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*:
‹$r^{++}\ a\ b \Longrightarrow r \le s \Longrightarrow s^{++}\ a\ b$›
using *rtranclp-mono* **by** (*auto dest*!: *tranclpD intro*: *rtranclp-into-tranclp2*)

lemma *tranclp-mono*:
assumes *mono*: ‹$r \le s$›
shows ‹$r^{++} \le s^{++}$›
using *rtranclp-mono*[*OF mono*] *mono* **by** (*auto dest*!: *tranclpD intro*: *rtranclp-into-tranclp2*)

lemma *tranclp-idemp-rel*:
‹$R^{++++}\ a\ b \longleftrightarrow R^{++}\ a\ b$›
apply (*rule iffI*)
prefer *2* **apply** *blast*
by (*induction rule*: *tranclp-induct*) *auto*

Equivalent of the theorem *rtranclp-idemp*

lemma *trancl-idemp*: ‹$(r^+)^+ = r^+$›
by *simp*

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

This theorem already exists as theroem *Nitpick.rtranclp-unfold* (and sledgehammer uses it), but it makes sense to duplicate it, because it is unclear how stable the lemmas in the `~~/src/HOL/Nitpick.thy` theory are.

lemma *rtranclp-unfold*: ‹*rtranclp r a b* ⟷ (*a* = *b* ∨ *tranclp r a b*)›
  by (*meson rtranclp.simps rtranclpD tranclp-into-rtranclp*)

lemma *tranclp-unfold-end*: ‹*tranclp r a b* ⟷ (∃ *a'*. *rtranclp r a a'* ∧ *r a' b*)›
  by (*metis rtranclp.rtrancl-refl rtranclp-into-tranclp1 tranclp.cases tranclp-into-rtranclp*)

Near duplicate of theorem *tranclpD*:

lemma *tranclp-unfold-begin*: ‹*tranclp r a b* ⟷ (∃ *a'*. *r a a'* ∧ *rtranclp r a' b*)›
  by (*meson rtranclp-into-tranclp2 tranclpD*)

lemma *trancl-set-tranclp*: ‹(*a*, *b*) ∈ {(*b*,*a*). *P a b*}$^+$ ⟷ P^{++} *b a*›
  apply (*rule iffI*)
    apply (*induction rule*: *trancl-induct*; *simp*)
  apply (*induction rule*: *tranclp-induct*; *auto simp*: *trancl-into-trancl2*)
  done

lemma *tranclp-rtranclp-rtranclp-rel*: ‹R^{++**} *a b* ⟷ R^{**} *a b*›
  by (*simp add*: *rtranclp-unfold*)

lemma *tranclp-rtranclp-rtranclp*[*simp*]: ‹$R^{++**} = R^{**}$›
  by (*fastforce simp*: *rtranclp-unfold*)

lemma *rtranclp-exists-last-with-prop*:
  assumes ‹*R x z*› **and** ‹R^{**} *z z'*› **and** ‹*P x z*›
  shows ‹∃ *y y'*. R^{**} *x y* ∧ *R y y'* ∧ *P y y'* ∧ (λ*a b*. *R a b* ∧ ¬*P a b*)** *y' z'*›
  using *assms*(*2*,*1*,*3*)
proof *induction*
  case *base*
  then show *?case* **by** *auto*
next
  case (*step z' z''*) **note** *z* = *this*(*2*) **and** *IH* = *this*(*3*)[*OF this*(*4*−*5*)]
  show *?case*
    apply (*cases* ‹*P z' z''*›)
      apply (*rule exI*[*of - z'*], *rule exI*[*of - z''*])
      using *z assms*(*1*) *step.hyps*(*1*) *step.prems*(*2*) **apply** (*auto*; *fail*)[*1*]
    using *IH z* **by** (*fastforce intro*: *rtranclp.rtrancl-into-rtrancl*)
qed

lemma *rtranclp-and-rtranclp-left*: ‹(λ *a b*. *P a b* ∧ *Q a b*)** *S T* ⟹ P^{**} *S T*›
  by (*induction rule*: *rtranclp-induct*) *auto*

1.1.2 Full Transitions

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

abbreviation (*input*) *no-step* :: (*'a* ⇒ *'b* ⇒ *bool*) ⇒ *'a* ⇒ *bool* **where**
no-step step S ≡ ∀ *S'*. ¬*step S S'*

definition *full1* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *'a* ⇒ *'a* ⇒ *bool* **where**
full1 transf = (λ*S S'*. *tranclp transf S S'* ∧ *no-step transf S'*)

definition *full*:: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *'a* ⇒ *'a* ⇒ *bool* **where**

full transf = ($\lambda S\ S'$. *rtranclp transf S S'* ∧ *no-step transf S'*)

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

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

Some Properties **lemma** *rtranclp-full1I*:
‹R^{**} *a b* ⟹ *full1 R b c* ⟹ *full1 R a c*›
unfolding *full1-def* **by** *auto*

lemma *tranclp-full1I*:
‹R^{++} *a b* ⟹ *full1 R b c* ⟹ *full1 R a c*›
unfolding *full1-def* **by** *auto*

lemma *rtranclp-fullI*:
‹R^{**} *a b* ⟹ *full R b c* ⟹ *full R a c*›
unfolding *full-def* **by** *auto*

lemma *tranclp-full-full1I*:
‹R^{++} *a b* ⟹ *full R b c* ⟹ *full1 R a c*›
unfolding *full-def full1-def* **by** *auto*

lemma *full-fullI*:
‹*R a b* ⟹ *full R b c* ⟹ *full1 R a c*›
unfolding *full-def full1-def* **by** *auto*

lemma *full-unfold*:
‹*full r S S'* ⟷ ((*S* = *S'* ∧ *no-step r S'*) ∨ *full1 r S S'*)›
unfolding *full-def full1-def* **by** (*auto simp add*: *rtranclp-unfold*)

lemma *full1-is-full*[*intro*]: ‹*full1 R S T* ⟹ *full R S T*›
by (*simp add*: *full-unfold*)

lemma *not-full1-rtranclp-relation*: ¬*full1* R^{**} *a b*
by (*auto simp*: *full1-def*)

lemma *not-full-rtranclp-relation*: ¬*full* R^{**} *a b*
by (*auto simp*: *full-def*)

lemma *full1-tranclp-relation-full*:
‹*full1* R^{++} *a b* ⟷ *full1 R a b*›
by (*metis converse-tranclpE full1-def reflclp-tranclp rtranclpD rtranclp-idemp rtranclp-reflclp tranclp.r-into-trancl tranclp-into-rtranclp*)

lemma *full-tranclp-relation-full*:
‹*full* R^{++} *a b* ⟷ *full R a b*›
by (*metis full-unfold full1-tranclp-relation-full tranclp.r-into-trancl tranclpD*)

lemma *tranclp-full1-full1*:
‹(*full1 R*)$^{++}$ *a b* ⟷ *full1 R a b*›
by (*metis* (*mono-tags*) *full1-def predicate2I tranclp.r-into-trancl tranclp-idemp tranclp-mono-explicit tranclp-unfold-end*)

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

```
  unfolding rtranclp-unfold tranclp-full1-full1 by simp

lemma no-step-full-iff-eq:
  ‹no-step R S ⟹ full R S T ⟷ S = T›
  unfolding full-def
  by (meson rtranclp.rtrancl-refl rtranclpD tranclpD)
```

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 (rule ccontr)
  assume ‹¬ ?thesis›
  then have H: ‹⋀b. ¬ R** a b ∨ ¬no-step R b›
    by blast
  define F where ‹F = rec-nat a (λi b. SOME c. R b c)›
  have [simp]: ‹F 0 = a›
    unfolding F-def by auto
  have [simp]: ‹⋀i. F (Suc i) = (SOME b. R (F i) b)›
    unfolding F-def by simp
  { fix i
    have ‹∀ j<i. R (F j) (F (Suc j))›
    proof (induction i)
      case 0
      then show ?case by auto
    next
      case (Suc i)
      then have ‹R** a (F i)›
        by (induction i) auto
      then have ‹R (F i) (SOME b. R (F i) b)›
        using H by (simp add: someI-ex)
      then have ‹∀ j < Suc i. R (F j) (F (Suc j))›
        using H Suc by (simp add: less-Suc-eq)
      then show ?case by fast
    qed
  }
  then have ‹∀ j. R (F j) (F (Suc j))› by blast
  then show False
    using wf unfolding wfP-def wf-iff-no-infinite-down-chain by blast
qed

lemma wf-exists-normal-form-full:
  assumes wf: ‹wf {(x, y). R y x}›
  shows ‹∃ b. full R a b›
  using wf-exists-normal-form[OF assms] unfolding full-def by blast
```

1.1.4 More Well-Foundedness

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

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

```
lemma wf-if-measure-in-wf:
  ‹wf R ⟹ (⋀a b. (a, b) ∈ S ⟹ (ν a, ν b)∈R) ⟹ wf S›
```

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

lemma *wfP-if-measure*: **fixes** *f* :: ‹′*a* ⇒ *nat*›
shows ‹(⋀*x y*. *P x* ⟹ *g x y* ⟹ *f y* < *f x*) ⟹ *wf* {(*y*,*x*). *P x* ∧ *g x y*}›
apply (*insert wf-measure*[*of f*])
apply (*simp only*: *measure-def inv-image-def less-than-def less-eq*)
apply (*erule wf-subset*)
apply *auto*
done

lemma *wf-if-measure-f*:
assumes ‹*wf r*›
shows ‹*wf* {(*b*, *a*). (*f b*, *f a*) ∈ *r*}›
using *assms* **by** (*metis inv-image-def wf-inv-image*)

lemma *wf-wf-if-measure′*:
assumes ‹*wf r*› **and** *H*: ‹⋀*x y*. *P x* ⟹ *g x y* ⟹ (*f y*, *f x*) ∈ *r*›
shows ‹*wf* {(*y*,*x*). *P x* ∧ *g x y*}›
proof −
have ‹*wf* {(*b*, *a*). (*f b*, *f a*) ∈ *r*}› **using** *assms*(*1*) *wf-if-measure-f* **by** *auto*
then have ‹*wf* {(*b*, *a*). *P a* ∧ *g a b* ∧ (*f b*, *f a*) ∈ *r*}›
using *wf-subset*[*of* - ‹{(*b*, *a*). *P a* ∧ *g a b* ∧ (*f b*, *f a*) ∈ *r*}›] **by** *auto*
moreover have ‹{(*b*, *a*). *P a* ∧ *g a b* ∧ (*f b*, *f a*) ∈ *r*} ⊆ {(*b*, *a*). (*f b*, *f a*) ∈ *r*}› **by** *auto*
moreover have ‹{(*b*, *a*). *P a* ∧ *g a b* ∧ (*f b*, *f a*) ∈ *r*} = {(*b*, *a*). *P a* ∧ *g a b*}› **using** *H* **by** *auto*
ultimately show *?thesis* **using** *wf-subset* **by** *simp*
qed

lemma *wf-lex-less*: ‹*wf* (*lex less-than*)›
by (*auto simp*: *wf-less*)

lemma *wfP-if-measure2*: **fixes** *f* :: ‹′*a* ⇒ *nat*›
shows ‹(⋀*x y*. *P x y* ⟹ *g x y* ⟹ *f x* < *f y*) ⟹ *wf* {(*x*,*y*). *P x y* ∧ *g x y*}›
apply (*insert wf-measure*[*of f*])
apply (*simp only*: *measure-def inv-image-def less-than-def less-eq*)
apply (*erule wf-subset*)
apply *auto*
done

lemma *lexord-on-finite-set-is-wf*:
assumes
P-finite: ‹⋀*U*. *P U* ⟶ *U* ∈ *A*› **and**
finite: ‹*finite A*› **and**
wf: ‹*wf R*› **and**
trans: ‹*trans R*›
shows ‹*wf* {(*T*, *S*). (*P S* ∧ *P T*) ∧ (*T*, *S*) ∈ *lexord R*}›
proof (*rule wfP-if-measure2*)
fix *T S*
assume *P*: ‹*P S* ∧ *P T*› **and**
s-le-t: ‹(*T*, *S*) ∈ *lexord R*›
let *?f* = ‹λ*S*. {*U*. (*U*, *S*) ∈ *lexord R* ∧ *P U* ∧ *P S*}›
have ‹*?f T* ⊆ *?f S*›
using *s-le-t P lexord-trans trans* **by** *auto*
moreover have ‹*T* ∈ *?f S*›
using *s-le-t P* **by** *auto*
moreover have ‹*T* ∉ *?f T*›
using *s-le-t* **by** (*auto simp add*: *lexord-irreflexive local.wf*)

```
  ultimately have ‹{U. (U, T) ∈ lexord R ∧ P U ∧ P T} ⊂ {U. (U, S) ∈ lexord R ∧ P U ∧ P S}›
    by auto
  moreover have ‹finite {U. (U, S) ∈ lexord R ∧ P U ∧ P S}›
    using finite by (metis (no-types, lifting) P-finite finite-subset mem-Collect-eq subsetI)
  ultimately show ‹card (?f T) < card (?f S)› by (simp add: psubset-card-mono)
qed


