

Formalisation of Ground Resolution and CDCL in Isabelle/HOL

Mathias Fleury and Jasmin Blanchette

April 25, 2020

Contents

1	More Standard Theorems	5
1.1	Transitions	5
1.1.1	More theorems about Closures	5
1.1.2	Full Transitions	6
1.1.3	Well-Foundedness and Full Transitions	8
1.1.4	More Well-Foundedness	8
1.2	Various Lemmas	13
1.2.1	Not-Related to Refinement or lists	13
1.3	More Lists	15
1.3.1	set, nth, tl	15
1.3.2	List Updates	18
1.3.3	Take and drop	18
1.3.4	Replicate	19
1.3.5	List intervals (<i>upt</i>)	20
1.3.6	Lexicographic Ordering	23
1.3.7	Remove	26
1.3.8	Sorting	28
1.3.9	Distinct Multisets	29
1.3.10	Set of Distinct Multisets	29
1.3.11	Sublists	30
1.3.12	Product Case	33
1.3.13	More about <i>list-all2</i> and <i>map</i>	36
1.3.14	Multisets	36
1.4	Finite maps and multisets	45
1.4.1	Explore command	51
1.4.2	Examples	58

Chapter 1

More Standard Theorems

This chapter contains additional lemmas built on top of HOL. Some of the additional lemmas are not included here. Most of them are too specialised to move to HOL.

1.1 Transitions

This theory contains some facts about closure, the definition of full transformations, and well-foundedness.

```
theory Wellfounded-More
imports Main
```

```
begin
```

1.1.1 More theorems about Closures

This is the equivalent of the theorem *rtranclp-mono* for *tranclp*

lemma *tranclp-mono-explicit*:

$\langle r^{++} \ a \ b \implies r \leq s \implies s^{++} \ a \ b \rangle$

using *rtranclp-mono* **by** (*auto dest!*: *tranclpD intro: rtranclp-into-tranclp2*)

lemma *tranclp-mono*:

assumes *mono*: $\langle r \leq s \rangle$

shows $\langle r^{++} \leq s^{++} \rangle$

using *rtranclp-mono[OF mono]* *mono* **by** (*auto dest!*: *tranclpD intro: rtranclp-into-tranclp2*)

lemma *tranclp-idemp-rel*:

$\langle R^{++++} \ a \ b \longleftrightarrow R^{++} \ a \ b \rangle$

apply (*rule iffI*)

prefer 2 **apply** *blast*

by (*induction rule: tranclp-induct*) *auto*

Equivalent of the theorem *rtranclp-idemp*

lemma *trancl-idemp*: $\langle (r^+)^+ = r^+ \rangle$

by *simp*

lemmas *tranclp-idemp[simp]* = *trancl-idemp[to-pred]*

This theorem already exists as theroem *Nitpick.rtranclp-unfold* (and sledgehammer uses it), but

it makes sense to duplicate it, because it is unclear how stable the lemmas in the `~/src/HOL/Nitpick.thy` theory are.

lemma *rtranclp-unfold*: $\langle \text{rtranclp } r \ a \ b \longleftrightarrow (a = b \vee \text{trancpl } r \ a \ b) \rangle$
by (*meson rtranclp.simps rtranclpD trancpl-into-rtranclp*)

lemma *trancpl-unfold-end*: $\langle \text{trancpl } r \ a \ b \longleftrightarrow (\exists a'. \text{rtranclp } r \ a \ a' \wedge r \ a' \ b) \rangle$
by (*metis rtranclp.rtrancl-refl rtranclp-into-trancpl1 trancpl.cases trancpl-into-rtranclp*)

Near duplicate of theorem *trancplD*:

lemma *trancpl-unfold-begin*: $\langle \text{trancpl } r \ a \ b \longleftrightarrow (\exists a'. r \ a \ a' \wedge \text{rtranclp } r \ a' \ b) \rangle$
by (*meson rtranclp-into-trancpl2 trancplD*)

lemma *trancpl-set-trancpl*: $\langle (a, b) \in \{(b, a). P \ a \ b\}^+ \longleftrightarrow P^{++} \ b \ a \rangle$
apply (*rule iffI*)
apply (*induction rule: trancpl-induct; simp*)
apply (*induction rule: trancpl-induct; auto simp: trancpl-into-trancpl2*)
done

lemma *trancpl-rtranclp-rtranclp-rel*: $\langle R^{+++} \ a \ b \longleftrightarrow R^{**} \ a \ b \rangle$
by (*simp add: rtranclp-unfold*)

lemma *trancpl-rtranclp-rtranclp[simp]*: $\langle R^{+++} = R^{**} \rangle$
by (*fastforce simp: rtranclp-unfold*)

lemma *rtranclp-exists-last-with-prop*:
assumes $\langle R \ x \ z \rangle$ **and** $\langle R^{**} \ z \ z' \rangle$ **and** $\langle P \ x \ z \rangle$
shows $\langle \exists y \ y'. R^{**} \ x \ y \wedge R \ y \ y' \wedge P \ y \ y' \wedge (\lambda a \ b. R \ a \ b \wedge \neg P \ a \ b)^{**} \ y' \ z' \rangle$
using *assms(2,1,3)*

proof *induction*

case *base*

then show *?case* **by** *auto*

next

case (*step z' z''*) **note** $z = \text{this}(2)$ **and** $IH = \text{this}(3)[OF \ \text{this}(4-5)]$

show *?case*

apply (*cases* $\langle P \ z' \ z'' \rangle$)

apply (*rule exI[of - z'], rule exI[of - z'']*)

using $z \ \text{assms}(1) \ \text{step.hyps}(1) \ \text{step.prem}(2)$ **apply** (*auto; fail*)[1]

using $IH \ z$ **by** (*fastforce intro: rtranclp.rtrancl-into-rtrancl*)

qed

lemma *rtranclp-and-rtranclp-left*: $\langle (\lambda a \ b. P \ a \ b \wedge Q \ a \ b)^{**} \ S \ T \Longrightarrow P^{**} \ S \ T \rangle$
by (*induction rule: rtranclp-induct*) *auto*

1.1.2 Full Transitions

Definition We define here predicates to define properties after all possible transitions.

abbreviation (*input*) *no-step* :: $(a \Rightarrow b \Rightarrow \text{bool}) \Rightarrow a \Rightarrow \text{bool}$ **where**
no-step step S $\equiv \forall S'. \neg \text{step } S \ S'$

definition *full1* :: $(a \Rightarrow a \Rightarrow \text{bool}) \Rightarrow a \Rightarrow a \Rightarrow \text{bool}$ **where**
full1 transf = $(\lambda S \ S'. \text{trancpl } \text{transf } S \ S' \wedge \text{no-step } \text{transf } S')$

definition *full*:: $(a \Rightarrow a \Rightarrow \text{bool}) \Rightarrow a \Rightarrow a \Rightarrow \text{bool}$ **where**

$full\ transf = (\lambda S\ S'.\ rtrancpl\ transf\ S\ S' \wedge no\text{-}step\ transf\ S')$

We define output notations only for printing (to ease reading):

notation (output) $full1\ (-^{+\downarrow})$

notation (output) $full\ (-^{\downarrow})$

Some Properties lemma *rtrancpl-full1I*:

$\langle R^{**}\ a\ b \implies full1\ R\ b\ c \implies full1\ R\ a\ c \rangle$

unfolding *full1-def* **by** *auto*

lemma *trancpl-full1I*:

$\langle R^{++}\ a\ b \implies full1\ R\ b\ c \implies full1\ R\ a\ c \rangle$

unfolding *full1-def* **by** *auto*

lemma *rtrancpl-fullI*:

$\langle R^{**}\ a\ b \implies full\ R\ b\ c \implies full\ R\ a\ c \rangle$

unfolding *full-def* **by** *auto*

lemma *trancpl-full-full1I*:

$\langle R^{++}\ a\ b \implies full\ R\ b\ c \implies full1\ R\ a\ c \rangle$

unfolding *full-def full1-def* **by** *auto*

lemma *full-fullI*:

$\langle R\ a\ b \implies full\ R\ b\ c \implies full1\ R\ a\ c \rangle$

unfolding *full-def full1-def* **by** *auto*

lemma *full-unfold*:

$\langle full\ r\ S\ S' \longleftrightarrow ((S = S' \wedge no\text{-}step\ r\ S') \vee full1\ r\ S\ S') \rangle$

unfolding *full-def full1-def* **by** (*auto simp add: rtrancpl-unfold*)

lemma *full1-is-full[intro]*: $\langle full1\ R\ S\ T \implies full\ R\ S\ T \rangle$

by (*simp add: full-unfold*)

lemma *not-full1-rtrancpl-relation*: $\neg full1\ R^{**}\ a\ b$

by (*auto simp: full1-def*)

lemma *not-full-rtrancpl-relation*: $\neg full\ R^{**}\ a\ b$

by (*auto simp: full-def*)

lemma *full1-trancpl-relation-full*:

$\langle full1\ R^{++}\ a\ b \longleftrightarrow full1\ R\ a\ b \rangle$

by (*metis converse-trancplE full1-def reflclp-trancpl rtrancplD rtrancpl-idemp rtrancpl-reflclp trancpl.r-into-trancpl trancpl-into-rtrancpl*)

lemma *full-trancpl-relation-full*:

$\langle full\ R^{++}\ a\ b \longleftrightarrow full\ R\ a\ b \rangle$

by (*metis full-unfold full1-trancpl-relation-full trancpl.r-into-trancpl trancplD*)

lemma *trancpl-full1-full1*:

$\langle (full1\ R)^{++}\ a\ b \longleftrightarrow full1\ R\ a\ b \rangle$

by (*metis (mono-tags) full1-def predicate2I trancpl.r-into-trancpl trancpl-idemp trancpl-mono-explicit trancpl-unfold-end*)

lemma *rtrancpl-full1-eq-or-full1*:

$\langle (full1\ R)^{**}\ a\ b \longleftrightarrow (a = b \vee full1\ R\ a\ b) \rangle$

unfolding *rtrancp-unfold trancp-full1-full1* **by** *simp*

lemma *no-step-full-iff-eq*:

$\langle \text{no-step } R \ S \implies \text{full } R \ S \ T \iff S = T \rangle$

unfolding *full-def*

by (*meson rtrancp.rtranc-refl rtrancpD trancpD*)

1.1.3 Well-Foundedness and Full Transitions

lemma *wf-exists-normal-form*:

assumes *wf*: $\langle \text{wf } \{(x, y). R \ y \ x\} \rangle$

shows $\langle \exists b. R^{**} \ a \ b \wedge \text{no-step } R \ b \rangle$

proof (*rule ccontr*)

assume $\langle \neg \text{?thesis} \rangle$

then have *H*: $\langle \bigwedge b. \neg R^{**} \ a \ b \vee \neg \text{no-step } R \ b \rangle$

by *blast*

define *F* **where** $\langle F = \text{rec-nat } a \ (\lambda i \ b. \text{SOME } c. R \ b \ c) \rangle$

have [*simp*]: $\langle F \ 0 = a \rangle$

unfolding *F-def* **by** *auto*

have [*simp*]: $\langle \bigwedge i. F \ (\text{Suc } i) = (\text{SOME } b. R \ (F \ i) \ b) \rangle$

unfolding *F-def* **by** *simp*

{ fix *i*

have $\langle \forall j < i. R \ (F \ j) \ (F \ (\text{Suc } j)) \rangle$

proof (*induction i*)

case *0*

then show *?case* **by** *auto*

next

case (*Suc i*)

then have $\langle R^{**} \ a \ (F \ i) \rangle$

by (*induction i*) *auto*

then have $\langle R \ (F \ i) \ (\text{SOME } b. R \ (F \ i) \ b) \rangle$

using *H* **by** (*simp add: someI-ex*)

then have $\langle \forall j < \text{Suc } i. R \ (F \ j) \ (F \ (\text{Suc } j)) \rangle$

using *H Suc* **by** (*simp add: less-Suc-eq*)

then show *?case* **by** *fast*

qed

}

then have $\langle \forall j. R \ (F \ j) \ (F \ (\text{Suc } j)) \rangle$ **by** *blast*

then show *False*

using *wf* **unfolding** *wfP-def wf-iff-no-infinite-down-chain* **by** *blast*

qed

lemma *wf-exists-normal-form-full*:

assumes *wf*: $\langle \text{wf } \{(x, y). R \ y \ x\} \rangle$

shows $\langle \exists b. \text{full } R \ a \ b \rangle$

using *wf-exists-normal-form[OF assms]* **unfolding** *full-def* **by** *blast*

1.1.4 More Well-Foundedness

A little list of theorems that could be useful, but are hidden:

- link between *wf* and infinite chains: theorems *wf-iff-no-infinite-down-chain* and *wf-no-infinite-down-chain*

lemma *wf-if-measure-in-wf*:

$\langle \text{wf } R \implies (\bigwedge a \ b. (a, b) \in S \implies (\nu \ a, \nu \ b) \in R) \implies \text{wf } S \rangle$

by (metis inv-image wfE-min wfI-min wf-inv-image)

lemma *wfP-if-measure*: fixes $f :: \langle 'a \Rightarrow \text{nat} \rangle$
 shows $\langle (\bigwedge x y. P x \implies g x y \implies f y < f x) \implies \text{wf } \{(y,x). P x \wedge g x y\} \rangle$
 apply (insert wf-measure[of f])
 apply (simp only: measure-def inv-image-def less-than-def less-eq)
 apply (erule wf-subset)
 apply auto
 done

lemma *wf-if-measure-f*:
 assumes $\langle \text{wf } r \rangle$
 shows $\langle \text{wf } \{(b, a). (f b, f a) \in r\} \rangle$
 using assms by (metis inv-image-def wf-inv-image)

lemma *wf-wf-if-measure'*:
 assumes $\langle \text{wf } r \rangle$ and $H: \langle \bigwedge x y. P x \implies g x y \implies (f y, f x) \in r \rangle$
 shows $\langle \text{wf } \{(y,x). P x \wedge g x y\} \rangle$
proof –
 have $\langle \text{wf } \{(b, a). (f b, f a) \in r\} \rangle$ using assms(1) *wf-if-measure-f* by auto
 then have $\langle \text{wf } \{(b, a). P a \wedge g a b \wedge (f b, f a) \in r\} \rangle$
 using wf-subset[of - $\langle \{(b, a). P a \wedge g a b \wedge (f b, f a) \in r\} \rangle$] by auto
 moreover have $\langle \{(b, a). P a \wedge g a b \wedge (f b, f a) \in r\} \subseteq \{(b, a). (f b, f a) \in r\} \rangle$ by auto
 moreover have $\langle \{(b, a). P a \wedge g a b \wedge (f b, f a) \in r\} = \{(b, a). P a \wedge g a b\} \rangle$ using H by auto
 ultimately show ?thesis using wf-subset by simp
qed

lemma *wf-lex-less*: $\langle \text{wf } (\text{lex less-than}) \rangle$
 by (auto simp: wf-less)

lemma *wfP-if-measure2*: fixes $f :: \langle 'a \Rightarrow \text{nat} \rangle$
 shows $\langle (\bigwedge x y. P x y \implies g x y \implies f x < f y) \implies \text{wf } \{(x,y). P x y \wedge g x y\} \rangle$
 apply (insert wf-measure[of f])
 apply (simp only: measure-def inv-image-def less-than-def less-eq)
 apply (erule wf-subset)
 apply auto
 done

lemma *lexord-on-finite-set-is-wf*:
 assumes
 $P\text{-finite}: \langle \bigwedge U. P U \longrightarrow U \in A \rangle$ and
 $\text{finite}: \langle \text{finite } A \rangle$ and
 $\text{wf}: \langle \text{wf } R \rangle$ and
 $\text{trans}: \langle \text{trans } R \rangle$
 shows $\langle \text{wf } \{(T, S). (P S \wedge P T) \wedge (T, S) \in \text{lexord } R\} \rangle$
proof (rule *wfP-if-measure2*)
 fix $T S$
 assume $P: \langle P S \wedge P T \rangle$ and
 $s\text{-le-}t: \langle (T, S) \in \text{lexord } R \rangle$
 let $?f = \langle \lambda S. \{U. (U, S) \in \text{lexord } R \wedge P U \wedge P S\} \rangle$
 have $\langle ?f T \subseteq ?f S \rangle$
 using $s\text{-le-}t$ P *lexord-trans* *trans* by auto
 moreover have $\langle T \in ?f S \rangle$
 using $s\text{-le-}t$ P by auto
 moreover have $\langle T \notin ?f T \rangle$
 using $s\text{-le-}t$ by (auto simp add: *lexord-irreflexive local.wf*)

ultimately have $\langle \{U. (U, T) \in \text{lexord } R \wedge P \ U \wedge P \ T\} \subset \{U. (U, S) \in \text{lexord } R \wedge P \ U \wedge P \ S\} \rangle$
 by *auto*
 moreover have $\langle \text{finite } \{U. (U, S) \in \text{lexord } R \wedge P \ U \wedge P \ S\} \rangle$
 using *finite* by (*metis* (*no-types*, *lifting*) *P-finite finite-subset mem-Collect-eq subsetI*)
 ultimately show $\langle \text{card } (?f \ T) < \text{card } (?f \ S) \rangle$ by (*simp add: psubset-card-mono*)
 qed

lemma *wf-fst-wf-pair*:
 assumes $\langle \text{wf } \{(M', M). R \ M' \ M\} \rangle$
 shows $\langle \text{wf } \{((M', N'), (M, N)). R \ M' \ M\} \rangle$
 proof –
 have $\langle \text{wf } (\{(M', M). R \ M' \ M\} <*\text{lex}* \ \{\}) \rangle$
 using *assms* by *auto*
 then show *?thesis*
 by (*rule wf-subset*) *auto*
 qed

lemma *wf-snd-wf-pair*:
 assumes $\langle \text{wf } \{(M', M). R \ M' \ M\} \rangle$
 shows $\langle \text{wf } \{((M', N'), (M, N)). R \ N' \ N\} \rangle$
 proof –
 have *wf*: $\langle \text{wf } \{((M', N'), (M, N)). R \ M' \ M\} \rangle$
 using *assms wf-fst-wf-pair* by *auto*
 then have *wf*: $\langle \bigwedge P. (\forall x. (\forall y. (y, x) \in \{((M', N'), M, N). R \ M' \ M\} \longrightarrow P \ y) \longrightarrow P \ x) \implies \text{All } P \rangle$
 unfolding *wf-def* by *auto*
 show *?thesis*
 unfolding *wf-def*
 proof (*intro allI impI*)
 fix *P* :: $\langle 'c \times 'a \Rightarrow \text{bool} \rangle$ and *x* :: $\langle 'c \times 'a \rangle$
 assume *H*: $\langle \forall x. (\forall y. (y, x) \in \{((M', N'), M, y). R \ N' \ y\} \longrightarrow P \ y) \longrightarrow P \ x \rangle$
 obtain *a b* where *x*: $\langle x = (a, b) \rangle$ by (*cases x*)
 have *P*: $\langle P \ x = (P \circ (\lambda(a, b). (b, a))) (b, a) \rangle$
 unfolding *x* by *auto*
 show $\langle P \ x \rangle$
 using *wf*[*of* $\langle P \circ (\lambda(a, b). (b, a)) \rangle$] apply *rule*
 using *H* apply *simp*
 unfolding *P* by *blast*
 qed
 qed

lemma *wf-if-measure-f-notation2*:
 assumes $\langle \text{wf } r \rangle$
 shows $\langle \text{wf } \{(b, h \ a) \mid b \ a. (f \ b, f \ (h \ a)) \in r\} \rangle$
 apply (*rule wf-subset*)
 using *wf-if-measure-f[OF assms, of f]* by *auto*

lemma *wf-wf-if-measure'-notation2*:
 assumes $\langle \text{wf } r \rangle$ and *H*: $\langle \bigwedge x \ y. P \ x \implies g \ x \ y \implies (f \ y, f \ (h \ x)) \in r \rangle$
 shows $\langle \text{wf } \{(y, h \ x) \mid y \ x. P \ x \wedge g \ x \ y\} \rangle$
 proof –
 have $\langle \text{wf } \{(b, h \ a) \mid b \ a. (f \ b, f \ (h \ a)) \in r\} \rangle$ using *assms(1) wf-if-measure-f-notation2* by *auto*
 then have $\langle \text{wf } \{(b, h \ a) \mid b \ a. P \ a \wedge g \ a \ b \wedge (f \ b, f \ (h \ a)) \in r\} \rangle$
 using *wf-subset*[*of* - $\langle \{(b, h \ a) \mid b \ a. P \ a \wedge g \ a \ b \wedge (f \ b, f \ (h \ a)) \in r\} \rangle$] by *auto*
 moreover have $\langle \{(b, h \ a) \mid b \ a. P \ a \wedge g \ a \ b \wedge (f \ b, f \ (h \ a)) \in r\} \subseteq \{(b, h \ a) \mid b \ a. (f \ b, f \ (h \ a)) \in r\} \rangle$ by *auto*

moreover have $\langle \{(b, h\ a) \mid b\ a.\ P\ a \wedge g\ a\ b \wedge (f\ b, f\ (h\ a)) \in r\} = \{(b, h\ a) \mid b\ a.\ P\ a \wedge g\ a\ b\} \rangle$
using H **by** $auto$
ultimately show $?thesis$ **using** $wf\text{-}subset$ **by** $simp$
qed

lemma $power\text{-}ex\text{-}decomp$:
assumes $\langle (R \text{ } \widehat{\sim} \text{ } n)\ S\ T \rangle$
shows
 $\langle \exists f. f\ 0 = S \wedge f\ n = T \wedge (\forall i. i < n \longrightarrow R\ (f\ i)\ (f\ (Suc\ i))) \rangle$
using $assms$
proof ($induction\ n\ arbitrary: T$)
case 0
then show $\langle ?case \rangle$ **by** $auto$
next
case $(Suc\ n)$ **note** $IH = this(1)$ **and** $R = this(2)$
from R **obtain** T' **where**
 ST : $\langle (R \text{ } \widehat{\sim} \text{ } n)\ S\ T' \rangle$ **and**
 $T'T$: $\langle R\ T'\ T \rangle$
by $auto$
obtain f **where**
 $[simp]$: $\langle f\ 0 = S \rangle$ **and**
 $[simp]$: $\langle f\ n = T' \rangle$ **and**
 H : $\langle \bigwedge i. i < n \implies R\ (f\ i)\ (f\ (Suc\ i)) \rangle$
using $IH[OF\ ST]$ **by** $fast$
let $?f = \langle f\ (Suc\ n := T) \rangle$
show $?case$
by ($rule\ exI[of\ -\ ?f]$)
 $(use\ H\ ST\ T'T\ in\ auto)$
qed

