Formalisation of Ground Resolution and CDCL in Isabelle/HOL

Mathias Fleury and Jasmin Blanchette

July 17, 2023

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

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

Near duplicate of theorem *tranclpD*:

lemma *tranclp-unfold-begin*: ‹$tranclp\ r\ a\ b \longleftrightarrow (\exists a'.\ r\ a\ a' \land rtranclp\ r\ a'\ b)$›
⟨*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 ‹$\exists y\ y'.\ R^{**}\ x\ y \land R\ y\ y' \land P\ y\ y' \land (\lambda a\ b.\ R\ a\ b \land \neg P\ a\ b)^{**}\ y'\ z'$›
⟨*proof*⟩

lemma *rtranclp-and-rtranclp-left*: ‹$(\lambda\ a\ b.\ P\ a\ b \land Q\ a\ b)^{**}\ S\ T \Longrightarrow P^{**}\ S\ T$›
⟨*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**
$full1\ transf = (\lambda S\ S'.\ tranclp\ transf\ S\ S' \land no\text{-}step\ transf\ S')$

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

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* ⟹ *full R b c* ⟹ *full1 R a c*›
⟨*proof*⟩

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

lemma *full-unfold*:
‹*full r S S'* ⟷ ((*S* = *S'* ∧ *no-step r S'*) ∨ *full1 r S S'*)›
⟨*proof*⟩

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

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

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

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

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

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

lemma *rtranclp-full1-eq-or-full1*:
‹(*full1 R*)** *a b* ⟷ (*a* = *b* ∨ *full1 R a b*)›
⟨*proof*⟩

lemma *no-step-full-iff-eq*:
‹*no-step R S* ⟹ *full R S T* ⟷ *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 ‹∃ *b*. R^{**} *a b* ∧ *no-step R b*›
⟨*proof*⟩

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

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

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

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

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

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

lemma *lexord-on-finite-set-is-wf*:
assumes
P-finite: ‹$\bigwedge U.\ P\ U \longrightarrow U \in A$› **and**
finite: ‹$finite\ A$› **and**
wf: ‹$wf\ R$› **and**
trans: ‹$trans\ R$›
shows ‹$wf\ \{(T,\ S).\ (P\ S \wedge P\ T) \wedge (T,\ S) \in lexord\ R\}$›
⟨*proof*⟩

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

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

lemma *wf-if-measure-f-notation2*:
assumes ‹$wf\ r$›
shows ‹$wf\ \{(b,\ h\ a)|b\ a.\ (f\ b,\ f\ (h\ a)) \in r\}$›
⟨*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 ‹$(R^{\frown\frown}n)\ S\ T$›
  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: ‹$\bigwedge i\ j.\ i \le n \Longrightarrow j < m \Longrightarrow (f\ i)\ !\ j \in \{0..<k\}$› **and**
    ‹*finite A*› **and**
    k: ‹$k > 1$›
  shows ‹$n < k\ \hat{}\ Suc\ m$›
⟨*proof*⟩

end
theory *WB-List-More*
  imports *HOL−Library.Finite-Map*
    Nested-Multisets-Ordinals.Duplicate-Free-Multiset
    HOL−Eisbach.Eisbach
    HOL−Eisbach.Eisbach-Tools
    HOL−Library.FuncSet
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**

$\langle\bigwedge n.\ (\forall m < Suc\ n.\ P\ m) \Longrightarrow P\ (Suc\ n)\rangle$
shows $\langle P\ n\rangle$
$\langle proof\rangle$

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

lemma *if-0-1-ge-0*[*simp*]:
$\langle 0 < (\textit{if}\ P\ \textit{then}\ a\ \textit{else}\ (0{::}nat)) \longleftrightarrow P \wedge 0 < a\rangle$
$\langle proof\rangle$

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

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

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

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

Bounded function have not yet been defined in Isabelle.

