Euler's Partition Theorem

Lukas Bulwahn

December 17, 2016

Abstract

Euler's Partition Theorem states that the number of partitions with only distinct parts is equal to the number of partitions with only odd parts. The combinatorial proof follows John Harrison's pre-existing HOL Light formalization [1]. To understand the rough idea of the proof, I read the lecture notes of the MIT course 18.312 on Algebraic Combinatorics [2] by Gregg Musiker. This theorem is the 45th theorem of the Top 100 Theorems list.

Contents

1 (Finite) multisets

theory *Multiset*
imports *Main*
begin

1.1 The type of multisets

definition *multiset* = $\{f :: 'a \Rightarrow nat.\ finite\ \{x.\ f\ x > 0\}\}$

typedef $'a$ *multiset* = *multiset* :: $('a \Rightarrow nat)$ *set*
  morphisms *count Abs-multiset*
  unfolding *multiset-def*
proof
  show $(\lambda x.\ 0::nat) \in \{f.\ finite\ \{x.\ f\ x > 0\}\}$ **by** *simp*
qed

setup-lifting *type-definition-multiset*

lemma *multiset-eq-iff*: $M = N \longleftrightarrow (\forall a.\ count\ M\ a = count\ N\ a)$
  by (*simp only*: *count-inject* [*symmetric*] *fun-eq-iff*)

lemma *multiset-eqI*: $(\bigwedge x.\ count\ A\ x = count\ B\ x) \Longrightarrow A = B$
  using *multiset-eq-iff* **by** *auto*

Preservation of the representing set *multiset*.

lemma *const0-in-multiset*: $(\lambda a.\ 0) \in multiset$
  by (*simp add*: *multiset-def*)

lemma *only1-in-multiset*: $(\lambda b.\ if\ b = a\ then\ n\ else\ 0) \in multiset$
  by (*simp add*: *multiset-def*)

lemma *union-preserves-multiset*: $M \in multiset \Longrightarrow N \in multiset \Longrightarrow (\lambda a.\ M\ a + N\ a) \in multiset$
  by (*simp add*: *multiset-def*)

lemma *diff-preserves-multiset*:
  assumes $M \in multiset$
  shows $(\lambda a.\ M\ a - N\ a) \in multiset$
proof −
  have $\{x.\ N\ x < M\ x\} \subseteq \{x.\ 0 < M\ x\}$
    by *auto*
  with *assms* **show** *?thesis*
    by (*auto simp add*: *multiset-def intro*: *finite-subset*)
qed

lemma *filter-preserves-multiset*:
  assumes $M \in multiset$
  shows $(\lambda x.\ if\ P\ x\ then\ M\ x\ else\ 0) \in multiset$
proof −

have $\{x.\ (P\ x \longrightarrow 0 < M\ x) \wedge P\ x\} \subseteq \{x.\ 0 < M\ x\}$
by *auto*
with *assms* **show** *?thesis*
by (*auto simp add*: *multiset-def intro*: *finite-subset*)
qed

lemmas *in-multiset* = *const0-in-multiset only1-in-multiset*
union-preserves-multiset diff-preserves-multiset filter-preserves-multiset

1.2 Representing multisets

Multiset enumeration

instantiation *multiset* :: (*type*) *cancel-comm-monoid-add*
begin

lift-definition *zero-multiset* :: *'a multiset* **is** $\lambda a.\ 0$
by (*rule const0-in-multiset*)

abbreviation *Mempty* :: *'a multiset* ($\{\#\}$) **where**
Mempty $\equiv$ *0*

lift-definition *plus-multiset* :: *'a multiset* $\Rightarrow$ *'a multiset* $\Rightarrow$ *'a multiset* **is** $\lambda M\ N.$
$(\lambda a.\ M\ a + N\ a)$
by (*rule union-preserves-multiset*)

lift-definition *minus-multiset* :: *'a multiset* $\Rightarrow$ *'a multiset* $\Rightarrow$ *'a multiset* **is** $\lambda\ M$
$N.\ \lambda a.\ M\ a - N\ a$
by (*rule diff-preserves-multiset*)

instance
by (*standard*; *transfer*; *simp add*: *fun-eq-iff*)

end

context
begin

qualified definition *is-empty* :: *'a multiset* $\Rightarrow$ *bool* **where**
[*code-abbrev*]: *is-empty A* $\longleftrightarrow$ *A* = $\{\#\}$

end

lemma *add-mset-in-multiset*:
assumes *M*: ‹*M* $\in$ *multiset*›
shows ‹$(\lambda b.$ *if b* = *a then Suc* (*M b*) *else M b*) $\in$ *multiset*›
using *assms* **by** (*simp add*: *multiset-def insert-Collect*[*symmetric*])

lift-definition *add-mset* :: *'a* $\Rightarrow$ *'a multiset* $\Rightarrow$ *'a multiset* **is**
$\lambda a\ M\ b.$ *if b* = *a then Suc* (*M b*) *else M b*

by (*rule add-mset-in-multiset*)

syntax
-multiset :: *args* ⇒ *'a multiset* ({#(-)#})
translations
{#*x*, *xs*#} == *CONST add-mset x* {#*xs*#}
{#*x*#} == *CONST add-mset x* {#}

lemma *count-empty* [*simp*]: *count* {#} *a* = *0*
by (*simp add*: *zero-multiset.rep-eq*)

lemma *count-add-mset* [*simp*]:
count (*add-mset b A*) *a* = (*if b* = *a then Suc* (*count A a*) *else count A a*)
by (*simp add*: *add-mset.rep-eq*)

lemma *count-single*: *count* {#*b*#} *a* = (*if b* = *a then 1 else 0*)
by *simp*

lemma
add-mset-not-empty [*simp*]: ‹*add-mset a A* ≠ {#}› **and**
empty-not-add-mset [*simp*]: {#} ≠ *add-mset a A*
by (*auto simp*: *multiset-eq-iff*)

lemma *add-mset-add-mset-same-iff* [*simp*]:
add-mset a A = *add-mset a B* ⟷ *A* = *B*
by (*auto simp*: *multiset-eq-iff*)

lemma *add-mset-commute*:
add-mset x (*add-mset y M*) = *add-mset y* (*add-mset x M*)
by (*auto simp*: *multiset-eq-iff*)

1.3 Basic operations

1.3.1 Conversion to set and membership

definition *set-mset* :: *'a multiset* ⇒ *'a set*
where *set-mset M* = {*x*. *count M x* > *0*}

abbreviation *Melem* :: *'a* ⇒ *'a multiset* ⇒ *bool*
where *Melem a M* ≡ *a* ∈ *set-mset M*

notation
Melem (*op* ∈#) **and**
Melem ((-/ ∈# -) [*51*, *51*] *50*)

notation (*ASCII*)
Melem (*op* :#) **and**
Melem ((-/ :# -) [*51*, *51*] *50*)

abbreviation *not-Melem* :: *'a* ⇒ *'a multiset* ⇒ *bool*

where *not-Melem a M* ≡ *a* ∉ *set-mset M*

notation
not-Melem (*op* ∉#) **and**
not-Melem ((-/ ∉# -) [*51*, *51*] *50*)

notation (*ASCII*)
not-Melem (*op* ~:#) **and**
not-Melem ((-/ ~:# -) [*51*, *51*] *50*)

context
begin

qualified abbreviation *Ball* :: *'a multiset* ⇒ (*'a* ⇒ *bool*) ⇒ *bool*
where *Ball M* ≡ *Set.Ball* (*set-mset M*)

