Formalisation of Ground Resolution and CDCL in Isabelle/HOL

Mathias Fleury and Jasmin Blanchette

December 6, 2019

Contents

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*:
‹$r^{++}\ a\ b \Longrightarrow r \leq s \Longrightarrow s^{++}\ a\ b$›
⟨*proof*⟩

lemma *tranclp-mono*:
assumes *mono*: ‹$r \leq s$›
shows ‹$r^{++} \leq s^{++}$›
⟨*proof*⟩

lemma *tranclp-idemp-rel*:
‹$R^{++++}\ a\ b \longleftrightarrow R^{++}\ a\ b$›
⟨*proof*⟩

Equivalent of the theorem *rtranclp-idemp*

lemma *trancl-idemp*: ‹$(r^+)^+ = r^+$›
⟨*proof*⟩

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*: ‹$rtranclp\ r\ a\ b \longleftrightarrow (a = b \vee tranclp\ r\ a\ b)$›
⟨*proof*⟩

lemma *tranclp-unfold-end*: ‹$tranclp\ r\ a\ b \longleftrightarrow (\exists\, a'.\ rtranclp\ r\ a\ a' \wedge r\ a'\ b)$›
⟨*proof*⟩

Near duplicate of theorem *tranclpD*:

lemma *tranclp-unfold-begin*: ‹$tranclp\ r\ a\ b \longleftrightarrow (\exists\, a'.\ r\ a\ a' \wedge rtranclp\ r\ a'\ b)$›
⟨*proof*⟩

lemma *trancl-set-tranclp*: ‹$(a,\ b) \in \{(b,a).\ P\ a\ b\}^+ \longleftrightarrow P^{++}\ b\ a$›
⟨*proof*⟩

lemma *tranclp-rtranclp-rtranclp-rel*: ‹$R^{++**}\ a\ b \longleftrightarrow R^{**}\ a\ b$›
⟨*proof*⟩

lemma *tranclp-rtranclp-rtranclp*[*simp*]: ‹$R^{++**} = R^{**}$›
⟨*proof*⟩

lemma *rtranclp-exists-last-with-prop*:
assumes ‹$R\ x\ z$› **and** ‹$R^{**}\ z\ z'$› **and** ‹$P\ x\ z$›
shows ‹$\exists\, y\ y'.\ R^{**}\ x\ y \wedge R\ y\ y' \wedge P\ y\ y' \wedge (\lambda a\ b.\ R\ a\ b \wedge \neg P\ a\ b)^{**}\ y'\ z'$›
⟨*proof*⟩

lemma *rtranclp-and-rtranclp-left*: ‹$(\lambda\ a\ b.\ P\ a\ b \wedge Q\ a\ b)^{**}\ S\ T \Longrightarrow P^{**}\ S\ T$›
⟨*proof*⟩

1.1.2 Full Transitions

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

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

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

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

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

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

Some Properties **lemma** *rtranclp-full1I*:
‹$R^{**}\ a\ b \Longrightarrow full1\ R\ b\ c \Longrightarrow full1\ R\ a\ c$›
⟨*proof*⟩

lemma *tranclp-full1I*:
‹$R^{++}\ a\ b \Longrightarrow full1\ R\ b\ c \Longrightarrow full1\ R\ a\ c$›
⟨*proof*⟩

lemma *rtranclp-fullI*:
‹$R^{**}\ a\ b \Longrightarrow full\ R\ b\ c \Longrightarrow full\ R\ a\ c$›
⟨*proof*⟩

lemma *tranclp-full-full1I*:
‹R^{++} a $b \Longrightarrow$ *full* R b $c \Longrightarrow$ *full1* R a c›
⟨*proof*⟩

lemma *full-fullI*:
‹R a $b \Longrightarrow$ *full* R b $c \Longrightarrow$ *full1* R a c›
⟨*proof*⟩

lemma *full-unfold*:
‹*full* r S $S' \longleftrightarrow ((S = S' \wedge$ *no-step* r $S') \vee$ *full1* r S $S')$›
⟨*proof*⟩

lemma *full1-is-full*[*intro*]: ‹*full1* R S $T \Longrightarrow$ *full* R S T›
⟨*proof*⟩

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

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

lemma *full1-tranclp-relation-full*:
‹*full1* R^{++} a $b \longleftrightarrow$ *full1* R a b›
⟨*proof*⟩

lemma *full-tranclp-relation-full*:
‹*full* R^{++} a $b \longleftrightarrow$ *full* R a b›
⟨*proof*⟩

lemma *tranclp-full1-full1*:
‹(*full1* $R)^{++}$ a $b \longleftrightarrow$ *full1* R a b›
⟨*proof*⟩

lemma *rtranclp-full1-eq-or-full1*:
‹(*full1* $R)^{**}$ a $b \longleftrightarrow (a = b \vee$ *full1* R a $b)$›
⟨*proof*⟩

lemma *no-step-full-iff-eq*:
‹*no-step* R $S \Longrightarrow$ *full* R S $T \longleftrightarrow S = T$›
⟨*proof*⟩

1.1.3 Well-Foundedness and Full Transitions

lemma *wf-exists-normal-form*:
assumes *wf*: ‹*wf* $\{(x, y).\ R\ y\ x\}$›
shows ‹$\exists\, b.\ R^{**}$ a $b \wedge$ *no-step* R b›
⟨*proof*⟩

lemma *wf-exists-normal-form-full*:
assumes *wf*: ‹*wf* $\{(x, y).\ R\ y\ x\}$›
shows ‹$\exists\, b.$ *full* R a b›
⟨*proof*⟩

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-chainI*

lemma *wf-if-measure-in-wf*:
$\langle wf\ R \Longrightarrow (\bigwedge a\ b.\ (a,\ b) \in S \Longrightarrow (\nu\ a,\ \nu\ b){\in}R) \Longrightarrow wf\ S\rangle$
⟨*proof*⟩

lemma *wfP-if-measure*: **fixes** f :: $\langle 'a \Rightarrow nat\rangle$
shows $\langle(\bigwedge x\ y.\ P\ x \Longrightarrow g\ x\ y \Longrightarrow f\ y < f\ x) \Longrightarrow wf\ \{(y,x).\ P\ x \wedge g\ x\ y\}\rangle$
⟨*proof*⟩

lemma *wf-if-measure-f*:
assumes $\langle wf\ r\rangle$
shows $\langle wf\ \{(b,\ a).\ (f\ b,\ f\ a) \in r\}\rangle$
⟨*proof*⟩

lemma *wf-wf-if-measure′*:
assumes $\langle wf\ r\rangle$ **and** H: $\langle\bigwedge x\ y.\ P\ x \Longrightarrow g\ x\ y \Longrightarrow (f\ y,\ f\ x) \in r\rangle$
shows $\langle wf\ \{(y,x).\ P\ x \wedge g\ x\ y\}\rangle$
⟨*proof*⟩

lemma *wf-lex-less*: $\langle wf\ (lex\ less\text{-}than)\rangle$
⟨*proof*⟩

lemma *wfP-if-measure2*: **fixes** f :: $\langle 'a \Rightarrow nat\rangle$
shows $\langle(\bigwedge x\ y.\ P\ x\ y \Longrightarrow g\ x\ y \Longrightarrow f\ x < f\ y) \Longrightarrow wf\ \{(x,y).\ P\ x\ y \wedge g\ x\ y\}\rangle$
⟨*proof*⟩

