Rightarrow \infty)) \mid \infty \Rightarrow \infty))$)

definition

mod-enat-def [*code del*]:

$a \text{ mod } b \equiv (\text{case } a \text{ of}$
 $(\text{enat } x) \Rightarrow (\text{case } b \text{ of } (\text{enat } y) \Rightarrow \text{enat } (x \text{ mod } y) \mid \infty \Rightarrow a) \mid$
 $\infty \Rightarrow \infty)$

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*: $(x - y = (z::\text{nat})) = (\text{if } y < x \text{ then } x = y + z \text{ else } z = 0)$

by *auto*

lemma *idiff-eq-conv*:

$(x - y = (z::\text{enat})) =$

$(\text{if } y < x \text{ then } x = y + z \text{ else if } x \neq \infty \text{ then } z = 0 \text{ else } z = \infty)$

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*: $y \leq (x::\text{enat}) \implies (z \leq x - y) = (z + y \leq x)$

by (*case-tac x, case-tac y, case-tac z, fastforce+*)

lemma *eSuc-pred*: $0 < n \implies \text{eSuc } (n - \text{eSuc } 0) = n$

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*: $(n::\text{enat}) \leq n + m$
by (*case-tac m*, *case-tac n*, *simp-all*)
lemma *ile-add2*: $(n::\text{enat}) \leq m + n$
by (*simp only: add.commute*[*of m*] *ile-add1*)

lemma *iadd-iless-mono*: $\llbracket (i::\text{enat}) < j; k < l \rrbracket \implies i + k < j + l$
by (*case-tac i*, *case-tac k*, *case-tac j*, *case-tac l*, *simp-all*)

lemma *trans-ile-iadd1*: $i \leq (j::\text{enat}) \implies i \leq j + m$
by (*rule order-trans*[*OF - ile-add1*])
lemma *trans-ile-iadd2*: $i \leq (j::\text{enat}) \implies i \leq m + j$
by (*rule order-trans*[*OF - ile-add2*])

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

lemma *iadd-ileD1*: $m + k \leq (n::\text{enat}) \implies m \leq n$
by (*case-tac m*, *case-tac n*, *case-tac k*, *simp-all*)

lemma *iadd-ileD2*: $m + k \leq (n::\text{enat}) \implies k \leq n$
by (*rule iadd-ileD1*, *simp only: add.commute*[*of m*])

lemma *idiff-ile-mono*: $m \leq (n::\text{enat}) \implies m - l \leq n - l$
by (*case-tac m*, *case-tac n*, *case-tac l*, *simp-all*)

lemma *idiff-ile-mono2*: $m \leq (n::\text{enat}) \implies l - n \leq l - m$
by (*case-tac m*, *case-tac n*, *case-tac l*, *simp-all*, *case-tac l*, *simp-all*)

lemma *idiff-iless-mono*: $\llbracket m < (n::\text{enat}); l \leq m \rrbracket \implies m - l < n - l$
by (*case-tac m*, *case-tac n*, *case-tac l*, *simp-all*, *case-tac l*, *simp-all*)

lemma *idiff-iless-mono2*: $\llbracket m < (n::\text{enat}); m < l \rrbracket \implies l - n \leq l - m$
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*: $0 < n \implies (\infty::\text{enat}) \text{ div } n = \infty$
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*]: $n \neq \infty \implies n \text{ div } (\infty::\text{enat}) = 0$
unfolding *div-enat-def* **by** (*case-tac n, simp-all*)

lemma *idiv-infinity-if*: $n \text{ div } \infty = (\text{if } n = \infty \text{ then } \infty \text{ else } 0::\text{enat})$
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*: $\infty * n = (\text{if } n = 0 \text{ then } 0 \text{ else } \infty::\text{enat})$
by (*case-tac n, simp-all*)
lemma *imult-infinity-right-if*: $n * \infty = (\text{if } n = 0 \text{ then } 0 \text{ else } \infty::\text{enat})$
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*: $(a::\text{enat}) \text{ div } 0 = 0$
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*: $0 \text{ div } (a::\text{enat}) = 0$
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*: $(a::\text{enat}) \text{ mod } 0 = a$
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*: $0 \text{ mod } (a::\text{enat}) = 0$
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*]: $\text{enat } a \text{ mod enat } b = \text{enat } (a \text{ mod } b)$
unfolding *mod-enat-def* **by** *simp*
lemma *imod-infinity*[*simp*, *code*]: $\infty \text{ mod } n = (\infty :: \text{enat})$
unfolding *mod-enat-def* **by** *simp*
lemma *imod-infinity-right*[*simp*, *code*]: $n \text{ mod } (\infty :: \text{enat}) = n$
unfolding *mod-enat-def* **by** (*case-tac* *n*) *simp-all*

lemma *idiv-self*: $\llbracket 0 < (n :: \text{enat}); n \neq \infty \rrbracket \implies n \text{ div } n = 1$
by (*case-tac* *n*, *simp-all* *add: one-enat-def*)
lemma *imod-self*: $n \neq \infty \implies (n :: \text{enat}) \text{ mod } n = 0$
by (*case-tac* *n*, *simp-all*)

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

lemma *imod-iless-divisor*: $\llbracket 0 < (n :: \text{enat}); m \neq \infty \rrbracket \implies m \text{ mod } n < n$
by (*case-tac* *m*, *simp-all*) (*case-tac* *n*, *simp-all*)
lemma *imod-ile-dividend*: $(m :: \text{enat}) \text{ mod } n \leq m$
by (*case-tac* *m*, *simp-all*) (*case-tac* *n*, *simp-all*)
lemma *idiv-ile-dividend*: $(m :: \text{enat}) \text{ div } n \leq m$
by (*case-tac* *m*, *simp-all*) (*case-tac* *n*, *simp-all*)

lemma *idiv-imult2-eq*: $(a :: \text{enat}) \text{ div } (b * c) = a \text{ div } b \text{ div } c$
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*: $\llbracket (i :: \text{enat}) \leq j; k \leq l \rrbracket \implies i * k \leq j * l$
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*: $(i :: \text{enat}) \leq j \implies i * k \leq j * k$
by (*rule* *imult-ile-mono[OF order-refl]*)

lemma *imult-ile-mono2*: $(i :: \text{enat}) \leq j \implies k * i \leq k * j$
by (*rule* *imult-ile-mono[OF order-refl]*)

lemma *imult-iless-mono1*: $\llbracket (i :: \text{enat}) < j; 0 < k; k \neq \infty \rrbracket \implies i * k \leq j * k$
by (*case-tac* *i*, *case-tac* *j*, *case-tac* *k*, *simp-all*)
lemma *imult-iless-mono2*: $\llbracket (i :: \text{enat}) < j; 0 < k; k \neq \infty \rrbracket \implies k * i \leq k * j$
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*: $(\text{Suc } 0 < n) = (n \neq 0 \wedge n \neq \text{Suc } 0)$
by *fastforce*

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

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

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

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

lemma *mult-pred-right*: $m * (n - \text{Suc } 0) = m * n - m$
by (*simp add: diff-mult-distrib2*)

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

corollary *mult-cancel1-gr0*:
 $(0::\text{nat}) < k \implies (k * m = k * n) = (m = n)$ **by** *simp*

corollary *mult-cancel2-gr0*:
 $(0::\text{nat}) < k \implies (m * k = n * k) = (m = n)$ **by** *simp*

corollary *mult-le-cancel1-gr0*:
 $(0::\text{nat}) < k \implies (k * m \leq k * n) = (m \leq n)$ **by** *simp*

corollary *mult-le-cancel2-gr0*:
 $(0::\text{nat}) < k \implies (m * k \leq n * k) = (m \leq n)$ **by** *simp*

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

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

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

lemma *less-imp-Suc-mult-pred-less*: $\llbracket m < n; 0 < k \rrbracket \implies \text{Suc } m * k - \text{Suc } 0 < n * k$

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*: $(0 < (b::'a::ordered-ab-group-add) - a) = (a < b)$
by (*simp add: less-diff-eq*)

lemma *ord-zero-le-diff*: $(0 \leq (b::'a::ordered-ab-group-add) - a) = (a \leq b)$
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*: $(0::nat) < j \implies i < i + j$ **by** *simp*

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

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

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

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

lemma *diff-less-self*: $\llbracket (0::nat) < i; 0 < j \rrbracket \implies i - j < i$ **by** *simp*

lemma

ge-less- neg-conv : $((a::'a::linorder) \leq n) = (\forall x. x < a \longrightarrow n \neq x)$ **and**

le-greater- neg-conv : $(n \leq (a::'a::linorder)) = (\forall x. a < x \longrightarrow n \neq x)$

by (*subst linorder-not-less*[*symmetric*], *blast*)+

lemma

greater-le- neg-conv : $((a::'a::linorder) < n) = (\forall x. x \leq a \longrightarrow n \neq x)$ **and**

less-ge- neg-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*: $\llbracket 0 \leq (a::'a::ordered-ab-group-add-abs); a \leq b \rrbracket \implies |a| \leq |b|$

by *simp*

lemma *leq-neg-imp-abs-geq*: $\llbracket (a::'a::ordered-ab-group-add-abs) \leq 0; b \leq a \rrbracket \implies |a| \leq |b|$

by *simp*

lemma *abs-range*: $\llbracket 0 \leq (a::'a::\{ordered-ab-group-add-abs,abs-if\}); -a \leq x; x \leq a \rrbracket \implies |x| \leq a$

apply (*clarsimp simp: abs-if*)

apply (*rule neg-le-iff-le*[*THEN iffD1*], *simp*)

done

Lemmas for @termsgn function

lemma *sgn-abs*: $(x::'a::linordered-idom) \neq 0 \implies |\text{sgn } x| = 1$

by (*case-tac x < 0, simp+*)

lemma *sgn-mult-abs*: $|x| * |\text{sgn } (a::'a::linordered-idom)| = |x * \text{sgn } a|$

by (*fastforce simp add: sgn-if abs-if*)

lemma *abs-imp-sgn-abs*: $|a| = |b| \implies |\text{sgn } (a::'a::\text{linordered-idom})| = |\text{sgn } b|$
by (*fastforce simp add: abs-if*)

lemma *sgn-mono*: $a \leq b \implies \text{sgn } (a::'a::\{\text{linordered-idom}, \text{linordered-semidom}\}) \leq \text{sgn } b$

by (*auto simp add: sgn-if*)

4.2 Additional facts about inequalities

lemma *add-diff-le*: $k \leq n \implies m + k - n \leq (m::\text{nat})$

by (*case-tac m + k < n, simp-all*)

lemma *less-add-diff*: $k < (n::\text{nat}) \implies m < n + m - k$

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

lemma *add-diff-less*: $\llbracket k < n; 0 < m \rrbracket \implies m + k - n < (m::\text{nat})$

by (*case-tac m + k < n, simp-all*)

lemma *add-le-imp-le-diff1*: $i + k \leq j \implies i \leq j - (k::\text{nat})$

by (*case-tac k ≤ j, simp-all*)

lemma *add-le-imp-le-diff2*: $k + i \leq j \implies i \leq j - (k::\text{nat})$ **by** *simp*

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

lemma *diff-less-conv*: $0 < i \implies (j - (k::\text{nat}) < i) = (j < i + k)$

by (*safe, simp-all*)

lemma *le-diff-swap*: $\llbracket i \leq (k::\text{nat}); j \leq k \rrbracket \implies (k - j \leq i) = (k - i \leq j)$

by (*safe, simp-all*)

lemma *diff-less-imp-swap*: $\llbracket 0 < (i::\text{nat}); k - i < j \rrbracket \implies (k - j < i)$ **by** *simp*

lemma *diff-less-swap*: $\llbracket 0 < (i::\text{nat}); 0 < j \rrbracket \implies (k - j < i) = (k - i < j)$

by (*blast intro: diff-less-imp-swap*)

lemma *less-diff-imp-less*: $(i::\text{nat}) < j - m \implies i < j$ **by** *simp*

lemma *le-diff-imp-le*: $(i::\text{nat}) \leq j - m \implies i \leq j$ **by** *simp*

lemma *less-diff-le-imp-less*: $\llbracket (i::\text{nat}) < j - m; n \leq m \rrbracket \implies i < j - n$ **by** *simp*

lemma *le-diff-le-imp-le*: $\llbracket (i::\text{nat}) \leq j - m; n \leq m \rrbracket \implies i \leq j - n$ **by** *simp*

lemma *le-imp-diff-le*: $(j::\text{nat}) \leq k \implies j - n \leq k$ **by** *simp*

4.3 Inequalities for Suc and pred

corollary *less-eq-le-pred*: $0 < (n::\text{nat}) \implies (m < n) = (m \leq n - \text{Suc } 0)$

by (*safe, simp-all*)

corollary *less-imp-le-pred*: $m < n \implies m \leq n - \text{Suc } 0$ **by** *simp*

corollary *le-pred-imp-less*: $\llbracket 0 < n; m \leq n - \text{Suc } 0 \rrbracket \implies m < n$ **by** *simp*

corollary *pred-less-eq-le*: $0 < m \implies (m - \text{Suc } 0 < n) = (m \leq n)$
by (*safe, simp-all*)

corollary *pred-less-imp-le*: $m - \text{Suc } 0 < n \implies m \leq n$ **by** *simp*

corollary *le-imp-pred-less*: $\llbracket 0 < m; m \leq n \rrbracket \implies m - \text{Suc } 0 < n$ **by** *simp*

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

lemma *pred-mono*: $\llbracket m < n; 0 < m \rrbracket \implies m - \text{Suc } 0 < n - \text{Suc } 0$ **by** *simp*

corollary *pred-Suc-mono*: $\llbracket m < \text{Suc } n; 0 < m \rrbracket \implies m - \text{Suc } 0 < n$ **by** *simp*

lemma *Suc-less-pred-conv*: $(\text{Suc } m < n) = (m < n - \text{Suc } 0)$ **by** (*safe, simp-all*)

lemma *Suc-le-pred-conv*: $0 < n \implies (\text{Suc } m \leq n) = (m \leq n - \text{Suc } 0)$ **by** (*safe, simp-all*)

lemma *Suc-le-imp-le-pred*: $\text{Suc } m \leq n \implies m \leq n - \text{Suc } 0$ **by** *simp*

4.4 Additional facts about cancellation in (in-)equalities

lemma *diff-cancel-imp-eq*: $\llbracket 0 < (n::\text{nat}); n + i - j = n \rrbracket \implies i = j$ **by** *simp*

lemma *nat-diff-left-cancel-less*: $k - m < k - (n::\text{nat}) \implies n < m$ **by** *simp*

lemma *nat-diff-right-cancel-less*: $n - k < (m::\text{nat}) - k \implies n < m$ **by** *simp*

lemma *nat-diff-left-cancel-le1*: $\llbracket k - m \leq k - (n::\text{nat}); m < k \rrbracket \implies n \leq m$ **by** *simp*

lemma *nat-diff-left-cancel-le2*: $\llbracket k - m \leq k - (n::\text{nat}); n \leq k \rrbracket \implies n \leq m$ **by** *simp*

lemma *nat-diff-right-cancel-le1*: $\llbracket m - k \leq n - (k::\text{nat}); k < m \rrbracket \implies m \leq n$ **by** *simp*

lemma *nat-diff-right-cancel-le2*: $\llbracket m - k \leq n - (k::\text{nat}); k \leq n \rrbracket \implies m \leq n$ **by** *simp*

lemma *nat-diff-left-cancel-eq1*: $\llbracket k - m = k - (n::\text{nat}); m < k \rrbracket \implies m = n$ **by** *simp*

lemma *nat-diff-left-cancel-eq2*: $\llbracket k - m = k - (n::\text{nat}); n < k \rrbracket \implies m = n$ **by** *simp*

lemma *nat-diff-right-cancel-eq1*: $\llbracket m - k = n - (k::\text{nat}); k < m \rrbracket \implies m = n$ **by** *simp*

lemma *nat-diff-right-cancel-eq2*: $\llbracket m - k = n - (k::\text{nat}); k < n \rrbracket \implies m = n$ **by** *simp*

lemma *eq-diff-left-iff*: $\llbracket (m::\text{nat}) \leq k; n \leq k \rrbracket \implies (k - m = k - n) = (m = n)$
by (*safe, simp-all*)

lemma *eq-imp-diff-eq*: $m = (n::\text{nat}) \implies m - k = n - k$ **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  $\implies$  Suc (n mod m)  $\leq$  m
by (rule Suc-leI, rule mod-less-divisor)
```

```
lemma mod-less-dividend:  $\llbracket 0 < m; m \leq n \rrbracket \implies n \text{ mod } m < (n::nat)$ 
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)  $\implies$  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  $\implies$  Suc 0 mod m = Suc 0
by (case-tac m, simp-all)
```

```
corollary Suc0-mod-subst:
 $\llbracket m \neq \text{Suc } 0; P (\text{Suc } 0) \rrbracket \implies P (\text{Suc } 0 \text{ mod } m)$ 
by (blast intro: subst[OF Suc0-mod[symmetric]])
```

```
corollary Suc0-mod-cong:
m  $\neq$  Suc 0  $\implies$  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*:

$((\text{Suc } a) \text{ mod } m = (\text{Suc } b) \text{ mod } m) = (a \text{ mod } m = b \text{ mod } m)$

by (*simp add: mod-Suc*)

lemma *mod-Suc'*:

$0 < n \implies \text{Suc } m \text{ mod } n = (\text{if } m \text{ mod } n < n - \text{Suc } 0 \text{ then } \text{Suc } (m \text{ mod } n) \text{ else } 0)$

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*:

$((a + k) \text{ mod } m = (b + k) \text{ mod } m) =$

$((a::\text{nat}) \text{ mod } m = b \text{ mod } m)$

by (*induct k, simp-all add: mod-Suc-conv*)

corollary *mod-sub-add*:

$k \leq (a::\text{nat}) \implies$

$((a - k) \text{ mod } m = b \text{ mod } m) = (a \text{ mod } m = (b + k) \text{ mod } m)$

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*:

$a + b \leq (n::\text{nat}) \implies$

$((n - a) \text{ mod } m = b \text{ mod } m) = ((n - (a + b)) \text{ mod } m = 0)$

by (*insert mod-add[of n-(a+b) b m 0], simp*)

lemma *mod-sub-eq-mod-swap*:

$\llbracket a \leq (n::\text{nat}); b \leq n \rrbracket \implies$

$((n - a) \text{ mod } m = b \text{ mod } m) = ((n - b) \text{ mod } m = a \text{ mod } m)$

by (*simp add: mod-sub-add add.commute*)

lemma *le-mod-greater-imp-div-less*:

$\llbracket a \leq (b::\text{nat}); a \text{ mod } m > b \text{ mod } m \rrbracket \implies a \text{ div } m < b \text{ div } m$

apply (*rule ccontr, simp add: linorder-not-less*)

apply (*drule mult-le-mono1[of b div m - m]*)

apply (*drule add-less-le-mono1[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*: $\llbracket a < (b::\text{nat}); a \text{ mod } m \geq b \text{ mod } m \rrbracket \implies a \text{ div } m < b \text{ div } m$

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*: $\llbracket a < (b::nat); b \bmod m = 0 \rrbracket \implies a \operatorname{div} m < b \operatorname{div} m$
by (*simp add: less-mod-ge-imp-div-less*)

lemma *mod-diff-right-eq*:
 $(a::nat) \leq b \implies (b - a) \bmod m = (b - a \bmod m) \bmod m$
proof –
assume *a-as*: $a \leq b$
have $(b - a) \bmod m = (b - a + a \operatorname{div} m * m) \bmod m$ **by** *simp*
also have $\dots = (b + a \operatorname{div} m * m - a) \bmod m$ **using** *a-as* **by** *simp*
also have $\dots = (b + a \operatorname{div} m * m - (a \operatorname{div} m * m + a \bmod m)) \bmod m$ **by** *simp*
also have $\dots = (b + a \operatorname{div} m * m - a \operatorname{div} m * m - a \bmod m) \bmod m$
by (*simp only: diff-diff-left[symmetric]*)
also have $\dots = (b - a \bmod m) \bmod m$ **by** *simp*
finally show *?thesis* .

qed
corollary *mod-eq-imp-diff-mod-eq*:
 $\llbracket x \bmod m = y \bmod m; x \leq (t::nat); y \leq t \rrbracket \implies (t - x) \bmod m = (t - y) \bmod m$
by (*simp only: mod-diff-right-eq*)

lemma *mod-eq-imp-diff-mod-eq2*:
 $\llbracket x \bmod m = y \bmod m; (t::nat) \leq x; t \leq y \rrbracket \implies (x - t) \bmod m = (y - t) \bmod m$
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*:
 $(m + b \bmod m - a \bmod m) \bmod (m::nat) = ($
if $a \bmod m \leq b \bmod m$
then $(b \bmod m - a \bmod m)$
else $(m + b \bmod m - a \bmod m)$
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*:
 $a \bmod m \leq b \bmod m \implies (m + b \bmod m - a \bmod m) \bmod (m::nat) = b \bmod m - a \bmod m$
by (*simp add: divisor-add-diff-mod-if*)

corollary *divisor-add-diff-mod-eq2*:

$b \bmod m < a \bmod m \implies$
 $(m + b \bmod m - a \bmod m) \bmod (m::\text{nat}) = m + b \bmod m - a \bmod m$
by (*simp add: divisor-add-diff-mod-if*)

lemma *mod-add-mod-if*:
 $(a \bmod m + b \bmod m) \bmod (m::\text{nat}) =$
if $a \bmod m + b \bmod m < m$
then $a \bmod m + b \bmod m$
else $a \bmod m + b \bmod m - m$
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*:
 $a \bmod m + b \bmod m < m \implies$
 $(a \bmod m + b \bmod m) \bmod (m::\text{nat}) = a \bmod m + b \bmod m$
by (*simp add: mod-add-mod-if*)
corollary *mod-add-mod-eq2*:
 $m \leq a \bmod m + b \bmod m \implies$
 $(a \bmod m + b \bmod m) \bmod (m::\text{nat}) = a \bmod m + b \bmod m - m$
by (*simp add: mod-add-mod-if*)

lemma *mod-add1-eq-if*:
 $(a + b) \bmod (m::\text{nat}) =$
if $(a \bmod m + b \bmod m < m)$ *then* $a \bmod m + b \bmod m$
else $a \bmod m + b \bmod m - m$
by (*simp add: mod-add-eq[symmetric, of a b] mod-add-mod-if*)

lemma *mod-add-eq-mod-conv*: $0 < (m::\text{nat}) \implies$
 $((x + a) \bmod m = b \bmod m) =$
 $(x \bmod m = (m + b \bmod m - a \bmod m) \bmod m)$
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*:
 $(a::\text{nat}) \leq b \implies (b - a) \bmod m = (m + b \bmod m - a \bmod m) \bmod m$
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 $\dots = (b - a \bmod m + m) \bmod m$
by *simp*
also have $\dots = (b + m - a \bmod m) \bmod m$
using $a-mod-le-b-s$ **by** *simp*
also have $\dots = (b \operatorname{div} m * m + b \bmod m + m - a \bmod m) \bmod m$
by *simp*
also have $\dots = (b \operatorname{div} m * m + (b \bmod m + m - a \bmod m)) \bmod m$
by (*simp add: diff-add-assoc[OF mod-le-divisor, OF m-as]*)
also have $\dots = ((b \bmod m + m - a \bmod m) + b \operatorname{div} m * m) \bmod m$
by *simp*
also have $\dots = (b \bmod m + m - a \bmod m) \bmod m$
by *simp*
also have $\dots = (m + b \bmod m - a \bmod m) \bmod m$
by (*simp only: add commute*)
finally show *?thesis* .
qed
corollary *mod-diff1-eq-if*:
 $(a::nat) \leq b \implies (b - a) \bmod m = ($
 $\quad \text{if } a \bmod m \leq b \bmod m \text{ then } b \bmod m - a \bmod m$
 $\quad \text{else } m + b \bmod m - a \bmod m)$
by (*simp only: mod-diff1-eq divisor-add-diff-mod-if*)
corollary *mod-diff1-eq1*:
 $\llbracket (a::nat) \leq b; a \bmod m \leq b \bmod m \rrbracket$
 $\implies (b - a) \bmod m = b \bmod m - a \bmod m$
by (*simp add: mod-diff1-eq-if*)
corollary *mod-diff1-eq2*:
 $\llbracket (a::nat) \leq b; b \bmod m < a \bmod m \rrbracket$
 $\implies (b - a) \bmod m = m + b \bmod m - a \bmod m$
by (*simp add: mod-diff1-eq-if*)

5.2.1 Divisor subtraction with *div* and *mod*

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

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

lemma *div-diff-self1*: $0 < (n::nat) \implies (m - n) \text{ div } m = 0$

by (case-tac m = 0, simp-all)

lemma *div-diff-self2*: $(n - m) \text{ div } m = n \text{ div } m - \text{Suc } 0$

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*:

$(n - k * m) \text{ div } m = n \text{ div } m - (k::nat)$

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*:

$(n - m * k) \text{ div } m = n \text{ div } m - (k::nat)$

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*:

$(a::nat) \text{ mod } m = b \text{ mod } m \implies (b - a) \text{ mod } m = 0$

(is ?P \implies ?Q)

proof –

assume *as1*: ?P

have $b - a = b \text{ div } m * m + b \text{ mod } m - (a \text{ div } m * m + a \text{ mod } m)$

 by simp

also have $\dots = b \text{ div } m * m + b \text{ mod } m - (a \text{ mod } m + a \text{ div } m * m)$

 by simp

also have $\dots = b \text{ div } m * m + b \text{ mod } m - a \text{ mod } m - a \text{ div } m * m$

 by simp

also have $\dots = b \text{ div } m * m + b \text{ mod } m - b \text{ mod } m - a \text{ div } m * m$

 using *as1* by simp

also have $\dots = b \text{ div } m * m - a \text{ div } m * m$

 by (simp only: diff-add-inverse2)

also have $\dots = (b \text{ div } m - a \text{ div } m) * m$

 by (simp only: diff-mult-distrib)

finally have $b - a = (b \text{ div } m - a \text{ div } m) * m$.
hence $(b - a) \text{ mod } m = (b \text{ div } m - a \text{ div } m) * m \text{ mod } m$
by (rule arg-cong)
thus ?thesis **by** (simp only: mod-mult-self2-is-0)
qed
corollary mod-eq-imp-diff-dvd:
 $(a::nat) \text{ mod } m = b \text{ mod } m \implies m \text{ dvd } b - a$
by (rule dvd-eq-mod-eq-0[THEN iffD2, OF mod-eq-imp-diff-mod-0])

lemma mod-neq-imp-diff-mod-neq0:
 $\llbracket (a::nat) \text{ mod } m \neq b \text{ mod } m; a \leq b \rrbracket \implies 0 < (b - a) \text{ mod } m$
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:
 $\llbracket (a::nat) \text{ mod } m \neq b \text{ mod } m; a \leq b \rrbracket \implies \neg m \text{ dvd } b - a$
by (simp add: dvd-eq-mod-eq-0 mod-neq-imp-diff-mod-neq0)

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

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

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

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

lemma *zdiff-mod-0-imp-mod-eq--pos*:

$\llbracket (b - a) \bmod m = 0; 0 < (m::int) \rrbracket \implies a \bmod m = b \bmod m$
 (is $\llbracket ?P; ?Pm \rrbracket \implies ?Q$)

proof -

assume *as1*: $?P$
 and *as2*: $0 < m$

obtain *r1* where $a-r1:r1 = a \bmod m$ by *blast*
 obtain *r2* where $b-r2:r2 = b \bmod m$ by *blast*

obtain *q1* where $a-q1: q1 = a \operatorname{div} m$ by *blast*
 obtain *q2* where $b-q2: q2 = b \operatorname{div} m$ 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 $\dots = m * q2 + r2 - m * q1 - r1$
 by *simp*
 also have $\dots = m * q2 - m * q1 + r2 - r1$
 by *simp*
 finally have $b - a = m * (q2 - q1) + (r2 - r1)$
 by (*simp add: right-diff-distrib*)
 hence $(b - a) \bmod m = (r2 - r1) \bmod m$
 by (*simp add: mod-add-eq*)
 hence $r2-r1\text{-mod-}m\text{-}0:(r2 - r1) \bmod m = 0$ (is *?R1*)
 by (*simp only: as1*)

have $r1 = r2$

proof (*rule notI[of r1 ≠ r2, simplified]*)

assume *as1'*: $r1 \neq r2$

have *diff-le-s*: $\bigwedge a b (m::int). \llbracket 0 \leq a; b < m \rrbracket \implies b - a < m$
 by *simp*

from *as2 a-r1 b-r2* have *s-r1*: $0 \leq r1 \wedge r1 < m$ and *s-r2*: $0 \leq r2 \wedge r2 < m$
 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 $0 \leq r2 - r1 \implies (r2 - r1) \bmod m = (r2 - r1)$

using *r2r1m* by (*blast intro: mod-pos-pos-trivial*)

hence *s1-pos*: $0 \leq r2 - r1 \implies r2 - r1 = 0$

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 $\exists (a::nat) b m. (b - a) \text{ mod } m = 0 \wedge a \text{ mod } m \neq b \text{ mod } m$
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*:
 $\llbracket (0::int) \leq x; a \leq b; 0 < d \rrbracket \implies x * a \text{ div } d \leq x * b \text{ div } d$
by (*metis mult-right-mono zdiv-mono1 mult.commute*)

lemma *zmult-div-leq-mono-neg*:
 $\llbracket x \leq (0::int); a \leq b; 0 < d \rrbracket \implies x * b \text{ div } d \leq x * a \text{ div } d$
by (*metis mult-left-mono-neg zdiv-mono1*)

lemma *zmult-div-pos-le*:
 $\llbracket (0::int) \leq a; 0 \leq b; b \leq c \rrbracket \implies a * b \text{ div } c \leq a$
apply (*case-tac* $b = 0$, *simp*)
apply (*subgoal-tac* $b * a \leq c * a$)
prefer 2
apply (*simp only: mult-right-mono*)
apply (*simp only: mult.commute*)
apply (*subgoal-tac* $a * b \text{ div } c \leq a * c \text{ div } c$)
prefer 2
apply (*simp only: zdiv-mono1*)
apply *simp*
done

lemma *zmult-div-neg-le*:
 $\llbracket a \leq (0::int); 0 < c; c \leq b \rrbracket \implies a * b \text{ div } c \leq a$
apply (*subgoal-tac* $b * a \leq c * a$)
prefer 2
apply (*simp only: mult-right-mono-neg*)
apply (*simp only: mult.commute*)
apply (*subgoal-tac* $a * b \text{ div } c \leq a * c \text{ div } c$)
prefer 2
apply (*simp only: zdiv-mono1*)
apply *simp*
done

lemma *zmult-div-ge-0*: $\llbracket (0::int) \leq x; 0 \leq a; 0 < c \rrbracket \implies 0 \leq a * x \text{ div } c$
by (*metis pos-imp-zdiv-nonneg-iff split-mult-pos-le*)

corollary *zmult-div-plus-ge-0*:
 $\llbracket (0::int) \leq x; 0 \leq a; 0 \leq b; 0 < c \rrbracket \implies 0 \leq a * x \text{ div } c + b$
by (*insert zmult-div-ge-0[*of* $x a c$], *simp**)

lemma *zmult-div-abs-ge*:

$\llbracket (0::int) \leq b; b \leq b'; 0 \leq a; 0 < c \rrbracket \implies$
 $|a * b \text{ div } c| \leq |a * b' \text{ div } c|$

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*:

$\llbracket (0::int) \leq b; b \leq b'; 0 \leq a; 0 < c \rrbracket \implies$
 $|a * b \text{ div } c + a| \leq |a * b' \text{ div } c + a|$

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*:

$\llbracket (x::nat) < y; x \text{ mod } m = y \text{ mod } m \rrbracket \implies x + m \leq y$

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 $x \text{ mod } m = y \text{ mod } m$

hence *y-x-mod-m*: $(y - x) \text{ mod } m = 0$

by (*simp only: mod-eq-imp-diff-mod-0*)

assume $\neg x + m \leq y$

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 $(y - x) \text{ mod } m = y - x$

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*:

$(x::nat) < n \text{ div } m \implies x * m + m \leq n$

apply (*case-tac m = 0, simp*)

apply (*case-tac n < m, simp*)

```

apply (simp add: linorder-not-less)
apply (subgoal-tac m ≤ n - n mod m)
prefer 2
apply (drule div-le-mono[of m - m])
apply (simp only: div-self)
apply (drule mult-le-mono2[of 1 - m])
apply (simp only: mult-1-right 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:
   $((n + k) \bmod m = n \bmod m) = (k \bmod m = 0)$ 
by (metis add-eq-if mod-add mod-add-self1 mod-self add.commute)

```

```

lemma between-imp-mod-between:
   $\llbracket b < (m::nat); m * k + a \leq n; n \leq m * k + b \rrbracket \implies$ 
   $a \leq n \bmod m \wedge n \bmod m \leq b$ 
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:
   $\llbracket b < (m::nat); m * k \leq n; n \leq m * k + b \rrbracket \implies n \bmod m \leq b$ 
by (insert between-imp-mod-between[of b m k 0 n], simp)
corollary between-imp-mod-gr0:
   $\llbracket (m::nat) * k < n; n < m * k + m \rrbracket \implies 0 < n \bmod m$ 
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:
   $0 < m \implies (k * m \leq n \wedge n < \text{Suc } k * m) = (n \bmod m = k)$ 
by (auto simp add: ac-simps intro: div-nat-eqI dividend-less-times-div)

```

lemma *le-less-imp-div*:

$\llbracket k * m \leq n; n < \text{Suc } k * m \rrbracket \implies n \text{ div } m = k$

by (*auto simp add: ac-simps intro: div-nat-eqI*)

lemma *div-imp-le-less*:

