

# Formalisation of Ground Resolution and CDCL in Isabelle/HOL

Mathias Fleury and Jasmin Blanchette

January 20, 2020



# Contents

<b>1</b>	<b>More Standard Theorems</b>	<b>5</b>
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	7
1.1.4	More Well-Foundedness	8
1.2	Various Lemmas	9
1.2.1	Not-Related to Refinement or lists	9
1.3	More Lists	11
1.3.1	set, nth, tl	11
1.3.2	List Updates	12
1.3.3	Take and drop	13
1.3.4	Replicate	13
1.3.5	List intervals ( <i>upt</i> )	13
1.3.6	Lexicographic Ordering	15
1.3.7	Remove	16
1.3.8	Sorting	18
1.3.9	Distinct Multisets	18
1.3.10	Set of Distinct Multisets	18
1.3.11	Sublists	19
1.3.12	Product Case	20
1.3.13	More about <i>list-all2</i> and <i>map</i>	22
1.3.14	Multisets	23
1.4	Finite maps and multisets	28
1.4.1	Explore command	32
1.4.2	Examples	40



# 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:  
   $\langle r^{++} a b \implies r \leq s \implies s^{++} a b \rangle$   
  <proof>
```

```
lemma tranclp-mono:  
  assumes mono:  $\langle r \leq s \rangle$   
  shows  $\langle r^{++} \leq s^{++} \rangle$   
  <proof>
```

```
lemma tranclp-idemp-rel:  
   $\langle R^{++++} a b \longleftrightarrow R^{++} a b \rangle$   
  <proof>
```

Equivalent of the theorem *rtranclp-idemp*

```
lemma trancl-idemp:  $\langle (r^+)^+ = r^+ \rangle$   
  <proof>
```

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

This theorem already exists as theorem *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*:  $\langle \text{rtranclp } r \ a \ b \longleftrightarrow (a = b \vee \text{tranclp } r \ a \ b) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *tranclp-unfold-end*:  $\langle \text{tranclp } r \ a \ b \longleftrightarrow (\exists a'. \text{rtranclp } r \ a \ a' \wedge r \ a' \ b) \rangle$   
 $\langle \text{proof} \rangle$

Near duplicate of theorem *tranclpD*:

**lemma** *tranclp-unfold-begin*:  $\langle \text{tranclp } r \ a \ b \longleftrightarrow (\exists a'. r \ a \ a' \wedge \text{rtranclp } r \ a' \ b) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *trancl-set-tranclp*:  $\langle (a, b) \in \{(b, a). P \ a \ b\}^+ \longleftrightarrow P^{++} \ b \ a \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *tranclp-rtranclp-rtranclp-rel*:  $\langle R^{+++} \ a \ b \longleftrightarrow R^{**} \ a \ b \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *tranclp-rtranclp-rtranclp[simp]*:  $\langle R^{+++} = R^{**} \rangle$   
 $\langle \text{proof} \rangle$

**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**  $\langle \exists y \ y'. R^{**} \ x \ y \wedge R \ y \ y' \wedge P \ y \ y' \wedge (\lambda a \ b. R \ a \ b \wedge \neg P \ a \ b)^{**} \ y' \ z' \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *rtranclp-and-rtranclp-left*:  $\langle (\lambda a \ b. P \ a \ b \wedge Q \ a \ b)^{**} \ S \ T \Longrightarrow P^{**} \ S \ T \rangle$   
 $\langle \text{proof} \rangle$

## 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* =  $(\lambda S \ S'. \text{tranclp } \text{transf } S \ S' \wedge \text{no-step } \text{transf } S')$

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

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

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

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

**Some Properties** **lemma** *rtranclp-full1I*:  
 $\langle R^{**} \ a \ b \Longrightarrow \text{full1 } R \ b \ c \Longrightarrow \text{full1 } R \ a \ c \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *tranclp-full1I*:  
 $\langle R^{++} \ a \ b \Longrightarrow \text{full1 } R \ b \ c \Longrightarrow \text{full1 } R \ a \ c \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *rtranclp-fullI*:  
 $\langle R^{**} \ a \ b \Longrightarrow \text{full } R \ b \ c \Longrightarrow \text{full } R \ a \ c \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *tranclp-full-full1I*:

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

**lemma** *full-full1*:

$\langle R a b \implies \text{full } R b c \implies \text{full1 } R a c \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *full-unfold*:

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

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

**lemma** *full1-tranclp-relation-full*:

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

**lemma** *full-tranclp-relation-full*:

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

**lemma** *tranclp-full1-full1*:

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

**lemma** *rtranclp-full1-eq-or-full1*:

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

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

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

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

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

**assumes** *wf*:  $\langle \text{wf } \{(x, y). R y x\} \rangle$   
**shows**  $\langle \exists b. \text{full } R a b \rangle$   
 $\langle \text{proof} \rangle$

### 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*:

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

**lemma** *wfP-if-measure*: **fixes**  $f :: \langle 'a \Rightarrow nat \rangle$

**shows**  $\langle (\bigwedge x\ y. P\ x \implies g\ x\ y \implies f\ y < f\ x) \implies wf\ \{(y, x). P\ x \wedge g\ x\ y\} \rangle$   
 $\langle proof \rangle$

**lemma** *wf-if-measure-f*:

**assumes**  $\langle wf\ r \rangle$   
**shows**  $\langle wf\ \{(b, a). (f\ b, f\ a) \in r\} \rangle$   
 $\langle proof \rangle$

**lemma** *wf-wf-if-measure'*:

**assumes**  $\langle wf\ r \rangle$  **and**  $H: \langle \bigwedge x\ y. P\ x \implies g\ x\ y \implies (f\ y, f\ x) \in r \rangle$   
**shows**  $\langle wf\ \{(y, x). P\ x \wedge g\ x\ y\} \rangle$   
 $\langle proof \rangle$

**lemma** *wf-lex-less*:  $\langle wf\ (lex\ less-than) \rangle$

$\langle proof \rangle$

**lemma** *wfP-if-measure2*: **fixes**  $f :: \langle 'a \Rightarrow nat \rangle$

**shows**  $\langle (\bigwedge x\ y. P\ x\ y \implies g\ x\ y \implies f\ x < f\ y) \implies wf\ \{(x, y). P\ x\ y \wedge g\ x\ y\} \rangle$   
 $\langle proof \rangle$

**lemma** *lexord-on-finite-set-is-wf*:

**assumes**  
*P-finite*:  $\langle \bigwedge U. P\ U \longrightarrow U \in A \rangle$  **and**  
*finite*:  $\langle finite\ A \rangle$  **and**  
*wf*:  $\langle wf\ R \rangle$  **and**  
*trans*:  $\langle trans\ R \rangle$   
**shows**  $\langle wf\ \{(T, S). (P\ S \wedge P\ T) \wedge (T, S) \in lexord\ R\} \rangle$   
 $\langle proof \rangle$

**lemma** *wf-fst-wf-pair*:

**assumes**  $\langle wf\ \{(M', M). R\ M' M\} \rangle$   
**shows**  $\langle wf\ \{((M', N'), (M, N)). R\ M' M\} \rangle$   
 $\langle proof \rangle$

**lemma** *wf-snd-wf-pair*:

**assumes**  $\langle wf\ \{(M', M). R\ M' M\} \rangle$   
**shows**  $\langle wf\ \{((M', N'), (M, N)). R\ N' N\} \rangle$   
 $\langle proof \rangle$

**lemma** *wf-if-measure-f-notation2*:

**assumes**  $\langle wf\ r \rangle$   
**shows**  $\langle wf\ \{(b, h\ a) | b\ a. (f\ b, f\ (h\ a)) \in r\} \rangle$   
 $\langle proof \rangle$



**lemma** *wf-wf-if-measure'-notation2*:  
**assumes**  $\langle wf\ r \rangle$  **and**  $H: \langle \bigwedge x\ y. P\ x \implies g\ x\ y \implies (f\ y, f\ (h\ x)) \in r \rangle$   
**shows**  $\langle wf\ \{(y, h\ x) \mid y\ x. P\ x \wedge g\ x\ y\} \rangle$   
 $\langle proof \rangle$

**lemma** *power-ex-decomp*:  
**assumes**  $\langle (R \overset{\sim}{\sim} n)\ S\ T \rangle$   
**shows**  
 $\langle \exists f. f\ 0 = S \wedge f\ n = T \wedge (\forall i. i < n \longrightarrow R\ (f\ i)\ (f\ (Suc\ i))) \rangle$   
 $\langle proof \rangle$

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-number-of-transition*:  
**assumes**  
*le*:  $\langle \bigwedge i. i < n \implies ((f\ (Suc\ i)), (f\ i)) \in lexn\ less-than\ m \rangle$  **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**  $\langle n < k \wedge Suc\ m \rangle$   
 $\langle proof \rangle$

**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* =  
 $(match\ premises\ in$   
 $\quad J[thin]: \langle \exists x. \rightarrow \Rightarrow \langle rule\ exE[OF\ J] \rangle$   
 $\quad | J[thin]: \langle \rightarrow \wedge \rightarrow \Rightarrow \langle rule\ conjE[OF\ J] \rangle$   
 $\quad )$

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**  $\langle P\ n \rangle$

⟨proof⟩

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]*:

⟨ $0 < (\text{if } P \text{ then } a \text{ else } (0::\text{nat})) \longleftrightarrow P \wedge 0 < a$ ⟩  
⟨proof⟩

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

⟨proof⟩

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

⟨proof⟩

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

⟨proof⟩

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

⟨proof⟩

Bounded function have not yet been defined in Isabelle.