qualified abbreviation *Bex* :: *'a multiset* ⇒ (*'a* ⇒ *bool*) ⇒ *bool*
where *Bex M* ≡ *Set.Bex* (*set-mset M*)

end

syntax
-MBall :: *pttrn* ⇒ *'a set* ⇒ *bool* ⇒ *bool* ((*3*∀ -∈#-./ -) [*0*, *0*, *10*] *10*)
-MBex :: *pttrn* ⇒ *'a set* ⇒ *bool* ⇒ *bool* ((*3*∃ -∈#-./ -) [*0*, *0*, *10*] *10*)

syntax (*ASCII*)
-MBall :: *pttrn* ⇒ *'a set* ⇒ *bool* ⇒ *bool* ((*3*∀ -:#-./ -) [*0*, *0*, *10*] *10*)
-MBex :: *pttrn* ⇒ *'a set* ⇒ *bool* ⇒ *bool* ((*3*∃ -:#-./ -) [*0*, *0*, *10*] *10*)

translations
∀ *x*∈#*A*. *P* ⇌ *CONST Multiset.Ball A* (λ*x*. *P*)
∃ *x*∈#*A*. *P* ⇌ *CONST Multiset.Bex A* (λ*x*. *P*)

lemma *count-eq-zero-iff*:
count M x = *0* ⟷ *x* ∉# *M*
by (*auto simp add*: *set-mset-def*)

lemma *not-in-iff*:
x ∉# *M* ⟷ *count M x* = *0*
by (*auto simp add*: *count-eq-zero-iff*)

lemma *count-greater-zero-iff* [*simp*]:
count M x > *0* ⟷ *x* ∈# *M*
by (*auto simp add*: *set-mset-def*)

lemma *count-inI*:
assumes *count M x* = *0* ⟹ *False*
shows *x* ∈# *M*
proof (*rule ccontr*)

assume $x \notin\# M$
with *assms* **show** *False* **by** (*simp add*: *not-in-iff*)
qed

lemma *in-countE*:
assumes $x \in\# M$
obtains n **where** *count* $M\ x$ = *Suc* n
proof −
from *assms* **have** *count* $M\ x > 0$ **by** *simp*
then obtain n **where** *count* $M\ x$ = *Suc* n
using *gr0-conv-Suc* **by** *blast*
with *that* **show** *thesis* **.**
qed

lemma *count-greater-eq-Suc-zero-iff* [*simp*]:
count $M\ x \geq$ *Suc* $0 \longleftrightarrow x \in\# M$
by (*simp add*: *Suc-le-eq*)

lemma *count-greater-eq-one-iff* [*simp*]:
count $M\ x \geq 1 \longleftrightarrow x \in\# M$
by *simp*

lemma *set-mset-empty* [*simp*]:
set-mset $\{\#\} = \{\}$
by (*simp add*: *set-mset-def*)

lemma *set-mset-single*:
set-mset $\{\#b\#\} = \{b\}$
by (*simp add*: *set-mset-def*)

lemma *set-mset-eq-empty-iff* [*simp*]:
set-mset $M = \{\} \longleftrightarrow M = \{\#\}$
by (*auto simp add*: *multiset-eq-iff count-eq-zero-iff*)

lemma *finite-set-mset* [*iff*]:
finite (*set-mset* M)
using *count* [*of* M] **by** (*simp add*: *multiset-def*)

lemma *set-mset-add-mset-insert* [*simp*]: ‹*set-mset* (*add-mset a A*) = *insert a* (*set-mset* *A*)›
by (*auto simp del*: *count-greater-eq-Suc-zero-iff*
simp: *count-greater-eq-Suc-zero-iff*[*symmetric*] *split*: *if-splits*)

lemma *multiset-nonemptyE* [*elim*]:
assumes $A \neq \{\#\}$
obtains x **where** $x \in\# A$
proof −
have $\exists x.\ x \in\# A$ **by** (*rule ccontr*) (*insert assms*, *auto*)
with *that* **show** *?thesis* **by** *blast*

qed

1.3.2 Union

lemma *count-union* [*simp*]:
count (*M* + *N*) *a* = *count M a* + *count N a*
by (*simp add*: *plus-multiset.rep-eq*)

lemma *set-mset-union* [*simp*]:
set-mset (*M* + *N*) = *set-mset M* ∪ *set-mset N*
by (*simp only*: *set-eq-iff count-greater-zero-iff* [*symmetric*] *count-union*) *simp*

lemma *union-mset-add-mset-left* [*simp*]:
add-mset a A + *B* = *add-mset a* (*A* + *B*)
by (*auto simp*: *multiset-eq-iff*)

lemma *union-mset-add-mset-right* [*simp*]:
A + *add-mset a B* = *add-mset a* (*A* + *B*)
by (*auto simp*: *multiset-eq-iff*)

lemma *add-mset-add-single*: ‹*add-mset a A* = *A* + {#*a*#}›
by (*subst union-mset-add-mset-right*, *subst add.comm-neutral*) *standard*

1.3.3 Difference

instance *multiset* :: (*type*) *comm-monoid-diff*
by *standard* (*transfer*; *simp add*: *fun-eq-iff*)

lemma *count-diff* [*simp*]:
count (*M* − *N*) *a* = *count M a* − *count N a*
by (*simp add*: *minus-multiset.rep-eq*)

lemma *add-mset-diff-bothsides*:
‹*add-mset a M* − *add-mset a A* = *M* − *A*›
by (*auto simp*: *multiset-eq-iff*)

lemma *in-diff-count*:
a ∈# *M* − *N* ⟷ *count N a* < *count M a*
by (*simp add*: *set-mset-def*)

lemma *count-in-diffI*:
assumes ⋀*n*. *count N x* = *n* + *count M x* ⟹ *False*
shows *x* ∈# *M* − *N*
proof (*rule ccontr*)
assume *x* ∉# *M* − *N*
then have *count N x* = (*count N x* − *count M x*) + *count M x*
by (*simp add*: *in-diff-count not-less*)
with *assms* **show** *False* **by** *auto*
qed

lemma *in-diff-countE*:
assumes $x \in\# M - N$
obtains n **where** $count\ M\ x = Suc\ n + count\ N\ x$
proof −
from *assms* **have** $count\ M\ x - count\ N\ x > 0$ **by** (*simp add*: *in-diff-count*)
then have $count\ M\ x > count\ N\ x$ **by** *simp*
then obtain n **where** $count\ M\ x = Suc\ n + count\ N\ x$
using *less-iff-Suc-add* **by** *auto*
with *that* **show** *thesis* **.**
qed

lemma *in-diffD*:
assumes $a \in\# M - N$
shows $a \in\# M$
proof −
have $0 \leq count\ N\ a$ **by** *simp*
also from *assms* **have** $count\ N\ a < count\ M\ a$
by (*simp add*: *in-diff-count*)
finally show *?thesis* **by** *simp*
qed

lemma *set-mset-diff*:
$set\text{-}mset\ (M - N) = \{a.\ count\ N\ a < count\ M\ a\}$
by (*simp add*: *set-mset-def*)

lemma *diff-empty* [*simp*]: $M - \{\#\} = M \wedge \{\#\} - M = \{\#\}$
by *rule* (*fact Groups.diff-zero*, *fact Groups.zero-diff*)

lemma *diff-cancel*: $A - A = \{\#\}$
by (*fact Groups.diff-cancel*)

lemma *diff-union-cancelR*: $M + N - N = (M::'a\ multiset)$
by (*fact add-diff-cancel-right'*)

lemma *diff-union-cancelL*: $N + M - N = (M::'a\ multiset)$
by (*fact add-diff-cancel-left'*)