The following lemma gives a bound on the maximal number of transitions of a sequence that is well-founded under the lexicographic ordering $lexn$ on natural numbers.

lemma $lexn\text{-}number\text{-}of\text{-}transition$:
assumes
 le : $\langle \bigwedge i. i < n \implies ((f\ (Suc\ i)), (f\ i)) \in lexn\ less\text{-}than\ m \rangle$ **and**
 $upper$: $\langle \bigwedge i\ j. i \leq n \implies j < m \implies (f\ i) ! j \in \{0..<k\} \rangle$ **and**
 $\langle finite\ A \rangle$ **and**
 k : $\langle k > 1 \rangle$
shows $\langle n < k \wedge Suc\ m \rangle$
proof –
define r **where**
 $\langle r\ x = zip\ x\ (map\ (\lambda i. k \wedge (length\ x - i))\ [0..<length\ x]) \rangle$ **for** $x :: \langle nat\ list \rangle$

define s **where**
 $\langle s\ x = foldr\ (\lambda a\ b. a + b)\ (map\ (\lambda(a, b). a * b)\ x)\ 0 \rangle$ **for** $x :: \langle (nat \times nat)\ list \rangle$

have $[simp]$: $\langle r\ [] = [] \rangle$ $\langle s\ [] = 0 \rangle$
by ($auto\ simp: r\text{-}def\ s\text{-}def$)

have upt' : $\langle m > 0 \implies [0..<m] = 0 \# map\ Suc\ [0..<m-1] \rangle$ **for** m
by ($auto\ simp: map\text{-}Suc\text{-}upt\ upt\text{-}conv\ Cons$)

have upt'' : $\langle m > 0 \implies [0..<m] = [0..<m-1] @ [m-1] \rangle$ **for** m
by ($cases\ m$) ($auto\ simp:$)

have $Cons$: $\langle r\ (x \# xs) = (x, k \wedge (Suc\ (length\ xs))) \# (r\ xs) \rangle$ **for** $x\ xs$

```

unfolding r-def
apply (subst upt')
apply (clarsimp simp add: upt'' comp-def nth-append Suc-diff-le simp flip: zip-map2)
apply (clarsimp simp add: upt'' comp-def nth-append Suc-diff-le simp flip: zip-map2)
done

have [simp]:  $\langle s (ab \# xs) = \text{fst } ab * \text{snd } ab + s \ xs \rangle$  for  $ab \ xs$ 
  unfolding s-def by (cases ab) auto

have le2:  $\langle (\forall a \in \text{set } b. a < k) \implies (k \wedge (\text{Suc } (\text{length } b))) > s ((r \ b)) \rangle$  for  $b$ 
  apply (induction b arbitrary: f)
  using k apply (auto simp: Cons)
  apply (rule order.strict-trans1)
  apply (rule-tac j =  $\langle (k - 1) * k * k \wedge \text{length } b \rangle$  in Nat.add-le-mono1)
  subgoal for  $a \ b$ 
    by auto
  apply (rule order.strict-trans2)
  apply (rule-tac b =  $\langle (k - 1) * k * k \wedge \text{length } b \rangle$  and  $d = \langle (k * k \wedge \text{length } b) \rangle$  in add-le-less-mono)
  apply (auto simp: mult.assoc comm-semiring-1-class.semiring-normalization-rules(2))
  done

have  $\langle s (r (f (\text{Suc } i))) < s (r (f i)) \rangle$  if  $\langle i < n \rangle$  for  $i$ 
proof -
  have i-n:  $\langle \text{Suc } i \leq n \rangle \langle i \leq n \rangle$ 
    using that by auto
  have length:  $\langle \text{length } (f i) = m \rangle \langle \text{length } (f (\text{Suc } i)) = m \rangle$ 
    using le[OF that] by (auto dest: lexn-length)
  define xs and ys where  $\langle xs = f i \rangle$  and  $\langle ys = f (\text{Suc } i) \rangle$ 

  show ?thesis
    using le[OF that] upper[OF i-n(2)] upper[OF i-n(1)] length Cons
    unfolding xs-def[symmetric] ys-def[symmetric]
  proof (induction m arbitrary: xs ys)
    case 0 then show ?case by auto
  next
    case (Suc m) note IH = this(1) and H = this(2) and p = this(3-)
    have IH:  $\langle (tl \ ys, tl \ xs) \in \text{lexn less-than } m \implies s (r (tl \ ys)) < s (r (tl \ xs)) \rangle$ 
      apply (rule IH)
      subgoal by auto
      subgoal for  $i$  using p(1)[of  $\langle \text{Suc } i \rangle$ ] p by (cases xs; auto)
      subgoal for  $i$  using p(2)[of  $\langle \text{Suc } i \rangle$ ] p by (cases ys; auto)
      subgoal using p by (cases xs) auto
      subgoal using p by auto
      subgoal using p by auto
    done
    have  $\langle s (r (tl \ ys)) < k \wedge (\text{Suc } (\text{length } (tl \ ys))) \rangle$ 
      apply (rule le2)
      unfolding all-set-conv-all-nth
      using p by (simp add: nth-tl)
    then have  $\langle ab * (k * k \wedge \text{length } (tl \ ys)) + s (r (tl \ ys)) <$ 
       $ab * (k * k \wedge \text{length } (tl \ ys)) + (k * k \wedge \text{length } (tl \ ys)) \rangle$  for  $ab$ 
      by auto
    also have  $\langle \dots ab \leq (ab + 1) * (k * k \wedge \text{length } (tl \ ys)) \rangle$  for  $ab$ 
      by auto
    finally have less:  $\langle ab < ac \implies ab * (k * k \wedge \text{length } (tl \ ys)) + s (r (tl \ ys)) <$ 
       $ac * (k * k \wedge \text{length } (tl \ ys)) \rangle$  for  $ab \ ac$ 

```

```

proof –
  assume  $a1: \bigwedge ab. ab * (k * k \wedge \text{length } (tl \ ys)) + s \ (r \ (tl \ ys)) <$ 
     $(ab + 1) * (k * k \wedge \text{length } (tl \ ys))$ 
  assume  $ab < ac$ 
  then have  $\neg ac * (k * k \wedge \text{length } (tl \ ys)) < (ab + 1) * (k * k \wedge \text{length } (tl \ ys))$ 
    by (metis (no-types) One-nat-def Suc-leI add.right-neutral add-Suc-right
      mult-less-cancel2 not-less)
  then show ?thesis
    using  $a1$  by (meson le-less-trans not-less)
qed

have  $\langle ab < ac \implies$ 
   $ab * (k * k \wedge \text{length } (tl \ ys)) + s \ (r \ (tl \ ys))$ 
   $< ac * (k * k \wedge \text{length } (tl \ xs)) + s \ (r \ (tl \ xs)) \rangle$  for  $ab \ ac$ 
  using less[of ab ac] p by auto
then show ?case
  apply (cases xs; cases ys)
  using IH H p(3-5) by auto
qed
qed
then have  $\langle i \leq n \implies s \ (r \ (f \ i)) + i \leq s \ (r \ (f \ 0)) \rangle$  for  $i$ 
  apply (induction i)
  subgoal by auto
  subgoal premises  $p$  for  $i$ 
    using  $p(3)[\text{of } \langle i-1 \rangle]$   $p(1,2)$ 
    apply auto
    by (meson Nat.le-diff-conv2 Suc-leI Suc-le-lessD add-leD2 less-diff-conv less-le-trans p(3))
  done
from this[of n] show  $\langle n < k \wedge \text{Suc } m \rangle$ 
  using le2[of f 0] upper[of 0] k
  using le[of 0] apply (cases n = 0)
  by (auto dest!: lexn-length simp: all-set-conv-all-nth eq-commute[of - m])
qed

end
theory WB-List-More
  imports Nested-Multisets-Ordinals.Multiset-More HOL-Library.Finite-Map
    HOL-Eisbach.Eisbach
    HOL-Eisbach.Eisbach-Tools
begin

```

This theory contains various lemmas that have been used in the formalisation. Some of them could probably be moved to the Isabelle distribution or *Nested-Multisets-Ordinals.Multiset-More*.

More Sledgehammer parameters

1.2 Various Lemmas

1.2.1 Not-Related to Refinement or lists

Unlike *clarify*/*auto*/*simp*, this does not split tuple of the form $\exists T. P \ T$ in the assumption. After calling it, as the variable are not quantified anymore, the *simproc* does not trigger, allowing to safely call *auto*/*simp*/...

method *normalize-goal* =

```

(match premises in
  J[thin]:  $\langle \exists x. \neg \Rightarrow \langle \text{rule } exE[OF J] \rangle$ 
| J[thin]:  $\langle \neg \wedge \neg \Rightarrow \langle \text{rule } conjE[OF J] \rangle$ 
)

```

Close to the theorem *nat-less-induct* ($(\bigwedge n. \forall m < n. ?P\ m \implies ?P\ n) \implies ?P\ ?n$), but with a separation between the zero and non-zero case.

lemma *nat-less-induct-case*[case-names 0 Suc]:

```

assumes
   $\langle P\ 0 \rangle$  and
   $\langle \bigwedge n. (\forall m < Suc\ n. P\ m) \implies P\ (Suc\ n) \rangle$ 
shows  $\langle P\ n \rangle$ 
apply (induction rule: nat-less-induct)
by (rename-tac n, case-tac n) (auto intro: assms)

```

This is only proved in simple cases by auto. In assumptions, nothing happens, and the theorem *if-split-asm* can blow up goals (because of other if-expressions either in the context or as simplification rules).

lemma *if-0-1-ge-0[simp]*:

```

 $\langle 0 < (if\ P\ then\ a\ else\ (0::nat)) \longleftrightarrow P \wedge 0 < a \rangle$ 
by auto

```

lemma *bex-lessI*: $P\ j \implies j < n \implies \exists j < n. P\ j$

by auto

lemma *bex-gtI*: $P\ j \implies j > n \implies \exists j > n. P\ j$

by auto

lemma *bex-geI*: $P\ j \implies j \geq n \implies \exists j \geq n. P\ j$

by auto

lemma *bex-leI*: $P\ j \implies j \leq n \implies \exists j \leq n. P\ j$

by auto

Bounded function have not yet been defined in Isabelle.

definition *bounded* :: $\langle 'a \Rightarrow 'b::ord \rangle \Rightarrow bool$ **where**

$\langle bounded\ f \longleftrightarrow (\exists b. \forall n. f\ n \leq b) \rangle$

abbreviation *unbounded* :: $\langle 'a \Rightarrow 'b::ord \rangle \Rightarrow bool$ **where**

$\langle unbounded\ f \equiv \neg\ bounded\ f \rangle$

lemma *not-bounded-nat-exists-larger*:

```

fixes f ::  $\langle nat \Rightarrow nat \rangle$ 
assumes unbound:  $\langle unbounded\ f \rangle$ 
shows  $\langle \exists n. f\ n > m \wedge n > n_0 \rangle$ 

```

proof (rule ccontr)

assume H: $\langle \neg\ ?thesis \rangle$

have $\langle finite\ \{f\ n \mid n. n \leq n_0\} \rangle$

by auto

have $\langle \bigwedge n. f\ n \leq Max\ (\{f\ n \mid n. n \leq n_0\} \cup \{m\}) \rangle$

apply (case-tac $\langle n \leq n_0 \rangle$)

apply (metis (mono-tags, lifting) Max-ge Un-insert-right $\langle finite\ \{f\ n \mid n. n \leq n_0\} \rangle$
finite-insert insertCI mem-Collect-eq sup-bot.right-neutral)

by (metis (no-types, lifting) H Max-less-iff Un-insert-right $\langle finite\ \{f\ n \mid n. n \leq n_0\} \rangle$
finite-insert insertI1 insert-not-empty leI sup-bot.right-neutral)

```

then show False
  using unbound unfolding bounded-def by auto
qed

```

A function is bounded iff its product with a non-zero constant is bounded. The non-zero condition is needed only for the reverse implication (see for example $k = 0$ and $f = (\lambda i. i)$ for a counter-example).

```

lemma bounded-const-product:
  fixes  $k :: \text{nat}$  and  $f :: \langle \text{nat} \Rightarrow \text{nat} \rangle$ 
  assumes  $\langle k > 0 \rangle$ 
  shows  $\langle \text{bounded } f \longleftrightarrow \text{bounded } (\lambda i. k * f i) \rangle$ 
  unfolding bounded-def apply (rule iffI)
  using mult-le-mono2 apply blast
  by (metis Suc-leI add.right-neutral assms mult.commute mult-0-right mult-Suc-right mult-le-mono
      nat-mult-le-cancel1)

```

```

lemma bounded-const-add:
  fixes  $k :: \text{nat}$  and  $f :: \langle \text{nat} \Rightarrow \text{nat} \rangle$ 
  assumes  $\langle k > 0 \rangle$ 
  shows  $\langle \text{bounded } f \longleftrightarrow \text{bounded } (\lambda i. k + f i) \rangle$ 
  unfolding bounded-def apply (rule iffI)
  using nat-add-left-cancel-le apply blast
  using add-leE by blast

```

This lemma is not used, but here to show that property that can be expected from *bounded* holds.

```

lemma bounded-finite-linorder:
  fixes  $f :: \langle 'a::\text{finite} \Rightarrow 'b :: \{\text{linorder}\} \rangle$ 
  shows  $\langle \text{bounded } f \rangle$ 
proof -
  have  $\langle \text{finite } (f \text{ ' UNIV}) \rangle$ 
  by simp
  then have  $\langle \bigwedge x. f x \leq \text{Max } (f \text{ ' UNIV}) \rangle$ 
  by (auto intro: Max-ge)
  then show ?thesis
  unfolding bounded-def by blast
qed

```

1.3 More Lists

1.3.1 set, nth, tl

```

lemma ex-geI:  $\langle P n \implies n \geq m \implies \exists n \geq m. P n \rangle$ 
  by auto

```

```

lemma Ball-atLeastLessThan-iff:  $\langle (\forall L \in \{a..<b\}. P L) \longleftrightarrow (\forall L. L \geq a \wedge L < b \longrightarrow P L) \rangle$ 
  unfolding set-nths by auto

```

```

lemma nth-in-set-tl:  $\langle i > 0 \implies i < \text{length } xs \implies xs ! i \in \text{set } (\text{tl } xs) \rangle$ 
  by (cases xs) auto

```

```

lemma tl-drop-def:  $\langle \text{tl } N = \text{drop } 1 N \rangle$ 
  by (cases N) auto

```

```

lemma in-set-remove1D:

```

$\langle a \in \text{set } (\text{remove1 } x \text{ } xs) \implies a \in \text{set } xs \rangle$
by (*meson notin-set-remove1*)

lemma *take-length-takeWhile-eq-takeWhile*:
 $\langle \text{take } (\text{length } (\text{takeWhile } P \text{ } xs)) \text{ } xs = \text{takeWhile } P \text{ } xs \rangle$
by (*induction xs*) *auto*

lemma *fold-cons-replicate*: $\langle \text{fold } (\lambda \text{ } xs. a \# xs) \text{ } [0..<n] \text{ } xs = \text{replicate } n \text{ } a @ xs \rangle$
by (*induction n*) *auto*

lemma *Collect-minus-single-Collect*: $\langle \{x. P \text{ } x\} - \{a\} = \{x. P \text{ } x \wedge x \neq a\} \rangle$
by *auto*

lemma *in-set-image-subsetD*: $\langle f ' A \subseteq B \implies x \in A \implies f \text{ } x \in B \rangle$
by *blast*

lemma *mset-tl*:
 $\langle \text{mset } (\text{tl } xs) = \text{remove1-mset } (\text{hd } xs) \text{ } (\text{mset } xs) \rangle$
by (*cases xs*) *auto*

lemma *hd-list-update-If*:
 $\langle \text{outl}' \neq [] \implies \text{hd } (\text{outl}'[i := w]) = (\text{if } i = 0 \text{ then } w \text{ else } \text{hd } \text{outl}') \rangle$
by (*cases outl'*) (*auto split: nat.splits*)

lemma *list-update-id'*:
 $\langle x = xs ! i \implies xs[i := x] = xs \rangle$
by *auto*

This lemma is not general enough to move to Isabelle, but might be interesting in other cases.

lemma *set-Collect-Pair-to-fst-snd*:
 $\langle \{(a, b), (a', b')\}. P \text{ } a \text{ } b \text{ } a' \text{ } b'\} = \{(e, f). P \text{ } (\text{fst } e) \text{ } (\text{snd } e) \text{ } (\text{fst } f) \text{ } (\text{snd } f)\} \rangle$
by *auto*

lemma *butlast-Nil-iff*: $\langle \text{butlast } xs = [] \longleftrightarrow \text{length } xs = 1 \vee \text{length } xs = 0 \rangle$
by (*cases xs*) *auto*

lemma *Set-remove-diff-insert*: $\langle a \in B - A \implies B - \text{Set.remove } a \text{ } A = \text{insert } a \text{ } (B - A) \rangle$
by *auto*

lemma *Set-insert-diff-remove*: $\langle B - \text{insert } a \text{ } A = \text{Set.remove } a \text{ } (B - A) \rangle$
by *auto*

lemma *Set-remove-insert*: $\langle a \notin A' \implies \text{Set.remove } a \text{ } (\text{insert } a \text{ } A') = A' \rangle$
by (*auto simp: Set.remove-def*)

lemma *diff-eq-insertD*:
 $\langle B - A = \text{insert } a \text{ } A' \implies a \in B \rangle$
by *auto*

lemma *in-set-tlD*: $\langle x \in \text{set } (\text{tl } xs) \implies x \in \text{set } xs \rangle$
by (*cases xs*) *auto*

This lemma is only useful if *set xs* can be simplified (which also means that this simp-rule should not be used...)

lemma (*in -*) *in-list-in-setD*: $\langle xs = \text{it } @ x \# \sigma \implies x \in \text{set } xs \rangle$

by auto

lemma *Collect-eq-comp'*: $\langle \{(x, y). P\ x\ y\} \ O\ \{(c, a). c = f\ a\} = \{(x, a). P\ x\ (f\ a)\} \rangle$
by auto

lemma (*in* $-$) *filter-disj-eq*:
 $\langle \{x \in A. P\ x \vee Q\ x\} = \{x \in A. P\ x\} \cup \{x \in A. Q\ x\} \rangle$
by auto

lemma *zip-cong*:
 $\langle (\bigwedge i. i < \min(\text{length}\ xs)\ (\text{length}\ ys) \implies (xs\ !\ i, ys\ !\ i) = (xs'\ !\ i, ys'\ !\ i)) \implies$
 $\text{length}\ xs = \text{length}\ xs' \implies \text{length}\ ys = \text{length}\ ys' \implies \text{zip}\ xs\ ys = \text{zip}\ xs'\ ys' \rangle$

proof (*induction* xs arbitrary: $xs'\ ys'\ ys$)

case *Nil*

then show ?case by auto

next

case (*Cons* $x\ xs\ xs'\ ys'\ ys$) note $IH = \text{this}(1)$ and $eq = \text{this}(2)$ and $p = \text{this}(3-)$

thm IH

have $\langle \text{zip}\ xs\ (tl\ ys) = \text{zip}\ (tl\ xs')\ (tl\ ys') \rangle$ for i

apply (rule IH)

subgoal for i

using $p\ eq[of\ \langle \text{Suc}\ i \rangle]$ by (auto simp: $nth\ tl$)

subgoal using p by auto

subgoal using p by auto

done

moreover have $\langle hd\ xs' = x \rangle \langle hd\ ys = hd\ ys' \rangle$ if $\langle ys \neq [] \rangle$

using $eq[of\ 0]$ that $p[symmetric]$ apply (auto simp: $hd\ conv\ nth$)

apply (subst $hd\ conv\ nth$)

apply auto

apply (subst $hd\ conv\ nth$)

apply auto

done

ultimately show ?case

using p by (cases xs' ; cases ys' ; cases ys)

auto

qed

lemma *zip-cong2*:
 $\langle (\bigwedge i. i < \min(\text{length}\ xs)\ (\text{length}\ ys) \implies (xs\ !\ i, ys\ !\ i) = (xs'\ !\ i, ys'\ !\ i)) \implies$
 $\text{length}\ xs = \text{length}\ xs' \implies \text{length}\ ys \leq \text{length}\ ys' \implies \text{length}\ ys \geq \text{length}\ xs \implies$
 $\text{zip}\ xs\ ys = \text{zip}\ xs'\ ys' \rangle$

proof (*induction* xs arbitrary: $xs'\ ys'\ ys$)

case *Nil*

then show ?case by auto

next

case (*Cons* $x\ xs\ xs'\ ys'\ ys$) note $IH = \text{this}(1)$ and $eq = \text{this}(2)$ and $p = \text{this}(3-)$

have $\langle \text{zip}\ xs\ (tl\ ys) = \text{zip}\ (tl\ xs')\ (tl\ ys') \rangle$ for i

apply (rule IH)

subgoal for i

using $p\ eq[of\ \langle \text{Suc}\ i \rangle]$ by (auto simp: $nth\ tl$)

subgoal using p by auto

subgoal using p by auto

subgoal using p by auto

done

moreover have $\langle hd\ xs' = x \rangle \langle hd\ ys = hd\ ys' \rangle$ if $\langle ys \neq [] \rangle$

```

    using eq[of 0] that p apply (auto simp: hd-conv-nth)
    apply (subst hd-conv-nth)
    apply auto
    apply (subst hd-conv-nth)
    apply auto
    done
ultimately show ?case
  using p by (cases xs'; cases ys'; cases ys)
    auto
qed