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

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

lemma *not-bounded-nat-exists-larger*:
fixes f :: $\langle nat \Rightarrow nat\rangle$
assumes *unbound*: $\langle unbounded\ f\rangle$
shows $\langle\exists n.\ f\ n > m \wedge n > n_0\rangle$
$\langle proof\rangle$

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 :: $\langle nat \Rightarrow nat\rangle$
assumes $\langle k > 0\rangle$
shows $\langle bounded\ f \longleftrightarrow bounded\ (\lambda i.\ k * f\ i)\rangle$
$\langle proof\rangle$

lemma *bounded-const-add*:
fixes k :: *nat* **and** f :: $\langle nat \Rightarrow nat\rangle$
assumes $\langle k > 0\rangle$
shows $\langle bounded\ f \longleftrightarrow bounded\ (\lambda i.\ k + f\ i)\rangle$
$\langle proof\rangle$

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 ⇒ ′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*:
‹$(\bigwedge i.\ i < min\ (length\ xs)\ (length\ ys) \Longrightarrow (xs\ !\ i,\ ys\ !\ i) = (xs'\ !\ i,\ ys'\ !\ i)) \Longrightarrow$
$length\ xs = length\ xs' \Longrightarrow length\ ys \le length\ ys' \Longrightarrow length\ ys \ge length\ xs \Longrightarrow$
$zip\ xs\ ys = zip\ xs'\ ys'$›
⟨*proof*⟩

1.3.2 List Updates

lemma *tl-update-swap*:
‹$i \ge 1 \Longrightarrow tl\ (N[i := C]) = (tl\ N)[i-1 := C]$›
⟨*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 the Word library.

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*: ‹$aa < length\ xs \Longrightarrow nths\ xs\ \{0..<Suc\ aa\} = nths\ xs\ \{0..<aa\}\ @\ [xs\ !\ aa]$›
⟨*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*: ‹$mset\ (drop\ a\ N) = \{\#N!i.\ i \in\#\ mset\text{-}set\ \{a..<length\ N\}\#\}$›
⟨*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*:
‹$n > 0 \Longrightarrow (x\ \#\ xs, y\ \#\ ys) \in lexn\ r\ n \longleftrightarrow$
$(length\ xs = n-1 \wedge length\ ys = n-1) \wedge ((x, y) \in r \vee (x = y \wedge (xs, ys) \in lexn\ r\ (n - 1)))$›
⟨*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*:
  $‹([a, b, c], [a', b', c']) \in \textit{lexn } r \textit{ 3} \longleftrightarrow$
    $(a, a') \in r \lor (a = a' \land (b, b') \in r) \lor (a = a' \land b = b' \land (c, c') \in r)›$
  ⟨*proof*⟩

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

lemma *append-same-lexn*:
  assumes *irrefl*: ‹*irrefl R*›
  shows $‹(B \mathbin{@} A , C \mathbin{@} A) \in \textit{lexn } R\ n \longleftrightarrow (B, C) \in \textit{lexn } R\ (n - \textit{length } A)›$ (**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*:
  $‹\textit{distinct } xs \Longrightarrow xs \neq [] \Longrightarrow \textit{remove1 } (\textit{last } xs)\ xs = \textit{butlast } xs›$
  ⟨*proof*⟩

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

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

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

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

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

Remove under condition

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

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

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

lemma *mset-map-mset-remove1-cond*:
  $‹\textit{mset } (\textit{map mset } (\textit{remove1-cond } (\lambda L.\ \textit{mset } L = \textit{mset } a)\ C)) =$

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* (λ*x*. ¬*P x*) *xs*›
⟨*proof*⟩

lemma *mset-map-mset-removeAll-cond*:
‹*mset* (*map mset* (*removeAll-cond* (λ*b*. *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*›
⟨*proof*⟩

1.3.8 Sorting

See ⟦*sorted ?xs*; *distinct ?xs*; *sorted ?ys*; *distinct ?ys*; *set ?xs = set ?ys*⟧ ⟹ *?xs = ?ys*.