lemma *diff-right-commute*:
fixes $M\ N\ Q :: 'a\ multiset$
shows $M - N - Q = M - Q - N$
by (*fact diff-right-commute*)

lemma *diff-add*:
fixes $M\ N\ Q :: 'a\ multiset$
shows $M - (N + Q) = M - N - Q$
by (*rule sym*) (*fact diff-diff-add*)

lemma *insert-DiffM* [*simp*]: $x \in\# M \Longrightarrow add\text{-}mset\ x\ (M - \{\#x\#\}) = M$
by (*clarsimp simp*: *multiset-eq-iff*)

lemma *insert-DiffM2*: $x \in\# M \Longrightarrow (M - \{\#x\#\}) + \{\#x\#\} = M$
by *simp*

lemma *diff-union-swap*: $a \neq b \Longrightarrow$ *add-mset* b $(M - \{\#a\#\}) =$ *add-mset* b $M - \{\#a\#\}$
by (*auto simp add*: *multiset-eq-iff*)

lemma *diff-add-mset-swap* [*simp*]: $b \notin\# A \Longrightarrow$ *add-mset* b $M - A =$ *add-mset* b $(M - A)$
by (*auto simp add*: *multiset-eq-iff simp*: *not-in-iff*)

lemma *diff-union-swap2* [*simp*]: $y \in\# M \Longrightarrow$ *add-mset* x $M - \{\#y\#\} =$ *add-mset* x $(M - \{\#y\#\})$
by (*metis add-mset-diff-bothsides diff-union-swap diff-zero insert-DiffM*)

lemma *diff-diff-add-mset* [*simp*]: $(M::'a\ multiset) - N - P = M - (N + P)$
by (*rule diff-diff-add*)

lemma *diff-union-single-conv*:
$a \in\# J \Longrightarrow I + J - \{\#a\#\} = I + (J - \{\#a\#\})$
by (*simp add*: *multiset-eq-iff Suc-le-eq*)

lemma *mset-add* [*elim?*]:
assumes $a \in\# A$
obtains B **where** $A =$ *add-mset* a B
proof −
from *assms* **have** $A =$ *add-mset* a $(A - \{\#a\#\})$
by *simp*
with *that* **show** *thesis* **.**
qed

lemma *union-iff*:
$a \in\# A + B \longleftrightarrow a \in\# A \vee a \in\# B$
by *auto*

1.3.4 Equality of multisets

lemma *single-eq-single* [*simp*]: $\{\#a\#\} = \{\#b\#\} \longleftrightarrow a = b$
by (*auto simp add*: *multiset-eq-iff*)

lemma *union-eq-empty* [*iff*]: $M + N = \{\#\} \longleftrightarrow M = \{\#\} \wedge N = \{\#\}$
by (*auto simp add*: *multiset-eq-iff*)

lemma *empty-eq-union* [*iff*]: $\{\#\} = M + N \longleftrightarrow M = \{\#\} \wedge N = \{\#\}$
by (*auto simp add*: *multiset-eq-iff*)

lemma *multi-self-add-other-not-self* [*simp*]: $M =$ *add-mset* x $M \longleftrightarrow$ *False*
by (*auto simp add*: *multiset-eq-iff*)

lemma *add-mset-remove-trivial* [*simp*]: ‹*add-mset x M* − {#*x*#} = *M*›
by (*auto simp*: *multiset-eq-iff*)

lemma *diff-single-trivial*: ¬ *x* ∈# *M* ⟹ *M* − {#*x*#} = *M*
by (*auto simp add*: *multiset-eq-iff not-in-iff*)

lemma *diff-single-eq-union*: *x* ∈# *M* ⟹ *M* − {#*x*#} = *N* ⟷ *M* = *add-mset x N*
by *auto*

lemma *union-single-eq-diff*: *add-mset x M* = *N* ⟹ *M* = *N* − {#*x*#}
unfolding *add-mset-add-single*[*of* - *M*] **by** (*fact add-implies-diff*)

lemma *union-single-eq-member*: *add-mset x M* = *N* ⟹ *x* ∈# *N*
by *auto*

lemma *add-mset-remove-trivial-If*:
add-mset a (*N* − {#*a*#}) = (*if a* ∈# *N then N else add-mset a N*)
by (*simp add*: *diff-single-trivial*)

lemma *add-mset-remove-trivial-eq*: ‹*N* = *add-mset a* (*N* − {#*a*#}) ⟷ *a* ∈# *N*›
by (*auto simp*: *add-mset-remove-trivial-If*)

lemma *union-is-single*:
M + *N* = {#*a*#} ⟷ *M* = {#*a*#} ∧ *N* = {#} ∨ *M* = {#} ∧ *N* = {#*a*#}
(**is** *?lhs* = *?rhs*)
proof
show *?lhs* **if** *?rhs* **using** *that* **by** *auto*
show *?rhs* **if** *?lhs*
by (*metis Multiset.diff-cancel add.commute add-diff-cancel-left' diff-add-zero diff-single-trivial insert-DiffM that*)
qed

lemma *single-is-union*: {#*a*#} = *M* + *N* ⟷ {#*a*#} = *M* ∧ *N* = {#} ∨ *M* = {#} ∧ {#*a*#} = *N*
by (*auto simp add*: *eq-commute* [*of* {#*a*#} *M* + *N*] *union-is-single*)

lemma *add-eq-conv-diff*:
add-mset a M = *add-mset b N* ⟷ *M* = *N* ∧ *a* = *b* ∨ *M* = *add-mset b* (*N* − {#*a*#}) ∧ *N* = *add-mset a* (*M* − {#*b*#})
(**is** *?lhs* ⟷ *?rhs*)

proof
show *?lhs* **if** *?rhs*
using *that*
by (*auto simp add*: *add-mset-commute*[*of a b*])
show *?rhs* **if** *?lhs*

```
  proof (cases a = b)
    case True with ‹?lhs› show ?thesis by simp
  next
    case False
    from ‹?lhs› have a ∈# add-mset b N by (rule union-single-eq-member)
    with False have a ∈# N by auto
   moreover from ‹?lhs› have M = add-mset b N − {#a#} by (rule union-single-eq-diff)
    moreover note False
   ultimately show ?thesis by (auto simp add: diff-right-commute [of - {#a#}])
  qed
qed

lemma add-mset-eq-single [iff]: add-mset b M = {#a#} ⟷ b = a ∧ M = {#}
  by (auto simp: add-eq-conv-diff)

lemma single-eq-add-mset [iff]: {#a#} = add-mset b M ⟷ b = a ∧ M = {#}
  by (auto simp: add-eq-conv-diff)

lemma insert-noteq-member:
  assumes BC: add-mset b B = add-mset c C
   and bnotc: b ≠ c
  shows c ∈# B
proof −
  have c ∈# add-mset c C by simp
  have nc: ¬ c ∈# {#b#} using bnotc by simp
  then have c ∈# add-mset b B using BC by simp
  then show c ∈# B using nc by simp
qed

lemma add-eq-conv-ex:
  (add-mset a M = add-mset b N) =
    (M = N ∧ a = b ∨ (∃ K. M = add-mset b K ∧ N = add-mset a K))
  by (auto simp add: add-eq-conv-diff)

lemma multi-member-split: x ∈# M ⟹ ∃ A. M = add-mset x A
  by (rule exI [where x = M − {#x#}]) simp

lemma multiset-add-sub-el-shuffle:
  assumes c ∈# B
    and b ≠ c
  shows add-mset b (B − {#c#}) = add-mset b B − {#c#}
proof −
  from ‹c ∈# B› obtain A where B: B = add-mset c A
    by (blast dest: multi-member-split)
  have add-mset b A = add-mset c (add-mset b A) − {#c#} by simp
  then have add-mset b A = add-mset b (add-mset c A) − {#c#}
    by (simp add: ‹b ≠ c›)
  then show ?thesis using B by simp
qed
```

lemma *add-mset-eq-singleton-iff*[*iff*]:
add-mset x M = {#*y*#} ⟷ *M* = {#} ∧ *x* = *y*
by *auto*