$\llbracket n \text{ div } m = k; 0 < m \rrbracket \implies k * m \leq n \wedge n < \text{Suc } k * m$

by (*auto simp add: ac-simps intro: dividend-less-times-div*)

lemma *div-le-mod-le-imp-le*:

$\llbracket (a::\text{nat}) \text{ div } m \leq b \text{ div } m; a \bmod m \leq b \bmod m \rrbracket \implies a \leq b$

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*:

$\llbracket a \leq b; (a + k) \bmod m = (b::\text{nat}) \bmod m \rrbracket \implies a + k \bmod m \leq b$

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*:

$\llbracket (m::\text{nat}) * k \leq n; r \leq n \bmod m \rrbracket \implies m * k + r \leq n$

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*:

$n \bmod m = (0::\text{nat}) \implies n * k \bmod m = 0$

by *fastforce*

lemma *mod-0-imp-mod-mult-left-0*:

$n \bmod m = (0::\text{nat}) \implies k * n \bmod m = 0$

by *fastforce*

lemma *mod-0-imp-div-mult-left-eq*:

$n \bmod m = (0::\text{nat}) \implies k * n \text{ div } m = k * (n \text{ div } m)$

by *fastforce*

lemma *mod-0-imp-div-mult-right-eq*:

$n \bmod m = (0::\text{nat}) \implies n * k \text{ div } m = k * (n \text{ div } m)$

by *fastforce*

lemma *mod-0-imp-mod-factor-0-left*:

$n \bmod (m * m') = (0::\text{nat}) \implies n \bmod m = 0$

by *fastforce*

lemma *mod-0-imp-mod-factor-0-right*:

$n \text{ mod } (m * m') = (0::\text{nat}) \implies n \text{ mod } m' = 0$
by *fastforce*

5.6 Some factor distribution facts for *mod*

lemma *mod-eq-mult-distrib*:

$(a::\text{nat}) \text{ mod } m = b \text{ mod } m \implies$
 $a * k \text{ mod } (m * k) = b * k \text{ mod } (m * k)$
by *simp*

lemma *mod-mult-eq-imp-mod-eq*:

$(a::\text{nat}) \text{ mod } (m * k) = b \text{ mod } (m * k) \implies a \text{ mod } m = b \text{ mod } m$
apply (*simp only: mod-mult2-eq*)
apply (*drule-tac arg-cong[where f= $\lambda x. x \text{ mod } m$]*)
apply (*simp add: add commute*)
done

corollary *mod-eq-mod-0-imp-mod-eq*:

$\llbracket (a::\text{nat}) \text{ mod } m' = b \text{ mod } m'; m' \text{ mod } m = 0 \rrbracket$
 $\implies a \text{ mod } m = b \text{ mod } m$
using *mod-mod-cancel [of m m' a] mod-mod-cancel [of m m' b]* **by** *auto*

lemma *mod-factor-imp-mod-0*:

$\llbracket (x::\text{nat}) \text{ mod } (m * k) = y * k \text{ mod } (m * k) \rrbracket \implies x \text{ mod } k = 0$
(is $\llbracket ?P1 \rrbracket \implies ?Q$ **)**

proof –

assume *as1: ?P1*
have $y * k \text{ mod } (m * k) = y \text{ mod } m * k$
by *simp*
hence $x \text{ mod } (m * k) = y \text{ mod } m * k$
using *as1* **by** *simp*
hence $y \text{ mod } m * k = k * (x \text{ div } k \text{ mod } m) + x \text{ mod } k$ **(is ?l1 = ?r1)**
by (*simp only: ac-simps mod-mult2-eq*)
hence $(y \text{ mod } m * k) \text{ mod } k = ?r1 \text{ mod } k$
by *simp*
hence $0 = ?r1 \text{ mod } k$
by *simp*
thus $x \text{ mod } k = 0$
by (*simp add: mod-add-eq*)

qed

corollary *mod-factor-div*:

$\llbracket (x::\text{nat}) \text{ mod } (m * k) = y * k \text{ mod } (m * k) \rrbracket \implies x \text{ div } k * k = x$
by (*blast intro: mod-factor-imp-mod-0[THEN mod-0-div-mult-cancel[THEN iffD1]]*)

lemma *mod-factor-div-mod*:

$\llbracket (x::\text{nat}) \text{ mod } (m * k) = y * k \text{ mod } (m * k); 0 < k \rrbracket$
 $\implies x \text{ div } k \text{ mod } m = y \text{ mod } m$
(is $\llbracket ?P1; ?P2 \rrbracket \implies ?L = ?R$ **)**

proof –

assume *as1: ?P1*


```

assume as2: ?P2
have x-mod-k-0:  $x \bmod k = 0$ 
  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:  $0 < m \implies$ 
   $(a + b) \operatorname{div} (m::\text{nat}) = a \operatorname{div} m + b \operatorname{div} m +$ 
  (if  $a \bmod m + b \bmod m < m$  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:
   $a \bmod m + b \bmod m < (m::\text{nat}) \implies$ 
   $(a + b) \operatorname{div} (m::\text{nat}) = a \operatorname{div} m + b \operatorname{div} m$ 
apply (case-tac m = 0, simp)
apply (simp add: div-add1-eq-if)
done
corollary div-add1-eq1-mod-0-left:
   $a \bmod m = 0 \implies (a + b) \operatorname{div} (m::\text{nat}) = a \operatorname{div} m + b \operatorname{div} m$ 
apply (case-tac m = 0, simp)
apply (simp add: div-add1-eq1)
done
corollary div-add1-eq1-mod-0-right:
   $b \bmod m = 0 \implies (a + b) \operatorname{div} (m::\text{nat}) = a \operatorname{div} m + b \operatorname{div} m$ 
by (fastforce simp: div-add1-eq1-mod-0-left)
corollary div-add1-eq2:
   $\llbracket 0 < m; (m::\text{nat}) \leq a \bmod m + b \bmod m \rrbracket \implies$ 
   $(a + b) \operatorname{div} (m::\text{nat}) = \operatorname{Suc} (a \operatorname{div} m + b \operatorname{div} m)$ 
by (simp add: div-add1-eq-if)

```

```

lemma div-Suc:
   $0 < n \implies \operatorname{Suc} m \operatorname{div} n = (\text{if } \operatorname{Suc} (m \bmod n) = n \text{ then } \operatorname{Suc} (m \operatorname{div} n) \text{ else } m \operatorname{div} n)$ 

```

```

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  $\implies$  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  $\leq$  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  $\leq$  b
  and m-as: 0 < m
  have div-le:a div m  $\leq$  b div m
  using le-as by (simp only: div-le-mono)
  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  $\leq$  b mod m)
  case True

```

hence as' : $a \bmod m \leq b \bmod m$.

have $(b - a) \operatorname{div} m = ?b-a1 \operatorname{div} m$

using $b-a\text{-div-s}$.

also have $\dots = ((b \operatorname{div} m - a \operatorname{div} m) * m + (b \bmod m - a \bmod m)) \operatorname{div} m$

using as' by *simp*

also have $\dots = b \operatorname{div} m - a \operatorname{div} m + (b \bmod m - a \bmod m) \operatorname{div} m$

apply (*simp only: add.commute*)

by (*simp only: div-mult-self1[OF less-imp-neq[OF m-as, THEN not-sym]]*)

finally have $b-a\text{-div-s}'$: $(b - a) \operatorname{div} m = \dots$.

have $(b \bmod m - a \bmod m) \operatorname{div} m = 0$

by (*rule div-less, rule less-imp-diff-less,*
rule mod-less-divisor, rule m-as)

thus $?thesis$

using $b-a\text{-div-s}' as'$

by *simp*

next

case *False*

hence $as1'$: $\neg a \bmod m \leq b \bmod m$.

hence as' : $b \bmod m < a \bmod m$ by *simp*

have $a\text{-div-less}$: $a \operatorname{div} m < b \operatorname{div} m$

using $le-as as'$

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

have $b \operatorname{div} m - a \operatorname{div} m = b \operatorname{div} m - a \operatorname{div} m - (\operatorname{Suc} 0 - \operatorname{Suc} 0)$

by *simp*

also have $\dots = b \operatorname{div} m - a \operatorname{div} m + \operatorname{Suc} 0 - \operatorname{Suc} 0$

by *simp*

also have $\dots = b \operatorname{div} m - a \operatorname{div} m - \operatorname{Suc} 0 + \operatorname{Suc} 0$

by (*simp only: diff-add-assoc2*)

$a\text{-div-less}[THEN zero-less-diff[THEN iffD2], THEN Suc-le-eq[THEN iffD2]]$)

finally have $b-a\text{-div-s}'$: $b \operatorname{div} m - a \operatorname{div} m = \dots$.

have $(b - a) \operatorname{div} m = ?b-a1 \operatorname{div} m$

using $b-a\text{-div-s}$.

also have $\dots = ((b \operatorname{div} m - a \operatorname{div} m - \operatorname{Suc} 0 + \operatorname{Suc} 0) * m$
 $+ b \bmod m - a \bmod m) \operatorname{div} m$

using $b-a\text{-div-s}'$ by (*rule arg-cong*)

also have $\dots = ((b \operatorname{div} m - a \operatorname{div} m - \operatorname{Suc} 0) * m$

$+ \operatorname{Suc} 0 * m + b \bmod m - a \bmod m) \operatorname{div} m$

by (*simp only: add-mult-distrib*)

also have $\dots = ((b \operatorname{div} m - a \operatorname{div} m - \operatorname{Suc} 0) * m$

$+ m + b \bmod m - a \bmod m) \operatorname{div} m$

by *simp*

also have $\dots = ((b \operatorname{div} m - a \operatorname{div} m - \operatorname{Suc} 0) * m$

$+ (m + b \bmod m - a \bmod m)) \operatorname{div} m$

by (*simp only: add.assoc m-as*)

$\operatorname{diff-add-assoc}[of a \bmod m m + b \bmod m]$

$trans-le-add1$ [of $a \bmod m$, $OF \text{ mod-le-divisor}$])
also have $\dots = b \text{ div } m - a \text{ div } m - \text{Suc } 0$
 $+ (m + b \bmod m - a \bmod m) \text{ div } m$
by (*simp only: add.commute div-mult-self1* [$OF \text{ less-imp-neq}$ [$OF \text{ m-as}$, $THEN \text{ not-sym}$]])
finally have $b-a\text{-div-s'}$: $(b - a) \text{ div } m = \dots$.

have $div-0-s$: $(m + b \bmod m - a \bmod m) \text{ div } m = 0$
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 $div\text{-diff1-eq}$:
 $(b - a) \text{ div } (m::nat) =$
 $b \text{ div } m - a \text{ div } m - (m + a \bmod m - \text{Suc } (b \bmod m)) \text{ div } m$
apply (*case-tac m = 0, simp*)
apply (*simp only: neq0-conv*)
apply (*rule subst*[of
 $\text{if } a \bmod m \leq b \bmod m \text{ then } 0 \text{ else } \text{Suc } 0$
 $(m + a \bmod m - \text{Suc}(b \bmod m)) \text{ div } m$])
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 $b \bmod m$, $THEN \text{ iffD2}$])
apply (*simp only: diff-add-assoc*)
apply (*simp only: div-add-self1*)
apply (*simp add: less-imp-diff-less*)
done

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

$\llbracket (x::nat) \leq y; y - x < m \rrbracket \implies x = y - (y - x) \text{ mod } m$

by *simp*

lemma *diff-ge-divisor-imp-sub-mod-less*:

$\llbracket (x::nat) \leq y; m \leq y - x; 0 < m \rrbracket \implies x < y - (y - x) \text{ mod } m$

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*:

$(x::nat) \leq y \implies x \leq y - (y - x) \text{ mod } m$

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*:

$\llbracket n \text{ mod } m < r; r \leq m; r \leq (n::nat) \rrbracket \implies$
 $(n - r) \text{ mod } m = m + n \text{ mod } m - r$

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*:

$\llbracket 0 < (n::nat); n \text{ mod } m = 0 \rrbracket \implies$
 $(n - \text{Suc } 0) \text{ mod } m = m - \text{Suc } 0$

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*:

$0 < n \implies$

$(n - \text{Suc } 0) \text{ mod } m = ($
 $\text{if } n \text{ mod } m = 0 \text{ then } m - \text{Suc } 0 \text{ else } n \text{ mod } m - \text{Suc } 0)$

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= $\lambda x. x \leq n \text{ mod } m$], simp*)

```

apply (simp only: mod-diff1-eq1)
apply (simp add: Suc0-mod)
done
corollary mod-pred-Suc-mod:
   $0 < n \implies \text{Suc } ((n - \text{Suc } 0) \bmod m) \bmod m = n \bmod m$ 
apply (case-tac m = 0, simp)
apply (simp add: mod-pred)
done
corollary diff-mod-pred:
   $a < b \implies$ 
   $(b - \text{Suc } a) \bmod m =$ 
   $\text{if } a \bmod m = b \bmod m \text{ then } m - \text{Suc } 0 \text{ else } (b - a) \bmod m - \text{Suc } 0$ 
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:
   $a < b \implies \text{Suc } ((b - \text{Suc } a) \bmod m) \bmod m = (b - a) \bmod m$ 
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:
   $\llbracket a < b; 0 < m; a \bmod m = b \bmod m \rrbracket \implies$ 
   $\text{Suc } ((b - \text{Suc } a) \bmod m) = m$ 
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:
   $r \leq t \implies (t - (t - r) \bmod m) \bmod (m::\text{nat}) = r \bmod m$ 
by (metis mod-diff-right-eq diff-diff-cancel diff-le-self)

lemma sub-diff-mod-eq':
   $r \leq t \implies (k * m + t - (t - r) \bmod m) \bmod (m::\text{nat}) = r \bmod m$ 
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  $\implies ((x + k - \text{Suc } 0) \bmod k = 0) = (x \bmod k = \text{Suc } 0)$ 
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*: $Suc\ 0 < k \implies (x\ mod\ k = k - Suc\ 0) = (Suc\ x\ mod\ k = 0)$
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*:
 $\llbracket n\ mod\ m < r; r \leq m + n\ mod\ m \rrbracket \implies (n - r)\ div\ m = n\ div\ m - Suc\ 0$
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*:
 $\llbracket n\ mod\ m = 0; 0 < r; r \leq m \rrbracket \implies (n - r)\ div\ m = n\ div\ m - Suc\ 0$
by (*simp add: mod-less-imp-diff-div-conv*)

corollary *mod-0-less-imp-diff-Suc-div-conv*:
 $\llbracket n\ mod\ m = 0; r < m \rrbracket \implies (n - Suc\ r)\ div\ m = n\ div\ m - Suc\ 0$
by (*drule mod-0-le-imp-diff-div-conv[where r=Suc r], simp-all*)

corollary *mod-0-imp-diff-Suc-div-conv*:
 $(n - r)\ mod\ m = 0 \implies (n - Suc\ r)\ div\ m = (n - r)\ div\ m - Suc\ 0$
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*:
 $n\ mod\ m = 0 \implies (n - Suc\ 0)\ div\ m = n\ div\ m - Suc\ 0$
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*: $0 < m \implies n\ div\ m \leq k = (n \leq Suc\ k * m - Suc\ 0)$
apply (*rule iffI*)
apply (*drule mult-le-mono1[of - - m]*)
apply (*simp only: mult commute[of - m] 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:  $0 < (m::nat) \implies (n \leq k \text{ div } m) = (n * m \leq k)$ 
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:  $n < k * m \implies n \text{ div } m < (k::nat)$ 
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:  $\llbracket 0 < (m::nat); n \text{ div } m < k \rrbracket \implies n < k * m$ 
apply (rule ccontr, simp only: linorder-not-less)
apply (drule div-le-mono[of - - m])
apply simp
done

```

```

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

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

lemma *add-le-divisor-imp-le-Suc-div*:
 $\llbracket x \text{ div } m \leq n; y \leq m \rrbracket \implies (x + y) \text{ div } m \leq \text{Suc } n$
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  $\Rightarrow$  'b::order)  $\Rightarrow$  'a set  $\Rightarrow$  bool
  where mono-on f A  $\equiv \forall a \in A. \forall b \in A. a \leq b \longrightarrow f a \leq f b$ 

```

```

definition strict-mono-on :: ('a::order  $\Rightarrow$  'b::order)  $\Rightarrow$  'a set  $\Rightarrow$  bool
  where strict-mono-on f A  $\equiv \forall a \in A. \forall b \in A. a < b \longrightarrow f a < f b$ 

```

```

lemma mono-on-subset:  $\llbracket \text{mono-on } f A ; B \subseteq A \rrbracket \Longrightarrow \text{mono-on } f B$ 
unfolding mono-on-def by blast

```

```

lemma strict-mono-on-subset:  $\llbracket \text{strict-mono-on } f A ; B \subseteq A \rrbracket \Longrightarrow \text{strict-mono-on } f B$ 
unfolding strict-mono-on-def by blast

```

```

lemma mono-on-UNIV-mono-conv:  $\text{mono-on } f \text{ UNIV} = \text{mono } f$ 
unfolding mono-on-def mono-def by blast

```

```

lemma strict-mono-on-UNIV-strict-mono-conv:
   $\text{strict-mono-on } f \text{ UNIV} = \text{strict-mono } f$ 
unfolding strict-mono-on-def strict-mono-def by blast

```

```

lemma mono-imp-mono-on:  $\text{mono } f \Longrightarrow \text{mono-on } f A$ 
unfolding mono-on-def mono-def by blast

```

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

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

lemma *strict-mono-on-imp-inj-on*:
 $strict\ mono\ on\ f\ (A::'a::linorder\ set) \implies inj\ on\ f\ A$
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*: $strict\ mono\ (f::('a::linorder \Rightarrow 'b::order)) \implies inj\ f$
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*:
 $\llbracket inj\ on\ f\ A; B \subseteq A; a \in A \rrbracket \implies (f\ a \in f\ 'B) = (a \in B)$
unfolding *inj-on-def* **by** *blast*

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

6.1.3 Surjectivity

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

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

unfolding *surj-on-def* ..

lemma *surj-on-image-conv*: (surj-on f A B) = (B ⊆ f ' A)

unfolding *surj-on-conv* **by** *blast*

lemma *surj-on-id*: *surj-on* id A A

unfolding *id-def surj-on-conv* **by** *blast*

lemma

surj-onI: [[∀ b ∈ B. ∃ a∈A. b = f a]] ⇒ *surj-on* f A B **and**

surj-onD2: *surj-on* f A B ⇒ ∀ b ∈ B. ∃ a∈A. b = f a **and**

surj-onD: [*surj-on* f A B; b ∈ B] ⇒ ∃ a∈A. b = f a

unfolding *surj-on-conv*

by *blast+*

lemma *comp-surj-on*:

[*surj-on* f A B; *surj-on* g B C] ⇒ *surj-on* (g ∘ f) A C

unfolding *comp-def surj-on-image-conv* **by** *blast*

lemma *surj-on-Un-right*: *surj-on* f A (B1 ∪ B2) = (*surj-on* f A B1 ∧ *surj-on* f A B2)

unfolding *surj-on-image-conv*

by *blast*

lemma *surj-on-Un-left*:

surj-on f (A1 ∪ A2) B =

(∃ B1. ∃ B2. B ⊆ B1 ∪ B2 ∧ *surj-on* f A1 B1 ∧ *surj-on* f A2 B2)

unfolding *surj-on-image-conv*

apply (*rule iffI*)

apply (*rule-tac* x=f ' A1 **in** *exI*)

apply (*rule-tac* x=f ' A2 **in** *exI*)

apply *blast*

apply *blast*

done

lemma *surj-on-diff-right*: *surj-on* f A B ⇒ *surj-on* f A (B - B')

unfolding *surj-on-conv* **by** *blast*

lemma *surj-on-empty-right*: *surj-on* f A {}

unfolding *surj-on-conv* **by** *blast*

lemma *surj-on-empty-left*: *surj-on* f {} B = (B = {})

unfolding *surj-on-conv* **by** *blast*

lemma *surj-on-imageI*: $\text{surj-on } (g \circ f) A B \implies \text{surj-on } g (f \text{ ' } A) B$
unfolding *surj-on-conv* **by** *fastforce*

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

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

lemma *surj-on-subset-right*: $\llbracket \text{surj-on } f A B; B' \subseteq B \rrbracket \implies \text{surj-on } f A B'$
unfolding *surj-on-conv* **by** *blast*

lemma *surj-on-subset-left*: $\llbracket \text{surj-on } f A B; A \subseteq A' \rrbracket \implies \text{surj-on } f A' B$
unfolding *surj-on-conv* **by** *blast*

lemma *bij-betw-imp-surj-on*: $\text{bij-betw } f A B \implies \text{surj-on } f A B$
unfolding *bij-betw-def surj-on-image-conv* **by** *simp*

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

6.1.4 Induction over natural sets

lemma *image-nat-induct*:
 $\llbracket P (f 0); \bigwedge n. P (f n) \implies P (f (\text{Suc } n)); \text{surj-on } f \text{ UNIV } I; a \in I \rrbracket \implies P a$

proof –

assume *as-P0*: $P (f 0)$
and *as-IA*: $\bigwedge n. P (f n) \implies P (f (\text{Suc } n))$
and *as-surj-f*: $\text{surj-on } f \text{ UNIV } I$
and *as-a*: $a \in I$

have *P-n*: $\bigwedge n. P (f n)$
apply (*induct-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]*:
 $\llbracket P n0; \bigwedge n. \llbracket n0 \leq n; P n \rrbracket \implies P (\text{Suc } n); n0 \leq n \rrbracket \implies P n$
by (*insert nat-induct[where n=n-n0 and P= $\lambda n. P (n0+n)$], simp*)

lemma *enat-induct*:

```

[[ P 0; P ∞; ∧n. P n ⇒ P (eSuc n) ] ⇒ 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*:

```

[[ ∧n. P n ⇒ P (eSuc n); n0' ≤ n'; P (enat n') ] ⇒ P (enat (Suc n'))
by (simp only: enat-defs enat.splits)

```

lemma *eSuc-imp-Suc-aux-n0*:

```

[[ ∧n. [[ enat n0' ≤ n; P n ] ⇒ P (eSuc n); n0' ≤ n'; P (enat n') ] ⇒ P (enat
(Suc n'))

```

proof –

```

assume IA: ∧n. [[ enat n0' ≤ n; P n ] ⇒ P (eSuc n)
and n0-n: n0' ≤ n'
and Pn: P (enat n')
from n0-n
have (enat n0' ≤ 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 ∞; ∧n. [[ n0 ≤ n; P n ] ⇒ P (eSuc n); n0 ≤ n ] ⇒ 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=λn. 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 ∈ (I::nat set) ∧ y ∈ I ∧ x < y }
apply (rule wf-subsetwhere
  p={ (x,y). x ∈ I ∧ y ∈ I ∧ x < y } and
  r={ (x,y). x < y })

```


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

lemma *interval-induct*:

$\llbracket \bigwedge x. \forall y. (x \in (I::\text{nat set}) \wedge y \in I \wedge y < x \longrightarrow P y) \Longrightarrow P x \rrbracket$
 $\Longrightarrow P a$
(is $\llbracket \bigwedge x. \forall y. ?IA x y \Longrightarrow P x \rrbracket \Longrightarrow P a$ **)**
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*:

$\llbracket \bigwedge x. (\bigwedge y. (x \in (I::\text{nat set}) \wedge y \in I \wedge y < x \Longrightarrow P y)) \Longrightarrow P x \rrbracket$
 $\Longrightarrow P a$
by (*blast intro: interval-induct*)

6.1.5 Monotonicity and injectivity of arithmetic operators

lemma *add-left-inj*: *inj* ($\lambda x. n + (x::'a::\text{cancel-ab-semigroup-add})$)
by (*simp add: inj-on-def*)

lemma *add-right-inj*: *inj* ($\lambda x. x + (n::'a::\text{cancel-ab-semigroup-add})$)
by (*simp add: inj-on-def*)

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

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

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

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

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

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

lemma *mult-left-strict-mono*: $0 < n \Longrightarrow \text{strict-mono } (\lambda x. n * (x::\text{nat}))$
by (*unfold strict-mono-def, clarify, simp*)

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

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

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

corollary *mod-eq-imp-div-right-inj-on*:
 $\llbracket 0 < (k::nat); \forall x \in I. \forall y \in I. x \text{ mod } k = y \text{ mod } k \rrbracket \Longrightarrow$
inj-on $(\lambda x. x \text{ div } k) I$
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*: $\llbracket k \in I; \bigwedge x. x \in I \implies P x \rrbracket \implies P (iMin I)$
by (*blast intro: iMinI*)

lemma *iMinI2-ex*: $\llbracket \exists x. x \in I; \bigwedge x. x \in I \implies P x \rrbracket \implies P (iMin I)$
by (*blast intro: iMinI-ex*)

lemma *iMinI2-ex2*: $\llbracket I \neq \{\} ; \bigwedge x. x \in I \implies P x \rrbracket \implies P (iMin I)$
by (*blast intro: iMinI-ex2*)

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

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

lemma *iMin-mono*:
 $\llbracket mono f; \exists x. x \in I \rrbracket \implies iMin (f ' I) = f (iMin I)$
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*:
 $\llbracket mono f; I \neq \{\} \rrbracket \implies iMin (f ' I) = f (iMin I)$
by (*blast intro: iMin-mono*)

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

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

lemma *Collect-iMin-le*: $P k \implies iMin \{x. P x\} \leq k$
by (*rule iMin-le, blast*)

lemma *Collect-minI*: $\llbracket k \in I; P (k::('a::wellorder)) \rrbracket \implies \exists x \in I. P x \wedge (\forall y \in I. y < x \longrightarrow \neg P y)$
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*: $\exists k \in I. P (k::('a::wellorder)) \implies \exists x \in I. P x \wedge (\forall y \in I. y < x \longrightarrow \neg P y)$

by (*erule* *bexE*, *rule* *Collect-minI*)

corollary *Collect-minI-ex2*: $\{k \in I. P (k::('a::wellorder))\} \neq \{\} \implies \exists x \in I. P x \wedge (\forall y \in I. y < x \longrightarrow \neg P y)$

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*:

$\llbracket k \in I; \bigwedge x. x \in I \implies k \leq x \rrbracket \implies iMin I = k$

unfolding *iMin-def*

by (*blast intro*: *Least-equality*)

lemma *iMin-mono-on*:

$\llbracket mono-on f I; \exists x. x \in I \rrbracket \implies iMin (f ' I) = f (iMin I)$

apply (*unfold mono-on-def*)

apply (*rule iMin-equality*)

apply (*blast intro*: *iMinI-ex*)+

done

lemma *iMin-mono-on2*:

$\llbracket mono-on f I; I \neq \{\} \rrbracket \implies iMin (f ' I) = f (iMin I)$

by (*rule iMin-mono-on*, *blast+*)

lemma *iMinI2-order*:

$\llbracket k \in I; \bigwedge y. y \in I \implies k \leq y; \bigwedge x. \llbracket x \in I; \forall y \in I. x \leq y \rrbracket \implies P x \rrbracket \implies P (iMin I)$

by (*simp add*: *iMin-def LeastI2-order*)

lemma *wellorder-iMin-lemma*:

$k \in I \implies iMin I \in I \wedge iMin I \leq k$

by (*blast intro*: *iMinI iMin-le*)

lemma *iMin-Min-conv*:

$\llbracket finite I; I \neq \{\} \rrbracket \implies iMin I = Min I$

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*]: $\llbracket \text{finite } I; k \in I; k \neq \text{Min } I \rrbracket \Longrightarrow \text{Min } I < k$
by (rule *order-neq-le-trans*[*OF not-sym Min-le*])

lemma *Max-neq-imp-less*[*dest*]: $\llbracket \text{finite } I; k \in I; k \neq \text{Max } I \rrbracket \Longrightarrow k < \text{Max } I$
by (rule *order-neq-le-trans*[*OF - Max-ge*])

lemma *nat-Least-mono*:
 $\llbracket A \neq \{\}; \text{mono } (f::(\text{nat} \Rightarrow \text{nat})) \rrbracket \Longrightarrow$
 $(\text{LEAST } x. x \in f \text{' } A) = f (\text{LEAST } x. x \in A)$
unfolding *iMin-def*[*symmetric*]
by (*blast intro: iMin-mono2*)

lemma *Least-disj*:
 $\llbracket \exists x. P x; \exists x. Q x \rrbracket \Longrightarrow$
 $(\text{LEAST } (x::'a::\text{wellorder}). (P x \vee Q x)) = \text{min } (\text{LEAST } x. P x) (\text{LEAST } x. Q x)$
apply (*elim exE*, *rename-tac x1 x2*)
apply (*subgoal-tac* $\bigwedge x1 x2 P Q. \llbracket P x1; Q x2; \text{Least } P \leq \text{Least } Q \rrbracket \Longrightarrow (\text{LEAST } x. P x \vee Q x) = \text{Least } P$)
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*:
 $\llbracket \exists x. P x; \bigwedge x. P x \Longrightarrow Q x \rrbracket \Longrightarrow$
 $(\text{LEAST } (x::'a::\text{wellorder}). Q x) \leq (\text{LEAST } x. P x)$
by (*blast intro: Least-le LeastI2-ex*)

lemma *Least-imp-disj-eq*:
 $\llbracket \exists x. P x; \bigwedge x. P x \Longrightarrow Q x \rrbracket \Longrightarrow$
 $(\text{LEAST } (x::'a::\text{wellorder}). P x \vee Q x) = (\text{LEAST } x. Q x)$
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*:

$\llbracket \exists x. P x; \exists x. Q x; \bigwedge x y. \llbracket P x; Q y \rrbracket \implies x \leq y \rrbracket \implies$
 $(LEAST (x::'a::wellorder). P x) \leq (LEAST (x::'a::wellorder). Q x)$

by (*blast intro: LeastI*)

lemma *Least-le-imp-le-disj*:

$\llbracket \exists x. P x; \bigwedge x y. \llbracket P x; Q y \rrbracket \implies x \leq y \rrbracket \implies$
 $(LEAST (x::'a::wellorder). P x \vee Q x) = (LEAST (x::'a::wellorder). P x)$

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*: $\llbracket (k::'a::linorder) \in A; \text{finite } A; \bigwedge x. x \in A \implies x \leq k \rrbracket \implies$
 $Max A = k$

by (*rule Max-eqI*)

lemma *not-greater-Max*: $\llbracket \text{finite } A; Max A < k \rrbracket \implies k \notin A$

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*: $\llbracket \text{finite } \{x. P x\}; Max \{x. P x\} < k \rrbracket \implies \neg P k$
by (*drule not-greater-Max, assumption, drule Collect-not-in-imp-not*)

lemma *Collect-Max-ge*: $\llbracket \text{finite } \{x. P x\}; P k \rrbracket \implies k \leq Max \{x. P x\}$

by (*rule Max-ge, assumption, rule CollectI*)

lemma *MaxI*: $\llbracket k \in A; \text{finite } A \rrbracket \implies Max A \in A$

by (*case-tac A = {}, simp-all*)

lemma *MaxI-ex*: $\llbracket \exists x. x \in A; \text{finite } A \rrbracket \implies Max A \in A$

by (*blast intro: MaxI*)

lemma *MaxI-ex2*: $\llbracket A \neq \{\}; \text{finite } A \rrbracket \implies Max A \in A$

by (*blast intro: MaxI*)

lemma *MaxI2*: $\llbracket k \in A; \bigwedge x. x \in A \implies P x; \text{finite } A \rrbracket \implies P (Max A)$

by (*drule Max-in, blast+*)

lemma *MaxI2-ex*: $\llbracket \exists x. x \in A; \bigwedge x. x \in A \implies P x; \text{finite } A \rrbracket \implies P (Max A)$

by (*blast intro: MaxI2*)

lemma *MaxI2-ex2*: $\llbracket A \neq \{\}; \bigwedge x. x \in A \implies P x; \text{finite } A \rrbracket \implies P (Max A)$

by (*blast intro: MaxI2*)

lemma *Max-mono*: $\llbracket \text{mono } f; \exists x. x \in A; \text{finite } A \rrbracket \implies \text{Max } (f \text{ ' } A) = f (\text{Max } A)$
apply (*unfold mono-def*)
apply *clarify*
apply (*frule Max-in, blast*)
apply (*rule Max-equality, clarsimp+*)
done

lemma *Max-mono2*: $\llbracket \text{mono } f; A \neq \{\}; \text{finite } A \rrbracket \implies \text{Max } (f \text{ ' } A) = f (\text{Max } A)$
by (*blast intro: Max-mono*)

lemma *Max-mono-on*: $\llbracket \text{mono-on } f \text{ } A; \exists x. x \in A; \text{finite } A \rrbracket \implies \text{Max } (f \text{ ' } A) = f (\text{Max } A)$
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*:
 $\llbracket \text{mono-on } f \text{ } A; A \neq \{\}; \text{finite } A \rrbracket \implies \text{Max } (f \text{ ' } A) = f (\text{Max } A)$
by (*rule Max-mono-on, blast+*)

lemma *Max-the*:
 $\llbracket A \neq \{\}; \text{finite } A \rrbracket \implies$
 $\text{Max } A = (\text{THE } x. x \in A \wedge (\forall y. y \in A \longrightarrow y \leq x))$
apply (*rule iffD1[OF eq-commute]*)
apply (*rule the-equality, simp*)
apply (*rule sym*)
apply (*rule Max-equality*)
apply *blast+*
done

lemma *Max-the2*: $\llbracket A \neq \{\}; \text{finite } A \rrbracket \implies$
 $\text{Max } A = (\text{THE } x. x \in A \wedge (\forall y \in A. y \leq x))$
apply (*simp add: Max-the*)
apply (*subgoal-tac $\bigwedge x. (\forall y \in A. y \leq x) = (\forall y. y \in A \longrightarrow y \leq x)$*)
prefer 2
apply *blast*
apply *simp*
done

lemma *wellorder-Max-lemma*: $\llbracket k \in A; \text{finite } A \rrbracket \implies \text{Max } A \in A \wedge k \leq \text{Max } A$
by (*case-tac A = {}*, *simp-all*)

lemma *MaxI2-order*: $\llbracket k \in A; \text{finite } A; \bigwedge y. y \in A \implies y \leq k; \bigwedge x. \llbracket x \in A; \forall y \in A. y \leq x \rrbracket \implies P \text{ } x \rrbracket \implies P (\text{Max } A)$

by (simp add: Max-equality)

lemma *Min-le-Max*: $\llbracket \text{finite } A; A \neq \{\} \rrbracket \implies \text{Min } A \leq \text{Max } A$
 by (rule Max-ge[OF - Min-in])

lemma *iMin-le-Max*: $\llbracket \text{finite } A; A \neq \{\} \rrbracket \implies \text{iMin } A \leq \text{Max } A$
 by (rule ssubst[OF iMin-Min-conv], assumption+, rule Min-le-Max)

6.2.2 Max for sets over enat

definition *iMax* :: $\text{nat set} \Rightarrow \text{enat}$
 where $\text{iMax } i \equiv \text{if } (\text{finite } i) \text{ then } (\text{enat } (\text{Max } i)) \text{ else } \infty$

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

lemma *iMax-infinite-conv*: $\text{infinite } I = (\text{iMax } I = \infty)$
 by (simp add: iMax-def)

lemma *class.distrib-lattice* ($\text{min}::('a::\text{linorder} \Rightarrow 'a \Rightarrow 'a)$) (\leq) ($<$) *max*
 apply (subgoal-tac class.order (\leq) ($<$))
 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 (\leq) ($<$))
 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  $\implies$ 
   $A \neq \{\}$   $\longrightarrow$ 
   $\max x (\text{Min } A) = \text{Min } \{\max x a \mid a. a \in A\}$ 
apply (rule finite.induct[of A], simp-all del: insert-iff)
apply (rename-tac A1 a)
apply (case-tac A1 = \{\}, simp)
apply (rule-tac
   $t = \{\max x b \mid b. b \in \text{insert } a \text{ } A1\}$  and
   $s = \text{insert } (\max x a) \{\max x b \mid b. b \in A1\}$ 
  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:
   $\llbracket \text{finite } A; A \neq \{\} \rrbracket \implies$ 
   $\max x (\text{iMin } A) = \text{iMin } \{\max x a \mid a. a \in A\}$ 
apply (simp add: iMin-Min-conv)
apply (insert iMin-Min-conv[of  $\{\max x a \mid a. a \in A\}$ ], simp)
apply (subgoal-tac finite  $\{\max x a \mid a. a \in A\}$ )
prefer 2
apply simp
apply (simp add: max-Min-eq-Min-max)
done