lemma *sorted-mset-unique*:
fixes *xs* :: ‹′*a* :: *linorder list*›
shows ‹*sorted xs* ⟹ *sorted ys* ⟹ *mset xs = mset ys* ⟹ *xs = ys*›
⟨*proof*⟩

lemma *insort-upt*: ‹*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])›
⟨*proof*⟩

lemma *removeAll-insert-removeAll*: ‹*removeAll k (insort k xs) = removeAll k xs*›
⟨*proof*⟩

lemma *filter-sorted*: ‹*sorted xs* ⟹ *sorted (filter P xs)*›
⟨*proof*⟩

lemma *removeAll-insort*:
‹*sorted xs* ⟹ *k ≠ k′* ⟹ *removeAll k′ (insort k xs) = insort k (removeAll k′ xs)*›
⟨*proof*⟩

1.3.9 Distinct Multisets

lemma *distinct-mset-remdups-mset-id*: ‹*distinct-mset C* ⟹ *remdups-mset C = C*›
⟨*proof*⟩

lemma *notin-add-mset-remdups-mset*:
‹*a ∉# A* ⟹ *add-mset a (remdups-mset A) = remdups-mset (add-mset a A)*›
⟨*proof*⟩

lemma *distinct-mset-image-mset*:
‹*distinct-mset (image-mset f (mset xs))* ⟷ *distinct (map f xs)*›
⟨*proof*⟩

lemma *distinct-image-mset-not-equal*:
assumes
LL′: ‹*L ≠ L′*› **and**
dist: ‹*distinct-mset (image-mset f M)*› **and**
L: ‹*L ∈# M*› **and**
L′: ‹*L′ ∈# M*› **and**
fLL′[*simp*]: ‹*f L = f L′*›
shows ‹*False*›
⟨*proof*⟩

lemma *distinct-mset-remdups-mset*[*simp*]: ‹*distinct-mset (remdups-mset S)*›
⟨*proof*⟩

lemma *remdups-mset-idem*: ‹$remdups\text{-}mset\ (remdups\text{-}mset\ a) = remdups\text{-}mset\ a$›
⟨*proof*⟩

1.3.10 Set of Distinct Multisets

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

lemma *distinct-mset-set-empty*[*simp*]: ‹$distinct\text{-}mset\text{-}set\ \{\}$›
⟨*proof*⟩

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

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

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

lemma *in-distinct-mset-set-distinct-mset*:
‹$a \in \Sigma \Longrightarrow distinct\text{-}mset\text{-}set\ \Sigma \Longrightarrow distinct\text{-}mset\ a$›
⟨*proof*⟩

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

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

lemma *distinct-mset-set-distinct*: ‹$distinct\text{-}mset\text{-}set\ (mset\ `\ set\ Cs) \longleftrightarrow (\forall c \in set\ Cs.\ distinct\ c)$›
⟨*proof*⟩

1.3.11 Sublists

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

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

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

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

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

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

⟨*proof*⟩

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

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*: ‹$(\forall L \in set\ (nths\ xs\ A).\ P\ L) \longleftrightarrow (\forall i \in A \cap \{0..<length\ xs\}.\ P\ (xs\ !\ i))$ ›
⟨*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*]:
$(\bigwedge a\ b\ c\ d\ e\ f\ g\ h.\ P\ (a, b, c, d, e, f, g, h)) \Longrightarrow P\ x$
⟨*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*]:
$(\bigwedge a\ b\ c\ d\ e\ f\ g\ h\ i.\ P\ (a, b, c, d, e, f, g, h, i)) \Longrightarrow P\ x$
⟨*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*]:
$(\bigwedge a\ b\ c\ d\ e\ f\ g\ h\ i\ j.\ P\ (a, b, c, d, e, f, g, h, i, j)) \Longrightarrow P\ x$
⟨*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*]:

$(\bigwedge a\ b\ c\ d\ e\ f\ g\ h\ i\ j\ k.\ P\ (a,\ b,\ c,\ d,\ e,\ f,\ g,\ h,\ i,\ j,\ k)) \Longrightarrow P\ x$
⟨*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*: ‹$list\text{-}all2\ (\lambda L.\ (=)\ (f\ L))\ a\ aa \longleftrightarrow aa = map\ f\ a$ ›
⟨*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*:
‹$list\text{-}all2\ (\lambda x\ y.\ P\ x\ y \wedge Q\ x\ y)\ xs\ ys \longleftrightarrow list\text{-}all2\ P\ xs\ ys \wedge list\text{-}all2\ Q\ xs\ ys$›
⟨*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.Multise* 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*: ‹$size\ M \leq Suc\ 0 \longleftrightarrow ((\exists\ a\ b.\ M = \{\#a\#\}) \vee M = \{\#\})$›
⟨*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*:
‹*mset xs* = {#*a*, *b*#} ⟷ *xs* = [*a*, *b*] ∨ *xs* = [*b*, *a*]›
⟨*proof*⟩

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

lemma *mset-butlast-remove1-mset*: ‹*xs* ≠ [] ⟹ *mset* (*butlast xs*) = *remove1-mset* (*last xs*) (*mset xs*)›
⟨*proof*⟩

lemma *distinct-mset-mono*: ‹*D*′ ⊆# *D* ⟹ *distinct-mset D* ⟹ *distinct-mset D*′›
⟨*proof*⟩

lemma *distinct-mset-mono-strict*: ‹*D*′ ⊂# *D* ⟹ *distinct-mset D* ⟹ *distinct-mset D*′›
⟨*proof*⟩

lemma *subset-mset-trans-add-mset*:
‹*D* ⊆# *D*′ ⟹ *D* ⊆# *add-mset L D*′›
⟨*proof*⟩

lemma *subset-add-mset-notin-subset*: ‹*L* ∉# *E* ⟹ *E* ⊆# *add-mset L D* ⟷ *E* ⊆# *D*›
⟨*proof*⟩

lemma *remove1-mset-empty-iff*: ‹*remove1-mset L N* = {#} ⟷ *N* = {#*L*#} ∨ *N* = {#}›
⟨*proof*⟩

lemma *mset-set-subset-iff*:
‹*mset-set A* ⊆# *I* ⟷ *infinite A* ∨ *A* ⊆ *set-mset I*›
⟨*proof*⟩

lemma *distinct-subseteq-iff*:
assumes *dist*: ‹*distinct-mset M*›
shows ‹*set-mset M* ⊆ *set-mset N* ⟷ *M* ⊆# *N*›
⟨*proof*⟩

lemma *distinct-set-mset-eq-iff*:
assumes ‹*distinct-mset M*› ‹*distinct-mset N*›
shows ‹*set-mset M* = *set-mset N* ⟷ *M* = *N*›
⟨*proof*⟩

lemma (**in** −) *distinct-mset-union2*:
‹*distinct-mset* (*A* + *B*) ⟹ *distinct-mset B*›
⟨*proof*⟩

lemma *in-remove1-msetI*: ‹*x* ≠ *a* ⟹ *x* ∈# *M* ⟹ *x* ∈# *remove1-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*:
‹$count\ (image\text{-}mset\ f\ M)\ L \geq Suc\ 0 \Longrightarrow\ \exists K.\ f\ K = L \wedge K \in\#\ M$›
⟨*proof*⟩

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

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

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

lemma *diff-union-single-conv3*: ‹$a \notin\#\ I \Longrightarrow remove1\text{-}mset\ a\ (I + J) = I + remove1\text{-}mset\ a\ J$›
⟨*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*:
‹$distinct\text{-}mset\ C \Longrightarrow a \in\#\ C - D \longleftrightarrow a \in\#\ C \wedge a \notin\#\ D$›
⟨*proof*⟩

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

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

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

⟨*proof*⟩

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

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

lemma *set-mset-set-mset-eq-iff*: ‹*set-mset* A = *set-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-mset* a ($M \cup\# N$) = *remove1-mset* a $M \cup\#$ *remove1-mset* a N›
⟨*proof*⟩