**definition** *bounded* ::  $(\ 'a \Rightarrow \ 'b::\text{ord}) \Rightarrow \text{bool}$  **where**

⟨ $\text{bounded } f \longleftrightarrow (\exists b. \forall n. f \ n \leq b)$ ⟩

**abbreviation** *unbounded* ::  $(\ 'a \Rightarrow \ 'b::\text{ord}) \Rightarrow \text{bool}$  **where**

⟨ $\text{unbounded } f \equiv \neg \text{bounded } f$ ⟩

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

**fixes**  $f :: \langle \text{nat} \Rightarrow \text{nat} \rangle$

**assumes**  $\text{unbound}: \langle \text{unbounded } f \rangle$

**shows**  $\langle \exists n. f \ n > m \wedge n > n_0 \rangle$

⟨proof⟩

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**  $\langle \text{bounded } f \longleftrightarrow \text{bounded } (\lambda i. k * f \ i) \rangle$

⟨proof⟩

**lemma** *bounded-const-add*:

**fixes**  $k :: \text{nat}$  **and**  $f :: \langle \text{nat} \Rightarrow \text{nat} \rangle$

**assumes**  $\langle k > 0 \rangle$

**shows**  $\langle \text{bounded } f \longleftrightarrow \text{bounded } (\lambda i. k + f \ i) \rangle$

⟨proof⟩

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⟩

## 1.3 More Lists

### 1.3.1 set, nth, tl

**lemma** *ex-geI*:  $\langle P\ n \implies n \geq m \implies \exists n \geq m. P\ n \rangle$   
*<proof>*

**lemma** *Ball-atLeastLessThan-iff*:  $\langle (\forall L \in \{a..<b\}. P\ L) \longleftrightarrow (\forall L. L \geq a \wedge L < b \longrightarrow P\ L) \rangle$   
*<proof>*

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

**lemma** *tl-drop-def*:  $\langle \text{tl}\ N = \text{drop}\ 1\ N \rangle$   
*<proof>*

**lemma** *in-set-remove1D*:  
 $\langle a \in \text{set}\ (\text{remove1}\ x\ xs) \implies a \in \text{set}\ xs \rangle$   
*<proof>*

**lemma** *take-length-takeWhile-eq-takeWhile*:  
 $\langle \text{take}\ (\text{length}\ (\text{takeWhile}\ P\ xs))\ xs = \text{takeWhile}\ P\ xs \rangle$   
*<proof>*

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

**lemma** *Collect-minus-single-Collect*:  $\langle \{x. P\ x\} - \{a\} = \{x. P\ x \wedge x \neq a\} \rangle$   
*<proof>*

**lemma** *in-set-image-subsetD*:  $\langle f\ ' A \subseteq B \implies x \in A \implies f\ x \in B \rangle$   
*<proof>*

**lemma** *mset-tl*:  
 $\langle \text{mset}\ (\text{tl}\ xs) = \text{remove1-mset}\ (\text{hd}\ xs)\ (\text{mset}\ xs) \rangle$   
*<proof>*

**lemma** *hd-list-update-If*:  
 $\langle \text{outl}' \neq [] \implies \text{hd}\ (\text{outl}'[i := w]) = (\text{if}\ i = 0\ \text{then}\ w\ \text{else}\ \text{hd}\ \text{outl}') \rangle$   
*<proof>*

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

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

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

**lemma** *butlast-Nil-iff*:  $\langle \text{butlast}\ xs = [] \longleftrightarrow \text{length}\ xs = 1 \vee \text{length}\ xs = 0 \rangle$   
*<proof>*

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

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

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

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

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

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*:  $\langle xs = \text{it } @ \ x \ \# \ \sigma \implies x \in \text{set } xs \rangle$   
 $\langle \text{proof} \rangle$

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

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

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

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

### 1.3.2 List Updates

**lemma** *tl-update-swap*:  
 $\langle i \geq 1 \implies \text{tl } (N[i := C]) = (\text{tl } N)[i-1 := C] \rangle$   
 $\langle \text{proof} \rangle$

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

**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]*:  
 $j < \text{length } xs \implies (xs[i:=x])!j = (\text{if } i = j \ \text{then } x \ \text{else } xs!j)$   
 $\langle \text{proof} \rangle$

### 1.3.3 Take and drop

**lemma** *take-2-if*:

$\langle \text{take } 2 \ C = (\text{if } C = [] \text{ then } [] \text{ else if length } C = 1 \text{ then } [\text{hd } C] \text{ else } [C!0, C!1]) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *in-set-take-conv-nth*:

$\langle x \in \text{set } (\text{take } n \ xs) \longleftrightarrow (\exists m < \text{min } n \ (\text{length } xs). \ xs \ ! \ m = x) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *in-set-dropI*:

$\langle m < \text{length } xs \implies m \geq n \implies xs \ ! \ m \in \text{set } (\text{drop } n \ xs) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *in-set-drop-conv-nth*:

$\langle x \in \text{set } (\text{drop } n \ xs) \longleftrightarrow (\exists m \geq n. \ m < \text{length } xs \wedge xs \ ! \ m = x) \rangle$   
 $\langle \text{proof} \rangle$

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

**lemma** *atd-lem*:  $\langle \text{take } n \ xs = t \implies \text{drop } n \ xs = d \implies xs = t @ d \rangle$

$\langle \text{proof} \rangle$

**lemma** *drop-take-drop-drop*:

$\langle j \geq i \implies \text{drop } i \ xs = \text{take } (j - i) \ (\text{drop } i \ xs) @ \text{drop } j \ xs \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *in-set-conv-iff*:

$\langle x \in \text{set } (\text{take } n \ xs) \longleftrightarrow (\exists i < n. \ i < \text{length } xs \wedge xs \ ! \ i = x) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *distinct-in-set-take-iff*:

$\langle \text{distinct } D \implies b < \text{length } D \implies D \ ! \ b \in \text{set } (\text{take } a \ D) \longleftrightarrow b < a \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *in-set-distinct-take-drop-iff*:

**assumes**

$\langle \text{distinct } D \rangle$  **and**

$\langle b < \text{length } D \rangle$

**shows**  $\langle D \ ! \ b \in \text{set } (\text{take } (a - \text{init}) \ (\text{drop } \text{init } D)) \longleftrightarrow (\text{init} \leq b \wedge b < a) \rangle$

$\langle \text{proof} \rangle$

### 1.3.4 Replicate

**lemma** *list-eq-replicate-iff-nempty*:

$\langle n > 0 \implies xs = \text{replicate } n \ x \longleftrightarrow n = \text{length } xs \wedge \text{set } xs = \{x\} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *list-eq-replicate-iff*:

$\langle xs = \text{replicate } n \ x \longleftrightarrow (n = 0 \wedge xs = []) \vee (n = \text{length } xs \wedge \text{set } xs = \{x\}) \rangle$   
 $\langle \text{proof} \rangle$