```

```

lemma  $\llbracket \text{finite } A; A \neq \{\} \rrbracket \implies \forall x \in A. x \leq \text{Max } A$ 
by simp

```

6.2.3 *Min* and *Max* for set operations

```

lemma iMin-subset:  $\llbracket A \neq \{\}; A \subseteq B \rrbracket \implies \text{iMin } B \leq \text{iMin } A$ 
by (blast intro: iMin-le iMinI-ex2)

```

```

lemma Max-subset:  $\llbracket A \neq \{\}; A \subseteq B; \text{finite } B \rrbracket \implies \text{Max } A \leq \text{Max } B$ 
by (rule linorder-class.Max-mono)

```

```

lemma Min-subset:  $\llbracket A \neq \{\}; A \subseteq B; \text{finite } B \rrbracket \implies \text{Min } B \leq \text{Min } A$ 
by (rule linorder-class.Min-antimono)

```

lemma *iMin-Int-ge1*: $(A \cap B) \neq \{\} \implies iMin A \leq iMin (A \cap B)$
by (rule *iMin-subset*[*OF - Int-lower1*])

lemma *iMin-Int-ge2*: $(A \cap B) \neq \{\} \implies iMin B \leq iMin (A \cap B)$
by (rule *iMin-subset*[*OF - Int-lower2*])

lemma *iMin-Int-ge*: $(A \cap B) \neq \{\} \implies \max (iMin A) (iMin B) \leq iMin (A \cap B)$
apply (rule *conj-le-imp-max*)
apply (rule *iMin-Int-ge1*, *assumption*)
apply (rule *iMin-Int-ge2*, *assumption*)
done

lemma *Max-Int-le1*: $\llbracket (A \cap B) \neq \{\}; \text{finite } A \rrbracket \implies \text{Max } (A \cap B) \leq \text{Max } A$
by (rule *Max-subset*[*OF - Int-lower1*])

lemma *Max-Int-le2*: $\llbracket (A \cap B) \neq \{\}; \text{finite } B \rrbracket \implies \text{Max } (A \cap B) \leq \text{Max } B$
by (rule *Max-subset*[*OF - Int-lower2*])

lemma *Max-Int-le*: $\llbracket (A \cap B) \neq \{\}; \text{finite } A; \text{finite } B \rrbracket \implies$
 $\text{Max } (A \cap B) \leq \min (\text{Max } A) (\text{Max } B)$
apply (rule *conj-le-imp-min*)
apply (rule *Max-Int-le1*, *assumption+*)
apply (rule *Max-Int-le2*, *assumption+*)
done

lemma *iMin-Un*[*rule-format*]:
 $\llbracket A \neq \{\}; B \neq \{\} \rrbracket \implies$
 $iMin (A \cup B) = \min (iMin A) (iMin B)$
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*]: $iMax \{a\} = \text{enat } a$
by (*simp add: iMax-def*)

lemma *Max-le-Min-imp-singleton*:
 $\llbracket \text{finite } A; A \neq \{\}; \text{Max } A \leq \text{Min } A \rrbracket \implies A = \{\text{Min } A\}$
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*:
 $\llbracket \text{finite } A; A \neq \{\} \rrbracket \implies (\text{Max } A \leq \text{Min } A) = (\exists x. A = \{x\})$
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*:
 $\llbracket \text{finite } A; \text{Max } A \leq \text{iMin } B; a \in A; b \in B \rrbracket \implies a \leq b$
by (*blast dest: Max-ge intro: order-trans*)

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

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

lemma *Max-less-iMin-imp-less*:
 $\llbracket \text{finite } A; \text{Max } A < \text{iMin } B; a \in A; b \in B \rrbracket \implies a < b$
by (*blast dest: Max-less-iff intro: order-less-le-trans iMin-le*)

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

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

lemma *Max-less-iMin-imp-disjoint*:
 $\llbracket \text{finite } A; \text{Max } A < \text{iMin } B \rrbracket \implies A \cap B = \{\}$
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*: $n \in I \implies \min n (iMin I) = iMin I$
by (*simp add: iMin-le min-eqR*)

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

lemma *iMin-subset-idem*: $\llbracket B \neq \{\}; B \subseteq A \rrbracket \implies \min (iMin B) (iMin A) = iMin A$
by (*metis iMin-subset min.absorb2*)

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

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

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

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

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

lemma *hom-iMin-commute*: $\llbracket \bigwedge x y. h (\min x y) = \min (h x) (h y); I \neq \{\} \rrbracket \implies iMin (h ` I) = h (iMin I)$
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*: $\{a\} \cap A = (\text{if } a \in A \text{ then } \{a\} \text{ else } \{\})$
by (*split if-split, blast*)

lemma *singleton-Int2-if*: $A \cap \{a\} = (\text{if } a \in A \text{ then } \{a\} \text{ else } \{\})$
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* $A \implies \neg (\exists a. A = \{a\})$ **by** *blast*

lemma *infinite-insert*: *infinite* (*insert* a A) = *infinite* A
by *simp*

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

lemma *subset-finite-imp-finite*: $\llbracket \text{finite } A; B \subseteq A \rrbracket \implies \text{finite } B$
by (*rule finite-subset*)

lemma *infinite-not-subset-finite*: $\llbracket \text{infinite } A; \text{finite } B \rrbracket \implies \neg A \subseteq B$
by (*blast intro: subset-finite-imp-finite*)

lemma *Un-infinite-right*: *infinite* $T \implies \text{infinite } (S \cup T)$ **by** *blast*

lemma *Un-infinite-iff*: *infinite* ($S \cup T$) = (*infinite* $S \vee \text{infinite } T$) **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* $A = \text{finite } (\text{int } ` A)$

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*: $\text{infinite } A = \text{infinite } (\text{int } ' A)$
by (*simp only: finite-A-int-A-conv*)

lemma *cartesian-product-infiniteL-imp-infinite*: $\llbracket \text{infinite } A; B \neq \{\} \rrbracket \implies \text{infinite } (A \times B)$
by (*blast dest: finite-cartesian-productD1*)

lemma *cartesian-product-infiniteR-imp-infinite*: $\llbracket \text{infinite } B; A \neq \{\} \rrbracket \implies \text{infinite } (A \times B)$
by (*blast dest: finite-cartesian-productD2*)

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

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

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

lemma *infinite-nat-iff-asc-chain*:
 $S \neq \{\} \implies \text{infinite } S = (\forall m \in S. \exists n \in S. m < (n::\text{nat}))$
by (*metis Max-in infinite-nat-iff-unbounded not-greater-Max*)

lemma *infinite-imp-asc-chain*: $\text{infinite } S \implies \forall m \in S. \exists n \in S. m < (n::\text{nat})$
by (*rule infinite-nat-iff-asc-chain[THEN iffD1, OF infinite-imp-nonempty]*)

lemma *infinite-image-imp-infinite*: $\text{infinite } (f ' A) \implies \text{infinite } A$
by *fastforce*

lemma *inj-on-imp-infinite-image*: $\llbracket \text{infinite } A; \text{inj-on } f A \rrbracket \implies \text{infinite } (f ' A)$
apply (*frule card-image*)
apply (*fastforce simp: card-eq-0-iff*)

done

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

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

lemma *nat-ex-greater-finite-Max-conv*:
 $A \neq \{\} \implies (\exists x \in A. (n::\text{nat}) < x) = (\text{finite } A \longrightarrow n < \text{Max } A)$
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'*:
 $(\exists x \in A. (n::\text{nat}) < x) = (\text{finite } A \wedge A \neq \{\} \wedge n < \text{Max } A \vee \text{infinite } A)$
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*:

$\llbracket A \cap B = \{\}; x \in A \cup B \rrbracket \implies x \notin A \vee x \notin B$
by (blast *intro: disjoint-iff-in-not-in1[THEN iffD1]*)**+**

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

lemma *disjoint-partition-Int*:

$\forall c1 \in C. \forall c2 \in C. c1 \neq c2 \longrightarrow c1 \cap c2 = \{\} \implies$
 $\forall a1 \in \{A \cap c \mid c. c \in C\}. \forall a2 \in \{A \cap c \mid c. c \in C\}.$
 $a1 \neq a2 \longrightarrow a1 \cap a2 = \{\}$

by *blast*

lemma $\{f x \mid x. x \in A\} = (\bigcup x \in A. \{f x\})$

by *fastforce*

This lemma version drops the superfluous precondition *finite* $(\bigcup C)$ (and

turns the resulting equation to the sensible order $\text{card } .. = k * \text{card } ..$).

lemma *card-partition*:

$\llbracket \text{finite } C; \bigwedge c. c \in C \implies \text{card } c = k; \bigwedge c1\ c2. \llbracket c1 \in C; c2 \in C; c1 \neq c2 \rrbracket \implies$
 $c1 \cap c2 = \{\} \rrbracket \implies$
 $\text{card } (\bigcup C) = k * \text{card } C$

by (*metis card.infinite card-partition finite-Union mult-eq-if*)

6.3.5 Some auxiliary results for subset relation

lemma *subset-image-Int*: $A \subseteq B \implies f' (A \cap B) = f' A \cap f' B$

by (*simp only: Int-absorb2 image-mono*)

lemma *image-diff-disjoint-image-Int*:

$\llbracket f' (A - B) \cap f' B = \{\} \rrbracket \implies$
 $f' (A \cap B) = f' A \cap f' B$

apply (*rule set-eqI*)

apply (*simp add: image-iff*)

apply *blast*

done

lemma *subset-imp-Int-subset1*: $A \subseteq C \implies A \cap B \subseteq C$

by (*rule subset-trans[of - C \cap B, OF Int-mono, OF - subset-refl Int-lower1]*)

lemma *subset-imp-Int-subset2*: $B \subseteq C \implies A \cap B \subseteq C$

by (*simp only: Int-commute[of A], rule subset-imp-Int-subset1*)

6.3.6 Auxiliary results for intervals from *SetInterval*

lemmas *set-interval-defs* =

lessThan-def atMost-def

greaterThan-def atLeast-def

greaterThanLessThan-def atLeastLessThan-def

greaterThanAtMost-def atLeastAtMost-def

lemma *image-add-atLeast*:

$(\lambda n::\text{nat}. n+k) ' \{i..\} = \{i+k..\} \text{ (is } ?A = ?B)$

proof

show $?A \subseteq ?B$ **by** *fastforce*

next

show $?B \subseteq ?A$

proof

fix n **assume** $a: n : ?B$

hence $n - k \in \{i..\}$ **by** *simp*

moreover have $n = (n - k) + k$ **using** a **by** *fastforce*

ultimately show $n \in ?A$ **by** *blast*

qed

qed

lemma *image-add-atMost*:

$(\lambda n::nat. n+k) \cdot \{..i\} = \{k..i+k\}$ (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*: $Suc \cdot \{i..\} = \{Suc\ i..\}$
by (*insert image-add-atLeast[of Suc 0], simp*)

corollary *image-Suc-atMost*: $Suc \cdot \{..i\} = \{Suc\ 0..Suc\ i\}$
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*: $(k::'a::order) \leq k' \implies \{l..k\} \subseteq \{..k'\}$
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*: $finite\ A \implies A \subseteq \{Min\ A..Max\ A\}$
by (*simp add: subset-eq*)

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

lemma *subset-atMost-imp-Max-le*:
 $\llbracket finite\ A; A \neq \{\}; A \subseteq \{..n\} \rrbracket \implies Max\ A \leq n$
by (*simp add: subset-iff*)

lemma *subset-atMost-Max-le-conv*:
 $\llbracket finite\ A; A \neq \{\} \rrbracket \implies (A \subseteq \{..n\}) = (Max\ A \leq n)$
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*: $0 < n \implies iMin\ \{..<(n::nat)\} = 0$

by (rule *iMin-equality*, *simp-all*)

lemma *Max-atMost*: $\text{Max } \{..(n::\text{nat})\} = n$

by (rule *Max-equality*[*OF - finite-atMost*], *simp-all*)

lemma *Max-lessThan*: $0 < n \implies \text{Max } \{..<n\} = n - \text{Suc } 0$

by (rule *Max-equality*[*OF - finite-lessThan*], *simp-all*)

lemma *iMin-atLeastLessThan*: $m < n \implies \text{iMin } \{m..<n\} = m$

by (rule *iMin-equality*, *simp-all*)

lemma *Max-atLeastLessThan*: $m < n \implies \text{Max } \{m..<n\} = n - \text{Suc } 0$

by (rule *Max-equality*[*OF - finite-atLeastLessThan*], *simp-all* add: *less-imp-le-pred*)

lemma *iMin-greaterThanLessThan*: $\text{Suc } m < n \implies \text{iMin } \{m<..<n\} = \text{Suc } m$

by (*simp only*: *atLeastSucLessThan-greaterThanLessThan*[*symmetric*] *iMin-atLeastLessThan*)

lemma *Max-greaterThanLessThan*: $\text{Suc } m < n \implies \text{Max } \{m<..<n\} = n - \text{Suc } 0$

by (*simp only*: *atLeastSucLessThan-greaterThanLessThan*[*symmetric*] *Max-atLeastLessThan*)

lemma *iMin-greaterThanAtMost*: $m < n \implies \text{iMin } \{m<..n\} = \text{Suc } m$

by (*simp only*: *atLeastSucAtMost-greaterThanAtMost*[*symmetric*] *atLeastLessThanSuc-atLeastAtMost*[*symmetric*] *iMin-atLeastLessThan*)

lemma *Max-greaterThanAtMost*: $m < n \implies \text{Max } \{m<..(n::\text{nat})\} = n$

by (*simp add*: *atLeastSucAtMost-greaterThanAtMost*[*symmetric*] *atLeastLessThanSuc-atLeastAtMost*[*symmetric*] *Max-atLeastLessThan*)

lemma *iMin-atLeastAtMost*: $m \leq n \implies \text{iMin } \{m..n\} = m$

by (rule *iMin-equality*, *simp-all*)

lemma *Max-atLeastAtMost*: $m \leq n \implies \text{Max } \{m..(n::\text{nat})\} = n$

by (rule *Max-equality*[*OF - finite-atLeastAtMost*], *simp-all*)

lemma *infinite-atLeast*: $\text{infinite } \{(n::\text{nat})..\}$

by (rule *unbounded-k-infinite*[*of n*], *fastforce*)

lemma *infinite-greaterThan*: $\text{infinite } \{(n::\text{nat})<..\}$

by (*simp add*: *atLeast-Suc-greaterThan*[*symmetric*] *infinite-atLeast*)

lemma *infinite-atLeast-int*: $\text{infinite } \{(n::\text{int})..\}$

apply (rule-tac $f = \lambda x. \text{nat } (x - n)$ **in** *inj-on-infinite-image-iff*[*THEN iffD1*, *rule-format*])

apply (*fastforce simp*: *inj-on-def*)

apply (rule-tac $t = ((\lambda x. \text{nat } (x - n)) \text{ ' } \{n..\})$ **and** $s = \{0..\}$ **in** *subst*)

apply (*simp add*: *set-eq-iff image-iff Bex-def*)

apply (*clarify*, *rename-tac n1*)

apply (rule-tac $x = n + \text{int } n1$ **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  $f = \lambda x. n - x$  in inj-on-infinite-image-iff [THEN iffD1, rule-format])
apply (simp add: inj-on-def)
apply (rule-tac  $t = ((-) n \text{ ' } \{..n\})$  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:  $(\sum x \in \{a\}. f x) = f a$ 
by simp

```

```

lemma card-singleton:  $\text{card } \{a\} = \text{Suc } 0$ 
by simp

```

```

lemma card-cartesian-product-singleton-right:  $\text{card } (A \times \{x\}) = \text{card } A$ 
by (simp add: card-cartesian-product)

```

```

lemma card-1-imp-singleton:  $\text{card } A = \text{Suc } 0 \implies (\exists a. A = \{a\})$ 
by (metis card-eq-SucD)

```

```

lemma card-1-singleton-conv:  $(\text{card } A = \text{Suc } 0) = (\exists a. A = \{a\})$ 
apply (rule iffI)
apply (simp add: card-1-imp-singleton)
apply fastforce
done

```

```

lemma card-gr0-imp-finite:  $0 < \text{card } A \implies \text{finite } A$ 
by (rule ccontr, simp)

```

```

lemma card-gr0-imp-not-empty:  $(0 < \text{card } A) \implies A \neq \{\}$ 
by (rule ccontr, simp)

```

```

lemma not-empty-card-gr0-conv:  $\text{finite } A \implies (A \neq \{\}) = (0 < \text{card } A)$ 
by fastforce

```

```

lemma nat-card-le-Max:  $\text{card } (A::\text{nat set}) \leq \text{Suc } (\text{Max } A)$ 

```

```

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

```

```

lemma card-Un-conv:
   $\llbracket$  finite A; finite B  $\rrbracket \implies$ 
  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:
   $\llbracket$  finite A; finite B  $\rrbracket \implies$ 
  card (A  $\cap$  B) = card A + card B - card (A  $\cup$  B)
by (simp only: card-Un-Int diff-add-inverse)

```

Pigeonhole principle, dirichlet's box principle

```

lemma pigeonhole-principle[rule-format]:
  card (f ' A) < card A  $\longrightarrow$  ( $\exists x \in A. \exists y \in A. x \neq y \wedge f x = f y$ )
apply (case-tac finite A)
  prefer 2
  apply simp
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 ' A) < card (A::'a::linorder set)  $\implies$  ( $\exists x \in A. \exists y \in A. x < y \wedge f x = f y$ )
apply (drule pigeonhole-principle, clarify)
apply (drule neq-iff[THEN iffD1])
apply fastforce
done

```

```

corollary pigeonhole-mod:
   $\llbracket 0 < m; m < \text{card } A \rrbracket \implies \exists x \in A. \exists y \in A. x < y \wedge x \bmod m = y \bmod m$ 
apply (rule pigeonhole-principle-linorder)

```

```

apply (rule le-less-trans[of - card {.. $m$ }])
apply (rule card-mono)
apply fastforce+
done

```

corollary *pigeonhole-mod2*:

$\llbracket (0::nat) < m; m \leq c; inj\text{-on } f \{..c\} \rrbracket \implies \exists x \leq c. \exists y \leq c. x < y \wedge f x \bmod m = f y \bmod m$

```

apply (insert pigeonhole-mod[of  $m$   $f$  ‘{.. $c$ }’])
apply (clarsimp simp add: card-image, rename-tac  $x$   $y$ )
apply (subgoal-tac  $x \neq y$ )
  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 \Rightarrow $'a$ set) \Rightarrow bool
where *set-restriction* $f \equiv \exists P. \forall A. f A = \{x \in A. P x\}$

lemma *set-restrictionD*: *set-restriction* $f \implies \exists P. \forall A. f A = \{x \in A. P x\}$

unfolding *set-restriction-def* **by** blast

lemma *set-restrictionD-spec*: *set-restriction* $f \implies \exists P. f A = \{x \in A. P x\}$

unfolding *set-restriction-def* **by** blast

lemma *set-restrictionI*: $f = (\lambda A. \{x \in A. P x\}) \implies \text{set-restriction } f$

unfolding *set-restriction-def* **by** blast

lemma *set-restriction-comp*:

$\llbracket \text{set-restriction } f; \text{set-restriction } g \rrbracket \implies \text{set-restriction } (f \circ g)$

apply (unfold *set-restriction-def*)

apply (elim exE, rename-tac $P1$ $P2$)

apply (rule-tac $x = \lambda x. P1 x \wedge P2 x$ **in** exI)

apply fastforce

done

lemma *set-restriction-commute*:

$\llbracket \text{set-restriction } f; \text{set-restriction } g \rrbracket \implies f (g I) = g (f I)$

unfolding *set-restriction-def* **by** *fastforce*

Constructs a set restriction function with the given restriction predicate

definition

$\text{set-restriction-fun} :: ('a \Rightarrow \text{bool}) \Rightarrow ('a \text{ set} \Rightarrow 'a \text{ set})$

where

$\text{set-restriction-fun } P \equiv \lambda A. \{x \in A. P x\}$

lemma *set-restriction-fun-is-set-restriction*:

$\text{set-restriction } (\text{set-restriction-fun } P)$

unfolding *set-restriction-def set-restriction-fun-def* **by** *blast*

lemma *set-restriction-Int-conv*:

$\text{set-restriction } f = (\exists B. \forall A. f A = A \cap B)$

apply (*unfold set-restriction-def*)

apply (*rule iffI*)

apply (*erule exE, rule-tac x=Collect P in exI, blast*)

apply (*erule exE, rule-tac x= $\lambda x. x \in B$ in exI, blast*)

done

lemma *set-restriction-Un*:

$\text{set-restriction } f \implies f (A \cup B) = f A \cup f B$

unfolding *set-restriction-def* **by** *fastforce*

lemma *set-restriction-Int*:

$\text{set-restriction } f \implies f (A \cap B) = f A \cap f B$

unfolding *set-restriction-def* **by** *fastforce*

lemma *set-restriction-Diff*:

$\text{set-restriction } f \implies f (A - B) = f A - f B$

unfolding *set-restriction-def* **by** *fastforce*

lemma *set-restriction-mono*:

$\llbracket \text{set-restriction } f; A \subseteq B \rrbracket \implies f A \subseteq f B$

unfolding *set-restriction-def* **by** *fastforce*

lemma *set-restriction-absorb*:

$\text{set-restriction } f \implies f (f A) = f A$

unfolding *set-restriction-def* **by** *fastforce*

lemma *set-restriction-empty*:

$\text{set-restriction } f \implies f \{\} = \{\}$

unfolding *set-restriction-def* **by** *blast*

lemma *set-restriction-non-empty-imp*:

$\llbracket \text{set-restriction } f; f A \neq \{\} \rrbracket \implies A \neq \{\}$

unfolding *set-restriction-def* **by** *blast*

lemma *set-restriction-subset*:

$\text{set-restriction } f \implies f A \subseteq A$

unfolding *set-restriction-def* **by** *blast*

lemma *set-restriction-finite*:

$\llbracket \text{set-restriction } f; \text{finite } A \rrbracket \implies \text{finite } (f A)$

unfolding *set-restriction-def* **by** *fastforce*


```

lemma set-restriction-card:
   $\llbracket \text{set-restriction } f; \text{finite } A \rrbracket \implies$ 
   $\text{card } (f A) = \text{card } A - \text{card } \{a \in A. f \{a\} = \{\}\}$ 
apply (unfold set-restriction-def)
apply (subgoal-tac  $\{a \in A. f \{a\} = \{\}\} \subseteq A$ )
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:
   $\llbracket \text{set-restriction } f; \text{finite } A \rrbracket \implies \text{card } (f A) \leq \text{card } A$ 
by (simp add: set-restriction-card)

```

```

lemma set-restriction-not-in-imp:
   $\llbracket \text{set-restriction } f; x \notin A \rrbracket \implies x \notin f A$ 
unfolding set-restriction-def by blast

```

```

lemma set-restriction-in-imp:
   $\llbracket \text{set-restriction } f; x \in f A \rrbracket \implies x \in A$ 
unfolding set-restriction-def by blast

```

```

lemma set-restriction-fun-singleton:
   $\text{set-restriction-fun } P \{a\} = (\text{if } P a \text{ then } \{a\} \text{ else } \{\})$ 
unfolding set-restriction-fun-def by force
lemma set-restriction-fun-all-conv:
   $((\text{set-restriction-fun } P) A = A) = (\forall x \in A. P x)$ 
unfolding set-restriction-fun-def by blast
lemma set-restriction-fun-empty-conv:
   $((\text{set-restriction-fun } P) A = \{\}) = (\forall x \in A. \neg P x)$ 
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  $\Rightarrow$  'a  $\Rightarrow$  'a set (infixl  $\downarrow \leq$  100)
  where  $I \downarrow \leq t \equiv \{x \in I. x \leq t\}$ 

```

```

definition cut-less :: 'a::linorder set  $\Rightarrow$  'a  $\Rightarrow$  'a set (infixl  $\downarrow <$  100)
  where  $I \downarrow < t \equiv \{x \in I. x < t\}$ 

```

```

definition cut-ge :: 'a::linorder set  $\Rightarrow$  'a  $\Rightarrow$  'a set (infixl  $\downarrow \geq$  100)
  where  $I \downarrow \geq t \equiv \{x \in I. t \leq x\}$ 

```

```

definition cut-greater :: 'a::linorder set  $\Rightarrow$  'a  $\Rightarrow$  'a set (infixl  $\downarrow >$  100)
  where  $I \downarrow > t \equiv \{x \in I. t < x\}$ 

```

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

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

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

lemma
cut-less-bound: $n \in I \Downarrow < t \implies n < t$ **and**
cut-le-bound: $n \in I \Downarrow \leq t \implies n \leq t$ **and**
cut-greater-bound: $n \in i \Downarrow > t \implies t < n$ **and**
cut-ge-bound: $n \in i \Downarrow \geq t \implies t \leq n$
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': $(\lambda I. I \downarrow < t) = \text{set-restriction-fun } (\lambda x. x < t)$

and

cut-le-eq-set-restriction-fun': $(\lambda I. I \downarrow \leq t) = \text{set-restriction-fun } (\lambda x. x \leq t)$

and

cut-greater-eq-set-restriction-fun': $(\lambda I. I \downarrow > t) = \text{set-restriction-fun } (\lambda x. x > t)$

and

cut-ge-eq-set-restriction-fun': $(\lambda I. I \downarrow \geq t) = \text{set-restriction-fun } (\lambda x. x \geq t)$

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: $I \downarrow < t = \text{set-restriction-fun } (\lambda x. x < t) I$ **and**

cut-le-eq-set-restriction-fun: $I \downarrow \leq t = \text{set-restriction-fun } (\lambda x. x \leq t) I$ **and**

cut-greater-eq-set-restriction-fun: $I \downarrow > t = \text{set-restriction-fun } (\lambda x. x > t) I$ **and**

cut-ge-eq-set-restriction-fun: $I \downarrow \geq t = \text{set-restriction-fun } (\lambda x. x \geq t) I$

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*:

$\llbracket \text{cut-op} = (\downarrow <) \vee \text{cut-op} = (\downarrow \leq) \vee$

$\text{cut-op} = (\downarrow >) \vee \text{cut-op} = (\downarrow \geq);$

$f = (\lambda I. \text{cut-op } I t) \rrbracket \implies \text{set-restriction } f$

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*: \llbracket

cut-op1 = ($\downarrow <$) \vee *cut-op1* = ($\downarrow \leq$) \vee
cut-op1 = ($\downarrow >$) \vee *cut-op1* = ($\downarrow \geq$);
cut-op2 = ($\downarrow <$) \vee *cut-op2* = ($\downarrow \leq$) \vee
cut-op2 = ($\downarrow >$) \vee *cut-op2* = ($\downarrow \geq$) $\rrbracket \implies$
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: $I \downarrow < t \neq \{\} \implies I \neq \{\}$ **and**
cut-le-not-empty-imp: $I \downarrow \leq t \neq \{\} \implies I \neq \{\}$ **and**
cut-greater-not-empty-imp: $I \downarrow > t \neq \{\} \implies I \neq \{\}$ **and**
cut-ge-not-empty-imp: $I \downarrow \geq t \neq \{\} \implies I \neq \{\}$

by *blast+*

lemma

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

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*:

$\llbracket \text{cut-op} = (\downarrow <) \vee \text{cut-op} = (\downarrow \leq) \vee$
 $\text{cut-op} = (\downarrow >) \vee \text{cut-op} = (\downarrow \geq) \rrbracket$
 $\implies \text{cut-op } (A \cup B) t = \text{cut-op } A t \cup \text{cut-op } B t$

apply (*drule i-cut-set-restriction-disj*[**where** $f = \lambda I. \text{cut-op } I t$], *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*:

$\llbracket \text{cut-op} = (\downarrow <) \vee \text{cut-op} = (\downarrow \leq) \vee$
 $\text{cut-op} = (\downarrow >) \vee \text{cut-op} = (\downarrow \geq) \rrbracket$
 $\implies \text{cut-op } (A \cap B) t = \text{cut-op } A t \cap \text{cut-op } B t$

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

corollary

cut-less-subset-mono: $A \subseteq B \implies A \downarrow < t \subseteq B \downarrow < t$ **and**
cut-le-subset-mono: $A \subseteq B \implies A \downarrow \leq t \subseteq B \downarrow \leq t$ **and**
cut-greater-subset-mono: $A \subseteq B \implies A \downarrow > t \subseteq B \downarrow > t$ **and**
cut-ge-subset-mono: $A \subseteq B \implies A \downarrow \geq t \subseteq B \downarrow \geq t$

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: $t \leq t' \implies I \downarrow < t \subseteq I \downarrow < t'$ **and**
cut-greater-mono: $t' \leq t \implies I \downarrow > t \subseteq I \downarrow > t'$ **and**
cut-le-mono: $t \leq t' \implies I \downarrow \leq t \subseteq I \downarrow \leq t'$ **and**
cut-ge-mono: $t' \leq t \implies I \downarrow \geq t \subseteq I \downarrow \geq t'$

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*:

$\llbracket \text{cut-op} = (\downarrow <) \vee \text{cut-op} = (\downarrow \leq) \vee$
 $\text{cut-op} = (\downarrow >) \vee \text{cut-op} = (\downarrow \geq) \rrbracket$
 $\implies \text{cut-op} (\text{cut-op } I t) t = \text{cut-op } I t$

apply (*drule i-cut-set-restriction-disj*[**where** $f = \lambda I. \text{cut-op } I t$], *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* $I \implies I \downarrow \geq (t::nat) \neq \{\}$
by (drule *infinite-nat-iff-unbounded-le*[*THEN iffD1*], *blast*)

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

corollary

cut-le-not-in-imp: $x \notin I \implies x \notin I \downarrow \leq t$ **and**
cut-less-not-in-imp: $x \notin I \implies x \notin I \downarrow < t$ **and**
cut-ge-not-in-imp: $x \notin I \implies x \notin I \downarrow \geq t$ **and**
cut-greater-not-in-imp: $x \notin I \implies x \notin I \downarrow > t$

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: $x \in I \downarrow \leq t \implies x \in I$ **and**
cut-less-in-imp: $x \in I \downarrow < t \implies x \in I$ **and**
cut-ge-in-imp: $x \in I \downarrow \geq t \implies x \in I$ **and**
cut-greater-in-imp: $x \in I \downarrow > t \implies x \in I$

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*: $\llbracket k \in I; P (k::('a::\text{wellorder})) \rrbracket \implies \exists x \in I. P x \wedge (\forall y \in (I \downarrow < x). \neg P y)$

by (*drule Collect-minI*, *assumption*, *blast*)

corollary *Collect-minI-ex-cut*: $\exists k \in I. P (k::('a::\text{wellorder})) \implies \exists x \in I. P x \wedge (\forall y \in (I \downarrow < x). \neg P y)$

by (*drule Collect-minI-ex*, *blast*)

corollary *Collect-minI-ex2-cut*: $\{k \in I. P (k::('a::\text{wellorder}))\} \neq \{\} \implies \exists x \in I. P x \wedge (\forall y \in (I \downarrow < x). \neg P y)$

by (*drule Collect-minI-ex2*, *blast*)

lemma *cut-le-cut-greater-ident*: $t2 \leq t1 \implies I \downarrow \leq t1 \cup I \downarrow > t2 = I$

by *fastforce*

lemma *cut-less-cut-ge-ident*: $t2 \leq t1 \implies I \downarrow < t1 \cup I \downarrow \geq t2 = I$

by *fastforce*

lemma *cut-le-cut-ge-ident*: $t2 \leq t1 \implies I \downarrow \leq t1 \cup I \downarrow \geq t2 = I$

by *fastforce*

lemma *cut-less-cut-greater-ident*:

$$\llbracket t2 \leq t1; I \cap \{t1..t2\} = \{\} \rrbracket \implies I \downarrow < t1 \cup I \downarrow > t2 = I$$

by *fastforce*

corollary *cut-less-cut-greater-ident'*:

$$t \notin I \implies I \downarrow < t \cup I \downarrow > t = I$$

by (*simp add: cut-less-cut-greater-ident*)