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

lemma wf-snd-wf-pair:
  assumes ‹wf {(M', M). R M' M} ›
  shows ‹wf {((M', N'), (M, N)). R N' N}›
proof –
  have wf: ‹wf {((M', N'), (M, N)). R M' M}›
    using assms wf-fst-wf-pair by auto
  then have wf: ‹⋀P. (∀x. (∀y. (y, x) ∈ {((M', N'), M, N). R M' M} ⟶ P y) ⟶ P x) ⟹ All P›
    unfolding wf-def by auto
  show ?thesis
    unfolding wf-def
    proof (intro allI impI)
      fix P :: ‹'c × 'a ⇒ bool› and x :: ‹'c × 'a›
      assume H: ‹∀x. (∀y. (y, x) ∈ {((M', N'), M, y). R N' y} ⟶ P y) ⟶ P x›
      obtain a b where x: ‹x = (a, b)› by (cases x)
      have P: ‹P x = (P ∘ (λ(a, b). (b, a))) (b, a)›
        unfolding x by auto
      show ‹P x›
        using wf[of ‹P o (λ(a, b). (b, a))›] apply rule
          using H apply simp
        unfolding P by blast
    qed
qed

lemma wf-if-measure-f-notation2:
  assumes ‹wf r›
  shows ‹wf {(b, h a)|b a. (f b, f (h a)) ∈ r}›
  apply (rule wf-subset)
  using wf-if-measure-f[OF assms, of f] by auto

lemma wf-wf-if-measure'-notation2:
  assumes ‹wf r› and H: ‹⋀x y. P x ⟹ g x y ⟹ (f y, f (h x)) ∈ r›
  shows ‹wf {(y,h x)| y x. P x ∧ g x y}›
proof –
  have ‹wf {(b, h a)|b a. (f b, f (h a)) ∈ r}› using assms(1) wf-if-measure-f-notation2 by auto
  then have ‹wf {(b, h a)|b a. P a ∧ g a b ∧ (f b, f (h a)) ∈ r}›
    using wf-subset[of - ‹{(b, h a)| b a. P a ∧ g a b ∧ (f b, f (h a)) ∈ r}›] by auto
  moreover have ‹{(b, h a)|b a. P a ∧ g a b ∧ (f b, f (h a)) ∈ r}
    ⊆ {(b, h a)|b a. (f b, f (h a)) ∈ r}› by auto
```

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

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

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

lemma *lexn-number-of-transition*:
  assumes
    le: ‹$\bigwedge i.\ i < n \Longrightarrow ((f\ (Suc\ i)), (f\ i)) \in lexn\ less\text{-}than\ m$› **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 ‹$n < k\ \widehat{}\ Suc\ m$›
proof −
  define *r* **where**
    ‹$r\ x = zip\ x\ (map\ (\lambda i.\ k\ \widehat{}\ (length\ x - i))\ [0..<length\ x])$› **for** *x* :: ‹*nat list*›

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

  have [*simp*]: ‹$r\ [] = []$› ‹$s\ [] = 0$›
    by (*auto simp*: *r-def s-def*)

  have *upt′*: ‹$m > 0 \Longrightarrow [0..< m] = 0\ \#\ map\ Suc\ [0..< m - 1]$› **for** *m*
    by (*auto simp*: *map-Suc-upt upt-conv-Cons*)

  have *upt″*: ‹$m > 0 \Longrightarrow [0..< m] = [0..< m - 1]\ @\ [m-1]$› **for** *m*
    by (*cases m*) (*auto simp*:)

  have *Cons*: ‹$r\ (x\ \#\ xs) = (x, k\widehat{}(Suc\ (length\ xs)))\ \#\ (r\ xs)$› **for** *x xs*

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

have [simp]: ‹s (ab # xs) = fst ab ∗ snd ab + s xs› for ab xs
  unfolding s-def by (cases ab) auto

have le2: ‹(∀ a ∈ set b. a < k) ⟹ (k ^(Suc (length b))) > s ((r b))› for b
  apply (induction b arbitrary: f)
  using k apply (auto simp: Cons)
  apply (rule order.strict-trans1)
  apply (rule-tac j = ‹(k − 1) ∗ k ∗k ^ length b› in Nat.add-le-mono1)
  subgoal for a b
    by auto
  apply (rule order.strict-trans2)
  apply (rule-tac b = ‹(k − 1) ∗ k ∗ k ^ length b› and d = ‹(k ∗ k ^ length b)› in add-le-less-mono)
  apply (auto simp: mult.assoc comm-semiring-1-class.semiring-normalization-rules(2))
  done

have ‹s (r (f (Suc i))) < s (r (f i))› if ‹i < n› for i
proof −
  have i-n: ‹Suc i ≤ n› ‹i ≤ n›
    using that by auto
  have length: ‹length (f i) = m›  ‹length (f (Suc i)) = m›
   using le[OF that] by (auto dest: lexn-length)
  define xs and ys where ‹xs = f i› and ‹ys = f (Suc i)›

  show ?thesis
    using le[OF that] upper[OF i-n(2)] upper[OF i-n(1)] length Cons
    unfolding xs-def[symmetric] ys-def[symmetric]
  proof (induction m arbitrary: xs ys)
    case 0 then show ?case by auto
  next
    case (Suc m) note IH = this(1) and H = this(2) and p = this(3−)
    have IH: ‹(tl ys, tl xs) ∈ lexn less-than m ⟹ s (r (tl ys)) < s (r (tl xs))›
      apply (rule IH)
      subgoal by auto
      subgoal for i using p(1)[of ‹Suc i›] p by (cases xs; auto)
      subgoal for i using p(2)[of ‹Suc i›] p by (cases ys; auto)
      subgoal using p by (cases xs) auto
      subgoal using p by auto
      subgoal using p by auto
      done
   have ‹s (r (tl ys)) < k ^(Suc (length (tl ys)))›
     apply (rule le2)
     unfolding all-set-conv-all-nth
     using p by (simp add: nth-tl)
   then have ‹ab ∗ (k ∗ k ^ length (tl ys)) + s (r (tl ys)) <
          ab ∗ (k ∗ k ^ length (tl ys)) + (k ∗ k ^ length (tl ys))› for ab
     by auto
   also have ‹. . . ab  ≤ (ab + 1) ∗ (k ∗ k ^ length (tl ys))› for ab
     by auto
   finally have less: ‹ab < ac ⟹ ab ∗ (k ∗ k ^ length (tl ys)) + s (r (tl ys)) <
                          ac ∗ (k ∗ k ^ length (tl ys))› for ab ac
```

```
    proof -
      assume a1: ⋀ab. ab * (k * k ^ length (tl ys)) + s (r (tl ys)) <
              (ab + 1) * (k * k ^ length (tl ys))
      assume ab < ac
      then have ¬ ac * (k * k ^ length (tl ys)) < (ab + 1) * (k * k ^ length (tl ys))
        by (metis (no-types) One-nat-def Suc-leI add.right-neutral add-Suc-right
           mult-less-cancel2 not-less)
      then show ?thesis
        using a1 by (meson le-less-trans not-less)
    qed

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

end
theory WB-List-More
  imports 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**
‹$\bigwedge n.\ (\forall m < Suc\ n.\ P\ m) \Longrightarrow P\ (Suc\ n)$›
shows ‹*P n*›
apply (*induction rule*: *nat-less-induct*)
by (*rename-tac n*, *case-tac n*) (*auto intro*: *assms*)

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

lemma *if-0-1-ge-0*[*simp*]:
‹$0 < (if\ P\ then\ a\ else\ (0::nat)) \longleftrightarrow P \wedge 0 < a$›
by *auto*

lemma *bex-lessI*: $P\ j \Longrightarrow j < n \Longrightarrow \exists j<n.\ P\ j$
by *auto*

lemma *bex-gtI*: $P\ j \Longrightarrow j > n \Longrightarrow \exists j>n.\ P\ j$
by *auto*

lemma *bex-geI*: $P\ j \Longrightarrow j \geq n \Longrightarrow \exists j\geq n.\ P\ j$
by *auto*

lemma *bex-leI*: $P\ j \Longrightarrow j \leq n \Longrightarrow \exists j\leq n.\ P\ j$
by *auto*

Bounded function have not yet been defined in Isabelle.

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

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

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 (*rule ccontr*)
assume *H*: ‹¬ *?thesis*›
have ‹$finite\ \{f\ n|n.\ n \leq n_0\}$›
by *auto*
have ‹$\bigwedge n.\ f\ n \leq Max\ (\{f\ n|n.\ n \leq n_0\} \cup \{m\})$›
apply (*case-tac* ‹$n \leq n_0$›)
apply (*metis* (*mono-tags*, *lifting*) *Max-ge Un-insert-right* ‹$finite\ \{f\ n\ |n.\ n \leq n_0\}$›
finite-insert insertCI mem-Collect-eq sup-bot.right-neutral)
by (*metis* (*no-types*, *lifting*) *H Max-less-iff Un-insert-right* ‹$finite\ \{f\ n\ |n.\ n \leq n_0\}$›

finite-insert insertI1 insert-not-empty leI sup-bot.right-neutral)
then show *False*
using *unbound* **unfolding** *bounded-def* **by** *auto*
qed

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

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

lemma *bounded-const-add*:
fixes k :: *nat* **and** f :: ‹*nat* ⇒ *nat*›
assumes ‹$k > 0$›
shows ‹*bounded f* ⟷ *bounded* ($\lambda i.\ k + f\ i$)›
unfolding *bounded-def* **apply** (*rule iffI*)
using *nat-add-left-cancel-le* **apply** *blast*
using *add-leE* **by** *blast*

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

lemma *bounded-finite-linorder*:
fixes f :: ‹*'a::finite* ⇒ *'b* :: {*linorder*}›
shows ‹*bounded f*›
proof −
have ‹*finite* (*f* ‘ *UNIV*)›
by *simp*
then have ‹⋀*x*. *f x* ≤ *Max* (*f* ‘ *UNIV*)›
by (*auto intro*: *Max-ge*)
then show *?thesis*
unfolding *bounded-def* **by** *blast*
qed

1.3 More Lists

1.3.1 set, nth, tl

lemma *ex-geI*: ‹$P\ n \Longrightarrow n \geq m \Longrightarrow \exists n{\geq}m.\ P\ n$›
by *auto*

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

lemma *nth-in-set-tl*: ‹$i > 0 \Longrightarrow i <$ *length xs* ⟹ *xs* ! $i \in$ *set* (*tl xs*)›
by (*cases xs*) *auto*

lemma *tl-drop-def*: ‹*tl N* = *drop 1 N*›
by (*cases N*) *auto*

lemma *in-set-remove1D*:
‹$a \in set\ (remove1\ x\ xs) \Longrightarrow a \in set\ xs$›
by (*meson notin-set-remove1*)

lemma *take-length-takeWhile-eq-takeWhile*:
‹$take\ (length\ (takeWhile\ P\ xs))\ xs = takeWhile\ P\ xs$›
by (*induction xs*) *auto*

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

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

lemma *in-set-image-subsetD*: ‹ $f\ `\ A \subseteq B \Longrightarrow x \in A \Longrightarrow f\ x \in B$›
by *blast*

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

lemma *hd-list-update-If*:
‹$outl' \neq [] \Longrightarrow hd\ (outl'[i := w]) = (if\ i = 0\ then\ w\ else\ hd\ outl')$›
by (*cases outl'*) (*auto split*: *nat.splits*)

lemma *list-update-id'*:
‹$x = xs\ !\ i \Longrightarrow xs[i := x] = xs$›
by *auto*

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

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

lemma *butlast-Nil-iff*: ‹$butlast\ xs = [] \longleftrightarrow length\ xs = 1 \vee length\ xs = 0$›
by (*cases xs*) *auto*

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

lemma *Set-insert-diff-remove*: ‹$B - insert\ a\ A = Set.remove\ a\ (B - A)$›
by *auto*

lemma *Set-remove-insert*: ‹$a \notin A' \Longrightarrow Set.remove\ a\ (insert\ a\ A') = A'$›
by (*auto simp*: *Set.remove-def*)

lemma *diff-eq-insertD*:
‹$B - A = insert\ a\ A' \Longrightarrow a \in B$›
by *auto*

lemma *in-set-tlD*: ‹$x \in set\ (tl\ xs) \Longrightarrow x \in set\ xs$›
by (*cases xs*) *auto*

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*: ‹$xs = it\ @\ x\ \#\ \sigma \Longrightarrow x \in set\ xs$›

```
  by auto

lemma Collect-eq-comp′: ‹ {(x, y). P x y} O {(c, a). c = f a} = {(x, a). P x (f a)}›
  by auto

lemma (in −) filter-disj-eq:
  ‹{x ∈ A. P x ∨ Q x} = {x ∈ A. P x} ∪ {x ∈ A. Q x}›
  by auto


lemma zip-cong:
  ‹(⋀i. i < min (length xs) (length ys) ⟹ (xs ! i, ys ! i) = (xs′ ! i, ys′ ! i)) ⟹
     length xs = length xs′ ⟹ length ys = length ys′ ⟹ zip xs ys = zip xs′ ys′›
proof (induction xs arbitrary: xs′ ys′ ys)
  case Nil
  then show ?case by auto
next
  case (Cons x xs xs′ ys′ ys) note IH = this(1) and eq = this(2) and p = this(3−)
thm IH
  have ‹zip xs (tl ys) = zip (tl xs′) (tl ys′)› for i
    apply (rule IH)
    subgoal for i
      using p eq[of ‹Suc i›] by (auto simp: nth-tl)
    subgoal using p by auto
    subgoal using p by auto
    done
  moreover have ‹hd xs′ = x› ‹hd ys = hd ys′› if ‹ys ≠ []›
    using eq[of 0] that p[symmetric] apply (auto simp: hd-conv-nth)
    apply (subst hd-conv-nth)
    apply auto
    apply (subst hd-conv-nth)
    apply auto
    done
  ultimately show ?case
    using p by (cases xs′; cases ys′; cases ys)
      auto
qed

lemma zip-cong2:
  ‹(⋀i. i < min (length xs) (length ys) ⟹ (xs ! i, ys ! i) = (xs′ ! i, ys′ ! i)) ⟹
     length xs = length xs′ ⟹ length ys ≤ length ys′ ⟹ length ys ≥ length xs ⟹
     zip xs ys = zip xs′ ys′›
proof (induction xs arbitrary: xs′ ys′ ys)
  case Nil
  then show ?case by auto
next
  case (Cons x xs xs′ ys′ ys) note IH = this(1) and eq = this(2) and p = this(3−)
  have ‹zip xs (tl ys) = zip (tl xs′) (tl ys′)› for i
    apply (rule IH)
    subgoal for i
      using p eq[of ‹Suc i›] by (auto simp: nth-tl)
    subgoal using p by auto
    subgoal using p by auto
    subgoal using p by auto
    done
  moreover have ‹hd xs′ = x› ‹hd ys = hd ys′› if ‹ys ≠ []›
```

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

1.3.2 List Updates

lemma *tl-update-swap*:
‹$i \geq 1 \Longrightarrow tl\ (N[i := C]) = (tl\ N)[i-1 := C]$›
by (*auto simp*: *drop-Suc*[*of 0*, *symmetric*, *simplified*] *drop-update-swap*)

lemma *tl-update-0*[*simp*]: ‹$tl\ (N[0 := x]) = tl\ N$›
by (*cases N*) *auto*

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

1.3.3 Take and drop

lemma *take-2-if*:
‹$take\ 2\ C = (if\ C = []\ then\ []\ else\ if\ length\ C = 1\ then\ [hd\ C]\ else\ [C!0,\ C!1])$›
by (*cases C*; *cases* ‹*tl C*›) *auto*

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

lemma *in-set-dropI*:
‹$m < length\ xs \Longrightarrow m \geq n \Longrightarrow xs\ !\ m \in set\ (drop\ n\ xs)$›
unfolding *in-set-conv-nth*
by (*rule exI*[*of -* ‹$m - n$›]) *auto*

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

Taken from the Word library.