```

1.3.2 List Updates

```

lemma tl-update-swap:
   $\langle i \geq 1 \implies \text{tl } (N[i := C]) = (\text{tl } N)[i-1 := C] \rangle$ 
  by (auto simp: drop-Suc[of 0, symmetric, simplified] drop-update-swap)

```

```

lemma tl-update-0[simp]:  $\langle \text{tl } (N[0 := x]) = \text{tl } N \rangle$ 
  by (cases N) auto

```

```

declare nth-list-update[simp]

```

This a version of $?i < \text{length } ?xs \implies ?xs[?i := ?x] ! ?j = (\text{if } ?i = ?j \text{ then } ?x \text{ else } ?xs ! ?j)$ with a different condition (j instead of i). This is more useful in some cases.

```

lemma nth-list-update-le[simp]:
   $j < \text{length } xs \implies (xs[i := x])!j = (\text{if } i = j \text{ then } x \text{ else } xs!j)$ 
  by (induct xs arbitrary: i j) (auto simp add: nth-Cons split: nat.split)

```

1.3.3 Take and drop

```

lemma take-2-if:
   $\langle \text{take } 2 \ C = (\text{if } C = [] \text{ then } [] \text{ else if } \text{length } C = 1 \text{ then } [\text{hd } C] \text{ else } [C!0, C!1]) \rangle$ 
  by (cases C; cases  $\langle \text{tl } C \rangle$ ) auto

```

```

lemma in-set-take-conv-nth:
   $\langle x \in \text{set } (\text{take } n \ xs) \longleftrightarrow (\exists m < \min n \ (\text{length } xs). xs ! m = x) \rangle$ 
  by (metis in-set-conv-nth length-take min.commute min.strict-boundedE nth-take)

```

```

lemma in-set-dropI:
   $\langle m < \text{length } xs \implies m \geq n \implies xs ! m \in \text{set } (\text{drop } n \ xs) \rangle$ 
  unfolding in-set-conv-nth
  by (rule exI[of -  $\langle m - n \rangle$ ]) auto

```

```

lemma in-set-drop-conv-nth:
   $\langle x \in \text{set } (\text{drop } n \ xs) \longleftrightarrow (\exists m \geq n. m < \text{length } xs \wedge xs ! m = x) \rangle$ 
  apply (rule iffI)
  subgoal
    apply (subst (asm) in-set-conv-nth)
    apply clarsimp
    apply (rule-tac x =  $\langle n+i \rangle$  in exI)
    apply (auto)
  done
  subgoal
    by (auto intro: in-set-dropI)
  done

```

Taken from `~/src/HOL/Word/Word.thy`

lemma *atd-lem*: $\langle \text{take } n \text{ } xs = t \implies \text{drop } n \text{ } xs = d \implies xs = t @ d \rangle$
by (*auto intro: append-take-drop-id [symmetric]*)

lemma *drop-take-drop-drop*:
 $\langle j \geq i \implies \text{drop } i \text{ } xs = \text{take } (j - i) (\text{drop } i \text{ } xs) @ \text{drop } j \text{ } xs \rangle$
apply (*induction* $\langle j - i \rangle$ *arbitrary: j i*)
subgoal by *auto*
subgoal by (*auto simp add: atd-lem*)
done

lemma *in-set-conv-iff*:
 $\langle x \in \text{set } (\text{take } n \text{ } xs) \longleftrightarrow (\exists i < n. i < \text{length } xs \wedge xs ! i = x) \rangle$
apply (*induction n*)
subgoal by *auto*
subgoal for *n*
apply (*cases* $\langle \text{Suc } n < \text{length } xs \rangle$)
subgoal by (*auto simp: take-Suc-conv-app-nth less-Suc-eq dest: in-set-takeD*)
subgoal
apply (*cases* $\langle n < \text{length } xs \rangle$)
subgoal
apply (*auto simp: in-set-conv-nth*)
by (*rule-tac x=i in exI; auto; fail*)+
subgoal
apply (*auto simp: take-Suc-conv-app-nth dest: in-set-takeD*)
by (*rule-tac x=i in exI; auto; fail*)+
done
done
done

lemma *distinct-in-set-take-iff*:
 $\langle \text{distinct } D \implies b < \text{length } D \implies D ! b \in \text{set } (\text{take } a \text{ } D) \longleftrightarrow b < a \rangle$
apply (*induction a arbitrary: b*)
subgoal by *simp*
subgoal for *a*
by (*cases* $\langle \text{Suc } a < \text{length } D \rangle$)
(auto simp: take-Suc-conv-app-nth nth-eq-iff-index-eq)
done

lemma *in-set-distinct-take-drop-iff*:
assumes
 $\langle \text{distinct } D \rangle$ **and**
 $\langle b < \text{length } D \rangle$
shows $\langle D ! b \in \text{set } (\text{take } (a - \text{init}) (\text{drop } \text{init } D)) \longleftrightarrow (\text{init} \leq b \wedge b < a) \rangle$
using *assms* **apply** (*auto 5 5 simp: distinct-in-set-take-iff in-set-conv-iff*
nth-eq-iff-index-eq dest: in-set-takeD)
by (*metis add-diff-cancel-left' diff-less-mono le-iff-add*)

1.3.4 Replicate

lemma *list-eq-replicate-iff-nempty*:
 $\langle n > 0 \implies xs = \text{replicate } n \text{ } x \longleftrightarrow n = \text{length } xs \wedge \text{set } xs = \{x\} \rangle$
by (*metis length-replicate neq0-conv replicate-length-same set-replicate singletonD*)

lemma *list-eq-replicate-iff*:
 $\langle xs = \text{replicate } n \text{ } x \longleftrightarrow (n = 0 \wedge xs = []) \vee (n = \text{length } xs \wedge \text{set } xs = \{x\}) \rangle$

by (cases n) (auto simp: list-eq-replicate-iff-nempty simp del: replicate.simps)

1.3.5 List intervals (upt)

The simplification rules are not very handy, because theorem *upt.simps* (2) (i.e. $[?i..<Suc\ ?j] = (if\ ?i \leq\ ?j\ then\ [?i..<?j]\ @\ [?j]\ else\ [])$) leads to a case distinction, that we usually do not want if the condition is not already in the context.

lemma *upt-Suc-le-append*: $\langle \neg i \leq j \implies [i..<Suc\ j] = [] \rangle$
by auto

lemmas *upt.simps[simp]* = *upt-Suc-append upt-Suc-le-append*

declare *upt.simps*(2)[*simp del*]

The counterpart for this lemma when $n - m < i$ is theorem *take-all*. It is close to theorem $?i + ?m \leq ?n \implies take\ ?m\ [?i..<?n] = [?i..<?i + ?m]$, but seems more general.

lemma *take-upt-bound-minus[simp]*:
assumes $\langle i \leq n - m \rangle$
shows $\langle take\ i\ [m..<n] = [m..<m+i] \rangle$
using *assms* by (induction i) auto

lemma *append-cons-eq-upt*:
assumes $\langle A @ B = [m..<n] \rangle$
shows $\langle A = [m..<m+length\ A] \rangle$ and $\langle B = [m + length\ A..<n] \rangle$
proof –
have $\langle take\ (length\ A)\ (A @ B) = A \rangle$ by auto
moreover {
 have $\langle length\ A \leq n - m \rangle$ using *assms* linear calculation by fastforce
 then have $\langle take\ (length\ A)\ [m..<n] = [m..<m+length\ A] \rangle$ by auto }
ultimately show $\langle A = [m..<m+length\ A] \rangle$ using *assms* by auto
show $\langle B = [m + length\ A..<n] \rangle$ using *assms* by (metis *append-eq-conv-conj drop-upt*)
qed

The converse of theorem *append-cons-eq-upt* does not hold, for example if @ term $B:: nat\ list$ is empty and A is $[0::'a]$:

lemma $\langle A @ B = [m..<n] \longleftrightarrow A = [m..<m+length\ A] \wedge B = [m + length\ A..<n] \rangle$
oops

A more restrictive version holds:

lemma $\langle B \neq [] \implies A @ B = [m..<n] \longleftrightarrow A = [m..<m+length\ A] \wedge B = [m + length\ A..<n] \rangle$
(is $\langle ?P \implies ?A = ?B \rangle$)

proof
assume $?A$ then show $?B$ by (auto simp add: *append-cons-eq-upt*)
next
assume $?P$ and $?B$
then show $?A$ using *append-eq-conv-conj* by fastforce
qed

lemma *append-cons-eq-upt-length-i*:
assumes $\langle A @ i \# B = [m..<n] \rangle$
shows $\langle A = [m..<i] \rangle$
proof –
have $\langle A = [m..<m + length\ A] \rangle$ using *assms* *append-cons-eq-upt* by auto
have $\langle (A @ i \# B) ! (length\ A) = i \rangle$ by auto

moreover have $\langle n - m = \text{length } (A @ i \# B) \rangle$
using *assms length-upt by presburger*
then have $\langle [m..<n] ! (\text{length } A) = m + \text{length } A \rangle$ **by** *simp*
ultimately have $\langle i = m + \text{length } A \rangle$ **using** *assms by auto*
then show *?thesis* **using** $\langle A = [m ..< m + \text{length } A] \rangle$ **by** *auto*
qed

lemma *append-cons-eq-upt-length:*
assumes $\langle A @ i \# B = [m..<n] \rangle$
shows $\langle \text{length } A = i - m \rangle$
using *assms*
proof (*induction A arbitrary: m*)
case *Nil*
then show *?case* **by** (*metis append-Nil diff-is-0-eq list.size(3) order-refl upt-eq-Cons-conv*)
next
case (*Cons a A*)
then have $A: \langle A @ i \# B = [m + 1..<n] \rangle$ **by** (*metis append-Cons upt-eq-Cons-conv*)
then have $\langle m < i \rangle$ **by** (*metis Cons.premis append-cons-eq-upt-length-i upt-eq-Cons-conv*)
with *Cons.IH[OF A]* **show** *?case* **by** *auto*
qed

lemma *append-cons-eq-upt-length-i-end:*
assumes $\langle A @ i \# B = [m..<n] \rangle$
shows $\langle B = [\text{Suc } i ..<n] \rangle$
proof –
have $\langle B = [\text{Suc } m + \text{length } A..<n] \rangle$ **using** *assms append-cons-eq-upt[of $\langle A @ [i] \rangle B m n$ by auto]*
have $\langle (A @ i \# B) ! (\text{length } A) = i \rangle$ **by** *auto*
moreover have $\langle n - m = \text{length } (A @ i \# B) \rangle$
using *assms length-upt by auto*
then have $\langle [m..<n] ! (\text{length } A) = m + \text{length } A \rangle$ **by** *simp*
ultimately have $\langle i = m + \text{length } A \rangle$ **using** *assms by auto*
then show *?thesis* **using** $\langle B = [\text{Suc } m + \text{length } A..<n] \rangle$ **by** *auto*
qed

lemma *Max-n-upt:* $\langle \text{Max } (\text{insert } 0 \{ \text{Suc } 0..<n \}) = n - \text{Suc } 0 \rangle$
proof (*induct n*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*) **note** *IH = this*
have $i: \langle \text{insert } 0 \{ \text{Suc } 0..<\text{Suc } n \} = \text{insert } 0 \{ \text{Suc } 0..<n \} \cup \{n\} \rangle$ **by** *auto*
show *?case* **using** *IH unfolding i by auto*
qed

lemma *upt-decomp-lt:*
assumes $H: \langle xs @ i \# ys @ j \# zs = [m ..< n] \rangle$
shows $\langle i < j \rangle$
proof –
have $xs: \langle xs = [m ..< i] \rangle$ **and** $ys: \langle ys = [\text{Suc } i ..< j] \rangle$ **and** $zs: \langle zs = [\text{Suc } j ..< n] \rangle$
using *H by (auto dest: append-cons-eq-upt-length-i append-cons-eq-upt-length-i-end)*
show *?thesis*
by (*metis append-cons-eq-upt-length-i-end assms lessI less-trans self-append-conv2 upt-eq-Cons-conv upt-rec ys*)
qed

lemma *nths-upt-upto-Suc:* $\langle aa < \text{length } xs \implies \text{nths } xs \{ 0..<\text{Suc } aa \} = \text{nths } xs \{ 0..<aa \} @ [xs ! aa] \rangle$

by (simp add: atLeast0LessThan take-Suc-conv-app-nth)

The following two lemmas are useful as simp rules for case-distinction. The case *length* $l = 0$ is already simplified by default.

lemma *length-list-Suc-0*:

```

  ⟨length W = Suc 0 ⟷ (∃ L. W = [L])⟩
  apply (cases W)
  apply (simp; fail)
  apply (rename-tac a W', case-tac W')
  apply auto
  done

```

lemma *length-list-2*: ⟨length $S = 2 \longleftrightarrow (\exists a\ b. S = [a, b])$ ⟩

```

  apply (cases S)
  apply (simp; fail)
  apply (rename-tac a S')
  apply (case-tac S')
  by simp-all

```

lemma *finite-bounded-list*:

```

  fixes b :: nat
  shows ⟨finite {xs. length xs < s ∧ (∀ i < length xs. xs ! i < b)}⟩ (is ⟨finite (?S s)⟩)

```

proof –

```

  have H: ⟨finite {xs. set xs ⊆ {0..b} ∧ length xs ≤ s}⟩
  by (rule finite-lists-length-le[of {0..b} ⟨s⟩]) auto
  show ?thesis
  by (rule finite-subset[OF - H]) (auto simp: in-set-conv-nth)

```

qed

lemma *last-in-set-dropWhile*:

```

  assumes ⟨∃ L ∈ set (xs @ [x]). ¬P L⟩
  shows ⟨x ∈ set (dropWhile P (xs @ [x]))⟩
  using assms by (induction xs) auto

```

lemma *mset-drop-upto*: ⟨mset (drop a N) = {#N!i. i ∈# mset-set {a..**length** N}#}⟩

proof (induction N arbitrary: a)

```

  case Nil
  then show ?case by simp

```

next

```

  case (Cons c N)
  have upt: ⟨{0..Suc (length N)} = insert 0 {1..Suc (length N)}⟩
  by auto
  then have H: ⟨mset-set {0..Suc (length N)} = add-mset 0 (mset-set {1..Suc (length N)})⟩
  unfolding upt by auto
  have mset-case-Suc: ⟨{#case x of 0 ⇒ c | Suc x ⇒ N ! x . x ∈# mset-set {Suc a..Suc b}#} =
    {#N ! (x-1) . x ∈# mset-set {Suc a..Suc b}#}⟩ for a b
  by (rule image-mset-cong) (auto split: nat.splits)
  have Suc-Suc: ⟨{Suc a..Suc b} = Suc ‘ {a..b}⟩ for a b
  by auto
  then have mset-set-Suc-Suc: ⟨mset-set {Suc a..Suc b} = {#Suc n. n ∈# mset-set {a..b}#}⟩ for
a b
  unfolding Suc-Suc by (subst image-mset-mset-set[symmetric]) auto
  have *: ⟨{#N ! (x-Suc 0) . x ∈# mset-set {Suc a..Suc b}#} = {#N ! x . x ∈# mset-set {a..b}#}⟩
  for a b
  by (auto simp add: mset-set-Suc-Suc)
  show ?case

```

```

    apply (cases a)
    using Cons[of 0] Cons by (auto simp: nth-Cons drop-Cons H mset-case-Suc *)
qed

```

```

lemma last-list-update-to-last:
  ⟨last (xs[x := last xs]) = last xs⟩
  by (metis last-list-update list-update.simps(1))

```

```

lemma take-map-nth-alt-def: ⟨take n xs = map ((! xs) [0..

```

1.3.6 Lexicographic Ordering

```

lemma lexn-Suc:
  ⟨(x # xs, y # ys) ∈ lexn r (Suc n) ⟷
    (length xs = n ∧ length ys = n) ∧ ((x, y) ∈ r ∨ (x = y ∧ (xs, ys) ∈ lexn r n))⟩
  by (auto simp: map-prod-def image-iff lex-prod-def)

```

```

lemma lexn-n:
  ⟨n > 0 ⟹ (x # xs, y # ys) ∈ lexn r n ⟷
    (length xs = n-1 ∧ length ys = n-1) ∧ ((x, y) ∈ r ∨ (x = y ∧ (xs, ys) ∈ lexn r (n - 1)))⟩
  apply (cases n)
  apply simp
  by (auto simp: map-prod-def image-iff lex-prod-def)

```

There is some subtle point in the previous theorem explaining *why* it is useful. The term 1 is converted to $\text{Suc } 0$, but 2 is not, meaning that 1 is automatically simplified by default allowing the use of the default simplification rule lexn.simps . However, for 2 one additional simplification rule is required (see the proof of the theorem above).

```

lemma lexn2-conv:
  ⟨([a, b], [c, d]) ∈ lexn r 2 ⟷ (a, c) ∈ r ∨ (a = c ∧ (b, d) ∈ r)⟩
  by (auto simp: lexn-n simp del: lexn.simps(2))

```

```

lemma lexn3-conv:
  ⟨([a, b, c], [a', b', c']) ∈ lexn r 3 ⟷
    (a, a') ∈ r ∨ (a = a' ∧ (b, b') ∈ r) ∨ (a = a' ∧ b = b' ∧ (c, c') ∈ r)⟩