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

by *fastforce*

7.2.3 Relations between cut operators

lemma *insert-Int-conv-if*: $A \cap (insert\ x\ B) = ($

$$if\ x \in A\ then\ insert\ x\ (A \cap B)\ else\ A \cap B)$$

by *simp*

lemma *cut-le-less-conv-if*: $I \downarrow \leq t = ($

$$if\ t \in I\ then\ insert\ t\ (I \downarrow < t)\ else\ (I \downarrow < t))$$

by (*simp add: i-cut-Int-conv lessThan-insert[symmetric] insert-Int-conv-if*)

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

by *fastforce*

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

by *fastforce*

lemma *cut-less-le-conv-if*: $I \downarrow < t = ($

$$if\ t \in I\ then\ (I \downarrow \leq t) - \{t\}\ else\ (I \downarrow \leq t))$$

by (*simp only: cut-less-le-conv, force*)

lemma *cut-ge-greater-conv-if*: $I \downarrow \geq t = ($

$$if\ t \in I\ then\ insert\ t\ (I \downarrow > t)\ else\ (I \downarrow > t))$$

by (*simp add: i-cut-Int-conv greaterThan-insert[symmetric] insert-Int-conv-if*)

lemma *cut-ge-greater-conv*: $I \downarrow \geq t = (\{t\} \cap I) \cup (I \downarrow > t)$

apply (*simp only: cut-ge-greater-conv-if*)

apply (*case-tac t \in I*)

apply *simp-all*

done

lemma *cut-greater-ge-conv*: $I \downarrow > t = (I \downarrow \geq t) - \{t\}$

by *fastforce*

lemma *cut-greater-ge-conv-if*: $I \downarrow > t = ($

$$if\ t \in I\ then\ (I \downarrow \geq t) - \{t\}\ else\ (I \downarrow \geq t))$$

by (*simp only: cut-greater-ge-conv, force*)

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

7.2.4 Function images with cut operators

lemma *cut-less-image*:
 $\llbracket \text{strict-mono-on } f \ A; I \subseteq A; n \in A \rrbracket \implies$
 $(f \ ' \ I) \downarrow < f \ n = f \ ' \ (I \downarrow < n)$
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*:
 $\llbracket \text{strict-mono-on } f \ A; I \subseteq A; n \in A \rrbracket \implies$
 $(f \ ' \ I) \downarrow \leq f \ n = f \ ' \ (I \downarrow \leq n)$
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*:
 $\llbracket \text{strict-mono-on } f \ A; I \subseteq A; n \in A \rrbracket \implies$
 $(f \ ' \ I) \downarrow > f \ n = f \ ' \ (I \downarrow > n)$
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*:
 $\llbracket \text{strict-mono-on } f \ A; I \subseteq A; n \in A \rrbracket \implies$
 $(f \ ' \ I) \downarrow \geq f \ n = f \ ' \ (I \downarrow \geq n)$
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 \implies finite (I $\downarrow \leq$ t) and
cut-less-finite: finite I \implies finite (I $\downarrow <$ t) and
cut-ge-finite: finite I \implies finite (I $\downarrow \geq$ t) and
cut-greater-finite: finite I \implies 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 \implies card (I $\downarrow \leq$ t) \leq card I and
cut-less-card: finite I \implies card (I $\downarrow <$ t) \leq card I and
cut-ge-card: finite I \implies card (I $\downarrow \geq$ t) \leq card I and
cut-greater-card: finite I \implies 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*: $finite\ I \implies I \downarrow < (Max\ I) = I - \{Max\ I\}$

by *blast*

lemma *cut-le-Min-empty*: $t < iMin\ I \implies I \downarrow \leq t = \{\}$

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

lemma *cut-less-Min-empty*: $t \leq iMin\ I \implies I \downarrow < t = \{\}$

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

lemma *cut-le-Min-not-empty*: $\llbracket I \neq \{\}; iMin\ I \leq t \rrbracket \implies I \downarrow \leq t \neq \{\}$

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*: $\llbracket I \neq \{\}; iMin\ I < t \rrbracket \implies I \downarrow < t \neq \{\}$

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*: $t \leq iMin\ I \implies I \downarrow \geq t = I$

apply (*simp add: i-cut-defs*)

apply *safe*

apply (*drule iMin-le, simp*)

done

lemma *cut-greater-Min-all*: $t < iMin\ I \implies I \downarrow > t = I$

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*: $finite\ I \implies Max\ I < t \implies I \downarrow \geq t = \{\}$

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

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

lemma *cut-ge-Max-not-empty*: $\llbracket I \neq \{\}; \text{finite } I; t \leq \text{Max } I \rrbracket \implies I \downarrow \geq t \neq \{\}$
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*: $\llbracket I \neq \{\}; \text{finite } I; t < \text{Max } I \rrbracket \implies I \downarrow > t \neq \{\}$
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*: $\text{finite } I \implies \text{Max } I \leq t \implies I \downarrow \leq t = I$
by (*fastforce simp: i-cut-defs*)

lemma *cut-less-Max-all*: $\text{finite } I \implies \text{Max } I < t \implies I \downarrow < t = I$
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*:
 $\llbracket \text{finite } (I \downarrow < t); I \downarrow < t \neq \{\} \rrbracket \implies \text{Max } (I \downarrow < t) < t$
by (*rule cut-less-bound[OF Max-in]*)

lemma *cut-le-Max-le*:
 $\llbracket \text{finite } (I \downarrow \leq t); I \downarrow \leq t \neq \{\} \rrbracket \implies \text{Max } (I \downarrow \leq t) \leq t$
by (*rule cut-le-bound[OF Max-in]*)

lemma *nat-cut-less-Max-less*:
 $I \downarrow < t \neq \{\} \implies \text{Max } (I \downarrow < t) < (t::\text{nat})$
by (*rule cut-less-bound[OF Max-in[OF nat-cut-less-finite]]*)

lemma *nat-cut-le-Max-le*:
 $I \downarrow \leq t \neq \{\} \implies \text{Max } (I \downarrow \leq t) \leq (t::\text{nat})$
by (*rule cut-le-bound[OF Max-in[OF nat-cut-le-finite]]*)

lemma *cut-greater-Min-greater*:
 $I \downarrow > t \neq \{\} \implies i\text{Min } (I \downarrow > t) > t$
by (*rule cut-greater-bound[OF iMinI-ex2]*)

lemma *cut-ge-Min-greater*:

$I \downarrow \geq t \neq \{\}$ $\implies iMin (I \downarrow \geq t) \geq t$
 by (rule cut-ge-bound[OF iMinI-ex2])

lemma cut-less-Min-eq: $I \downarrow < t \neq \{\} \implies iMin (I \downarrow < t) = iMin I$
 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: $I \downarrow \leq t \neq \{\} \implies iMin (I \downarrow \leq t) = iMin I$
 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: $\llbracket finite I; I \downarrow \geq t \neq \{\} \rrbracket \implies Max (I \downarrow \geq t) = Max I$
 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: $\llbracket finite I; I \downarrow > t \neq \{\} \rrbracket \implies Max (I \downarrow > t) = Max I$
 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: $\{..n\} \downarrow \leq t = (\text{if } n \leq t \text{ then } \{..n\} \text{ else } \{..t\})$ **and**
atMost-cut-less: $\{..n\} \downarrow < t = (\text{if } n < t \text{ then } \{..n\} \text{ else } \{..<t\})$ **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: $\{n<..\} \downarrow \geq t = (\text{if } t \leq n \text{ then } \{n<..\} \text{ else } \{t..n\})$ **and**
greaterThan-cut-greater: $\{n<..\} \downarrow > t = (\text{if } t \leq n \text{ then } \{n<..\} \text{ else } \{t<..\})$ **and**
atLeast-cut-le: $\{n..\} \downarrow \leq t = \{n..t\}$ **and**
atLeast-cut-less: $\{n..\} \downarrow < t = \{n..<t\}$ **and**
atLeast-cut-ge: $\{n..\} \downarrow \geq t = (\text{if } t \leq n \text{ then } \{n..\} \text{ else } \{t..n\})$ **and**
atLeast-cut-greater: $\{n..\} \downarrow > t = (\text{if } t \leq n \text{ then } \{n..\} \text{ else } \{t..n\})$
apply (*simp-all add: set-eq-iff i-cut-mem-iff linorder-not-le linorder-not-less*)
apply *fastforce+*
done

lemma

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

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

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

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

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

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

lemma *nat-mirror-add*:
 $\text{nat-mirror } (n + k) \ l \ r = (\text{nat-mirror } n \ l \ r) - k$
unfolding *nat-mirror-def* **by** *simp*

lemma *nat-mirror-diff*:
 $\llbracket k \leq n; n \leq l + r \rrbracket \implies$
 $\text{nat-mirror } (n - k) \ l \ r = (\text{nat-mirror } n \ l \ r) + k$
unfolding *nat-mirror-def* **by** *simp*

lemma *nat-mirror-le*: $a \leq b \implies \text{nat-mirror } b \ l \ r \leq \text{nat-mirror } a \ l \ r$
unfolding *nat-mirror-def* **by** *simp*

lemma *nat-mirror-le-conv*:
 $a \leq l + r \implies (\text{nat-mirror } b \ l \ r \leq \text{nat-mirror } a \ l \ r) = (a \leq b)$
unfolding *nat-mirror-def* **by** *fastforce*

lemma *nat-mirror-less*:
 $\llbracket a < b; a < l + r \rrbracket \implies$
 $\text{nat-mirror } b \ l \ r < \text{nat-mirror } a \ l \ r$
unfolding *nat-mirror-def* **by** *simp*

lemma *nat-mirror-less-imp-less*:
 $\text{nat-mirror } b \ l \ r < \text{nat-mirror } a \ l \ r \implies a < b$
unfolding *nat-mirror-def* **by** *simp*

lemma *nat-mirror-less-conv*:
 $a < l + r \implies (\text{nat-mirror } b \ l \ r < \text{nat-mirror } a \ l \ r) = (a < b)$
unfolding *nat-mirror-def* **by** *fastforce*

lemma *nat-mirror-eq-conv*:

$\llbracket a \leq l + r; b \leq l + r \rrbracket \implies$
 $(\text{nat-mirror } a \ l \ r = \text{nat-mirror } b \ l \ r) = (a = b)$
unfolding *nat-mirror-def* **by** *fastforce*

Mirroring a single element n between the interval boundaries of I

definition

mirror-elem :: *nat* \Rightarrow *nat set* \Rightarrow *nat*

where

mirror-elem $n \ I \equiv \text{nat-mirror } n \ (\text{iMin } I) \ (\text{Max } I)$

lemma *mirror-elem-inj-on*: *finite I* \implies *inj-on* $(\lambda x. \text{mirror-elem } x \ I) \ I$

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 $\implies \text{mirror-elem } (n + k) \ I = \text{mirror-elem } n \ I - k$

unfolding *mirror-elem-def* **by** (*rule nat-mirror-add*)

lemma *mirror-elem-diff*:

$\llbracket \text{finite } I; k \leq n; n \in I \rrbracket \implies \text{mirror-elem } (n - k) \ I = \text{mirror-elem } n \ I + k$

apply (*unfold mirror-elem-def*)

apply (*rule nat-mirror-diff, assumption*)

apply (*simp add: trans-le-add2*)

done

lemma *mirror-elem-Min*:

$\llbracket \text{finite } I; I \neq \{\} \rrbracket \implies \text{mirror-elem } (\text{iMin } I) \ I = \text{Max } I$

unfolding *mirror-elem-def nat-mirror-def* **by** *simp*

lemma *mirror-elem-Max*:

$\llbracket \text{finite } I; I \neq \{\} \rrbracket \implies \text{mirror-elem } (\text{Max } I) \ I = \text{iMin } I$

unfolding *mirror-elem-def nat-mirror-def* **by** *simp*

lemma *mirror-elem-mirror-elem-ident*:

$\llbracket \text{finite } I; n \leq \text{iMin } I + \text{Max } I \rrbracket \implies \text{mirror-elem } (\text{mirror-elem } n \ I) \ I = n$

unfolding *mirror-elem-def nat-mirror-def* **by** *simp*

lemma *mirror-elem-le-conv*:

$\llbracket \text{finite } I; a \in I; b \in I \rrbracket \implies$

$(\text{mirror-elem } b \ I \leq \text{mirror-elem } a \ I) = (a \leq b)$

apply (*unfold mirror-elem-def*)

apply (*rule nat-mirror-le-conv*)

apply (*simp add: trans-le-add2*)

done

lemma *mirror-elem-less-conv*:

$\llbracket \text{finite } I; a \in I; b \in I \rrbracket \implies$

$(\text{mirror-elem } b \ I < \text{mirror-elem } a \ I) = (a < b)$

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*:

$\llbracket a \leq iMin\ I + Max\ I; b \leq iMin\ I + Max\ I \rrbracket \implies$
 $(mirror\text{-}elem\ a\ I = mirror\text{-}elem\ b\ I) = (a = b)$

by (*simp add: mirror-elem-def nat-mirror-eq-conv*)

lemma *mirror-elem-eq-conv'*:

$\llbracket finite\ I; a \in I; b \in I \rrbracket \implies (mirror\text{-}elem\ a\ I = mirror\text{-}elem\ b\ I) = (a = b)$

apply (*rule mirror-elem-eq-conv*)

apply (*simp-all add: trans-le-add2*)

done

definition

imirror-bounds :: *nat set* \Rightarrow *nat* \Rightarrow *nat* \Rightarrow *nat set*

where

imirror-bounds *I l r* $\equiv (\lambda x. nat\text{-}mirror\ x\ l\ r) \text{ ' } I$

Mirroring all elements between the interval boundaries of I

definition

imirror :: *nat set* \Rightarrow *nat set*

where

imirror *I* $\equiv (\lambda x. iMin\ I + Max\ I - x) \text{ ' } I$

lemma *imirror-eq-nat-mirror-image*:

imirror *I* = $(\lambda x. nat\text{-}mirror\ x\ (iMin\ I)\ (Max\ I)) \text{ ' } I$

unfolding *imirror-def nat-mirror-def* **by** *simp*

lemma *imirror-eq-mirror-elem-image*:

imirror *I* = $(\lambda x. mirror\text{-}elem\ x\ I) \text{ ' } I$

by (*simp add: mirror-elem-def imirror-eq-nat-mirror-image*)

lemma *imirror-eq-imirror-bounds*:

imirror *I* = *imirror-bounds* *I* $(iMin\ I)\ (Max\ I)$

unfolding *imirror-def imirror-bounds-def nat-mirror-def* **by** *simp*

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

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

lemma *imirror-is-empty*: *imirror* *I* = $\{\}$ = (*I* = $\{\}$)

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

lemma *imirror-not-empty*: *I* $\neq \{\}$ $\implies imirror\ I \neq \{\}$

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

lemma *imirror-singleton*: *imirror* $\{a\}$ = $\{a\}$

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

lemma *imirror-finite*: *finite* *I* $\implies finite\ (imirror\ I)$

unfolding *imirror-def* by *simp*

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

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

lemma *imirror-Max*: $\text{finite } I \implies Max\ (imirror\ I) = Max\ I$
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*: $\llbracket \text{finite } I; n \in imirror\ I \rrbracket \implies iMin\ I \leq n \wedge n \leq$
 $Max\ I$
apply (*frule Max-ge[OF imirror-finite, of - n], assumption*)
apply (*fastforce simp: imirror-iMin imirror-Max*)
done

lemma *imirror-bounds-iff*:
 $(n \in imirror\text{-}bounds\ I\ l\ r) = (\exists x \in I. n = l + r - x)$
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*:
 $\llbracket \text{finite } I; Max\ I \leq l + r \rrbracket \implies$
 $imirror\text{-}bounds\ (imirror\text{-}bounds\ I\ l\ r)\ l\ r = I$
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*: $\text{finite } I \implies \text{imirror } (\text{imirror } I) = I$
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*:
 $\text{finite } I \implies \text{mirror-elem } t \ (\text{imirror } I) = \text{mirror-elem } t \ I$
by (*simp add: mirror-elem-def imirror-iMin imirror-Max*)

lemma *imirror-card*: $\text{finite } I \implies \text{card } (\text{imirror } I) = \text{card } I$
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*:
 $\llbracket \text{finite } I; n \leq l + r; \text{Max } I \leq l + r \rrbracket \implies$
 $((\text{nat-mirror } n \ l \ r) \in \text{imirror-bounds } I \ l \ r) = (n \in I)$
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*:
 $\llbracket \text{finite } I; n \leq \text{iMin } I + \text{Max } I \rrbracket \implies ((\text{mirror-elem } n \ I) \in \text{imirror } I) = (n \in I)$
by (*simp add: mirror-elem-def imirror-eq-imirror-bounds imirror-bounds-elem-conv*)

corollary *in-imp-mirror-elem-in*:
 $\llbracket \text{finite } I; n \in I \rrbracket \implies (\text{mirror-elem } n \ I) \in \text{imirror } I$
by (*rule imirror-mem-conv[OF - trans-le-add2[OF Max-ge], THEN iffD2]*)

lemma *imirror-cut-less*:
 $\text{finite } I \implies$
 $\text{imirror } I \downarrow < (\text{mirror-elem } t \ I) =$
 $\text{imirror-bounds } (I \downarrow > t) \ (\text{iMin } I) \ (\text{Max } I)$
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:*

$\llbracket \text{finite } I; t \leq iMin\ I + Max\ I \rrbracket \implies$
 $imirror\ I \downarrow \leq (mirror\text{-}elem\ t\ I) =$
 $imirror\text{-}bounds\ (I \downarrow \geq t)\ (iMin\ I)\ (Max\ I)$
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:*

$finite\ I \implies$
 $imirror\ I \downarrow \geq (mirror\text{-}elem\ t\ I) =$
 $imirror\text{-}bounds\ (I \downarrow \leq t)\ (iMin\ I)\ (Max\ I)$
 $(is\ ?P \implies ?lhs\ I = ?rhs\ I\ t)$
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) $\downarrow \leq$ mirror-elem (mirror-elem t I) (imirror I) I $\downarrow \leq$ t]*)
apply (*simp add: imirror-imirror-ident mirror-elem-imirror mirror-elem-mirror-elem-ident*)
apply (*subgoal-tac mirror-elem t I \leq 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:* $\llbracket \text{finite } I; t \leq iMin\ I + Max\ I \rrbracket \implies$

$imirror\ I \downarrow > (mirror\text{-}elem\ t\ I) =$
 $imirror\text{-}bounds\ (I \downarrow < t)\ (iMin\ I)\ (Max\ I)$
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* \Rightarrow *nat set* \Rightarrow *nat*

where

```
inext n I  $\equiv$  (
  if (n  $\in$  I  $\wedge$  (I  $\downarrow$ > n  $\neq$  {}))
  then iMin (I  $\downarrow$ > n)
  else n)
```

definition *iprev* :: *nat* \Rightarrow *nat set* \Rightarrow *nat*

where

```
iprev n I  $\equiv$  (
  if (n  $\in$  I  $\wedge$  (I  $\downarrow$ < n  $\neq$  {}))
  then Max (I  $\downarrow$ < 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* \notin *I* \Longrightarrow *inext* *n* *I* = *n*

unfolding *inext-def* **by** *simp*

lemma *not-in-iprev-fix*: *n* \notin *I* \Longrightarrow *iprev* *n* *I* = *n*

unfolding *iprev-def* **by** *simp*

lemma *inext-all-le-fix*: $\forall x \in I. x \leq n \Longrightarrow$ *inext* *n* *I* = *n*

unfolding *inext-def* **by** *force*

lemma *iprev-all-ge-fix*: $\forall x \in I. n \leq x \Longrightarrow$ *iprev* *n* *I* = *n*

unfolding *iprev-def* **by** *force*

lemma *inext-Max*: *finite* *I* \Longrightarrow *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*: \llbracket *finite* *I*; *Max* *I* \leq *n* $\rrbracket \Longrightarrow$ *inext* *n* *I* = *n*

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

lemma *iprev-le-iMin*: $n \leq iMin\ I \implies iprev\ n\ I = n$
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*: $n \in I \implies inext\ n\ I \in I$
apply (*clarsimp simp: inext-def*)
apply (*rule subsetD[OF cut-greater-subset]*)
apply (*rule iMinI-ex2, assumption*)
done

lemma *iprev-closed*: $n \in I \implies iprev\ n\ I \in I$
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*: $inext\ n\ I \in I \implies n \in I$
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*: $\llbracket n \in B; A \subseteq B \rrbracket \implies inext\ n\ A \in B$
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*: $\llbracket inext\ n\ A \in B; A \subseteq B \rrbracket \implies n \in B$
apply (*case-tac n $\in A$*)
apply *fastforce*
apply (*simp add: not-in-inext-fix*)
done

lemma *subset-inext-in-iff*: $A \subseteq B \implies (inext\ n\ A \in B) = (n \in B)$
apply (*rule iffI*)
apply (*rule subset-inext-in-imp-in, assumption+*)
apply (*rule subset-inext-closed, assumption+*)
done

lemma *iprev-in-imp-in*: $iprev\ n\ I \in I \implies n \in I$
apply (*case-tac n $\in I$*)

apply (*simp-all add: not-in-iprev-fix*)
done

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

lemma *subset-iprev-closed*: $\llbracket n \in B; A \subseteq B \rrbracket \implies iprev\ n\ A \in B$
apply (*case-tac n \in A*)
apply (*fastforce simp: iprev-closed*)
apply (*simp add: not-in-iprev-fix*)
done

lemma *subset-iprev-in-imp-in*: $\llbracket iprev\ n\ A \in B; A \subseteq B \rrbracket \implies n \in B$
apply (*case-tac n \in A*)
apply *fastforce*
apply (*simp add: not-in-iprev-fix*)
done

lemma *subset-iprev-in-iff*: $A \subseteq B \implies (iprev\ n\ A \in B) = (n \in B)$
apply (*rule iffI*)
apply (*rule subset-iprev-in-imp-in, assumption+*)
apply (*rule subset-iprev-closed, assumption+*)
done

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

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

lemma *inext-mono2*: $\llbracket n \in I; \exists x \in I. n < x \rrbracket \implies n < inext\ n\ I$
by (*fastforce simp add: inext-def i-cut-defs iMin-gr-iff*)

lemma *inext-mono2-infin*: $\llbracket n \in I; infinite\ I \rrbracket \implies n < inext\ n\ I$
apply (*simp add: inext-def i-cut-defs iMin-gr-iff*)
apply (*fastforce simp: infinite-nat-iff-unbounded*)
done

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

lemma *inext-mono2-infin-fin*:
 $\llbracket n \in I; n \neq Max\ I \vee infinite\ I \rrbracket \implies n < inext\ n\ I$
by (*blast intro: inext-mono2-infin inext-mono2-fin*)

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

lemma *Max-le-iMin-imp-singleton*: $\llbracket \text{finite } I; I \neq \{\}; \text{Max } I \leq \text{iMin } I \rrbracket \implies I = \{\text{iMin } I\}$
by (*simp add: iMin-Min-conv Max-le-Min-imp-singleton*)

lemma *inext-neq-iMin-not-singleton*:
 $\llbracket I \neq \{\}; \neg(\exists a. I = \{a\}) \rrbracket \implies \text{inext } n \ I \neq \text{iMin } I$
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*:
 $\llbracket I \neq \{\}; \text{card } I \neq \text{Suc } 0 \rrbracket \implies \text{inext } n \ I \neq \text{iMin } I$
by (*simp add: inext-neq-iMin-not-singleton card-1-singleton-conv*)

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

lemma *inext-less-conv*: $(n \in I \wedge (n < \text{Max } I \vee \text{infinite } I)) = (n < \text{inext } n \ I)$
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*: $\llbracket n < k; k < \text{inext } n \ I \rrbracket \implies k \notin I$
apply (*case-tac* $n \in I$)
prefer 2
apply (*simp add: inext-def*)

apply (*rule* *contrapos-pn*[*of* $k < \text{inext } n \ I \ k \in I$], *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*: $\neg(\exists k \in I. n < k \wedge k < \text{inext } n \ I)$
by (*clarsimp* *simp* *add*: *inext-min-step*)

lemma *min-step-inext*[*rule-format*]:
 $\llbracket x < y; x \in I; y \in I; \bigwedge k. \llbracket x < k; k < y \rrbracket \implies k \notin I \rrbracket \implies$
 $\text{inext } x \ I = y$
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*]:
 $\llbracket x < y; x \in I; y \in I; \neg(\exists k \in I. x < k \wedge k < y) \rrbracket \implies$
 $\text{inext } x \ I = y$
by (*blast* *intro*: *min-step-inext*)
lemma *between-empty-imp-inext-eq*:
 $\llbracket n \in A; n < \text{inext } n \ A; n \in B; \text{inext } n \ A \in B; B \downarrow > n \downarrow < (\text{inext } n \ A) = \{\} \rrbracket$
 \implies
 $\text{inext } n \ B = \text{inext } n \ A$
by (*blast* *intro*: *min-step-inext2*)

lemma *inext-le-mono*: $\llbracket a \leq b; a \in I; b \in I \rrbracket \implies \text{inext } a \ I \leq \text{inext } b \ I$
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*:
 $\llbracket a < b; a \in I; b \in I; \exists x \in I. b < x \rrbracket \implies \text{inext } a \ I < \text{inext } b \ I$
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*:

$\llbracket a < b; a \in I; b \in I; \text{finite } I; b \neq \text{Max } I \rrbracket \implies \text{inext } a \ I < \text{inext } b \ I$
by (*blast intro: inext-less-mono Max-in*)

lemma *inext-less-mono-infin*:

$\llbracket a < b; a \in I; b \in I; \text{infinite } I \rrbracket \implies \text{inext } a \ I < \text{inext } b \ I$
apply (*rule inext-less-mono, assumption+*)
apply (*blast dest: infinite-imp-asc-chain*)
done

lemma *inext-less-mono-infin-fin*:

$\llbracket a < b; a \in I; b \in I; b \neq \text{Max } I \vee \text{infinite } I \rrbracket \implies \text{inext } a \ I < \text{inext } b \ I$
by (*blast intro: inext-less-mono-infin inext-less-mono-fin*)

lemma *inext-le-mono-rev*:

$\llbracket \text{inext } a \ I \leq \text{inext } b \ I; a \in I; b \in I; \exists x \in I. \text{inext } a \ I < x \rrbracket \implies a \leq b$
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*:

$\llbracket \text{inext } a \ I \leq \text{inext } b \ I; a \in I; b \in I; \text{finite } I; \text{inext } a \ I \neq \text{Max } I \rrbracket \implies a \leq b$
by (*metis inext-in-iff inext-le-mono-rev inext-mono2-infin-fin*)

lemma *inext-le-mono-infin-rev*:

$\llbracket \text{inext } a \ I \leq \text{inext } b \ I; a \in I; b \in I; \text{infinite } I \rrbracket \implies a \leq b$
by (*metis inext-in-iff inext-le-mono-rev inext-mono2-infin-fin*)

lemma *inext-le-mono-infin-fin-rev*:

$\llbracket \text{inext } a \ I \leq \text{inext } b \ I; a \in I; b \in I; \text{inext } a \ I \neq \text{Max } I \vee \text{infinite } I \rrbracket \implies a \leq b$
by (*blast intro: inext-le-mono-infin-rev inext-le-mono-fin-rev*)

lemma *inext-less-mono-rev*:

$\llbracket \text{inext } a \ I < \text{inext } b \ I; a \in I; b \in I \rrbracket \implies a < b$
by (*metis inext-le-mono not-le*)

lemma *less-imp-inext-le*: $\llbracket a < b; a \in I; b \in I \rrbracket \implies \text{inext } a \ I \leq b$
by (*metis inext-min-step not-le*)

lemma *iprev-mono*: $\text{iprev } n \ I \leq n$

unfolding *iprev-def i-cut-defs* **by** *simp*

corollary *iprev-neq-imp-greater*: $n \neq \text{iprev } n \ I \implies \text{iprev } n \ I < n$

by (*insert iprev-mono[of n I], simp*)

lemma *iprev-mono2*: $\llbracket n \in I; \exists x \in I. x < n \rrbracket \implies \text{iprev } n \ I < n$

apply (*unfold iprev-def i-cut-defs, clarsimp*)

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

lemma *iprev-mono2-if-neq-iMin*: $\llbracket n \in I; iMin\ I \neq n \rrbracket \implies iprev\ n\ I < n$
by (*blast intro: iMinI iprev-mono2*)

lemma *iprev-neq-Max*: $\llbracket finite\ I; \exists x \in I. x < n \rrbracket \implies iprev\ n\ I \neq Max\ I$
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*:
 $\llbracket finite\ I; I \neq \{\}; \neg(\exists a. I = \{a\}) \rrbracket \implies iprev\ n\ I \neq Max\ I$
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*:
 $\llbracket finite\ I; I \neq \{\}; card\ I \neq Suc\ 0 \rrbracket \implies iprev\ n\ I \neq Max\ I$
apply (*rule iprev-neq-Max-not-singleton, assumption+*)
apply (*simp add: card-1-singleton-conv*)
done

lemma *iprev-neq-imp-iMin*: $iprev\ n\ I \neq n \implies iMin\ I < n$
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*: $\llbracket \text{iprev } n \ I < k; k < n \rrbracket \implies k \notin I$
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*: $\neg(\exists x \in I. \text{iprev } n \ I < x \wedge x < n)$
by (*clarsimp simp add: iprev-min-step*)

lemma *min-step-iprev*:
 $\llbracket x < y; x \in I; y \in I; \bigwedge k. \llbracket x < k; k < y \rrbracket \implies k \notin I \rrbracket \implies$
 $\text{iprev } y \ I = x$
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*]:
 $\llbracket x < y; x \in I; y \in I; \neg(\exists k \in I. x < k \wedge k < y) \rrbracket \implies$
 $\text{iprev } y \ I = x$
by (*blast intro: min-step-iprev*)

lemma *between-empty-imp-iprev-eq*:
 $\llbracket n \in A; \text{iprev } n \ A < n; n \in B; \text{iprev } n \ A \in B; B \downarrow > (\text{iprev } n \ A) \downarrow < n = \{\} \rrbracket$
 \implies
 $\text{iprev } n \ B = \text{iprev } n \ A$
by (*blast intro: min-step-iprev2*)

lemma *iprev-le-mono*: $\llbracket a \leq b; a \in I; b \in I \rrbracket \implies \text{iprev } a \ I \leq \text{iprev } b \ I$
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*:
 $\llbracket a < b; a \in I; b \in I; \exists x \in I. x < a \rrbracket \implies \text{iprev } a \ I < \text{iprev } b \ I$
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*:

$\llbracket a < b; a \in I; b \in I; iMin\ I \neq a \rrbracket \implies iprev\ a\ I < iprev\ b\ I$
by (metis *iprev-in-iff iprev-less-mono iprev-mono2-if-neq-iMin*)

lemma *iprev-le-mono-rev*:

$\llbracket iprev\ a\ I \leq iprev\ b\ I; a \in I; b \in I; iMin\ I \neq iprev\ b\ I \rrbracket \implies a \leq b$
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*:

$\llbracket iprev\ a\ I < iprev\ b\ I; a \in I; b \in I \rrbracket \implies a < b$
apply (rule *ccontr*, simp add: *linorder-not-less*)
by (metis *iprev-le-mono less-le-not-le*)

lemma *set-restriction-inext-eq*:

$\llbracket set-restriction\ interval-fun; n \in interval-fun\ I; inext\ n\ I \in interval-fun\ I \rrbracket \implies$
 $inext\ n\ (interval-fun\ I) = inext\ n\ I$
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 $Max\ (interval-fun\ I) = Max\ I$)
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*:

$\llbracket set-restriction\ interval-fun; n \in interval-fun\ I; inext\ n\ I \in interval-fun\ I \rrbracket \implies$
 $\{inext\ n\ (interval-fun\ I)\} = interval-fun\ \{inext\ n\ I\}$
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*: $iMin\ I \neq n \implies inext\ (iprev\ n\ I)\ I = n$
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*: $infinite\ I \implies iprev\ (inext\ n\ I)\ I = n$
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*:
 $\llbracket finite\ I; n \neq Max\ I \rrbracket \implies iprev\ (inext\ n\ I)\ I = n$
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*:
 $n \neq Max\ I \vee infinite\ I \implies iprev\ (inext\ n\ I)\ I = n$
by (*blast intro: iprev-inext-infin iprev-inext-fin*)

lemma *inext-eq-infin*:
 $\llbracket inext\ a\ I = inext\ b\ I; infinite\ I \rrbracket \implies a = b$
apply (*drule arg-cong*[**where** $f = \lambda x. iprev\ x\ I$])
apply (*simp add: iprev-inext-infin*)
done

lemma *inext-eq-fin*:
 $\llbracket inext\ a\ I = inext\ b\ I; finite\ I; a \neq Max\ I; b \neq Max\ I \rrbracket \implies a = b$
apply (*drule arg-cong*[**where** $f = \lambda x. iprev\ x\ I$])
apply (*simp add: iprev-inext-fin*)
done

lemma *inext-eq-infin-fin*:
 $\llbracket inext\ a\ I = inext\ b\ I; a \neq Max\ I \wedge b \neq Max\ I \vee infinite\ I \rrbracket \implies a = b$
by (*blast intro: inext-eq-fin inext-eq-infin*)**+**

lemma *inext-eq*:
 $\llbracket inext\ a\ I = inext\ b\ I; \exists x \in I. a < x; \exists x \in I. b < x \rrbracket \implies a = b$
by (*metis iprev-inext not-le wellorder-Max-lemma*)

lemma *iprev-eq-if-neq-iMin*:
 $\llbracket iprev\ a\ I = iprev\ b\ I; iMin\ I \neq a; iMin\ I \neq b \rrbracket \implies a = b$
apply (*drule arg-cong*[**where** $f = \lambda x. inext\ x\ I$])
apply (*simp add: inext-iprev*)

done

lemma *iprev-eq*:

$\llbracket \text{iprev } a \ I = \text{iprev } b \ I; \exists x \in I. x < a; \exists x \in I. x < b \rrbracket \implies a = b$
by (*metis iprev-eq-if-neq-iMin not-less-iMin*)

lemma *greater-imp-iprev-ge*: $\llbracket b < a; a \in I; b \in I \rrbracket \implies b \leq \text{iprev } a \ I$

apply (*rule ccontr, simp add: linorder-not-le*)

apply (*blast dest: iprev-min-step*)

done

lemma *inext-cut-less-conv*: $\text{inext } n \ I < t \implies \text{inext } n \ (I \downarrow < t) = \text{inext } n \ I$

apply (*frule le-less-trans[OF inext-mono]*)

apply (*case-tac n \in I*)

apply (*simp add: inext-def*)

apply (*simp add: i-cut-commute-disj[of (\downarrow <) (\downarrow >)] cut-less-mem-iff*)

apply (*case-tac I \downarrow > n \neq \{\}*)

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*: $\text{inext } n \ I \leq t \implies \text{inext } n \ (I \downarrow \leq t) = \text{inext } n \ I$

by (*simp add: nat-cut-le-less-conv inext-cut-less-conv*)

lemma *inext-cut-greater-conv*: $t < n \implies \text{inext } n \ (I \downarrow > t) = \text{inext } n \ I$

apply (*case-tac n \in I*)

apply (*frule cut-greater-mem-iff[THEN iffD2, OF conjI], simp*)

apply (*simp add: inext-def i-cut-commute-disj[of (\downarrow >) (\downarrow >)] cut-cut-greater max-def*)

apply (*simp add: not-in-inext-fix cut-greater-not-in-imp*)

done

lemma *inext-cut-ge-conv*: $t \leq n \implies \text{inext } n \ (I \downarrow \geq t) = \text{inext } n \ I$

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*: $t < \text{iprev } n \ I \implies \text{iprev } n \ (I \downarrow > t) = \text{iprev } n \ I$

apply (*frule less-le-trans[OF iprev-mono]*)

apply (*case-tac n \in I*)