### 1.3.5 List intervals (*upt*)

The simplification rules are not very handy, because theorem *upt.simps* ( 2 ) (i.e.  $[\text{?i}..<\text{Suc } \text{?j}] = (\text{if } \text{?i} \leq \text{?j} \text{ then } [\text{?i}..<\text{?j}] @ [\text{?j}] \text{ else } [])$ ) 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*:  $\langle \neg i \leq j \implies [i..<Suc\ j] = [] \rangle$   
 $\langle proof \rangle$

**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**  $\langle take\ i\ [m..<n] = [m\ ..<m+i] \rangle$   
 $\langle proof \rangle$

**lemma** *append-cons-eq-upt*:  
**assumes**  $\langle A\ @\ B = [m..<n] \rangle$   
**shows**  $\langle A = [m\ ..<m+length\ A] \rangle$  **and**  $\langle B = [m + length\ A..<n] \rangle$   
 $\langle proof \rangle$

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**  $\langle A\ @\ B = [m..<n] \iff A = [m\ ..<m+length\ A] \wedge B = [m + length\ A..<n] \rangle$   
 $\langle proof \rangle$

A more restrictive version holds:

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

**lemma** *append-cons-eq-upt-length-i*:  
**assumes**  $\langle A\ @\ i\ \# B = [m..<n] \rangle$   
**shows**  $\langle A = [m\ ..<i] \rangle$   
 $\langle proof \rangle$

**lemma** *append-cons-eq-upt-length*:  
**assumes**  $\langle A\ @\ i\ \# B = [m..<n] \rangle$   
**shows**  $\langle length\ A = i - m \rangle$   
 $\langle proof \rangle$

**lemma** *append-cons-eq-upt-length-i-end*:  
**assumes**  $\langle A\ @\ i\ \# B = [m..<n] \rangle$   
**shows**  $\langle B = [Suc\ i\ ..<n] \rangle$   
 $\langle proof \rangle$

**lemma** *Max-n-upt*:  $\langle Max\ (insert\ 0\ \{Suc\ 0..<n\}) = n - Suc\ 0 \rangle$   
 $\langle proof \rangle$

**lemma** *upt-decomp-lt*:  
**assumes**  $H: \langle xs\ @\ i\ \# ys\ @\ j\ \# zs = [m\ ..<n] \rangle$   
**shows**  $\langle i < j \rangle$   
 $\langle proof \rangle$

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

⟨proof⟩

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*:

⟨ $length\ W = Suc\ 0 \longleftrightarrow (\exists L. W = [L])$ ⟩

⟨proof⟩

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

⟨proof⟩

**lemma** *finite-bounded-list*:

**fixes**  $b :: nat$

**shows** ⟨ $finite\ \{xs.\ length\ xs < s \wedge (\forall i < length\ xs.\ xs\ !\ i < b)\}$ ⟩ (**is** ⟨ $finite\ (?S\ s)$ ⟩)

⟨proof⟩

**lemma** *last-in-set-dropWhile*:

**assumes** ⟨ $\exists L \in set\ (xs\ @\ [x]). \neg P\ L$ ⟩

**shows** ⟨ $x \in set\ (dropWhile\ P\ (xs\ @\ [x]))$ ⟩

⟨proof⟩

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

⟨proof⟩

**lemma** *last-list-update-to-last*:

⟨ $last\ (xs[x := last\ xs]) = last\ xs$ ⟩

⟨proof⟩

**lemma** *take-map-nth-alt-def*: ⟨ $take\ n\ xs = map\ (!)\ xs\ [0..<min\ n\ (length\ xs)]$ ⟩

⟨proof⟩

### 1.3.6 Lexicographic Ordering

**lemma** *lexn-Suc*:

⟨ $(x\ \#\ xs,\ y\ \#\ ys) \in lexn\ r\ (Suc\ n) \longleftrightarrow$

⟨ $length\ xs = n \wedge length\ ys = n \wedge ((x,\ y) \in r \vee (x = y \wedge (xs,\ ys) \in lexn\ r\ n))$ ⟩

⟨proof⟩

**lemma** *lexn-n*:

⟨ $n > 0 \implies (x\ \#\ xs,\ y\ \#\ ys) \in lexn\ r\ n \longleftrightarrow$

⟨ $length\ xs = n - 1 \wedge length\ ys = n - 1 \wedge ((x,\ y) \in r \vee (x = y \wedge (xs,\ ys) \in lexn\ r\ (n - 1)))$ ⟩

⟨proof⟩

There is some subtle point in the previous theorem explaining *why* it is useful. The term  $1$  is converted to  $Suc\ 0$ , 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*:

⟨ $([a, b], [c, d]) \in lexn\ r\ 2 \longleftrightarrow (a, c) \in r \vee (a = c \wedge (b, d) \in r)$ ⟩

⟨proof⟩

**lemma** *lexn3-conv*:

⟨ $([a, b, c], [a', b', c']) \in lexn\ r\ 3 \longleftrightarrow$

⟨ $(a, a') \in r \vee (a = a' \wedge (b, b') \in r) \vee (a = a' \wedge b = b' \wedge (c, c') \in r)$ ⟩

⟨proof⟩

**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**  $\langle ?A \longleftrightarrow ?B \rangle$ )  
 $\langle \text{proof} \rangle$

**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**  $\langle ?A \longleftrightarrow ?B \rangle$ )  
 $\langle \text{proof} \rangle$

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

### 1.3.7 Remove

#### More lemmas about remove

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

**lemma** *remove1-Nil-iff*:  
 $\langle \text{remove1 } x \ xs = [] \longleftrightarrow xs = [] \vee xs = [x] \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *removeAll-upt*:  
 $\langle \text{removeAll } k \ [a..<b] = (\text{if } k \geq a \wedge k < b \text{ then } [a..<k] @ [\text{Suc } k..<b] \text{ else } [a..<b]) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *remove1-upt*:  
 $\langle \text{remove1 } k \ [a..<b] = (\text{if } k \geq a \wedge k < b \text{ then } [a..<k] @ [\text{Suc } k..<b] \text{ else } [a..<b]) \rangle$   
 $\langle \text{proof} \rangle$

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

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

#### Remove under condition

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

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

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

**lemma** *mset-map-mset-remove1-cond*:  
 $\langle \text{mset } (\text{map } \text{mset } (\text{remove1-cond } (\lambda L. \text{mset } L = \text{mset } a) \ C)) =$   
 $\text{remove1-mset } (\text{mset } a) \ (\text{mset } (\text{map } \text{mset } C)) \rangle$   
 $\langle \text{proof} \rangle$

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



**fun** *removeAll-cond* ::  $\langle 'a \Rightarrow \text{bool} \rangle \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list} \rangle$  **where**  
 $\langle \text{removeAll-cond } f [] = [] \rangle$  |  
 $\langle \text{removeAll-cond } f (C' \# L) = (\text{if } f C' \text{ then } \text{removeAll-cond } f L \text{ else } C' \# \text{removeAll-cond } f L) \rangle$

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

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

**lemma** *mset-map-mset-removeAll-cond*:  
 $\langle \text{mset } (\text{map } \text{mset } (\text{removeAll-cond } (\lambda b. \text{mset } b = \text{mset } a) \ C))$   
 $= \text{removeAll-mset } (\text{mset } a) (\text{mset } (\text{map } \text{mset } C)) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *count-mset-count-list*:  
 $\langle \text{count } (\text{mset } xs) \ x = \text{count-list } xs \ x \rangle$   
 $\langle \text{proof} \rangle$

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

**lemma** *removeAll-notin*:  $\langle a \notin \# A \Longrightarrow \text{removeAll-mset } a \ A = A \rangle$   
 $\langle \text{proof} \rangle$