```

by (auto simp: lexn-n simp del: lexn.simps(2))

lemma *prepend-same-lexn*:
 assumes *irrefl*: $\langle \text{irrefl } R \rangle$
 shows $\langle (A @ B, A @ C) \in \text{lexn } R \ n \longleftrightarrow (B, C) \in \text{lexn } R \ (n - \text{length } A) \rangle$ (is $\langle ?A \longleftrightarrow ?B \rangle$)

proof
 assume $?A$
 then obtain *xys x xs y ys* where
 len-B: $\langle \text{length } B = n - \text{length } A \rangle$ and
 len-C: $\langle \text{length } C = n - \text{length } A \rangle$ and
 AB: $\langle A @ B = \text{xys} @ x \# xs \rangle$ and
 AC: $\langle A @ C = \text{xys} @ y \# ys \rangle$ and
 xy: $\langle (x, y) \in R \rangle$
 by (auto simp: lexn-conv)
 have *x-neq-y*: $\langle x \neq y \rangle$
 using *xy irrefl* by (auto simp add: irrefl-def)
 then have $\langle B = \text{drop } (\text{length } A) \ \text{xys} @ x \# xs \rangle$
 using *arg-cong*[OF *AB*, of $\langle \text{drop } (\text{length } A) \rangle$]
 apply (cases $\langle \text{length } A - \text{length } \text{xys} \rangle$)
 apply (auto; fail)
 by (metis *AB AC nth-append nth-append-length zero-less-Suc zero-less-diff*)

moreover have $\langle C = \text{drop } (\text{length } A) \ \text{xys} @ y \# ys \rangle$
 using *arg-cong*[OF *AC*, of $\langle \text{drop } (\text{length } A) \rangle$] *x-neq-y*
 apply (cases $\langle \text{length } A - \text{length } \text{xys} \rangle$)
 apply (auto; fail)
 by (metis *AB AC nth-append nth-append-length zero-less-Suc zero-less-diff*)

ultimately show $?B$
 using *len-B*[*symmetric*] *len-C*[*symmetric*] *xy*
 by (auto simp: lexn-conv)

next
 assume $?B$
 then obtain *xys x xs y ys* where
 len-B: $\langle \text{length } B = n - \text{length } A \rangle$ and
 len-C: $\langle \text{length } C = n - \text{length } A \rangle$ and
 AB: $\langle B = \text{xys} @ x \# xs \rangle$ and
 AC: $\langle C = \text{xys} @ y \# ys \rangle$ and
 xy: $\langle (x, y) \in R \rangle$
 by (auto simp: lexn-conv)
 define *Axys* where $\langle Axys = A @ \text{xys} \rangle$

have $\langle A @ B = Axys @ x \# xs \rangle$
 using *AB Axys-def* by auto

moreover have $\langle A @ C = Axys @ y \# ys \rangle$
 using *AC Axys-def* by auto

moreover have $\langle \text{Suc } (\text{length } Axys + \text{length } xs) = n \rangle$ and
 $\langle \text{length } ys = \text{length } xs \rangle$
 using *len-B len-C AB AC Axys-def* by auto

ultimately show $?A$
 using *len-B*[*symmetric*] *len-C*[*symmetric*] *xy*
 by (auto simp: lexn-conv)

qed

lemma *append-same-lexn*:
 assumes *irrefl*: $\langle \text{irrefl } R \rangle$

shows $\langle (B @ A, C @ A) \in \text{lexn } R \ n \longleftrightarrow (B, C) \in \text{lexn } R \ (n - \text{length } A) \rangle$ (is $\langle ?A \longleftrightarrow ?B \rangle$)

proof

assume $?A$

then obtain $\text{rys } x \text{ xs } y \text{ ys}$ where

len-B: $\langle n = \text{length } B + \text{length } A \rangle$ and

len-C: $\langle n = \text{length } C + \text{length } A \rangle$ and

AB: $\langle B @ A = \text{rys} @ x \# \text{xs} \rangle$ and

AC: $\langle C @ A = \text{rys} @ y \# \text{ys} \rangle$ and

xy: $\langle (x, y) \in R \rangle$

by (auto simp: lexn-conv)

have $x \neq y$: $\langle x \neq y \rangle$

using xy irrefl by (auto simp add: irrefl-def)

have len-C-B: $\langle \text{length } C = \text{length } B \rangle$

using len-B len-C by simp

have len-B-rys: $\langle \text{length } B > \text{length } \text{rys} \rangle$

apply (rule ccontr)

using arg-cong[OF AB, of $\langle \text{take } (\text{length } B) \rangle$] arg-cong[OF AB, of $\langle \text{drop } (\text{length } B) \rangle$]

arg-cong[OF AC, of $\langle \text{drop } (\text{length } C) \rangle$] x-neq-y len-C-B

by auto

then have B: $\langle B = \text{rys} @ x \# \text{take } (\text{length } B - \text{Suc } (\text{length } \text{rys})) \text{ xs} \rangle$

using arg-cong[OF AB, of $\langle \text{take } (\text{length } B) \rangle$]

by (cases $\langle \text{length } B - \text{length } \text{rys} \rangle$) simp-all

have C: $\langle C = \text{rys} @ y \# \text{take } (\text{length } C - \text{Suc } (\text{length } \text{rys})) \text{ ys} \rangle$

using arg-cong[OF AC, of $\langle \text{take } (\text{length } C) \rangle$] x-neq-y len-B-rys unfolding len-C-B[symmetric]

by (cases $\langle \text{length } C - \text{length } \text{rys} \rangle$) auto

show $?B$

using len-B[symmetric] len-C[symmetric] xy B C

by (auto simp: lexn-conv)

next

assume $?B$

then obtain $\text{rys } x \text{ xs } y \text{ ys}$ where

len-B: $\langle \text{length } B = n - \text{length } A \rangle$ and

len-C: $\langle \text{length } C = n - \text{length } A \rangle$ and

AB: $\langle B = \text{rys} @ x \# \text{xs} \rangle$ and

AC: $\langle C = \text{rys} @ y \# \text{ys} \rangle$ and

xy: $\langle (x, y) \in R \rangle$

by (auto simp: lexn-conv)

define Ays Axs where $\langle Ays = \text{ys} @ A \rangle$ and $\langle Axs = \text{xs} @ A \rangle$

have $\langle B @ A = \text{rys} @ x \# Axs \rangle$

using AB Axs-def by auto

moreover have $\langle C @ A = \text{rys} @ y \# Ays \rangle$

using AC Ays-def by auto

moreover have $\langle \text{Suc } (\text{length } \text{rys} + \text{length } Axs) = n \rangle$ and

$\langle \text{length } Ays = \text{length } Axs \rangle$

using len-B len-C AB AC Axs-def Ays-def by auto

ultimately show $?A$

using len-B[symmetric] len-C[symmetric] xy

by (auto simp: lexn-conv)

qed

lemma irrefl-less-than [simp]: $\langle \text{irrefl less-than} \rangle$

by (auto simp: irrefl-def)

1.3.7 Remove

More lemmas about remove

lemma *distinct-remove1-last-butlast*:

$\langle \text{distinct } xs \implies xs \neq [] \implies \text{remove1 } (\text{last } xs) \text{ } xs = \text{butlast } xs \rangle$
by (*metis append-Nil2 append-butlast-last-id distinct-butlast not-distinct-conv-prefix remove1.simps(2) remove1-append*)

lemma *remove1-Nil-iff*:

$\langle \text{remove1 } x \text{ } xs = [] \iff xs = [] \vee xs = [x] \rangle$
by (*cases xs*) *auto*

lemma *removeAll-upt*:

$\langle \text{removeAll } k \text{ } [a..<b] = (\text{if } k \geq a \wedge k < b \text{ then } [a..<k] @ [\text{Suc } k..<b] \text{ else } [a..<b]) \rangle$
by (*induction b*) *auto*

lemma *remove1-upt*:

$\langle \text{remove1 } k \text{ } [a..<b] = (\text{if } k \geq a \wedge k < b \text{ then } [a..<k] @ [\text{Suc } k..<b] \text{ else } [a..<b]) \rangle$
by (*subst distinct-remove1-removeAll*) (*auto simp: removeAll-upt*)

lemma *sorted-removeAll*: $\langle \text{sorted } C \implies \text{sorted } (\text{removeAll } k \text{ } C) \rangle$

by (*metis map-ident removeAll-filter-not-eq sorted-filter*)

lemma *distinct-remove1-rev*: $\langle \text{distinct } xs \implies \text{remove1 } x \text{ } (\text{rev } xs) = \text{rev } (\text{remove1 } x \text{ } xs) \rangle$

using *split-list[of x xs]*
by (*cases (x ∈ set xs)*) (*auto simp: remove1-append remove1-idem*)

Remove under condition

This function removes the first element such that the condition f holds. It generalises *remove1*.

fun *remove1-cond* **where**

$\langle \text{remove1-cond } f \text{ } [] = [] \rangle \mid$
 $\langle \text{remove1-cond } f \text{ } (C' \# L) = (\text{if } f \text{ } C' \text{ then } L \text{ else } C' \# \text{remove1-cond } f \text{ } L) \rangle$

lemma $\langle \text{remove1 } x \text{ } xs = \text{remove1-cond } ((=) \text{ } x) \text{ } xs \rangle$

by (*induction xs*) *auto*

lemma *mset-map-mset-remove1-cond*:

$\langle \text{mset } (\text{map } \text{mset } (\text{remove1-cond } (\lambda L. \text{mset } L = \text{mset } a) \text{ } C)) =$
 $\text{remove1-mset } (\text{mset } a) \text{ } (\text{mset } (\text{map } \text{mset } C)) \rangle$
by (*induction C*) *auto*

We can also generalise *removeAll*, which is close to *filter*:

fun *removeAll-cond* :: $\langle 'a \Rightarrow \text{bool} \rangle \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}$ **where**

$\langle \text{removeAll-cond } f \text{ } [] = [] \rangle \mid$
 $\langle \text{removeAll-cond } f \text{ } (C' \# L) = (\text{if } f \text{ } C' \text{ then } \text{removeAll-cond } f \text{ } L \text{ else } C' \# \text{removeAll-cond } f \text{ } L) \rangle$

lemma *removeAll-removeAll-cond*: $\langle \text{removeAll } x \text{ } xs = \text{removeAll-cond } ((=) \text{ } x) \text{ } xs \rangle$

by (*induction xs*) *auto*

lemma *removeAll-cond-filter*: $\langle \text{removeAll-cond } P \text{ } xs = \text{filter } (\lambda x. \neg P \text{ } x) \text{ } xs \rangle$

by (*induction xs*) *auto*

lemma *mset-map-mset-removeAll-cond*:

$\langle \text{mset } (\text{map } \text{mset } (\text{removeAll-cond } (\lambda b. \text{mset } b = \text{mset } a) \text{ } C)) =$

$= \text{removeAll-mset } (\text{mset } a) (\text{mset } (\text{map } \text{mset } C))$
by (*induction C*) *auto*

lemma *count-mset-count-list*:

$\langle \text{count } (\text{mset } xs) \ x = \text{count-list } xs \ x \rangle$
by (*induction xs*) *auto*

lemma *length-removeAll-count-list*:

$\langle \text{length } (\text{removeAll } x \ xs) = \text{length } xs - \text{count-list } xs \ x \rangle$

proof –

have $\langle \text{length } (\text{removeAll } x \ xs) = \text{size } (\text{removeAll-mset } x \ (\text{mset } xs)) \rangle$
by *auto*
also have $\langle \dots = \text{size } (\text{mset } xs) - \text{count } (\text{mset } xs) \ x \rangle$
by (*metis count-le-replicate-mset-subset-eq le-refl size-Diff-submset size-replicate-mset*)
also have $\langle \dots = \text{length } xs - \text{count-list } xs \ x \rangle$
unfolding *count-mset-count-list* **by** *simp*
finally show *?thesis* .

qed

lemma *removeAll-notin*: $\langle a \notin \# A \implies \text{removeAll-mset } a \ A = A \rangle$
using *count-inI* **by** *force*

Filter

lemma *distinct-filter-eq-if*:

$\langle \text{distinct } C \implies \text{length } (\text{filter } ((=) \ L) \ C) = (\text{if } L \in \text{set } C \text{ then } 1 \text{ else } 0) \rangle$
by (*induction C*) *auto*

lemma *length-filter-update-true*:

assumes $\langle i < \text{length } xs \rangle$ **and** $\langle P \ (xs \ ! \ i) \rangle$
shows $\langle \text{length } (\text{filter } P \ (xs[i := x])) = \text{length } (\text{filter } P \ xs) - (\text{if } P \ x \text{ then } 0 \text{ else } 1) \rangle$
apply (*subst (5) append-take-drop-id[of i, symmetric]*)
using *assms upd-conv-take-nth-drop[of i xs x] Cons-nth-drop-Suc[of i xs, symmetric]*
unfolding *filter-append length-append*
by *simp*

lemma *length-filter-update-false*:

assumes $\langle i < \text{length } xs \rangle$ **and** $\langle \neg P \ (xs \ ! \ i) \rangle$
shows $\langle \text{length } (\text{filter } P \ (xs[i := x])) = \text{length } (\text{filter } P \ xs) + (\text{if } P \ x \text{ then } 1 \text{ else } 0) \rangle$
apply (*subst (5) append-take-drop-id[of i, symmetric]*)
using *assms upd-conv-take-nth-drop[of i xs x] Cons-nth-drop-Suc[of i xs, symmetric]*
unfolding *filter-append length-append*
by *simp*

lemma *mset-set-mset-set-minus-id-iff*:

assumes $\langle \text{finite } A \rangle$
shows $\langle \text{mset-set } A = \text{mset-set } (A - B) \longleftrightarrow (\forall b \in B. \ b \notin A) \rangle$

proof –

have *f1*: $\text{mset-set } A = \text{mset-set } (A - B) \longleftrightarrow A - B = A$
using *assms* **by** (*metis (no-types) finite-Diff finite-set-mset-mset-set*)
then show *?thesis*
by *blast*

qed

lemma *mset-set-eq-mset-set-more-conds*:

$\langle \text{finite } \{x. \ P \ x\} \implies \text{mset-set } \{x. \ P \ x\} = \text{mset-set } \{x. \ Q \ x \wedge P \ x\} \longleftrightarrow (\forall x. \ P \ x \longrightarrow Q \ x) \rangle$

```

(is (⟦?F⟧ ⟹ ?A ⟷ ?B⟧)
proof -
  assume ?F
  then have (⟦?A ⟷ (∀ x ∈ {x. P x}. x ∈ {x. Q x ∧ P x})⟧)
    by (subst mset-set-eq-iff) auto
  also have (⟦... ⟷ (∀ x. P x ⟹ Q x)⟧)
    by blast
  finally show ?thesis .
qed

```

```

lemma count-list-filter: (count-list xs x = length (filter ((=) x) xs))
  by (induction xs) auto

```

```

lemma sum-length-filter-compl': (length [x←xs . ¬ P x] + length (filter P xs) = length xs)
  using sum-length-filter-compl[of P xs] by auto

```

1.3.8 Sorting

See $\llbracket \text{sorted } ?xs; \text{distinct } ?xs; \text{sorted } ?ys; \text{distinct } ?ys; \text{set } ?xs = \text{set } ?ys \rrbracket \implies ?xs = ?ys$.

```

lemma sorted-mset-unique:
  fixes xs :: (α :: linorder) list
  shows (sorted xs ⟹ sorted ys ⟹ mset xs = mset ys ⟹ xs = ys)
  using properties-for-sort by auto

```

```

lemma insort-upt: (insort k [a..] =
  (if k < a then k # [a..]
   else if k < b then [a..] @ k # [k..]
   else [a..] @ [k]))

```

```

proof -
  have H: (k < Suc b ⟹ ¬ k < a ⟹ {a..] = {a..] ∪ {k..] for a b :: nat
    by (simp add: ivl-disj-un-two(3))
  show ?thesis
  apply (induction b)
  apply (simp; fail)
  apply (case-tac (¬ k < a ∧ k < Suc b))
  apply (rule sorted-mset-unique)
    apply ((auto simp add: sorted-append sorted-insort ac-simps mset-set-Union
      dest!: H; fail)+)[2]
    apply (auto simp: insort-is-Cons sorted-insort-is-snoc sorted-append mset-set-Union
      ac-simps dest: H; fail)+
  done
qed

```

```

lemma removeAll-insert-removeAll: (removeAll k (insort k xs) = removeAll k xs)
  by (simp add: filter-insort-triv removeAll-filter-not-eq)

```

```

lemma filter-sorted: (sorted xs ⟹ sorted (filter P xs))
  by (metis list.map-ident sorted-filter)

```

```

lemma removeAll-insort:
  (sorted xs ⟹ k ≠ k' ⟹ removeAll k' (insort k xs) = insort k (removeAll k' xs))
  by (simp add: filter-insort removeAll-filter-not-eq)

```

1.3.9 Distinct Multisets

lemma *distinct-mset-remdups-mset-id*: $\langle \text{distinct-mset } C \implies \text{remdups-mset } C = C \rangle$
by (*induction C*) *auto*

lemma *notin-add-mset-remdups-mset*:
 $\langle a \notin \# A \implies \text{add-mset } a (\text{remdups-mset } A) = \text{remdups-mset } (\text{add-mset } a A) \rangle$
by *auto*

lemma *distinct-mset-image-mset*:
 $\langle \text{distinct-mset } (\text{image-mset } f (\text{mset } xs)) \longleftrightarrow \text{distinct } (\text{map } f xs) \rangle$
apply (*subst mset-map[symmetric]*)
apply (*subst distinct-mset-mset-distinct*)
..

lemma *distinct-image-mset-not-equal*:
assumes
 LL' : $\langle L \neq L' \rangle$ **and**
 $dist$: $\langle \text{distinct-mset } (\text{image-mset } f M) \rangle$ **and**
 L : $\langle L \in \# M \rangle$ **and**
 L' : $\langle L' \in \# M \rangle$ **and**
 fLL' [*simp*]: $\langle f L = f L' \rangle$
shows $\langle \text{False} \rangle$
proof –
obtain $M1$ **where** $M1$: $\langle M = \text{add-mset } L M1 \rangle$
using *multi-member-split[OF L]* **by** *blast*
obtain $M2$ **where** $M2$: $\langle M1 = \text{add-mset } L' M2 \rangle$
using *multi-member-split[of L' M1]* $LL' L'$ **unfolding** $M1$ **by** (*auto simp: add-mset-eq-add-mset*)
show *False*
using $dist$ **unfolding** $M1 M2$ **by** *auto*
qed

1.3.10 Set of Distinct Multisets

definition *distinct-mset-set* :: $\langle 'a \text{ multiset set} \Rightarrow \text{bool} \rangle$ **where**
 $\langle \text{distinct-mset-set } \Sigma \longleftrightarrow (\forall S \in \Sigma. \text{distinct-mset } S) \rangle$

lemma *distinct-mset-set-empty[simp]*: $\langle \text{distinct-mset-set } \{\} \rangle$
unfolding *distinct-mset-set-def* **by** *auto*

lemma *distinct-mset-set-singleton[iff]*: $\langle \text{distinct-mset-set } \{A\} \longleftrightarrow \text{distinct-mset } A \rangle$
unfolding *distinct-mset-set-def* **by** *auto*

lemma *distinct-mset-set-insert[iff]*:
 $\langle \text{distinct-mset-set } (\text{insert } S \Sigma) \longleftrightarrow (\text{distinct-mset } S \wedge \text{distinct-mset-set } \Sigma) \rangle$
unfolding *distinct-mset-set-def* **by** *auto*

lemma *distinct-mset-set-union[iff]*:
 $\langle \text{distinct-mset-set } (\Sigma \cup \Sigma') \longleftrightarrow (\text{distinct-mset-set } \Sigma \wedge \text{distinct-mset-set } \Sigma') \rangle$
unfolding *distinct-mset-set-def* **by** *auto*

lemma *in-distinct-mset-set-distinct-mset*:
 $\langle a \in \Sigma \implies \text{distinct-mset-set } \Sigma \implies \text{distinct-mset } a \rangle$
unfolding *distinct-mset-set-def* **by** *auto*

lemma *distinct-mset-remdups-mset[simp]*: $\langle \text{distinct-mset } (\text{remdups-mset } S) \rangle$

using *count-remdups-mset-eq-1* **unfolding** *distinct-mset-def* **by** *metis*

lemma *distinct-mset-mset-set*: $\langle \text{distinct-mset } (\text{mset-set } A) \rangle$
unfolding *distinct-mset-def* *count-mset-set-if* **by** *(auto simp: not-in-iff)*

lemma *distinct-mset-filter-mset-set[simp]*: $\langle \text{distinct-mset } \{\#a \in \# \text{ mset-set } A. P \ a\# \} \rangle$
by *(simp add: distinct-mset-filter distinct-mset-mset-set)*

lemma *distinct-mset-set-distinct*: $\langle \text{distinct-mset-set } (\text{mset } \text{' set } Cs) \longleftrightarrow (\forall c \in \text{ set } Cs. \text{ distinct } c) \rangle$
unfolding *distinct-mset-set-def* **by** *auto*

1.3.11 Sublists

lemma *nths-single-if*: $\langle \text{nths } l \ \{n\} = (\text{if } n < \text{length } l \text{ then } [!n] \text{ else } []) \rangle$

proof –

have *[simp]*: $\langle 0 < n \implies \{j. \text{Suc } j = n\} = \{n-1\} \rangle$ **for** *n*
by *auto*

show *?thesis*

apply *(induction l arbitrary: n)*

subgoal by *(auto simp: nths-def)*

subgoal by *(auto simp: nths-Cons)*

done

qed

lemma *atLeastLessThan-Collect*: $\langle \{a..<b\} = \{j. j \geq a \wedge j < b\} \rangle$
by *auto*

lemma *mset-nths-subset-mset*: $\langle \text{mset } (\text{nths } xs \ A) \subseteq \# \text{ mset } xs \rangle$

apply *(induction xs arbitrary: A)*

subgoal by *auto*

subgoal for *a xs A*

using *subset-mset.add-increasing2* *[of add-mset - {#}]* $\langle \text{mset } (\text{nths } xs \ \{j. \text{Suc } j \in A\}) \rangle$
 $\langle \text{mset } xs \rangle$

by *(auto simp: nths-Cons)*

done

lemma *nths-id-iff*:

$\langle \text{nths } xs \ A = xs \longleftrightarrow \{0..<\text{length } xs\} \subseteq A \rangle$

proof –

have $\langle \{j. \text{Suc } j \in A\} = (\lambda j. j-1) \text{' } (A - \{0\}) \rangle$ **for** *A*

using *DiffI* **by** *(fastforce simp: image-iff)*

have *1*: $\langle \{0..<b\} \subseteq \{j. \text{Suc } j \in A\} \longleftrightarrow (\forall x. x-1 < b \longrightarrow x \neq 0 \longrightarrow x \in A) \rangle$

for *A :: nat set* **and** *b :: nat*

by *auto*

have *[simp]*: $\langle \{0..<b\} \subseteq \{j. \text{Suc } j \in A\} \longleftrightarrow (\forall x. x-1 < b \longrightarrow x \in A) \rangle$

if $\langle 0 \in A \rangle$ **for** *A :: nat set* **and** *b :: nat*

using *that* **unfolding** *1* **by** *auto*

have *[simp]*: $\langle \text{nths } xs \ \{j. \text{Suc } j \in A\} = a \ \# \ xs \longleftrightarrow \text{False} \rangle$

for *a :: 'a* **and** *xs :: 'a list* **and** *A :: nat set*

using *mset-nths-subset-mset* *[of xs {j. Suc j ∈ A}]* **by** *auto*

show *?thesis*

apply *(induction xs arbitrary: A)*

subgoal by *auto*

subgoal

by *(auto 5 5 simp: nths-Cons) fastforce*

done

qed

lemma *nts-upt-length[simp]*: $\langle nths\ xs\ \{0..<length\ xs\} = xs \rangle$
 by (auto simp: nths-id-iff)

lemma *nts-shift-lemma'*:

$\langle map\ fst\ [p \leftarrow zip\ xs\ [i..<i+n].\ snd\ p + b \in A] = map\ fst\ [p \leftarrow zip\ xs\ [0..<n].\ snd\ p + b + i \in A] \rangle$

proof (induct xs arbitrary: i n b)

case Nil

then show ?case by simp

next

case (Cons a xs)

have 1: $\langle map\ fst\ [p \leftarrow zip\ (a \# xs)\ (i \# [Suc\ i..<i+n]).\ snd\ p + b \in A] =$
 $(if\ i + b \in A\ then\ a \# map\ fst\ [p \leftarrow zip\ xs\ [Suc\ i..<i+n].\ snd\ p + b \in A]$
 $else\ map\ fst\ [p \leftarrow zip\ xs\ [Suc\ i..<i+n].\ snd\ p + b \in A]) \rangle$

by simp

have 2: $\langle map\ fst\ [p \leftarrow zip\ (a \# xs)\ [0..<n].\ snd\ p + b + i \in A] =$
 $(if\ i + b \in A\ then\ a \# map\ fst\ [p \leftarrow zip\ xs\ [1..<n].\ snd\ p + b + i \in A]$
 $else\ map\ fst\ [p \leftarrow zip\ (xs)\ [1..<n].\ snd\ p + b + i \in A]) \rangle$

if $\langle n > 0 \rangle$

by (subst upt-conv-Cons) (use that in (auto simp: ac-simps))

show ?case

proof (cases n)

case 0

then show ?thesis by simp

next

case n: (Suc m)

then have *i-n-m*: $\langle i + n = Suc\ i + m \rangle$

by auto

have 3: $\langle map\ fst\ [p \leftarrow zip\ xs\ [Suc\ i..<i+n].\ snd\ p + b \in A] =$
 $map\ fst\ [p \leftarrow zip\ xs\ [0..<m].\ snd\ p + b + Suc\ i \in A] \rangle$

using Cons[of b (Suc i) m] unfolding *i-n-m* .

have 4: $\langle map\ fst\ [p \leftarrow zip\ xs\ [1..<n].\ snd\ p + b + i \in A] =$
 $map\ fst\ [p \leftarrow zip\ xs\ [0..<m].\ Suc\ (snd\ p + b + i) \in A] \rangle$

using Cons[of (b+i) 1 m] unfolding n Suc-eq-plus1-left add.commute[of 1]

by (simp-all add: ac-simps)

show ?thesis

apply (subst upt-conv-Cons)

using n apply (simp; fail)

apply (subst 1)

apply (subst 2)

using n apply (simp; fail)

apply (subst 3)

apply (subst 3)

apply (subst 4)

apply (subst 4)

by force

qed

qed

lemma *nts-Cons-upt-Suc*: $\langle nths\ (a \# xs)\ \{0..<Suc\ n\} = a \# nths\ xs\ \{0..<n\} \rangle$

unfolding nths-def

apply (subst upt-conv-Cons)

apply simp

using nths-shift-lemma'[of 0 {0..<Suc n} xs 1 (length xs)]

by (simp-all add: ac-simps)

lemma *nths-empty-iff*: $\langle \text{nths } xs \ A = [] \longleftrightarrow \{..$

proof (induction xs arbitrary: A)

case Nil

then show ?case by auto

next

case (Cons a xs) note IH = this(1)

have $\langle (\forall x < length\ xs. x \neq 0 \longrightarrow x \notin A) \rangle$

if a1: $\langle \{..$

proof (intro allI impI)

fix nn

assume nn: $\langle nn < length\ xs \rangle \langle nn \neq 0 \rangle$

moreover have $\forall n. Suc\ n \notin A \vee \neg n < length\ xs$

using a1 by blast

then show $nn \notin A$

using nn

by (metis (no-types) lessI less-trans list-decode.cases)

qed

show ?case

proof (cases $\langle 0 \in A \rangle$)

case True

then show ?thesis by (subst nths-Cons) auto

next

case False

then show ?thesis

by (subst nths-Cons) (use less-Suc-eq-0-disj IH in auto)

qed

qed

lemma *nths-upt-Suc*:

assumes $\langle i < length\ xs \rangle$

shows $\langle \text{nths } xs \ \{i..$

proof –

have upt: $\langle \{i.. for $i\ k :: nat$$

by auto

show ?thesis

using assms

proof (induction xs arbitrary: i)

case Nil

then show ?case by simp

next

case (Cons a xs i) note IH = this(1) and i-le = this(2)

have [simp]: $\langle i - Suc\ 0 \leq j \longleftrightarrow i \leq Suc\ j \rangle$ if $\langle i > 0 \rangle$ for j

using that by auto

show ?case

using IH[of $\langle i-1 \rangle$] i-le

by (auto simp add: nths-Cons upt)

qed

qed

lemma *nths-upt-Suc'*:

assumes $\langle i < b \rangle$ and $\langle b \leq length\ xs \rangle$

shows $\langle \text{nths } xs \ \{i..<b\} = xs!i \ \# \ \text{nths } xs \ \{Suc\ i..<b\} \rangle$

proof –


```

have S1: {j. i ≤ Suc j ∧ j < b - Suc 0} = {j. i ≤ Suc j ∧ Suc j < b} for i b
  by auto
have S2: {j. i ≤ j ∧ j < b - Suc 0} = {j. i ≤ j ∧ Suc j < b} for i b
  by auto
have upt: {i..<k} = {j. i ≤ j ∧ j < k} for i k :: nat
  by auto
show ?thesis
  using assms
proof (induction xs arbitrary: i b)
  case Nil
  then show ?case by simp
next
  case (Cons a xs i) note IH = this(1) and i-le = this(2,3)
  have [simp]: ⟨i - Suc 0 ≤ j ⟷ i ≤ Suc j⟩ if ⟨i > 0⟩ for j
    using that by auto
  have ⟨i - Suc 0 < b - Suc 0 ∨ (i = 0)⟩
    using i-le by linarith
  moreover have ⟨b - Suc 0 ≤ length xs ∨ xs = []⟩
    using i-le by auto
  ultimately show ?case
    using IH[of ⟨i-1⟩ ⟨b-1⟩] i-le
    apply (subst nth-Cons)
    apply (subst nth-Cons)
    by (auto simp: upt S1 S2)
qed
qed

lemma Ball-set-nths: (∀ L ∈ set (nths xs A). P L) ⟷ (∀ i ∈ A ∩ {0..<length xs}. P (xs ! i))
  unfolding set-nths by fastforce

```

1.3.12 Product Case

The splitting of tuples is done for sizes strictly less than 8. As we want to manipulate tuples of size 8, here is some more setup for larger sizes.

```

lemma prod-cases8 [cases type]:
  obtains (fields) a b c d e f g h where y = (a, b, c, d, e, f, g, h)
  by (cases y, cases ⟨snd y⟩) auto

lemma prod-induct8 [case-names fields, induct type]:
  (∧ a b c d e f g h. P (a, b, c, d, e, f, g, h)) ⟹ P x
  by (cases x) blast

lemma prod-cases9 [cases type]:
  obtains (fields) a b c d e f g h i where y = (a, b, c, d, e, f, g, h, i)
  by (cases y, cases ⟨snd y⟩) auto

lemma prod-induct9 [case-names fields, induct type]:
  (∧ a b c d e f g h i. P (a, b, c, d, e, f, g, h, i)) ⟹ P x
  by (cases x) blast

lemma prod-cases10 [cases type]:
  obtains (fields) a b c d e f g h i j where y = (a, b, c, d, e, f, g, h, i, j)
  by (cases y, cases ⟨snd y⟩) auto

lemma prod-induct10 [case-names fields, induct type]:

```

$(\bigwedge a\ b\ c\ d\ e\ f\ g\ h\ i\ j. P\ (a, b, c, d, e, f, g, h, i, j)) \implies P\ x$
by (cases x) blast

lemma prod-cases11 [cases type]:
obtains (fields) a b c d e f g h i j k **where** $y = (a, b, c, d, e, f, g, h, i, j, k)$
by (cases y, cases (snd y)) auto

lemma prod-induct11 [case-names fields, induct type]:
 $(\bigwedge a\ b\ c\ d\ e\ f\ g\ h\ i\ j\ k. P\ (a, b, c, d, e, f, g, h, i, j, k)) \implies P\ x$
by (cases x) blast

lemma prod-cases12 [cases type]:
obtains (fields) a b c d e f g h i j k l **where** $y = (a, b, c, d, e, f, g, h, i, j, k, l)$
by (cases y, cases (snd y)) auto

lemma prod-induct12 [case-names fields, induct type]:
 $(\bigwedge a\ b\ c\ d\ e\ f\ g\ h\ i\ j\ k\ l. P\ (a, b, c, d, e, f, g, h, i, j, k, l)) \implies P\ x$
by (cases x) blast

lemma prod-cases13 [cases type]:
obtains (fields) a b c d e f g h i j k l m **where** $y = (a, b, c, d, e, f, g, h, i, j, k, l, m)$
by (cases y, cases (snd y)) auto

lemma prod-induct13 [case-names fields, induct type]:
 $(\bigwedge a\ b\ c\ d\ e\ f\ g\ h\ i\ j\ k\ l\ m. P\ (a, b, c, d, e, f, g, h, i, j, k, l, m)) \implies P\ x$
by (cases x) blast

lemma prod-cases14 [cases type]:
obtains (fields) a b c d e f g h i j k l m n **where** $y = (a, b, c, d, e, f, g, h, i, j, k, l, m, n)$
by (cases y, cases (snd y)) auto

lemma prod-induct14 [case-names fields, induct type]:
 $(\bigwedge a\ b\ c\ d\ e\ f\ g\ h\ i\ j\ k\ l\ m\ n. P\ (a, b, c, d, e, f, g, h, i, j, k, l, m, n)) \implies P\ x$
by (cases x) blast

lemma prod-cases15 [cases type]:
obtains (fields) a b c d e f g h i j k l m n p **where**
 $y = (a, b, c, d, e, f, g, h, i, j, k, l, m, n, p)$
by (cases y, cases (snd y)) auto

lemma prod-induct15 [case-names fields, induct type]:
 $(\bigwedge a\ b\ c\ d\ e\ f\ g\ h\ i\ j\ k\ l\ m\ n\ p. P\ (a, b, c, d, e, f, g, h, i, j, k, l, m, n, p)) \implies P\ x$
by (cases x) blast

lemma prod-cases16 [cases type]:
obtains (fields) a b c d e f g h i j k l m n p q **where**
 $y = (a, b, c, d, e, f, g, h, i, j, k, l, m, n, p, q)$
by (cases y, cases (snd y)) auto

lemma prod-induct16 [case-names fields, induct type]:
 $(\bigwedge a\ b\ c\ d\ e\ f\ g\ h\ i\ j\ k\ l\ m\ n\ p\ q. P\ (a, b, c, d, e, f, g, h, i, j, k, l, m, n, p, q)) \implies P\ x$
by (cases x) blast

lemma prod-cases17 [cases type]:
obtains (fields) a b c d e f g h i j k l m n p q r **where**
 $y = (a, b, c, d, e, f, g, h, i, j, k, l, m, n, p, q, r)$

by (cases y, cases ⟨snd y⟩) auto

lemma prod-induct17 [case-names fields, induct type]:

($\bigwedge a\ b\ c\ d\ e\ f\ g\ h\ i\ j\ k\ l\ m\ n\ p\ q\ r. P\ (a, b, c, d, e, f, g, h, i, j, k, l, m, n, p, q, r)$) $\implies P\ x$
by (cases x) blast

lemma prod-cases18 [cases type]:

obtains (fields) a b c d e f g h i j k l m n p q r s **where**
y = (a, b, c, d, e, f, g, h, i, j, k, l, m, n, p, q, r, s)
by (cases y, cases ⟨snd y⟩) auto

lemma prod-induct18 [case-names fields, induct type]:

($\bigwedge a\ b\ c\ d\ e\ f\ g\ h\ i\ j\ k\ l\ m\ n\ p\ q\ r\ s. P\ (a, b, c, d, e, f, g, h, i, j, k, l, m, n, p, q, r, s)$) $\implies P\ x$
by (cases x) blast

lemma prod-cases19 [cases type]:

obtains (fields) a b c d e f g h i j k l m n p q r s t **where**
y = (a, b, c, d, e, f, g, h, i, j, k, l, m, n, p, q, r, s, t)
by (cases y, cases ⟨snd y⟩) auto

lemma prod-induct19 [case-names fields, induct type]:

($\bigwedge a\ b\ c\ d\ e\ f\ g\ h\ i\ j\ k\ l\ m\ n\ p\ q\ r\ s\ t. P\ (a, b, c, d, e, f, g, h, i, j, k, l, m, n, p, q, r, s, t)$) $\implies P\ x$
by (cases x) blast

lemma prod-cases20 [cases type]:

obtains (fields) a b c d e f g h i j k l m n p q r s t u **where**
y = (a, b, c, d, e, f, g, h, i, j, k, l, m, n, p, q, r, s, t, u)
by (cases y, cases ⟨snd y⟩) auto

lemma prod-induct20 [case-names fields, induct type]:

($\bigwedge a\ b\ c\ d\ e\ f\ g\ h\ i\ j\ k\ l\ m\ n\ p\ q\ r\ s\ t\ u. P\ (a, b, c, d, e, f, g, h, i, j, k, l, m, n, p, q, r, s, t, u)$) $\implies P\ x$
by (cases x) blast

lemma prod-cases21 [cases type]:

obtains (fields) a b c d e f g h i j k l m n p q r s t u v **where**
y = (a, b, c, d, e, f, g, h, i, j, k, l, m, n, p, q, r, s, t, u, v)
by (cases y, cases ⟨snd y⟩) auto

lemma prod-induct21 [case-names fields, induct type]:

($\bigwedge a\ b\ c\ d\ e\ f\ g\ h\ i\ j\ k\ l\ m\ n\ p\ q\ r\ s\ t\ u\ v. P\ (a, b, c, d, e, f, g, h, i, j, k, l, m, n, p, q, r, s, t, u, v)$) $\implies P\ x$
by (cases x) (blast 43)

lemma prod-cases22 [cases type]:

obtains (fields) a b c d e f g h i j k l m n p q r s t u v w **where**
y = (a, b, c, d, e, f, g, h, i, j, k, l, m, n, p, q, r, s, t, u, v, w)
by (cases y, cases ⟨snd y⟩) auto

lemma prod-induct22 [case-names fields, induct type]:

($\bigwedge a\ b\ c\ d\ e\ f\ g\ h\ i\ j\ k\ l\ m\ n\ p\ q\ r\ s\ t\ u\ v\ w. P\ (a, b, c, d, e, f, g, h, i, j, k, l, m, n, p, q, r, s, t, u, v, w)$) $\implies P\ x$
by (cases x) (blast 43)

lemma prod-cases23 [cases type]:

obtains (*fields*) $a\ b\ c\ d\ e\ f\ g\ h\ i\ j\ k\ l\ m\ n\ p\ q\ r\ s\ t\ u\ v\ w\ x$ **where**
 $y = (a, b, c, d, e, f, g, h, i, j, k, l, m, n, p, q, r, s, t, u, v, w, x)$
by (*cases* y , *cases* $\langle \text{snd } y \rangle$) *auto*

lemma *prod-induct23* [*case-names fields, induct type*]:

$(\bigwedge a\ b\ c\ d\ e\ f\ g\ h\ i\ j\ k\ l\ m\ n\ p\ q\ r\ s\ t\ u\ v\ w\ y.$
 $P\ (a, b, c, d, e, f, g, h, i, j, k, l, m, n, p, q, r, s, t, u, v, w, y)) \implies P\ x$
by (*cases* x) (*blast* 43)

1.3.13 More about *list-all2* and *map*

More properties on the relator *list-all2* and *map*. These theorems are mostly used during the refinement and especially the lifting from a deterministic relator to its list version.

lemma *list-all2-op-eq-map-right-iff*: $\langle \text{list-all2 } (\lambda L. (=) (f\ L))\ a\ aa \longleftrightarrow aa = \text{map } f\ a \rangle$
apply (*induction* a *arbitrary*: aa)
apply (*auto*; *fail*)
by (*rename-tac* aa , *case-tac* aa) (*auto*)

lemma *list-all2-op-eq-map-right-iff'*: $\langle \text{list-all2 } (\lambda L\ L'. L' = f\ L)\ a\ aa \longleftrightarrow aa = \text{map } f\ a \rangle$
apply (*induction* a *arbitrary*: aa)
apply (*auto*; *fail*)
by (*rename-tac* aa , *case-tac* aa) *auto*

lemma *list-all2-op-eq-map-left-iff*: $\langle \text{list-all2 } (\lambda L'\ L. L' = (f\ L))\ a\ aa \longleftrightarrow a = \text{map } f\ aa \rangle$
apply (*induction* a *arbitrary*: aa)
apply (*auto*; *fail*)
by (*rename-tac* aa , *case-tac* aa) (*auto*)

lemma *list-all2-op-eq-map-map-right-iff*:
 $\langle \text{list-all2 } (\text{list-all2 } (\lambda L. (=) (f\ L)))\ xs'\ x \longleftrightarrow x = \text{map } (\text{map } f)\ xs' \rangle$ **for** x
apply (*induction* xs' *arbitrary*: x)
apply (*auto*; *fail*)
apply (*case-tac* x)
by (*auto simp: list-all2-op-eq-map-right-iff*)

lemma *list-all2-op-eq-map-map-left-iff*:
 $\langle \text{list-all2 } (\text{list-all2 } (\lambda L'\ L. L' = f\ L))\ xs'\ x \longleftrightarrow xs' = \text{map } (\text{map } f)\ x \rangle$
apply (*induction* xs' *arbitrary*: x)
apply (*auto*; *fail*)
apply (*rename-tac* x , *case-tac* x)
by (*auto simp: list-all2-op-eq-map-left-iff*)

lemma *list-all2-conj*:
 $\langle \text{list-all2 } (\lambda x\ y. P\ x\ y \wedge Q\ x\ y)\ xs\ ys \longleftrightarrow \text{list-all2 } P\ xs\ ys \wedge \text{list-all2 } Q\ xs\ ys \rangle$
by (*auto simp: list-all2-conv-all-nth*)

lemma *list-all2-replicate*:
 $\langle (bi, b) \in R' \implies \text{list-all2 } (\lambda x\ x'. (x, x') \in R')\ (\text{replicate } n\ bi)\ (\text{replicate } n\ b) \rangle$
by (*induction* n) *auto*

1.3.14 Multisets

We have a lit of lemmas about multisets. Some of them have already moved to *Nested-Multisets-Ordinals.Multisets* but others are too specific (especially the *distinct-mset* property, which roughly corresponds to finite sets).

notation *image-mset* (**infixr** ‘#’ 90)

lemma *in-multiset-empty*: $\langle L \in\# D \implies D \neq \{\#\} \rangle$
by *auto*

The definition and the correctness theorem are from the multiset theory `~/src/HOL/Library/Multiset.thy`, but a name is necessary to refer to them:

definition *union-mset-list* **where**

$\langle \text{union-mset-list } xs \ ys \equiv \text{case-prod append (fold } (\lambda x \ (ys, zs). \text{ remove1 } x \ ys, x \ \# \ zs)) \ xs \ (ys, []) \rangle$

lemma *union-mset-list*:

$\langle \text{mset } xs \cup\# \text{ mset } ys = \text{mset } (\text{union-mset-list } xs \ ys) \rangle$

proof –

have $\langle \bigwedge zs. \text{mset } (\text{case-prod append (fold } (\lambda x \ (ys, zs). \text{ remove1 } x \ ys, x \ \# \ zs)) \ xs \ (ys, zs))) =$
 $(\text{mset } xs \cup\# \text{ mset } ys) + \text{mset } zs \rangle$

by (*induct xs arbitrary: ys*) (*simp-all add: multiset-eq-iff*)

then show *?thesis* **by** (*simp add: union-mset-list-def*)

qed

lemma *union-mset-list-Nil*[*simp*]: $\langle \text{union-mset-list } [] \ bi = bi \rangle$

by (*auto simp: union-mset-list-def*)

lemma *size-le-Suc-0-iff*: $\langle \text{size } M \leq \text{Suc } 0 \longleftrightarrow ((\exists a \ b. M = \{\#a\# \}) \vee M = \{\#\}) \rangle$
using *size-1-singleton-mset* **by** (*auto simp: le-Suc-eq*)

lemma *size-2-iff*: $\langle \text{size } M = 2 \longleftrightarrow (\exists a \ b. M = \{\#a, b\# \}) \rangle$

by (*metis One-nat-def Suc-1 Suc-pred empty-not-add-mset nonempty-has-size size-Diff-singleton*
size-eq-Suc-imp-eq-union size-single union-single-eq-diff union-single-eq-member)

lemma *subset-eq-mset-single-iff*: $\langle x2 \subseteq\# \{\#L\# \} \longleftrightarrow x2 = \{\#\} \vee x2 = \{\#L\# \} \rangle$

by (*metis single-is-union subset-mset.add-diff-inverse subset-mset.eq-refl subset-mset.zero-le*)

lemma *mset-eq-size-2*:

$\langle \text{mset } xs = \{\#a, b\# \} \longleftrightarrow xs = [a, b] \vee xs = [b, a] \rangle$

by (*cases xs*) (*auto simp: add-mset-eq-add-mset Diff-eq-empty-iff-mset subset-eq-mset-single-iff*)

lemma *butlast-list-update*:

$\langle w < \text{length } xs \implies \text{butlast } (xs[w := \text{last } xs]) = \text{take } w \ xs \ @ \ \text{butlast } (\text{last } xs \ \# \ \text{drop } (\text{Suc } w) \ xs) \rangle$

by (*induction xs arbitrary: w*) (*auto split: nat.splits if-splits simp: upd-conv-take-nth-drop*)

lemma *mset-butlast-remove1-mset*: $\langle xs \neq [] \implies \text{mset } (\text{butlast } xs) = \text{remove1-mset } (\text{last } xs) \ (\text{mset } xs) \rangle$

apply (*subst(2) append-butlast-last-id[of xs, symmetric]*)

apply *assumption*

apply (*simp only: mset-append*)

by *auto*

lemma *distinct-mset-mono*: $\langle D' \subseteq\# D \implies \text{distinct-mset } D \implies \text{distinct-mset } D' \rangle$

by (*metis distinct-mset-union subset-mset.le-iff-add*)

lemma *distinct-mset-mono-strict*: $\langle D' \subset\# D \implies \text{distinct-mset } D \implies \text{distinct-mset } D' \rangle$

using *distinct-mset-mono* **by** *auto*

lemma *subset-mset-trans-add-mset*:

$\langle D \subseteq\# D' \implies D \subseteq\# \text{add-mset } L \ D' \rangle$

by (*metis add-mset-remove-trivial diff-subset-eq-self subset-mset.dual-order.trans*)

lemma *subset-add-mset-notin-subset*: $\langle L \notin\# E \implies E \subseteq\# \text{add-mset } L \ D \longleftrightarrow E \subseteq\# D \rangle$
by (*meson subset-add-mset-notin-subset-mset subset-mset-trans-add-mset*)

lemma *remove1-mset-empty-iff*: $\langle \text{remove1-mset } L \ N = \{\#\} \longleftrightarrow N = \{\#L\# \} \vee N = \{\#\} \rangle$
by (*cases* $\langle L \in\# N \rangle$; *cases* N) *auto*

lemma *mset-set-subset-iff*:
 $\langle \text{mset-set } A \subseteq\# I \longleftrightarrow \text{infinite } A \vee A \subseteq \text{set-mset } I \rangle$
by (*metis finite-set-mset finite-set-mset-mset-set mset-set.infinite mset-set-set-mset-subseteq*
set-mset-mono subset-imp-msubset-mset-set subset-mset.bot.extremum subset-mset.dual-order.trans)

lemma *distinct-subseteq-iff*:
assumes *dist*: $\langle \text{distinct-mset } M \rangle$
shows $\langle \text{set-mset } M \subseteq \text{set-mset } N \longleftrightarrow M \subseteq\# N \rangle$

proof

assume $\langle \text{set-mset } M \subseteq \text{set-mset } N \rangle$
then show $\langle M \subseteq\# N \rangle$
using *dist* **by** (*metis distinct-mset-set-mset-ident mset-set-subset-iff*)

next

assume $\langle M \subseteq\# N \rangle$
then show $\langle \text{set-mset } M \subseteq \text{set-mset } N \rangle$
by (*metis set-mset-mono*)

qed

lemma *distinct-set-mset-eq-iff*:
assumes $\langle \text{distinct-mset } M \rangle \langle \text{distinct-mset } N \rangle$
shows $\langle \text{set-mset } M = \text{set-mset } N \longleftrightarrow M = N \rangle$
using *assms distinct-mset-set-mset-ident* **by** *fastforce*

lemma (*in* $-$) *distinct-mset-union2*:
 $\langle \text{distinct-mset } (A + B) \implies \text{distinct-mset } B \rangle$
using *distinct-mset-union*[*of* $B \ A$]
by (*auto simp: ac-simps*)

lemma *in-remove1-msetI*: $\langle x \neq a \implies x \in\# M \implies x \in\# \text{remove1-mset } a \ M \rangle$
by (*simp add: in-remove1-mset-neg*)

lemma *count-multi-member-split*:
 $\langle \text{count } M \ a \geq n \implies \exists M'. M = \text{replicate-mset } n \ a + M' \rangle$
apply (*induction n arbitrary: M*)
subgoal by *auto*
subgoal premises *IH* **for** $n \ M$
using *IH*(1)[*of* $\langle \text{remove1-mset } a \ M \rangle$] *IH*(2)
apply (*cases* $\langle n \leq \text{count } M \ a - \text{Suc } 0 \rangle$)
apply (*auto dest!: Suc-le-D*)
by (*metis count-greater-zero-iff insert-DiffM zero-less-Suc*)
done

lemma *count-image-mset-multi-member-split*:
 $\langle \text{count } (\text{image-mset } f \ M) \ L \geq \text{Suc } 0 \implies \exists K. f \ K = L \wedge K \in\# M \rangle$
by *auto*

lemma *count-image-mset-multi-member-split-2*:
assumes *count*: $\langle \text{count } (\text{image-mset } f \ M) \ L \geq 2 \rangle$
shows $\langle \exists K \ K' \ M'. f \ K = L \wedge K \in\# M \wedge f \ K' = L \wedge K' \in\# \text{remove1-mset } K \ M \wedge$

$M = \{\#K, K'\# \} + M'$
proof –
obtain K **where**
 $K: \langle f K = L \rangle \langle K \in \# M \rangle$
using *count-image-mset-multi-member-split*[of $f M L$] *count* **by** *fastforce*
then obtain K' **where**
 $K': \langle f K' = L \rangle \langle K' \in \# \text{remove1-mset } K M \rangle$
using *count-image-mset-multi-member-split*[of $f \langle \text{remove1-mset } K M \rangle L$] *count*
by (*auto dest!:* *multi-member-split*)
moreover have $\exists M'. M = \{\#K, K'\# \} + M'$
using *multi-member-split*[of $K M$] *multi-member-split*[of $K' \langle \text{remove1-mset } K M \rangle$] $K K'$
by (*auto dest!:* *multi-member-split*)
then show *?thesis*
using $K K'$ **by** *blast*
qed

lemma *minus-notin-trivial*: $L \notin \# A \implies A - \text{add-mset } L B = A - B$
by (*metis diff-intersect-left-idem inter-add-right1*)

lemma *minus-notin-trivial2*: $\langle b \notin \# A \implies A - \text{add-mset } e (\text{add-mset } b B) = A - \text{add-mset } e B \rangle$
by (*subst add-mset-commute*) (*auto simp: minus-notin-trivial*)

lemma *diff-union-single-conv3*: $\langle a \notin \# I \implies \text{remove1-mset } a (I + J) = I + \text{remove1-mset } a J \rangle$
by (*metis diff-union-single-conv remove-1-mset-id-iff-notin union-iff*)

lemma *filter-union-or-split*:
 $\langle \{\#L \in \# C. P L \vee Q L\# \} = \{\#L \in \# C. P L\# \} + \{\#L \in \# C. \neg P L \wedge Q L\# \} \rangle$
by (*induction C*) *auto*

lemma *subset-mset-minus-eq-add-mset-noteq*: $\langle A \subset \# C \implies A - B \neq C \rangle$
by (*auto simp: dest: in-diffD*)

lemma *minus-eq-id-forall-notin-mset*:
 $\langle A - B = A \longleftrightarrow (\forall L \in \# B. L \notin \# A) \rangle$
by (*induction A*)
(auto dest!: *multi-member-split simp: subset-mset-minus-eq-add-mset-noteq*)

lemma *in-multiset-minus-notin-snd*[*simp*]: $\langle a \notin \# B \implies a \in \# A - B \longleftrightarrow a \in \# A \rangle$
by (*metis count-greater-zero-iff count-inI in-diff-count*)

lemma *distinct-mset-in-diff*:
 $\langle \text{distinct-mset } C \implies a \in \# C - D \longleftrightarrow a \in \# C \wedge a \notin \# D \rangle$
by (*meson distinct-mem-diff-mset in-multiset-minus-notin-snd*)

lemma *diff-le-mono2-mset*: $\langle A \subseteq \# B \implies C - B \subseteq \# C - A \rangle$
apply (*auto simp: subseq-mset-def ac-simps*)
by (*simp add: diff-le-mono2*)

lemma *subseq-remove1*[*simp*]: $\langle C \subseteq \# C' \implies \text{remove1-mset } L C \subseteq \# C' \rangle$
by (*meson diff-subset-eq-self subset-mset.dual-order.trans*)

lemma *filter-mset-cong2*:
 $\langle (\bigwedge x. x \in \# M \implies f x = g x) \implies M = N \implies \text{filter-mset } f M = \text{filter-mset } g N \rangle$
by (*hypsubst, rule filter-mset-cong, simp*)

lemma *filter-mset-cong-inner-outer*:

```

assumes
   $M\text{-eq}: \langle \bigwedge x. x \in \# M \implies f\ x = g\ x \rangle$  and
   $\text{notin}: \langle \bigwedge x. x \in \# N - M \implies \neg g\ x \rangle$  and
   $MN: \langle M \subseteq \# N \rangle$ 
shows  $\langle \text{filter-mset } f\ M = \text{filter-mset } g\ N \rangle$ 
proof -
  define  $NM$  where  $\langle NM = N - M \rangle$ 
  have  $N: \langle N = M + NM \rangle$ 
    unfolding  $NM\text{-def}$  using  $MN$  by simp
  have  $\langle \text{filter-mset } g\ NM = \{\#\} \rangle$ 
    using  $\text{notin}$  unfolding  $NM\text{-def}$   $[\text{symmetric}]$  by  $(\text{auto simp: filter-mset-empty-conv})$ 
  moreover have  $\langle \text{filter-mset } f\ M = \text{filter-mset } g\ M \rangle$ 
    by  $(\text{rule filter-mset-cong})$   $(\text{use } M\text{-eq in auto})$ 
  ultimately show ?thesis
    unfolding  $N$  by simp
qed

lemma notin-filter-mset:
   $\langle K \notin \# C \implies \text{filter-mset } P\ C = \text{filter-mset } (\lambda L. P\ L \wedge L \neq K)\ C \rangle$ 
  by  $(\text{rule filter-mset-cong})$  auto

lemma distinct-mset-add-mset-filter:
  assumes  $\langle \text{distinct-mset } C \rangle$  and  $\langle L \in \# C \rangle$  and  $\langle \neg P\ L \rangle$ 
  shows  $\langle \text{add-mset } L\ (\text{filter-mset } P\ C) = \text{filter-mset } (\lambda x. P\ x \vee x = L)\ C \rangle$ 
  using assms
proof  $(\text{induction } C)$ 
  case empty
    then show ?case by simp
next
  case  $(\text{add } x\ C)$  note  $\text{dist} = \text{this}(2)$  and  $LC = \text{this}(3)$  and  $P[\text{simp}] = \text{this}(4)$  and  $- = \text{this}$ 
  then have  $IH: \langle L \in \# C \implies \text{add-mset } L\ (\text{filter-mset } P\ C) = \{\#x \in \# C. P\ x \vee x = L\ \# \}$  by auto
  show ?case
  proof  $(\text{cases } \langle x = L \rangle)$ 
  case  $[\text{simp}]: \text{True}$ 
    have  $\langle \text{filter-mset } P\ C = \{\#x \in \# C. P\ x \vee x = L\ \# \} \rangle$ 
      by  $(\text{rule filter-mset-cong2})$   $(\text{use dist in auto})$ 
    then show ?thesis
      by auto
  next
  case False
    then show ?thesis
      using  $IH\ LC$  by auto
qed
qed

lemma set-mset-set-mset-eq-iff:  $\langle \text{set-mset } A = \text{set-mset } B \longleftrightarrow (\forall a \in \# A. a \in \# B) \wedge (\forall a \in \# B. a \in \# A) \rangle$ 
  by blast

lemma remove1-mset-union-distrib:
   $\langle \text{remove1-mset } a\ (M \cup \# N) = \text{remove1-mset } a\ M \cup \# \text{remove1-mset } a\ N \rangle$ 
  by  $(\text{auto simp: multiset-eq-iff})$ 

lemma member-add-mset:  $\langle a \in \# \text{add-mset } x\ xs \longleftrightarrow a = x \vee a \in \# xs \rangle$ 
  by simp

```



```

lemma sup-union-right-if:
   $\langle N \cup \# \text{ add-mset } x \ M =$ 
     $(\text{if } x \notin \# N \text{ then add-mset } x \ (N \cup \# M) \text{ else add-mset } x \ (\text{remove1-mset } x \ N \cup \# M)) \rangle$ 
  by (auto simp: sup-union-right2)

lemma same-mset-distinct-iff:
   $\langle \text{mset } M = \text{mset } M' \implies \text{distinct } M \longleftrightarrow \text{distinct } M' \rangle$ 
  by (auto simp: distinct-mset-mset-distinct[symmetric] simp del: distinct-mset-mset-distinct)

lemma inj-on-image-mset-eq-iff:
  assumes inj:  $\langle \text{inj-on } f \ (\text{set-mset } (M + M')) \rangle$ 
  shows  $\langle \text{image-mset } f \ M' = \text{image-mset } f \ M \longleftrightarrow M' = M \rangle$  (is  $\langle ?A = ?B \rangle$ )
proof
  assume  $?B$ 
  then show  $?A$  by auto
next
  assume  $?A$ 
  then show  $?B$ 
    using inj
  proof(induction M arbitrary: M')
    case empty
    then show  $?case$  by auto
  next
    case (add x M) note  $IH = \text{this}(1)$  and  $H = \text{this}(2)$  and  $\text{inj} = \text{this}(3)$ 

    obtain  $M1 \ x'$  where
       $M': \langle M' = \text{add-mset } x' \ M1 \rangle$  and
       $f\text{-}xx': \langle f \ x' = f \ x \rangle$  and
       $M1\text{-}M: \langle \text{image-mset } f \ M1 = \text{image-mset } f \ M \rangle$ 
      using  $H$  by (auto dest!: mset-map-invR)
    moreover have  $\langle M1 = M \rangle$ 
      apply (rule IH[OF M1-M])
      using inj by (auto simp: M')
    moreover have  $\langle x = x' \rangle$ 
      using inj f-xx' by (auto simp: M')
    ultimately show  $?case$  by fast
  qed
qed

lemma inj-image-mset-eq-iff:
  assumes inj:  $\langle \text{inj } f \rangle$ 
  shows  $\langle \text{image-mset } f \ M' = \text{image-mset } f \ M \longleftrightarrow M' = M \rangle$ 
  using inj-on-image-mset-eq-iff[of f M' M] assms
  by (simp add: inj-eq multiset.inj-map)

lemma singleton-eq-image-mset-iff:  $\langle \{ \# a \# \} = f \text{' } \# \ NE' \longleftrightarrow (\exists b. NE' = \{ \# b \# \} \wedge f \ b = a) \rangle$ 
  by (cases NE') auto

lemma image-mset-If-eq-notin:
   $\langle C \notin \# A \implies \{ \# f \ (\text{if } x = C \text{ then } a \ x \text{ else } b \ x). \ x \in \# A \# \} = \{ \# f(b \ x). \ x \in \# A \# \} \rangle$ 
  by (induction A) auto

lemma finite-mset-set-inter:
   $\langle \text{finite } A \implies \text{finite } B \implies \text{mset-set } (A \cap B) = \text{mset-set } A \cap \# \text{ mset-set } B \rangle$ 
  apply (induction A rule: finite-induct)

```

```

subgoal by auto
subgoal for a A
  apply (cases ⟨a ∈ B⟩; cases ⟨a ∈# mset-set B⟩)
  using multi-member-split[of a ⟨mset-set B⟩]
  by (auto simp: mset-set.insert-remove)
done

lemma distinct-mset-inter-remdups-mset:
  assumes dist: ⟨distinct-mset A⟩
  shows ⟨A ∩# remdups-mset B = A ∩# B⟩
proof -
  have [simp]: ⟨A' ∩# remove1-mset a (remdups-mset Aa) = A' ∩# Aa⟩
  if
    ⟨A' ∩# remdups-mset Aa = A' ∩# Aa⟩ and
    ⟨a ∉# A'⟩ and
    ⟨a ∈# Aa⟩
  for A' Aa :: ⟨'a multiset⟩ and a
  by (metis insert-DiffM inter-add-right1 set-mset-remdups-mset that)

show ?thesis
  using dist
  apply (induction A)
  subgoal by auto
  subgoal for a A'
    apply (cases ⟨a ∈# B⟩)
    using multi-member-split[of a ⟨B⟩] multi-member-split[of a ⟨A⟩]
    by (auto simp: mset-set.insert-remove)
  done
qed

lemma mset-butlast-update-last[simp]:
  ⟨w < length xs ⟹ mset (butlast (xs[w := last (xs)])) = remove1-mset (xs ! w) (mset xs)⟩
  by (cases ⟨xs = []⟩)
  (auto simp add: last-list-update-to-last mset-butlast-remove1-mset mset-update)

lemma in-multiset-ge-Max: ⟨a ∈# N ⟹ a > Max (insert 0 (set-mset N)) ⟹ False⟩
  by (simp add: leD)

lemma distinct-mset-set-mset-remove1-mset:
  ⟨distinct-mset M ⟹ set-mset (remove1-mset c M) = set-mset M - {c}⟩
  by (cases ⟨c ∈# M⟩) (auto dest!: multi-member-split simp: add-mset-eq-add-mset)

lemma distinct-count-msetD:
  ⟨distinct xs ⟹ count (mset xs) a = (if a ∈ set xs then 1 else 0)⟩
  unfolding distinct-count-atmost-1 by auto

lemma filter-mset-and-implied:
  ⟨(∧ ia. ia ∈# xs ⟹ Q ia ⟹ P ia) ⟹ {#ia ∈# xs. P ia ∧ Q ia#} = {#ia ∈# xs. Q ia#}⟩
  by (rule filter-mset-cong2) auto

lemma filter-mset-eq-add-msetD: ⟨filter-mset P xs = add-mset a A ⟹ a ∈# xs ∧ P a⟩
  by (induction xs arbitrary: A)
  (auto split: if-splits simp: add-mset-eq-add-mset)

lemma filter-mset-eq-add-msetD': ⟨add-mset a A = filter-mset P xs ⟹ a ∈# xs ∧ P a⟩
  using filter-mset-eq-add-msetD[of P xs a A] by auto

```

lemma *image-filter-replicate-mset*:
 $\langle \# Ca \in \# \text{ replicate-mset } m \ C. \ P \ Ca \# \rangle = (\text{if } P \ C \text{ then } \text{replicate-mset } m \ C \text{ else } \{\#\})$
by (*induction m*) *auto*

lemma *size-Union-mset-image-mset*:
 $\langle \text{size } (\bigcup \# A) = (\sum i \in \# A. \text{size } i) \rangle$
by (*induction A*) *auto*

lemma *image-mset-minus-inj-on*:
 $\langle \text{inj-on } f \ (\text{set-mset } A \cup \text{set-mset } B) \implies f \ ' \# (A - B) = f \ ' \# A - f \ ' \# B \rangle$
apply (*induction A arbitrary: B*)
subgoal by *auto*
subgoal for $x \ A \ B$
apply (*cases* $\langle x \in \# B \rangle$)
apply (*auto dest!: multi-member-split*)
apply (*subst diff-add-mset-swap*)
apply *auto*
done
done

lemma *filter-mset-mono-subset*:
 $\langle A \subseteq \# B \implies (\bigwedge x. x \in \# A \implies P \ x \implies Q \ x) \implies \text{filter-mset } P \ A \subseteq \# \text{filter-mset } Q \ B \rangle$
by (*metis multiset-filter-mono multiset-filter-mono2 subset-mset.order-trans*)

lemma *mset-inter-empty-set-mset*: $\langle M \cap \# xc = \{\#\} \longleftrightarrow \text{set-mset } M \cap \text{set-mset } xc = \{\} \rangle$
by (*induction xc*) *auto*

lemma *sum-mset-mset-set-sum-set*:
 $\langle (\sum A \in \# \text{mset-set } As. \ f \ A) = (\sum A \in As. \ f \ A) \rangle$
apply (*cases* $\langle \text{finite } As \rangle$)
by (*induction As rule: finite-induct*) *auto*

lemma *sum-mset-sum-count*:
 $\langle (\sum A \in \# As. \ f \ A) = (\sum A \in \text{set-mset } As. \ \text{count } As \ A * f \ A) \rangle$
proof (*induction As*)
case *empty*
then show ?*case* **by** *auto*
next
case (*add x As*)
define n **where** $\langle n = \text{count } As \ x \rangle$
define As' **where** $\langle As' \equiv \text{removeAll-mset } x \ As \rangle$
have $As: \langle As = As' + \text{replicate-mset } n \ x \rangle$
by (*auto simp: As'-def n-def intro!: multiset-eqI*)
have [*simp*]: $\langle \text{set-mset } As' - \{x\} = \text{set-mset } As' \rangle \langle \text{count } As' \ x = 0 \rangle \langle x \notin \# As' \rangle$
unfolding $As'\text{-def}$
by *auto*
have $\langle (\sum A \in \text{set-mset } As'. \$
 $(\text{if } x = A \text{ then } \text{Suc } (\text{count } (As' + \text{replicate-mset } n \ x) \ A)$
 $\text{else } \text{count } (As' + \text{replicate-mset } n \ x) \ A) *$
 $f \ A) =$
 $(\sum A \in \text{set-mset } As'. \$
 $(\text{count } (As' + \text{replicate-mset } n \ x) \ A) *$
 $f \ A) \rangle$
by (*rule sum.cong*) *auto*

then show ?case using add by (auto simp: As sum.insert-remove)
qed

lemma *sum-mset-inter-restrict*:

$\langle (\sum x \in \# \text{ filter-mset } P \ M. f \ x) = (\sum x \in \# \ M. \text{ if } P \ x \text{ then } f \ x \text{ else } 0) \rangle$
by (induction M) auto

lemma *sumset-diff-constant-left*:

assumes $\langle \bigwedge x. x \in \# \ A \implies f \ x \leq n \rangle$
shows $\langle (\sum x \in \# \ A. n - f \ x) = \text{size } A * n - (\sum x \in \# \ A. f \ x) \rangle$

proof -

have $\langle \text{size } A * n \geq (\sum x \in \# \ A. f \ x) \rangle$
if $\langle \bigwedge x. x \in \# \ A \implies f \ x \leq n \rangle$ for A
using that
by (induction A) (force simp: ac-simps)+
then show ?thesis
using assms
by (induction A) (auto simp: ac-simps)

qed

lemma *mset-set-eq-mset-iff*: $\langle \text{finite } x \implies \text{mset-set } x = \text{mset } xs \longleftrightarrow \text{distinct } xs \wedge x = \text{set } xs \rangle$

apply (auto simp flip: distinct-mset-mset-distinct eq-commute[of - $\langle \text{mset-set } _ \rangle$])
simp: distinct-mset-mset-set mset-set-set
apply (metis finite-set-mset-mset-set set-mset-mset)
apply (metis finite-set-mset-mset-set set-mset-mset)
done

lemma *distinct-mset-iff*:

$\langle \neg \text{distinct-mset } C \longleftrightarrow (\exists a \ C'. C = \text{add-mset } a \ (\text{add-mset } a \ C')) \rangle$
by (metis (no-types, hide-lams) One-nat-def
count-add-mset distinct-mset-add-mset distinct-mset-def
member-add-mset mset-add not-in-iff)

lemma *diff-add-mset-remove1*: $\langle \text{NO-MATCH } \{\#\} \ N \implies M - \text{add-mset } a \ N = \text{remove1-mset } a \ (M - N) \rangle$

by auto

lemma *remdups-mset-sum-subset*: $\langle C \subseteq \# \ C' \implies \text{remdups-mset } (C + C') = \text{remdups-mset } C' \rangle$

$\langle C \subseteq \# \ C' \implies \text{remdups-mset } (C' + C) = \text{remdups-mset } C' \rangle$
apply (metis remdups-mset-def set-mset-mono set-mset-union sup.absorb-iff2)
by (metis add.commute le-iff-sup remdups-mset-def set-mset-mono set-mset-union)

lemma *distinct-mset-subset-iff-remdups*:

$\langle \text{distinct-mset } a \implies a \subseteq \# \ b \longleftrightarrow a \subseteq \# \ \text{remdups-mset } b \rangle$
by (simp add: distinct-mset-inter-remdups-mset subset-mset.le-iff-inf)

lemma *remdups-mset-subset-add-mset*: $\langle \text{remdups-mset } C' \subseteq \# \ \text{add-mset } L \ C' \rangle$

by (meson distinct-mset-remdups-mset distinct-mset-subset-iff-remdups subset-mset.order-refl
subset-mset-trans-add-mset)

1.4 Finite maps and multisets

Finite sets and multisets

abbreviation $mset\text{-}fset :: \langle 'a\ fset \Rightarrow 'a\ multiset \rangle$ **where**
 $\langle mset\text{-}fset\ N \equiv mset\text{-}set\ (fset\ N) \rangle$

definition $fset\text{-}mset :: \langle 'a\ multiset \Rightarrow 'a\ fset \rangle$ **where**
 $\langle fset\text{-}mset\ N \equiv Abs\text{-}fset\ (set\text{-}mset\ N) \rangle$

lemma $fset\text{-}mset\text{-}mset\text{-}fset$: $\langle fset\text{-}mset\ (mset\text{-}fset\ N) = N \rangle$
by $(auto\ simp: fset.fset\text{-}inverse\ fset\text{-}mset\text{-}def)$

lemma $mset\text{-}fset\text{-}fset\text{-}mset[simp]$:
 $\langle mset\text{-}fset\ (fset\text{-}mset\ N) = remdups\text{-}mset\ N \rangle$
by $(auto\ simp: fset.fset\text{-}inverse\ fset\text{-}mset\text{-}def\ Abs\text{-}fset\text{-}inverse\ remdups\text{-}mset\text{-}def)$

lemma $in\text{-}mset\text{-}fset\text{-}fmember[simp]$: $\langle x \in\# mset\text{-}fset\ N \longleftrightarrow x \in | N \rangle$
by $(auto\ simp: fmember.rep\text{-}eq)$

lemma $in\text{-}fset\text{-}mset\text{-}mset[simp]$: $\langle x \in | fset\text{-}mset\ N \longleftrightarrow x \in\# N \rangle$
by $(auto\ simp: fmember.rep\text{-}eq\ fset\text{-}mset\text{-}def\ Abs\text{-}fset\text{-}inverse)$

Finite map and multisets

Roughly the same as ran and dom , but with duplication in the content (unlike their finite sets counterpart) while still working on finite domains (unlike a function mapping). Remark that $dom\text{-}m$ (the keys) does not contain duplicates, but we keep for symmetry (and for easier use of multiset operators as in the definition of $ran\text{-}m$).

definition $dom\text{-}m$ **where**
 $\langle dom\text{-}m\ N = mset\text{-}fset\ (fmdom\ N) \rangle$

definition $ran\text{-}m$ **where**
 $\langle ran\text{-}m\ N = the\ \text{'}\# fmlookup\ N\ \text{'}\# dom\text{-}m\ N \rangle$

lemma $dom\text{-}m\text{-}fmdrop[simp]$: $\langle dom\text{-}m\ (fmdrop\ C\ N) = remove1\text{-}mset\ C\ (dom\text{-}m\ N) \rangle$
unfolding $dom\text{-}m\text{-}def$
by $(cases\ \langle C \in | fmdom\ N \rangle)$
 $(auto\ simp: mset\text{-}set.remove\ fmmember.rep\text{-}eq)$

lemma $dom\text{-}m\text{-}fmdrop\text{-}All$: $\langle dom\text{-}m\ (fmdrop\ C\ N) = removeAll\text{-}mset\ C\ (dom\text{-}m\ N) \rangle$
unfolding $dom\text{-}m\text{-}def$
by $(cases\ \langle C \in | fmdom\ N \rangle)$
 $(auto\ simp: mset\text{-}set.remove\ fmmember.rep\text{-}eq)$

lemma $dom\text{-}m\text{-}fmupd[simp]$: $\langle dom\text{-}m\ (fmupd\ k\ C\ N) = add\text{-}mset\ k\ (remove1\text{-}mset\ k\ (dom\text{-}m\ N)) \rangle$
unfolding $dom\text{-}m\text{-}def$
by $(cases\ \langle k \in | fmdom\ N \rangle)$
 $(auto\ simp: mset\text{-}set.remove\ fmmember.rep\text{-}eq\ mset\text{-}set.insert\text{-}remove)$

lemma $distinct\text{-}mset\text{-}dom$: $\langle distinct\text{-}mset\ (dom\text{-}m\ N) \rangle$
by $(simp\ add: distinct\text{-}mset\text{-}mset\text{-}set\ dom\text{-}m\text{-}def)$

lemma $in\text{-}dom\text{-}m\text{-}lookup\text{-}iff$: $\langle C \in\# dom\text{-}m\ N' \longleftrightarrow fmlookup\ N'\ C \neq None \rangle$
by $(auto\ simp: dom\text{-}m\text{-}def\ fmdom.rep\text{-}eq\ fmlookup\text{-}dom'\text{-}iff)$

lemma *in-dom-in-ran-m*[simp]: $\langle i \in \# \text{ dom-m } N \implies \text{the } (\text{fmlookup } N \ i) \in \# \text{ ran-m } N \rangle$
by (auto simp: ran-m-def)

lemma *fmupd-same*[simp]:
 $\langle x1 \in \# \text{ dom-m } x1aa \implies \text{fmupd } x1 \ (\text{the } (\text{fmlookup } x1aa \ x1)) \ x1aa = x1aa \rangle$
by (metis fmap-ext fmupd-lookup in-dom-m-lookup-iff option.collapse)

lemma *ran-m-fmempty*[simp]: $\langle \text{ran-m fmempty} = \{\#\} \rangle$ **and**
 $\text{dom-m-fmempty}[simp]: \langle \text{dom-m fmempty} = \{\#\} \rangle$
by (auto simp: ran-m-def dom-m-def)

lemma *fmrestrict-set-fmupd*:
 $\langle a \in xs \implies \text{fmrestrict-set } xs \ (\text{fmupd } a \ C \ N) = \text{fmupd } a \ C \ (\text{fmrestrict-set } xs \ N) \rangle$
 $\langle a \notin xs \implies \text{fmrestrict-set } xs \ (\text{fmupd } a \ C \ N) = \text{fmrestrict-set } xs \ N \rangle$
by (auto simp: fmfilter-alt-defs)

lemma *fset-fmdom-fmrestrict-set*:
 $\langle \text{fset } (\text{fmdom } (\text{fmrestrict-set } xs \ N)) = \text{fset } (\text{fmdom } N) \cap xs \rangle$
by (auto simp: fmfilter-alt-defs)

lemma *dom-m-fmrestrict-set*: $\langle \text{dom-m } (\text{fmrestrict-set } (\text{set } xs) \ N) = \text{mset } xs \cap \# \text{ dom-m } N \rangle$
using *fset-fmdom-fmrestrict-set*[of $\langle \text{set } xs \rangle \ N$] *distinct-mset-dom*[of N]
distinct-mset-inter-remdups-mset[of $\langle \text{mset-fset } (\text{fmdom } N) \rangle \ \langle \text{mset } xs \rangle$]
by (auto simp: dom-m-def fset-mset-mset-fset finite-mset-set-inter multiset-inter-commute
remdups-mset-def)

lemma *dom-m-fmrestrict-set'*: $\langle \text{dom-m } (\text{fmrestrict-set } xs \ N) = \text{mset-set } (xs \cap \text{set-mset } (\text{dom-m } N)) \rangle$
using *fset-fmdom-fmrestrict-set*[of $\langle xs \rangle \ N$] *distinct-mset-dom*[of N]
by (auto simp: dom-m-def fset-mset-mset-fset finite-mset-set-inter multiset-inter-commute
remdups-mset-def)

lemma *indom-mI*: $\langle \text{fmlookup } m \ x = \text{Some } y \implies x \in \# \text{ dom-m } m \rangle$
by (drule fmdomI) (auto simp: dom-m-def fmlookup.rep-eq)

lemma *fmupd-fmdrop-id*:
assumes $\langle k \in \# \text{ fmdom } N' \rangle$
shows $\langle \text{fmupd } k \ (\text{the } (\text{fmlookup } N' \ k)) \ (\text{fmdrop } k \ N') = N' \rangle$
proof –
have [simp]: $\langle \text{map-upd } k \ (\text{the } (\text{fmlookup } N' \ k)) \ (\lambda x. \text{if } x \neq k \text{ then } \text{fmlookup } N' \ x \text{ else None}) = \text{map-upd } k \ (\text{the } (\text{fmlookup } N' \ k)) \ (\text{fmlookup } N') \rangle$
by (auto intro!: ext simp: map-upd-def)
have [simp]: $\langle \text{map-upd } k \ (\text{the } (\text{fmlookup } N' \ k)) \ (\text{fmlookup } N') = \text{fmlookup } N' \rangle$
using *assms*
by (auto intro!: ext simp: map-upd-def)
have [simp]: $\langle \text{finite } (\text{dom } (\lambda x. \text{if } x = k \text{ then None else } \text{fmlookup } N' \ x)) \rangle$
by (subst dom-if) auto
show ?thesis
apply (auto simp: fmupd-def fmupd.abs-eq[symmetric])
unfolding *fmlookup-drop*
apply (simp add: fmlookup-inverse)
done
qed


```

have  $\langle C \notin \# \text{ dom-}m \ N \rangle$ 
  using  $NC \text{ distinct-mset-dom[of } N]$  unfolding  $N\text{-}N'$  by auto
then show ?thesis
  by (auto simp: N-N' ran-m-def mset-set.insert-remove image-mset-remove1-mset-if
    intro!: image-mset-cong)
qed

```

```

lemma ran-m-mapsto-upd-notin:
  assumes  $NC: \langle C \notin \# \text{ dom-}m \ N \rangle$ 
  shows  $\langle \text{ran-}m \ (\text{fmupd } C \ C' \ N) = \text{add-mset } C' \ (\text{ran-}m \ N) \rangle$ 
  using  $NC$ 
  by (auto simp: ran-m-def mset-set.insert-remove image-mset-remove1-mset-if
    intro!: image-mset-cong split: if-splits)

```

```

lemma ran-m-fmdrop:
   $\langle C \in \# \text{ dom-}m \ N \implies \text{ran-}m \ (\text{fmdrop } C \ N) = \text{remove1-mset } (\text{the } (\text{fmlookup } N \ C)) \ (\text{ran-}m \ N) \rangle$ 
  using  $\text{distinct-mset-dom[of } N]$ 
  by (cases  $\langle \text{fmlookup } N \ C \rangle$ 
    (auto simp: ran-m-def image-mset-If-eq-notin[of } C - \langle \lambda x. \text{fst } (\text{the } x) \rangle]
      dest!: multi-member-split
      intro!: filter-mset-cong2 image-mset-cong2)

```

```

lemma ran-m-fmdrop-notin:
   $\langle C \notin \# \text{ dom-}m \ N \implies \text{ran-}m \ (\text{fmdrop } C \ N) = \text{ran-}m \ N \rangle$ 
  using  $\text{distinct-mset-dom[of } N]$ 
  by (auto simp: ran-m-def image-mset-If-eq-notin[of } C - \langle \lambda x. \text{fst } (\text{the } x) \rangle]
    dest!: multi-member-split
    intro!: filter-mset-cong2 image-mset-cong2)

```

```

lemma ran-m-fmdrop-If:
   $\langle \text{ran-}m \ (\text{fmdrop } C \ N) = (\text{if } C \in \# \text{ dom-}m \ N \text{ then } \text{remove1-mset } (\text{the } (\text{fmlookup } N \ C)) \ (\text{ran-}m \ N) \text{ else } \text{ran-}m \ N) \rangle$ 
  using  $\text{distinct-mset-dom[of } N]$ 
  by (auto simp: ran-m-def image-mset-If-eq-notin[of } C - \langle \lambda x. \text{fst } (\text{the } x) \rangle]
    dest!: multi-member-split
    intro!: filter-mset-cong2 image-mset-cong2)

```

Compact domain for finite maps

packed is a predicate to indicate that the domain of finite mapping starts at 1 and does not contain holes. We used it in the SAT solver for the mapping from indexes to clauses, to ensure that there not holes and therefore giving an upper bound on the highest key.

TODO KILL!

```

definition Max-dom where
   $\langle \text{Max-dom } N = \text{Max } (\text{set-mset } (\text{add-mset } 0 \ (\text{dom-}m \ N))) \rangle$ 

```

```

definition packed where
   $\langle \text{packed } N \longleftrightarrow \text{dom-}m \ N = \text{mset } [1..<\text{Suc } (\text{Max-dom } N)] \rangle$ 

```

Marking this rule as simp is not compatible with unfolding the definition of packed when marked as:

```

lemma Max-dom-empty:  $\langle \text{dom-}m \ b = \{\#\} \implies \text{Max-dom } b = 0 \rangle$ 
  by (auto simp: Max-dom-def)

```


lemma *Max-dom-fmempty*: $\langle \text{Max-dom fmempty} = 0 \rangle$
by (*auto simp: Max-dom-empty*)

lemma *packed-empty[simp]*: $\langle \text{packed fmempty} \rangle$
by (*auto simp: packed-def Max-dom-empty*)

lemma *packed-Max-dom-size*:
assumes *p*: $\langle \text{packed } N \rangle$
shows $\langle \text{Max-dom } N = \text{size } (\text{dom-m } N) \rangle$

proof –

have *1*: $\langle \text{dom-m } N = \text{mset } [1..<\text{Suc } (\text{Max-dom } N)] \rangle$
using *p* **unfolding** *packed-def* *Max-dom-def[symmetric]* .
have $\langle \text{size } (\text{dom-m } N) = \text{size } (\text{mset } [1..<\text{Suc } (\text{Max-dom } N)]) \rangle$
unfolding *1* ..
also have $\langle \dots = \text{length } [1..<\text{Suc } (\text{Max-dom } N)] \rangle$
unfolding *size-mset* ..
also have $\langle \dots = \text{Max-dom } N \rangle$
unfolding *length-upt* **by** *simp*
finally show *?thesis*
by *simp*

qed

lemma *Max-dom-le*:
 $\langle L \in \# \text{ dom-m } N \implies L \leq \text{Max-dom } N \rangle$
by (*auto simp: Max-dom-def*)

lemma *remove1-mset-ge-Max-some*: $\langle a > \text{Max-dom } b \implies \text{remove1-mset } a \text{ } (\text{dom-m } b) = \text{dom-m } b \rangle$
by (*auto simp: Max-dom-def remove-1-mset-id-iff-notin*
dest!: multi-member-split)

lemma *Max-dom-fmupd-irrel*:
 $\langle (a :: 'a :: \{\text{zero}, \text{linorder}\}) > \text{Max-dom } M \implies \text{Max-dom } (\text{fmupd } a \text{ } C \text{ } M) = \max a \text{ } (\text{Max-dom } M) \rangle$
by (*cases* $\langle \text{dom-m } M \rangle$)
(auto simp: Max-dom-def remove1-mset-ge-Max-some ac-simps)

lemma *Max-dom-alt-def*: $\langle \text{Max-dom } b = \text{Max } (\text{insert } 0 \text{ } (\text{set-mset } (\text{dom-m } b))) \rangle$
unfolding *Max-dom-def* **by** *auto*

lemma *Max-insert-Suc-Max-dim-dom[simp]*:
 $\langle \text{Max } (\text{insert } (\text{Suc } (\text{Max-dom } b)) \text{ } (\text{set-mset } (\text{dom-m } b))) = \text{Suc } (\text{Max-dom } b) \rangle$
unfolding *Max-dom-alt-def*
by (*cases* $\langle \text{set-mset } (\text{dom-m } b) = \{\} \rangle$) *auto*

lemma *size-dom-m-Max-dom*:
 $\langle \text{size } (\text{dom-m } N) \leq \text{Suc } (\text{Max-dom } N) \rangle$

proof –

have $\langle \text{dom-m } N \subseteq \# \text{ mset-set } \{0..<\text{Suc } (\text{Max-dom } N)\} \rangle$
apply (*rule distinct-finite-set-mset-subseteq-iff[THEN iffD1]*)
subgoal by (*auto simp: distinct-mset-dom*)
subgoal by *auto*
subgoal by (*auto dest: Max-dom-le*)
done

from *size-mset-mono[OF this]* **show** *?thesis* **by** *auto*

qed

lemma *Max-atLeastLessThan-plus*: $\langle \text{Max } \{(a::\text{nat}) ..< a+n\} = (\text{if } n = 0 \text{ then } \text{Max } \{\} \text{ else } a+n-1) \rangle$

apply (*induction n arbitrary: a*)
subgoal by *auto*
subgoal for *n a*
by (*cases n*)
*(auto simp: image-Suc-atLeastLessThan[symmetric] mono-Max-commute[symmetric] mono-def
atLeastLessThanSuc
simp del: image-Suc-atLeastLessThan)*
done

lemma *Max-atLeastLessThan*: $\langle \text{Max } \{(a::\text{nat}) \dots b\} = (\text{if } b \leq a \text{ then Max } \{\} \text{ else } b - 1) \rangle$
using *Max-atLeastLessThan-plus[of a (b-a)]*
by *auto*

lemma *Max-insert-Max-dom-into-packed*:
 $\langle \text{Max } (\text{insert } (\text{Max-dom } bc) \{\text{Suc } 0..<\text{Max-dom } bc\}) = \text{Max-dom } bc \rangle$
by (*cases (Max-dom bc); cases (Max-dom bc - 1)*)
(auto simp: Max-dom-empty Max-atLeastLessThan)

lemma *packed0-fmud-Suc-Max-dom*: $\langle \text{packed } b \implies \text{packed } (\text{fmupd } (\text{Suc } (\text{Max-dom } b)) \ C \ b) \rangle$
by (*auto simp: packed-def remove1-mset-ge-Max-some Max-dom-fmupd-irrel max-def*)

lemma *ge-Max-dom-notin-dom-m*: $\langle a > \text{Max-dom } ao \implies a \notin \# \text{ dom-m } ao \rangle$
by (*auto simp: Max-dom-def*)

lemma *packed-in-dom-mI*: $\langle \text{packed } bc \implies j \leq \text{Max-dom } bc \implies 0 < j \implies j \in \# \text{ dom-m } bc \rangle$
by (*auto simp: packed-def*)

lemma *mset-fset-empty-iff*: $\langle \text{mset-fset } a = \{\#\} \longleftrightarrow a = \text{fempty} \rangle$
by (*cases a*) (*auto simp: mset-set-empty-iff*)

lemma *dom-m-empty-iff[iff]*:
 $\langle \text{dom-m } NU = \{\#\} \longleftrightarrow NU = \text{fmempty} \rangle$
by (*cases NU*) (*auto simp: dom-m-def mset-fset-empty-iff mset-set.insert-remove*)

lemma *nat-power-div-base*:
fixes *k :: nat*
assumes $0 < m \ 0 < k$
shows $k \wedge m \text{ div } k = (k::\text{nat}) \wedge (m - \text{Suc } 0)$
proof –
have *eq*: $k \wedge m = k \wedge ((m - \text{Suc } 0) + \text{Suc } 0)$
by (*simp add: assms*)
show *?thesis*
using *assms* **by** (*simp only: power-add eq*) *auto*
qed

lemma *eq-insertD*: $\langle A = \text{insert } a \ B \implies a \in A \wedge B \subseteq A \rangle$
by *auto*

lemma *length-list-ge2*: $\langle \text{length } S \geq 2 \longleftrightarrow (\exists a \ b \ S'. S = [a, b] @ S') \rangle$
apply (*cases S*)
apply (*simp; fail*)
apply (*rename-tac a S'*)
apply (*case-tac S'*)

```

by simp-all

end

theory Explorer
imports Main
keywords explore explore-have explore-lemma explore-context :: diag
begin

```

1.4.1 Explore command

This theory contains the definition of four tactics that work on goals and put them in an Isar proof:

- *explore* generates an assume-show proof block
- *explore-have* generates an have-if-for block
- *lemma* generates a lemma-fixes-assumes-shows block
- *explore-context* is mostly meaningful on several goals: it combines assumptions and variables between the goals to generate a context-fixes-begin-end bloc with lemmas in the middle. This tactic is mostly useful when a lot of assumption and proof steps would be shared.