```
lemma atd-lem: ‹take n xs = t ⟹ drop n xs = d ⟹ xs = t @ d›
  by (auto intro: append-take-drop-id [symmetric])

lemma drop-take-drop-drop:
  ‹j ≥ i ⟹ drop i xs = take (j − i) (drop i xs) @ drop j xs›
  apply (induction ‹j − i› arbitrary: j i)
  subgoal by auto
  subgoal by (auto simp add: atd-lem)
  done

lemma in-set-conv-iff:
  ‹x ∈ set (take n xs) ⟷ (∃ i < n. i < length xs ∧ xs ! i = x)›
  apply (induction n)
  subgoal by auto
  subgoal for n
    apply (cases ‹Suc n < length xs›)
    subgoal by (auto simp: take-Suc-conv-app-nth less-Suc-eq dest: in-set-takeD)
    subgoal
      apply (cases ‹n < length xs›)
      subgoal
        apply (auto simp: in-set-conv-nth)
        by (rule-tac x=i in exI; auto; fail)+
      subgoal
        apply (auto simp: take-Suc-conv-app-nth dest: in-set-takeD)
        by (rule-tac x=i in exI; auto; fail)+
      done
    done
  done

lemma distinct-in-set-take-iff:
  ‹distinct D ⟹ b < length D ⟹ D ! b ∈ set (take a D) ⟷ b < a›
  apply (induction a arbitrary: b)
  subgoal by simp
  subgoal for a
    by (cases ‹Suc a < length D›)
      (auto simp: take-Suc-conv-app-nth nth-eq-iff-index-eq)
  done

lemma in-set-distinct-take-drop-iff:
  assumes
    ‹distinct D› and
    ‹b < length D›
  shows ‹D ! b ∈ set (take (a − init) (drop init D)) ⟷ (init ≤ b ∧ b < a)›
  using assms apply (auto 5 5 simp: distinct-in-set-take-iff in-set-conv-iff
      nth-eq-iff-index-eq dest: in-set-takeD)
  by (metis add-diff-cancel-left′ diff-less-mono le-iff-add)
```

1.3.4 Replicate

```
lemma list-eq-replicate-iff-nempty:
  ‹n > 0 ⟹ xs = replicate n x ⟷ n = length xs ∧ set xs = {x}›
  by (metis length-replicate neq0-conv replicate-length-same set-replicate singletonD)

lemma list-eq-replicate-iff:
  ‹xs = replicate n x ⟷ (n = 0 ∧ xs = []) ∨ (n = length xs ∧ set xs = {x})›
```

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

1.3.5 List intervals (*upt*)

The simplification rules are not very handy, because theorem *upt.simps* (*2*) (i.e. [*?i..<Suc ?j*] = (*if ?i* ≤ *?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: ‹¬i ≤ j ⟹ [i..<Suc j] = []›
  by auto

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

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

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

```
lemma take-upt-bound-minus[simp]:
  assumes ‹i ≤ n − m›
  shows ‹take i [m..<n] = [m ..<m+i]›
  using assms by (induction i) auto

lemma append-cons-eq-upt:
  assumes ‹A @ B = [m..<n]›
  shows ‹A = [m ..<m+length A]› and ‹B = [m + length A..<n]›
proof −
  have ‹take (length A) (A @ B) = A› by auto
  moreover {
    have ‹length A ≤ n − m› using assms linear calculation by fastforce
    then have ‹take (length A) [m..<n] = [m ..<m+length A]› by auto }
  ultimately show ‹A = [m ..<m+length A]› using assms by auto
  show ‹B = [m + length A..<n]› using assms by (metis append-eq-conv-conj drop-upt)
qed
```

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

```
lemma ‹A @ B = [m..< n] ⟷ A = [m ..<m+length A] ∧ B = [m + length A..<n]›
oops
```

A more restrictive version holds:

```
lemma ‹B ≠ [] ⟹ A @ B = [m..< n] ⟷ A = [m ..<m+length A] ∧ B = [m + length A..<n]›
  (is ‹?P ⟹ ?A = ?B›)
proof
  assume ?A then show ?B by (auto simp add: append-cons-eq-upt)
next
  assume ?P and ?B
  then show ?A using append-eq-conv-conj by fastforce
qed

lemma append-cons-eq-upt-length-i:
  assumes ‹A @ i # B = [m..<n]›
  shows ‹A = [m ..<i]›
proof −
  have ‹A = [m ..< m + length A]› using assms append-cons-eq-upt by auto
  have ‹(A @ i # B) ! (length A) = i› by auto
```

```
  moreover have ‹n − m = length (A @ i # B)›
    using assms length-upt by presburger
  then have ‹[m..<n] ! (length A) = m + length A› by simp
  ultimately have ‹i = m + length A› using assms by auto
  then show ?thesis using ‹A = [m ..< m + length A]› by auto
qed

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

lemma append-cons-eq-upt-length-i-end:
  assumes ‹A @ i # B = [m..<n]›
  shows ‹B = [Suc i ..<n]›
proof −
  have ‹B = [Suc m + length A..<n]› using assms append-cons-eq-upt[of ‹A @ [i]› B m n] by auto
  have ‹(A @ i # B) ! (length A) = i› by auto
  moreover have ‹n − m = length (A @ i # B)›
    using assms length-upt by auto
  then have ‹[m..<n]! (length A) = m + length A› by simp
  ultimately have ‹i = m + length A› using assms by auto
  then show ?thesis using ‹B = [Suc m + length A..<n]› by auto
qed

lemma Max-n-upt: ‹Max (insert 0 {Suc 0..<n}) = n − Suc 0›
proof (induct n)
  case 0
  then show ?case by simp
next
  case (Suc n) note IH = this
  have i: ‹insert 0 {Suc 0..<Suc n} = insert 0 {Suc 0..< n} ∪ {n}› by auto
  show ?case using IH unfolding i by auto
qed

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

lemma nths-upt-upto-Suc: ‹aa < length xs ⟹ nths xs {0..<Suc aa} = nths xs {0..<aa} @ [xs ! aa]›
```

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

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

```
lemma length-list-Suc-0:
  ‹length W = Suc 0 ⟷ (∃ L. W = [L])›
  apply (cases W)
    apply (simp; fail)
  apply (rename-tac a W', case-tac W')
  apply auto
  done

lemma length-list-2: ‹length S = 2 ⟷ (∃ a b. S = [a, b])›
  apply (cases S)
   apply (simp; fail)
  apply (rename-tac a S')
  apply (case-tac S')
  by simp-all

lemma finite-bounded-list:
  fixes b :: nat
  shows ‹finite {xs. length xs < s ∧ (∀ i< length xs. xs ! i < b)}› (is ‹finite (?S s)›)
proof −
  have H: ‹finite {xs. set xs ⊆ {0..<b} ∧ length xs ≤ s}›
    by (rule finite-lists-length-le[of ‹{0..<b}› ‹s›]) auto
  show ?thesis
    by (rule finite-subset[OF - H]) (auto simp: in-set-conv-nth)
qed

lemma last-in-set-dropWhile:
  assumes ‹∃ L ∈ set (xs @ [x]). ¬P L›
  shows ‹x ∈ set (dropWhile P (xs @ [x]))›
  using assms by (induction xs) auto

lemma mset-drop-upto: ‹mset (drop a N) = {#N!i. i ∈# mset-set {a..<length N}#}›
proof (induction N arbitrary: a)
  case Nil
  then show ?case by simp
next
  case (Cons c N)
  have upt: ‹{0..<Suc (length N)} = insert 0 {1..<Suc (length N)}›
    by auto
  then have H: ‹mset-set {0..<Suc (length N)} = add-mset 0 (mset-set {1..<Suc (length N)})›
    unfolding upt by auto
  have mset-case-Suc: ‹{#case x of 0 ⇒ c | Suc x ⇒ N ! x . x ∈# mset-set {Suc a..<Suc b}#} =
    {#N ! (x−1) . x ∈# mset-set {Suc a..<Suc b}#}› for a b
    by (rule image-mset-cong) (auto split: nat.splits)
  have Suc-Suc: ‹{Suc a..<Suc b} = Suc ` {a..<b}› for a b
    by auto
  then have mset-set-Suc-Suc: ‹mset-set {Suc a..<Suc b} = {#Suc n. n ∈# mset-set {a..<b}#}› for
a b
    unfolding Suc-Suc by (subst image-mset-mset-set[symmetric]) auto
 have ∗: ‹{#N ! (x−Suc 0) . x ∈# mset-set {Suc a..<Suc b}#} = {#N ! x . x ∈# mset-set {a..<b}#}›
    for a b
    by (auto simp add: mset-set-Suc-Suc)
  show ?case
```

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

lemma *last-list-update-to-last*:
  ‹*last* (*xs*[*x* := *last xs*]) = *last xs*›
  by (*metis last-list-update list-update.simps*(*1*))

lemma *take-map-nth-alt-def*: ‹*take n xs* = *map* ((!) *xs*) [*0*..<*min n* (*length xs*)]›
proof (*induction xs rule*: *rev-induct*)
  case *Nil*
  then show *?case* **by** *auto*
next
  case (*snoc x xs*) **note** *IH* = *this*
  show *?case*
  proof (*cases* ‹*n* < *length* (*xs* @ [*x*])›)
    case *True*
    then show *?thesis*
      using *IH* **by** (*auto simp*: *min-def nth-append*)
  next
    case *False*
    have [*simp*]:
      ‹*map* (λ*a*. *if a* < *length xs then xs* ! *a else* [*x*] ! (*a* − *length xs*)) [*0*..<*length xs*] =
       map (λ*a*. *xs* ! *a*) [*0*..<*length xs*]› **for** *xs* **and** *x* :: *'b*
      by (*rule map-cong*) *auto*
    show *?thesis*
      using *IH False* **by** (*auto simp*: *nth-append min-def*)
  qed
qed

1.3.6 Lexicographic Ordering

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

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

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

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

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

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

lemma prepend-same-lexn:
  assumes irrefl: ‹irrefl R›
  shows ‹(A @ B, A @ C) ∈ lexn R n ⟷ (B, C) ∈ lexn R (n − length A)› (is ‹?A ⟷ ?B›)
proof
  assume ?A
  then obtain xys x xs y ys where
    len-B: ‹length B = n − length A› and
    len-C: ‹length C = n − length A› and
    AB: ‹A @ B = xys @ x # xs› and
    AC: ‹A @ C = xys @ y # ys› and
    xy: ‹(x, y) ∈ R›
    by (auto simp: lexn-conv)
  have x-neq-y: ‹x ≠ y›
    using xy irrefl by (auto simp add: irrefl-def)
  then have ‹B = drop (length A) xys @ x # xs›
    using arg-cong[OF AB, of ‹drop (length A)›]
    apply (cases ‹length A − length xys›)
     apply (auto; fail)
    by (metis AB AC nth-append nth-append-length zero-less-Suc zero-less-diff)

  moreover have ‹C = drop (length A) xys @ y # ys›
    using arg-cong[OF AC, of ‹drop (length A)›] x-neq-y
    apply (cases ‹length A − length xys›)
     apply (auto; fail)
    by (metis AB AC nth-append nth-append-length zero-less-Suc zero-less-diff)
  ultimately show ?B
    using len-B[symmetric] len-C[symmetric] xy
    by (auto simp: lexn-conv)
next
  assume ?B
  then obtain xys x xs y ys where
    len-B: ‹length B = n − length A› and
    len-C: ‹length C = n − length A› and
    AB: ‹B = xys @ x # xs› and
    AC: ‹C = xys @ y # ys› and
    xy: ‹(x, y) ∈ R›
    by (auto simp: lexn-conv)
  define Axys where ‹Axys = A @ xys›

  have ‹A @ B = Axys @ x # xs›
    using AB Axys-def by auto

  moreover have ‹A @ C = Axys @ y # ys›
    using AC Axys-def by auto
  moreover have ‹Suc (length Axys + length xs) = n› and
    ‹length ys = length xs›
    using len-B len-C AB AC Axys-def by auto
  ultimately show ?A
    using len-B[symmetric] len-C[symmetric] xy
    by (auto simp: lexn-conv)
qed

lemma append-same-lexn:
  assumes irrefl: ‹irrefl R›
```

```
  shows ‹(B @ A , C @ A) ∈ lexn R n ⟷ (B, C) ∈ lexn R (n − length A)› (is ‹?A ⟷ ?B›)
proof
  assume ?A
  then obtain xys x xs y ys where
    len-B: ‹n = length B + length A› and
    len-C: ‹n = length C + length A› and
    AB: ‹B @ A = xys @ x # xs› and
    AC: ‹C @ A = xys @ y # ys› and
    xy: ‹(x, y) ∈ R›
    by (auto simp: lexn-conv)
  have x-neq-y: ‹x ≠ y›
    using xy irrefl by (auto simp add: irrefl-def)
  have len-C-B: ‹length C = length B›
    using len-B len-C by simp
  have len-B-xys: ‹length B > length xys›
    apply (rule ccontr)
    using arg-cong[OF AB, of ‹take (length B)›] arg-cong[OF AB, of ‹drop (length B)›]
      arg-cong[OF AC, of ‹drop (length C)›]  x-neq-y len-C-B
    by auto

  then have B: ‹B = xys @ x # take (length B − Suc (length xys)) xs›
    using arg-cong[OF AB, of ‹take (length B)›]
    by (cases ‹length B − length xys›) simp-all

  have C: ‹C = xys @ y # take (length C − Suc (length xys)) ys›
    using arg-cong[OF AC, of ‹take (length C)›] x-neq-y len-B-xys unfolding len-C-B[symmetric]
    by (cases ‹length C − length xys›)  auto
  show ?B
    using len-B[symmetric] len-C[symmetric] xy B C
    by (auto simp: lexn-conv)
next
  assume ?B
  then obtain xys x xs y ys where
    len-B: ‹length B = n − length A› and
    len-C: ‹length C = n − length A› and
    AB: ‹B = xys @ x # xs› and
    AC: ‹C = xys @ y # ys› and
    xy: ‹(x, y) ∈ R›
    by (auto simp: lexn-conv)
  define Ays Axs where ‹Ays = ys @ A› and‹Axs = xs @ A›

  have ‹B @ A = xys @ x # Axs›
    using AB Axs-def by auto

  moreover have ‹C @ A = xys @ y # Ays›
    using AC Ays-def by auto
  moreover have ‹Suc (length xys + length Axs) = n› and
    ‹length Ays = length Axs›
    using len-B len-C AB AC Axs-def Ays-def by auto
  ultimately show ?A
    using len-B[symmetric] len-C[symmetric] xy
    by (auto simp: lexn-conv)
qed

lemma irrefl-less-than [simp]: ‹irrefl less-than›
  by (auto simp: irrefl-def)
```