## Filter

**lemma** *distinct-filter-eq-if*:  
 $\langle \text{distinct } C \Longrightarrow \text{length } (\text{filter } ((=) \ L) \ C) = (\text{if } L \in \text{set } C \text{ then } 1 \text{ else } 0) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *length-filter-update-true*:  
**assumes**  $\langle i < \text{length } xs \rangle$  **and**  $\langle P \ (xs \ ! \ i) \rangle$   
**shows**  $\langle \text{length } (\text{filter } P \ (xs[i := x])) = \text{length } (\text{filter } P \ xs) - (\text{if } P \ x \text{ then } 0 \text{ else } 1) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *length-filter-update-false*:  
**assumes**  $\langle i < \text{length } xs \rangle$  **and**  $\langle \neg P \ (xs \ ! \ i) \rangle$   
**shows**  $\langle \text{length } (\text{filter } P \ (xs[i := x])) = \text{length } (\text{filter } P \ xs) + (\text{if } P \ x \text{ then } 1 \text{ else } 0) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *mset-set-mset-set-minus-id-iff*:  
**assumes**  $\langle \text{finite } A \rangle$   
**shows**  $\langle \text{mset-set } A = \text{mset-set } (A - B) \longleftrightarrow (\forall b \in B. b \notin A) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *mset-set-eq-mset-set-more-conds*:  
 $\langle \text{finite } \{x. P \ x\} \Longrightarrow \text{mset-set } \{x. P \ x\} = \text{mset-set } \{x. Q \ x \wedge P \ x\} \longleftrightarrow (\forall x. P \ x \longrightarrow Q \ x) \rangle$   
**(is**  $\langle ?F \Longrightarrow ?A \longleftrightarrow ?B \rangle$ )  
 $\langle \text{proof} \rangle$

**lemma** *count-list-filter*:  $\langle \text{count-list } xs \ x = \text{length } (\text{filter } ((=) \ x) \ xs) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *sum-length-filter-compl'*:  $\langle \text{length } [x \leftarrow xs . \neg P \ x] + \text{length } (\text{filter } P \ xs) = \text{length } xs \rangle$

⟨proof⟩

### 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**  $xs :: \langle 'a :: \text{linorder list} \rangle$

**shows**  $\langle \text{sorted } xs \implies \text{sorted } ys \implies \text{mset } xs = \text{mset } ys \implies xs = ys \rangle$

⟨proof⟩

**lemma** *insort-upt*:  $\langle \text{insort } k [a..<b] =$

$\langle \text{if } k < a \text{ then } k \# [a..<b]$

$\langle \text{else if } k < b \text{ then } [a..<k] @ k \# [k ..<b]$

$\langle \text{else } [a..<b] @ [k] \rangle$

⟨proof⟩

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

⟨proof⟩

**lemma** *filter-sorted*:  $\langle \text{sorted } xs \implies \text{sorted } (\text{filter } P xs) \rangle$

⟨proof⟩

**lemma** *removeAll-insort*:

$\langle \text{sorted } xs \implies k \neq k' \implies \text{removeAll } k' (\text{insort } k xs) = \text{insort } k (\text{removeAll } k' xs) \rangle$

⟨proof⟩

### 1.3.9 Distinct Multisets

**lemma** *distinct-mset-remdups-mset-id*:  $\langle \text{distinct-mset } C \implies \text{remdups-mset } C = C \rangle$

⟨proof⟩

**lemma** *notin-add-mset-remdups-mset*:

$\langle a \notin \# A \implies \text{add-mset } a (\text{remdups-mset } A) = \text{remdups-mset } (\text{add-mset } a A) \rangle$

⟨proof⟩

**lemma** *distinct-mset-image-mset*:

$\langle \text{distinct-mset } (\text{image-mset } f (\text{mset } xs)) \longleftrightarrow \text{distinct } (\text{map } f xs) \rangle$

⟨proof⟩

**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⟩

### 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]*:  $\langle \text{distinct-mset-set } \{\} \rangle$

⟨proof⟩

**lemma** *distinct-mset-set-singleton*[iff]:  $\langle \text{distinct-mset-set } \{A\} \longleftrightarrow \text{distinct-mset } A \rangle$   
⟨proof⟩

**lemma** *distinct-mset-set-insert*[iff]:  
 $\langle \text{distinct-mset-set } (\text{insert } S \ \Sigma) \longleftrightarrow (\text{distinct-mset } S \ \wedge \ \text{distinct-mset-set } \Sigma) \rangle$   
⟨proof⟩

**lemma** *distinct-mset-set-union*[iff]:  
 $\langle \text{distinct-mset-set } (\Sigma \cup \Sigma') \longleftrightarrow (\text{distinct-mset-set } \Sigma \ \wedge \ \text{distinct-mset-set } \Sigma') \rangle$   
⟨proof⟩

**lemma** *in-distinct-mset-set-distinct-mset*:  
 $\langle a \in \Sigma \implies \text{distinct-mset-set } \Sigma \implies \text{distinct-mset } a \rangle$   
⟨proof⟩

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

**lemma** *distinct-mset-mset-set*:  $\langle \text{distinct-mset } (\text{mset-set } A) \rangle$   
⟨proof⟩

**lemma** *distinct-mset-filter-mset-set*[simp]:  $\langle \text{distinct-mset } \{\#a \in \# \text{mset-set } A. P \ a\# \} \rangle$   
⟨proof⟩

**lemma** *distinct-mset-set-distinct*:  $\langle \text{distinct-mset-set } (\text{mset } \text{' set } Cs) \longleftrightarrow (\forall c \in \text{set } Cs. \text{distinct } c) \rangle$   
⟨proof⟩

### 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⟩

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

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

**lemma** *nths-id-iff*:  
 $\langle \text{nths } xs \ A = xs \longleftrightarrow \{0..<\text{length } xs\} \subseteq A \rangle$   
⟨proof⟩

**lemma** *nts-upt-length*[simp]:  $\langle \text{nths } xs \ \{0..<\text{length } xs\} = xs \rangle$   
⟨proof⟩

**lemma** *nths-shift-lemma'*:  
 $\langle \text{map } \text{fst } [p \leftarrow \text{zip } xs \ [i..<i + n]. \ \text{snd } p + b \in A] = \text{map } \text{fst } [p \leftarrow \text{zip } xs \ [0..<n]. \ \text{snd } p + b + i \in A] \rangle$   
⟨proof⟩

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

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

*<proof>*

**lemma** *nths-upt-Suc*:

**assumes**  $\langle i < \text{length } xs \rangle$

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

*<proof>*

**lemma** *nths-upt-Suc'*:

**assumes**  $\langle i < b \rangle$  **and**  $\langle b \leq \text{length } xs \rangle$

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

*<proof>*

**lemma** *Ball-set-nths*:  $\langle (\forall L \in \text{set } (\text{nths } xs\ A). P\ L) \longleftrightarrow (\forall i \in A \cap \{0..<\text{length } xs\}. P\ (xs\ !\ i)) \rangle$

*<proof>*

### 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**  $y = (a, b, c, d, e, f, g, h)$

*<proof>*

**lemma** *prod-induct8* [*case-names fields, induct type*]:

$(\bigwedge a\ b\ c\ d\ e\ f\ g\ h. P\ (a, b, c, d, e, f, g, h)) \implies P\ x$

*<proof>*

**lemma** *prod-cases9* [*cases type*]:

**obtains** (*fields*)  $a\ b\ c\ d\ e\ f\ g\ h\ i$  **where**  $y = (a, b, c, d, e, f, g, h, i)$

*<proof>*