1.3.5 Pointwise ordering induced by count

definition *subseteq-mset* :: *'a multiset* ⇒ *'a multiset* ⇒ *bool* (**infix** ⊆# *50*)
where *A* ⊆# *B* = (∀ *a*. *count A a* ≤ *count B a*)

definition *subset-mset* :: *'a multiset* ⇒ *'a multiset* ⇒ *bool* (**infix** ⊂# *50*)
where *A* ⊂# *B* = (*A* ⊆# *B* ∧ *A* ≠ *B*)

abbreviation (*input*) *supseteq-mset* :: *'a multiset* ⇒ *'a multiset* ⇒ *bool* (**infix** ⊇# *50*)
where *supseteq-mset A B* ≡ *B* ⊆# *A*

abbreviation (*input*) *supset-mset* :: *'a multiset* ⇒ *'a multiset* ⇒ *bool* (**infix** ⊃# *50*)
where *supset-mset A B* ≡ *B* ⊂# *A*

notation (*input*)
subseteq-mset (**infix** ≤# *50*) **and**
supseteq-mset (**infix** ≥# *50*)

notation (*ASCII*)
subseteq-mset (**infix** <=# *50*) **and**
subset-mset (**infix** <# *50*) **and**
supseteq-mset (**infix** >=# *50*) **and**
supset-mset (**infix** ># *50*)

interpretation *subset-mset*: *ordered-ab-semigroup-add-imp-le op* + *op* − *op* ⊆# *op* ⊂#
by *standard* (*auto simp add*: *subset-mset-def subseteq-mset-def multiset-eq-iff intro*: *order-trans antisym*)

interpretation *subset-mset*: *ordered-ab-semigroup-monoid-add-imp-le op* + *0 op* − *op* ≤# *op* <#
by *standard*

lemma *mset-subset-eqI*:
(⋀*a*. *count A a* ≤ *count B a*) ⟹ *A* ⊆# *B*
by (*simp add*: *subseteq-mset-def*)

lemma *mset-subset-eq-count*:
A ⊆# *B* ⟹ *count A a* ≤ *count B a*
by (*simp add*: *subseteq-mset-def*)

lemma *mset-subset-eq-exists-conv*: (*A*::*'a multiset*) ⊆# *B* ⟷ (∃ *C*. *B* = *A* + *C*)

unfolding *subseteq-mset-def*
apply (*rule iffI*)
apply (*rule exI* [**where** $x = B - A$])
apply (*auto intro*: *multiset-eq-iff* [*THEN iffD2*])
done

interpretation *subset-mset*: *ordered-cancel-comm-monoid-diff op + 0 op* $\leq\#$ *op* $<\#$ *op* $-$
by *standard* (*simp*, *fact mset-subset-eq-exists-conv*)

declare *subset-mset.add-diff-assoc*[*simp*] *subset-mset.add-diff-assoc2*[*simp*]

lemma *mset-subset-eq-mono-add-right-cancel*: $(A::'a\ multiset) + C \subseteq\# B + C \longleftrightarrow A \subseteq\# B$
by (*fact subset-mset.add-le-cancel-right*)

lemma *mset-subset-eq-mono-add-left-cancel*: $C + (A::'a\ multiset) \subseteq\# C + B \longleftrightarrow A \subseteq\# B$
by (*fact subset-mset.add-le-cancel-left*)

lemma *mset-subset-eq-mono-add*: $(A::'a\ multiset) \subseteq\# B \Longrightarrow C \subseteq\# D \Longrightarrow A + C \subseteq\# B + D$
by (*fact subset-mset.add-mono*)

lemma *mset-subset-eq-add-left*: $(A::'a\ multiset) \subseteq\# A + B$
by *simp*

lemma *mset-subset-eq-add-right*: $B \subseteq\# (A::'a\ multiset) + B$
by *simp*

lemma *single-subset-iff* [*simp*]:
$\{\#a\#\} \subseteq\# M \longleftrightarrow a \in\# M$
by (*auto simp add*: *subseteq-mset-def Suc-le-eq*)

lemma *mset-subset-eq-single*: $a \in\# B \Longrightarrow \{\#a\#\} \subseteq\# B$
by *simp*

lemma *mset-subset-eq-add-mset-cancel*: ‹*add-mset a A* $\subseteq\#$ *add-mset a B* $\longleftrightarrow$ *A* $\subseteq\#$ *B*›
unfolding *add-mset-add-single*[*of - A*] *add-mset-add-single*[*of - B*]
by (*rule mset-subset-eq-mono-add-right-cancel*)

lemma *multiset-diff-union-assoc*:
fixes *A B C D* :: *'a multiset*
shows $C \subseteq\# B \Longrightarrow A + B - C = A + (B - C)$
by (*fact subset-mset.diff-add-assoc*)

lemma *mset-subset-eq-multiset-union-diff-commute*:
fixes *A B C D* :: *'a multiset*

shows $B \subseteq\# A \Longrightarrow A - B + C = A + C - B$
by (*fact subset-mset.add-diff-assoc2*)

lemma *diff-subset-eq-self*[*simp*]:
(M::'*a multiset*) − $N \subseteq\# M$
by (*simp add*: *subseteq-mset-def*)

lemma *mset-subset-eqD*:
assumes $A \subseteq\# B$ **and** $x \in\# A$
shows $x \in\# B$
proof −
from ‹$x \in\# A$› **have** *count A x* > *0* **by** *simp*
also from ‹$A \subseteq\# B$› **have** *count A x* ≤ *count B x*
by (*simp add*: *subseteq-mset-def*)
finally show *?thesis* **by** *simp*
qed

lemma *mset-subsetD*:
$A \subset\# B \Longrightarrow x \in\# A \Longrightarrow x \in\# B$
by (*auto intro*: *mset-subset-eqD* [*of A*])

lemma *set-mset-mono*:
$A \subseteq\# B \Longrightarrow$ *set-mset A* ⊆ *set-mset B*
by (*metis mset-subset-eqD subsetI*)

lemma *mset-subset-eq-insertD*:
add-mset x A $\subseteq\# B \Longrightarrow x \in\# B \wedge A \subset\# B$
apply (*rule conjI*)
apply (*simp add*: *mset-subset-eqD*)
apply (*clarsimp simp*: *subset-mset-def subseteq-mset-def*)
apply *safe*
apply (*erule-tac x* = *a* **in** *allE*)
apply (*auto split*: *if-split-asm*)
done

lemma *mset-subset-insertD*:
add-mset x A $\subset\# B \Longrightarrow x \in\# B \wedge A \subset\# B$
by (*rule mset-subset-eq-insertD*) *simp*

lemma *mset-subset-of-empty*[*simp*]: $A \subset\# \{\#\} \longleftrightarrow$ *False*
by (*simp only*: *subset-mset.not-less-zero*)

lemma *empty-subset-add-mset*[*simp*]: {#} <# *add-mset x M*
by(*auto intro*: *subset-mset.gr-zeroI*)

lemma *empty-le*: {#} ⊆# *A*
by (*fact subset-mset.zero-le*)