```

apply (simp add: iprev-def)
apply (simp add: i-cut-commute-disj[of ( $\downarrow >$ ) ( $\downarrow <$ )] cut-greater-mem-iff)
apply (case-tac  $I \downarrow < n \neq \{\}$ )
apply simp
apply (metis cut-greater-Max-eq cut-greater-Max-not-empty nat-cut-less-finite)
apply (simp add: i-cut-empty)
apply (simp add: not-in-iprev-fix cut-greater-not-in-imp)
done

```

```

lemma iprev-cut-ge-conv:  $t \leq \text{iprev } n \ I \implies \text{iprev } n \ (I \downarrow \geq t) = \text{iprev } n \ I$ 
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:  $n < t \implies \text{iprev } n \ (I \downarrow < t) = \text{iprev } n \ I$ 
apply (case-tac  $n \in I$ )
apply (frule cut-less-mem-iff[THEN iffD2, OF conjI], simp)
apply (simp add: iprev-def i-cut-commute-disj[of ( $\downarrow <$ ) ( $\downarrow <$ )] cut-cut-less min-def)
apply (simp add: not-in-iprev-fix cut-less-not-in-imp)
done

```

```

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

```

```

lemma inext-cut-le-fix:  $t < \text{inext } n \ I \implies \text{inext } n \ (I \downarrow \leq t) = n$ 
by (simp add: nat-cut-le-less-conv inext-cut-less-fix)

```

```

lemma iprev-cut-greater-fix:  $\text{iprev } n \ I \leq t \implies \text{iprev } n \ (I \downarrow > t) = n$ 

```

```

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  $n$   $I < t \implies$  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 \implies *Com-*
muteWithIntervalCut2 fun

unfolding *CommuteWithIntervalCut2-def* *CommuteWithIntervalCut4-def* **by** blast

lemma *nat-CommuteWithIntervalCut2-4-eq*:

CommuteWithIntervalCut4 (fun::nat set \Rightarrow nat set) = *CommuteWithInterval-*
Cut2 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 (λI. I ↓< t)
and
  cut-le-CommuteWithIntervalCut4: CommuteWithIntervalCut4 (λI. I ↓≤ t)
and
  cut-greater-CommuteWithIntervalCut4: CommuteWithIntervalCut4 (λI. I ↓> t)
and
  cut-ge-CommuteWithIntervalCut4: CommuteWithIntervalCut4 (λI. I ↓≥ 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 ∈ I; strict-mono-on f I ] ⇒ inext (f n) (f ' I) = f (inext n I)
apply (case-tac ∃ x ∈ 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 ∃ x ∈ 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 ∈ I; strict-mono-on f I ] ⇒ iprev (f n) (f ' I) = f (iprev n I)
apply (case-tac ∃ x ∈ 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 ∃ x ∈ 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 \implies *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 \implies *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*:

\llbracket finite I; n \leq iMin I + Max I $\rrbracket \implies$
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'*:

\llbracket finite I; n \in I $\rrbracket \implies$
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*:

\llbracket finite I; n \leq iMin I + Max I $\rrbracket \implies$
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:
   $\llbracket t \in I; t \neq \text{Max } I \vee \text{infinite } I \rrbracket \implies \text{inext } t \ I \neq \text{iMin } I$ 
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:
   $\llbracket t \in I; t \neq \text{Max } I \vee \text{infinite } I \rrbracket \implies \text{iMin } I < \text{inext } t \ I$ 
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:
   $\llbracket t \in I; t \neq \text{Max } I \vee \text{infinite } I \rrbracket \implies I \downarrow \leq t = I \downarrow < (\text{inext } t \ I)$ 
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:
   $\llbracket t \in I; t \neq \text{iMin } I \rrbracket \implies I \downarrow \geq t = I \downarrow > (\text{iprev } t \ I)$ 
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:
   $\llbracket t \in I; t \neq \text{Max } I \vee \text{infinite } I \rrbracket \implies I \downarrow > t = I \downarrow \geq (\text{inext } t \ I)$ 

```



```

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 ]] ⇒

```

$inext\ n\ (A \cup B) = inext\ n\ B$
by (*simp add: inext-append*)
corollary *inext-append-eq3*:
 $\llbracket finite\ A; A \neq \{\}; B \neq \{\}; Max\ A < iMin\ B \rrbracket \implies$
 $inext\ (Max\ A)\ (A \cup B) = iMin\ B$
by (*simp add: inext-append not-less-iMin*)

lemma *iprev-append*: $\llbracket finite\ A; A \neq \{\}; B \neq \{\}; Max\ A < iMin\ B \rrbracket \implies$
 $iprev\ n\ (A \cup B) = (if\ n \in A\ then\ iprev\ n\ A\ else\ (if\ n = iMin\ B\ then\ Max\ A\ else\ iprev\ n\ B))$
apply (*case-tac n \in A \cup 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 \downarrow < 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 \downarrow < n = A*)
prefer 2
apply (*simp add: cut-less-all-iff*)
apply *fastforce*
apply (*subgoal-tac B \downarrow < n \neq \{\}*)
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*:
 $\llbracket finite\ A; A \neq \{\}; B \neq \{\}; Max\ A < iMin\ B; n \in A \rrbracket \implies$
 $iprev\ n\ (A \cup B) = iprev\ n\ A$
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*: $\bigwedge a.$

```

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

```

```

lemma iprev-predicate-change-exists:
   $\llbracket a \leq b; a \in I; b \in I; \neg P b; P a \rrbracket \implies$ 
   $\exists n \in I. a < n \wedge n \leq b \wedge \neg P n \wedge P (iprev\ n\ I)$ 
apply (frule inext-predicate-change-exists[of a b I 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:
   $\llbracket a \leq b; \neg P a; P b \rrbracket \implies \exists n \geq a. n < b \wedge \neg P n \wedge P (Suc\ n)$ 
apply (drule inext-predicate-change-exists[OF - UNIV-I UNIV-I], assumption+)
apply (simp add: inext-UNIV)
done

```

```

corollary nat-pred-predicate-change-exists:
   $\llbracket a \leq b; \neg P b; P a \rrbracket \implies \exists n \leq b. a < n \wedge \neg P n \wedge P (n - Suc\ 0)$ 
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:
   $\llbracket (a::nat) \leq b; a \in I; b \in I; \neg P a; \forall k \in I \downarrow \geq b. P k \rrbracket \implies$ 
   $\exists n \in I. a \leq n \wedge n < b \wedge \neg P n \wedge (\forall k \in I \downarrow > n. P k)$ 
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  $\lambda n. \text{if } (n = a) \text{ then } P\ n \text{ else } (\forall k \in I \downarrow \geq n. P\ k)$ ])
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* $a < \text{inext } n \ I$)
prefer 2
apply (*blast intro: inext-mono order-less-le-trans*)
apply (*subgoal-tac* $I \downarrow \geq \text{inext } n \ I = I \downarrow > n$)
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:*
 $\llbracket (a::\text{nat}) \leq b; a \in I; b \in I; \neg P a; P b \rrbracket \implies$
 $\exists n \in I. a \leq n \wedge n < b \wedge \neg P n \wedge (\forall k \in I. n < k \wedge k \leq b \longrightarrow P k)$
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:*
 $\llbracket (a::\text{nat}) \leq b; \neg P a; \forall k \geq b. P k \rrbracket \implies$
 $\exists n \geq a. n < b \wedge \neg P n \wedge (\forall k > n. P k)$
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:*
 $\llbracket (a::\text{nat}) \leq b; \neg P a; P b \rrbracket \implies$
 $\exists n \geq a. n < b \wedge \neg P n \wedge (\forall k \leq b. n < k \longrightarrow P k)$
apply (*drule inext-predicate-change-exists2[of a b UNIV]*)
apply *simp+*
apply *blast*
done

lemma *iprev-predicate-change-exists2-all:*
 $\llbracket (a::\text{nat}) \leq b; a \in I; b \in I; \neg P b; \forall k \in I \downarrow \leq a. P k \rrbracket \implies$
 $\exists n \in I. a < n \wedge n \leq b \wedge \neg P n \wedge (\forall k \in I \downarrow < n. P k)$
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  $bxI$ )
  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:
   $\llbracket (a::nat) \leq b; a \in I; b \in I; \neg P b; P a \rrbracket \implies$ 
   $\exists n \in I. a < n \wedge n \leq b \wedge \neg P n \wedge (\forall k \in I. a \leq k \wedge k < n \longrightarrow P k)$ 
apply (frule iprev-predicate-change-exists2-all[of  $a$   $b$   $I$   $\downarrow \geq a$ ])
  apply (simp add: i-cut-mem-iff)+
  apply fastforce
apply blast
done

```

```

corollary nat-pred-predicate-change-exists2-all:
   $\llbracket (a::nat) \leq b; \neg P b; \forall k \leq a. P k \rrbracket \implies$ 
   $\exists n > a. n \leq b \wedge \neg P n \wedge (\forall k < n. P k)$ 
apply (drule iprev-predicate-change-exists2-all[rule-format, OF - UNIV-I UNIV-I])
apply (simp add: i-cut-mem-iff Ball-def)+
done

```

```

corollary nat-pred-predicate-change-exists2:
   $\llbracket (a::nat) \leq b; \neg P b; P a \rrbracket \implies$ 
   $\exists n > a. n \leq b \wedge \neg P n \wedge (\forall k \geq a. k < n \longrightarrow P k)$ 
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 \Rightarrow nat \Rightarrow nat ((- \rightarrow -) [100, 100] 60)
where

$I \rightarrow 0 = iMin\ I$
| $I \rightarrow Suc\ n = inext\ (inext-nth\ I\ n)\ I$

lemma *inext-nth-closed*: $I \neq \{\}$ \Longrightarrow $I \rightarrow n \in I$
apply (induct n)
 apply (simp add: *iMinI-ex2*)
 apply (simp add: *inext-closed*)
done

lemma *inext-nth-image*:
[[$I \neq \{\}$; *strict-mono-on* $f\ I$]] \Longrightarrow $(f\ 'I) \rightarrow n = f\ (I \rightarrow n)$
apply (induct n)
 apply (simp add: *iMin-mono-on2* *strict-mono-on-imp-mono-on*)
 apply (simp add: *inext-image* *inext-nth-closed*)
done

lemma *inext-nth-Suc-mono*: $I \rightarrow n \leq I \rightarrow Suc\ n$
by (simp add: *inext-mono*)

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

lemma *inext-nth-Suc-mono2*: $\exists x \in I. I \rightarrow n < x \Longrightarrow I \rightarrow n < I \rightarrow Suc\ n$
apply simp
 apply (rule *inext-mono2*)
 apply (blast intro: *inext-nth-closed* *inext-mono2*)
done

lemma *inext-nth-mono2*: $\exists x \in I. I \rightarrow a < x \Longrightarrow (I \rightarrow a < I \rightarrow b) = (a < b)$
apply (subgoal-tac $I \neq \{\}$)
 prefer 2
 apply blast
 apply (rule *iffI*)
 apply (rule *ccontr*)
 apply (simp add: *linorder-not-less*)


```

apply (drule inext-nth-mono[of - - I])
apply simp
apply clarify
apply (induct b)
apply blast
apply (drule less-Suc-eq[THEN iffD1], erule disjE)
apply (blast intro: order-less-le-trans inext-nth-Suc-mono)
apply (blast intro: inext-nth-Suc-mono2)
done

```

```

lemma inext-nth-mono2-infin:
  infinite I  $\implies$  (I  $\rightarrow$  a < I  $\rightarrow$  b) = (a < b)
apply (drule infinite-nat-iff-unbounded[THEN iffD1])
apply (rule inext-nth-mono2)
apply blast
done

```

```

lemma inext-nth-Max-fix:
   $\llbracket$  finite I; I  $\neq$  {} ; I  $\rightarrow$  a = Max I ; a  $\leq$  b  $\rrbracket \implies$  I  $\rightarrow$  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  $\rightarrow$  n < t  $\implies$  (I  $\downarrow$ < t)  $\rightarrow$  n = I  $\rightarrow$  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  $\implies$ 
  (I - {iMin I})  $\rightarrow$  n = I  $\rightarrow$  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*: $Suc\ 0 < card\ I \implies (I - \{iMin\ I\}) \rightarrow n = I \rightarrow Suc\ n$
by (simp add: remove-Min-inext-nth-Suc-conv)
corollary *remove-Min-inext-nth-Suc-conv-infinite*: $infinite\ I \implies (I - \{iMin\ I\}) \rightarrow n = I \rightarrow Suc\ n$
by (simp add: remove-Min-inext-nth-Suc-conv)

lemma *remove-Max-eq*: $\llbracket finite\ I; I \neq \{\}; n \neq Max\ I \rrbracket \implies Max\ (I - \{n\}) = Max\ I$
by (rule Max-equality, simp+)
lemma *remove-iMin-eq*: $\llbracket I \neq \{\}; n \neq iMin\ I \rrbracket \implies iMin\ (I - \{n\}) = iMin\ I$
by (rule iMin-equality, simp-all add: iMinI-ex2 iMin-le)
lemma *remove-Min-eq*: $\llbracket finite\ I; I \neq \{\}; n \neq Min\ I \rrbracket \implies Min\ (I - \{n\}) = Min\ I$
by (rule Min-eqI, simp+)
lemma *Max-le-iMin-conv-singleton*: $\llbracket finite\ I; I \neq \{\} \rrbracket \implies (Max\ I \leq iMin\ I) = (\exists x. I = \{x\})$
by (simp add: iMin-Min-conv Max-le-Min-conv-singleton del: Max-le-iff Min-ge-iff)

lemma *inext-nth-card-less-Max*:

```

 $\bigwedge I. Suc\ n < card\ I \implies I \rightarrow n < Max\ I$ 
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  $\neg I \subseteq \{iMin I\}$ )
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  $Max I \neq iMin I$ )
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'*:
 $n < card I - Suc 0 \implies I \rightarrow n < Max I$
by (simp add: inext-nth-card-less-Max)

lemma *inext-nth-card-Max-aux*:
 $\bigwedge I. card I = Suc n \implies I \rightarrow n = Max I$
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 $card (I \downarrow < Max I) = Suc n$)
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 $Max (I \downarrow < Max I) < iMin \{Max I\}$)
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 ↓ < Max I and k=k in not-greater-Max, assumption*)
apply (*simp add: cut-less-mem-iff*)
done

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

lemma *inext-nth-card-Max*:
 $\llbracket \text{finite } I; I \neq \{\}; \text{card } I \leq \text{Suc } n \rrbracket \implies I \rightarrow n = \text{Max } I$
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'*:
 $\llbracket \text{finite } I; I \neq \{\}; \text{card } I - \text{Suc } 0 \leq n \rrbracket \implies I \rightarrow n = \text{Max } I$
by (*simp add: inext-nth-card-Max*)

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

lemma *inext-nth-eq-Min-conv*:
 $I \neq \{\} \implies (I \rightarrow n = \text{iMin } I) = (n = 0 \vee (\exists a. I = \{a\}))$
apply (*rule iffI*)
apply (*case-tac n, simp*)
apply (*rename-tac n'*)
apply (*rule ccontr*)
apply (*drule-tac n=I → 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*:
 $I \neq \{\} \implies (\text{iMin } I < I \rightarrow n) = (0 < n \wedge \neg(\exists a. I = \{a\}))$
apply (*rule subst[of iMin I ≠ I → n iMin I < I → 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*:
 $\text{infinite } I \implies (\text{iMin } I < I \rightarrow n) = (0 < n)$
by (*simp add: inext-nth-gr-Min-conv infinite-imp-nonempty infinite-imp-not-singleton*)

lemma *inext-nth-cut-ge-inext-nth*: $\bigwedge I b.$
 $I \neq \{\} \implies I \downarrow_{\geq} (I \rightarrow a) \rightarrow b = I \rightarrow (a + b)$
apply (*induct a*)
apply (*simp add: cut-ge-Min-all*)
apply (*case-tac card I = Suc 0*)
apply (*drule card-1-imp-singleton, clarify*)
apply (*simp add: inext-nth-singleton inext-singleton cut-ge-Min-all*)
apply (*subgoal-tac Suc 0 < card I \vee infinite I*)
prefer 2
apply (*rule ccontr, clarsimp simp: linorder-not-less not-empty-card-gr0-conv*)
apply (*case-tac I - {iMin I} = {}*)
apply (*rule-tac t=I and s={iMin I} in subst, blast*)
apply (*simp (no-asm) add: inext-nth-singleton inext-singleton cut-ge-Min-all*)
apply (*simp add: subset-singleton-conv*)
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 \rightarrow a) I, simp*)
apply (*rule le-neq-trans[OF - not-sym]*)
apply (*simp add: iMin-le inext-closed inext-nth-closed*)
apply (*erule disjE*)
apply (*simp add: inext-neq-iMin-not-card-1 inext-neq-iMin-infin*)
done

lemma *inext-nth-append-eq1*:
 $\llbracket \text{finite } A; A \neq \{\}; \text{Max } A < \text{iMin } B; A \rightarrow n \neq \text{Max } A \rrbracket \implies$
 $(A \cup B) \rightarrow n = A \rightarrow n$
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*:
 $\bigwedge A B. \llbracket \text{Max } A < \text{iMin } B; n < \text{card } A \rrbracket \implies$
 $(A \cup B) \rightarrow n = A \rightarrow n$
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 → Suc n)
prefer 2
apply (subgoal-tac A → Suc n = Max A)
prefer 2
apply (rule inext-nth-card-Max)
apply simp+
apply (rule-tac n=A → n and k=k in inext-min-step, simp+)
apply (rule not-less-iMin)
apply (rule-tac y=Max A in order-less-trans)
apply simp+
done

```

```

lemma inext-nth-card-append-eq3:
  [[ finite A; B ≠ {}; Max A < iMin B ]] ⇒
  (A ∪ B) → (card A) = iMin B
apply (case-tac A = {}, simp)
apply (frule not-empty-card-gr0-conv[THEN iffD1], assumption)
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 ≠ {}; B ≠ {}; Max A < iMin B; card A ≤ n ]] ⇒
  (A ∪ B) → n = B → (n - card A)
apply (rule-tac t=(A ∪ B) → n and s=(A ∪ B) → (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 ≠ {}; B ≠ {}; Max A < iMin B ]] ⇒
  (A ∪ B) → n = (if n < card A then A → n else B → (n - card A))

```

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

lemma *inext-nth-insert-Suc*:

$\llbracket I \neq \{\}; a < iMin\ I \rrbracket \implies (insert\ a\ I) \rightarrow Suc\ n = I \rightarrow n$
 apply (frule not-less-iMin)
 apply (rule-tac $t=I \rightarrow n$ and $s=(insert\ a\ I - \{iMin\ (insert\ a\ I)\}) \rightarrow n$ 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*:

$n < card\ (I \downarrow < t) \implies (I \downarrow < t) \rightarrow n = I \rightarrow n$
 apply (rule-tac $t=I \rightarrow n$ and $s=(I \downarrow < t \cup I \downarrow \geq t) \rightarrow n$ 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*:

$n < card\ (I \downarrow < t) \implies I \rightarrow n < t$
 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*:

$I \downarrow \geq t \neq \{\} \implies (I \rightarrow n < t) = (n < card\ (I \downarrow < t))$
 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 $\lambda x. x \rightarrow n < t - t$])
 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 $\bigwedge x. I \downarrow \geq t \rightarrow x \geq t$)
 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:
   $n < \text{card } I \vee \text{infinite } I \implies \text{card } (I \downarrow < (I \rightarrow n)) = n$ 
apply (case-tac I = {}, simp)
apply (induct n)
apply (simp add: card-eq-0-iff nat-cut-less-finite cut-less-Min-empty)
apply (subgoal-tac  $n < \text{card } I \vee \text{infinite } I$ )
prefer 2
apply fastforce
apply simp
apply (subgoal-tac  $I \rightarrow n \neq \text{Max } I \vee \text{infinite } I$ )
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:
   $\llbracket \text{finite } I; \text{card } I \leq \text{Suc } n \rrbracket \implies \text{card } (I \downarrow < (I \rightarrow n)) = \text{card } I - \text{Suc } 0$ 
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:
   $\text{card } (I \downarrow < (I \rightarrow n)) = ($ 
     $\text{if } (n < \text{card } I \vee \text{infinite } I) \text{ then } n \text{ else } \text{card } I - \text{Suc } 0)$ 
by (simp add: cut-less-inext-nth-card-eq1 cut-less-inext-nth-card-eq2)

```

```

lemma cut-le-inext-nth-card-eq1:
   $n < \text{card } I \vee \text{infinite } I \implies \text{card } (I \downarrow \leq (I \rightarrow n)) = \text{Suc } n$ 
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:
   $\llbracket \text{finite } I; \text{card } I \leq \text{Suc } n \rrbracket \implies \text{card } (I \downarrow \leq (I \rightarrow n)) = \text{card } I$ 
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:
   $\text{card } (I \downarrow \leq (I \rightarrow n)) = ($ 

```


if $(n < \text{card } I \vee \text{infinite } I)$ then $\text{Suc } n$ else $\text{card } I$
 by (simp add: cut-le-inext-nth-card-eq1 cut-le-inext-nth-card-eq2)

primrec $\text{iprev-nth} :: \text{nat set} \Rightarrow \text{nat} \Rightarrow \text{nat} \ ((- \leftarrow -) [100, 100] 60)$
where

$I \leftarrow 0 = \text{Max } I$
 $| I \leftarrow \text{Suc } n = \text{iprev } (\text{iprev-nth } I \ n) \ I$

lemma $\text{iprev-nth-closed}: \llbracket \text{finite } I; I \neq \{\} \rrbracket \Longrightarrow I \leftarrow n \in I$
apply (induct n)
apply simp
apply (simp add: iprev-closed)
done

lemma iprev-nth-image :
 $\llbracket \text{finite } I; I \neq \{\}; \text{strict-mono-on } f \ I \rrbracket \Longrightarrow (f \ ' \ I) \leftarrow n = f \ (I \leftarrow n)$
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 $\text{iprev-nth-Suc-mono}: I \leftarrow (\text{Suc } n) \leq I \leftarrow n$
by (simp add: iprev-mono)

lemma $\text{iprev-nth-mono}: a \leq b \Longrightarrow I \leftarrow b \leq I \leftarrow a$
apply (induct b)
apply simp
apply (drule le-Suc-eq[THEN iffD1], erule disjE)
apply (rule-tac $y = \text{iprev-nth } I \ b$ in order-trans)
apply (rule iprev-nth-Suc-mono)
apply simp
apply simp
done

lemma $\text{iprev-nth-Suc-mono2}$:
 $\llbracket \text{finite } I; \exists x \in I. x < I \leftarrow n \rrbracket \Longrightarrow I \leftarrow (\text{Suc } n) < I \leftarrow n$
apply simp
apply (rule iprev-mono2)
apply (blast intro: iprev-nth-closed)+
done

lemma iprev-nth-mono2 :
 $\llbracket \text{finite } I; \exists x \in I. x < I \leftarrow a \rrbracket \Longrightarrow (I \leftarrow b < I \leftarrow a) = (a < b)$
apply (subgoal-tac $I \neq \{\}$)
prefer 2
apply blast
apply (rule iffI)
apply (rule ccontr)

```

apply (simp add: linorder-not-less)
apply (drule 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:
   $\llbracket I \neq \{\}; I \leftarrow a = iMin\ I; a \leq b \rrbracket \implies I \leftarrow b = iMin\ I$ 
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  $\neq \{\}$ 

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

  have s-not-ex:  $\bigwedge x. \llbracket x \in I; ?P\ x \rrbracket \implies x \in S$ 
  using s-S by blast

  have s-subset:S  $\subseteq I$ 
  using s-S by blast
  have i-not-empty: I  $\neq \{\}$ 
  using as-S-not-empty by blast

  have s-iMin-S: iMin S  $\in S$ 
  using S-not-empty by (simp add: iMinI-ex2)
  hence s-iMin-i: iMin S  $\in I$ 

```

```

using s-subset by blast

show False
proof cases
  assume as:iMin I < iMin S

  obtain prev where s-prev: prev = iprev (iMin S) I by blast
  have s-prev-in: prev ∈ 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 beI)
    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 ( $\lambda n. I \rightarrow n$ ) UNIV I
  apply (simp add: surj-on-conv)
  by (insert inext-nth-surj-aux1 [of I], blast)

```

```

corollary in-imp-ex-inext-nth:  $x \in I \implies \exists n. x = I \rightarrow n$ 
  apply (rule surj-onD [where A=UNIV, simplified])
  apply (rule inext-nth-surj-on)
  apply assumption
  done

```

```

lemma inext-induct:
  [  $P (iMin I); \bigwedge n. [ n \in I; P n ] \implies P (inext n I); n \in I ] \implies P n$ 
  apply (rule-tac f= $\lambda n. I \rightarrow n$  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  $\implies \{ x \in I. \neg(\exists n. I \leftarrow n = x) \} = \{ \}$ 
  apply (case-tac I = { }, blast)
  proof (rule ccontr)
    assume as-finite-i: finite I
    let ?S = {  $x \in I. \neg(\exists n. I \leftarrow n = x)$  }
    assume as-S-not-empty: ?S  $\neq \{ \}$ 

```

```

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

```

```

    have s-not-ex:  $\bigwedge x. [ x \in I; \neg(\exists n. I \leftarrow n = x) ] \implies x \in S$ 
      using s-S by blast

```

```

    have s-subset: S  $\subseteq I$ 
      using s-S by blast
    have i-not-empty: I  $\neq \{ \}$ 
      using as-S-not-empty by blast

```

```

    from as-finite-i
    have S-finite: finite S
      using s-subset by (blast intro: finite-subset)

```

```

    have s-Max-S: Max S  $\in S$ 

```

```

using S-not-empty S-finite by simp
hence s-Max-i:  $\text{Max } S \in I$ 
using s-subset by blast

show False
proof cases
  assume as:  $\text{Max } S < \text{Max } I$ 

  obtain next' where s-next:  $\text{next}' = \text{inext } (\text{Max } S) I$  by blast
  have s-next-in:  $\text{next}' \in I$ 
    by (simp add: s-next inext-closed s-Max-i)

  have s-next-prev-max:  $\text{iprev } \text{next}' I = \text{Max } S$ 
    apply (simp add: s-next)
    apply (rule iprev-inext)
    apply (insert as, simp)
    done

  have s-next-max-1:  $\text{Max } S < \text{next}'$ 
    apply (simp add: s-next)
    apply (rule inext-mono2[of  $\text{Max } S I$ ])
    apply (rule s-Max-i)
    apply (rule-tac x=Max I in beI)
    apply (rule as)
    apply (simp add: as-finite-i i-not-empty)
    done

  hence next-not-in-s:  $\text{next}' \notin S$ 
    using S-finite S-not-empty
    apply clarify
    apply (drule Max-ge[of  $\text{next}'$ ])
    apply simp-all
    done

  have  $\exists n. I \leftarrow n = \text{next}'$ 
    by (insert next-not-in-s s-not-ex[of  $\text{next}'$ ] s-next-in, blast)
  then obtain nNext where s-nNext:  $I \leftarrow n\text{Next} = \text{next}'$  by blast
  hence  $I \leftarrow (\text{Suc } n\text{Next}) = \text{iprev } \text{next}' I$  by simp
  hence  $I \leftarrow (\text{Suc } n\text{Next}) = \text{Max } S$ 
    using s-next-prev-max by simp
  hence  $\exists n. I \leftarrow n = \text{Max } S$  by blast
  hence  $\text{Max } S \notin S$ 
    using s-Max-i s-S by blast
  thus False
    using s-Max-S by blast+
next
assume as:  $\neg(\text{Max } S < \text{Max } I)$ 

have  $\text{Max } S = \text{Max } I$ 
  apply (insert s-subset S-not-empty as-finite-i as)
  apply (drule Max-subset[of  $I$ ])

```

```

    by simp-all
  hence  $\exists n. I \leftarrow n \in S$ 
  apply (rule-tac  $x=0$  in  $exI$ )
  apply (insert  $s\text{-Max}\text{-}S$ )
  apply simp
  done
  thus False
  using  $s\text{-}S$  by blast
qed
qed

```

lemma *iprev-nth-surj-on*: $finite\ I \implies surj\text{-}on\ (\lambda n. I \leftarrow n)\ UNIV\ I$
 apply (simp add: *surj-on-def*)
 by (insert *iprev-nth-surj-aux1*[of I], *blast*)

corollary *in-imp-ex-iprev-nth*:
 $\llbracket finite\ I; x \in I \rrbracket \implies \exists n. x = I \leftarrow n$
 apply (rule *surj-onD*[of $UNIV\ I$, *simplified*])
 apply (rule *iprev-nth-surj-on*)
 apply *assumption*+
 done

lemma *iprev-induct*:
 $\llbracket P\ (Max\ I); \bigwedge n. \llbracket n \in I; P\ n \rrbracket \implies P\ (iprev\ n\ I); finite\ I; n \in I \rrbracket \implies P\ n$
 apply (rule-tac $f=\lambda n. I \leftarrow n$ 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*: $n \leq t \implies inext\ t\ \{n..\} = Suc\ t$
 apply (unfold *inext-def*)
 apply (subgoal-tac $Suc\ t \in \{n..\} \downarrow > t$)
 prefer 2
 apply (simp add: *cut-greater-mem-iff*)
 apply (simp add: *in-imp-not-empty*)
 apply (rule *iMin-equality*, *assumption*)
 apply (simp add: *cut-greater-mem-iff*)
 done

lemma *iprev-atLeast'*: $n \leq t \implies iprev\ (Suc\ t)\ \{n..\} = t$
 apply (rule *subst*[*OF* *inext-atLeast*], *assumption*)
 apply (rule *iprev-inext-infin*[*OF* *infinite-atLeast*])
 done

lemma *iprev-atLeast*: $n < t \implies iprev\ t\ \{n..\} = t - Suc\ 0$
 by (insert *iprev-atLeast'*[of $n\ t - Suc\ 0$], *simp*)

lemma *inext-atMost*: $t < n \implies inext\ t\ \{..n\} = Suc\ t$

```

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:  $a \leq n \implies \{..n\} \rightarrow a = a$ 
apply (insert inext-nth-atLeastAtMost[of a n 0])
apply (simp add: atMost-atLeastAtMost-0-conv)
done
lemma iprev-nth-atMost:  $a \leq n \implies \{..n\} \leftarrow a = n - a$ 
apply (insert iprev-nth-atLeastAtMost[of a n 0])
apply (simp add: atMost-atLeastAtMost-0-conv)
done

lemma inext-nth-lessThan :  $a < n \implies \{..<n\} \rightarrow a = a$ 
apply (case-tac n, simp)
apply (simp add: lessThan-Suc-atMost inext-nth-atMost)
done
lemma iprev-nth-lessThan:  $a < n \implies \{..<n\} \leftarrow a = n - \text{Suc } a$ 
apply (case-tac n, simp)
apply (simp add: lessThan-Suc-atMost iprev-nth-atMost)
done

lemma inext-nth-UNIV:  $\text{UNIV} \rightarrow a = a$ 
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:
   $i\text{Min } I \neq I \leftarrow n \implies \text{inext } (I \leftarrow \text{Suc } n) I = I \leftarrow n$ 
by (simp add: inext-iprev)

lemma inext-iprev-nth-pred:
   $\llbracket \text{finite } I; i\text{Min } I \neq I \leftarrow (n - \text{Suc } 0) \rrbracket \implies$ 
   $\text{inext } (I \leftarrow n) I = I \leftarrow (n - \text{Suc } 0)$ 
apply (case-tac n)
apply (simp add: inext-Max)
apply (simp add: inext-iprev)
done

lemma iprev-inext-nth-Suc:
   $I \rightarrow n \neq \text{Max } I \vee \text{infinite } I \implies \text{iprev } (I \rightarrow \text{Suc } n) I = I \rightarrow n$ 
by (simp add: iprev-inext)
lemma iprev-inext-nth-pred:
   $I \rightarrow (n - \text{Suc } 0) \neq \text{Max } I \vee \text{infinite } I \implies$ 
   $\text{iprev } (I \rightarrow n) I = I \rightarrow (n - \text{Suc } 0)$ 
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*: $Suc\ n < card\ I \implies iMin\ I < I \leftarrow n$