1.3.7 Remove

More lemmas about remove

lemma *distinct-remove1-last-butlast*:
 ‹*distinct xs* ⟹ *xs* ≠ [] ⟹ *remove1* (*last xs*) *xs* = *butlast xs*›
 by (*metis append-Nil2 append-butlast-last-id distinct-butlast not-distinct-conv-prefix remove1.simps*(*2*) *remove1-append*)

lemma *remove1-Nil-iff*:
 ‹*remove1 x xs* = [] ⟷ *xs* = [] ∨ *xs* = [*x*]›
 by (*cases xs*) *auto*

lemma *removeAll-upt*:
 ‹*removeAll k* [*a*..<*b*] = (*if k* ≥ *a* ∧ *k* < *b then* [*a*..<*k*] @ [*Suc k*..<*b*] *else* [*a*..<*b*])›
 by (*induction b*) *auto*

lemma *remove1-upt*:
 ‹*remove1 k* [*a*..<*b*] = (*if k* ≥ *a* ∧ *k* < *b then* [*a*..<*k*] @ [*Suc k*..<*b*] *else* [*a*..<*b*])›
 by (*subst distinct-remove1-removeAll*) (*auto simp*: *removeAll-upt*)

lemma *sorted-removeAll*: ‹*sorted C* ⟹ *sorted* (*removeAll k C*)›
 by (*metis map-ident removeAll-filter-not-eq sorted-filter*)

lemma *distinct-remove1-rev*: ‹*distinct xs* ⟹ *remove1 x* (*rev xs*) = *rev* (*remove1 x xs*)›
 using *split-list*[*of x xs*]
 by (*cases* ‹*x* ∈ *set xs*›) (*auto simp*: *remove1-append remove1-idem*)

Remove under condition

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

fun *remove1-cond* **where**
‹*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*›
 by (*induction xs*) *auto*

lemma *mset-map-mset-remove1-cond*:
 ‹*mset* (*map mset* (*remove1-cond* (λ*L*. *mset L* = *mset a*) *C*)) =
  remove1-mset (*mset a*) (*mset* (*map mset C*))›
 by (*induction C*) *auto*

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*›
 by (*induction xs*) *auto*

lemma *removeAll-cond-filter*: ‹*removeAll-cond P xs* = *filter* (λ*x*. ¬*P x*) *xs*›
 by (*induction xs*) *auto*

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*))›
by (*induction C*) *auto*

lemma *count-mset-count-list*:
‹*count* (*mset xs*) *x* = *count-list xs x*›
by (*induction xs*) *auto*

lemma *length-removeAll-count-list*:
‹*length* (*removeAll x xs*) = *length xs* − *count-list xs x*›
proof −
have ‹*length* (*removeAll x xs*) = *size* (*removeAll-mset x* (*mset xs*))›
by *auto*
also have ‹. . . = *size* (*mset xs*) − *count* (*mset xs*) *x*›
by (*metis count-le-replicate-mset-subset-eq le-refl size-Diff-submset size-replicate-mset*)
also have ‹ . . . = *length xs* − *count-list xs x*›
unfolding *count-mset-count-list* **by** *simp*
finally show *?thesis* **.**
qed

lemma *removeAll-notin*: ‹$a \notin\# A \Longrightarrow$ *removeAll-mset a A* = *A*›
using *count-inI* **by** *force*

Filter

lemma *distinct-filter-eq-if*:
‹*distinct C* ⟹ *length* (*filter* ((=) *L*) *C*) = (*if L* ∈ *set C then 1 else 0*)›
by (*induction C*) *auto*

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*)›
apply (*subst* (*5*) *append-take-drop-id*[*of i*, *symmetric*])
using *assms upd-conv-take-nth-drop*[*of i xs x*] *Cons-nth-drop-Suc*[*of i xs*, *symmetric*]
unfolding *filter-append length-append*
by *simp*

lemma *length-filter-update-false*:
assumes ‹*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*)›
apply (*subst* (*5*) *append-take-drop-id*[*of i*, *symmetric*])
using *assms upd-conv-take-nth-drop*[*of i xs x*] *Cons-nth-drop-Suc*[*of i xs*, *symmetric*]
unfolding *filter-append length-append*
by *simp*

lemma *mset-set-mset-set-minus-id-iff*:
assumes ‹*finite A*›
shows ‹*mset-set A* = *mset-set* (*A* − *B*) ⟷ (∀ *b* ∈ *B*. *b* ∉ *A*)›
proof −
have *f1*: *mset-set A* = *mset-set* (*A* − *B*) ⟷ *A* − *B* = *A*
using *assms* **by** (*metis* (*no-types*) *finite-Diff finite-set-mset-mset-set*)
then show *?thesis*
by *blast*
qed

lemma *mset-set-eq-mset-set-more-conds*:
‹*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 −
assume *?F*
then have ‹?A ⟷ (∀ x ∈ {x. P x}. x ∈ {x. Q x ∧ P x})›
by (*subst mset-set-eq-iff*) *auto*
also have ‹. . . ⟷ (∀ x. P x ⟶ Q x)›
by *blast*
finally show *?thesis* **.**
qed

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

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

1.3.8 Sorting

See ⟦*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›
using *properties-for-sort* **by** *auto*

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

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

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

lemma *removeAll-insort*:
‹sorted xs ⟹ k ≠ k′ ⟹ removeAll k′ (insort k xs) = insort k (removeAll k′ xs)›
by (*simp add*: *filter-insort removeAll-filter-not-eq*)

1.3.9 Distinct Multisets

lemma *distinct-mset-remdups-mset-id*: ‹*distinct-mset C* ⟹ *remdups-mset C* = *C*›
  by (*induction C*) *auto*

lemma *notin-add-mset-remdups-mset*:
  ‹*a* ∉# *A* ⟹ *add-mset a* (*remdups-mset A*) = *remdups-mset* (*add-mset a A*)›
  by *auto*

lemma *distinct-mset-image-mset*:
  ‹*distinct-mset* (*image-mset f* (*mset xs*)) ⟷ *distinct* (*map f xs*)›
  apply (*subst mset-map*[*symmetric*])
  apply (*subst distinct-mset-mset-distinct*)
  ..

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 −
  obtain *M1* **where** *M1*: ‹*M* = *add-mset L M1*›
    using *multi-member-split*[*OF L*] **by** *blast*
  obtain *M2* **where** *M2*: ‹*M1* = *add-mset L′ M2*›
    using *multi-member-split*[*of L′ M1*] *LL′ L′* **unfolding** *M1* **by** (*auto simp*: *add-mset-eq-add-mset*)
  show *False*
    using *dist* **unfolding** *M1 M2* **by** *auto*
qed

lemma *distinct-mset-remdups-mset*[*simp*]: ‹*distinct-mset* (*remdups-mset S*)›
  using *count-remdups-mset-eq-1* **unfolding** *distinct-mset-def* **by** *metis*

lemma *remdups-mset-idem*: ‹*remdups-mset* (*remdups-mset a*) = *remdups-mset a*›
  using *distinct-mset-remdups-mset distinct-mset-remdups-mset-id* **by** *fast*

1.3.10 Set of Distinct Multisets

definition *distinct-mset-set* :: ‹*′a multiset set* ⇒ *bool*› **where**
  ‹*distinct-mset-set* Σ ⟷ (∀ *S* ∈ Σ. *distinct-mset S*)›

lemma *distinct-mset-set-empty*[*simp*]: ‹*distinct-mset-set* {}›
  unfolding *distinct-mset-set-def* **by** *auto*

lemma *distinct-mset-set-singleton*[*iff*]: ‹*distinct-mset-set* {*A*} ⟷ *distinct-mset A*›
  unfolding *distinct-mset-set-def* **by** *auto*

lemma *distinct-mset-set-insert*[*iff*]:
  ‹*distinct-mset-set* (*insert S* Σ) ⟷ (*distinct-mset S* ∧ *distinct-mset-set* Σ)›
  unfolding *distinct-mset-set-def* **by** *auto*

lemma *distinct-mset-set-union*[*iff*]:
  ‹*distinct-mset-set* (Σ ∪ Σ′) ⟷ (*distinct-mset-set* Σ ∧ *distinct-mset-set* Σ′)›
  unfolding *distinct-mset-set-def* **by** *auto*

lemma *in-distinct-mset-set-distinct-mset*:
‹$a \in \Sigma \Longrightarrow$ *distinct-mset-set* $\Sigma \Longrightarrow$ *distinct-mset a*›
unfolding *distinct-mset-set-def* **by** *auto*

lemma *distinct-mset-mset-set*: ‹*distinct-mset* (*mset-set A*)›
unfolding *distinct-mset-def count-mset-set-if* **by** (*auto simp*: *not-in-iff*)

lemma *distinct-mset-filter-mset-set*[*simp*]: ‹*distinct-mset* {#*a* ∈# *mset-set A*. *P a*#}›
by (*simp add*: *distinct-mset-filter distinct-mset-mset-set*)

lemma *distinct-mset-set-distinct*: ‹*distinct-mset-set* (*mset* ‘ *set Cs*) ⟷ (∀ *c*∈ *set Cs*. *distinct c*)›
unfolding *distinct-mset-set-def* **by** *auto*

1.3.11 Sublists

lemma *nths-single-if*: ‹*nths l* {*n*} = (*if n* < *length l then* [*l*!*n*] *else* [])›
proof −
have [*simp*]: ‹*0* < *n* ⟹ {*j*. *Suc j* = *n*} = {*n*−*1*}› **for** *n*
by *auto*
show *?thesis*
apply (*induction l arbitrary*: *n*)
subgoal by (*auto simp*: *nths-def*)
subgoal by (*auto simp*: *nths-Cons*)
done
qed

lemma *atLeastLessThan-Collect*: ‹{*a*..<*b*} = {*j*. *j* ≥ *a* ∧ *j* < *b*}›
by *auto*

lemma *mset-nths-subset-mset*: ‹*mset* (*nths xs A*) ⊆# *mset xs*›
apply (*induction xs arbitrary*: *A*)
subgoal by *auto*
subgoal for *a xs A*
using *subset-mset.add-increasing2*[*of* ‹*add-mset* - {#}› ‹*mset* (*nths xs* {*j*. *Suc j* ∈ *A*})›
‹*mset xs*›]
by (*auto simp*: *nths-Cons*)
done