**lemma** *prod-induct9* [*case-names fields, induct type*]:

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

*<proof>*

**lemma** *prod-cases10* [*cases type*]:

**obtains** (*fields*)  $a\ b\ c\ d\ e\ f\ g\ h\ i\ j$  **where**  $y = (a, b, c, d, e, f, g, h, i, j)$

*<proof>*

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

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

*<proof>*

**lemma** *prod-cases11* [*cases type*]:

**obtains** (*fields*)  $a\ b\ c\ d\ e\ f\ g\ h\ i\ j\ k$  **where**  $y = (a, b, c, d, e, f, g, h, i, j, k)$

*<proof>*

**lemma** *prod-induct11* [*case-names fields, induct type*]:

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

*<proof>*

**lemma** *prod-cases12* [*cases type*]:

**obtains** (*fields*)  $a\ b\ c\ d\ e\ f\ g\ h\ i\ j\ k\ l$  **where**  $y = (a, b, c, d, e, f, g, h, i, j, k, l)$

*<proof>*

**lemma** *prod-induct12* [*case-names fields, induct type*]:  
 $(\bigwedge a b c d e f g h i j k l. P (a, b, c, d, e, f, g, h, i, j, k, l)) \implies P x$   
 $\langle \text{proof} \rangle$

**lemma** *prod-cases13* [*cases type*]:  
**obtains** (*fields*)  $a b c d e f g h i j k l m$  **where**  $y = (a, b, c, d, e, f, g, h, i, j, k, l, m)$   
 $\langle \text{proof} \rangle$

**lemma** *prod-induct13* [*case-names fields, induct type*]:  
 $(\bigwedge a b c d e f g h i j k l m. P (a, b, c, d, e, f, g, h, i, j, k, l, m)) \implies P x$   
 $\langle \text{proof} \rangle$

**lemma** *prod-cases14* [*cases type*]:  
**obtains** (*fields*)  $a b c d e f g h i j k l m n$  **where**  $y = (a, b, c, d, e, f, g, h, i, j, k, l, m, n)$   
 $\langle \text{proof} \rangle$

**lemma** *prod-induct14* [*case-names fields, induct type*]:  
 $(\bigwedge a b c d e f g h i j k l m n. P (a, b, c, d, e, f, g, h, i, j, k, l, m, n)) \implies P x$   
 $\langle \text{proof} \rangle$

**lemma** *prod-cases15* [*cases type*]:  
**obtains** (*fields*)  $a b c d e f g h i j k l m n p$  **where**  
 $y = (a, b, c, d, e, f, g, h, i, j, k, l, m, n, p)$   
 $\langle \text{proof} \rangle$

**lemma** *prod-induct15* [*case-names fields, induct type*]:  
 $(\bigwedge a b c d e f g h i j k l m n p. P (a, b, c, d, e, f, g, h, i, j, k, l, m, n, p)) \implies P x$   
 $\langle \text{proof} \rangle$

**lemma** *prod-cases16* [*cases type*]:  
**obtains** (*fields*)  $a b c d e f g h i j k l m n p q$  **where**  
 $y = (a, b, c, d, e, f, g, h, i, j, k, l, m, n, p, q)$   
 $\langle \text{proof} \rangle$

**lemma** *prod-induct16* [*case-names fields, induct type*]:  
 $(\bigwedge a b c d e f g h i j k l m n p q. P (a, b, c, d, e, f, g, h, i, j, k, l, m, n, p, q)) \implies P x$   
 $\langle \text{proof} \rangle$

**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**  
 $y = (a, b, c, d, e, f, g, h, i, j, k, l, m, n, p, q, r)$   
 $\langle \text{proof} \rangle$

**lemma** *prod-induct17* [*case-names fields, induct type*]:  
 $(\bigwedge a b c d e f g h i j k l m n p q r. P (a, b, c, d, e, f, g, h, i, j, k, l, m, n, p, q, r)) \implies P x$   
 $\langle \text{proof} \rangle$

**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**  
 $y = (a, b, c, d, e, f, g, h, i, j, k, l, m, n, p, q, r, s)$   
 $\langle \text{proof} \rangle$

**lemma** *prod-induct18* [*case-names fields, induct type*]:  
 $(\bigwedge a b c d e f g h i j k l m n p q r s. P (a, b, c, d, e, f, g, h, i, j, k, l, m, n, p, q, r, s)) \implies P x$   
 $\langle \text{proof} \rangle$

**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**  
 $y = (a, b, c, d, e, f, g, h, i, j, k, l, m, n, p, q, r, s, t)$   
(*proof*)

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

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

**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**  
 $y = (a, b, c, d, e, f, g, h, i, j, k, l, m, n, p, q, r, s, t, u)$   
(*proof*)

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

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

**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**  
 $y = (a, b, c, d, e, f, g, h, i, j, k, l, m, n, p, q, r, s, t, u, v)$   
(*proof*)

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

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

**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**  
 $y = (a, b, c, d, e, f, g, h, i, j, k, l, m, n, p, q, r, s, t, u, v, w)$   
(*proof*)

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

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

**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**  
 $y = (a, b, c, d, e, f, g, h, i, j, k, l, m, n, p, q, r, s, t, u, v, w, x)$   
(*proof*)

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

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

### 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*:  $\langle \text{list-all2 } (\lambda L. (=) (f L))\ a\ aa \longleftrightarrow aa = \text{map } f\ a \rangle$

⟨proof⟩

**lemma** *list-all2-op-eq-map-right-iff*: ⟨list-all2 (λL L'. L' = f L) a aa ⟷ aa = map f a⟩  
⟨proof⟩

**lemma** *list-all2-op-eq-map-left-iff*: ⟨list-all2 (λL' L. L' = (f L)) a aa ⟷ a = map f aa⟩  
⟨proof⟩

**lemma** *list-all2-op-eq-map-map-right-iff*:  
⟨list-all2 (list-all2 (λL. (=) (f L))) xs' x ⟷ x = map (map f) xs'⟩ **for** x  
⟨proof⟩

**lemma** *list-all2-op-eq-map-map-left-iff*:  
⟨list-all2 (list-all2 (λL' L. L' = f L)) xs' x ⟷ xs' = map (map f) x⟩  
⟨proof⟩

**lemma** *list-all2-conj*:  
⟨list-all2 (λx y. P x y ∧ Q x y) xs ys ⟷ list-all2 P xs ys ∧ list-all2 Q xs ys⟩  
⟨proof⟩

**lemma** *list-all2-replicate*:  
⟨(bi, b) ∈ R' ⟹ list-all2 (λx x'. (x, x') ∈ R') (replicate n bi) (replicate n b)⟩  
⟨proof⟩

### 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-nempty*: ⟨L ∈# D ⟹ D ≠ {#}⟩  
⟨proof⟩

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**  
⟨union-mset-list xs ys ≡ case-prod append (fold (λx (ys, zs). (remove1 x ys, x # zs)) xs (ys, []))⟩

**lemma** *union-mset-list*:  
⟨mset xs ∪# mset ys = mset (union-mset-list xs ys)⟩  
⟨proof⟩

**lemma** *union-mset-list-Nil[simp]*: ⟨union-mset-list [] bi = bi⟩  
⟨proof⟩

**lemma** *size-le-Suc-0-iff*: ⟨size M ≤ Suc 0 ⟷ ((∃ a b. M = {#a#}) ∨ M = {#})⟩  
⟨proof⟩

**lemma** *size-2-iff*: ⟨size M = 2 ⟷ (∃ a b. M = {#a, b#})⟩  
⟨proof⟩

**lemma** *subset-eq-mset-single-iff*: ⟨x2 ⊆# {#L#} ⟷ x2 = {#} ∨ x2 = {#L#}⟩  
⟨proof⟩

**lemma** *mset-eq-size-2*:

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

**lemma** *butlast-list-update*:

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