```

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*:

$\llbracket \text{finite } I; I \neq \{\}; \text{card } I \leq \text{Suc } n \rrbracket \implies I \leftarrow n = iMin\ I$
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'*:

$\llbracket \text{finite } I; I \neq \{\}; \text{card } I - \text{Suc } 0 \leq n \rrbracket \implies I \leftarrow n = iMin\ I$
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* \Rightarrow *nat* \Rightarrow *'a list* (**infixl** \downarrow 100)

where *xs* \downarrow *n* \equiv *take n xs*

abbreviation *f-drop'* :: *'a list* \Rightarrow *nat* \Rightarrow *'a list* (**infixl** \uparrow 100)

where *xs* \uparrow *n* \equiv *drop n xs*

lemma *append-eq-Cons*: $[x] @ xs = x \# xs$

by *simp*

lemma *length-Cons*: $\text{length } (x \# xs) = \text{Suc } (\text{length } xs)$

by *simp*

lemma *length-snoc*: $\text{length } (xs @ [x]) = \text{Suc } (\text{length } xs)$

by *simp*

9.1.1 Additional lemmata about list emptiness

lemma *length-greater-imp-not-empty*: $n < \text{length } xs \implies xs \neq []$
by *fastforce*

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

lemma *length-take-le*: $\text{length } (xs \downarrow n) \leq \text{length } xs$
by *simp*

lemma *take-not-empty-conv*: $(xs \downarrow n \neq []) = (0 < n \wedge xs \neq [])$
by *simp*

lemma *drop-not-empty-conv*: $(xs \uparrow n \neq []) = (n < \text{length } xs)$
by *fastforce*

lemma *zip-eq-Nil*: $(\text{zip } xs \text{ } ys = []) = (xs = [] \vee ys = [])$
by (*force simp: length-0-conv[symmetric] min-def simp del: length-0-conv*)

lemma *zip-not-empty-conv*: $(\text{zip } xs \text{ } ys \neq []) = (xs \neq [] \wedge ys \neq [])$
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*:

$\text{Suc } n \leq \text{length } xs \implies (\text{tl } xs) ! n = xs ! \text{Suc } n$

by (*rule hd-Cons-tl[OF length-ge-Suc-imp-not-empty, THEN subst], simp+*)

corollary *nth-tl-eq-nth-Suc2*:

$n < \text{length } xs \implies (\text{tl } xs) ! n = xs ! \text{Suc } n$

by (*simp add: nth-tl-eq-nth-Suc*)

lemma *hd-eq-first*: $xs \neq [] \implies xs ! 0 = \text{hd } xs$

by (*induct xs, simp-all*)

corollary *take-first*: $xs \neq [] \implies xs \downarrow (\text{Suc } 0) = [xs ! 0]$

by (*induct xs, simp-all*)

corollary *take-hd*: $xs \neq [] \implies xs \downarrow (\text{Suc } 0) = [\text{hd } xs]$

by (*simp add: take-first hd-eq-first*)

theorem *last-nth*: $xs \neq [] \implies \text{last } xs = xs ! (\text{length } xs - \text{Suc } 0)$

by (*simp add: last-conv-nth*)

lemma *last-take*: $n < \text{length } xs \implies \text{last } (xs \downarrow \text{Suc } n) = xs ! n$

by (*simp add: last-nth length-greater-imp-not-empty min-eqR*)

corollary *last-take2*:

$[0 < n; n \leq \text{length } xs] \implies \text{last } (xs \downarrow n) = xs ! (n - \text{Suc } 0)$

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

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

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

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

lemma *nth-append1*: $n < \text{length } xs \implies (xs @ ys) ! n = xs ! n$
by (*simp add: nth-append*)

lemma *nth-append2*: $\text{length } xs \leq n \implies (xs @ ys) ! n = ys ! (n - \text{length } xs)$
by (*simp add: nth-append*)

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

lemma *expand-list-eq*:
 $\bigwedge ys. (xs = ys) = (\text{length } xs = \text{length } ys \wedge (\forall i < \text{length } xs. xs ! i = ys ! i))$
by (*rule list-eq-iff-nth-eq*)
lemmas *list-eq-iff = expand-list-eq*

lemma *list-take-drop-imp-eq*:
 $\llbracket xs \downarrow n = ys \downarrow n; xs \uparrow n = ys \uparrow n \rrbracket \implies xs = ys$
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 \Rightarrow nat \Rightarrow 'a list (- [1000,65])
where $x^n \equiv \text{replicate } n \ x$

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

lemma *eq-replicate-conv*: $(\forall i < \text{length } xs. xs ! i = m) = (xs = m^{\text{length } xs})$
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*: $\text{length } (x \# a^n) = \text{Suc } n$
by *simp*

lemma *replicate-pred-Cons-length*: $0 < n \Longrightarrow \text{length } (x \# a^n - \text{Suc } 0) = n$
by *simp*

lemma *replicate-le-diff*: $m \leq n \Longrightarrow x^m @ x^{n-m} = x^n$
by (*simp add: replicate-add[symmetric]*)

lemma *replicate-le-diff2*: $\llbracket k \leq m; m \leq n \rrbracket \Longrightarrow x^{m-k} @ x^{n-m} = x^{n-k}$
by (*subst replicate-add[symmetric], simp*)

lemma *append-constant-length-induct-aux*: $\bigwedge xs.$
 $\llbracket \text{length } xs \text{ div } k = n; \bigwedge ys. k = 0 \vee \text{length } ys < k \Longrightarrow P \ ys;$
 $\bigwedge xs \ ys. \llbracket \text{length } xs = k; P \ ys \rrbracket \Longrightarrow P \ (xs @ ys) \rrbracket \Longrightarrow P \ xs$
apply (*case-tac* $k = 0$, *blast*)

apply *simp*
apply (*induct* n)
apply (*simp add: div-eq-0-conv'*)
apply (*subgoal-tac* $k \leq \text{length } xs$)
prefer 2
apply (*rule div-gr-imp-gr-divisor[of 0], simp*)
apply (*simp only: atomize-all atomize-imp,clarsimp*)
apply (*erule-tac* $x = \text{drop } k \ xs$ **in** *allE*)
apply (*simp add: div-diff-self2*)
apply (*erule-tac* $x = \text{undefined}$ **in** *allE*)
apply (*erule-tac* $x = \text{take } k \ xs$ **in** *allE*)
apply (*simp add: min-eqR*)
apply (*erule-tac* $x = \text{drop } k \ xs$ **in** *allE*)
apply *simp*
done

lemma *append-constant-length-induct*:

[[$\bigwedge ys. k = 0 \vee \text{length } ys < k \implies P \text{ } ys$;
 $\bigwedge xs \text{ } ys. [\text{length } xs = k; P \text{ } ys] \implies P (xs @ ys)$]] $\implies P \text{ } xs$
by (*simp add: append-constant-length-induct-aux[of - - length xs div k]*)

lemma *zip-swap*: $\text{map } (\lambda(y,x). (x,y)) (\text{zip } ys \text{ } xs) = (\text{zip } xs \text{ } ys)$
by (*simp add: expand-list-eq*)

lemma *zip-takeL*: $(\text{zip } xs \text{ } ys) \downarrow n = \text{zip } (xs \downarrow n) \text{ } ys$
by (*simp add: expand-list-eq*)

lemma *zip-takeR*: $(\text{zip } xs \text{ } ys) \downarrow n = \text{zip } xs \text{ } (ys \downarrow n)$
apply (*subst zip-swap[of ys, symmetric]*)
apply (*subst take-map*)
apply (*subst zip-takeL*)
apply (*simp add: zip-swap*)
done

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

lemma *hd-zip*: [[$xs \neq []$; $ys \neq []$]] $\implies \text{hd } (\text{zip } xs \text{ } ys) = (\text{hd } xs, \text{hd } ys)$
by (*simp add: hd-conv-nth zip-not-empty-conv*)

lemma *map-id*: $\text{map } id \text{ } xs = xs$
by (*simp add: id-def*)

lemma *map-id-subst*: $P (\text{map } id \text{ } xs) \implies P \text{ } xs$
by (*subst map-id[symmetric]*)

lemma *map-one*: $\text{map } f [x] = [f \text{ } x]$
by *simp*

lemma *map-last*: $xs \neq [] \implies \text{last } (\text{map } f \text{ } xs) = f (\text{last } xs)$
by (*rule last-map*)

lemma *filter-list-all*: $\text{list-all } P \text{ } xs \implies \text{filter } P \text{ } xs = xs$
by (*induct xs, simp+*)

lemma *filter-snoc*: $\text{filter } P (xs @ [x]) = (\text{if } P \text{ } x \text{ then } (\text{filter } P \text{ } xs) @ [x] \text{ else } \text{filter } P \text{ } xs)$
by (*case-tac P x, simp+*)

lemma *filter-filter-eq*: $\text{list-all } (\lambda x. P \text{ } x = Q \text{ } x) \text{ } xs \implies \text{filter } P \text{ } xs = \text{filter } Q \text{ } xs$
by (*induct xs, simp+*)

lemma *filter-nth*: $\bigwedge n. n < \text{length } (\text{filter } P \text{ } xs) \implies$

```

(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 = λk i. i = 0 ∧ i ≤ k ∧ P ([x] ! i) and
  s = λk i. i = 0 ∧ P x
  in subst)
apply (simp add: fun-eq-iff)
apply fastforce
apply (fastforce simp: Least-def)
apply (rule-tac
  t = λk. card {i. i ≤ k ∧ i < Suc (length xs) ∧ P ((xs @ [x]) ! i)} and
  s = λk. (card {i. i ≤ k ∧ i < length xs ∧ P (xs ! i)} +
    (if k ≥ length xs ∧ P x then Suc 0 else 0))
  in subst)
apply (clarsimp simp: fun-eq-iff split del: 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 = λi. i < length xs ∧ P ((xs @ [x]) ! i) and
  s = λi. i < length xs ∧ P (xs ! i)
  in subst)
apply (rule fun-eq-iff[THEN iffD2])
apply (fastforce simp: nth-append1)
apply (rule add-left-cancel[THEN iffD2])
apply (rule-tac
  t = λi. i = length xs ∧ i ≤ k ∧ P ((xs @ [x]) ! i) and
  s = λi. i = length xs ∧ i ≤ k ∧ P x
  in subst)
apply (rule fun-eq-iff[THEN iffD2])
apply fastforce
apply (case-tac length xs ≤ k)
apply clarsimp
apply (rule-tac
  t = λi. i = length xs ∧ i ≤ k and
  s = λi. i = length xs
  in subst)
apply (rule fun-eq-iff[THEN iffD2])

```

```

apply fastforce
apply simp
apply simp
apply (simp split del: if-split add: less-Suc-eq conj-disj-distribL conj-disj-distribR)
apply (rule-tac
   $t = \lambda k. k < \text{length } xs \wedge$ 
     $n < \text{card } \{i. i \leq k \wedge i < \text{length } xs \wedge P (xs ! i)\} + (\text{if } \text{length } xs \leq k \wedge P$ 
 $x \text{ then } \text{Suc } 0 \text{ else } 0)$  and
   $s = \lambda k. k < \text{length } xs \wedge n < \text{card } \{i. i \leq k \wedge i < \text{length } xs \wedge P (xs ! i)\}$ 
  in subst)
apply (simp add: fun-eq-iff)
apply (rule-tac
   $t = \lambda k. k = \text{length } xs \wedge$ 
     $n < \text{card } \{i. i \leq k \wedge i < \text{length } xs \wedge P (xs ! i)\} + (\text{if } \text{length } xs \leq k \wedge P$ 
 $x \text{ then } \text{Suc } 0 \text{ else } 0)$  and
   $s = \lambda k. k = \text{length } xs \wedge$ 
     $n < \text{card } \{i. i \leq k \wedge i < \text{length } xs \wedge P (xs ! i)\} + (\text{if } P x \text{ then } \text{Suc } 0 \text{ else}$ 
 $0)$ 
  in subst)
apply (simp add: fun-eq-iff)
apply (case-tac n < length (filter P xs))
apply (rule-tac
   $t = (\text{if } P x \text{ then } \text{filter } P xs @ [x] \text{ else } \text{filter } P xs) ! n$  and
   $s = (\text{filter } P xs) ! n$ 
  in subst)
apply (simp add: nth-append1)
apply (simp split del: if-split)
apply (subgoal-tac  $\exists k < \text{length } xs. n < \text{card } \{i. i \leq k \wedge i < \text{length } xs \wedge P (xs !$ 
 $i)\}$ )
prefer 2
apply (rule-tac  $x = \text{length } xs - \text{Suc } 0$  in exI)
apply (simp add: length-filter-conv-card less-eq-le-pred[symmetric])
apply (subgoal-tac  $\exists k \leq \text{length } xs. n < \text{card } \{i. i \leq k \wedge i < \text{length } xs \wedge P (xs !$ 
 $i)\}$ )
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 =  $\lambda k. \text{card } \{i. i < \text{length } xs \wedge P (xs ! i)\} < \text{card } \{i. i \leq k \wedge i < \text{length } xs \wedge P (xs ! i)\}$  and
  s =  $\lambda k. \text{False}$ 
  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  $\{i. i < \text{length } xs \wedge P (xs ! i)\} < \text{Suc } (\text{card } \{i. i \leq k \wedge i < \text{length } xs \wedge P (xs ! i)\})$ ) 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-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:  $\llbracket \text{list-ord ord } xs; \text{ord } x \text{ (hd } xs) \rrbracket \implies \text{list-ord ord } (x \#$ 
xs)
by (induct xs, simp+)
lemma list-ord-append:  $\bigwedge ys.$ 
  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) = ( $\forall n < \text{length } xs - 1. \text{ord } (xs ! n) (xs ! \text{Suc } n)$ )
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:
   $\llbracket \bigwedge x y. \text{ord } x y \implies \text{ord}' x y; \text{list-ord ord } xs \rrbracket \implies$ 
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 ($xs::'a::preorder\ list$) \implies *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 ($xs::'a::preorder\ list$) \implies *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*: $\bigwedge i.$

$\llbracket transP\ ord; list-ord\ ord\ xs; j < length\ xs; i < j \rrbracket \implies$

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 \implies

(*list-ord ord xs*) =

($\forall j < length\ xs. \forall i < j. ord\ (xs\ !\ i)\ (xs\ !\ j)$)

apply (*rule iffI*)

apply (*simp add: list-ord-trans-imp*)

apply (*simp add: list-ord-all-conv*)

done

lemma *list-ord-trans-refl-le*:

$\llbracket transP\ ord; reflP\ ord \rrbracket \implies$

(*list-ord ord xs*) =

($\forall j < length\ xs. \forall i \leq j. ord\ (xs\ !\ i)\ (xs\ !\ j)$)

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*:

$\llbracket transP\ ord; \bigwedge x\ y. ord\ x\ y \implies ord'\ x\ y; reflP\ ord' \rrbracket \implies$

list-ord ord xs \implies

($\forall j < length\ xs. \forall i \leq j. ord'\ (xs\ !\ i)\ (xs\ !\ j)$)

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-asc (xs::'a::preorder list)) =
  (∀ j < length xs. ∀ i < j. xs ! i ≤ xs ! j) and
  list-strict-asc-trans:
  (list-strict-asc (xs::'a::preorder list)) =
  (∀ j < length xs. ∀ i < j. xs ! i < xs ! j) and
  list-desc-trans:
  (list-desc (xs::'a::preorder list)) =
  (∀ j < length xs. ∀ i < j. xs ! j ≤ xs ! i) and
  list-strict-desc-trans:
  (list-strict-desc (xs::'a::preorder list)) =
  (∀ j < length xs. ∀ i < j. xs ! j < xs ! i)
apply (unfold list-asc-def list-strict-asc-def list-desc-def list-strict-desc-def)
apply (rule list-ord-trans, unfold trans-def, blast intro: order-trans order-less-trans)+
done

```

corollary

```

  list-asc-trans-le:
  (list-asc (xs::'a::preorder list)) =
  (∀ j < length xs. ∀ i ≤ j. xs ! i ≤ xs ! j) and
  list-desc-trans-le:
  (list-desc (xs::'a::preorder list)) =
  (∀ j < length xs. ∀ i ≤ j. xs ! j ≤ xs ! i)
apply (unfold list-asc-def list-strict-asc-def list-desc-def list-strict-desc-def)
apply (rule list-ord-trans-refl-le, unfold trans-def, blast intro: order-trans, simp
  add: refl-on-def)+
done

```

corollary

```

  list-strict-asc-trans-le:
  (list-strict-asc (xs::'a::preorder list)) ⇒
  (∀ j < length xs. ∀ i ≤ j. xs ! i ≤ xs ! j)
apply (unfold list-strict-asc-def)
apply (rule list-ord-trans-refl-le-imp[where ord=(≤)])
  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-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 ( $\leq$ ) 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:
   $\llbracket \text{irrefl } \{(a, b). \text{ord } a \ b\}; \text{transP } \text{ord}; \text{list-ord } \text{ord } \text{xs};$ 
   $i < \text{length } \text{xs}; j < \text{length } \text{xs}; i < j \rrbracket \implies$ 
   $\text{xs } ! \ i \neq \text{xs } ! \ j$ 
apply (subgoal-tac  $\bigwedge x \ y. \text{ord } x \ y \implies x \neq y$ )
prefer 2
apply (rule ccontr)
apply (simp add: irrefl-def)
apply (simp add: list-ord-trans)
done

```

```

lemma list-ord-distinct:
   $\llbracket \text{irrefl } \{(a, b). \text{ord } a \ b\}; \text{transP } \text{ord}; \text{list-ord } \text{ord } \text{xs} \rrbracket \implies$ 
  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)  $\implies$  distinct xs
apply (rule-tac ord=( $<$ ) in list-ord-distinct)
apply (unfold irrefl-def list-strict-asc-def trans-def)
apply (blast intro: less-trans)
done

```

```

lemma list-strict-desc-distinct: list-strict-desc (xs::'a::preorder list)  $\implies$  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  $\Rightarrow$  nat list  $\Rightarrow$  '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:

```

```

 $\bigwedge$ 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  $\implies$  length ys = length zs
by (drule arg-cong[where f=length], simp add: sublist-list-length)

```

```

lemma sublist-list-nth:

```

```

 $\bigwedge$ n. n < length ys  $\implies$  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  $\leq$  length xs  $\implies$  xs  $\uparrow$  m  $\downarrow$  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 \Rightarrow nat list \Rightarrow '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*: $\bigwedge xs.$

```

  sublist-list-if xs ys = sublist-list xs (filter ( $\lambda i. i < \text{length } xs$ ) ys)
by (induct ys, simp+)

```

corollary *sublist-list-if-sublist-list-eq*: $\bigwedge xs.$

```

  list-all ( $\lambda i. i < \text{length } xs$ ) ys  $\implies$ 
  sublist-list-if xs ys = sublist-list xs ys
by (simp add: sublist-list-if-sublist-list-filter-conv filter-list-all)

```

corollary *sublist-list-if-sublist-list-eq2*: $\bigwedge xs.$

```

   $\forall n < \text{length } ys. ys ! n < \text{length } xs \implies$ 
  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 ( $\lambda i. i < \text{length } xs$ ) 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*:

$(\text{sublist-list-if } xs \text{ } ys = []) = (\text{list-all } (\lambda i. \text{length } xs \leq i) \text{ } ys)$

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*:

$n < \text{length } ((\text{filter } (\lambda i. i < \text{length } xs) \text{ } ys)) \implies$

$\text{sublist-list-if } xs \text{ } ys ! n = xs ! ((\text{filter } (\lambda i. i < \text{length } xs) \text{ } ys) ! n)$

by (*simp add: sublist-list-if-sublist-list-filter-conv sublist-list-nth*)

lemma *take-drop-eq-sublist-list-if*:

$m + n \leq \text{length } xs \implies xs \uparrow m \downarrow n = \text{sublist-list-if } xs [m..<m+n]$

by (*simp add: sublist-list-if-sublist-list-filter-conv take-drop-eq-sublist-list*)

lemma *nths-empty-conv*: $(\text{nths } xs \text{ } I = []) = (\forall i \in I. \text{length } xs \leq i)$

using [*hypsubst-thin = true*]

by (*fastforce simp: set-empty[symmetric] set-nths linorder-not-le[symmetric]*)

lemma *nths-singleton2*: $\text{nths } xs \text{ } \{y\} = (\text{if } y < \text{length } xs \text{ then } [xs ! y] \text{ else } [])$

apply (*unfold nths-def*)

apply (*induct xs rule: rev-induct, simp*)

apply (*simp add: nth-append*)

done

lemma *nths-take-eq*:

$[[\text{finite } I; \text{Max } I < n]] \implies \text{nths } (xs \downarrow n) \text{ } I = \text{nths } xs \text{ } I$

apply (*case-tac I = {}, simp*)

apply (*case-tac n < length xs*)

prefer 2

apply *simp*

apply (*rule-tac*

$t = \text{nths } xs \text{ } I$ **and**

$s = \text{nths } (xs \downarrow n @ xs \uparrow n) \text{ } I$

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*:

$n \leq iMin \text{ } I \implies \text{nths } (xs \uparrow n) \text{ } \{j. j + n \in I\} = \text{nths } xs \text{ } I$

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 = nthxs xs I and
  s = nthxs (xs ↓ n @ xs ↑ n) I
  in subst)
apply simp
apply (subst nthxs-append)
apply (fastforce simp: nthxs-empty-conv min-eqR)
done

```

```

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

```

```

lemma nthxs-disjoint-Un:
  [ finite A; Max A < iMin B ] ⇒ nthxs xs (A ∪ B) = nthxs xs A @ nthxs xs B
apply (case-tac A = {}, simp)
apply (case-tac B = {}, simp)
apply (case-tac length xs ≤ iMin B)
apply (subst nthxs-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: nthxs-empty-conv iMin-ge-iff)
apply (simp add: linorder-not-le)
apply (rule-tac
  t = nthxs xs (A ∪ B) and
  s = nthxs (xs ↓ (iMin B) @ xs ↑ (iMin B)) (A ∪ B)
  in subst)
apply simp
apply (subst nthxs-append)
apply (simp add: min-eqR)
apply (subst nthxs-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: nthxs-take-eq)
apply (rule-tac
  t = λj. j + iMin B ∈ A ∨ j + iMin B ∈ B and
  s = λj. j + iMin B ∈ B
  in subst)
apply (force simp: fun-eq-iff)
apply (simp add: nthxs-drop-eq)
done

```

```

corollary nthxs-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 *nth_s-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 *nth_s-all*: $\{..$

```

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 = {..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 *nth_s-UNIV*: $nth\ sub\ UNIV = xs$
by (rule *nth_s-all*[OF subset-UNIV])

lemma *sublist-list-nth_s-eq*: $\bigwedge xs.$

```

  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 nth-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[unfolding list-strict-asc-def,*
THEN iffD1, rule-format])
apply (*simp add: nth-append split del: if-split*)+
apply (*simp add: nth-singleton2*)
done

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

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

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

lemma *set-sublist-list-eq-set-sublist*:
list-all ($\lambda i. i < \text{length } xs$) *ys* \implies
set (*sublist-list* *xs ys*) = $\text{set } (\text{nths } xs \ (\text{set } ys))$
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* :: $'a \text{ list} \Rightarrow \text{nat set} \Rightarrow 'a \text{ set}$ (**infixr** f 90)
where $xs \ {}^f \ A \equiv \{y. \exists n \in A. n < \text{length } xs \wedge y = xs \ ! \ n\}$

abbreviation *f-range* :: $'a \text{ list} \Rightarrow 'a \text{ set}$
where $f\text{-range } xs \equiv f\text{-image } xs \ UNIV$

lemma *f-image-eqI*[*simp, intro*]:
 $\llbracket x = xs \ ! \ n; n \in A; n < \text{length } xs \rrbracket \implies x \in xs \ {}^f \ A$
by (*unfold f-image-def, blast*)

lemma *f-imageI*: $\llbracket n \in A; n < \text{length } xs \rrbracket \implies xs \ ! \ n \in xs \ {}^f \ A$
by *blast*

lemma *rev-f-imageI*: $\llbracket n \in A; n < \text{length } xs; x = xs \ ! \ n \rrbracket \implies x \in xs \ {}^f \ A$
by (*rule f-image-eqI*)

lemma *f-imageE*[*elim!*]:
 $\llbracket x \in xs \ {}^f \ A; \bigwedge n. \llbracket x = xs \ ! \ n; n \in A; n < \text{length } xs \rrbracket \implies P \rrbracket \implies P$

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

lemma *f-image-Un*: $xs \text{ 'f } (A \cup B) = xs \text{ 'f } A \cup xs \text{ 'f } B$
by *blast*

lemma *f-image-mono*: $A \subseteq B \implies xs \text{ 'f } A \subseteq xs \text{ 'f } B$
by *blast*

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

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

lemma *subset-f-image-iff*: $(B \subseteq xs \text{ 'f } A) = (\exists A' \subseteq A. B = xs \text{ 'f } A')$
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*:
 $\llbracket \bigwedge n. n \in A \wedge n < \text{length } xs \implies xs ! n \in B \rrbracket \implies xs \text{ 'f } A \subseteq B$
by *blast*

lemma *f-image-empty*: $xs \text{ 'f } \{\} = \{\}$
by *blast*

lemma *f-image-insert-if*:
 $xs \text{ 'f } (\text{insert } n \ A) =$
 $\text{if } n < \text{length } xs \text{ then } \text{insert } (xs ! n) (xs \text{ 'f } A) \text{ else } (xs \text{ 'f } A)$
by (*split if-split, blast*)

lemma *f-image-insert-eq1*:
 $n < \text{length } xs \implies xs \text{ 'f } (\text{insert } n \ A) = \text{insert } (xs ! n) (xs \text{ 'f } A)$
by (*simp add: f-image-insert-if*)

lemma *f-image-insert-eq2*:
 $\text{length } xs \leq n \implies xs \text{ 'f } (\text{insert } n \ A) = (xs \text{ 'f } A)$
by (*simp add: f-image-insert-if*)

lemma *insert-f-image*:
 $\llbracket n \in A; n < \text{length } xs \rrbracket \implies \text{insert } (xs ! n) (xs \text{ 'f } A) = (xs \text{ 'f } A)$
by *blast*

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

lemma *f-image-Collect*: $xs \text{ 'f } \{n. P n\} = \{xs ! n \mid n. P n \wedge n < \text{length } xs\}$
by *blast*

lemma *f-image-eq-set*: $\forall n < \text{length } xs. n \in A \implies xs \text{ 'f } A = \text{set } xs$
by (*fastforce simp: in-set-conv-nth*)

lemma *f-range-eq-set*: $f\text{-range } xs = \text{set } xs$
by (*simp add: f-image-eq-set*)

lemma *f-image-eq-set-nths*: $xs \text{ 'f } A = \text{set } (\text{nths } xs A)$
by (*unfold set-nths, blast*)

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

lemma *f-image-eq-set-sublist-list*:
 $\text{list-all } (\lambda i. i < \text{length } xs) \text{ } ys \implies xs \text{ 'f } (\text{set } ys) = \text{set } (\text{sublist-list } xs \text{ } ys)$
by (*simp add: sublist-list-if-sublist-list-eq f-image-eq-set-sublist-list-if*)

lemma *f-range-eqI*: $\llbracket x = xs ! n; n < \text{length } xs \rrbracket \implies x \in f\text{-range } xs$
by *blast*

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

lemma *f-rangeE[elim?]*:
 $\llbracket x \in f\text{-range } xs; \bigwedge n. \llbracket n < \text{length } xs; x = xs ! n \rrbracket \implies P \rrbracket \implies P$
by *blast*

9.1.6 Mapping lists of functions to lists

primrec *map-list* :: $('a \Rightarrow 'b) \text{ list} \Rightarrow 'a \text{ list} \Rightarrow 'b \text{ list}$
where

$\text{map-list } [] \text{ } xs = []$
 $\mid \text{map-list } (f \# fs) \text{ } xs = f \text{ } (\text{hd } xs) \# \text{map-list } fs \text{ } (\text{tl } xs)$

lemma *map-list-Nil*: $\text{map-list } [] \text{ } xs = []$
by *simp*

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

lemma *map-list-length*: $\bigwedge xs.$
 $\text{length } (\text{map-list } fs \text{ } xs) = \text{length } fs$
by (*induct fs, simp+*)

corollary *map-list-empty-conv*:
 $(\text{map-list } fs \text{ } xs = []) = (fs = [])$
by (*simp del: length-0-conv add: length-0-conv[symmetric] map-list-length*)

corollary *map-list-not-empty-conv*:
 $(\text{map-list } fs \ xs \neq []) = (fs \neq [])$
by (*simp add: map-list-empty-conv*)

lemma *map-list-nth*: $\bigwedge n \ xs.$
 $[[n < \text{length } fs; n < \text{length } xs] \implies$
 $(\text{map-list } fs \ xs \ ! \ n) =$
 $(fs \ ! \ n) \ (xs \ ! \ n)$
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*: $\bigwedge n \ xs.$
 $\text{length } fs \leq n \implies$
 $\text{map-list } fs \ (xs \ \downarrow \ n) =$
 $\text{map-list } fs \ xs$
apply (*induct fs, simp+*)
apply (*rename-tac fs n xs*)
apply (*simp add: tl-take*)
done

lemma *map-list-take*: $\bigwedge n \ xs.$
 $(\text{map-list } fs \ xs) \ \downarrow \ n =$
 $(\text{map-list } (fs \ \downarrow \ n) \ xs)$
apply (*induct fs, simp*)
apply (*case-tac n, simp+*)
done

lemma *map-list-take-take*: $\bigwedge n \ xs.$
 $(\text{map-list } fs \ xs) \ \downarrow \ n =$
 $(\text{map-list } (fs \ \downarrow \ n) \ (xs \ \downarrow \ n))$
by (*simp add: map-list-take map-list-xs-take*)

lemma *map-list-drop*: $\bigwedge n \ xs.$
 $(\text{map-list } fs \ xs) \ \uparrow \ n =$
 $(\text{map-list } (fs \ \uparrow \ n) \ (xs \ \uparrow \ n))$
apply (*induct fs, simp*)
apply (*case-tac n*)
apply (*simp add: drop-Suc*)
done

lemma *map-list-append-append*: $\bigwedge xs1 \ .$
 $\text{length } fs1 = \text{length } xs1 \implies$
 $\text{map-list } (fs1 \ @ \ fs2) \ (xs1 \ @ \ xs2) =$
 $\text{map-list } fs1 \ xs1 \ @$
 $\text{map-list } fs2 \ xs2$
apply (*induct fs1, simp+*)

```

apply (case-tac xs1, simp+)
done
lemma map-list-snoc-snoc:
  length fs = length xs  $\implies$ 
  map-list (fs @ [f]) (xs @ [x]) =
  map-list fs xs @ [f x]
by (simp add: map-list-append-append)
lemma map-list-snoc:  $\wedge$ xs.
  length fs < length xs  $\implies$ 
  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  $\neq$  []  $\implies$ 
  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:  $\wedge$ xs.
  [length fs  $\leq$  length xs; list-all ( $\lambda x. x = f$ ) fs]  $\implies$ 
  map-list fs xs = map f (xs  $\downarrow$  length fs)
apply (induct fs, simp+)
apply (case-tac xs, simp+)
done
lemma map-eq-map-list-take2:
  [length fs = length xs; list-all ( $\lambda x. x = f$ ) fs]  $\implies$ 
  map-list fs xs = map f xs
by (simp add: map-eq-map-list-take)
lemma map-eq-map-list-replicate:
  map-list (flength 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  $\Rightarrow$  'b  $\Rightarrow$  'c)  $\Rightarrow$ 
  — Lists of parameters
  'a list  $\Rightarrow$  'b list  $\Rightarrow$ 