If you use any of those tactic or have an idea how to improve it, please send an email to the current maintainer!

```

ML <
signature EXPLORER-LIB =
sig
  datatype explorer-quote = QUOTES | GUILLEMOTS
  val set-default-raw-param: theory -> theory
  val default-raw-params: theory -> string * explorer-quote
  val switch-to-cartouches: theory -> theory
  val switch-to-quotes: theory -> theory
end

structure Explorer-Lib : EXPLORER-LIB =
struct
  datatype explorer-quote = QUOTES | GUILLEMOTS
  type raw-param = string * explorer-quote
  val default-params = (explorer-quotes, QUOTES)

  structure Data = Theory-Data
  (
    type T = raw-param list
    val empty = single default-params
    val extend = I
    fun merge data : T = AList.merge (op =) (K true) data
  )

  fun set-default-raw-param thy =
    thy |> Data.map (AList.update (op =) default-params)

```

```

fun switch-to-quotes thy =
  thy |> Data.map (AList.update (op =) (explorer-quotes, QUOTES))

fun switch-to-cartouches thy =
  thy |> Data.map (AList.update (op =) (explorer-quotes, GUILLEMOTS))

fun default-raw-params thy =
  Data.get thy |> hd

end
>

setup Explorer-Lib.set-default-raw-param

ML <
  Explorer-Lib.default-raw-params @{theory}
>

ML <

signature EXPLORER =
sig
  datatype explore-kind = HAVE-IF | ASSUME-SHOW | ASSUMES-SHOWS | CONTEXT
  val explore: explore-kind -> Toplevel.state -> Proof.state
end

structure Explorer: EXPLORER =
struct
  datatype explore-kind = HAVE-IF | ASSUME-SHOW | ASSUMES-SHOWS | CONTEXT

  fun split-clause t =
    let
      val (fixes, horn) = funpow-1 yield (length (Term.strip-all-vars t)) Logic.dest-all t;
      val assms = Logic.strip-imp-prems horn;
      val shows = Logic.strip-imp-concl horn;
    in (fixes, assms, shows) end;

  fun space-implode-with-line-break l =
    if length l > 1 then
      \n ^ space-implode and \n l
    else
      space-implode and \n l

  fun keyword-fix HAVE-IF = for
    | keyword-fix ASSUME-SHOW = fix
    | keyword-fix ASSUMES-SHOWS = fixes

  fun keyword-assume HAVE-IF = if
    | keyword-assume ASSUME-SHOW = assume
    | keyword-assume ASSUMES-SHOWS = assumes

  fun keyword-goal HAVE-IF =
    | keyword-goal ASSUME-SHOW = show
    | keyword-goal ASSUMES-SHOWS = shows

```

```

fun isar-skeleton ctxt aim enclosure (fixes, assms, shows) =
  let
    val kw-fix = keyword-fix aim
    val kw-assume = keyword-assume aim
    val kw-goal = keyword-goal aim
    val fixes-s = if null fixes then NONE
      else SOME (kw-fix ^ space-implode and
        (map (fn (v, T) => v ^ :: ^ enclosure (Syntax.string-of-typ ctxt T)) fixes));
    val (-, ctxt') = Variable.add-fixes (map fst fixes) ctxt;
    val assumes-s = if null assms then NONE
      else SOME (kw-assume ^ space-implode-with-line-break
        (map (enclosure o Syntax.string-of-term ctxt') assms))
    val shows-s = (kw-goal ^ (enclosure o Syntax.string-of-term ctxt') shows)
    val s =
      (case aim of
        HAVE-IF => (map-filter I [fixes-s], map-filter I [assumes-s], shows-s)
      | ASSUME-SHOW => (map-filter I [fixes-s], map-filter I [assumes-s], shows-s ^ sorry)
      | ASSUMES-SHOWS => (map-filter I [fixes-s], map-filter I [assumes-s], shows-s));
  in
    s
  end;

fun generate-text ASSUME-SHOW context enclosure clauses =
  let val lines = clauses
  in
    |> map (isar-skeleton context ASSUME-SHOW enclosure)
    |> map (fn (a, b, c) => a @ b @ [c])
    |> map cat-lines
  in
    (proof - :: separate next lines @ [qed])
  end
| generate-text HAVE-IF context enclosure clauses =
  let
    val raw-lines = map (isar-skeleton context HAVE-IF enclosure) clauses
    fun treat-line (fixes-s, assumes-s, shows-s) =
      let val combined-line = [shows-s] @ assumes-s @ fixes-s |> cat-lines
      in
        have ^ combined-line ^ \nproof -\n show ?thesis sorry\nqed
      end
    val raw-lines-with-proof-body = map treat-line raw-lines
  in
    separate \n raw-lines-with-proof-body
  end
| generate-text ASSUMES-SHOWS context enclosure clauses =
  let
    val raw-lines = map (isar-skeleton context ASSUMES-SHOWS enclosure) clauses
    fun treat-line (fixes-s, assumes-s, shows-s) =
      let val combined-line = fixes-s @ assumes-s @ [shows-s] |> cat-lines
      in
        lemma\n ^ combined-line ^ \nproof -\n show ?thesis sorry\nqed
      end
    val raw-lines-with-lemma-and-proof-body = map treat-line raw-lines
  in
    separate \n raw-lines-with-lemma-and-proof-body
  end;

```

```

datatype proof-step = ASSUMPTION of term | FIXES of (string * typ) | GOAL of term
| Step of (proof-step * proof-step)
| Branch of (proof-step list)

datatype cproof-step = cASSUMPTION of term list | cFIXES of ((string * typ) list) | cGOAL of term
| cStep of (cproof-step * cproof-step)
| cBranch of (cproof-step list)
| cLemma of ((string * typ) list * term list * term)

fun explore-context-init (FIXES var :: cgoal) =
  Step ((FIXES var), explore-context-init cgoal)
| explore-context-init (ASSUMPTION assm :: cgoal) =
  Step ((ASSUMPTION assm), explore-context-init cgoal)
| explore-context-init ([GOAL show]) =
  GOAL show
| explore-context-init (GOAL show :: cgoal) =
  Step (GOAL show, explore-context-init cgoal)

fun branch-hd-fixes-is P (Step (FIXES var, -)) = P var
| branch-hd-fixes-is P - = false

fun branch-hd-assms-is P (Step (ASSUMPTION var, -)) = P var
| branch-hd-assms-is P (Step (GOAL var, -)) = P var
| branch-hd-assms-is P (GOAL var) = P var
| branch-hd-assms-is - = false

fun find-find-pos P brs =
  let
    fun f accs (br :: brs) = if P br then SOME (accs, br, brs)
      else f (accs @ [br]) brs
    | f - [] = NONE
  in f [] brs end
(* Term.exists-subterm (curry (op =) t) *)
fun explore-context-merge (FIXES var :: cgoal) (Step (FIXES var', steps)) =
  if var = var' then
    Step (FIXES var',
      explore-context-merge cgoal steps)
  else
    Step (FIXES var', explore-context-merge cgoal steps)

| explore-context-merge (FIXES var :: cgoal) (Branch brs) =
  (case find-find-pos (branch-hd-fixes-is (curry (op =) var)) brs of
    SOME (b, (Step (fixe, st)), after) =>
      Branch (b @ Step (fixe, explore-context-merge cgoal st) :: after)
  | NONE =>
    Branch (brs @ [Step (FIXES var, explore-context-init cgoal)]))
| explore-context-merge (FIXES var :: cgoal) steps =
  Branch (steps :: [Step (FIXES var, explore-context-init cgoal)])

| explore-context-merge (ASSUMPTION assm :: cgoal) (Step (ASSUMPTION assm', steps)) =
  if assm = assm' then
    Step (ASSUMPTION assm', explore-context-merge cgoal steps)
  else
    Branch [Step (ASSUMPTION assm', steps), explore-context-init (ASSUMPTION assm :: cgoal)]
| explore-context-merge (ASSUMPTION assm :: cgoal) (Step (GOAL assm', steps)) =
  if assm = assm' then

```

```

    Step (GOAL assm', explore-context-merge cgoal steps)
  else
    Branch [Step (GOAL assm', steps), explore-context-init (ASSUMPTION assm :: cgoal)]
| explore-context-merge (ASSUMPTION assm :: cgoal) (GOAL assm') =
  if assm = assm' then
    Step (GOAL assm', explore-context-init cgoal)
  else
    Branch [GOAL assm', explore-context-init (ASSUMPTION assm :: cgoal)]
| explore-context-merge (ASSUMPTION assm :: cgoal) (Branch brs) =
  (case find-find-pos (branch-hd-assms-is (fn t => assm = (t))) brs of
    SOME (b, (Step (assm, st)), after) =>
      Branch (b @ Step (assm, explore-context-merge cgoal st) :: after)
    | SOME (b, (GOAL goal), after) =>
      Branch (b @ Step (GOAL goal, explore-context-init cgoal) :: after)
    | NONE =>
      Branch (brs @ [Step (ASSUMPTION assm, explore-context-init cgoal)]))

| explore-context-merge (GOAL show :: []) (Step (GOAL show', steps)) =
  if show = show' then
    GOAL show'
  else
    Branch [Step (GOAL show', steps), GOAL show]
| explore-context-merge clause ps =
  Branch [ps, explore-context-init clause]

fun explore-context-all (clause :: clauses) =
  fold explore-context-merge clauses (explore-context-init clause)

fun convert-proof (ASSUMPTION a) = cASSUMPTION [a]
| convert-proof (FIXES a) = cFIXES [a]
| convert-proof (GOAL a) = cGOAL a
| convert-proof (Step (a, b)) = cStep (convert-proof a, convert-proof b)
| convert-proof (Branch brs) = cBranch (map convert-proof brs)

fun compress-proof (cStep (cASSUMPTION a, cStep (cASSUMPTION b, step))) =
  compress-proof (cStep (cASSUMPTION (a @ b), compress-proof step))
| compress-proof (cStep (cFIXES a, cStep (cFIXES b, step))) =
  compress-proof (cStep (cFIXES (a @ b), compress-proof step))
| compress-proof (cStep (cFIXES a, cStep (cASSUMPTION b,
  cStep (cFIXES a', step)))) =
  compress-proof (cStep (cFIXES (a @ a'), compress-proof (cStep (cASSUMPTION b, step))))

| compress-proof (cStep (a, b)) =
  let
    val a' = compress-proof a
    val b' = compress-proof b
  in
    if a = a' andalso b = b' then cStep (a', b')
    else compress-proof (cStep (a', b'))
  end
| compress-proof (cBranch brs) =
  cBranch (map compress-proof brs)
| compress-proof a = a

fun compress-proof2 (cStep (cFIXES a, cStep (cASSUMPTION b, cGOAL g))) =
  cLemma (a, b, g)

```

```

| compress-proof2 (cStep (cASSUMPTION b, cGOAL g)) =
  cLemma ([], b, g)
| compress-proof2 (cStep (cFIXES b, cGOAL g)) =
  cLemma (b, [], g)
| compress-proof2 (cStep (a, b)) =
  cStep (compress-proof2 a, compress-proof2 b)
| compress-proof2 (cBranch brs) =
  cBranch (map compress-proof2 brs)
| compress-proof2 a = a

fun reorder-assumptions-wrt-fixes (fixes, assms, goal) =
  let
    fun depends-on t (fix) = Term.exists-subterm (curry (op =) (Term.Free fix)) t
    fun depends-on-any t (fix :: fixes) = depends-on t fix orelse depends-on-any t fixes
    | depends-on-any - [] = false
    fun insert-all-assms [] assms = map ASSUMPTION assms
    | insert-all-assms fixes [] = map FIXES fixes
    | insert-all-assms (fix :: fixes) (assm :: assms) =
      if depends-on-any assm (fix :: fixes) then
        FIXES fix :: insert-all-assms fixes (assm :: assms)
      else
        ASSUMPTION assm :: insert-all-assms (fix :: fixes) assms
  in
    insert-all-assms fixes assms @ [GOAL goal]
  end
fun generate-context-proof ctxt enclosure (cFIXES fixes) =
  let
    val kw-fix = fixes
    val fixes-s = if null fixes then NONE
      else SOME (kw-fix ^ space-implode and
        (map (fn (v, T) => v ^ :: ^ enclosure (Syntax.string-of-typ ctxt T)) fixes));
    in the-default fixes-s end
  | generate-context-proof ctxt enclosure (cASSUMPTION assms) =
    let
      val kw-assume = assumes
      val assumes-s = if null assms then NONE
        else SOME (kw-assume ^ space-implode-with-line-break
          (map (enclosure o Syntax.string-of-term ctxt) assms))
      in the-default assumes-s end
    | generate-context-proof ctxt enclosure (cGOAL shows) =
      hd (generate-text ASSUMES-SHOWS ctxt enclosure [([], [], shows)])
    | generate-context-proof ctxt enclosure (cStep (cFIXES f, cStep (cASSUMPTION assms, st))) =
      let val (-, ctxt') = Variable.add-fixes (map fst f) ctxt in
        [context ,
         generate-context-proof ctxt enclosure (cFIXES f),
         generate-context-proof ctxt' enclosure (cASSUMPTION assms),
         begin,
         generate-context-proof ctxt' enclosure st,
         end]
      |> cat-lines
    end
  | generate-context-proof ctxt enclosure (cStep (cFIXES f, st)) =
    let val (-, ctxt') = Variable.add-fixes (map fst f) ctxt in
      [context ,
       generate-context-proof ctxt enclosure (cFIXES f),
       begin,

```



```

    generate-context-proof ctxt' enclosure st,
    end]
  |> cat-lines
end
| generate-context-proof ctxt enclosure (cStep (cASSUMPTION assms, st)) =
  [context ,
   generate-context-proof ctxt enclosure (cASSUMPTION assms),
   begin,
   generate-context-proof ctxt enclosure st,
   end]
  |> cat-lines
| generate-context-proof ctxt enclosure (cStep (st, st')) =
  [generate-context-proof ctxt enclosure st,
   generate-context-proof ctxt enclosure st']
  |> cat-lines
| generate-context-proof ctxt enclosure (cBranch st) =
  separate \n (map (generate-context-proof ctxt enclosure) st)
  |> cat-lines
| generate-context-proof ctxt enclosure (cLemma (fixes, assms, shows)) =
  hd (generate-text ASSUMES-SHOWS ctxt enclosure [(fixes, assms, shows)])

(*)
  We cannot reuse ATP-Util.maybe-quote because it does not support selecting the
  quoting function. But, this is a copy-paste of that function.
*)
val unquote-tvar = perhaps (try (unprefix `))
val unquery-var = perhaps (try (unprefix ?))

val is-long-identifier = forall Symbol.Pos.is-identifier o Long-Name.explode
fun maybe-quote-with keywords quote y =
  let val s = YXML.content-of y in
    y |> ((not (is-long-identifier (unquote-tvar s)) andalso
            not (is-long-identifier (unquery-var s)))) orelse
        Keyword.is-literal keywords s) ? quote
  end

fun explore aim st =
  let
    val thy = Toplevel.theory-of st
    val quote-type = Explorer-Lib.default-raw-params thy |> snd
    val ctxt = Toplevel.presentation-context st
    val enclosure =
      (case quote-type of
        Explorer-Lib.GUILLEMOTS => maybe-quote-with (Thy-Header.get-keywords' ctxt) cartouche
      | Explorer-Lib.QUOTES => maybe-quote-with (Thy-Header.get-keywords' ctxt) quote)
    val st = Toplevel.proof-of st
    val { context, facts = -, goal } = Proof.goal st;
    val goal-props = Logic.strip-imp-prems (Thm.prop-of goal);
    val clauses = map split-clause goal-props;
    val text =
      if aim = CONTEXT then
        (clauses
         |> map reorder-assumptions-wrt-fixes
         |> explore-context-all
         |> convert-proof
         |> compress-proof

```

```

      |> compress-proof2
      |> generate-context-proof context enclosure)
    else cat-lines (generate-text aim context enclosure clauses);
    val message = Active.sendback-markup-properties [] text;
  in
    st
    |> tap (fn - => Output.information (Proof outline with cases:\n ^ message))
  end

end

val explore-cmd =
  Toplevel.keep-proof (K () o Explorer.explore Explorer.ASSUME-SHOW)

val - =
  Outer-Syntax.command @{command-keyword explore}
    explore current goal state as Isar proof
    (Scan.succeed (explore-cmd))

val explore-have-cmd =
  Toplevel.keep-proof (K () o Explorer.explore Explorer.HAVE-IF)

val - =
  Outer-Syntax.command @{command-keyword explore-have}
    explore current goal state as Isar proof with have, if and for
    (Scan.succeed explore-have-cmd)

val explore-lemma-cmd =
  Toplevel.keep-proof (K () o Explorer.explore Explorer.ASSUMES-SHOWS)

val - =
  Outer-Syntax.command @{command-keyword explore-lemma}
    explore current goal state as Isar proof with lemma, fixes, assumes, and shows
    (Scan.succeed explore-lemma-cmd)

val explore-ctxt-cmd =
  Toplevel.keep-proof (K () o Explorer.explore Explorer.CONTEXT)

val - =
  Outer-Syntax.command @{command-keyword explore-context}
    explore current goal state as Isar proof with context and lemmas
    (Scan.succeed explore-ctxt-cmd)
}

```

1.4.2 Examples

You can choose cartouches

setup *Explorer-Lib.switch-to-cartouches*

lemma

distinct xs \implies P xs \implies length (filter ($\lambda x. x = y$) xs) \leq 1 for xs

apply (*induct xs*)

explore

explore-have

explore-lemma

oops

lemma

$\bigwedge x. A1\ x \implies A2$
 $\bigwedge x\ y. A1\ x \implies B2\ y$
 $\bigwedge x\ y\ z\ s. B2\ y \implies A1\ x \implies C2\ z \implies C3\ s$
 $\bigwedge x\ y\ z\ s. B2\ y \implies A1\ x \implies C2\ z \implies C4\ s$
 $\bigwedge x\ y\ z\ s\ t. B2\ y \implies A1\ x \implies C2\ z \implies C4\ s \implies C3'\ t$
 $\bigwedge x\ y\ z\ s\ t. B2\ y \implies A1\ x \implies C2\ z \implies C4\ s \implies C4'\ t$
 $\bigwedge x\ y\ z\ s\ t. B2\ y \implies A1\ x \implies C2\ z \implies C4\ s \implies C5'\ t$

explore-context

explore-have

explore-lemma

oops

You can also choose quotes

setup *Explorer-Lib.switch-to-quotes*

lemma

$distinct\ xs \implies P\ xs \implies length\ (filter\ (\lambda x. x = y)\ xs) \leq 1$ **for** xs
apply $(induct\ xs)$

explore

explore-have

explore-lemma

oops

And switch back

setup *Explorer-Lib.switch-to-cartouches*

lemma

$distinct\ xs \implies P\ xs \implies sh \implies length\ (filter\ (\lambda x. x = y)\ xs) \leq 1$ **for** xs
apply $(induct\ xs)$

explore

explore-have

explore-lemma

oops

end