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 $\langle wf\ \{(T,\ S).\ (P\ S \wedge P\ T) \wedge (T,\ S) \in lexord\ R\}\rangle$
⟨*proof*⟩

lemma *wf-fst-wf-pair*:
assumes $\langle wf\ \{(M',\ M).\ R\ M'\ M\}\ \rangle$
shows $\langle wf\ \{((M',\ N'),\ (M,\ N)).\ R\ M'\ M\}\rangle$
⟨*proof*⟩

lemma *wf-snd-wf-pair*:
assumes $\langle wf\ \{(M',\ M).\ R\ M'\ M\}\ \rangle$
shows $\langle wf\ \{((M',\ N'),\ (M,\ N)).\ R\ N'\ N\}\rangle$
⟨*proof*⟩

lemma *wf-if-measure-f-notation2*:
assumes $\langle wf\ r\rangle$
shows $\langle wf\ \{(b,\ h\ a)|b\ a.\ (f\ b,\ f\ (h\ a)) \in r\}\rangle$
⟨*proof*⟩

lemma *wf-wf-if-measure′-notation2*:
  assumes ‹*wf r*› **and** *H*: ‹$\bigwedge x\ y.\ P\ x \Longrightarrow g\ x\ y \Longrightarrow (f\ y,\ f\ (h\ x)) \in r$›
  shows ‹*wf* $\{(y,h\ x)|\ y\ x.\ P\ x \wedge g\ x\ y\}$›
⟨*proof*⟩

lemma *power-ex-decomp*:
  assumes ‹$(R$^^$n)\ S\ T$›
  shows
    ‹$\exists f.\ f\ 0 = S \wedge f\ n = T \wedge (\forall i.\ i < n \longrightarrow R\ (f\ i)\ (f\ (Suc\ i)))$›
  ⟨*proof*⟩

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: ‹$\bigwedge i.\ i < n \Longrightarrow ((f\ (Suc\ i)),\ (f\ i)) \in$ *lexn less-than m*› **and**
    upper: ‹$\bigwedge i\ j.\ i \leq n \Longrightarrow j < m \Longrightarrow (f\ i)\ !\ j \in \{0..<k\}$› **and**
    ‹*finite A*› **and**
    k: ‹$k > 1$›
  shows ‹$n < k$ ^ *Suc m*›
⟨*proof*⟩

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*]: ‹∃ *x*. -› ⇒ ‹*rule exE*[*OF J*]›
  | *J*[*thin*]: ‹- ∧ -› ⇒ ‹*rule conjE*[*OF J*]›
)

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

lemma *nat-less-induct-case*[*case-names 0 Suc*]:
  assumes
    ‹$P\ 0$› **and**
    ‹$\bigwedge n.\ (\forall m < Suc\ n.\ P\ m) \Longrightarrow P\ (Suc\ n)$›
  shows ‹$P\ n$›

⟨*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*]:
‹$0 < (\mathit{if}\ P\ \mathit{then}\ a\ \mathit{else}\ (0{::}nat)) \longleftrightarrow P \wedge 0 < a$›
⟨*proof*⟩

lemma *bex-lessI*: $P\ j \Longrightarrow j < n \Longrightarrow \exists j{<}n.\ P\ j$
⟨*proof*⟩

lemma *bex-gtI*: $P\ j \Longrightarrow j > n \Longrightarrow \exists j{>}n.\ P\ j$
⟨*proof*⟩

lemma *bex-geI*: $P\ j \Longrightarrow j \geq n \Longrightarrow \exists j{\geq}n.\ P\ j$
⟨*proof*⟩

lemma *bex-leI*: $P\ j \Longrightarrow j \leq n \Longrightarrow \exists j{\leq}n.\ P\ j$
⟨*proof*⟩

Bounded function have not yet been defined in Isabelle.

definition *bounded* :: $('a \Rightarrow 'b{::}ord) \Rightarrow bool$ **where**
‹$\mathit{bounded}\ f \longleftrightarrow (\exists b.\ \forall n.\ f\ n \leq b)$›

abbreviation *unbounded* :: ‹$('a \Rightarrow 'b{::}ord) \Rightarrow bool$› **where**
‹$\mathit{unbounded}\ f \equiv \neg\ \mathit{bounded}\ f$›

lemma *not-bounded-nat-exists-larger*:
fixes f :: ‹$nat \Rightarrow nat$›
assumes *unbound*: ‹*unbounded f*›
shows ‹$\exists n.\ f\ n > m \wedge n > n_0$›
⟨*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 :: *nat* **and** f :: ‹$nat \Rightarrow nat$›
assumes ‹$k > 0$›
shows ‹$\mathit{bounded}\ f \longleftrightarrow \mathit{bounded}\ (\lambda i.\ k * f\ i)$›
⟨*proof*⟩

lemma *bounded-const-add*:
fixes k :: *nat* **and** f :: ‹$nat \Rightarrow nat$›
assumes ‹$k > 0$›
shows ‹$\mathit{bounded}\ f \longleftrightarrow \mathit{bounded}\ (\lambda i.\ k + f\ i)$›
⟨*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 :: ‹$'a{::}finite \Rightarrow 'b :: \{linorder\}$›
shows ‹*bounded f*›
⟨*proof*⟩

1.3 More Lists

1.3.1 set, nth, tl

lemma *ex-geI*: ‹$P\ n \Longrightarrow n \geq m \Longrightarrow \exists\, n{\geq}m.\ P\ n$›
⟨*proof*⟩

lemma *Ball-atLeastLessThan-iff*: ‹$(\forall\, L{\in}\{a..{<}b\}.\ P\ L) \longleftrightarrow (\forall\, L.\ L \geq a \wedge L < b \longrightarrow P\ L)$ ›
⟨*proof*⟩

lemma *nth-in-set-tl*: ‹$i > 0 \Longrightarrow i < length\ xs \Longrightarrow xs\ !\ i \in set\ (tl\ xs)$›
⟨*proof*⟩

lemma *tl-drop-def*: ‹$tl\ N = drop\ 1\ N$›
⟨*proof*⟩

lemma *in-set-remove1D*:
‹$a \in set\ (remove1\ x\ xs) \Longrightarrow a \in set\ xs$›
⟨*proof*⟩

lemma *take-length-takeWhile-eq-takeWhile*:
‹$take\ (length\ (takeWhile\ P\ xs))\ xs = takeWhile\ P\ xs$›
⟨*proof*⟩

lemma *fold-cons-replicate*: ‹$fold\ (\lambda\text{-}\ xs.\ a\ \#\ xs)\ [0..{<}n]\ xs = replicate\ n\ a\ @\ xs$›
⟨*proof*⟩

lemma *Collect-minus-single-Collect*: ‹$\{x.\ P\ x\} - \{a\} = \{x\ .\ P\ x \wedge x \neq a\}$›
⟨*proof*⟩

lemma *in-set-image-subsetD*: ‹ f ‘ $A \subseteq B \Longrightarrow x \in A \Longrightarrow f\ x \in B$›
⟨*proof*⟩

lemma *mset-tl*:
‹$mset\ (tl\ xs) = remove1\text{-}mset\ (hd\ xs)\ (mset\ xs)$›
⟨*proof*⟩

lemma *hd-list-update-If*:
‹$outl' \neq [] \Longrightarrow hd\ (outl'[i := w]) = (if\ i = 0\ then\ w\ else\ hd\ outl')$›
⟨*proof*⟩

lemma *list-update-id'*:
‹$x = xs\ !\ i \Longrightarrow xs[i := x] = xs$›
⟨*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*:
‹$\{((a, b), (a', b')).\ P\ a\ b\ a'\ b'\} = \{(e, f).\ P\ (fst\ e)\ (snd\ e)\ (fst\ f)\ (snd\ f)\}$›
⟨*proof*⟩

lemma *butlast-Nil-iff*: ‹$butlast\ xs = [] \longleftrightarrow length\ xs = 1 \vee length\ xs = 0$›
⟨*proof*⟩

lemma *Set-remove-diff-insert*: ‹$a \in B - A \Longrightarrow B - Set.remove\ a\ A = insert\ a\ (B - A)$›
⟨*proof*⟩

lemma *Set-insert-diff-remove*: ‹$B - insert\ a\ A = Set.remove\ a\ (B - A)$›
⟨*proof*⟩

lemma *Set-remove-insert*: ‹$a \notin A' \Longrightarrow Set.remove\ a\ (insert\ a\ A') = A'$›
⟨*proof*⟩

lemma *diff-eq-insertD*:
‹$B - A = insert\ a\ A' \Longrightarrow a \in B$›
⟨*proof*⟩

lemma *in-set-tlD*: ‹$x \in set\ (tl\ xs) \Longrightarrow x \in set\ xs$›
⟨*proof*⟩