```

'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*: $\bigwedge ys. map2\ f\ xs\ ys = map-list\ (map\ f\ xs)\ ys$
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*: $\bigwedge ys. length\ (map2\ f\ xs\ ys) = length\ xs$
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*:
 $(map2\ f\ xs\ ys \neq []) = (xs \neq [])$
by (*simp add: map2-empty-conv*)

lemma *map2-nth*: $\bigwedge n\ ys.$
 $[n < length\ xs; n < length\ ys] \implies$
 $(map2\ f\ xs\ ys\ !\ n) =$
 $f\ (xs\ !\ n)\ (ys\ !\ n)$
by (*simp add: map2-map-list-conv map-list-nth*)

lemma *map2-ys-take*: $\bigwedge n\ ys.$
 $length\ xs \leq n \implies$
 $map2\ f\ xs\ (ys\ \downarrow\ n) =$
 $map2\ f\ xs\ ys$
by (*simp add: map2-map-list-conv map-list-xs-take*)

lemma *map2-take*: $\bigwedge n\ ys.$
 $(map2\ f\ xs\ ys)\ \downarrow\ n =$
 $(map2\ f\ (xs\ \downarrow\ n)\ ys)$
by (*simp add: map2-map-list-conv take-map map-list-take*)

lemma *map2-take-take*: $\bigwedge n\ ys.$
 $(map2\ f\ xs\ ys)\ \downarrow\ n =$
 $(map2\ f\ (xs\ \downarrow\ n)\ (ys\ \downarrow\ n))$
by (*simp add: map2-take map2-ys-take*)

lemma *map2-drop*: $\bigwedge n\ ys.$
 $(map2\ f\ xs\ ys)\ \uparrow\ n =$

($\text{map2 } f \ (xs \uparrow n) \ (ys \uparrow n)$)
by (*simp add: map2-map-list-conv map-list-drop drop-map*)

lemma *map2-append-append*: $\bigwedge ys1 \ .$
 $\text{length } xs1 = \text{length } ys1 \implies$
 $\text{map2 } f \ (xs1 \ @ \ xs2) \ (ys1 \ @ \ ys2) =$
 $\text{map2 } f \ xs1 \ ys1 \ @$
 $\text{map2 } f \ xs2 \ ys2$
by (*simp add: map2-map-list-conv map-list-append-append*)

lemma *map2-snoc-snoc*:
 $\text{length } xs = \text{length } ys \implies$
 $\text{map2 } f \ (xs \ @ \ [x]) \ (ys \ @ \ [y]) =$
 $\text{map2 } f \ xs \ ys \ @$
 $[f \ x \ y]$
by (*simp add: map2-append-append*)

lemma *map2-snoc*: $\bigwedge ys.$
 $\text{length } xs < \text{length } ys \implies$
 $\text{map2 } f \ (xs \ @ \ [x]) \ ys =$
 $\text{map2 } f \ xs \ ys \ @$
 $[f \ x \ (ys \ ! \ (\text{length } xs))]$
by (*simp add: map2-map-list-conv map-list-snoc*)

lemma *map2-Cons-if*:
 $\text{map2 } f \ xs \ (y \ # \ ys) =$
 $(\text{if } (xs = []) \ \text{then } [] \ \text{else } ($
 $\quad (f \ (\text{hd } xs) \ y) \ # \ \text{map2 } f \ (\text{tl } xs) \ ys))$
by (*case-tac xs, simp+*)

lemma *map2-Cons-not-empty*:
 $xs \neq [] \implies$
 $\text{map2 } f \ xs \ (y \ # \ ys) =$
 $(f \ (\text{hd } xs) \ y) \ # \ \text{map2 } f \ (\text{tl } xs) \ ys$
by (*simp add: map2-Cons-if*)

lemma *map2-append1-take-drop*:
 $\text{length } xs1 \leq \text{length } ys \implies$
 $\text{map2 } f \ (xs1 \ @ \ xs2) \ ys =$
 $\text{map2 } f \ xs1 \ (ys \ \downarrow \ \text{length } xs1) \ @$
 $\text{map2 } f \ xs2 \ (ys \ \uparrow \ \text{length } xs1)$
apply (*rule-tac*
 $t = \text{map2 } f \ (xs1 \ @ \ xs2) \ ys$ **and**
 $s = \text{map2 } f \ (xs1 \ @ \ xs2) \ (ys \ \downarrow \ \text{length } xs1 \ @ \ ys \ \uparrow \ \text{length } xs1)$
in *subst*)
apply *simp*
apply (*simp add: map2-append-append del: append-take-drop-id*)
done

lemma *map2-append2-take-drop*:
 $length\ ys1 \leq length\ xs \implies$
 $map2\ f\ xs\ (ys1\ @\ ys2) =$
 $map2\ f\ (xs\ \downarrow\ length\ ys1)\ ys1\ @$
 $map2\ f\ (xs\ \uparrow\ length\ ys1)\ ys2$
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*:
 $\llbracket xs1 = xs2; ys1 = ys2; length\ xs2 \leq length\ ys2;$
 $\bigwedge x\ y. \llbracket x \in set\ xs2; y \in set\ ys2 \rrbracket \implies f\ x\ y = g\ x\ y \rrbracket \implies$
 $map2\ f\ xs1\ ys1 = map2\ g\ xs2\ ys2$
by (*simp (no-asm-simp) add: expand-list-eq map2-length map2-nth*)

lemma *map2-eq-conv*:
 $length\ xs \leq length\ ys \implies$
 $(map2\ f\ xs\ ys = map2\ g\ xs\ ys) = (\forall i < length\ xs. f\ (xs\ !\ i)\ (ys\ !\ i) = g\ (xs\ !\ i)$
 $(ys\ !\ i))$
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*: $\bigwedge ys.$
 $length\ xs \leq length\ ys \implies$
 $map2\ f\ xs\ ys = map\ (\lambda(x,y). f\ x\ y)\ (zip\ xs\ ys)$
apply (*induct xs, simp*)
apply (*case-tac ys, simp+*)
done

lemma *map2-rev*: $\bigwedge ys.$
 $length\ xs = length\ ys \implies$
 $rev\ (map2\ f\ xs\ ys) = map2\ f\ (rev\ xs)\ (rev\ ys)$
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  $\Rightarrow$  enat
  where icard A  $\equiv$  if finite A then enat (card A) else  $\infty$ 
```

10.2 Results for icard

```
lemma icard-UNIV-nat: icard (UNIV::nat set) =  $\infty$ 
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 =  $\infty$ ) = infinite A
by (case-tac finite A, simp-all add: icard-def)
```

```
corollary icard-finite: finite A  $\Longrightarrow$  icard A = enat (card A)
by (rule icard-finite-conv[THEN iffD2])
```

```
corollary icard-infinite[simp]: infinite A  $\Longrightarrow$  icard A =  $\infty$ 
by (rule icard-infinite-conv[THEN iffD2])
```

```
lemma icard-eq-enat-imp: icard A = enat n  $\Longrightarrow$  finite A
by (case-tac finite A, simp-all)
```

```
lemma icard-eq-Infty-imp: icard A =  $\infty$   $\Longrightarrow$  infinite A
by (rule icard-infinite-conv[THEN iffD1])
```

```
lemma icard-the-enat: finite A  $\Longrightarrow$  the-enat (icard A) = card A
by (simp add: icard-def)
```

```
lemma icard-eq-enat-imp-card: icard A = enat n  $\Longrightarrow$  card A = n
by (frule icard-eq-enat-imp, simp add: icard-finite)
```

```
lemma icard-eq-enat-card-conv: 0 < n  $\Longrightarrow$  (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:  $(0 < \text{icard } A) = (A \neq \{\})$ 
by (simp add: icard-empty-iff[symmetric])
lemmas icard-not-empty-iff-enat = icard-not-empty-iff[unfolded zero-enat-def]

lemma icard-singleton:  $\text{icard } \{a\} = \text{eSuc } 0$ 
by (simp add: icard-finite eSuc-enat)
lemmas icard-singleton-enat[simp] = icard-singleton[unfolded zero-enat-def]
lemma icard-1-imp-singleton:  $\text{icard } A = \text{eSuc } 0 \implies \exists a. A = \{a\}$ 
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:  $(\text{icard } A = \text{eSuc } 0) = (\exists a. A = \{a\})$ 
apply (rule iffI)
  apply (simp add: icard-1-imp-singleton)
apply fastforce
done

lemma icard-insert-disjoint:  $x \notin A \implies \text{icard } (\text{insert } x A) = \text{eSuc } (\text{icard } A)$ 
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:  $\text{icard } (\text{insert } x A) = (\text{if } x \in A \text{ then } \text{icard } A \text{ else } \text{eSuc } (\text{icard } A))$ 
apply (case-tac  $x \in A$ )
  apply (simp add: insert-absorb)
apply (simp add: icard-insert-disjoint)
done

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

lemma icard-Suc-Diff1:  $x \in A \implies \text{eSuc } (\text{icard } (A - \{x\})) = \text{icard } A$ 
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:  $x \in A \implies \text{icard } (A - \{x\}) = \text{icard } A - 1$ 

```

```

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*: $\exists (A::\text{nat set})x. x \in A \wedge \neg \text{icard } (A - \{x\}) < \text{icard } A$
by (*rule-tac* $x=\{0..\}$ **in** *exI*, *simp*)

lemma *not-icard-Diff2-less*: $\exists (A::\text{nat set})x y. x \in A \wedge y \in A \wedge \neg \text{icard } (A - \{x\} - \{y\}) < \text{icard } A$
by (*rule-tac* $x=\{0..\}$ **in** *exI*, *simp*)

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

lemma *icard-psubset*: $\llbracket A \subseteq B; \text{icard } A < \text{icard } B \rrbracket \implies A \subset B$
by (*metis* *less-le* *psubset-eq*)

lemma *icard-partition*:

$\llbracket \bigwedge c. c \in C \implies \text{icard } c = k; \bigwedge c1 c2. \llbracket c1 \in C; c2 \in C; c1 \neq c2 \rrbracket \implies c1 \cap c2 = \{\} \rrbracket \implies$

$\text{icard } (\bigcup C) = k * \text{icard } C$

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* $\bigwedge c. c \in C \implies \text{card } c = k1$)

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*: $\text{icard } (f \text{ ' } A) \leq \text{icard } A$
apply (*case-tac finite A*)
apply (*simp add: icard-finite card-image-le*)
apply *simp*
done

lemma *icard-image*: $\text{inj-on } f A \implies \text{icard } (f \text{ ' } A) = \text{icard } A$
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*: $\exists (f::\text{nat} \Rightarrow \text{nat}) (A::\text{nat set}). \text{icard } (f \text{ ' } A) = \text{icard } A \wedge \neg \text{inj-on } f A$
apply (*rule-tac x= $\lambda n. (if n = 0 then Suc 0 else n)$ in exI*)
apply (*rule-tac x={0..} in exI*)
apply (*rule conjI*)
apply (*rule subst[of {1..}] (($\lambda n. if n = 0 then Suc 0 else n$) ' {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*: $\exists (f::\text{nat} \Rightarrow \text{nat}) (A::\text{nat set}). \neg (\text{inj-on } f A = (\text{icard } (f \text{ ' } A) = \text{icard } A))$
by (*insert not-eq-icard-imp-inj-on, blast*)

lemma *icard-inj-on-le*: $\llbracket \text{inj-on } f A; f \text{ ' } A \subseteq B \rrbracket \implies \text{icard } A \leq \text{icard } B$
apply (*case-tac finite B*)
apply (*metis icard-image icard-mono*)
apply *simp*
done

lemma *icard-bij-eq*:
 $\llbracket \text{inj-on } f A; f \text{ ' } A \subseteq B; \text{inj-on } g B; g \text{ ' } B \subseteq A \rrbracket \implies \text{icard } A = \text{icard } B$
by (*simp add: order-eq-iff icard-inj-on-le*)

lemma *icard-cartesian-product*: $\text{icard } (A \times B) = \text{icard } A * \text{icard } B$
apply (*case-tac A = {} \vee B = {}, fastforce*)
apply *clarsimp*
apply (*case-tac finite A \wedge finite B*)
apply (*simp add: icard-finite*)
apply (*simp only: de-Morgan-conj, erule disjE*)

apply (*simp-all add:*
icard-not-empty-iff[symmetric]
cartesian-product-infiniteL-imp-infinite cartesian-product-infiniteR-imp-infinite)
done

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

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

lemma
icard-lessThan: $\text{icard } \{..\} = \text{enat } u$ **and**
icard-atMost: $\text{icard } \{..u\} = \text{enat } (\text{Suc } u)$ **and**
icard-atLeastLessThan: $\text{icard } \{l..\} = \text{enat } (u - l)$ **and**
icard-atLeastAtMost: $\text{icard } \{l..u\} = \text{enat } (\text{Suc } u - l)$ **and**
icard-greaterThanAtMost: $\text{icard } \{l<..u\} = \text{enat } (u - l)$ **and**
icard-greaterThanLessThan: $\text{icard } \{l<..\} = \text{enat } (u - \text{Suc } l)$
by (*simp-all add: icard-finite*)

lemma *icard-atLeast*: $\text{icard } \{(u::\text{nat})..\} = \infty$
by (*simp add: infinite-atLeast*)

lemma *icard-greaterThan*: $\text{icard } \{(u::\text{nat})<..\} = \infty$
by (*simp add: infinite-greaterThan*)

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

lemma *icard-atLeast-int*: $\text{icard } \{(u::\text{int})..\} = \infty$
by (*simp add: infinite-atLeast-int*)

lemma *icard-greaterThan-int*: $\text{icard } \{(u::\text{int})<..\} = \infty$
by (*simp add: infinite-greaterThan-int*)

lemma *icard-atMost-int*: $\text{icard } \{..(u::\text{int})\} = \infty$
by (*simp add: infinite-atMost-int*)

lemma *icard-lessThan-int*: $\text{icard } \{..
by (*simp add: infinite-lessThan-int*)$

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 $\text{nat} \Rightarrow 'a$ as infinite lists over elements of $'a$. Mapping functions to intervals lists $[m..<n]$ yields 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  $xs \frown f \equiv \lambda n. \text{if } n < \text{length } xs \text{ then } xs ! n \text{ else } f (n - \text{length } xs)$ 
```

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:  $(xs \frown f) n = (\text{if } n < \text{length } xs \text{ then } xs ! n \text{ else } f (n - \text{length } xs))$ 
```

```
by (simp add: i-append-def)
```

```
lemma i-append-nth1[simp]:  $n < \text{length } xs \implies (xs \frown f) n = xs ! n$ 
```

```
by (simp add: i-append-def)
```

```
lemma i-append-nth2[simp]:  $\text{length } xs \leq n \implies (xs \frown f) n = f (n - \text{length } xs)$ 
```

```
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]:
```

```
   $\text{length } xs = \text{length } ys \implies$ 
```

```
   $(xs \frown f = ys \frown g) = (xs = ys \wedge f = g)$ 
```

```
apply (rule iffI)
```

```
prefer 2
```

```

apply simp
apply (simp add: expand-ilst-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:
   $\llbracket xs \frown f = ys \frown g; \text{length } xs \leq \text{length } ys \rrbracket \implies$ 
   $\exists zs. xs @ zs = ys \wedge f = zs \frown g$ 
apply (simp add: expand-ilst-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:
   $(xs \frown f = ys \frown g) =$ 
   $(\exists zs. xs = ys @ zs \wedge zs \frown f = g \vee xs @ zs = ys \wedge f = zs \frown g)$ 
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 \frown 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]:  $(xs \frown f = xs \frown g) = (f = g)$ 
apply (rule iffI)
apply (clarsimp simp: expand-ilst-eq, rename-tac i)
apply (erule-tac x=length xs + i in allE)
apply simp
apply simp
done

```

lemma *NOT-i-append-same-eq*:
 $\neg(\forall xs\ ys\ f. (xs \smallfrown (f::(nat \Rightarrow nat)) = ys \smallfrown f) = (xs = ys))$
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-ilst-eq i-append-nth*)
done

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

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

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

lemma *i-append-eq-i-appendI*:
 $\llbracket xs\ @\ xs' = ys; f = xs' \smallfrown g \rrbracket \implies xs \smallfrown f = ys \smallfrown g$
by *simp*

lemma *o-ext*:
 $(\forall x. (x \in range\ h \implies f\ x = g\ x)) \implies f \circ h = g \circ h$
by (*simp* *add: expand-ilst-eq*)

lemma *i-append-o[simp]*: $g \circ (xs \smallfrown f) = (map\ g\ xs) \smallfrown (g \circ f)$
by (*simp* *add: expand-ilst-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-ilst-eq*)

lemma *o-cong*:
 $\llbracket h = i; \bigwedge x. x \in range\ i \implies f\ x = g\ x \rrbracket \implies f \circ h = f \circ i$
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-ilst-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*:
 $\llbracket f \circ g = f \circ h; inj\ on\ f\ (range\ g \cup range\ h) \rrbracket \implies g = h$

```

apply (rule expand-ilst-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  $\cup$  range h)  $\implies$ 
  (f  $\circ$  g = f  $\circ$  h) = (g = h)
apply (rule iffI)
apply (rule o-inj-on, assumption+)
apply simp
done

```

```

lemma o-injective:  $\llbracket$  f  $\circ$  g = f  $\circ$  h; inj f  $\rrbracket \implies$  g = h
by (simp add: expand-ilst-eq inj-on-def)

```

```

lemma inj-o-eq-o: inj f  $\implies$  (f  $\circ$  g = f  $\circ$  h) = (g = h)
apply (rule iffI)
apply (rule o-injective, assumption+)
apply simp
done

```

```

lemma inj-oI: inj f  $\implies$  inj ( $\lambda$ g. f  $\circ$  g)
apply (simp add: inj-on-def)
apply (blast intro: o-inj-on[unfolded inj-on-def])
done

```

```

lemma inj-oD: inj ( $\lambda$ g. f  $\circ$  g)  $\implies$  inj f
apply (clarsimp simp add: inj-on-def, rename-tac g h)
apply (erule-tac x= $\lambda$ n. g in allE)
apply (erule-tac x= $\lambda$ n. h in allE)
apply (simp add: expand-ilst-eq)
done

```

```

lemma inj-o[iff]: inj ( $\lambda$ g. f  $\circ$  g) = inj f
apply (rule iffI)
apply (rule inj-oD, assumption)
apply (rule inj-oI, assumption)
done

```

```

lemma inj-on-oI:
  inj-on f ( $\bigcup$  (( $\lambda$ f. range f) ' A))  $\implies$  inj-on ( $\lambda$ g. f  $\circ$  g) A
apply (rule inj-onI)
apply (rule o-inj-on, assumption)
apply (unfold inj-on-def)
apply force
done

```

lemma *o-idI*: $\forall x. x \in \text{range } g \longrightarrow f x = x \implies f \circ g = g$
by (*simp add: expand-ilst-eq*)

lemma *o-fun-upd[simp]*: $y \notin \text{range } g \implies f (y := x) \circ g = f \circ g$
by (*fastforce simp: expand-ilst-eq*)

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

lemma *set-subset-i-append*: $\text{set } xs \subseteq \text{range } (xs \smallfrown f)$
by *simp*

lemma *range-subset-i-append*: $\text{range } f \subseteq \text{range } (xs \smallfrown f)$
by *simp*

lemma *range-ConsD*: $y \in \text{range } ([x] \smallfrown f) \implies y = x \vee y \in \text{range } f$
by *simp*

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

lemma *in-range-conv-decomp*:
 $(x \in \text{range } f) = (\exists xs g. f = xs \smallfrown ([x] \smallfrown g))$
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=λi. f (i + Suc n) in exI*)
apply (*simp add: expand-ilst-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]*: $((x \# xs) \smallfrown f) 0 = x$
by *simp*

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

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

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

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

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

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

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

lemma *all-nth-imp-all-range*: $\llbracket \forall x. P (f x); y \in \text{range } f \rrbracket \implies P y$
by *blast*

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

lemma *i-append-update1*:
 $n < \text{length } xs \implies (xs \smallfrown f) (n := x) = xs[n := x] \smallfrown f$
by (*simp add: expand-ilst-eq i-append-nth*)

lemma *i-append-update2*:
 $\text{length } xs \leq n \implies (xs \smallfrown f) (n := x) = xs \smallfrown (f(n - \text{length } xs := x))$
by (*fastforce simp: expand-ilst-eq i-append-nth*)

lemma *i-append-update*:
 $(xs \smallfrown f) (n := x) =$
 $(\text{if } n < \text{length } xs \text{ then } xs[n := x] \smallfrown f$
 $\text{else } xs \smallfrown (f(n - \text{length } xs := x)))$
by (*simp add: i-append-update1 i-append-update2*)

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

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

lemma *range-update-subsetI*:
 $\llbracket \text{range } f \subseteq A; x \in A \rrbracket \implies \text{range } (f(n := x)) \subseteq A$
by *fastforce*

lemma *range-update-memI*: $x \in \text{range } (f(n := x))$
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. $i\text{-take } f \ n = [f \ 0, f \ 1, \dots, f \ (n-1)]$. The *i-drop* operator drops the first n elements of an infinite list, i.e. $(i\text{-take } f \ n) \ 0 = f \ n$, $(i\text{-take } f \ n) \ 1 = f \ (n + 1)$, \dots

definition $i\text{-take} :: \text{nat} \Rightarrow 'a \ \text{ilist} \Rightarrow 'a \ \text{list}$
where $i\text{-take } n \ f \equiv \text{map } f \ [0..<n]$

definition $i\text{-drop} :: \text{nat} \Rightarrow 'a \ \text{ilist} \Rightarrow 'a \ \text{ilist}$
where $i\text{-drop } n \ f \equiv (\lambda x. f \ (n + x))$

abbreviation $i\text{-take}' :: 'a \ \text{ilist} \Rightarrow \text{nat} \Rightarrow 'a \ \text{list}$ (**infixl** \Downarrow 100)
where $f \ \Downarrow \ n \equiv i\text{-take } n \ f$

abbreviation $i\text{-drop}' :: 'a \ \text{ilist} \Rightarrow \text{nat} \Rightarrow 'a \ \text{ilist}$ (**infixl** \Uparrow 100)
where $f \ \Uparrow \ n \equiv i\text{-drop } n \ f$

lemma $f \ \Downarrow \ n = \text{map } f \ [0..<n]$

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

lemma $f \ \Uparrow \ n = (\lambda x. f \ (n + x))$

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

Basic results for *i-take* and *i-drop*

lemma $i\text{-take-first}: f \ \Downarrow \ \text{Suc } 0 = [f \ 0]$

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

lemma $i\text{-drop-i-take-1}: f \ \Uparrow \ n \ \Downarrow \ \text{Suc } 0 = [f \ n]$

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

lemma $i\text{-take-take-eq1}: m \leq n \Longrightarrow (f \ \Downarrow \ n) \ \Downarrow \ m = f \ \Downarrow \ m$

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

lemma $i\text{-take-take-eq2}: n \leq m \Longrightarrow (f \ \Downarrow \ n) \ \Downarrow \ m = f \ \Downarrow \ n$

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

lemma $i\text{-take-take}[simp]: (f \ \Downarrow \ n) \ \Downarrow \ m = f \ \Downarrow \ \text{min } n \ m$

by (*simp add: min-def i-take-take-eq1 i-take-take-eq2*)

lemma $i\text{-drop-nth}[simp]: (s \ \Uparrow \ n) \ x = s \ (n + x)$

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

lemma $i\text{-drop-nth-sub}: n \leq x \Longrightarrow (s \ \Uparrow \ n) \ (x - n) = s \ x$

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

theorem $i\text{-take-nth}[simp]: i < n \Longrightarrow (f \ \Downarrow \ n) \ ! \ i = f \ i$

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

lemma $i\text{-take-length}[simp]: \text{length } (f \ \Downarrow \ n) = n$

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

lemma $i\text{-take-0}[simp]: f \ \Downarrow \ 0 = []$

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*: $\text{last } (f \Downarrow \text{Suc } n) = f \ n$

by (*simp add: last-nth*)

lemma *last-i-take2*: $0 < n \implies \text{last } (f \Downarrow n) = f \ (n - \text{Suc } 0)$

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*]: $(\lambda n. x) \Downarrow i = \text{replicate } i \ x$

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

lemma *i-drop-const*[*simp*]: $(\lambda n. x) \uparrow i = (\lambda n. x)$

by (*simp add: expand-ilst-eq*)

lemma *i-append-i-take-eq1*:

$n \leq \text{length } xs \implies (xs \frown f) \Downarrow n = xs \Downarrow n$

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

lemma *i-append-i-take-eq2*:

$\text{length } xs \leq n \implies (xs \frown f) \Downarrow n = xs \ @ (f \Downarrow (n - \text{length } xs))$

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

lemma *i-append-i-take-if*:

$(xs \frown f) \Downarrow n = (\text{if } n \leq \text{length } xs \text{ then } xs \Downarrow n \text{ else } xs \ @ (f \Downarrow (n - \text{length } xs)))$

by (*simp add: i-append-i-take-eq1 i-append-i-take-eq2*)

lemma *i-append-i-take*[*simp*]:

$(xs \frown f) \Downarrow n = (xs \Downarrow n) \ @ (f \Downarrow (n - \text{length } xs))$

by (*simp add: i-append-i-take-if*)

lemma *i-append-i-drop-eq1*:

$n \leq \text{length } xs \implies (xs \frown f) \uparrow n = (xs \uparrow n) \frown f$

by (*simp add: expand-ilst-eq i-append-nth less-diff-conv add.commute[of - n]*)

lemma *i-append-i-drop-eq2*:

$\text{length } xs \leq n \implies (xs \frown f) \uparrow n = f \uparrow (n - \text{length } xs)$

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

lemma *i-append-i-drop-if*:

$(xs \smallfrown f) \uparrow n = (\text{if } n < \text{length } xs \text{ then } (xs \uparrow n) \smallfrown f \text{ else } f \uparrow (n - \text{length } xs))$
by (*simp add: i-append-i-drop-eq1 i-append-i-drop-eq2*)

lemma *i-append-i-drop[simp]*: $(xs \smallfrown f) \uparrow n = (xs \uparrow n) \smallfrown (f \uparrow (n - \text{length } xs))$
by (*simp add: i-append-i-drop-if*)

lemma *i-append-i-take-i-drop-id[simp]*: $(f \Downarrow n) \smallfrown (f \uparrow n) = f$
by (*simp add: expand-ilst-eq i-append-nth*)

lemma *ilst-i-take-i-drop-imp-eq*:

$\llbracket f \Downarrow n = g \Downarrow n; f \uparrow n = g \uparrow n \rrbracket \implies f = g$
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 *ilst-i-take-i-drop-eq-conv*:

$(f = g) = (\exists n. (f \Downarrow n = g \Downarrow n \wedge f \uparrow n = g \uparrow n))$
apply (*rule iffI, simp*)
apply (*blast intro: ilist-i-take-i-drop-imp-eq*)
done

lemma *ilst-i-take-eq-conv*: $(f = g) = (\forall n. f \Downarrow n = g \Downarrow n)$

apply (*rule iffI, simp*)
apply (*clarsimp simp: expand-ilst-eq, rename-tac i*)
apply (*drule-tac x=Suc i in spec*)
apply (*drule-tac f= $\lambda xs. xs ! i$ in arg-cong*)
apply *simp*
done

lemma *ilst-i-drop-eq-conv*: $(f = g) = (\forall n. f \uparrow n = g \uparrow n)$

apply (*rule iffI, simp*)
apply (*drule-tac x=0 in spec*)
apply *simp*
done

lemma *i-take-the-conv*:

$f \Downarrow k = (\text{THE } xs. \text{length } xs = k \wedge (\exists g. xs \smallfrown g = f))$
apply (*rule the1I2*)
apply (*rule-tac a=f \Downarrow k in ex1I*)
apply (*fastforce intro: i-append-i-take-i-drop-id*)
done

lemma *i-drop-the-conv*:

$f \uparrow k = (\text{THE } g. (\exists xs. \text{length } xs = k \wedge xs \smallfrown g = f))$
apply (*rule sym, rule the1-equality*)
apply (*rule-tac a=f \uparrow k in ex1I*)

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

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

corollary *i-take-Suc-Cons*: $([x] \frown f) \Downarrow Suc\ n = x \# (f \Downarrow n)$
by *simp*

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

corollary *i-drop-Suc-Cons*: $([x] \frown f) \Uparrow Suc\ n = f \Uparrow n$
by *simp*

lemma *i-take-Suc*: $f \Downarrow Suc\ n = f\ 0 \# (f \Uparrow Suc\ 0 \Downarrow n)$
by (*simp add: expand-list-eq nth-Cons'*)

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

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

corollary *i-drop-Suc*: $f \Uparrow Suc\ 0 \Uparrow n = f \Uparrow Suc\ n$
by *simp*

lemma *i-take-commute*: $s \Downarrow a \Downarrow b = s \Downarrow b \Downarrow a$
by (*simp add: ac-simps*)

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

corollary *i-drop-tl*: $f \Uparrow Suc\ 0 \Uparrow n = f \Uparrow n \Uparrow Suc\ 0$
by *simp*

lemma *nth-via-i-drop*: $(f \Uparrow n)\ 0 = x \implies f\ n = x$
by *simp*

lemma *i-drop-Suc-conv-tl*: $[f\ n] \frown (f \Uparrow Suc\ n) = f \Uparrow n$
by (*simp add: expand-ilst-eq i-append-nth*)

lemma *i-drop-Suc-conv-tl'*: $([f\ n] \frown f) \Uparrow Suc\ n = f \Uparrow n$
by (*simp add: i-drop-Suc-Cons*)

lemma *i-take-i-drop*: $f \Uparrow m \Downarrow n = f \Downarrow (n + m) \Uparrow m$

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

Appending an interval of a function

lemma *i-take-int-append*:

$$m \leq n \implies (f \Downarrow m) @ \text{map } f [m..<n] = f \Downarrow n$$

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*: $f \Downarrow n \uparrow m = \text{map } f [m..<n]$

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

corollary *i-take-drop-append[simp]*:

$$m \leq n \implies (f \Downarrow m) @ (f \Downarrow n \uparrow m) = f \Downarrow n$$

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]*: $(f \circ g) \Downarrow n = \text{map } f (g \Downarrow n)$

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-ilst-eq*)

lemma *set-i-take-subset*: $\text{set } (f \Downarrow n) \subseteq \text{range } f$

by (*fastforce simp: in-set-conv-nth*)

lemma *range-i-drop-subset*: $\text{range } (f \uparrow n) \subseteq \text{range } f$

by *fastforce*

lemma *in-set-i-takeD*: $x \in \text{set } (f \Downarrow n) \implies x \in \text{range } f$

by (*rule subsetD[OF set-i-take-subset]*)

lemma *in-range-i-takeD*: $x \in \text{range } (f \uparrow n) \implies x \in \text{range } f$

by (*rule subsetD[OF range-i-drop-subset]*)

lemma *i-append-eq-conv-conj*:

$$((xs \frown f) = g) = (xs = g \Downarrow \text{length } xs \wedge f = g \uparrow \text{length } xs)$$

apply (*simp add: expand-ilst-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*:

$length\ xs \leq length\ ys \implies$

$(xs \frown f = ys \frown g) = (xs = ys \downarrow length\ xs \wedge f = (ys \uparrow length\ xs) \frown g)$

apply (simp add: expand-list-eq expand-ilst-eq i-append-nth min-eqR)

apply (rule iffI)

apply simp

apply (clarify, rename-tac x)

apply (drule-tac x=length xs + x in spec)

apply (simp add: less-diff-conv add.commute[of - length xs])

apply simp

done

lemma *i-append-eq-i-append-conv-if*:

$(xs \frown f = ys \frown g) =$

$(if\ length\ xs \leq length\ ys$

$\ then\ xs = ys \downarrow length\ xs \wedge f = (ys \uparrow length\ xs) \frown g$

$\ else\ xs \downarrow length\ ys = ys \wedge (xs \uparrow length\ ys) \frown f = g)$

apply (split if-split, intro conjI impI)

apply (simp add: i-append-eq-i-append-conv-if-aux)

apply (force simp: eq-commute[of xs \frown f] i-append-eq-i-append-conv-if-aux)

done

lemma *i-take-hd-i-drop*: $(f \downarrow n) @ [(f \uparrow n) 0] = f \downarrow Suc\ n$

by (simp add: i-take-Suc-conv-app-nth)

lemma *id-i-take-nth-i-drop*: $f = (f \downarrow n) \frown (([f\ n] \frown f) \uparrow Suc\ n)$

by (simp add: i-drop-Suc-Cons)

lemma *upd-conv-i-take-nth-i-drop*:

$f\ (n := x) = (f \downarrow n) \frown ([x] \frown (f \uparrow Suc\ n))$

by (simp add: expand-ilst-eq nth-append i-append-nth)

theorem *i-take-induct*:

$\llbracket P\ (f \downarrow 0); \bigwedge n. P\ (f \downarrow n) \implies P\ (f \downarrow Suc\ n) \rrbracket \implies P\ (f \downarrow n)$

by (rule nat.induct)

theorem *take-induct[rule-format]*:

$\llbracket P\ (s \downarrow 0);$

$\ \bigwedge n. \llbracket Suc\ n < length\ s; P\ (s \downarrow n) \rrbracket \implies P\ (s \downarrow Suc\ n);$

$\ i < length\ s\rrbracket$

$\implies P\ (s \downarrow i)$

by (induct i, simp+)

theorem *i-drop-induct*:

$\llbracket P\ (f \uparrow 0); \bigwedge n. P\ (f \uparrow n) \implies P\ (f \uparrow Suc\ n) \rrbracket \implies P\ (f \uparrow n)$

by (rule nat.induct)

theorem *f-drop-induct*[rule-format]:

$$\begin{aligned} & \llbracket P (s \uparrow 0); \\ & \quad \bigwedge n. \llbracket \text{Suc } n < \text{length } s; P (s \uparrow n) \rrbracket \implies P (s \uparrow \text{Suc } n); \\ & \quad i < \text{length } s \rrbracket \\ & \implies P (s \uparrow i) \end{aligned}$$
by (*induct i, simp+*)

lemma *i-take-drop-eq-map*: $f \uparrow m \downarrow n = \text{map } f [m..<m+n]$
by (*simp add: expand-list-eq*)

lemma *o-eq-i-append-imp*:

$$f \circ g = ys \frown i \implies$$

$$\exists xs \ h. g = xs \frown h \wedge \text{map } f \ xs = ys \wedge f \circ h = i$$
apply (*rule-tac x=g \downarrow (length ys) in exI*)
apply (*rule-tac x=g \uparrow (length ys) in exI*)
apply (*frule arg-cong[where f=\lambda x. x \downarrow length ys]*)
apply (*drule arg-cong[where f=\lambda x. x \uparrow length ys]*)
apply *simp*
done

corollary *o-eq-i-append-conv*:

$$(f \circ g = ys \frown i) =$$

$$(\exists xs \ h. g = xs \frown h \wedge \text{map } f \ xs = ys \wedge f \circ h = i)$$
by (*fastforce simp: o-eq-i-append-imp*)

corollary *i-append-eq-o-conv*:

$$(ys \frown i = f \circ g) =$$

$$(\exists xs \ h. g = xs \frown h \wedge \text{map } f \ xs = ys \wedge f \circ h = i)$$
by (*fastforce simp: o-eq-i-append-imp*)