lemma *insert-subset-eq-iff*:

add-mset a A $\subseteq\#$ *B* $\longleftrightarrow$ *a* $\in\#$ *B* $\wedge$ *A* $\subseteq\#$ *B* $-$ $\{\#a\#\}$
using *le-diff-conv2* [*of Suc 0 count B a count A a*]
apply (*auto simp add*: *subseteq-mset-def not-in-iff Suc-le-eq*)
apply (*rule ccontr*)
apply (*auto simp add*: *not-in-iff*)
done

lemma *insert-union-subset-iff*:
add-mset a A $\subset\#$ *B* $\longleftrightarrow$ *a* $\in\#$ *B* $\wedge$ *A* $\subset\#$ *B* $-$ $\{\#a\#\}$
by (*auto simp add*: *insert-subset-eq-iff subset-mset-def*)

lemma *subset-eq-diff-conv*:
A $-$ *C* $\subseteq\#$ *B* $\longleftrightarrow$ *A* $\subseteq\#$ *B* $+$ *C*
by (*simp add*: *subseteq-mset-def le-diff-conv*)

lemma *multi-psub-of-add-self* [*simp*]: *A* $\subset\#$ *add-mset x A*
by (*auto simp*: *subset-mset-def subseteq-mset-def*)

lemma *multi-psub-self*: *A* $\subset\#$ *A* $=$ *False*
by *simp*

lemma *mset-subset-add-mset* [*simp*]: *add-mset x N* $\subset\#$ *add-mset x M* $\longleftrightarrow$ *N* $\subset\#$ *M*
unfolding *add-mset-add-single*[*of - N*] *add-mset-add-single*[*of - M*]
by (*fact subset-mset.add-less-cancel-right*)

lemma *mset-subset-diff-self*: *c* $\in\#$ *B* $\Longrightarrow$ *B* $-$ $\{\#c\#\}$ $\subset\#$ *B*
by (*auto simp*: *subset-mset-def elim*: *mset-add*)

lemma *Diff-eq-empty-iff-mset*: *A* $-$ *B* $=$ $\{\#\}$ $\longleftrightarrow$ *A* $\subseteq\#$ *B*
by (*auto simp*: *multiset-eq-iff subseteq-mset-def*)

lemma *add-mset-subseteq-single-iff*[*iff*]: *add-mset a M* $\subseteq\#$ $\{\#b\#\}$ $\longleftrightarrow$ *M* $=$ $\{\#\}$ $\wedge$ *a* $=$ *b*
proof
assume *A*: *add-mset a M* $\subseteq\#$ $\{\#b\#\}$
then have ‹*a* $=$ *b*›
by (*auto dest*: *mset-subset-eq-insertD*)
then show *M*$=\{\#\}$ $\wedge$ *a*$=$*b*
using *A* **by** (*simp add*: *mset-subset-eq-add-mset-cancel*)
qed *simp*

1.3.6 Intersection and bounded union

definition *inf-subset-mset* :: *'a multiset* $\Rightarrow$ *'a multiset* $\Rightarrow$ *'a multiset* (**infixl** $\cap\#$ *70*) **where**
multiset-inter-def: *inf-subset-mset A B* $=$ *A* $-$ (*A* $-$ *B*)

interpretation *subset-mset*: *semilattice-inf inf-subset-mset op* $\subseteq\#$ *op* $\subset\#$

proof −
have [*simp*]: $m \leq n \Longrightarrow m \leq q \Longrightarrow m \leq n - (n - q)$ **for** $m\ n\ q :: nat$
by *arith*
show *class.semilattice-inf op* $\cap\#$ *op* $\subseteq\#$ *op* $\subset\#$
by *standard* (*auto simp add*: *multiset-inter-def subseteq-mset-def*)
qed
— FIXME: avoid junk stemming from type class interpretation

definition *sup-subset-mset* :: *'a multiset* ⇒ *'a multiset* ⇒ *'a multiset*(**infixl** $\cup\#$ *70*)
where *sup-subset-mset A B* = *A* + (*B* − *A*) — FIXME irregular fact name

interpretation *subset-mset*: *semilattice-sup sup-subset-mset op* $\subseteq\#$ *op* $\subset\#$
proof −
have [*simp*]: $m \leq n \Longrightarrow q \leq n \Longrightarrow m + (q - m) \leq n$ **for** $m\ n\ q :: nat$
by *arith*
show *class.semilattice-sup op* $\cup\#$ *op* $\subseteq\#$ *op* $\subset\#$
by *standard* (*auto simp add*: *sup-subset-mset-def subseteq-mset-def*)
qed
— FIXME: avoid junk stemming from type class interpretation

interpretation *subset-mset*: *bounded-lattice-bot op* $\cap\#$ *op* $\subseteq\#$ *op* $\subset\#$
op $\cup\#$ {#}
by *standard auto*

1.3.7 Additional intersection facts

lemma *multiset-inter-count* [*simp*]:
fixes *A B* :: *'a multiset*
shows *count* (*A* $\cap\#$ *B*) *x* = *min* (*count A x*) (*count B x*)
by (*simp add*: *multiset-inter-def*)

lemma *set-mset-inter* [*simp*]:
set-mset (*A* $\cap\#$ *B*) = *set-mset A* ∩ *set-mset B*
by (*simp only*: *set-eq-iff count-greater-zero-iff* [*symmetric*] *multiset-inter-count*) *simp*

lemma *diff-intersect-left-idem* [*simp*]:
M − *M* $\cap\#$ *N* = *M* − *N*
by (*simp add*: *multiset-eq-iff min-def*)

lemma *diff-intersect-right-idem* [*simp*]:
M − *N* $\cap\#$ *M* = *M* − *N*
by (*simp add*: *multiset-eq-iff min-def*)

lemma *multiset-inter-single*[*simp*]: $a \neq b \Longrightarrow \{\#a\#\} \cap\# \{\#b\#\} = \{\#\}$
by (*rule multiset-eqI*) *auto*

lemma *multiset-union-diff-commute*:

assumes $B \cap\# C = \{\#\}$
shows $A + B - C = A - C + B$
proof (*rule multiset-eqI*)
fix x
from *assms* **have** *min* (*count B x*) (*count C x*) = *0*
by (*auto simp add*: *multiset-eq-iff*)
then have *count B x* = *0* ∨ *count C x* = *0*
unfolding *min-def* **by** (*auto split*: *if-splits*)
then show *count* (*A* + *B* − *C*) *x* = *count* (*A* − *C* + *B*) *x*
by *auto*
qed

lemma *disjunct-not-in*:
$A \cap\# B = \{\#\} \longleftrightarrow (\forall a.\ a \notin\# A \vee a \notin\# B)$ (**is** *?P* ⟷ *?Q*)
proof
assume *?P*
show *?Q*
proof
fix *a*
from ‹*?P*› **have** *min* (*count A a*) (*count B a*) = *0*
by (*simp add*: *multiset-eq-iff*)
then have *count A a* = *0* ∨ *count B a* = *0*
by (*cases count A a* ≤ *count B a*) (*simp-all add*: *min-def*)
then show *a* ∉# *A* ∨ *a* ∉# *B*
by (*simp add*: *not-in-iff*)
qed
next
assume *?Q*
show *?P*
proof (*rule multiset-eqI*)
fix *a*
from ‹*?Q*› **have** *count A a* = *0* ∨ *count B a* = *0*
by (*auto simp add*: *not-in-iff*)
then show *count* (*A* ∩# *B*) *a* = *count* {#} *a*
by *auto*
qed
qed