lemma *nths-id-iff*:
‹*nths xs A* = *xs* ⟷ {*0*..<*length xs*} ⊆ *A* ›
proof −
have ‹{*j*. *Suc j* ∈ *A*} = (λ*j*. *j*−*1*) ‘ (*A* − {*0*})› **for** *A*
using *DiffI* **by** (*fastforce simp*: *image-iff*)
have *1*: ‹{*0*..<*b*} ⊆ {*j*. *Suc j* ∈ *A*} ⟷ (∀ *x*. *x*−*1* < *b* ⟶ *x* ≠ *0* ⟶ *x* ∈ *A*)›
for *A* :: ‹*nat set*› **and** *b* :: *nat*
by *auto*
have [*simp*]: ‹{*0*..<*b*} ⊆ {*j*. *Suc j* ∈ *A*} ⟷ (∀ *x*. *x*−*1* < *b* ⟶ *x* ∈ *A*)›
if ‹*0* ∈ *A*› **for** *A* :: ‹*nat set*› **and** *b* :: *nat*
using *that* **unfolding** *1* **by** *auto*
have [*simp*]: ‹*nths xs* {*j*. *Suc j* ∈ *A*} = *a* # *xs* ⟷ *False*›
for *a* :: *'a* **and** *xs* :: ‹*'a list*› **and** *A* :: ‹*nat set*›
using *mset-nths-subset-mset*[*of xs* ‹{*j*. *Suc j* ∈ *A*}›] **by** *auto*
show *?thesis*
apply (*induction xs arbitrary*: *A*)
subgoal by *auto*

```
    subgoal
      by (auto 5 5 simp: nths-Cons) fastforce
    done
qed

lemma nts-upt-length[simp]: ‹nths xs {0..<length xs} = xs›
  by (auto simp: nths-id-iff)

lemma nths-shift-lemma':
  ‹map fst [p←zip xs [i..<i + n]. snd p + b ∈ A] = map fst [p←zip xs [0..<n]. snd p + b + i ∈ A]›
proof (induct xs arbitrary: i n b)
  case Nil
  then show ?case by simp
next
  case (Cons a xs)
  have 1: ‹map fst [p←zip (a # xs) (i # [Suc i..<i + n]). snd p + b ∈ A] =
     (if i + b ∈ A then a#map fst [p←zip xs [Suc i..<i + n]. snd p + b ∈ A]
     else map fst [p←zip xs [Suc i..<i + n]. snd p + b ∈A])›
    by simp
  have 2: ‹map fst [p←zip (a # xs) [0..<n] . snd p + b + i ∈ A] =
     (if i + b ∈ A then a # map fst [p←zip xs [1..<n]. snd p + b + i ∈ A]
      else map fst [p←zip (xs) [1..<n] . snd p + b + i ∈ A])›
    if ‹n > 0›
    by (subst upt-conv-Cons) (use that in ‹auto simp: ac-simps›)
  show ?case
  proof (cases n)
    case 0
    then show ?thesis by simp
  next
    case n: (Suc m)
    then have i-n-m: ‹i + n = Suc i + m›
      by auto
    have 3: ‹map fst [p←zip xs [Suc i..<i+n] . snd p + b ∈ A] =
             map fst [p←zip xs [0..<m] . snd p + b + Suc i ∈ A]›
      using Cons[of b ‹Suc i› m] unfolding i-n-m .
    have 4: ‹map fst [p←zip xs [1..<n] . snd p + b + i ∈ A] =
                 map fst [p←zip xs [0..<m] . Suc (snd p + b + i) ∈ A]›
      using Cons[of ‹b+i› 1 m] unfolding n Suc-eq-plus1-left add.commute[of 1]
      by (simp-all add: ac-simps)
    show ?thesis
      apply (subst upt-conv-Cons)
      using n apply (simp; fail)
      apply (subst 1)
      apply (subst 2)
      using n apply (simp; fail)
      apply (subst 3)
      apply (subst 3)

      apply (subst 4)
      apply (subst 4)
      by force
  qed
qed

lemma nths-Cons-upt-Suc: ‹nths (a # xs) {0..<Suc n} = a # nths xs {0..<n}›
  unfolding nths-def
```

```
  apply (subst upt-conv-Cons)
   apply simp
  using nths-shift-lemma'[of 0 ‹{0..<Suc n}› ‹xs› 1 ‹length xs›]
  by (simp-all add: ac-simps)


lemma nths-empty-iff: ‹nths xs A = [] ⟷ {..<length xs} ∩ A = {}›
proof (induction xs arbitrary: A)
  case Nil
  then show ?case by auto
next
  case (Cons a xs) note IH = this(1)
  have ‹(∀x<length xs. x ≠ 0 ⟶ x ∉ A)›
    if a1: ‹{..<length xs} ∩ {j. Suc j ∈ A} = {}›
  proof (intro allI impI)
    fix nn
    assume nn: ‹nn < length xs› ‹nn ≠ 0›
    moreover have ∀n. Suc n ∉ A ∨ ¬ n < length xs
      using a1 by blast
    then show nn ∉ A
      using nn
      by (metis (no-types) lessI less-trans list-decode.cases)
  qed
  show ?case
  proof (cases ‹0 ∈ A›)
    case True
    then show ?thesis by (subst nths-Cons) auto
  next
    case False
    then show ?thesis
      by (subst nths-Cons) (use less-Suc-eq-0-disj IH in auto)
  qed
qed

lemma nths-upt-Suc:
  assumes ‹i < length xs›
  shows ‹nths xs {i..<length xs} = xs!i # nths xs {Suc i..<length xs}›
proof −
  have upt: ‹{i..<k} = {j. i ≤ j ∧ j < k}› for i k :: nat
    by auto
  show ?thesis
    using assms
  proof (induction xs arbitrary: i)
    case Nil
    then show ?case by simp
  next
    case (Cons a xs i) note IH = this(1) and i-le = this(2)
    have [simp]: ‹i − Suc 0 ≤ j ⟷ i ≤ Suc j› if ‹i > 0› for j
      using that by auto
    show ?case
      using IH[of ‹i−1›] i-le
      by (auto simp add: nths-Cons upt)
  qed
qed

lemma nths-upt-Suc':
```

assumes ‹$i < b$› **and** ‹$b <= length\ xs$›
shows ‹$nths\ xs\ \{i..<b\} = xs!i\ \#\ nths\ xs\ \{Suc\ i..<b\}$›
proof −
have *S1*: ‹$\{j.\ i \leq Suc\ j \wedge j < b - Suc\ 0\} = \{j.\ i \leq Suc\ j \wedge Suc\ j < b\}$› **for** *i b*
by *auto*
have *S2*: ‹$\{j.\ i \leq j \wedge j < b - Suc\ 0\} = \{j.\ i \leq j \wedge Suc\ j < b\}$› **for** *i b*
by *auto*
have *upt*: ‹$\{i..<k\} = \{j.\ i \leq j \wedge j < k\}$› **for** *i k* :: *nat*
by *auto*
show *?thesis*
using *assms*
proof (*induction xs arbitrary*: *i b*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a xs i*) **note** *IH* = *this*(*1*) **and** *i-le* = *this*(*2*,*3*)
have [*simp*]: ‹$i - Suc\ 0 \leq j \longleftrightarrow i \leq Suc\ j$› **if** ‹$i > 0$› **for** *j*
using *that* **by** *auto*
have ‹$i - Suc\ 0 < b - Suc\ 0 \vee (i = 0)$›
using *i-le* **by** *linarith*
moreover have ‹$b - Suc\ 0 \leq length\ xs \vee xs = []$›
using *i-le* **by** *auto*
ultimately show *?case*
using *IH*[*of* ‹$i-1$› ‹$b-1$›] *i-le*
apply (*subst nths-Cons*)
apply (*subst nths-Cons*)
by (*auto simp*: *upt S1 S2*)
qed
qed

lemma *Ball-set-nths*: ‹$(\forall L \in set\ (nths\ xs\ A).\ P\ L) \longleftrightarrow (\forall i \in A \cap \{0..<length\ xs\}.\ P\ (xs\ !\ i))$ ›
unfolding *set-nths* **by** *fastforce*

1.3.12 Product Case

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

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

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

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

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

lemma *prod-cases10* [*cases type*]:
obtains (*fields*) *a b c d e f g h i j* **where** $y = (a, b, c, d, e, f, g, h, i, j)$

by (*cases y*, *cases* ‹*snd y*›) *auto*

lemma *prod-induct10* [*case-names fields*, *induct type*]:
($\bigwedge a\ b\ c\ d\ e\ f\ g\ h\ i\ j.\ P\ (a,\ b,\ c,\ d,\ e,\ f,\ g,\ h,\ i,\ j)) \Longrightarrow P\ x$
by (*cases x*) *blast*

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *prod-induct17* [*case-names fields*, *induct type*]:
(⋀*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*)) ⟹ *P x*
by (*cases x*) *blast*

lemma *prod-cases18* [*cases type*]:
obtains (*fields*) *a b c d e f g h i j k l m n p q r s* **where**
y = (*a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, *i*, *j*, *k*, *l*, *m*, *n*, *p*, *q*, *r*, *s*)
by (*cases y*, *cases* ‹*snd y*›) *auto*

lemma *prod-induct18* [*case-names fields*, *induct type*]:
(⋀*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*)) ⟹ *P x*
by (*cases x*) *blast*

lemma *prod-cases19* [*cases type*]:
obtains (*fields*) *a b c d e f g h i j k l m n p q r s t* **where**
y = (*a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, *i*, *j*, *k*, *l*, *m*, *n*, *p*, *q*, *r*, *s*, *t*)
by (*cases y*, *cases* ‹*snd y*›) *auto*

lemma *prod-induct19* [*case-names fields*, *induct type*]:
(⋀*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*)) ⟹ *P x*
by (*cases x*) *blast*

lemma *prod-cases20* [*cases type*]:
obtains (*fields*) *a b c d e f g h i j k l m n p q r s t u* **where**
y = (*a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, *i*, *j*, *k*, *l*, *m*, *n*, *p*, *q*, *r*, *s*, *t*, *u*)
by (*cases y*, *cases* ‹*snd y*›) *auto*

lemma *prod-induct20* [*case-names fields*, *induct type*]:
(⋀*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*)) ⟹ *P x*
by (*cases x*) *blast*

lemma *prod-cases21* [*cases type*]:
obtains (*fields*) *a b c d e f g h i j k l m n p q r s t u v* **where**
y = (*a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, *i*, *j*, *k*, *l*, *m*, *n*, *p*, *q*, *r*, *s*, *t*, *u*, *v*)
by (*cases y*, *cases* ‹*snd y*›) *auto*

lemma *prod-induct21* [*case-names fields*, *induct type*]:
(⋀*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*)) ⟹ *P x*
by (*cases x*) (*blast 43*)

lemma *prod-cases22* [*cases type*]:
obtains (*fields*) *a b c d e f g h i j k l m n p q r s t u v w* **where**
y = (*a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, *i*, *j*, *k*, *l*, *m*, *n*, *p*, *q*, *r*, *s*, *t*, *u*, *v*, *w*)
by (*cases y*, *cases* ‹*snd y*›) *auto*

lemma *prod-induct22* [*case-names fields*, *induct type*]:
(⋀*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*)) ⟹ *P x*

by (*cases x*) (*blast 43*)

lemma *prod-cases23* [*cases type*]:
obtains (*fields*) *a b c d e f g h i j k l m n p q r s t u v w x* **where**
$y = (a, b, c, d, e, f, g, h, i, j, k, l, m, n, p, q, r, s, t, u, v, w, x)$
by (*cases y*, *cases* ‹*snd y*›) *auto*

lemma *prod-induct23* [*case-names fields*, *induct type*]:
$(\bigwedge a\ b\ c\ d\ e\ f\ g\ h\ i\ j\ k\ l\ m\ n\ p\ q\ r\ s\ t\ u\ v\ w\ y.$
$P\ (a, b, c, d, e, f, g, h, i, j, k, l, m, n, p, q, r, s, t, u, v, w, y)) \Longrightarrow P\ x$
by (*cases x*) (*blast 43*)

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

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

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

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

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

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

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

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

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

1.3.14 Multisets

We have a lit of lemmas about multisets. Some of them have already moved to *Nested-Multisets-Ordinals.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*: ‹$L \in\# D \Longrightarrow D \neq \{\#\}$›
  by *auto*

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

definition *union-mset-list* **where**
‹*union-mset-list xs ys* ≡ *case-prod append* (*fold* (λ*x* (*ys*, *zs*). (*remove1 x ys*, *x* # *zs*)) *xs* (*ys*, []))›

lemma *union-mset-list*:
  ‹*mset xs* ∪# *mset ys* = *mset* (*union-mset-list xs ys*)›
proof −
  have ‹⋀*zs*. *mset* (*case-prod append* (*fold* (λ*x* (*ys*, *zs*). (*remove1 x ys*, *x* # *zs*)) *xs* (*ys*, *zs*))) =
      (*mset xs* ∪# *mset ys*) + *mset zs*›
    by (*induct xs arbitrary*: *ys*) (*simp-all add*: *multiset-eq-iff*)
  then show *?thesis* **by** (*simp add*: *union-mset-list-def*)
qed

lemma *union-mset-list-Nil*[*simp*]: ‹*union-mset-list* [] *bi* = *bi*›
  by (*auto simp*: *union-mset-list-def*)

lemma *size-le-Suc-0-iff*: ‹*size M* ≤ *Suc 0* ⟷ ((∃ *a b*. *M* = {#*a*#}) ∨ *M* = {#})›
   using *size-1-singleton-mset* **by** (*auto simp*: *le-Suc-eq*)

lemma *size-2-iff*: ‹*size M* = *2* ⟷ (∃ *a b*. *M* = {#*a*, *b*#})›
  by (*metis One-nat-def Suc-1 Suc-pred empty-not-add-mset nonempty-has-size size-Diff-singleton*
      size-eq-Suc-imp-eq-union size-single union-single-eq-diff union-single-eq-member)

lemma *subset-eq-mset-single-iff*: ‹*x2* ⊆# {#*L*#} ⟷ *x2* = {#} ∨ *x2* = {#*L*#}›
  by (*metis single-is-union subset-mset.add-diff-inverse subset-mset.eq-refl subset-mset.zero-le*)

lemma *mset-eq-size-2*:
  ‹*mset xs* = {#*a*, *b*#} ⟷ *xs* = [*a*, *b*] ∨ *xs* = [*b*, *a*]›
  by (*cases xs*) (*auto simp*: *add-mset-eq-add-mset Diff-eq-empty-iff-mset subset-eq-mset-single-iff*)

lemma *butlast-list-update*:
  ‹*w* < *length xs* ⟹ *butlast* (*xs*[*w* := *last xs*]) = *take w xs* @ *butlast* (*last xs* # *drop* (*Suc w*) *xs*)›
  by (*induction xs arbitrary*: *w*) (*auto split*: *nat.splits if-splits simp*: *upd-conv-take-nth-drop*)

lemma *mset-butlast-remove1-mset*: ‹*xs* ≠ [] ⟹ *mset* (*butlast xs*) = *remove1-mset* (*last xs*) (*mset xs*)›
  apply (*subst*(*2*) *append-butlast-last-id*[*of xs*, *symmetric*])
   apply *assumption*
  apply (*simp only*: *mset-append*)
  by *auto*

lemma *distinct-mset-mono*: ‹*D′* ⊆# *D* ⟹ *distinct-mset D* ⟹ *distinct-mset D′*›
  by (*metis distinct-mset-union subset-mset.le-iff-add*)

lemma *distinct-mset-mono-strict*: ‹D′ ⊂# D ⟹ distinct-mset D ⟹ distinct-mset D′›
  using *distinct-mset-mono* **by** *auto*

lemma *subset-mset-trans-add-mset*:
  ‹D ⊆# D′ ⟹ D ⊆# add-mset L D′›
  by (*metis add-mset-remove-trivial diff-subset-eq-self subset-mset.dual-order.trans*)

lemma *subset-add-mset-notin-subset*: ‹L ∉# E ⟹ E ⊆# add-mset L D ⟷ E ⊆# D›
  by (*meson subset-add-mset-notin-subset-mset subset-mset-trans-add-mset*)