This lemmma 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*: ‹$xs = it\ @\ x\ \#\ \sigma \Longrightarrow x \in set\ xs$›
⟨*proof*⟩

lemma *Collect-eq-comp'*: ‹ $\{(x, y).\ P\ x\ y\}\ O\ \{(c, a).\ c = f\ a\} = \{(x, a).\ P\ x\ (f\ a)\}$›
⟨*proof*⟩

lemma (**in** −) *filter-disj-eq*:
‹$\{x \in A.\ P\ x \vee Q\ x\} = \{x \in A.\ P\ x\} \cup \{x \in A.\ Q\ x\}$›
⟨*proof*⟩

lemma *zip-cong*:
‹$(\bigwedge i.\ i < min\ (length\ xs)\ (length\ ys) \Longrightarrow (xs\ !\ i,\ ys\ !\ i) = (xs'\ !\ i,\ ys'\ !\ i)) \Longrightarrow$
$length\ xs = length\ xs' \Longrightarrow length\ ys = length\ ys' \Longrightarrow zip\ xs\ ys = zip\ xs'\ ys'$›
⟨*proof*⟩

lemma *zip-cong2*:
‹$(\bigwedge i.\ i < min\ (length\ xs)\ (length\ ys) \Longrightarrow (xs\ !\ i,\ ys\ !\ i) = (xs'\ !\ i,\ ys'\ !\ i)) \Longrightarrow$
$length\ xs = length\ xs' \Longrightarrow length\ ys \leq length\ ys' \Longrightarrow length\ ys \geq length\ xs \Longrightarrow$
$zip\ xs\ ys = zip\ xs'\ ys'$›
⟨*proof*⟩

1.3.2 List Updates

lemma *tl-update-swap*:
‹$i \geq 1 \Longrightarrow tl\ (N[i := C]) = (tl\ N)[i-1 := C]$›
⟨*proof*⟩

lemma *tl-update-0*[*simp*]: ‹$tl\ (N[0 := x]) = tl\ N$›
⟨*proof*⟩

declare *nth-list-update*[*simp*]

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

lemma *nth-list-update-le'*[*simp*]:
$j < length\ xs \Longrightarrow (xs[i{:=}x])!j = (if\ i = j\ then\ x\ else\ xs!j)$
⟨*proof*⟩

1.3.3 Take and drop

lemma *take-2-if*:
‹$take\ 2\ C = (if\ C = []\ then\ []\ else\ if\ length\ C = 1\ then\ [hd\ C]\ else\ [C!0,\ C!1])$›
⟨*proof*⟩

lemma *in-set-take-conv-nth*:
‹$x \in set\ (take\ n\ xs) \longleftrightarrow (\exists m<min\ n\ (length\ xs).\ xs\ !\ m = x)$›
⟨*proof*⟩

lemma *in-set-dropI*:
‹$m < length\ xs \Longrightarrow m \geq n \Longrightarrow xs\ !\ m \in set\ (drop\ n\ xs)$›
⟨*proof*⟩

lemma *in-set-drop-conv-nth*:
‹$x \in set\ (drop\ n\ xs) \longleftrightarrow (\exists m \geq n.\ m < length\ xs \wedge xs\ !\ m = x)$›
⟨*proof*⟩

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

lemma *atd-lem*: ‹$take\ n\ xs = t \Longrightarrow drop\ n\ xs = d \Longrightarrow xs = t\ @\ d$›
⟨*proof*⟩

lemma *drop-take-drop-drop*:
‹$j \geq i \Longrightarrow drop\ i\ xs = take\ (j - i)\ (drop\ i\ xs)\ @\ drop\ j\ xs$›
⟨*proof*⟩

lemma *in-set-conv-iff*:
‹$x \in set\ (take\ n\ xs) \longleftrightarrow (\exists i < n.\ i < length\ xs \wedge xs\ !\ i = x)$›
⟨*proof*⟩

lemma *distinct-in-set-take-iff*:
‹$distinct\ D \Longrightarrow b < length\ D \Longrightarrow D\ !\ b \in set\ (take\ a\ D) \longleftrightarrow b < a$›
⟨*proof*⟩

lemma *in-set-distinct-take-drop-iff*:
assumes
‹$distinct\ D$› **and**
‹$b < length\ D$›
shows ‹$D\ !\ b \in set\ (take\ (a - init)\ (drop\ init\ D)) \longleftrightarrow (init \leq b \wedge b < a)$›
⟨*proof*⟩

1.3.4 Replicate

lemma *list-eq-replicate-iff-nempty*:
‹$n > 0 \Longrightarrow xs = replicate\ n\ x \longleftrightarrow n = length\ xs \wedge set\ xs = \{x\}$›
⟨*proof*⟩

lemma *list-eq-replicate-iff*:
‹$xs = replicate\ n\ x \longleftrightarrow (n = 0 \wedge xs = []) \vee (n = length\ xs \wedge set\ xs = \{x\})$›
⟨*proof*⟩

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*: ‹$\neg i \leq j \Longrightarrow [i..<Suc\ j] = []$›
  ⟨*proof*⟩

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 $?i + ?m \leq ?n \Longrightarrow take\ ?m\ [?i..<?n] = [?i..<?i + ?m]$, but seems more general.

lemma *take-upt-bound-minus*[*simp*]:
  assumes ‹$i \leq n - m$›
  shows ‹$take\ i\ [m..<n] = [m\ ..<m+i]$›
  ⟨*proof*⟩

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

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 ‹$A\ @\ B = [m..<\ n] \longleftrightarrow A = [m\ ..<m+length\ A] \wedge B = [m + length\ A..<n]$›
⟨*proof*⟩

A more restrictive version holds:

lemma ‹$B \neq [] \Longrightarrow A\ @\ B = [m..<\ n] \longleftrightarrow A = [m\ ..<m+length\ A] \wedge B = [m + length\ A..<n]$›
  (**is** ‹$?P \Longrightarrow ?A = ?B$›)
⟨*proof*⟩

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

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

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

lemma *Max-n-upt*: ‹$Max\ (insert\ 0\ \{Suc\ 0..<n\}) = n - Suc\ 0$›
⟨*proof*⟩

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

lemma *nths-upt-upto-Suc*: ‹$aa < length\ xs \Longrightarrow nths\ xs\ \{0..<Suc\ aa\} = nths\ xs\ \{0..<aa\}\ @\ [xs\ !\ aa]$›

$\langle proof \rangle$

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*])›
$\langle proof \rangle$

lemma *length-list-2*: ‹*length S = 2* $\longleftrightarrow$ ($\exists$ *a b*. *S* = [*a*, *b*])›
$\langle proof \rangle$

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*)›)
$\langle proof \rangle$

lemma *last-in-set-dropWhile*:
assumes ‹$\exists$ *L* $\in$ *set* (*xs* @ [*x*]). $\neg$*P L*›
shows ‹*x* $\in$ *set* (*dropWhile P* (*xs* @ [*x*]))›
$\langle proof \rangle$

lemma *mset-drop-upto*: ‹*mset* (*drop a N*) = {#*N*!*i*. *i* $\in$# *mset-set* {*a*..<*length N*}#}›
$\langle proof \rangle$

lemma *last-list-update-to-last*:
‹*last* (*xs*[*x* := *last xs*]) = *last xs*›
$\langle proof \rangle$

lemma *take-map-nth-alt-def*: ‹*take n xs* = *map* ((!) *xs*) [*0*..<*min n* (*length xs*)]›
$\langle proof \rangle$

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*))›
$\langle proof \rangle$

lemma *lexn-n*:
‹*n* > *0* $\Longrightarrow$ (*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*)))›
$\langle proof \rangle$