lemma *inter-mset-empty-distrib-right*: $A \cap\# (B + C) = \{\#\} \longleftrightarrow A \cap\# B = \{\#\} \wedge A \cap\# C = \{\#\}$
by (*meson disjunct-not-in union-iff*)

lemma *inter-mset-empty-distrib-left*: $(A + B) \cap\# C = \{\#\} \longleftrightarrow A \cap\# C = \{\#\} \wedge B \cap\# C = \{\#\}$
by (*meson disjunct-not-in union-iff*)

lemma *add-mset-inter-add-mset*[*simp*]:
add-mset a A ∩# *add-mset a B* = *add-mset a* (*A* ∩# *B*)
by (*metis add-mset-add-single add-mset-diff-bothsides diff-subset-eq-self multiset-inter-def*

subset-mset.diff-add-assoc2)

lemma *add-mset-disjoint* [*simp*]:
add-mset a A ∩# *B* = {#} ⟷ *a* ∉# *B* ∧ *A* ∩# *B* = {#}
{#} = *add-mset a A* ∩# *B* ⟷ *a* ∉# *B* ∧ {#} = *A* ∩# *B*
by (*auto simp*: *disjunct-not-in*)

lemma *disjoint-add-mset* [*simp*]:
B ∩# *add-mset a A* = {#} ⟷ *a* ∉# *B* ∧ *B* ∩# *A* = {#}
{#} = *A* ∩# *add-mset b B* ⟷ *b* ∉# *A* ∧ {#} = *A* ∩# *B*
by (*auto simp*: *disjunct-not-in*)

lemma *inter-add-left1*: ¬ *x* ∈# *N* ⟹ (*add-mset x M*) ∩# *N* = *M* ∩# *N*
by (*simp add*: *multiset-eq-iff not-in-iff*)

lemma *inter-add-left2*: *x* ∈# *N* ⟹ (*add-mset x M*) ∩# *N* = *add-mset x* (*M* ∩# (*N* − {#*x*#}))
by (*auto simp add*: *multiset-eq-iff elim*: *mset-add*)

lemma *inter-add-right1*: ¬ *x* ∈# *N* ⟹ *N* ∩# (*add-mset x M*) = *N* ∩# *M*
by (*simp add*: *multiset-eq-iff not-in-iff*)

lemma *inter-add-right2*: *x* ∈# *N* ⟹ *N* ∩# (*add-mset x M*) = *add-mset x* ((*N* − {#*x*#}) ∩# *M*)
by (*auto simp add*: *multiset-eq-iff elim*: *mset-add*)

lemma *disjunct-set-mset-diff*:
assumes *M* ∩# *N* = {#}
shows *set-mset* (*M* − *N*) = *set-mset M*
proof (*rule set-eqI*)
fix *a*
from *assms* **have** *a* ∉# *M* ∨ *a* ∉# *N*
by (*simp add*: *disjunct-not-in*)
then show *a* ∈# *M* − *N* ⟷ *a* ∈# *M*
by (*auto dest*: *in-diffD*) (*simp add*: *in-diff-count not-in-iff*)
qed

lemma *at-most-one-mset-mset-diff*:
assumes *a* ∉# *M* − {#*a*#}
shows *set-mset* (*M* − {#*a*#}) = *set-mset M* − {*a*}
using *assms* **by** (*auto simp add*: *not-in-iff in-diff-count set-eq-iff*)

lemma *more-than-one-mset-mset-diff*:
assumes *a* ∈# *M* − {#*a*#}
shows *set-mset* (*M* − {#*a*#}) = *set-mset M*
proof (*rule set-eqI*)
fix *b*
have *Suc 0* < *count M b* ⟹ *count M b* > *0* **by** *arith*
then show *b* ∈# *M* − {#*a*#} ⟷ *b* ∈# *M*

using *assms* **by** (*auto simp add*: *in-diff-count*)
qed

lemma *inter-iff*:
$a \in\# A \cap\# B \longleftrightarrow a \in\# A \wedge a \in\# B$
by *simp*

lemma *inter-union-distrib-left*:
$A \cap\# B + C = (A + C) \cap\# (B + C)$
by (*simp add*: *multiset-eq-iff min-add-distrib-left*)

lemma *inter-union-distrib-right*:
$C + A \cap\# B = (C + A) \cap\# (C + B)$
using *inter-union-distrib-left* [*of A B C*] **by** (*simp add*: *ac-simps*)

lemma *inter-subset-eq-union*:
$A \cap\# B \subseteq\# A + B$
by (*auto simp add*: *subseteq-mset-def*)

1.3.8 Additional bounded union facts

lemma *sup-subset-mset-count* [*simp*]: — FIXME irregular fact name
count $(A \cup\# B)$ x = *max* (*count A x*) (*count B x*)
by (*simp add*: *sup-subset-mset-def*)

lemma *set-mset-sup* [*simp*]:
set-mset $(A \cup\# B)$ = *set-mset A* $\cup$ *set-mset B*
by (*simp only*: *set-eq-iff count-greater-zero-iff* [*symmetric*] *sup-subset-mset-count*)
(*auto simp add*: *not-in-iff elim*: *mset-add*)

lemma *sup-union-left1* [*simp*]: $\neg x \in\# N \Longrightarrow$ (*add-mset x M*) $\cup\# N$ = *add-mset x* $(M \cup\# N)$
by (*simp add*: *multiset-eq-iff not-in-iff*)

lemma *sup-union-left2*: $x \in\# N \Longrightarrow$ (*add-mset x M*) $\cup\# N$ = *add-mset x* $(M \cup\# (N - \{\#x\#\}))$
by (*simp add*: *multiset-eq-iff*)

lemma *sup-union-right1* [*simp*]: $\neg x \in\# N \Longrightarrow N \cup\#$ (*add-mset x M*) = *add-mset x* $(N \cup\# M)$
by (*simp add*: *multiset-eq-iff not-in-iff*)

lemma *sup-union-right2*: $x \in\# N \Longrightarrow N \cup\#$ (*add-mset x M*) = *add-mset x* $((N - \{\#x\#\}) \cup\# M)$
by (*simp add*: *multiset-eq-iff*)

lemma *sup-union-distrib-left*:
$A \cup\# B + C = (A + C) \cup\# (B + C)$
by (*simp add*: *multiset-eq-iff max-add-distrib-left*)

lemma *union-sup-distrib-right*:
$C + A \cup\# B = (C + A) \cup\# (C + B)$
using *sup-union-distrib-left* [*of A B C*] **by** (*simp add*: *ac-simps*)

lemma *union-diff-inter-eq-sup*:
$A + B - A \cap\# B = A \cup\# B$
by (*auto simp add*: *multiset-eq-iff*)

lemma *union-diff-sup-eq-inter*:
$A + B - A \cup\# B = A \cap\# B$
by (*auto simp add*: *multiset-eq-iff*)

lemma *add-mset-union*:
‹*add-mset a A* $\cup\#$ *add-mset a B* = *add-mset a* (*A* $\cup\#$ *B*)›
by (*auto simp*: *multiset-eq-iff max-def*)

1.3.9 Subset is an order

interpretation *subset-mset*: *order op* $\subseteq\#$ *op* $\subset\#$ **by** *unfold-locales*