11.1.3 zip for infinite lists

definition *i-zip* :: 'a ilist \Rightarrow 'b ilist \Rightarrow ('a \times 'b) ilist
where *i-zip* *f g* $\equiv \lambda n. (f \ n, g \ n)$

lemma *i-zip-nth*: $(i\text{-zip } f \ g) \ n = (f \ n, g \ n)$
by (*simp add: i-zip-def*)

lemma *i-zip-swap*: $(\lambda(y, x). (x, y)) \circ i\text{-zip } g \ f = i\text{-zip } f \ g$
by (*simp add: expand-ilist-eq i-zip-nth*)

lemma *i-zip-i-take*: $(i\text{-zip } f \ g) \ \downarrow \ n = \text{zip } (f \ \downarrow \ n) (g \ \downarrow \ n)$
by (*simp add: expand-list-eq i-zip-nth*)

lemma *i-zip-i-drop*: $(i\text{-zip } f \ g) \ \uparrow \ n = i\text{-zip } (f \ \uparrow \ n) (g \ \uparrow \ n)$
by (*simp add: expand-ilist-eq i-zip-nth*)

lemma *fst-o-izip*: $\text{fst} \circ (i\text{-zip } f \ g) = f$
by (*simp add: expand-ilist-eq i-zip-nth*)

lemma *snd-o-i-zip*: $\text{snd} \circ (\text{i-zip } f \ g) = g$
by (*simp add: expand-ilst-eq i-zip-nth*)

lemma *update-i-zip*:
 $(\text{i-zip } f \ g)(n := xy) = \text{i-zip } (f(n := \text{fst } xy)) (g(n := \text{snd } xy))$
by (*simp add: expand-ilst-eq i-zip-nth*)

lemma *i-zip-Cons-Cons*:
 $\text{i-zip } ([x] \frown f) ([y] \frown g) = [(x, y)] \frown (\text{i-zip } f \ g)$
by (*simp add: expand-ilst-eq i-zip-nth i-append-nth*)

lemma *i-zip-i-append1*:
 $\text{i-zip } (xs \frown f) \ g = \text{zip } xs \ (g \Downarrow \text{length } xs) \frown (\text{i-zip } f \ (g \Uparrow \text{length } xs))$
by (*simp add: expand-ilst-eq i-zip-nth i-append-nth*)

lemma *i-zip-i-append2*:
 $\text{i-zip } f \ (ys \frown g) = \text{zip } (f \Downarrow \text{length } ys) \ ys \frown (\text{i-zip } (f \Uparrow \text{length } ys) \ g)$
by (*simp add: expand-ilst-eq i-zip-nth i-append-nth*)

lemma *i-zip-append*:
 $\text{length } xs = \text{length } ys \implies$
 $\text{i-zip } (xs \frown f) \ (ys \frown g) = \text{zip } xs \ ys \frown (\text{i-zip } f \ g)$
by (*simp add: expand-ilst-eq i-zip-nth i-append-nth*)

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

lemma *i-zip-update*:
 $\text{i-zip } (f(n := x)) \ (g(n := y)) = (\text{i-zip } f \ g)(n := (x, y))$
by (*simp add: update-i-zip*)

lemma *i-zip-const*: $\text{i-zip } (\lambda n. x) \ (\lambda n. y) = (\lambda n. (x, y))$
by (*simp add: expand-ilst-eq i-zip-nth*)

11.1.4 Mapping functions with two arguments to infinite lists

definition *i-map2* ::
 — Function taking two parameters
 $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow$
 — Lists of parameters
 $'a \ \text{ilst} \Rightarrow 'b \ \text{ilst} \Rightarrow$
 $'c \ \text{ilst}$

where
 $\text{i-map2 } f \ xs \ ys \equiv \lambda n. f \ (xs \ n) \ (ys \ n)$

lemma *i-map2-nth*: $(\text{i-map2 } f \ xs \ ys) \ n = f \ (xs \ n) \ (ys \ n)$
by (*simp add: i-map2-def*)

lemma *i-map2-Cons-Cons*:
 $i\text{-map2 } f \ ([x] \frown xs) \ ([y] \frown ys) =$
 $[f \ x \ y] \frown (i\text{-map2 } f \ xs \ ys)$
by (*simp add: fun-eq-iff i-map2-nth i-append-nth-Cons'*)

lemma *i-map2-take-ge*:
 $n \leq n1 \implies$
 $i\text{-map2 } f \ xs \ ys \Downarrow n =$
 $\text{map2 } f \ (xs \Downarrow n) \ (ys \Downarrow n1)$
by (*simp add: expand-list-eq map2-length i-map2-nth map2-nth*)

lemma *i-map2-take-take*:
 $i\text{-map2 } f \ xs \ ys \Downarrow n =$
 $\text{map2 } f \ (xs \Downarrow n) \ (ys \Downarrow n)$
by (*rule i-map2-take-ge[OF le-refl]*)

lemma *i-map2-drop*:
 $(i\text{-map2 } f \ xs \ ys) \Uparrow n =$
 $(i\text{-map2 } f \ (xs \Uparrow n) \ (ys \Uparrow n))$
by (*simp add: fun-eq-iff i-map2-nth*)

lemma *i-map2-append-append*:
 $\text{length } xs1 = \text{length } ys1 \implies$
 $i\text{-map2 } f \ (xs1 \frown xs) \ (ys1 \frown ys) =$
 $\text{map2 } f \ xs1 \ ys1 \frown i\text{-map2 } f \ xs \ ys$
by (*simp add: fun-eq-iff i-map2-nth i-append-nth map2-length map2-nth*)

lemma *i-map2-Cons-left*:
 $i\text{-map2 } f \ ([x] \frown xs) \ ys =$
 $[f \ x \ (ys \ 0)] \frown i\text{-map2 } f \ xs \ (ys \Uparrow \text{Suc } 0)$
by (*simp add: fun-eq-iff i-map2-nth i-append-nth-Cons'*)

lemma *i-map2-Cons-right*:
 $i\text{-map2 } f \ xs \ ([y] \frown ys) =$
 $[f \ (xs \ 0) \ y] \frown i\text{-map2 } f \ (xs \Uparrow \text{Suc } 0) \ ys$
by (*simp add: fun-eq-iff i-map2-nth i-append-nth-Cons'*)

lemma *i-map2-append-take-drop-left*:
 $i\text{-map2 } f \ (xs1 \frown xs) \ ys =$
 $\text{map2 } f \ xs1 \ (ys \Downarrow \text{length } xs1) \frown$
 $i\text{-map2 } f \ xs \ (ys \Uparrow \text{length } xs1)$
by (*simp add: fun-eq-iff map2-nth i-map2-nth i-append-nth map2-length*)

lemma *i-map2-append-take-drop-right*:
 $i\text{-map2 } f \ xs \ (ys1 \frown ys) =$
 $\text{map2 } f \ (xs \Downarrow \text{length } ys1) \ ys1 \frown$
 $i\text{-map2 } f \ (xs \Uparrow \text{length } ys1) \ ys$
by (*simp add: fun-eq-iff map2-nth i-map2-nth i-append-nth map2-length*)

lemma *i-map2-cong*:
 $\llbracket xs1 = xs2; ys1 = ys2; \rrbracket$

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

lemma *i-map2-eq-conv*:

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

lemma *i-map2-replicate*: $i\text{-map2 } f (\lambda n. x) (\lambda n. y) = (\lambda n. f x y)$

by (*simp add: fun-eq-iff i-map2-nth*)

lemma *i-map2-i-zip-conv*:

$i\text{-map2 } f xs ys = (\lambda(x,y). f x y) \circ (i\text{-zip } xs ys)$
by (*simp add: fun-eq-iff i-map2-nth i-zip-nth*)

11.2 Generalised lists as combination of finite and infinite lists

11.2.1 Basic definitions

datatype (*gset: 'a*) *glist* = *FL 'a list* | *IL 'a ilist* **for** *map: gmap*

definition *glength :: 'a glist \Rightarrow enat*

where

$glength a \equiv \text{case } a \text{ of}$
 $FL xs \Rightarrow \text{enat } (\text{length } xs) \mid$
 $IL f \Rightarrow \infty$

definition *gCons :: 'a \Rightarrow 'a glist \Rightarrow 'a glist* (**infixr** $\#_g$ 65)

where

$x \#_g a \equiv \text{case } a \text{ of}$
 $FL xs \Rightarrow FL (x \# xs) \mid$
 $IL g \Rightarrow IL ([x] \frown g)$

definition *gappend :: 'a glist \Rightarrow 'a glist \Rightarrow 'a glist* (**infixr** $@_g$ 65)

where

$gappend a b \equiv \text{case } a \text{ of}$
 $FL xs \Rightarrow (\text{case } b \text{ of } FL ys \Rightarrow FL (xs @ ys) \mid IL f \Rightarrow IL (xs \frown f)) \mid$
 $IL f \Rightarrow IL f$

definition *gtake :: enat \Rightarrow 'a glist \Rightarrow 'a glist*

where

$gtake n a \equiv \text{case } n \text{ of}$
 $\text{enat } m \Rightarrow FL (\text{case } a \text{ of}$
 $FL xs \Rightarrow xs \downarrow m \mid$
 $IL f \Rightarrow f \Downarrow m) \mid$
 $\infty \Rightarrow a$

definition *gdrop :: enat \Rightarrow 'a glist \Rightarrow 'a glist*

where

$$\begin{aligned}
\text{gdrop } n \ a &\equiv \text{case } n \ \text{of} \\
\text{enat } m &\Rightarrow (\text{case } a \ \text{of} \\
&\quad \text{FL } xs \Rightarrow \text{FL } (xs \uparrow m) \mid \\
&\quad \text{IL } f \Rightarrow \text{IL } (f \uparrow m)) \mid \\
\infty &\Rightarrow \text{FL } []
\end{aligned}$$

definition $\text{gnth} :: 'a \ \text{glist} \Rightarrow \text{nat} \Rightarrow 'a$ (**infixl** $!_g$ 100)

where

$$\begin{aligned}
a \ !_g \ n &\equiv \text{case } a \ \text{of} \\
\text{FL } xs &\Rightarrow xs \ ! \ n \mid \\
\text{IL } f &\Rightarrow f \ n
\end{aligned}$$

abbreviation $\text{g-take}' :: 'a \ \text{glist} \Rightarrow \text{enat} \Rightarrow 'a \ \text{glist}$ (**infixl** \downarrow_g 100)

where $a \ \downarrow_g \ n \equiv \text{gtake } n \ a$

abbreviation $\text{g-drop}' :: 'a \ \text{glist} \Rightarrow \text{enat} \Rightarrow 'a \ \text{glist}$ (**infixl** \uparrow_g 100)

where $a \ \uparrow_g \ n \equiv \text{gdrop } n \ a$

11.2.2 *glength*

lemma $\text{glength-fin}[\text{simp}]$: $\text{glength } (\text{FL } xs) = \text{enat } (\text{length } xs)$

by ($\text{simp add: glength-def}$)

lemma $\text{glength-infin}[\text{simp}]$: $\text{glength } (\text{IL } f) = \infty$

by ($\text{simp add: glength-def}$)

lemma $\text{gappend-glength}[\text{simp}]$: $\text{glength } (a \ @_g \ b) = \text{glength } a + \text{glength } b$

by ($\text{unfold gappend-def, case-tac } a, \text{ case-tac } b, \text{ simp+}$)

lemma $\text{gmap-glength}[\text{simp}]$: $\text{glength } (\text{gmap } f \ a) = \text{glength } a$

by ($\text{case-tac } a, \text{ simp+}$)

lemma $\text{glength-0-conv}[\text{simp}]$: $(\text{glength } a = 0) = (a = \text{FL } [])$

by ($\text{unfold glength-def, case-tac } a, \text{ simp+}$)

lemma $\text{glength-greater-0-conv}[\text{simp}]$: $(0 < \text{glength } a) = (a \neq \text{FL } [])$

by ($\text{simp add: glength-0-conv}[\text{symmetric}]$)

lemma glength-gSuc-conv :

$$\begin{aligned}
(\text{glength } a = \text{eSuc } n) &= \\
(\exists x \ b. a = x \ \#_g \ b \ \wedge \ \text{glength } b = n)
\end{aligned}$$

apply ($\text{unfold glength-def gCons-def, rule iffI}$)

apply ($\text{case-tac } a, \text{ rename-tac } a'$)

apply ($\text{case-tac } n, \text{ rename-tac } n'$)

apply ($\text{rule-tac } x = \text{hd } a' \ \text{in } \text{exI}$)

apply ($\text{rule-tac } x = \text{FL } (\text{tl } a') \ \text{in } \text{exI}$)

apply ($\text{simp add: eSuc-enat}$)

apply ($\text{subgoal-tac } a' \neq []$)

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-glenth-conv*:
 $(eSuc\ n = glenth\ a) =$
 $(\exists x\ b. a = x \#_g b \wedge glenth\ b = n)$
by (*simp add: eq-commute[of - glenth a] glenth-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]*: $(a @_g\ b = FL\ []) = (a = FL\ [] \wedge b = FL\ [])$
by (*unfold gappend-def, case-tac a, case-tac b, simp+*)

lemma *Nil-is-gappend-conv[simp]*: $(FL\ [] = a @_g\ b) = (a = FL\ [] \wedge b = FL\ [])$
by (*simp add: eq-commute[of FL []]*)

lemma *gappend-assoc[simp]*: $(a @_g\ b) @_g\ c = a @_g\ b @_g\ c$
by (*unfold gappend-def, case-tac a, case-tac b, case-tac c, simp+*)

lemma *gappend-infin[simp]*: $IL\ f @_g\ b = IL\ f$
by (*simp add: gappend-def*)

lemma *same-gappend-eq-disj[simp]*: $(a @_g\ b = a @_g\ c) = (glenth\ a = \infty \vee b = c)$

```

  apply (case-tac a)
  apply simp
  apply (case-tac b, case-tac c)
  apply (simp add: gappend-def)+
  apply (case-tac c)
  apply simp+

```

done

lemma *same-gappend-eq*:

$glength\ a < \infty \implies (a @_g b = a @_g c) = (b = c)$

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]*: $(gmap\ f\ a = gmap\ g\ a) = (\forall x \in gset\ a.\ f\ x = g\ x)$

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*:

$gmap\ f\ a = gmap\ f\ b \implies glength\ a = glength\ b$

by (*drule arg-cong[where f=length], simp*)

11.2.5 *gset*

lemma *gset-gappend[simp]*:

$gset\ (a @_g b) =$

$(case\ a\ of\ FL\ a' \Rightarrow set\ a' \cup gset\ b \mid IL\ a' \Rightarrow range\ a')$

by (*unfold gappend-def, case-tac a, case-tac b, simp+*)

lemma *gset-gappend-if*:

$gset\ (a @_g b) =$

$(if\ glength\ a < \infty\ then\ gset\ a \cup gset\ b\ else\ gset\ a)$

by (*unfold 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*: $icard\ (gset\ a) \leq glength\ a$

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

lemma *gnth-gappend*:

$(a @_g b) !_g n =$
(if enat $n < \text{glength } a$ then $a !_g n$
else $b !_g (n - \text{the-enat } (\text{glength } a))$)

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]: $(FL \text{ } xs @_g b) !_g (\text{length } xs + n) = b !_g n$
by (*simp add: gnth-gappend*)

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

lemma *in-gset-cong-gnth*: $(x \in \text{gset } a) = (\exists i. \text{enat } i < \text{glength } a \wedge a !_g i = x)$
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]: $\text{glength } a \leq n \implies a \downarrow_g n = a$
by (*unfold gtake-def, case-tac a, case-tac n, simp+*)

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

lemma *gtake-eSuc-gCons*[simp]: $(x \#_g a) \downarrow_g (\text{eSuc } n) = x \#_g a \downarrow_g n$
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*: $a \neq FL\ [] \implies a \downarrow_g (eSuc\ n) = a !_g 0 \#_g (a \uparrow_g (eSuc\ 0) \downarrow_g n)$

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*: $a \uparrow_g (enat\ n) = x \#_g b \implies a !_g n = x$

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 \implies 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 \implies 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 = ( $\exists n. ys \downarrow n = xs$ )
apply (unfold prefix-def, safe)
apply (rule-tac x=length xs in exI, simp)
apply (rule-tac x=ys  $\uparrow$  n in exI, simp)
done

```

```

lemma prefix-take-eq-prefix-take-ex: ( $ys \downarrow (\text{length } xs) = xs$ ) = ( $\exists n. ys \downarrow n = xs$ )
by (fastforce simp: min-def)

```

```

lemma prefix-eq-prefix-take: prefix xs ys = ( $ys \downarrow (\text{length } xs) = xs$ )
by (simp only: prefix-eq-prefix-take-ex prefix-take-eq-prefix-take-ex)

```

```

lemma strict-prefix-take-eq-strict-prefix-take-ex:
  ( $ys \downarrow (\text{length } xs) = xs \wedge xs \neq ys$ ) =
  ( $(\exists n. ys \downarrow n = xs) \wedge xs \neq ys$ )
by (simp add: prefix-take-eq-prefix-take-ex)

```

```

lemma strict-prefix-eq-strict-prefix-take-ex: strict-prefix xs ys = ( $(\exists n. ys \downarrow n = xs)$ 
 $\wedge xs \neq ys$ )
by (simp add: strict-prefix-def prefix-eq-prefix-take-ex)

```

```

lemma strict-prefix-eq-strict-prefix-take: strict-prefix xs ys = ( $ys \downarrow (\text{length } xs) =$ 
 $xs \wedge xs \neq ys$ )
by (simp only: strict-prefix-eq-strict-prefix-take-ex strict-prefix-take-eq-strict-prefix-take-ex)

```

```

lemma take-imp-prefix: prefix (xs  $\downarrow$  n) xs
by (rule take-is-prefix)

```

```

lemma eq-imp-prefix: xs = (ys::'a list)  $\implies$  prefix xs ys
by simp

```

```

lemma le-take-imp-prefix:  $a \leq b \implies$  prefix (xs  $\downarrow$  a) (xs  $\downarrow$  b)
by (fastforce simp: prefix-eq-prefix-take-ex min-def)

```

```

lemma take-prefix-imp-le:
  [ $a \leq \text{length } xs$ ; prefix (xs  $\downarrow$  a) (xs  $\downarrow$  b)]  $\implies a \leq b$ 
by (drule prefix-length-le, simp)

```

```

lemma take-prefixeq-le-conv:
   $a \leq \text{length } xs \implies$  prefix (xs  $\downarrow$  a) (xs  $\downarrow$  b) = ( $a \leq b$ )
apply (rule iffI)

```

```

apply (rule take-prefix-imp-le, assumption+)
apply (rule le-take-imp-prefix, assumption+)
done
lemma append-imp-prefix[simp, intro]: prefix a (a @ b)
by (unfold prefix-def, blast)

```

```

lemma prefix-imp-take-eq:
   $\llbracket n \leq \text{length } xs; \text{prefix } xs \text{ } ys \rrbracket \implies xs \downarrow n = ys \downarrow n$ 
by (clarsimp simp: prefix-def)

```

```

lemma prefix-length-le-eq-conv: (prefix xs ys  $\wedge$  length ys  $\leq$  length xs) = (xs = ys)
apply (rule iffI)
apply (erule conjE)
apply (frule prefix-length-le)
apply (simp add: prefix-eq-prefix-take)
apply simp
done

```

```

lemma take-length-prefix-conv:
  length xs  $\leq$  length ys  $\implies$  prefix (ys  $\downarrow$  length xs) xs = prefix xs ys
by (fastforce simp: prefix-eq-prefix-take)

```

```

lemma append-eq-imp-take:
   $\llbracket k \leq \text{length } xs; \text{length } r1 = k; r1 @ r2 = xs \rrbracket \implies r1 = xs \downarrow k$ 
by fastforce

```

```

lemma take-the-conv:
   $xs \downarrow k = (\text{if } \text{length } xs \leq k \text{ then } xs \text{ else } (\text{THE } r. \text{length } r = k \wedge (\exists r2. r @ r2 = xs)))$ 
apply (clarsimp simp: linorder-not-le)
apply (rule the1I2)
apply (simp add: Ex1-def)
apply (rule-tac x=xs  $\downarrow$  k in exI)
apply (intro conjI)
  apply (simp add: min-eqR)
  apply (rule-tac x=xs  $\uparrow$  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:  $\llbracket \text{prefix } xs \text{ } ys; \text{prefix } (ys::'a \text{ list}) \text{ } zs \rrbracket \implies \text{prefix } xs \text{ } zs$ 
by (rule prefix-order.order-trans)

```

```

lemma prefixeq-antisym:  $\llbracket \text{prefix } xs \text{ } ys; \text{prefix } (ys::'a \text{ list}) \text{ } xs \rrbracket \implies xs = ys$ 
by (rule prefix-order.antisym)

```

12.2 Counting equal pairs

Counting number of equal elements in two lists

definition *mirror-pair* :: ('a × 'b) ⇒ ('b × 'a)
where *mirror-pair* p ≡ (snd p, fst p)

lemma *zip-mirror*[*rule-format*]:
 [[*i* < min (length *xs*) (length *ys*);
p1 = (zip *xs ys*) ! *i*; *p2* = (zip *ys xs*) ! *i*]] ⇒
mirror-pair p1 = *p2*
by (*simp add: mirror-pair-def*)

definition *equal-pair* :: ('a × 'a) ⇒ bool
where *equal-pair* p ≡ (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 × 'a) list ⇒ nat

where

equal-pair-count [] = 0

| *equal-pair-count* (p # ps) = (
 if (fst p = snd p)
 then Suc (*equal-pair-count* ps)
 else 0)

lemma *equal-pair-count-le*: *equal-pair-count xs* ≤ length *xs*
by (*induct xs, simp-all*)

lemma *equal-pair-count-0*:

fst (hd ps) ≠ *snd* (hd ps) ⇒ *equal-pair-count ps* = 0

by (*case-tac ps, simp-all*)

lemma *equal-pair-count-Suc*:

equal-pair-count ((a, a) # ps) = Suc (*equal-pair-count ps*)

by *simp*

lemma *equal-pair-count-eq-pairwise*[*rule-format*]:

[[length *ps1* = length *ps2*;

∀ *i* < length *ps2*. *equal-pair* (*ps1* ! *i*) = *equal-pair*(*ps2* ! *i*)]]

⇒ *equal-pair-count ps1* = *equal-pair-count ps2*

apply (*induct rule: list-induct2*)

apply *simp*

apply (*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 \Rightarrow 'a list \Rightarrow nat
where *inf-prefix-length* *xs ys* \equiv *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* \leq *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* \leq *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* \leq *min* (*length* *xs*) (*length* *ys*)
by (*simp add: inf-prefix-length-le*)

lemma *hd-inf-prefix-length-0*:
 hd *xs* \neq hd *ys* \implies *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]: $\text{inf-prefix-length } [] \text{ } ys = 0$
by (*simp add: inf-prefix-length-def*)
lemma *inf-prefix-length-NilR*[simp]: $\text{inf-prefix-length } xs \text{ } [] = 0$
by (*simp add: inf-prefix-length-def*)

lemma *inf-prefix-length-Suc*[simp]:
 $\text{inf-prefix-length } (a \# xs) (a \# ys) = \text{Suc } (\text{inf-prefix-length } xs \text{ } ys)$
by (*simp add: inf-prefix-length-def*)

lemma *inf-prefix-length-correct*:
 $i < \text{inf-prefix-length } xs \text{ } ys \implies xs ! i = ys ! i$
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*:
 $xs ! i \neq ys ! i \implies \text{inf-prefix-length } xs \text{ } ys \leq i$
apply (*rule ccontr*)
apply (*simp add: inf-prefix-length-correct*)
done

lemma *inf-prefix-length-maximality1*[*rule-format*]:
 $\text{inf-prefix-length } xs \text{ } ys \neq \min (\text{length } xs) (\text{length } ys) \implies$
 $xs ! (\text{inf-prefix-length } xs \text{ } ys) \neq ys ! (\text{inf-prefix-length } xs \text{ } ys)$
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*]:
 $\llbracket \text{inf-prefix-length } xs \text{ } ys \neq \min (\text{length } xs) (\text{length } ys);$
 $\text{inf-prefix-length } xs \text{ } ys \leq i \rrbracket \implies$
 $\exists j \leq i. xs ! j \neq ys ! j$
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]:
 $\text{inf-prefix-length } (zs @ xs) (zs @ ys) =$
 $\text{length } zs + \text{inf-prefix-length } xs \text{ } ys$

```

apply (induct zs)
apply simp
apply (simp add: inf-prefix-length-def)
done

```

```

lemma inf-prefix-length-take-correct:
   $n \leq \text{inf-prefix-length } xs \ ys \implies xs \downarrow n = ys \downarrow n$ 
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-neg:
   $\llbracket xs \neq []; ys \neq []; \text{inf-prefix-length } xs \ ys = 0 \rrbracket \implies \text{hd } xs \neq \text{hd } ys$ 
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  $\Rightarrow$  'a list  $\Rightarrow$  'a list (infixl  $\sqcap$  70)
  where  $xs \sqcap ys \equiv xs \downarrow (\text{inf-prefix-length } xs \ ys)$ 

```

```

lemma length-inf-prefix:  $\text{length } (xs \sqcap ys) = \text{inf-prefix-length } xs \ ys$ 
by (simp add: inf-prefix-def min-eqR inf-prefix-length-leL)

```

```

lemma inf-prefix-commute:  $xs \sqcap ys = ys \sqcap xs$ 
by (simp add: inf-prefix-def inf-prefix-length-commute[of ys] inf-prefix-length-take-correct)

```

```

lemma inf-prefix-takeL:  $xs \sqcap ys = xs \downarrow (\text{inf-prefix-length } xs \ ys)$ 
by (simp add: inf-prefix-def)

```

```

lemma inf-prefix-takeR:  $xs \sqcap ys = ys \downarrow (\text{inf-prefix-length } xs \ ys)$ 
by (subst inf-prefix-commute, subst inf-prefix-length-commute, rule inf-prefix-takeL)

```

```

lemma inf-prefix-correct:  $i < \text{length } (xs \sqcap ys) \implies xs ! i = ys ! i$ 
by (simp add: length-inf-prefix inf-prefix-length-correct)

```

```

corollary inf-prefix-correctL:
   $i < \text{length } (xs \sqcap ys) \implies (xs \sqcap ys) ! i = xs ! i$ 
by (simp add: inf-prefix-takeL)

```

```

corollary inf-prefix-correctR:
   $i < \text{length } (xs \sqcap ys) \implies (xs \sqcap ys) ! i = ys ! i$ 
by (simp add: inf-prefix-takeR)

```

```

lemma inf-prefix-take-correct:
   $n \leq \text{length } (xs \sqcap ys) \implies xs \downarrow n = ys \downarrow n$ 

```

by (*simp add: length-inf-prefix inf-prefix-length-take-correct*)

lemma *is-inf-prefix*[*rule-format*]:

$\llbracket \text{length } zs = \text{length } (xs \sqcap ys);$

$\bigwedge i. i < \text{length } (xs \sqcap ys) \implies zs ! i = xs ! i \wedge zs ! i = ys ! i \rrbracket \implies$

$zs = xs \sqcap ys$

by (*simp add: list-eq-iff inf-prefix-def*)

lemma *hd-inf-prefix-Nil*: $hd \ xs \neq hd \ ys \implies xs \sqcap ys = []$

by (*simp add: inf-prefix-def hd-inf-prefix-length-0*)

lemma *inf-prefix-Nil-imp-hd-neq*:

$\llbracket xs \neq []; ys \neq []; xs \sqcap ys = [] \rrbracket \implies hd \ xs \neq hd \ ys$

by (*simp add: inf-prefix-def inf-prefix-length-0-imp-hd-neq*)

lemma *length-inf-prefix-append*[*simp*]:

$\text{length } ((zs @ xs) \sqcap (zs @ ys)) =$

$\text{length } zs + \text{length } (xs \sqcap ys)$

by (*simp add: length-inf-prefix*)

lemma *inf-prefix-append*[*simp*]: $(zs @ xs) \sqcap (zs @ ys) = zs @ (xs \sqcap ys)$

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*:

$hd \ xs \neq hd \ ys \implies (zs @ xs) \sqcap (zs @ ys) = zs$

by (*simp add: hd-inf-prefix-Nil*)

lemma *inf-prefix-NilL*[*simp*]: $[] \sqcap ys = []$

by (*simp del: length-0-conv add: length-0-conv[symmetric] length-inf-prefix*)

corollary *inf-prefix-NilR*[*simp*]: $xs \sqcap [] = []$

by (*simp add: inf-prefix-commute*)

lemmas *inf-prefix-Nil = inf-prefix-NilL inf-prefix-NilR*

lemma *inf-prefix-Cons*[*simp*]: $(a \# xs) \sqcap (a \# ys) = a \# xs \sqcap ys$

by (*insert inf-prefix-append[of [a] xs ys], simp*)

corollary *inf-prefix-hd*[*simp*]: $hd \ ((a \# xs) \sqcap (a \# ys)) = a$

by *simp*

lemma *inf-prefix-le1*: $\text{prefix } (xs \sqcap ys) \ xs$

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

lemma *inf-prefix-le2*: $\text{prefix } (xs \sqcap ys) \ ys$

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

lemma *le-inf-prefix-iff*: $\text{prefix } x (y \sqcap z) = (\text{prefix } x y \wedge \text{prefix } x z)$
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*: $\llbracket \text{prefix } x y; \text{prefix } x z \rrbracket \implies \text{prefix } x (y \sqcap z)$
by (rule le-inf-prefix-iff[THEN iffD2], simp)

interpretation *prefix*:

semilattice-inf
 $(\sqcap) :: 'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}$
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* :: $'a \text{ list} \Rightarrow 'a \text{ ilist} \Rightarrow \text{bool}$ (**infixl** \sqsubseteq 50)
where $xs \sqsubseteq f \equiv \exists g. f = xs \frown g$

lemma *iprefix-eq-iprefix-take*: $(xs \sqsubseteq f) = (f \Downarrow \text{length } xs = xs)$
apply (unfold iprefix-def)
apply (rule iffI)
apply clarsimp
apply (rule-tac $x=f \uparrow (\text{length } xs)$ **in** *exI*)
apply (subst *i-append-i-take-i-drop-id*[**where** $n=\text{length } xs$, *symmetric*], simp)
done

lemma *iprefix-take-eq-iprefix-take-ex*:
 $(f \Downarrow \text{length } xs = xs) = (\exists n. f \Downarrow n = xs)$
apply (rule iffI)
apply (rule-tac $x=\text{length } xs$ **in** *exI*, assumption)
apply clarsimp
done

lemma *iprefix-eq-iprefix-take-ex*: $(xs \sqsubseteq f) = (\exists n. f \Downarrow n = xs)$
by (simp add: iprefix-eq-iprefix-take iprefix-take-eq-iprefix-take-ex)

lemma *i-take-imp-iprefix*[intro]: $f \Downarrow n \sqsubseteq f$
by (simp add: iprefix-eq-iprefix-take)

lemma *i-take-prefix-le-conv*: $\text{prefix } (f \Downarrow a) (f \Downarrow b) = (a \leq b)$
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*:
 $\llbracket n \leq \text{length } xs; xs \sqsubseteq f \rrbracket \implies xs \Downarrow n = f \Downarrow n$
by (*clarsimp simp: iprefix-eq-iprefix-take-ex min-eqR*)

lemma *prefixeq-iprefix-trans*: $\llbracket \text{prefix } xs \ ys; ys \sqsubseteq f \rrbracket \implies xs \sqsubseteq f$
by (*fastforce simp: iprefix-eq-iprefix-take-ex prefix-eq-prefix-take-ex*)

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

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

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

lemma *Nil-iprefix[iff]*: $\llbracket \rrbracket \sqsubseteq f$
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]*: $\text{prefix } xs \ ys \implies xs \sqsubseteq ys \frown f$
by (*clarsimp simp: prefix-def iprefix-def i-append-assoc[symmetric] simp del: i-append-assoc*)

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

lemma *iprefix-length-le-imp-prefix*:
 $\llbracket xs \sqsubseteq ys \frown f; \text{length } xs \leq \text{length } ys \rrbracket \implies \text{prefix } xs \ ys$
by (*clarsimp simp: iprefix-eq-iprefix-take-ex take-is-prefix*)

lemma *iprefix-i-append*:
 $(xs \sqsubseteq ys \frown f) = (\text{prefix } xs \ ys \vee (\exists zs. xs = ys @ zs \wedge zs \sqsubseteq f))$
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:
   $xs \sqsubseteq f \implies xs @ [f (length\ xs)] \sqsubseteq f$ 
by (simp add: iprefix-eq-iprefix-take i-take-Suc-conv-app-nth)

lemma iprefix-same-length-le:
   $\llbracket xs \sqsubseteq f; ys \sqsubseteq f; length\ xs \leq length\ ys \rrbracket \implies prefix\ xs\ ys$ 
by (clarsimp simp: iprefix-eq-iprefix-take-ex i-take-prefix-le-conv)

lemma iprefix-same-cases:
   $\llbracket xs \sqsubseteq f; ys \sqsubseteq f \rrbracket \implies prefix\ xs\ ys \vee prefix\ ys\ xs$ 
apply (case-tac length\ xs \leq length\ ys)
apply (simp add: iprefix-same-length-le)+
done

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

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

```