lemma *remove1-mset-empty-iff*: ‹remove1-mset L N = {#} ⟷ N = {#L#} ∨ N = {#}›
  by (*cases* ‹L ∈# N›; *cases N*) *auto*

lemma *mset-set-subset-iff*:
  ‹mset-set A ⊆# I ⟷ infinite A ∨ A ⊆ set-mset I›
  by (*metis finite-set-mset finite-set-mset-mset-set mset-set.infinite mset-set-set-mset-subseteq
    set-mset-mono subset-imp-msubset-mset-set subset-mset.dual-order.trans subset-mset.max-bot
    subset-mset.max-def*)

lemma *distinct-subseteq-iff*:
  assumes *dist*: ‹distinct-mset M›
  shows ‹set-mset M ⊆ set-mset N ⟷ M ⊆# N›
proof
  assume ‹set-mset M ⊆ set-mset N›
  then show ‹M ⊆# N›
    using *dist* **by** (*metis distinct-mset-set-mset-ident mset-set-subset-iff*)
next
  assume ‹M ⊆# N›
  then show ‹set-mset M ⊆ set-mset N›
    by (*metis set-mset-mono*)
qed

lemma *distinct-set-mset-eq-iff*:
  assumes ‹distinct-mset M› ‹distinct-mset N›
  shows ‹set-mset M = set-mset N ⟷ M = N›
  using *assms distinct-mset-set-mset-ident* **by** *fastforce*

lemma (**in** −) *distinct-mset-union2*:
  ‹distinct-mset (A + B) ⟹ distinct-mset B›
  using *distinct-mset-union*[*of B A*]
  by (*auto simp*: *ac-simps*)

lemma *in-remove1-msetI*: ‹x ≠ a ⟹ x ∈# M ⟹ x ∈# remove1-mset a M›
  by (*simp add*: *in-remove1-mset-neq*)

lemma *count-multi-member-split*:
   ‹count M a ≥ n ⟹ ∃ M′. M = replicate-mset n a + M′›
  apply (*induction n arbitrary*: *M*)
  subgoal by *auto*
  subgoal premises *IH* **for** *n M*
    using *IH*(*1*)[*of* ‹remove1-mset a M›] *IH*(*2*)
    apply (*cases* ‹n ≤ count M a − Suc 0›)
     apply (*auto dest*!: *Suc-le-D*)
    by (*metis count-greater-zero-iff insert-DiffM zero-less-Suc*)
  done

lemma *count-image-mset-multi-member-split*:
‹*count* (*image-mset f M*) *L* ≥ *Suc 0* ⟹ ∃ *K*. *f K* = *L* ∧ *K* ∈# *M*›
by *auto*

lemma *count-image-mset-multi-member-split-2*:
assumes *count*: ‹*count* (*image-mset f M*) *L* ≥ *2*›
shows ‹∃ *K K′ M′*. *f K* = *L* ∧ *K* ∈# *M* ∧ *f K′* = *L* ∧ *K′* ∈# *remove1-mset K M* ∧
M = {#*K*, *K′*#} + *M′*›
proof −
obtain *K* **where**
K: ‹*f K* = *L*› ‹*K* ∈# *M*›
using *count-image-mset-multi-member-split*[*of f M L*] *count* **by** *fastforce*
then obtain *K′* **where**
K′: ‹*f K′* = *L*› ‹*K′* ∈# *remove1-mset K M*›
using *count-image-mset-multi-member-split*[*of f* ‹*remove1-mset K M*› *L*] *count*
by (*auto dest*!: *multi-member-split*)
moreover have ‹∃ *M′*. *M* = {#*K*, *K′*#} + *M′*›
using *multi-member-split*[*of K M*] *multi-member-split*[*of K′* ‹*remove1-mset K M*›] *K K′*
by (*auto dest*!: *multi-member-split*)
then show *?thesis*
using *K K′* **by** *blast*
qed

lemma *minus-notin-trivial*: *L* ∉# *A* ⟹ *A* − *add-mset L B* = *A* − *B*
by (*metis diff-intersect-left-idem inter-add-right1*)

lemma *minus-notin-trivial2*: ‹*b* ∉# *A* ⟹ *A* − *add-mset e* (*add-mset b B*) = *A* − *add-mset e B*›
by (*subst add-mset-commute*) (*auto simp*: *minus-notin-trivial*)

lemma *diff-union-single-conv3*: ‹*a* ∉# *I* ⟹ *remove1-mset a* (*I* + *J*) = *I* + *remove1-mset a J*›
by (*metis diff-union-single-conv remove-1-mset-id-iff-notin union-iff*)

lemma *filter-union-or-split*:
‹{#*L* ∈# *C*. *P L* ∨ *Q L*#} = {#*L* ∈# *C*. *P L*#} + {#*L* ∈# *C*. ¬*P L* ∧ *Q L*#}›
by (*induction C*) *auto*

lemma *subset-mset-minus-eq-add-mset-noteq*: ‹*A* ⊂# *C* ⟹ *A* − *B* ≠ *C*›
by (*auto simp*: *dest*: *in-diffD*)

lemma *minus-eq-id-forall-notin-mset*:
‹*A* − *B* = *A* ⟷ (∀ *L* ∈# *B*. *L* ∉# *A*)›
by (*induction A*)
(*auto dest*!: *multi-member-split simp*: *subset-mset-minus-eq-add-mset-noteq*)

lemma *in-multiset-minus-notin-snd*[*simp*]: ‹*a* ∉# *B* ⟹ *a* ∈# *A* − *B* ⟷ *a* ∈# *A*›
by (*metis count-greater-zero-iff count-inI in-diff-count*)

lemma *distinct-mset-in-diff*:
‹*distinct-mset C* ⟹ *a* ∈# *C* − *D* ⟷ *a* ∈# *C* ∧ *a* ∉# *D*›
by (*meson distinct-mem-diff-mset in-multiset-minus-notin-snd*)

lemma *diff-le-mono2-mset*: ‹*A* ⊆# *B* ⟹ *C* − *B* ⊆# *C* − *A*›
apply (*auto simp*: *subseteq-mset-def ac-simps*)
by (*simp add*: *diff-le-mono2*)

lemma *subseteq-remove1*[*simp*]: ‹*C* ⊆# *C′* ⟹ *remove1-mset L C* ⊆# *C′*›

  by (*meson diff-subset-eq-self subset-mset.dual-order.trans*)

lemma *filter-mset-cong2*:
  $(\bigwedge x.\ x \in\# M \Longrightarrow f\ x = g\ x) \Longrightarrow M = N \Longrightarrow$ *filter-mset f M* = *filter-mset g N*
  by (*hypsubst*, *rule filter-mset-cong*, *simp*)

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-mset f M* = *filter-mset g N*›
proof −
  define *NM* **where** ‹$NM = N - M$›
  have *N*: ‹$N = M + NM$›
    unfolding *NM-def* **using** *MN* **by** *simp*
  have ‹*filter-mset g NM* = {#}›
    using *notin* **unfolding** *NM-def*[*symmetric*] **by** (*auto simp*: *filter-mset-empty-conv*)
  moreover have ‹*filter-mset f M* = *filter-mset g M*›
    by (*rule filter-mset-cong*) (*use M-eq* **in** *auto*)
  ultimately show *?thesis*
    unfolding *N* **by** *simp*
qed

lemma *notin-filter-mset*:
  ‹$K \notin\# C \Longrightarrow$ *filter-mset P C* = *filter-mset* $(\lambda L.\ P\ L \wedge L \neq K)\ C$›
  by (*rule filter-mset-cong*) *auto*