1.4 Replicate and repeat operations

definition *replicate-mset* :: *nat* ⇒ *'a* ⇒ *'a multiset* **where**
replicate-mset n x = (*add-mset x* ^^ *n*) {#}

lemma *replicate-mset-0*[*simp*]: *replicate-mset 0 x* = {#}
unfolding *replicate-mset-def* **by** *simp*

lemma *replicate-mset-Suc* [*simp*]: *replicate-mset* (*Suc n*) *x* = *add-mset x* (*replicate-mset n x*)
unfolding *replicate-mset-def* **by** (*induct n*) (*auto intro*: *add.commute*)

lemma *count-replicate-mset*[*simp*]: *count* (*replicate-mset n x*) *y* = (*if y* = *x then n else 0*)
unfolding *replicate-mset-def* **by** (*induct n*) *auto*

fun *repeat-mset* :: *nat* ⇒ *'a multiset* ⇒ *'a multiset* **where**
repeat-mset 0 - = {#} |
repeat-mset (*Suc n*) *A* = *A* + *repeat-mset n A*

lemma *count-repeat-mset* [*simp*]: *count* (*repeat-mset i A*) *a* = *i* ∗ *count A a*
by (*induction i*) *auto*

lemma *repeat-mset-right* [*simp*]: *repeat-mset a* (*repeat-mset b A*) = *repeat-mset* (*a* ∗ *b*) *A*
by (*auto simp*: *multiset-eq-iff left-diff-distrib'*)

lemma *left-diff-repeat-mset-distrib'*: ‹*repeat-mset* (*i* − *j*) *u* = *repeat-mset i u* − *repeat-mset j u*›

by (*auto simp*: *multiset-eq-iff left-diff-distrib'*)

lemma *left-add-mult-distrib-mset*:
repeat-mset i u + (*repeat-mset j u* + *k*) = *repeat-mset* (*i*+*j*) *u* + *k*
by (*auto simp*: *multiset-eq-iff add-mult-distrib*)

lemma *repeat-mset-distrib*:
repeat-mset (*m* + *n*) *A* = *repeat-mset m A* + *repeat-mset n A*
by (*auto simp*: *multiset-eq-iff Nat.add-mult-distrib*)

lemma *repeat-mset-distrib2*[*simp*]:
repeat-mset n (*A* + *B*) = *repeat-mset n A* + *repeat-mset n B*
by (*auto simp*: *multiset-eq-iff add-mult-distrib2*)

lemma *repeat-mset-replicate-mset*[*simp*]:
repeat-mset n {#*a*#} = *replicate-mset n a*
by (*auto simp*: *multiset-eq-iff*)

lemma *repeat-mset-distrib-add-mset*[*simp*]:
repeat-mset n (*add-mset a A*) = *replicate-mset n a* + *repeat-mset n A*
by (*auto simp*: *multiset-eq-iff*)

lemma *repeat-mset-empty*[*simp*]: *repeat-mset n* {#} = {#}
by (*induction n*) *simp-all*

1.4.1 Simprocs

lemma *mset-diff-add-eq1*:
j ≤ (*i*::*nat*) ⟹ ((*repeat-mset i u* + *m*) − (*repeat-mset j u* + *n*)) = ((*repeat-mset* (*i*−*j*) *u* + *m*) − *n*)
by (*auto simp*: *multiset-eq-iff nat-diff-add-eq1*)

lemma *mset-diff-add-eq2*:
i ≤ (*j*::*nat*) ⟹ ((*repeat-mset i u* + *m*) − (*repeat-mset j u* + *n*)) = (*m* − (*repeat-mset* (*j*−*i*) *u* + *n*))
by (*auto simp*: *multiset-eq-iff nat-diff-add-eq2*)

lemma *mset-eq-add-iff1*:
j ≤ (*i*::*nat*) ⟹ (*repeat-mset i u* + *m* = *repeat-mset j u* + *n*) = (*repeat-mset* (*i*−*j*) *u* + *m* = *n*)
by (*auto simp*: *multiset-eq-iff nat-eq-add-iff1*)

lemma *mset-eq-add-iff2*:
i ≤ (*j*::*nat*) ⟹ (*repeat-mset i u* + *m* = *repeat-mset j u* + *n*) = (*m* = *repeat-mset* (*j*−*i*) *u* + *n*)
by (*auto simp*: *multiset-eq-iff nat-eq-add-iff2*)

lemma *mset-subseteq-add-iff1*:
j ≤ (*i*::*nat*) ⟹ (*repeat-mset i u* + *m* ⊆# *repeat-mset j u* + *n*) = (*repeat-mset*

$(i-j)\ u + m \subseteq\# n)$
by (*auto simp add*: *subseteq-mset-def nat-le-add-iff1*)

lemma *mset-subseteq-add-iff2*:
$i \leq (j::nat) \Longrightarrow (repeat\text{-}mset\ i\ u + m \subseteq\#\ repeat\text{-}mset\ j\ u + n) = (m \subseteq\#$
repeat-mset $(j-i)\ u + n)$
by (*auto simp add*: *subseteq-mset-def nat-le-add-iff2*)

lemma *mset-subset-add-iff1*:
$j \leq (i::nat) \Longrightarrow (repeat\text{-}mset\ i\ u + m \subset\#\ repeat\text{-}mset\ j\ u + n) = (repeat\text{-}mset$
$(i-j)\ u + m \subset\#\ n)$
unfolding *subset-mset-def* **by** (*simp add*: *mset-eq-add-iff1 mset-subseteq-add-iff1*)

lemma *mset-subset-add-iff2*:
$i \leq (j::nat) \Longrightarrow (repeat\text{-}mset\ i\ u + m \subset\#\ repeat\text{-}mset\ j\ u + n) = (m \subset\#$
repeat-mset $(j-i)\ u + n)$
unfolding *subset-mset-def* **by** (*simp add*: *mset-eq-add-iff2 mset-subseteq-add-iff2*)

ML-file *multiset-simprocs-util.ML*
ML-file *multiset-simprocs.ML*

simproc-setup *mseteq-cancel-numerals*
$((l::'a\ multiset) + m = n \mid (l::'a\ multiset) = m + n \mid$
add-mset a m $= n \mid m =$ *add-mset a n* $\mid$
replicate-mset p a $= n \mid m =$ *replicate-mset p a* $\mid$
repeat-mset p m $= n \mid m =$ *repeat-mset p m*$) =$
‹*fn phi => Multiset-Simprocs.eq-cancel-msets*›

simproc-setup *msetless-cancel-numerals*
$((l::'a\ multiset) + m \subset\# n \mid (l::'a\ multiset) \subset\# m + n \mid$
add-mset a m $\subset\# n \mid m \subset\#$ *add-mset a n* $\mid$
replicate-mset p r $\subset\# n \mid m \subset\#$ *replicate-mset p r* $\mid$
repeat-mset p m $\subset\# n \mid m \subset\#$ *repeat-mset p m*$) =$
‹*fn phi => Multiset-Simprocs.subset-cancel-msets*›

simproc-setup *msetle-cancel-numerals*
$((l::'a\ multiset) + m \subseteq\# n \mid (l::'a\ multiset) \subseteq\# m + n \mid$
add-mset a m $\subseteq\# n \mid m \subseteq\#$ *add-mset a n* $\mid$
replicate-mset p r $\subseteq\# n \mid m \subseteq\#$ *replicate-mset p r* $\mid$
repeat-mset p m $\subseteq\# n \mid m \subseteq\#$ *repeat-mset p m*$) =$
‹*fn phi => Multiset-Simprocs.subseteq-cancel-msets*›

simproc-setup *msetdiff-cancel-numerals*
$(((l::'a\ multiset) + m) - n \mid (l::'a\ multiset) - (m + n) \mid$
add-mset a m $- n \mid m -$ *add-mset a n* $\mid$
replicate-mset p r $- n \mid m -$ *replicate-mset p r* $\mid$
repeat-mset p m $- n \mid m -$ *repeat-mset p m*$) =$
‹*fn phi => Multiset-Simprocs.diff-cancel-msets*›