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

lemma *sup-union-right-if*:
‹$N \cup\#$ *add-mset* x M =
(*if* $x \notin\# N$ *then add-mset* x ($N \cup\# M$) *else add-mset* x (*remove1-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-on* f (*set-mset* ($M + M'$))›
shows ‹*image-mset* f M' = *image-mset* f $M \longleftrightarrow M' = M$› (**is** ‹*?A* = *?B*›)
⟨*proof*⟩

lemma *image-msetI*: ‹$x \in\# A \Longrightarrow f\ x \in\# f$ ‘# A›
⟨*proof*⟩

lemma *inj-image-mset-eq-iff*:
assumes *inj*: ‹*inj* f›
shows ‹*image-mset* f M' = *image-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-set* ($A \cap B$) = *mset-set* $A \cap\#$ *mset-set* B›
⟨*proof*⟩

lemma *distinct-mset-inter-remdups-mset*:
assumes *dist*: ‹*distinct-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*:
‹$distinct\text{-}mset\ M \Longrightarrow set\text{-}mset\ (remove1\text{-}mset\ c\ M) = set\text{-}mset\ M - \{c\}$›
⟨*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*: ‹$filter\text{-}mset\ P\ xs = add\text{-}mset\ a\ A \Longrightarrow a \in\# \ xs \wedge P\ a$›
⟨*proof*⟩

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

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

lemma *size-Union-mset-image-mset*:
‹$size\ (\sum_{\#}\ (A :: 'a\ multiset\ multiset)) = (\sum i \in\# \ A.\ size\ i)$›
⟨*proof*⟩

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

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

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

lemma *sum-mset-cong*:
‹$(\bigwedge A.\ A \in\# \ M \Longrightarrow f\ A = g\ A) \Longrightarrow (\sum\ A \in\# \ M.\ f\ A) = (\sum\ A \in\# \ M.\ g\ A)$›
⟨*proof*⟩

lemma *sum-mset-mset-set-sum-set*:
‹$(\sum A \in\# \ mset\text{-}set\ As.\ f\ A) = (\sum A \in As.\ f\ A)$›

⟨*proof*⟩

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

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

lemma *diff-add-mset-remove1*: ‹$NO\text{-}MATCH\ \{\#\}\ N \Longrightarrow M - add\text{-}mset\ a\ N = remove1\text{-}mset\ a\ (M - N)$›
⟨*proof*⟩

lemma *remdups-mset-sum-subset*: ‹$C \subseteq\# C' \Longrightarrow remdups\text{-}mset\ (C + C') = remdups\text{-}mset\ C'$›
‹$C \subseteq\# C' \Longrightarrow remdups\text{-}mset\ (C' + C) = remdups\text{-}mset\ C'$›
⟨*proof*⟩

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

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

lemma *subset-mset-removeAll-iff*:
‹$M \subseteq\# removeAll\text{-}mset\ a\ M' \longleftrightarrow a \notin\# M \wedge M \subseteq\# M'$›
⟨*proof*⟩

lemma *remdups-mset-removeAll*: ‹$remdups\text{-}mset\ (removeAll\text{-}mset\ a\ A) = removeAll\text{-}mset\ a\ (remdups\text{-}mset\ A)$›
⟨*proof*⟩

This is an alternative to *remdups-mset-singleton-sum*.

lemma *remdups-mset-singleton-removeAll*:
$remdups\text{-}mset\ (add\text{-}mset\ a\ A) = add\text{-}mset\ a\ (removeAll\text{-}mset\ a\ (remdups\text{-}mset\ A))$
⟨*proof*⟩

lemma *mset-remove-filtered*: ‹$C - \{\#x \in\# C.\ P\ x\#\} = \{\#x \in\# C.\ \neg P\ x\#\}$›
⟨*proof*⟩