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

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

```
lemma remove1-mset-union-distrib:
  ‹remove1-mset a (M ∪# N) = remove1-mset a M ∪# remove1-mset a N›
  by (auto simp: multiset-eq-iff)


lemma member-add-mset: ‹a ∈# add-mset x xs ⟷ a = x ∨ a ∈# xs›
  by simp

lemma sup-union-right-if:
  ‹N ∪# add-mset x M =
     (if x ∉# N then add-mset x (N ∪# M) else add-mset x (remove1-mset x N ∪# M))›
  by (auto simp: sup-union-right2)

lemma same-mset-distinct-iff:
  ‹mset M = mset M' ⟹ distinct M ⟷ distinct M'›
  by (auto simp: distinct-mset-mset-distinct[symmetric] simp del: distinct-mset-mset-distinct)

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 ⟷ M' = M› (is ‹?A = ?B›)
proof
  assume ?B
  then show ?A by auto
next
  assume ?A
  then show ?B
    using inj
  proof(induction M arbitrary: M')
    case empty
    then show ?case by auto
  next
    case (add x M) note IH = this(1) and H = this(2) and inj = this(3)

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

lemma image-msetI: ‹x ∈# A ⟹ f x ∈# f `# A›
  by (auto)

lemma inj-image-mset-eq-iff:
  assumes inj: ‹inj f›
  shows ‹image-mset f M' = image-mset f M ⟷ M' = M›
  using inj-on-image-mset-eq-iff[of f M' M] assms
  by (simp add: inj-eq multiset.inj-map)
```

```
lemma singleton-eq-image-mset-iff: ‹{#a#} = f `# NE' ⟷ (∃ b. NE' = {#b#} ∧ f b = a)›
  by (cases NE') auto

lemma image-mset-If-eq-notin:
   ‹C ∉# A ⟹ {#f (if x = C then a x else b x). x ∈# A#} = {# f(b x). x ∈# A #}›
  by (induction A) auto

lemma finite-mset-set-inter:
  ‹finite A ⟹ finite B ⟹ mset-set (A ∩ B) = mset-set A ∩# mset-set B›
  apply (induction A rule: finite-induct)
  subgoal by auto
  subgoal for a A
    apply (cases ‹a ∈ B›; cases ‹a ∈# mset-set B›)
    using multi-member-split[of a ‹mset-set B›]
    by (auto simp: mset-set.insert-remove)
  done

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

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

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

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

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

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

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

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

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

lemma *image-filter-replicate-mset*:
‹{#*Ca* ∈# *replicate-mset m C*. *P Ca*#} = (*if P C then replicate-mset m C else* {#})›
by (*induction m*) *auto*

lemma *size-Union-mset-image-mset*:
‹*size* ($\sum_{\#}$ (*A* :: '*a multiset multiset*)) = (∑ *i* ∈# *A*. *size i*)›
by (*induction A*) *auto*

lemma *image-mset-minus-inj-on*:
‹*inj-on f* (*set-mset A* ∪ *set-mset B*) ⟹ *f* '# (*A* − *B*) = *f* '# *A* − *f* '# *B*›
apply (*induction A arbitrary*: *B*)
subgoal by *auto*
subgoal for *x A B*
apply (*cases* ‹*x* ∈# *B*›)
apply (*auto dest*!: *multi-member-split*)
apply (*subst diff-add-mset-swap*)
apply *auto*
done
done

lemma *filter-mset-mono-subset*:
‹*A* ⊆# *B* ⟹ (⋀*x*. *x* ∈# *A* ⟹ *P x* ⟹ *Q x*) ⟹ *filter-mset P A* ⊆# *filter-mset Q B*›
by (*metis multiset-filter-mono multiset-filter-mono2 subset-mset.order-trans*)

lemma *mset-inter-empty-set-mset*: ‹*M* ∩# *xc* = {#} ⟷ *set-mset M* ∩ *set-mset xc* = {}›
by (*induction xc*) *auto*

lemma *sum-mset-cong*:
‹(⋀*A*. *A* ∈# *M* ⟹ *f A* = *g A*) ⟹ (∑ *A* ∈# *M*. *f A*) = (∑ *A* ∈# *M*. *g A*)›
by (*induction M*) *auto*

lemma *sum-mset-mset-set-sum-set*:
‹(∑ *A* ∈# *mset-set As*. *f A*) = (∑ *A* ∈ *As*. *f A*)›
apply (*cases* ‹*finite As*›)
by (*induction As rule*: *finite-induct*) *auto*

lemma *sum-mset-sum-count*:
‹(∑ *A* ∈# *As*. *f A*) = (∑ *A* ∈ *set-mset As*. *count As A* ∗ *f A*)›
proof (*induction As*)
case *empty*
then show *?case* **by** *auto*
next
case (*add x As*)
define *n* **where** ‹*n* = *count As x*›

define As' **where** ‹$As' \equiv$ *removeAll-mset x As*›
have *As*: ‹$As = As' +$ *replicate-mset n x*›
by (*auto simp*: As'*-def n-def intro*!: *multiset-eqI*)
have [*simp*]: ‹*set-mset* $As' - \{x\} =$ *set-mset* As'› ‹*count* As' $x = 0$› ‹$x \notin\#$ As'›
unfolding As'*-def*
by *auto*
have ‹ ($\sum A \in$*set-mset* As'.
(*if* $x = A$ *then Suc* (*count* ($As' +$ *replicate-mset n x*) *A*)
else count ($As' +$ *replicate-mset n x*) *A*) $*$
f A) =
($\sum A \in$*set-mset* As'.
(*count* ($As' +$ *replicate-mset n x*) *A*) $*$
f A)›
by (*rule sum.cong*) *auto*
then show *?case* **using** *add* **by** (*auto simp*: *As sum.insert-remove*)
qed

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*)›
by (*induction M*) *auto*

lemma *sumset-diff-constant-left*:
assumes ‹$\bigwedge x.\ x \in\# A \Longrightarrow f\ x \leq n$›
shows ‹($\sum x \in\# A\ .\ n - f\ x$) = *size* $A * n - (\sum x \in\# A\ .\ f\ x)$›
proof −
have ‹*size* $A * n \geq (\sum x \in\# A\ .\ f\ x)$›
if ‹$\bigwedge x.\ x \in\# A \Longrightarrow f\ x \leq n$› **for** *A*
using *that*
by (*induction A*) (*force simp*: *ac-simps*)+
then show *?thesis*
using *assms*
by (*induction A*) (*auto simp*: *ac-simps*)
qed

lemma *mset-set-eq-mset-iff*: ‹*finite* $x \Longrightarrow$ *mset-set* $x =$ *mset* $xs \longleftrightarrow$ *distinct* $xs \wedge x =$ *set* xs›
apply (*auto simp flip*: *distinct-mset-mset-distinct eq-commute*[*of* - ‹*mset-set* -›]
simp: *distinct-mset-mset-set mset-set-set*)
apply (*metis finite-set-mset-mset-set set-mset-mset*)
apply (*metis finite-set-mset-mset-set set-mset-mset*)
done

lemma *distinct-mset-iff*:
‹¬*distinct-mset* $C \longleftrightarrow (\exists\, a\ C'.\ C =$ *add-mset a* (*add-mset a* C'))›
by (*metis* (*no-types*, *opaque-lifting*) *One-nat-def*
count-add-mset distinct-mset-add-mset distinct-mset-def
member-add-mset mset-add not-in-iff)

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

lemma *remdups-mset-sum-subset*: ‹$C \subseteq\# C' \Longrightarrow$ *remdups-mset* $(C + C')$ = *remdups-mset* C'›
‹$C \subseteq\# C' \Longrightarrow$ *remdups-mset* $(C' + C)$ = *remdups-mset* C'›
apply (*metis remdups-mset-def set-mset-mono set-mset-union sup.absorb-iff2*)
by (*metis add.commute le-iff-sup remdups-mset-def set-mset-mono set-mset-union*)

lemma *distinct-mset-subset-iff-remdups*:
‹*distinct-mset a* ⟹ *a* ⊆# *b* ⟷ *a* ⊆# *remdups-mset b*›
by (*simp add*: *distinct-mset-inter-remdups-mset subset-mset.le-iff-inf*)

lemma *remdups-mset-subset-add-mset*: ‹*remdups-mset C'* ⊆# *add-mset L C'*›
by (*meson distinct-mset-remdups-mset distinct-mset-subset-iff-remdups subset-mset.order-refl subset-mset-trans-add-mset*)

lemma *subset-mset-removeAll-iff*:
‹*M* ⊆# *removeAll-mset a M'* ⟷ *a* ∉# *M* ∧ *M* ⊆# *M'*›
by (*smt* (*verit*, *del-insts*) *count-replicate-mset diff-le-mono diff-subset-eq-self in-diff-count in-replicate-mset minus-eq-id-forall-notin-mset minus-multiset.rep-eq mset-subset-eqD nat-less-le subset-mset.trans subseteq-mset-def*)

lemma *remdups-mset-removeAll*: ‹*remdups-mset* (*removeAll-mset a A*) = *removeAll-mset a* (*remdups-mset A*)›
by (*smt* (*verit*, *ccfv-threshold*) *add-mset-remove-trivial count-eq-zero-iff diff-zero distinct-mset-remdups-mset distinct-mset-remove1-All insert-DiffM order.refl remdups-mset-def remdups-mset-singleton-sum removeAll-subseteq-remove1-mset replicate-mset-eq-empty-iff set-mset-minus-replicate-mset*(*1*) *set-mset-remdups-mset subset-mset-removeAll-iff*)

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

lemma *remdups-mset-singleton-removeAll*:
remdups-mset (*add-mset a A*) = *add-mset a* (*removeAll-mset a* (*remdups-mset A*))
by (*metis dual-order.refl finite-set-mset mset-set.insert-remove remdups-mset-def remdups-mset-removeAll set-mset-add-mset-insert set-mset-minus-replicate-mset*(*1*))

lemma *mset-remove-filtered*: ‹*C* − {#*x* ∈# *C*. *P x*#} = {#*x* ∈# *C*. ¬*P x*#}›
by (*metis add-implies-diff union-filter-mset-complement*)

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*›
by (*auto simp*: *fset.fset-inverse fset-mset-def*)

lemma *mset-fset-fset-mset*[*simp*]:
‹*mset-fset* (*fset-mset N*) = *remdups-mset N*›
by (*auto simp*: *fset.fset-inverse fset-mset-def Abs-fset-inverse remdups-mset-def*)

lemma *in-mset-fset-fmember*[*simp*]: ‹*x* ∈# *mset-fset N* ⟷ *x* |∈| *N*›
by (*auto simp*: *fmember.rep-eq*)

lemma *in-fset-mset-mset*[*simp*]: ‹*x* |∈| *fset-mset N* ⟷ *x* ∈# *N*›
by (*auto simp*: *fmember.rep-eq fset-mset-def Abs-fset-inverse*)

Finite map and multisets

Roughly the same as *ran* and *dom*, but with duplication in the content (unlike their finite sets counterpart) while still working on finite domains (unlike a function mapping). Remark that *dom-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*)›
unfolding *dom-m-def*
by (*cases* ‹*C* |∈| *fmdom N*›)
(*auto simp*: *mset-set.remove fmember.rep-eq*)

lemma *dom-m-fmdrop-All*: ‹*dom-m* (*fmdrop C N*) = *removeAll-mset C* (*dom-m N*)›
unfolding *dom-m-def*
by (*cases* ‹*C* |∈| *fmdom N*›)
(*auto simp*: *mset-set.remove fmember.rep-eq*)

lemma *dom-m-fmupd*[*simp*]: ‹*dom-m* (*fmupd k C N*) = *add-mset k* (*remove1-mset k* (*dom-m N*))›
unfolding *dom-m-def*
by (*cases* ‹*k* |∈| *fmdom N*›)
(*auto simp*: *mset-set.remove fmember.rep-eq mset-set.insert-remove*)

lemma *distinct-mset-dom*: ‹*distinct-mset* (*dom-m N*)›
by (*simp add*: *distinct-mset-mset-set dom-m-def*)

lemma *in-dom-m-lookup-iff*: ‹*C* ∈# *dom-m N'* ⟷ *fmlookup N' C* ≠ *None*›
by (*auto simp*: *dom-m-def fmdom.rep-eq fmlookup-dom'-iff*)

lemma *in-dom-in-ran-m*[*simp*]: ‹*i* ∈# *dom-m N* ⟹ *the* (*fmlookup N i*) ∈# *ran-m N*›
by (*auto simp*: *ran-m-def*)

lemma *fmupd-same*[*simp*]:
‹*x1* ∈# *dom-m x1aa* ⟹ *fmupd x1* (*the* (*fmlookup x1aa x1*)) *x1aa* = *x1aa*›
by (*metis fmap-ext fmupd-lookup in-dom-m-lookup-iff option.collapse*)

lemma *ran-m-fmempty*[*simp*]: ‹*ran-m fmempty* = {#}› **and**
dom-m-fmempty[*simp*]: ‹*dom-m fmempty* = {#}›
by (*auto simp*: *ran-m-def dom-m-def*)

lemma *fmrestrict-set-fmupd*:
‹*a* ∈ *xs* ⟹ *fmrestrict-set xs* (*fmupd a C N*) = *fmupd a C* (*fmrestrict-set xs N*)›
‹*a* ∉ *xs* ⟹ *fmrestrict-set xs* (*fmupd a C N*) = *fmrestrict-set xs N*›
by (*auto simp*: *fmfilter-alt-defs*)

lemma *fset-fmdom-fmrestrict-set*:
‹*fset* (*fmdom* (*fmrestrict-set xs N*)) = *fset* (*fmdom N*) ∩ *xs*›
by (*auto simp*: *fmfilter-alt-defs*)

lemma *dom-m-fmrestrict-set*: ‹*dom-m* (*fmrestrict-set* (*set xs*) *N*) = *mset xs* ∩# *dom-m N*›
using *fset-fmdom-fmrestrict-set*[*of* ‹*set xs*› *N*] *distinct-mset-dom*[*of N*]

distinct-mset-inter-remdups-mset[*of* ‹*mset-fset* (*fmdom N*)› ‹*mset xs*›]
by (*auto simp*: *dom-m-def fset-mset-mset-fset finite-mset-set-inter multiset-inter-commute remdups-mset-def*)

lemma *dom-m-fmrestrict-set′*: ‹*dom-m* (*fmrestrict-set xs N*) = *mset-set* (*xs* ∩ *set-mset* (*dom-m N*))›
using *fset-fmdom-fmrestrict-set*[*of* ‹*xs*› *N*] *distinct-mset-dom*[*of N*]
by (*auto simp*: *dom-m-def fset-mset-mset-fset finite-mset-set-inter multiset-inter-commute remdups-mset-def*)

lemma *indom-mI*: ‹*fmlookup m x* = *Some y* ⟹ *x* ∈# *dom-m m*›
by (*drule fmdomI*) (*auto simp*: *dom-m-def fmember.rep-eq*)

lemma *fmupd-fmdrop-id*:
assumes ‹*k* |∈| *fmdom N′*›
shows ‹*fmupd k* (*the* (*fmlookup N′ k*)) (*fmdrop k N′*) = *N′*›
proof −
have [*simp*]: ‹*map-upd k* (*the* (*fmlookup N′ k*))
(λ*x*. *if x* ≠ *k then fmlookup N′ x else None*) =
map-upd k (*the* (*fmlookup N′ k*))
(*fmlookup N′*)›
by (*auto intro*!: *ext simp*: *map-upd-def*)
have [*simp*]: ‹*map-upd k* (*the* (*fmlookup N′ k*)) (*fmlookup N′*) = *fmlookup N′*›
using *assms*
by (*auto intro*!: *ext simp*: *map-upd-def*)
have [*simp*]: ‹*finite* (*dom* (λ*x*. *if x* = *k then None else fmlookup N′ x*))›
by (*subst dom-if*) *auto*
show *?thesis*
apply (*auto simp*: *fmupd-def fmupd.abs-eq*[*symmetric*])
unfolding *fmlookup-drop*
apply (*simp add*: *fmlookup-inverse*)
done
qed

lemma *fm-member-split*: ‹*k* |∈| *fmdom N′* ⟹ ∃ *N′′ v*. *N′* = *fmupd k v N′′* ∧ *the* (*fmlookup N′ k*) = *v* ∧
k |∉| *fmdom N′′*›
by (*rule exI*[*of* - ‹*fmdrop k N′*›])
(*auto simp*: *fmupd-fmdrop-id*)

lemma ‹*fmdrop k* (*fmupd k va N′′*) = *fmdrop k N′′*›
by (*simp add*: *fmap-ext*)

lemma *fmap-ext-fmdom*:
‹(*fmdom N* = *fmdom N′*) ⟹ (⋀ *x*. *x* |∈| *fmdom N* ⟹ *fmlookup N x* = *fmlookup N′ x*) ⟹
N = *N′*›
by (*rule fmap-ext*)
(*case-tac* ‹*x* |∈| *fmdom N*›, *auto simp*: *fmdom-notD*)

lemma *fmrestrict-set-insert-in*:
‹*xa* ∈ *fset* (*fmdom N*) ⟹
fmrestrict-set (*insert xa l1*) *N* = *fmupd xa* (*the* (*fmlookup N xa*)) (*fmrestrict-set l1 N*)›
apply (*rule fmap-ext-fmdom*)
apply (*auto simp*: *fset-fmdom-fmrestrict-set fmember.rep-eq notin-fset*; *fail*)[]
apply (*auto simp*: *fmlookup-dom-iff*; *fail*)
done

lemma *fmrestrict-set-insert-notin*:
‹*xa* ∉ *fset* (*fmdom N*) ⟹
fmrestrict-set (*insert xa l1*) *N* = *fmrestrict-set l1 N*›
by (*rule fmap-ext-fmdom*)
(*auto simp*: *fset-fmdom-fmrestrict-set fmember.rep-eq notin-fset*)

lemma *fmrestrict-set-insert-in-dom-m*[*simp*]:
‹*xa* ∈# *dom-m N* ⟹
fmrestrict-set (*insert xa l1*) *N* = *fmupd xa* (*the* (*fmlookup N xa*)) (*fmrestrict-set l1 N*)›
by (*simp add*: *fmrestrict-set-insert-in dom-m-def*)

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

lemma *fmlookup-restrict-set-id*: ‹*fset* (*fmdom N*) ⊆ *A* ⟹ *fmrestrict-set A N* = *N*›
by (*metis fmap-ext fmdom′-alt-def fmdom′-notD fmlookup-restrict-set subset-iff*)

lemma *fmlookup-restrict-set-id′*: ‹*set-mset* (*dom-m N*) ⊆ *A* ⟹ *fmrestrict-set A N* = *N*›
by (*rule fmlookup-restrict-set-id*)
(*auto simp*: *dom-m-def*)

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 −
define *N′* **where**
‹*N′* = *fmdrop C N*›
have *N-N′*: ‹*dom-m N* = *add-mset C* (*dom-m N′*)›
using *NC* **unfolding** *N′-def* **by** *auto*
have ‹*C* ∉# *dom-m N′*›
using *NC distinct-mset-dom*[*of N*] **unfolding** *N-N′* **by** *auto*
then show *?thesis*
by (*auto simp*: *N-N′ ran-m-def mset-set.insert-remove image-mset-remove1-mset-if*
intro!: *image-mset-cong*)
qed

lemma *ran-m-mapsto-upd-notin*:
assumes *NC*: ‹*C* ∉# *dom-m N*›
shows ‹*ran-m* (*fmupd C C′ N*) = *add-mset C′* (*ran-m N*)›
using *NC*
by (*auto simp*: *ran-m-def mset-set.insert-remove image-mset-remove1-mset-if*
intro!: *image-mset-cong split*: *if-splits*)

lemma *ran-m-fmdrop*:
‹*C* ∈# *dom-m N* ⟹ *ran-m* (*fmdrop C N*) = *remove1-mset* (*the* (*fmlookup N C*)) (*ran-m N*)›
using *distinct-mset-dom*[*of N*]
by (*cases* ‹*fmlookup N C*›)
(*auto simp*: *ran-m-def image-mset-If-eq-notin*[*of C* - ‹λ*x*. *fst* (*the x*)›]
dest!: *multi-member-split*
intro!: *filter-mset-cong2 image-mset-cong2*)

lemma *ran-m-fmdrop-notin*:

‹*C* ∉# *dom-m N* ⟹ *ran-m* (*fmdrop C N*) = *ran-m N*›
using *distinct-mset-dom*[*of N*]
by (*auto simp*: *ran-m-def image-mset-If-eq-notin*[*of C* - ‹λ*x*. *fst* (*the x*)›]
dest!: *multi-member-split*
intro!: *filter-mset-cong2 image-mset-cong2*)

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*)›
using *distinct-mset-dom*[*of N*]
by (*auto simp*: *ran-m-def image-mset-If-eq-notin*[*of C* - ‹λ*x*. *fst* (*the x*)›]
dest!: *multi-member-split*
intro!: *filter-mset-cong2 image-mset-cong2*)

Compact domain for finite maps

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

definition *Max-dom* **where**
‹*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*›
by (*auto simp*: *Max-dom-def*)

lemma *Max-dom-fmempty*: ‹*Max-dom fmempty* = *0*›
by (*auto simp*: *Max-dom-empty*)

lemma *packed-empty*[*simp*]: ‹*packed fmempty*›
by (*auto simp*: *packed-def Max-dom-empty*)

lemma *packed-Max-dom-size*:
assumes *p*: ‹*packed N*›
shows ‹*Max-dom N* = *size* (*dom-m N*)›
proof −
have *1*: ‹*dom-m N* = *mset* [*1..*<*Suc* (*Max-dom N*)]›
using *p* **unfolding** *packed-def Max-dom-def*[*symmetric*] **.**
have ‹*size* (*dom-m N*) = *size* (*mset* [*1..*<*Suc* (*Max-dom N*)])›
unfolding *1* **..**
also have ‹. . . = *length* [*1..*<*Suc* (*Max-dom N*)]›
unfolding *size-mset* **..**
also have ‹. . . = *Max-dom N*›
unfolding *length-upt* **by** *simp*
finally show *?thesis*
by *simp*
qed

lemma *Max-dom-le*:

‹L ∈# dom-m N ⟹ L ≤ Max-dom N›
by (auto simp: Max-dom-def)

lemma *remove1-mset-ge-Max-some*: ‹a > Max-dom b ⟹ remove1-mset a (dom-m b) = dom-m b›
by (*auto simp*: *Max-dom-def remove-1-mset-id-iff-notin*
dest!: *multi-member-split*)

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 –
have [*simp*]: ‹max 0 (max a A) = max a A› **for** *A*
using *assms*
by (*auto simp*: *Max-dom-def remove1-mset-ge-Max-some ac-simps*
Max.insert-remove split: *if-splits*)
have [*iff*]: *max a A = a* ⟷ (*A* ≤ *a*) **for** *A*
by (*auto split*: *if-splits simp*: *max-def*)

show *?thesis*
using *assms*
apply (*cases* ‹dom-m M›)
apply (*auto simp*: *Max-dom-def remove1-mset-ge-Max-some ac-simps*)[]
apply (*auto simp*: *Max-dom-def remove1-mset-ge-Max-some ac-simps*)
using *order-less-imp-le* **apply** *blast*
by (*meson in-diffD less-le-not-le*)
qed

lemma *Max-dom-alt-def*: ‹Max-dom b = Max (insert 0 (set-mset (dom-m b)))›
unfolding *Max-dom-def* **by** *auto*

lemma *Max-insert-Suc-Max-dim-dom*[*simp*]:
‹Max (insert (Suc (Max-dom b)) (set-mset (dom-m b))) = Suc (Max-dom b)›
unfolding *Max-dom-alt-def*
by (*cases* ‹set-mset (dom-m b) = {}›) *auto*

lemma *size-dom-m-Max-dom*:
‹size (dom-m N) ≤ Suc (Max-dom N)›
proof –
have ‹dom-m N ⊆# mset-set {0..< Suc (Max-dom N)}›
apply (*rule distinct-finite-set-mset-subseteq-iff*[*THEN iffD1*])
subgoal by (*auto simp*: *distinct-mset-dom*)
subgoal by *auto*
subgoal by (*auto dest*: *Max-dom-le*)
done
from *size-mset-mono*[*OF this*] **show** *?thesis* **by** *auto*
qed

lemma *Max-atLeastLessThan-plus*: ‹Max {(a::nat) ..< a+n} = (if n = 0 then Max {} else a+n − 1)›
apply (*induction n arbitrary*: *a*)
subgoal by *auto*
subgoal for *n a*
by (*cases n*)
(*auto simp*: *image-Suc-atLeastLessThan*[*symmetric*] *mono-Max-commute*[*symmetric*] *mono-def*
atLeastLessThanSuc
simp del: *image-Suc-atLeastLessThan*)

done

lemma *Max-atLeastLessThan*: ‹*Max* {(*a*::*nat*) ..< *b*} = (*if b* ≤ *a then Max* {} *else b* − *1*)›
using *Max-atLeastLessThan-plus*[*of a* ‹*b*−*a*›]
by *auto*

lemma *Max-insert-Max-dom-into-packed*:
‹*Max* (*insert* (*Max-dom bc*) {*Suc 0*..<*Max-dom bc*}) = *Max-dom bc*›
by (*cases* ‹*Max-dom bc*›; *cases* ‹*Max-dom bc* − *1*›)
(*auto simp*: *Max-dom-empty Max-atLeastLessThan*)

lemma *packed0-fmud-Suc-Max-dom*: ‹*packed b* ⟹ *packed* (*fmupd* (*Suc* (*Max-dom b*)) *C b*)›
by (*auto simp*: *packed-def remove1-mset-ge-Max-some Max-dom-fmupd-irrel max-def*)

lemma *ge-Max-dom-notin-dom-m*: ‹*a* > *Max-dom ao* ⟹ *a* ∉# *dom-m ao*›
by (*auto simp*: *Max-dom-def*)

lemma *packed-in-dom-mI*: ‹*packed bc* ⟹ *j* ≤ *Max-dom bc* ⟹ *0* < *j* ⟹ *j* ∈# *dom-m bc*›
by (*auto simp*: *packed-def*)

lemma *mset-fset-empty-iff*: ‹*mset-fset a* = {#} ⟷ *a* = *fempty*›
by (*cases a*) (*auto simp*: *mset-set-empty-iff*)

lemma *dom-m-empty-iff*[*iff*]:
‹*dom-m NU* = {#} ⟷ *NU* = *fmempty*›
by (*cases NU*) (*auto simp*: *dom-m-def mset-fset-empty-iff mset-set.insert-remove*)

lemma *nat-power-div-base*:
fixes *k* :: *nat*
assumes *0* < *m 0* < *k*
shows *k* ^ *m div k* = (*k*::*nat*) ^ (*m* − *Suc 0*)
proof −
have *eq*: *k* ^ *m* = *k* ^ ((*m* − *Suc 0*) + *Suc 0*)
by (*simp add*: *assms*)
show *?thesis*
using *assms* **by** (*simp only*: *power-add eq*) *auto*
qed

lemma *eq-insertD*: ‹*A* = *insert a B* ⟹ *a* ∈ *A* ∧ *B* ⊆ *A*›
by *auto*

lemma *length-list-ge2*: ‹*length S* ≥ *2* ⟷ (∃ *a b S*′. *S* = [*a*, *b*] @ *S*′)›
apply (*cases S*)
apply (*simp*; *fail*)
apply (*rename-tac a S*′)
apply (*case-tac S*′)
by *simp-all*

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*))›
  by (*auto simp*: *comp-fun-commute-def add-mset-commute intro*!: *ext*)

lemma *Pow-mset-alt-def*:
  Pow-mset (*mset A*) = *mset* '# *mset* (*subseqs A*)
  apply (*induction A*)
  subgoal by (*auto simp*: *Pow-mset-def*)
  subgoal
    by (*auto simp*: *Let-def Pow-mset-def*)
  done

lemma *Pow-mset-empty*[*simp*]:
  ‹*Pow-mset* {#} = {#{#}#}›
  by (*auto simp*: *Pow-mset-def*)

lemma *Pow-mset-add-mset*[*simp*]:
  ‹*Pow-mset* (*add-mset a A*) = *Pow-mset A* + (*add-mset a*) '# *Pow-mset A*›
  by (*auto simp*: *Let-def Pow-mset-def*)

lemma *in-Pow-mset-iff*:
  ‹*A* ∈# *Pow-mset B* ⟷ *A* ⊆# *B*›
proof
  assume ‹*A* ⊆# *B*›
  then show ‹*A* ∈# *Pow-mset B*›
    apply (*induction B arbitrary*: *A*)
    subgoal by *auto*
    subgoal premises *p* **for** *b B A*
      using *p*(*1*)[*of A*] *p*(*1*)[*of* ‹*A* − {#*b*#}›] *p*(*2*)
      apply (*cases* ‹*b* ∈# *A*›)
      by (*auto dest*: *subset-add-mset-notin-subset-mset*
          dest!: *multi-member-split*)
    done
next
  assume ‹*A* ∈# *Pow-mset B*›
  then show ‹*A* ⊆# *B*›
    apply (*induction B arbitrary*: *A*)
    subgoal by *auto*
    subgoal premises *p* **for** *b B A*
      using *p* **by** (*auto simp*: *subset-mset-trans-add-mset*)
    done
qed

lemma *size-Pow-mset*[*simp*]: ‹*size* (*Pow-mset A*) = *2*^(*size A*)›
  by (*induction A*) *auto*

lemma *set-Pow-mset*:
  ‹*set-mset* (*Pow-mset A*) = {*B*. *B* ⊆# *A*}›
  by (*auto simp*: *in-Pow-mset-iff*)

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 −
  define f :: 'a multiset ⇒ 'a ⇒ nat
    where f = (λB x. if x ∈# A then count B x else undefined)
  define g :: ('a ⇒ nat) ⇒ 'a multiset
    where g = (λh. Abs-multiset (λx. if x ∈# A then h x else 0))

  have count-g: count (g h) x = (if x ∈# A then h x else 0)
    if h ∈ (Π_E x∈set-mset A. {0..count A x}) for h x
  proof −
    have finite {x. (if x ∈# A then h x else 0) > 0}
      by (rule finite-subset[of - set-mset A]) (use that in auto)
    thus ?thesis
      using g-def by auto
  qed

  have f: f B ∈ (Π_E x∈set-mset A. {0..count A x}) if B ⊆# A for B
      using that by (auto simp: f-def subseteq-mset-def)

  have bij-betw f {B. B ⊆# A} (Π_E x∈set-mset A. {0..count A x})
  proof (rule bij-betwI[where g = g], goal-cases)
    case 1
    thus ?case using f by auto
  next
    case 2
    show ?case
      by (auto simp: Pi-def PiE-def count-g subseteq-mset-def)
  next
    case (3 B)
    have count (g (f B)) x = count B x for x
    proof −
      have count (g (f B)) x = (if x ∈# A then f B x else 0)
        using f 3 by (simp add: count-g)
      also have ... = count B x
        using 3 by (auto simp: f-def)
      finally show ?thesis .
    qed
    thus ?case
      by (auto simp: multiset-eq-iff)
  next
    case 4
    thus ?case
      by (auto simp: fun-eq-iff f-def count-g)
  qed
  hence card {B. B ⊆# A} = card (Π_E x∈set-mset A. {0..count A x})
    using bij-betw-same-card by blast
  thus ?thesis
    by (simp add: card-PiE set-Pow-mset)
qed

lemma empty-in-Pow-mset[iff]: ‹{#} ∈# Pow-mset B›
  by (induction B) auto
```