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*)›
$\langle proof \rangle$

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*)›
$\langle proof \rangle$

lemma *prepend-same-lexn*:
  assumes *irrefl*: ‹*irrefl R*›
  shows ‹$(A @ B,\ A @ C) \in$ *lexn R n* $\longleftrightarrow (B,\ C) \in$ *lexn R* $(n -$ *length A*$)$› (**is** ‹$?A \longleftrightarrow ?B$›)
⟨*proof*⟩

lemma *append-same-lexn*:
  assumes *irrefl*: ‹*irrefl R*›
  shows ‹$(B @ A\ ,\ C @ A) \in$ *lexn R n* $\longleftrightarrow (B,\ C) \in$ *lexn R* $(n -$ *length A*$)$› (**is** ‹$?A \longleftrightarrow ?B$›)
⟨*proof*⟩

lemma *irrefl-less-than* [*simp*]: ‹*irrefl less-than*›
  ⟨*proof*⟩

1.3.7 Remove

More lemmas about remove

lemma *distinct-remove1-last-butlast*:
  ‹*distinct xs* $\Longrightarrow xs \neq [] \Longrightarrow$ *remove1* (*last xs*) *xs* = *butlast xs*›
  ⟨*proof*⟩

lemma *remove1-Nil-iff*:
  ‹*remove1 x xs* $= [] \longleftrightarrow xs = [] \vee xs = [x]$›
  ⟨*proof*⟩

lemma *removeAll-upt*:
  ‹*removeAll k* $[a..<b] = ($*if* $k \geq a \wedge k < b$ *then* $[a..<k] @ [$*Suc* $k..<b]$ *else* $[a..<b])$›
  ⟨*proof*⟩

lemma *remove1-upt*:
  ‹*remove1 k* $[a..<b] = ($*if* $k \geq a \wedge k < b$ *then* $[a..<k] @ [$*Suc* $k..<b]$ *else* $[a..<b])$›
  ⟨*proof*⟩

lemma *sorted-removeAll*: ‹*sorted C* $\Longrightarrow$ *sorted* (*removeAll k C*)›
  ⟨*proof*⟩

lemma *distinct-remove1-rev*: ‹*distinct xs* $\Longrightarrow$ *remove1 x* (*rev xs*) = *rev* (*remove1 x xs*)›
  ⟨*proof*⟩

Remove under condition

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

fun *remove1-cond* **where**
‹*remove1-cond f* $[] = []$› |
‹*remove1-cond f* $(C' \# L) = ($*if f* C' *then* L *else* $C' \#$ *remove1-cond f L*$)$›

lemma ‹*remove1 x xs* = *remove1-cond* $((=)\ x)$ *xs*›
  ⟨*proof*⟩

lemma *mset-map-mset-remove1-cond*:
  ‹*mset* (*map mset* (*remove1-cond* (λL. *mset L* = *mset a*) *C*)) =
    remove1-mset (*mset a*) (*mset* (*map mset C*))›
  ⟨*proof*⟩

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

fun *removeAll-cond* :: ‹('a ⇒ bool) ⇒ 'a list ⇒ 'a list› **where**
‹removeAll-cond f [] = []› |
‹removeAll-cond f (C' # L) = (if f C' then removeAll-cond f L else C' # removeAll-cond f L)›

lemma *removeAll-removeAll-cond*: ‹removeAll x xs = removeAll-cond ((=) x) xs›
⟨proof⟩

lemma *removeAll-cond-filter*: ‹removeAll-cond P xs = filter (λx. ¬P x) xs›
⟨proof⟩

lemma *mset-map-mset-removeAll-cond*:
‹mset (map mset (removeAll-cond (λb. mset b = mset a) C))
= removeAll-mset (mset a) (mset (map mset C))›
⟨proof⟩

lemma *count-mset-count-list*:
‹count (mset xs) x = count-list xs x›
⟨proof⟩

lemma *length-removeAll-count-list*:
‹length (removeAll x xs) = length xs − count-list xs x›
⟨proof⟩

lemma *removeAll-notin*: ‹a ∉# A ⟹ removeAll-mset a A = A›
⟨proof⟩

Filter

lemma *distinct-filter-eq-if*:
‹distinct C ⟹ length (filter ((=) L) C) = (if L ∈ set C then 1 else 0)›
⟨proof⟩

lemma *length-filter-update-true*:
assumes ‹i < length xs› **and** ‹P (xs ! i)›
shows ‹length (filter P (xs[i := x])) = length (filter P xs) − (if P x then 0 else 1)›
⟨proof⟩

lemma *length-filter-update-false*:
assumes ‹i < length xs› **and** ‹¬P (xs ! i)›
shows ‹length (filter P (xs[i := x])) = length (filter P xs) + (if P x then 1 else 0)›
⟨proof⟩

lemma *mset-set-mset-set-minus-id-iff*:
assumes ‹finite A›
shows ‹mset-set A = mset-set (A − B) ⟷ (∀ b ∈ B. b ∉ A)›
⟨proof⟩

lemma *mset-set-eq-mset-set-more-conds*:
‹finite {x. P x} ⟹ mset-set {x. P x} = mset-set {x. Q x ∧ P x} ⟷ (∀ x. P x ⟶ Q x)›
(**is** ‹?F ⟹ ?A ⟷ ?B›)
⟨proof⟩

lemma *count-list-filter*: ‹count-list xs x = length (filter ((=) x) xs)›
⟨proof⟩

lemma *sum-length-filter-compl'*: ‹length [x←xs . ¬ P x] + length (filter P xs) = length xs›

$\langle proof \rangle$

1.3.8 Sorting

See $\llbracket sorted\ ?xs;\ distinct\ ?xs;\ sorted\ ?ys;\ distinct\ ?ys;\ set\ ?xs = set\ ?ys \rrbracket \Longrightarrow ?xs = ?ys$.

lemma *sorted-mset-unique*:
fixes $xs :: \langle 'a :: linorder\ list \rangle$
shows $\langle sorted\ xs \Longrightarrow sorted\ ys \Longrightarrow mset\ xs = mset\ ys \Longrightarrow xs = ys \rangle$
$\langle proof \rangle$

lemma *insort-upt*: $\langle insort\ k\ [a..<b] =$
$(if\ k < a\ then\ k\ \#\ [a..<b]$
$else\ if\ k < b\ then\ [a..<k]\ @\ k\ \#\ [k\ ..<b]$
$else\ [a..<b]\ @\ [k]) \rangle$
$\langle proof \rangle$

lemma *removeAll-insert-removeAll*: $\langle removeAll\ k\ (insort\ k\ xs) = removeAll\ k\ xs \rangle$
$\langle proof \rangle$

lemma *filter-sorted*: $\langle sorted\ xs \Longrightarrow sorted\ (filter\ P\ xs) \rangle$
$\langle proof \rangle$

lemma *removeAll-insort*:
$\langle sorted\ xs \Longrightarrow k \neq k' \Longrightarrow removeAll\ k'\ (insort\ k\ xs) = insort\ k\ (removeAll\ k'\ xs) \rangle$
$\langle proof \rangle$

1.3.9 Distinct Multisets

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

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

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

lemma *distinct-image-mset-not-equal*:
assumes
LL': $\langle L \neq L' \rangle$ **and**
dist: $\langle distinct\text{-}mset\ (image\text{-}mset\ f\ M) \rangle$ **and**
L: $\langle L \in\#\ M \rangle$ **and**
L': $\langle L' \in\#\ M \rangle$ **and**
fLL'[*simp*]: $\langle f\ L = f\ L' \rangle$
shows $\langle False \rangle$
$\langle proof \rangle$