1.4 Finite maps and multisets

Finite sets and multisets

abbreviation *mset-fset* :: ‹*'a fset* ⇒ *'a multiset*› **where**
‹*mset-fset N* ≡ *mset-set* (*fset N*)›

definition *fset-mset* :: ‹*'a multiset* ⇒ *'a fset*› **where**
‹*fset-mset N* ≡ *Abs-fset* (*set-mset N*)›

lemma *fset-mset-mset-fset*: ‹*fset-mset* (*mset-fset N*) = *N*›
⟨*proof*⟩

lemma *mset-fset-fset-mset*[*simp*]:
‹*mset-fset* (*fset-mset N*) = *remdups-mset N*›
⟨*proof*⟩

lemma *in-mset-fset-fmember*[*simp*]: ‹*x* ∈# *mset-fset N* ⟷ *x* |∈| *N*›
⟨*proof*⟩

lemma *in-fset-mset-mset*[*simp*]: ‹*x* |∈| *fset-mset N* ⟷ *x* ∈# *N*›
⟨*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-m N* = *mset-fset* (*fmdom N*)›

definition *ran-m* **where**
‹*ran-m N* = *the* '# *fmlookup N* '# *dom-m N*›

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

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

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

lemma *distinct-mset-dom*: ‹*distinct-mset* (*dom-m N*)›
⟨*proof*⟩

lemma *in-dom-m-lookup-iff*: ‹*C* ∈# *dom-m N'* ⟷ *fmlookup N' C* ≠ *None*›
⟨*proof*⟩

lemma *in-dom-in-ran-m*[*simp*]: ‹*i* ∈# *dom-m N* ⟹ *the* (*fmlookup N i*) ∈# *ran-m N*›
⟨*proof*⟩

lemma *fmupd-same*[*simp*]:
‹*x1* ∈# *dom-m x1aa* ⟹ *fmupd x1* (*the* (*fmlookup x1aa x1*)) *x1aa* = *x1aa*›

⟨proof⟩

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

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

lemma *fset-fmdom-fmrestrict-set*:
‹*fset* (*fmdom* (*fmrestrict-set xs N*)) = *fset* (*fmdom N*) ∩ *xs*›
⟨proof⟩

lemma *dom-m-fmrestrict-set*: ‹*dom-m* (*fmrestrict-set* (*set xs*) *N*) = *mset xs* ∩# *dom-m N*›
⟨proof⟩

lemma *dom-m-fmrestrict-set'*: ‹*dom-m* (*fmrestrict-set xs N*) = *mset-set* (*xs* ∩ *set-mset* (*dom-m N*))›
⟨proof⟩

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

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

lemma *fm-member-split*: ‹*k* |∈| *fmdom N'* $\Longrightarrow$ ∃ *N''* *v*. *N'* = *fmupd k v N''* ∧ *the* (*fmlookup N' k*) = *v*
∧
k |∉| *fmdom N''*›
⟨proof⟩

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

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

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

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

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

lemma *fmrestrict-set-insert-notin-dom-m*[*simp*]:
‹*xa* ∉# *dom-m N* ⟹
fmrestrict-set (*insert xa l1*) *N* = *fmrestrict-set l1 N*›
⟨*proof*⟩

lemma *fmlookup-restrict-set-id*: ‹*fset* (*fmdom N*) ⊆ *A* ⟹ *fmrestrict-set A N* = *N*›
⟨*proof*⟩

lemma *fmlookup-restrict-set-id'*: ‹*set-mset* (*dom-m N*) ⊆ *A* ⟹ *fmrestrict-set A N* = *N*›
⟨*proof*⟩

lemma *ran-m-mapsto-upd*:
assumes
NC: ‹*C* ∈# *dom-m N*›
shows ‹*ran-m* (*fmupd C C' N*) =
add-mset C' (*remove1-mset* (*the* (*fmlookup N C*)) (*ran-m N*))›
⟨*proof*⟩

lemma *ran-m-mapsto-upd-notin*:
assumes *NC*: ‹*C* ∉# *dom-m N*›
shows ‹*ran-m* (*fmupd C C' N*) = *add-mset C'* (*ran-m N*)›
⟨*proof*⟩

lemma *ran-m-fmdrop*:
‹*C* ∈# *dom-m N* ⟹ *ran-m* (*fmdrop C N*) = *remove1-mset* (*the* (*fmlookup N C*)) (*ran-m N*)›
⟨*proof*⟩