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

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

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

$\langle \text{proof} \rangle$

**lemma** *remove1-mset-empty-iff*:  $\langle \text{remove1-mset } L \text{ } N = \{\#\} \longleftrightarrow N = \{\#L\# \} \vee N = \{\#\} \rangle$

$\langle \text{proof} \rangle$

**lemma** *distinct-subseteq-iff* :

**assumes** *dist*:  $\text{distinct-mset } M$  **and** *fin*:  $\text{distinct-mset } N$

**shows**  $\text{set-mset } M \subseteq \text{set-mset } N \longleftrightarrow M \subseteq\# N$

$\langle \text{proof} \rangle$

**lemma** *distinct-set-mset-eq-iff*:

**assumes**  $\langle \text{distinct-mset } M \rangle \langle \text{distinct-mset } N \rangle$

**shows**  $\langle \text{set-mset } M = \text{set-mset } N \longleftrightarrow M = N \rangle$

$\langle \text{proof} \rangle$

**lemma** **(in -)** *distinct-mset-union2*:

$\langle \text{distinct-mset } (A + B) \implies \text{distinct-mset } B \rangle$

$\langle \text{proof} \rangle$

**lemma** *in-remove1-msetI*:  $\langle x \neq a \implies x \in\# M \implies x \in\# \text{remove1-mset } a \text{ } M \rangle$

$\langle \text{proof} \rangle$

**lemma** *count-multi-member-split*:

$\langle \text{count } M \text{ } a \geq n \implies \exists M'. M = \text{replicate-mset } n \text{ } a + M' \rangle$

$\langle \text{proof} \rangle$

**lemma** *count-image-mset-multi-member-split*:

$\langle \text{count } (\text{image-mset } f \text{ } M) \text{ } L \geq \text{Suc } 0 \implies \exists K. f \text{ } K = L \wedge K \in\# M \rangle$

$\langle \text{proof} \rangle$

**lemma** *count-image-mset-multi-member-split-2*:

**assumes** *count*:  $\langle \text{count } (\text{image-mset } f \text{ } M) \text{ } L \geq 2 \rangle$

**shows**  $\langle \exists K \text{ } K' \text{ } M'. f \text{ } K = L \wedge K \in\# M \wedge f \text{ } K' = L \wedge K' \in\# \text{remove1-mset } K \text{ } M \wedge$



$M = \{\#K, K'\#\} + M'$   
 ⟨proof⟩

**lemma** *minus-notin-trivial*:  $L \notin\# A \implies A - \text{add-mset } L B = A - B$   
 ⟨proof⟩

**lemma** *minus-notin-trivial2*:  $\langle b \notin\# A \implies A - \text{add-mset } e (\text{add-mset } b B) = A - \text{add-mset } e B \rangle$   
 ⟨proof⟩

**lemma** *diff-union-single-conv3*:  $\langle a \notin\# I \implies \text{remove1-mset } a (I + J) = I + \text{remove1-mset } a J \rangle$   
 ⟨proof⟩

**lemma** *filter-union-or-split*:  
 $\langle \{\#L \in\# C. P L \vee Q L\#\} = \{\#L \in\# C. P L\#\} + \{\#L \in\# C. \neg P L \wedge Q L\#\} \rangle$   
 ⟨proof⟩

**lemma** *subset-mset-minus-eq-add-mset-noteq*:  $\langle A \subseteq\# C \implies A - B \neq C \rangle$   
 ⟨proof⟩

**lemma** *minus-eq-id-forall-notin-mset*:  
 $\langle A - B = A \iff (\forall L \in\# B. L \notin\# A) \rangle$   
 ⟨proof⟩

**lemma** *in-multiset-minus-notin-snd[simp]*:  $\langle a \notin\# B \implies a \in\# A - B \iff a \in\# A \rangle$   
 ⟨proof⟩

**lemma** *distinct-mset-in-diff*:  
 $\langle \text{distinct-mset } C \implies a \in\# C - D \iff a \in\# C \wedge a \notin\# D \rangle$   
 ⟨proof⟩

**lemma** *diff-le-mono2-mset*:  $\langle A \subseteq\# B \implies C - B \subseteq\# C - A \rangle$   
 ⟨proof⟩

**lemma** *subsetq-remove1[simp]*:  $\langle C \subseteq\# C' \implies \text{remove1-mset } L C \subseteq\# C' \rangle$   
 ⟨proof⟩

**lemma** *filter-mset-cong2*:  
 $\langle (\bigwedge x. x \in\# M \implies f x = g x) \implies M = N \implies \text{filter-mset } f M = \text{filter-mset } g N \rangle$   
 ⟨proof⟩

**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⟩

**lemma** *notin-filter-mset*:  
 $\langle K \notin\# C \implies \text{filter-mset } P C = \text{filter-mset } (\lambda L. P L \wedge L \neq K) C \rangle$   
 ⟨proof⟩

**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**  $\langle \text{add-mset } L (\text{filter-mset } P C) = \text{filter-mset } (\lambda x. P x \vee x = L) C \rangle$   
 ⟨proof⟩

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

**lemma** *remove1-mset-union-distrib*:  
 $\langle \text{remove1-mset } a (M \cup \# N) = \text{remove1-mset } a M \cup \# \text{remove1-mset } a N \rangle$   
 $\langle \text{proof} \rangle$

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

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

**lemma** *same-mset-distinct-iff*:  
 $\langle \text{mset } M = \text{mset } M' \implies \text{distinct } M \longleftrightarrow \text{distinct } M' \rangle$   
 $\langle \text{proof} \rangle$

**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**  $\langle ?A = ?B \rangle$ )  
 $\langle \text{proof} \rangle$

**lemma** *inj-image-mset-eq-iff*:  
**assumes** *inj*:  $\langle \text{inj } f \rangle$   
**shows**  $\langle \text{image-mset } f M' = \text{image-mset } f M \longleftrightarrow M' = M \rangle$   
 $\langle \text{proof} \rangle$

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

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

**lemma** *finite-mset-set-inter*:  
 $\langle \text{finite } A \implies \text{finite } B \implies \text{mset-set } (A \cap B) = \text{mset-set } A \cap \# \text{mset-set } B \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *distinct-mset-inter-remdups-mset*:  
**assumes** *dist*:  $\langle \text{distinct-mset } A \rangle$   
**shows**  $\langle A \cap \# \text{remdups-mset } B = A \cap \# B \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *mset-butlast-update-last[simp]*:  
 $\langle w < \text{length } xs \implies \text{mset } (\text{butlast } (xs[w := \text{last } (xs)])) = \text{remove1-mset } (xs ! w) (\text{mset } xs) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *in-multiset-ge-Max*:  $\langle a \in \# N \implies a > \text{Max } (\text{insert } 0 (\text{set-mset } N)) \implies \text{False} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *distinct-mset-set-mset-remove1-mset*:

⟨distinct-mset  $M \implies \text{set-mset} (\text{remove1-mset } c \ M) = \text{set-mset } M - \{c\}$ ⟩  
 ⟨proof⟩

**lemma** *distinct-count-msetD*:

⟨distinct  $xs \implies \text{count} (\text{mset } xs) \ a = (\text{if } a \in \text{set } xs \text{ then } 1 \text{ else } 0)$ ⟩  
 ⟨proof⟩

**lemma** *filter-mset-and-implied*:

⟨ $(\bigwedge ia. ia \in\# \ xs \implies Q \ ia \implies P \ ia) \implies \{\#ia \in\# \ xs. P \ ia \wedge Q \ ia\# \} = \{\#ia \in\# \ xs. Q \ ia\# \}$ ⟩  
 ⟨proof⟩