1.3.10 Set of Distinct Multisets

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

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

$\langle proof \rangle$

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

lemma *distinct-mset-set-insert*[*iff*]:
‹*distinct-mset-set* (*insert* S Σ) $\longleftrightarrow$ (*distinct-mset* S $\wedge$ *distinct-mset-set* Σ)›
$\langle proof \rangle$

lemma *distinct-mset-set-union*[*iff*]:
‹*distinct-mset-set* ($\Sigma \cup \Sigma'$) $\longleftrightarrow$ (*distinct-mset-set* Σ $\wedge$ *distinct-mset-set* Σ')›
$\langle proof \rangle$

lemma *in-distinct-mset-set-distinct-mset*:
‹$a \in \Sigma \Longrightarrow$ *distinct-mset-set* $\Sigma \Longrightarrow$ *distinct-mset* a›
$\langle proof \rangle$

lemma *distinct-mset-remdups-mset*[*simp*]: ‹*distinct-mset* (*remdups-mset* S)›
$\langle proof \rangle$

lemma *distinct-mset-mset-set*: ‹*distinct-mset* (*mset-set* A)›
$\langle proof \rangle$

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

lemma *distinct-mset-set-distinct*: ‹*distinct-mset-set* (*mset* ‘ *set* Cs) $\longleftrightarrow$ ($\forall c \in$ *set* Cs. *distinct* c)›
$\langle proof \rangle$

1.3.11 Sublists

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

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

lemma *mset-nths-subset-mset*: ‹*mset* (*nths* xs A) $\subseteq\#$ *mset* xs›
$\langle proof \rangle$

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

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

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

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

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

⟨*proof*⟩

lemma *nths-upt-Suc*:
  assumes ‹i < *length xs*›
  shows ‹*nths xs* {i..<*length xs*} = *xs*!i # *nths xs* {*Suc* i..<*length xs*}›
⟨*proof*⟩

lemma *nths-upt-Suc′*:
  assumes ‹$i < b$› **and** ‹b <= *length xs*›
  shows ‹*nths xs* {i..<b} = *xs*!i # *nths xs* {*Suc* i..<b}›
⟨*proof*⟩

lemma *Ball-set-nths*: ‹(∀ *L*∈*set* (*nths xs A*). *P L*) ⟷ (∀ i ∈ *A* ∩ {*0*..<*length xs*}. *P* (*xs* ! i)) ›
  ⟨*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*]:
  (⋀*a b c d e f g h*. *P* (*a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*)) ⟹ *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*]:
  (⋀*a b c d e f g h i*. *P* (*a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, *i*)) ⟹ *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*]:
  (⋀*a b c d e f g h i j*. *P* (*a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, *i*, *j*)) ⟹ *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*]:
  (⋀*a b c d e f g h i j k*. *P* (*a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, *i*, *j*, *k*)) ⟹ *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*]:
$(\bigwedge a\ b\ c\ d\ e\ f\ g\ h\ i\ j\ k\ l.\ P\ (a,\ b,\ c,\ d,\ e,\ f,\ g,\ h,\ i,\ j,\ k,\ l)) \Longrightarrow P\ x$
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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

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

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*]:
$(\bigwedge a\ b\ c\ d\ e\ f\ g\ h\ i\ j\ k\ l\ m\ n\ p\ q\ r\ s\ t.$
$P\ (a, b, c, d, e, f, g, h, i, j, k, l, m, n, p, q, r, s, t)) \Longrightarrow P\ x$
⟨*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*]:
$(\bigwedge a\ b\ c\ d\ e\ f\ g\ h\ i\ j\ k\ l\ m\ n\ p\ q\ r\ s\ t\ u.$
$P\ (a, b, c, d, e, f, g, h, i, j, k, l, m, n, p, q, r, s, t, u)) \Longrightarrow P\ x$
⟨*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*]:
$(\bigwedge a\ b\ c\ d\ e\ f\ g\ h\ i\ j\ k\ l\ m\ n\ p\ q\ r\ s\ t\ u\ v.$
$P\ (a, b, c, d, e, f, g, h, i, j, k, l, m, n, p, q, r, s, t, u, v)) \Longrightarrow P\ x$
⟨*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*]:
$(\bigwedge a\ b\ c\ d\ e\ f\ g\ h\ i\ j\ k\ l\ m\ n\ p\ q\ r\ s\ t\ u\ v\ w.$
$P\ (a, b, c, d, e, f, g, h, i, j, k, l, m, n, p, q, r, s, t, u, v, w)) \Longrightarrow P\ x$
⟨*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*]:
$(\bigwedge a\ b\ c\ d\ e\ f\ g\ h\ i\ j\ k\ l\ m\ n\ p\ q\ r\ s\ t\ u\ v\ w\ y.$
$P\ (a, b, c, d, e, f, g, h, i, j, k, l, m, n, p, q, r, s, t, u, v, w, y)) \Longrightarrow P\ x$
⟨*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*: ‹*list-all2* $(\lambda L.\ (=)\ (f\ L))\ a\ aa \longleftrightarrow aa = map\ f\ a$ ›

⟨*proof*⟩

lemma *list-all2-op-eq-map-right-iff'*: ‹$list\text{-}all2\ (\lambda L\ L'.\ L' = f\ L)\ a\ aa \longleftrightarrow aa = map\ f\ a$›
⟨*proof*⟩

lemma *list-all2-op-eq-map-left-iff*: ‹$list\text{-}all2\ (\lambda L'\ L.\ L' = (f\ L))\ a\ aa \longleftrightarrow a = map\ f\ aa$›
⟨*proof*⟩

lemma *list-all2-op-eq-map-map-right-iff*:
‹$list\text{-}all2\ (list\text{-}all2\ (\lambda L.\ (=)\ (f\ L)))\ xs'\ x \longleftrightarrow x = map\ (map\ f)\ xs'$› **for** x
⟨*proof*⟩

lemma *list-all2-op-eq-map-map-left-iff*:
‹$list\text{-}all2\ (list\text{-}all2\ (\lambda L'\ L.\ L' = f\ L))\ xs'\ x \longleftrightarrow xs' = map\ (map\ f)\ x$›
⟨*proof*⟩

lemma *list-all2-conj*:
‹$list\text{-}all2\ (\lambda x\ y.\ P\ x\ y \land Q\ x\ y)\ xs\ ys \longleftrightarrow list\text{-}all2\ P\ xs\ ys \land list\text{-}all2\ Q\ xs\ ys$›
⟨*proof*⟩

lemma *list-all2-replicate*:
‹$(bi,\ b) \in R' \Longrightarrow list\text{-}all2\ (\lambda x\ x'.\ (x,\ x') \in R')\ (replicate\ n\ bi)\ (replicate\ n\ b)$›
⟨*proof*⟩

1.3.14 Multisets

We have a lit of lemmas about multisets. Some of them have already moved to *Nested-Multisets-Ordinals.Multiset* 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*: ‹$L \in\# D \Longrightarrow D \neq \{\#\}$›
⟨*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**
‹$union\text{-}mset\text{-}list\ xs\ ys \equiv case\text{-}prod\ append\ (fold\ (\lambda x\ (ys,\ zs).\ (remove1\ x\ ys,\ x\ \#\ zs))\ xs\ (ys,\ []))$›

lemma *union-mset-list*:
‹$mset\ xs \cup\# mset\ ys = mset\ (union\text{-}mset\text{-}list\ xs\ ys)$›
⟨*proof*⟩

lemma *union-mset-list-Nil*[*simp*]: ‹$union\text{-}mset\text{-}list\ []\ bi = bi$›
⟨*proof*⟩

lemma *size-le-Suc-0-iff*: ‹$size\ M \leq Suc\ 0 \longleftrightarrow ((\exists\ a\ b.\ M = \{\#a\#\}) \lor M = \{\#\})$›
⟨*proof*⟩

lemma *size-2-iff*: ‹$size\ M = 2 \longleftrightarrow (\exists\ a\ b.\ M = \{\#a,\ b\#\})$›
⟨*proof*⟩

lemma *subset-eq-mset-single-iff*: ‹$x2 \subseteq\# \{\#L\#\} \longleftrightarrow x2 = \{\#\} \lor x2 = \{\#L\#\}$›
⟨*proof*⟩