lemma *ran-m-fmdrop-notin*:
‹*C* ∉# *dom-m N* ⟹ *ran-m* (*fmdrop C N*) = *ran-m N*›
⟨*proof*⟩

lemma *ran-m-fmdrop-If*:
‹*ran-m* (*fmdrop C N*) = (*if C* ∈# *dom-m N then remove1-mset* (*the* (*fmlookup N C*)) (*ran-m N*) *else ran-m 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-dom N* = *Max* (*set-mset* (*add-mset 0* (*dom-m N*)))›

definition *packed* **where**
‹*packed N* ⟷ *dom-m N* = *mset* [*1*..<*Suc* (*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* = {#} ⟹ *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* = *size* (*dom-m N*)›
⟨*proof*⟩

lemma *Max-dom-le*:
‹*L* ∈# *dom-m N* ⟹ *L* ≤ *Max-dom N*›
⟨*proof*⟩

lemma *remove1-mset-ge-Max-some*: ‹*a* > *Max-dom b* ⟹ *remove1-mset a* (*dom-m b*) = *dom-m b*›
⟨*proof*⟩

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

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

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

lemma *size-dom-m-Max-dom*:
‹*size* (*dom-m N*) ≤ *Suc* (*Max-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* ≤ *a then Max* {} *else b* − *1*)›
⟨*proof*⟩

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

lemma *packed0-fmud-Suc-Max-dom*: ‹*packed b* ⟹ *packed* (*fmupd* (*Suc* (*Max-dom b*)) *C b*)›
⟨*proof*⟩

lemma *ge-Max-dom-notin-dom-m*: ‹*a* > *Max-dom ao* ⟹ *a* ∉# *dom-m ao*›
⟨*proof*⟩

lemma *packed-in-dom-mI*: ‹*packed bc* ⟹ *j* ≤ *Max-dom bc* ⟹ *0* < *j* ⟹ *j* ∈# *dom-m bc*›
⟨*proof*⟩

lemma *mset-fset-empty-iff*: ‹*mset-fset a* = {#} ⟷ *a* = *fempty*›

⟨*proof*⟩

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

lemma *nat-power-div-base*:
fixes *k* :: *nat*
assumes *0* < *m* *0* < *k*
shows *k* ^ *m div k* = (*k*::*nat*) ^(*m* − *Suc 0*)
⟨*proof*⟩

lemma *eq-insertD*: ‹*A* = *insert a B* ⟹ *a* ∈ *A* ∧ *B* ⊆ *A*›
⟨*proof*⟩

lemma *length-list-ge2*: ‹*length S* ≥ *2* ⟷ (∃ *a b S'*. *S* = [*a*, *b*] @ *S'*)›
⟨*proof*⟩

1.4.1 Multiset version of Pow

This development was never useful in my own formalisation, but some people saw an interest in this or in things related to this (even if they discarded it eventually). Therefore, I finally decided to save the definition from my mailbox.

If anyone ever uses that and adds the concept to the AFP, please tell me such that I can delete it.

definition *Pow-mset* **where**
‹*Pow-mset A* = *fold-mset* (λ*a A*. (*A* + (*add-mset a*) '# *A*)) {#{#}#} *A*›

interpretation *pow-mset-commute*: *comp-fun-commute* ‹(λ*a A*. (*A* + (*add-mset a*) '# *A*))›
⟨*proof*⟩

lemma *Pow-mset-alt-def*:
Pow-mset (*mset A*) = *mset* '# *mset* (*subseqs A*)
⟨*proof*⟩

lemma *Pow-mset-empty*[*simp*]:
‹*Pow-mset* {#} = {#{#}#}›
⟨*proof*⟩

lemma *Pow-mset-add-mset*[*simp*]:
‹*Pow-mset* (*add-mset a A*) = *Pow-mset A* + (*add-mset a*) '# *Pow-mset A*›
⟨*proof*⟩

lemma *in-Pow-mset-iff*:
‹*A* ∈# *Pow-mset B* ⟷ *A* ⊆# *B*›
⟨*proof*⟩

lemma *size-Pow-mset*[*simp*]: ‹*size* (*Pow-mset A*) = *2*^(*size A*)›
⟨*proof*⟩

lemma *set-Pow-mset*:

‹*set-mset* (*Pow-mset* *A*) = {*B*. *B* ⊆# *A*}›
⟨*proof*⟩

Proof by Manuel Eberl on Zulip https://isabelle.zulipchat.com/#narrow/stream/238552-Beginner-Questions/topic/Cardinality.20of.20powerset.20of.20a.20multiset/near/220827959.

lemma *bij-betw-submultisets*:
card {*B*. *B* ⊆# *A*} = (∏*x*∈*set-mset* *A*. *count* *A* *x* + *1*)
⟨*proof*⟩

lemma *empty-in-Pow-mset*[*iff*]: ‹{#} ∈# *Pow-mset* *B*›
⟨*proof*⟩

lemma *full-in-Pow-mset*[*iff*]: ‹*B* ∈# *Pow-mset* *B*›
⟨*proof*⟩

lemma *Pow-mset-nempty*[*iff*]: ‹*Pow-mset* *B* ≠ {#}›
⟨*proof*⟩

lemma *Pow-mset-single-empty*[*iff*]: ‹*Pow-mset* *B* = {#{#}#} ⟷ *B* = {#}›
⟨*proof*⟩

lemma *Pow-mset-mono*: ‹*A* ⊆# *B* ⟹ *Pow-mset* *A* ⊆# *Pow-mset* *B*›
⟨*proof*⟩

Variants around head and last

definition *option-hd* :: ‹'*a* *list* ⇒ '*a* *option*› **where**
‹*option-hd* *xs* = (*if* *xs* = [] *then* *None* *else* *Some* (*hd* *xs*))›

lemma *option-hd-None-iff*[*iff*]: ‹*option-hd* *zs* = *None* ⟷ *zs* = []› ‹*None* = *option-hd* *zs* ⟷ *zs* = []›
⟨*proof*⟩

lemma *option-hd-Some-iff*[*iff*]: ‹*option-hd* *zs* = *Some* *y* ⟷ (*zs* ≠ [] ∧ *y* = *hd* *zs*)›
‹*Some* *y* = *option-hd* *zs* ⟷ (*zs* ≠ [] ∧ *y* = *hd* *zs*)›
⟨*proof*⟩

lemma *option-hd-Some-hd*[*simp*]: ‹*zs* ≠ [] ⟹ *option-hd* *zs* = *Some* (*hd* *zs*)›
⟨*proof*⟩

lemma *option-hd-Nil*[*simp*]: ‹*option-hd* [] = *None*›
⟨*proof*⟩

definition *option-last* **where**
‹*option-last* *l* = (*if* *l* = [] *then* *None* *else* *Some* (*last* *l*))›

lemma
option-last-None-iff[*iff*]: ‹*option-last* *l* = *None* ⟷ *l* = []› ‹*None* = *option-last* *l* ⟷ *l* = []› **and**
option-last-Some-iff[*iff*]:
‹*option-last* *l* = *Some* *a* ⟷ *l* ≠ [] ∧ *a* = *last* *l*›
‹*Some* *a* = *option-last* *l* ⟷ *l* ≠ [] ∧ *a* = *last* *l*›
⟨*proof*⟩

lemma *option-last-Some*[*simp*]: ‹*l* ≠ [] ⟹ *option-last* *l* = *Some* (*last* *l*)›
⟨*proof*⟩

lemma *option-last-Nil*[*simp*]: ‹*option-last* [] = *None*›

⟨*proof*⟩

lemma *option-last-remove1-not-last*:
‹$x \neq$ *last xs* $\Longrightarrow$ *option-last xs* = *option-last* (*remove1 x xs*)›
⟨*proof*⟩

lemma *option-hd-rev*: ‹*option-hd* (*rev xs*) = *option-last xs*›
⟨*proof*⟩

lemma *map-option-option-last*:
‹*map-option f* (*option-last xs*) = *option-last* (*map f xs*)›
⟨*proof*⟩

end