**lemma** *filter-mset-eq-add-msetD*: ⟨filter-mset  $P \ xs = \text{add-mset } a \ A \implies a \in\# \ xs \wedge P \ a$ ⟩  
 ⟨proof⟩

**lemma** *filter-mset-eq-add-msetD'*: ⟨add-mset  $a \ A = \text{filter-mset } P \ xs \implies a \in\# \ xs \wedge P \ a$ ⟩  
 ⟨proof⟩

**lemma** *image-filter-replicate-mset*:

⟨ $\{\#Ca \in\# \ \text{replicate-mset } m \ C. P \ Ca\# \} = (\text{if } P \ C \text{ then } \text{replicate-mset } m \ C \text{ else } \{\#\})$ ⟩  
 ⟨proof⟩

**lemma** *size-Union-mset-image-mset*:

⟨size  $(\bigcup\# \ A) = (\sum i \in\# \ A. \text{size } i)$ ⟩  
 ⟨proof⟩

**lemma** *image-mset-minus-inj-on*:

⟨inj-on  $f \ (\text{set-mset } A \cup \text{set-mset } B) \implies f \ \#\ (A - B) = f \ \#\ A - f \ \#\ B$ ⟩  
 ⟨proof⟩

**lemma** *filter-mset-mono-subset*:

⟨ $A \subseteq\# \ B \implies (\bigwedge x. x \in\# \ A \implies P \ x \implies Q \ x) \implies \text{filter-mset } P \ A \subseteq\# \ \text{filter-mset } Q \ B$ ⟩  
 ⟨proof⟩

**lemma** *mset-inter-empty-set-mset*: ⟨ $M \cap\# \ xc = \{\#\} \iff \text{set-mset } M \cap \text{set-mset } xc = \{\}$ ⟩  
 ⟨proof⟩

**lemma** *sum-mset-mset-set-sum-set*:

⟨ $(\sum A \in\# \ \text{mset-set } As. f \ A) = (\sum A \in As. f \ A)$ ⟩  
 ⟨proof⟩

**lemma** *sum-mset-sum-count*:

⟨ $(\sum A \in\# \ As. f \ A) = (\sum A \in \text{set-mset } As. \text{count } As \ A * f \ A)$ ⟩  
 ⟨proof⟩

**lemma** *sum-mset-inter-restrict*:

⟨ $(\sum x \in\# \ \text{filter-mset } P \ M. f \ x) = (\sum x \in\# \ M. \text{if } P \ x \text{ then } f \ x \text{ else } 0)$ ⟩  
 ⟨proof⟩

**lemma** *mset-set-subset-iff*:

⟨ $\text{mset-set } A \subseteq\# \ I \iff \text{infinite } A \vee A \subseteq \text{set-mset } I$ ⟩  
 ⟨proof⟩

**lemma** *sumset-diff-constant-left*:

**assumes** ⟨ $\bigwedge x. x \in\# \ A \implies f \ x \leq n$ ⟩

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

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

**lemma** *distinct-mset-iff*:  
 $\langle \neg \text{distinct-mset } C \longleftrightarrow (\exists a C'. C = \text{add-mset } a (\text{add-mset } a C') \rangle$   
 $\langle \text{proof} \rangle$

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

## 1.4 Finite maps and multisets

### Finite sets and multisets

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

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

**lemma** *fset-mset-mset-fset*:  $\langle \text{fset-mset } (\text{mset-fset } N) = N \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *mset-fset-fset-mset[simp]*:  
 $\langle \text{mset-fset } (\text{fset-mset } N) = \text{remdups-mset } N \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *in-mset-fset-fmember[simp]*:  $\langle x \in \# \text{mset-fset } N \longleftrightarrow x \in | N \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *in-fset-mset-mset[simp]*:  $\langle x \in | \text{fset-mset } N \longleftrightarrow x \in \# N \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *distinct-mset-subset-iff-remdups*:  
 $\langle \text{distinct-mset } a \implies a \subseteq \# b \longleftrightarrow a \subseteq \# \text{remdups-mset } b \rangle$   
 $\langle \text{proof} \rangle$

### 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-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-m*).

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

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

**lemma** *dom-m-fmdrop[simp]*:  $\langle \text{dom-m } (\text{fmdrop } C \ N) = \text{remove1-mset } C \ (\text{dom-m } N) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *dom-m-fmdrop-All*:  $\langle \text{dom-m } (\text{fmdrop } C \ N) = \text{removeAll-mset } C \ (\text{dom-m } N) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *dom-m-fmupd[simp]*:  $\langle \text{dom-m } (\text{fmupd } k \ C \ N) = \text{add-mset } k \ (\text{remove1-mset } k \ (\text{dom-m } N)) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *distinct-mset-dom*:  $\langle \text{distinct-mset } (\text{dom-m } N) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *in-dom-m-lookup-iff*:  $\langle C \in\# \ \text{dom-m } N' \longleftrightarrow \text{fmlookup } N' \ C \neq \text{None} \rangle$   
 $\langle \text{proof} \rangle$

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

**lemma** *fmupd-same[simp]*:  
 $\langle x1 \in\# \ \text{dom-m } x1aa \implies \text{fmupd } x1 \ (\text{the } (\text{fmlookup } x1aa \ x1)) \ x1aa = x1aa \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *ran-m-fmempty[simp]*:  $\langle \text{ran-m } \text{fmempty} = \{\#\} \rangle$  **and**  
*dom-m-fmempty[simp]*:  $\langle \text{dom-m } \text{fmempty} = \{\#\} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *fmrestrict-set-fmupd*:  
 $\langle a \in xs \implies \text{fmrestrict-set } xs \ (\text{fmupd } a \ C \ N) = \text{fmupd } a \ C \ (\text{fmrestrict-set } xs \ N) \rangle$   
 $\langle a \notin xs \implies \text{fmrestrict-set } xs \ (\text{fmupd } a \ C \ N) = \text{fmrestrict-set } xs \ N \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *fset-fmdom-fmrestrict-set*:  
 $\langle \text{fset } (\text{fmdom } (\text{fmrestrict-set } xs \ N)) = \text{fset } (\text{fmdom } N) \cap xs \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *dom-m-fmrestrict-set*:  $\langle \text{dom-m } (\text{fmrestrict-set } (\text{set } xs) \ N) = \text{mset } xs \cap\# \ \text{dom-m } N \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *dom-m-fmrestrict-set'*:  $\langle \text{dom-m } (\text{fmrestrict-set } xs \ N) = \text{mset-set } (xs \cap \text{set-mset } (\text{dom-m } N)) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *indom-mI*:  $\langle \text{fmlookup } m \ x = \text{Some } y \implies x \in\# \ \text{dom-m } m \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *fmupd-fmdrop-id*:  
**assumes**  $\langle k \in\# \ \text{fmdom } N' \rangle$   
**shows**  $\langle \text{fmupd } k \ (\text{the } (\text{fmlookup } N' \ k)) \ (\text{fmdrop } k \ N') = N' \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *fm-member-split*:  $\langle k \in\# \ \text{fmdom } N' \implies \exists N'' \ v. N' = \text{fmupd } k \ v \ N'' \wedge \text{the } (\text{fmlookup } N' \ k) = v \wedge k \notin\# \ \text{fmdom } N'' \rangle$   
 $\langle \text{proof} \rangle$

**lemma**  $\langle \text{fmdrop } k \ (\text{fmupd } k \ va \ N'') = \text{fmdrop } k \ N'' \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *fmap-ext-fndom*:

$\langle (fndom\ N = fndom\ N') \implies (\bigwedge x. x \in | fndom\ N \implies fmllookup\ N\ x = fmllookup\ N'\ x) \implies N = N' \rangle$   