lemma *mset-eq-size-2*:
‹$mset\ xs = \{\#a,\ b\#\} \longleftrightarrow xs = [a,\ b] \lor xs = [b,\ a]$›
⟨*proof*⟩

lemma *butlast-list-update*:
‹$w < length\ xs \Longrightarrow butlast\ (xs[w := last\ xs]) = take\ w\ xs\ @\ butlast\ (last\ xs\ \#\ drop\ (Suc\ w)\ xs)$›
⟨*proof*⟩

lemma *mset-butlast-remove1-mset*: ‹$xs \neq [] \Longrightarrow mset\ (butlast\ xs) = remove1\text{-}mset\ (last\ xs)\ (mset\ xs)$›
⟨*proof*⟩

lemma *distinct-mset-mono*: ‹$D' \subseteq\#\ D \Longrightarrow distinct\text{-}mset\ D \Longrightarrow distinct\text{-}mset\ D'$›
⟨*proof*⟩

lemma *distinct-mset-mono-strict*: ‹$D' \subset\#\ D \Longrightarrow distinct\text{-}mset\ D \Longrightarrow distinct\text{-}mset\ D'$›
⟨*proof*⟩

lemma *subset-mset-trans-add-mset*:
‹$D \subseteq\#\ D' \Longrightarrow D \subseteq\#\ add\text{-}mset\ L\ D'$›
⟨*proof*⟩

lemma *subset-add-mset-notin-subset*: ‹$L \notin\#\ E \Longrightarrow E \subseteq\#\ add\text{-}mset\ L\ D \longleftrightarrow E \subseteq\#\ D$›
⟨*proof*⟩

lemma *remove1-mset-empty-iff*: ‹$remove1\text{-}mset\ L\ N = \{\#\} \longleftrightarrow N = \{\#L\#\} \lor N = \{\#\}$›
⟨*proof*⟩

lemma *distinct-subseteq-iff* :
assumes *dist*: *distinct-mset M* **and** *fin*: *distinct-mset N*
shows $set\text{-}mset\ M \subseteq set\text{-}mset\ N \longleftrightarrow M \subseteq\#\ N$
⟨*proof*⟩

lemma *distinct-set-mset-eq-iff*:
assumes ‹*distinct-mset M*› ‹*distinct-mset N*›
shows ‹$set\text{-}mset\ M = set\text{-}mset\ N \longleftrightarrow M = N$›
⟨*proof*⟩

lemma (**in** −) *distinct-mset-union2*:
‹$distinct\text{-}mset\ (A + B) \Longrightarrow distinct\text{-}mset\ B$›
⟨*proof*⟩

lemma *in-remove1-msetI*: ‹$x \neq a \Longrightarrow x \in\#\ M \Longrightarrow x \in\#\ remove1\text{-}mset\ a\ M$›
⟨*proof*⟩

lemma *count-multi-member-split*:
‹$count\ M\ a \geq n \Longrightarrow \exists M'.\ M = replicate\text{-}mset\ n\ a + M'$›
⟨*proof*⟩

lemma *count-image-mset-multi-member-split*:
‹$count\ (image\text{-}mset\ f\ M)\ L \geq Suc\ 0 \Longrightarrow\ \exists K.\ f\ K = L \land K \in\#\ M$›
⟨*proof*⟩

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

$M = \{\#K, K'\#\} + M'$›
⟨*proof*⟩

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

lemma *minus-notin-trivial2*: ‹$b \notin\# A \Longrightarrow A - \textit{add-mset } e\ (\textit{add-mset } b\ B) = A - \textit{add-mset } e\ B$›
⟨*proof*⟩

lemma *diff-union-single-conv3*: ‹$a \notin\# I \Longrightarrow \textit{remove1-mset } a\ (I + J) = I + \textit{remove1-mset } a\ J$›
⟨*proof*⟩

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

lemma *subset-mset-minus-eq-add-mset-noteq*: ‹$A \subset\# C \Longrightarrow A - B \neq C$›
⟨*proof*⟩

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

lemma *in-multiset-minus-notin-snd*[*simp*]: ‹$a \notin\# B \Longrightarrow a \in\# A - B \longleftrightarrow a \in\# A$›
⟨*proof*⟩

lemma *distinct-mset-in-diff*:
‹$\textit{distinct-mset } C \Longrightarrow a \in\# C - D \longleftrightarrow a \in\# C \wedge a \notin\# D$›
⟨*proof*⟩

lemma *diff-le-mono2-mset*: ‹$A \subseteq\# B \Longrightarrow C - B \subseteq\# C - A$›
⟨*proof*⟩

lemma *subseteq-remove1*[*simp*]: ‹$C \subseteq\# C' \Longrightarrow \textit{remove1-mset } L\ C \subseteq\# C'$›
⟨*proof*⟩

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

lemma *filter-mset-cong-inner-outer*:
assumes
M-eq: ‹$(\bigwedge x.\ x \in\# M \Longrightarrow f\ x = g\ x)$› **and**
notin: ‹$(\bigwedge x.\ x \in\# N - M \Longrightarrow \neg g\ x)$› **and**
MN: ‹$M \subseteq\# N$›
shows ‹$\textit{filter-mset } f\ M = \textit{filter-mset } g\ N$›
⟨*proof*⟩

lemma *notin-filter-mset*:
‹$K \notin\# C \Longrightarrow \textit{filter-mset } P\ C = \textit{filter-mset } (\lambda L.\ P\ L \wedge L \neq K)\ C$›
⟨*proof*⟩

lemma *distinct-mset-add-mset-filter*:
assumes ‹$\textit{distinct-mset } C$› **and** ‹$L \in\# C$› **and** ‹$\neg P\ L$›
shows ‹$\textit{add-mset } L\ (\textit{filter-mset } P\ C) = \textit{filter-mset } (\lambda x.\ P\ x \vee x = L)\ C$›
⟨*proof*⟩

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

lemma *remove1-mset-union-distrib*:
‹$remove1\text{-}mset\ a\ (M \cup\#\ N) = remove1\text{-}mset\ a\ M \cup\#\ remove1\text{-}mset\ a\ N$›
⟨*proof*⟩

lemma *member-add-mset*: ‹$a \in\#\ add\text{-}mset\ x\ xs \longleftrightarrow a = x \vee a \in\#\ xs$›
⟨*proof*⟩

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

lemma *same-mset-distinct-iff*:
‹$mset\ M = mset\ M' \Longrightarrow distinct\ M \longleftrightarrow distinct\ M'$›
⟨*proof*⟩

lemma *inj-on-image-mset-eq-iff*:
assumes *inj*: ‹$inj\text{-}on\ f\ (set\text{-}mset\ (M + M'))$›
shows ‹$image\text{-}mset\ f\ M' = image\text{-}mset\ f\ M \longleftrightarrow M' = M$› (**is** ‹$?A = ?B$›)
⟨*proof*⟩

lemma *inj-image-mset-eq-iff*:
assumes *inj*: ‹$inj\ f$›
shows ‹$image\text{-}mset\ f\ M' = image\text{-}mset\ f\ M \longleftrightarrow M' = M$›
⟨*proof*⟩

lemma *singleton-eq-image-mset-iff*: ‹$\{\#a\#\} = f\ `\#\ NE' \longleftrightarrow (\exists b.\ NE' = \{\#b\#\} \wedge f\ b = a)$›
⟨*proof*⟩

lemma *image-mset-If-eq-notin*:
‹$C \notin\#\ A \Longrightarrow \{\#f\ (if\ x = C\ then\ a\ x\ else\ b\ x).\ x \in\#\ A\#\} = \{\#\ f(b\ x).\ x \in\#\ A\ \#\}$›
⟨*proof*⟩

lemma *finite-mset-set-inter*:
‹$finite\ A \Longrightarrow finite\ B \Longrightarrow mset\text{-}set\ (A \cap B) = mset\text{-}set\ A \cap\#\ mset\text{-}set\ B$›
⟨*proof*⟩

lemma *distinct-mset-inter-remdups-mset*:
assumes *dist*: ‹$distinct\text{-}mset\ A$›
shows ‹$A \cap\#\ remdups\text{-}mset\ B = A \cap\#\ B$›
⟨*proof*⟩

lemma *mset-butlast-update-last*[*simp*]:
‹$w < length\ xs \Longrightarrow mset\ (butlast\ (xs[w := last\ (xs)])) = remove1\text{-}mset\ (xs\ !\ w)\ (mset\ xs)$›
⟨*proof*⟩

lemma *in-multiset-ge-Max*: ‹$a \in\#\ N \Longrightarrow a > Max\ (insert\ 0\ (set\text{-}mset\ N)) \Longrightarrow False$›
⟨*proof*⟩