lemma *full-in-Pow-mset*[*iff*]: ‹*B* ∈# *Pow-mset B*›
  by (*induction B*) *auto*

lemma *Pow-mset-nempty*[*iff*]: ‹*Pow-mset B* ≠ {#}›
  using *full-in-Pow-mset*[*of B*] **by** *force*

lemma *Pow-mset-single-empty*[*iff*]: ‹*Pow-mset B* = {#{#}#} ⟷ *B* = {#}›
  using *full-in-Pow-mset*[*of B*] **by** *fastforce*

lemma *Pow-mset-mono*: ‹*A* ⊆# *B* ⟹ *Pow-mset A* ⊆# *Pow-mset B*›
  apply (*induction A arbitrary*: *B*)
  subgoal by *auto*
  subgoal premises *p* **for** *x A B*
    using *p*(*1*)[*of* ‹*remove1-mset x B*›] *p*(*2*)
    by (*cases* ‹*x*∈#*B*›)
     (*auto dest*!: *multi-member-split*
      simp add: *image-mset-subseteq-mono subset-mset.add-mono*)
  done

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* = []›
  by (*auto simp*: *option-hd-def*)

lemma *option-hd-Some-iff*[*iff*]: ‹*option-hd zs* = *Some y* ⟷ (*zs* ≠ [] ∧ *y* = *hd zs*)›
  ‹*Some y* = *option-hd zs* ⟷ (*zs* ≠ [] ∧ *y* = *hd zs*)›
  by (*auto simp*: *option-hd-def*)

lemma *option-hd-Some-hd*[*simp*]: ‹*zs* ≠ [] ⟹ *option-hd zs* = *Some* (*hd zs*)›
  by (*auto simp*: *option-hd-def*)

lemma *option-hd-Nil*[*simp*]: ‹*option-hd* [] = *None*›
  by (*auto simp*: *option-hd-def*)

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*›
  by (*auto simp*: *option-last-def*)

lemma *option-last-Some*[*simp*]: ‹*l* ≠ [] ⟹ *option-last l* = *Some* (*last l*)›
  by (*auto simp*: *option-last-def*)

lemma *option-last-Nil*[*simp*]: ‹*option-last* [] = *None*›
  by (*auto simp*: *option-last-def*)

lemma *option-last-remove1-not-last*:
  ‹*x* ≠ *last xs* ⟹ *option-last xs* = *option-last* (*remove1 x xs*)›

by (*cases xs rule*: *rev-cases*)
(*auto simp*: *option-last-def remove1-Nil-iff remove1-append*)

lemma *option-hd-rev*: ‹*option-hd* (*rev xs*) = *option-last xs*›
by (*cases xs rule*: *rev-cases*) *auto*

lemma *map-option-option-last*:
‹*map-option f* (*option-last xs*) = *option-last* (*map f xs*)›
by (*cases xs rule*: *rev-cases*) *auto*

end