 $\langle proof \rangle$

**lemma** *fmrestrict-set-insert-in*:

$\langle xa \in fset\ (fndom\ N) \implies fmrestrict\text{-}set\ (insert\ xa\ l1)\ N = fmupd\ xa\ (the\ (fmllookup\ N\ xa))\ (fmrestrict\text{-}set\ l1\ N) \rangle$   
 $\langle proof \rangle$

**lemma** *fmrestrict-set-insert-notin*:

$\langle xa \notin fset\ (fndom\ N) \implies fmrestrict\text{-}set\ (insert\ xa\ l1)\ N = fmrestrict\text{-}set\ l1\ N \rangle$   
 $\langle proof \rangle$

**lemma** *fmrestrict-set-insert-in-dom-m[simp]*:

$\langle xa \in \# dom\text{-}m\ N \implies fmrestrict\text{-}set\ (insert\ xa\ l1)\ N = fmupd\ xa\ (the\ (fmllookup\ N\ xa))\ (fmrestrict\text{-}set\ l1\ N) \rangle$   
 $\langle proof \rangle$

**lemma** *fmrestrict-set-insert-notin-dom-m[simp]*:

$\langle xa \notin \# dom\text{-}m\ N \implies fmrestrict\text{-}set\ (insert\ xa\ l1)\ N = fmrestrict\text{-}set\ l1\ N \rangle$   
 $\langle proof \rangle$

**lemma** *fmllookup-restrict-set-id*:  $\langle fset\ (fndom\ N) \subseteq A \implies fmrestrict\text{-}set\ A\ N = N \rangle$

$\langle proof \rangle$

**lemma** *fmllookup-restrict-set-id'*:  $\langle set\text{-}mset\ (dom\text{-}m\ N) \subseteq A \implies fmrestrict\text{-}set\ A\ N = N \rangle$

$\langle proof \rangle$

**lemma** *ran-m-mapsto-upd*:

**assumes**

$NC: \langle C \in \# dom\text{-}m\ N \rangle$

**shows**  $\langle ran\text{-}m\ (fmupd\ C\ C'\ N) =$

$add\text{-}mset\ C'\ (remove1\text{-}mset\ (the\ (fmllookup\ N\ C))\ (ran\text{-}m\ N)) \rangle$

$\langle proof \rangle$

**lemma** *ran-m-mapsto-upd-notin*:

**assumes**

$NC: \langle C \notin \# dom\text{-}m\ N \rangle$

**shows**  $\langle ran\text{-}m\ (fmupd\ C\ C'\ N) = add\text{-}mset\ C'\ (ran\text{-}m\ N) \rangle$

$\langle proof \rangle$

**lemma** *ran-m-fmdrop*:

$\langle C \in \# dom\text{-}m\ N \implies ran\text{-}m\ (fmdrop\ C\ N) = remove1\text{-}mset\ (the\ (fmllookup\ N\ C))\ (ran\text{-}m\ N) \rangle$

$\langle proof \rangle$

**lemma** *ran-m-fmdrop-notin*:

$\langle C \notin \# dom\text{-}m\ N \implies ran\text{-}m\ (fmdrop\ C\ N) = ran\text{-}m\ N \rangle$

$\langle proof \rangle$

**lemma** *ran-m-fmdrop-If*:

$\langle ran\text{-}m\ (fmdrop\ C\ N) = (if\ C \in \# dom\text{-}m\ N\ then\ remove1\text{-}mset\ (the\ (fmllookup\ N\ C))\ (ran\text{-}m\ N)\ else\ ran\text{-}m\ N) \rangle$

⟨proof⟩

## 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*

⟨*Max-dom*  $N = \text{Max} (\text{set-mset} (\text{add-mset } 0 (\text{dom-m } N)))$ ⟩

**definition** *packed where*

⟨*packed*  $N \longleftrightarrow \text{dom-m } N = \text{mset} [1..<\text{Suc} (\text{Max-dom } N)]$ ⟩

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

**lemma** *Max-dom-empty*: ⟨*dom-m*  $b = \{\#\} \implies \text{Max-dom } b = 0$ ⟩

⟨proof⟩

**lemma** *Max-dom-fmempty*: ⟨*Max-dom fmempty* = 0⟩

⟨proof⟩

**lemma** *packed-empty[simp]*: ⟨*packed fmempty*⟩

⟨proof⟩

**lemma** *packed-Max-dom-size*:

**assumes**  $p$ : ⟨*packed*  $N$ ⟩

**shows** ⟨*Max-dom*  $N = \text{size} (\text{dom-m } N)$ ⟩

⟨proof⟩

**lemma** *Max-dom-le*:

⟨ $L \in \# \text{ dom-m } N \implies L \leq \text{Max-dom } N$ ⟩

⟨proof⟩

**lemma** *remove1-mset-ge-Max-some*: ⟨ $a > \text{Max-dom } b \implies \text{remove1-mset } a (\text{dom-m } b) = \text{dom-m } b$ ⟩

⟨proof⟩

**lemma** *Max-dom-fmupd-irrel*:

⟨ $(a :: 'a :: \{\text{zero}, \text{linorder}\}) > \text{Max-dom } M \implies \text{Max-dom} (\text{fmupd } a C M) = \text{max } a (\text{Max-dom } M)$ ⟩

⟨proof⟩

**lemma** *Max-dom-alt-def*: ⟨*Max-dom*  $b = \text{Max} (\text{insert } 0 (\text{set-mset} (\text{dom-m } b)))$ ⟩

⟨proof⟩

**lemma** *Max-insert-Suc-Max-dim-dom[simp]*:

⟨ $\text{Max} (\text{insert} (\text{Suc} (\text{Max-dom } b)) (\text{set-mset} (\text{dom-m } b))) = \text{Suc} (\text{Max-dom } b)$ ⟩

⟨proof⟩

**lemma** *size-dom-m-Max-dom*:

⟨*size* (*dom-m*  $N$ )  $\leq \text{Suc} (\text{Max-dom } N)$ ⟩

⟨proof⟩

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

⟨proof⟩

**lemma** *Max-atLeastLessThan*:  $\langle \text{Max } \{(a::\text{nat}) ..< b\} = (\text{if } b \leq a \text{ then Max } \{\} \text{ else } b - 1) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *Max-insert-Max-dom-into-packed*:  
 $\langle \text{Max } (\text{insert } (\text{Max-dom } bc) \{\text{Suc } 0..<\text{Max-dom } bc\}) = \text{Max-dom } bc \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *packed0-fmud-Suc-Max-dom*:  $\langle \text{packed } b \implies \text{packed } (\text{fmupd } (\text{Suc } (\text{Max-dom } b)) \ C \ b) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *ge-Max-dom-notin-dom-m*:  $\langle a > \text{Max-dom } ao \implies a \notin \# \text{ dom-m } ao \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *packed-in-dom-mI*:  $\langle \text{packed } bc \implies j \leq \text{Max-dom } bc \implies 0 < j \implies j \in \# \text{ dom-m } bc \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *mset-fset-empty-iff*:  $\langle \text{mset-fset } a = \{\#\} \longleftrightarrow a = \text{fempty} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *dom-m-empty-iff[iff]*:  
 $\langle \text{dom-m } NU = \{\#\} \longleftrightarrow NU = \text{fmempty} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *nat-power-div-base*:  
**fixes**  $k :: \text{nat}$   
**assumes**  $0 < m \ 0 < k$   
**shows**  $k \wedge^m \text{div } k = (k::\text{nat}) \wedge (m - \text{Suc } 0)$   
 $\langle \text{proof} \rangle$

**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 = HAVE-IF | ASSUME-SHOW | ASSUMES-SHOWS | CONTEXT
  val explore: explore -> Toplevel.state -> Proof.state
end

```

```

structure Explorer: EXPLORER =
struct
datatype explore = 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
  end

```

```

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-tyt 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)])

```

```

fun explore aim st =
  let
    val thy = Toplevel.theory-of st
    val quote-type = Explorer-Lib.default-raw-params thy |> snd
    val enclosure =
      (case quote-type of

```

```

    Explorer-Lib.GUILLEMOTS => cartouche
  | Explorer-Lib.QUOTES => 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*  
*<proof>*

**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**

*<proof>*

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*  
*<proof>*

And switch back

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

**lemma**

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

**end**