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

‹*distinct-mset* $M \Longrightarrow$ *set-mset* (*remove1-mset* c M) $=$ *set-mset* $M - \{c\}$›
⟨*proof*⟩

lemma *distinct-count-msetD*:
‹*distinct xs* $\Longrightarrow$ *count* (*mset xs*) $a = ($*if* $a \in$ *set xs then 1 else 0*)›
⟨*proof*⟩

lemma *filter-mset-and-implied*:
‹($\bigwedge ia.\ ia \in\#\ xs \Longrightarrow Q\ ia \Longrightarrow P\ ia) \Longrightarrow \{\#ia \in\#\ xs.\ P\ ia \wedge Q\ ia\#\} = \{\#ia \in\#\ xs.\ Q\ ia\#\}$›
⟨*proof*⟩

lemma *filter-mset-eq-add-msetD*: ‹*filter-mset* P *xs* $=$ *add-mset* a $A \Longrightarrow a \in\#\ xs \wedge P\ a$›
⟨*proof*⟩

lemma *filter-mset-eq-add-msetD'*: ‹*add-mset* a A $=$ *filter-mset* P *xs* $\Longrightarrow a \in\#\ xs \wedge P\ a$›
⟨*proof*⟩

lemma *image-filter-replicate-mset*:
‹$\{\#Ca \in\#$ *replicate-mset* m C. $P\ Ca\#\} = ($*if* P C *then replicate-mset* m C *else* $\{\#\})$›
⟨*proof*⟩

lemma *size-Union-mset-image-mset*:
‹*size* $(\bigcup\#\ A) = (\sum i \in\#\ A.$ *size* $i)$›
⟨*proof*⟩

lemma *image-mset-minus-inj-on*:
‹*inj-on* f (*set-mset* $A \cup$ *set-mset* B) $\Longrightarrow f$ ‘# $(A - B) = f$ ‘# $A - f$ ‘# B›
⟨*proof*⟩

lemma *filter-mset-mono-subset*:
‹$A \subseteq\#\ B \Longrightarrow (\bigwedge x.\ x \in\#\ A \Longrightarrow P\ x \Longrightarrow Q\ x) \Longrightarrow$ *filter-mset* P $A \subseteq\#$ *filter-mset* Q B›
⟨*proof*⟩

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

lemma *sum-mset-mset-set-sum-set*:
‹$(\sum A \in\#$ *mset-set* $As.\ f\ A) = (\sum A \in As.\ f\ A)$›
⟨*proof*⟩

lemma *sum-mset-sum-count*:
‹$(\sum A \in\#\ As.\ f\ A) = (\sum A \in$ *set-mset* $As.$ *count* $As\ A * f\ A)$›
⟨*proof*⟩

lemma *sum-mset-inter-restrict*:
‹$(\sum\ x \in\#$ *filter-mset* P $M.\ f\ x) = (\sum\ x \in\#\ M.$ *if* $P\ x$ *then* $f\ x$ *else 0*)›
⟨*proof*⟩

lemma *mset-set-subset-iff*:
‹*mset-set* $A \subseteq\#\ I \longleftrightarrow$ *infinite* $A \vee A \subseteq$ *set-mset* I›
⟨*proof*⟩

lemma *sumset-diff-constant-left*:
assumes ‹$\bigwedge x.\ x \in\#\ A \Longrightarrow f\ x \le n$›

shows ‹$(\sum x \in\# A . n - f x) = size\ A * n - (\sum x \in\# A . f x)$›
⟨*proof*⟩

lemma *mset-set-eq-mset-iff*: ‹$finite\ x \Longrightarrow mset\text{-}set\ x = mset\ xs \longleftrightarrow distinct\ xs \wedge x = set\ xs$›
⟨*proof*⟩

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

1.4 Finite maps and multisets

Finite sets and multisets

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

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

lemma *fset-mset-mset-fset*: ‹$fset\text{-}mset\ (mset\text{-}fset\ N) = N$›
⟨*proof*⟩

lemma *mset-fset-fset-mset*[*simp*]:
‹$mset\text{-}fset\ (fset\text{-}mset\ N) = remdups\text{-}mset\ N$›
⟨*proof*⟩

lemma *in-mset-fset-fmember*[*simp*]: ‹$x \in\# mset\text{-}fset\ N \longleftrightarrow x\ |\in|\ N$›
⟨*proof*⟩

lemma *in-fset-mset-mset*[*simp*]: ‹$x\ |\in|\ fset\text{-}mset\ N \longleftrightarrow x \in\# N$›
⟨*proof*⟩

lemma *distinct-mset-subset-iff-remdups*:
‹$distinct\text{-}mset\ a \Longrightarrow a \subseteq\# b \longleftrightarrow a \subseteq\# remdups\text{-}mset\ b$›
⟨*proof*⟩

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**
‹$dom\text{-}m\ N = mset\text{-}fset\ (fmdom\ N)$›

definition *ran-m* **where**
‹$ran\text{-}m\ N = the\ `\#\ fmlookup\ N\ `\#\ dom\text{-}m\ N$›

lemma *dom-m-fmdrop*[*simp*]: ‹$dom\text{-}m\ (fmdrop\ C\ N) = remove1\text{-}mset\ C\ (dom\text{-}m\ N)$›
⟨*proof*⟩

lemma *dom-m-fmdrop-All*: ‹$dom\text{-}m\ (fmdrop\ C\ N) = removeAll\text{-}mset\ C\ (dom\text{-}m\ N)$›
⟨*proof*⟩

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

lemma *distinct-mset-dom*: ‹$distinct\text{-}mset\ (dom\text{-}m\ N)$›
⟨*proof*⟩

lemma *in-dom-m-lookup-iff*: ‹$C \in\#\ dom\text{-}m\ N' \longleftrightarrow fmlookup\ N'\ C \neq None$›
⟨*proof*⟩

lemma *in-dom-in-ran-m*[*simp*]: ‹$i \in\#\ dom\text{-}m\ N \Longrightarrow the\ (fmlookup\ N\ i) \in\#\ ran\text{-}m\ N$›
⟨*proof*⟩

lemma *fmupd-same*[*simp*]:
‹$x1 \in\#\ dom\text{-}m\ x1aa \Longrightarrow fmupd\ x1\ (the\ (fmlookup\ x1aa\ x1))\ x1aa = x1aa$›
⟨*proof*⟩

lemma *ran-m-fmempty*[*simp*]: ‹$ran\text{-}m\ fmempty = \{\#\}$› **and**
dom-m-fmempty[*simp*]: ‹$dom\text{-}m\ fmempty = \{\#\}$›
⟨*proof*⟩

lemma *fmrestrict-set-fmupd*:
‹$a \in xs \Longrightarrow fmrestrict\text{-}set\ xs\ (fmupd\ a\ C\ N) = fmupd\ a\ C\ (fmrestrict\text{-}set\ xs\ N)$›
‹$a \notin xs \Longrightarrow fmrestrict\text{-}set\ xs\ (fmupd\ a\ C\ N) = fmrestrict\text{-}set\ xs\ N$›
⟨*proof*⟩

lemma *fset-fmdom-fmrestrict-set*:
‹$fset\ (fmdom\ (fmrestrict\text{-}set\ xs\ N)) = fset\ (fmdom\ N) \cap xs$›
⟨*proof*⟩

lemma *dom-m-fmrestrict-set*: ‹$dom\text{-}m\ (fmrestrict\text{-}set\ (set\ xs)\ N) = mset\ xs \cap\#\ dom\text{-}m\ N$›
⟨*proof*⟩

lemma *dom-m-fmrestrict-set'*: ‹$dom\text{-}m\ (fmrestrict\text{-}set\ xs\ N) = mset\text{-}set\ (xs \cap set\text{-}mset\ (dom\text{-}m\ N))$›
⟨*proof*⟩

lemma *indom-mI*: ‹$fmlookup\ m\ x = Some\ y \Longrightarrow x \in\#\ dom\text{-}m\ m$›
⟨*proof*⟩

lemma *fmupd-fmdrop-id*:
assumes ‹$k\ |\in|\ fmdom\ N'$›
shows ‹$fmupd\ k\ (the\ (fmlookup\ N'\ k))\ (fmdrop\ k\ N') = N'$›
⟨*proof*⟩

lemma *fm-member-split*: ‹$k\ |\in|\ fmdom\ N' \Longrightarrow \exists\, N''\ v.\ N' = fmupd\ k\ v\ N'' \wedge the\ (fmlookup\ N'\ k) = v$
$\wedge$
$k\ |\notin|\ fmdom\ N''$›
⟨*proof*⟩

lemma ‹$fmdrop\ k\ (fmupd\ k\ va\ N'') = fmdrop\ k\ N''$›
⟨*proof*⟩

lemma *fmap-ext-fmdom*:
‹$(fmdom\ N = fmdom\ N') \Longrightarrow (\bigwedge\ x.\ x\ |\in|\ fmdom\ N \Longrightarrow fmlookup\ N\ x = fmlookup\ N'\ x) \Longrightarrow$
$N = N'$›
⟨*proof*⟩

lemma *fmrestrict-set-insert-in*:
‹$xa \ \in fset\ (fmdom\ N) \Longrightarrow$
$fmrestrict\text{-}set\ (insert\ xa\ l1)\ N = fmupd\ xa\ (the\ (fmlookup\ N\ xa))\ (fmrestrict\text{-}set\ l1\ N)$›
⟨*proof*⟩

lemma *fmrestrict-set-insert-notin*:
‹$xa \ \notin fset\ (fmdom\ N) \Longrightarrow$
$fmrestrict\text{-}set\ (insert\ xa\ l1)\ N = fmrestrict\text{-}set\ l1\ N$›
⟨*proof*⟩

lemma *fmrestrict-set-insert-in-dom-m*[*simp*]:
‹$xa \ \in\#\ dom\text{-}m\ N \Longrightarrow$
$fmrestrict\text{-}set\ (insert\ xa\ l1)\ N = fmupd\ xa\ (the\ (fmlookup\ N\ xa))\ (fmrestrict\text{-}set\ l1\ N)$›
⟨*proof*⟩

lemma *fmrestrict-set-insert-notin-dom-m*[*simp*]:
‹$xa \ \notin\#\ dom\text{-}m\ N \Longrightarrow$
$fmrestrict\text{-}set\ (insert\ xa\ l1)\ N = fmrestrict\text{-}set\ l1\ N$›
⟨*proof*⟩

lemma *fmlookup-restrict-set-id*: ‹$fset\ (fmdom\ N) \subseteq A \Longrightarrow fmrestrict\text{-}set\ A\ N = N$›
⟨*proof*⟩

lemma *fmlookup-restrict-set-id'*: ‹$set\text{-}mset\ (dom\text{-}m\ N) \subseteq A \Longrightarrow fmrestrict\text{-}set\ A\ N = N$›
⟨*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**
‹$Max\text{-}dom\ N = Max\ (set\text{-}mset\ (add\text{-}mset\ 0\ (dom\text{-}m\ N)))$›

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

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

lemma *Max-dom-empty*: ‹$dom\text{-}m\ b = \{\#\} \Longrightarrow Max\text{-}dom\ b = 0$›
⟨*proof*⟩

lemma *Max-dom-fmempty*: ‹$Max\text{-}dom\ fmempty = 0$›
⟨*proof*⟩

lemma *packed-empty*[*simp*]: ‹$packed\ fmempty$›
⟨*proof*⟩

lemma *packed-Max-dom-size*:
assumes *p*: ‹$packed\ N$›
shows ‹$Max\text{-}dom\ N = size\ (dom\text{-}m\ N)$›
⟨*proof*⟩

lemma *Max-dom-le*:
‹$L \in\# \ dom\text{-}m\ N \Longrightarrow L \leq Max\text{-}dom\ N$›
⟨*proof*⟩

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

lemma *Max-dom-fmupd-irrel*:
‹$(a :: 'a :: \{zero,linorder\}) > Max\text{-}dom\ M \Longrightarrow Max\text{-}dom\ (fmupd\ a\ C\ M) = max\ a\ (Max\text{-}dom\ M)$›
⟨*proof*⟩

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

lemma *Max-insert-Suc-Max-dim-dom*[*simp*]:
‹$Max\ (insert\ (Suc\ (Max\text{-}dom\ b))\ (set\text{-}mset\ (dom\text{-}m\ b))) = Suc\ (Max\text{-}dom\ b)$›
⟨*proof*⟩

lemma *size-dom-m-Max-dom*:
‹$size\ (dom\text{-}m\ N) \leq Suc\ (Max\text{-}dom\ N)$›
⟨*proof*⟩

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

lemma *Max-atLeastLessThan*: ‹$Max\ \{(a::nat)\ ..<\ b\} = (if\ b \leq a\ then\ Max\ \{\}\ else\ b - 1)$›
⟨*proof*⟩

lemma *Max-insert-Max-dom-into-packed*:
‹$Max\ (insert\ (Max\text{-}dom\ bc)\ \{Suc\ 0..<Max\text{-}dom\ bc\}) = Max\text{-}dom\ bc$›
⟨*proof*⟩

lemma *packed0-fmud-Suc-Max-dom*: ‹$packed\ b \Longrightarrow packed\ (fmupd\ (Suc\ (Max\text{-}dom\ b))\ C\ b)$›
⟨*proof*⟩

lemma *ge-Max-dom-notin-dom-m*: ‹$a > Max\text{-}dom\ ao \Longrightarrow a \notin\# \ dom\text{-}m\ ao$›
⟨*proof*⟩

lemma *packed-in-dom-mI*: ‹$packed\ bc \Longrightarrow j \leq Max\text{-}dom\ bc \Longrightarrow 0 < j \Longrightarrow j \in\# \ dom\text{-}m\ bc$›
⟨*proof*⟩

lemma *mset-fset-empty-iff*: ‹$mset\text{-}fset\ a = \{\#\} \longleftrightarrow a = fempty$›
⟨*proof*⟩

lemma *dom-m-empty-iff*[*iff*]:
‹$dom\text{-}m\ NU = \{\#\} \longleftrightarrow NU = fmempty$›
⟨*proof*⟩

lemma *nat-power-div-base*:
fixes $k :: nat$
assumes $0 < m\ 0 < k$
shows $k\ \hat{}\ m\ div\ k = (k::nat)\ \hat{}\ (m - Suc\ 0)$

⟨*proof*⟩

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-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
      |> 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)])

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
$distinct\ xs \Longrightarrow P\ xs \Longrightarrow length\ (filter\ (\lambda x.\ x = y)\ xs) \leq 1$ **for** *xs*
⟨*proof*⟩

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

explore-context
explore-have
explore-lemma
⟨*proof*⟩

You can also choose quotes

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

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

$\langle proof \rangle$

And switch back

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

lemma
distinct xs $\Longrightarrow$ *P xs* $\Longrightarrow$ *length* (*filter* ($\lambda x.\ x = y$) *xs*) $\leq$ *1* **for** *xs*
$\langle proof \rangle$

end