1.4.2 Conditionally complete lattice

instantiation *multiset* :: (*type*) *Inf*
begin

lift-definition *Inf-multiset* :: *'a multiset set* ⇒ *'a multiset* **is**
  λA i. if A = {} then 0 else Inf ((*λf. f i*) ‘ *A*)
proof −
  fix *A* :: (*'a* ⇒ *nat*) *set* **assume** ∗: ⋀*x. x* ∈ *A* ⟹ *x* ∈ *multiset*
  have *finite* {*i. (if A = {} then 0 else Inf* ((*λf. f i*) ‘ *A*)) > *0*} **unfolding** *multiset-def*
  proof (*cases A = {}*)
    case *False*
    then obtain *f* **where** *f* ∈ *A* **by** *blast*
    hence {*i. Inf* ((*λf. f i*) ‘ *A*) > *0*} ⊆ {*i. f i > 0*}
      by (*auto intro*: *less-le-trans*[*OF - cInf-lower*])
    moreover from ‹*f* ∈ *A*› ∗ **have** *finite* ... **by** (*simp add*: *multiset-def*)
    ultimately have *finite* {*i. Inf* ((*λf. f i*) ‘ *A*) > *0*} **by** (*rule finite-subset*)
    with *False* **show** *?thesis* **by** *simp*
  qed *simp-all*
 thus (*λi. if A = {} then 0 else INF f*:*A. f i*) ∈ *multiset* **by** (*simp add*: *multiset-def*)
qed

instance ..

end

lemma *Inf-multiset-empty*: *Inf* {} = {#}
  by *transfer simp-all*

lemma *count-Inf-multiset-nonempty*: *A* ≠ {} ⟹ *count* (*Inf A*) *x* = *Inf* ((*λX. count X x*) ‘ *A*)
  by *transfer simp-all*

instantiation *multiset* :: (*type*) *Sup*
begin

definition *Sup-multiset* :: *'a multiset set* ⇒ *'a multiset* **where**
  Sup-multiset A = (*if A* ≠ {} ∧ *subset-mset.bdd-above A then*
        Abs-multiset (*λi. Sup* ((*λX. count X i*) ‘ *A*)) *else* {#})

lemma *Sup-multiset-empty*: *Sup* {} = {#}
  by (*simp add*: *Sup-multiset-def*)

lemma *Sup-multiset-unbounded*: ¬*subset-mset.bdd-above A* ⟹ *Sup A* = {#}
  by (*simp add*: *Sup-multiset-def*)

instance ..

end

lemma *bdd-above-multiset-imp-bdd-above-count*:
assumes *subset-mset.bdd-above* (*A* :: *'a multiset set*)
shows *bdd-above* ((λ*X*. *count X x*) ‘ *A*)
proof −
from *assms* **obtain** *Y* **where** *Y*: ∀ *X*∈*A*. *X* ⊆# *Y*
by (*auto simp*: *subset-mset.bdd-above-def*)
hence *count X x* ≤ *count Y x* **if** *X* ∈ *A* **for** *X*
using *that* **by** (*auto intro*: *mset-subset-eq-count*)
thus *?thesis* **by** (*intro bdd-aboveI*[*of* - *count Y x*]) *auto*
qed

lemma *bdd-above-multiset-imp-finite-support*:
assumes *A* ≠ {} *subset-mset.bdd-above* (*A* :: *'a multiset set*)
shows *finite* (⋃ *X*∈*A*. {*x*. *count X x* > *0*})
proof −
from *assms* **obtain** *Y* **where** *Y*: ∀ *X*∈*A*. *X* ⊆# *Y*
by (*auto simp*: *subset-mset.bdd-above-def*)
hence *count X x* ≤ *count Y x* **if** *X* ∈ *A* **for** *X x*
using *that* **by** (*auto intro*: *mset-subset-eq-count*)
hence (⋃ *X*∈*A*. {*x*. *count X x* > *0*}) ⊆ {*x*. *count Y x* > *0*}
by *safe* (*erule less-le-trans*)
moreover have *finite* ... **by** *simp*
ultimately show *?thesis* **by** (*rule finite-subset*)
qed

lemma *Sup-multiset-in-multiset*:
assumes *A* ≠ {} *subset-mset.bdd-above A*
shows (λ*i*. *SUP X*:*A*. *count X i*) ∈ *multiset*
unfolding *multiset-def*
proof
have {*i*. *Sup* ((λ*X*. *count X i*) ‘ *A*) > *0*} ⊆ (⋃ *X*∈*A*. {*i*. *0* < *count X i*})
proof *safe*
fix *i* **assume** *pos*: (*SUP X*:*A*. *count X i*) > *0*
show *i* ∈ (⋃ *X*∈*A*. {*i*. *0* < *count X i*})
proof (*rule ccontr*)
assume *i* ∉ (⋃ *X*∈*A*. {*i*. *0* < *count X i*})
hence ∀ *X*∈*A*. *count X i* ≤ *0* **by** (*auto simp*: *count-eq-zero-iff*)
with *assms* **have** (*SUP X*:*A*. *count X i*) ≤ *0*
by (*intro cSup-least bdd-above-multiset-imp-bdd-above-count*) *auto*
with *pos* **show** *False* **by** *simp*
qed
qed
moreover from *assms* **have** *finite* ... **by** (*rule bdd-above-multiset-imp-finite-support*)
ultimately show *finite* {*i*. *Sup* ((λ*X*. *count X i*) ‘ *A*) > *0*} **by** (*rule finite-subset*)
qed

lemma *count-Sup-multiset-nonempty*:
assumes $A \neq \{\}$ *subset-mset.bdd-above A*
shows *count* (*Sup A*) x = (*SUP X*:*A*. *count X x*)
using *assms* **by** (*simp add*: *Sup-multiset-def Abs-multiset-inverse Sup-multiset-in-multiset*)

interpretation *subset-mset*: *conditionally-complete-lattice Inf Sup op* $\cap\#$ *op* $\subseteq\#$ *op* $\subset\#$ *op* $\cup\#$
proof
fix X :: *'a multiset* **and** A
assume $X \in A$
show *Inf A* $\subseteq\#$ X
proof (*rule mset-subset-eqI*)
fix x
from ‹$X \in A$› **have** $A \neq \{\}$ **by** *auto*
hence *count* (*Inf A*) x = (*INF X*:*A*. *count X x*)
by (*simp add*: *count-Inf-multiset-nonempty*)
also from ‹$X \in A$› **have** ... $\leq$ *count X x*
by (*intro cInf-lower*) *simp-all*
finally show *count* (*Inf A*) $x \leq$ *count X x* **.**
qed
next
fix X :: *'a multiset* **and** A
assume *nonempty*: $A \neq \{\}$ **and** *le*: $\bigwedge Y.\ Y \in A \Longrightarrow X \subseteq\# Y$
show $X \subseteq\#$ *Inf A*
proof (*rule mset-subset-eqI*)
fix x
from *nonempty* **have** *count X x* $\leq$ (*INF X*:*A*. *count X x*)
by (*intro cInf-greatest*) (*auto intro*: *mset-subset-eq-count le*)
also from *nonempty* **have** ... = *count* (*Inf A*) x **by** (*simp add*: *count-Inf-multiset-nonempty*)
finally show *count X x* $\leq$ *count* (*Inf A*) x **.**
qed
next
fix X :: *'a multiset* **and** A
assume X: $X \in A$ **and** *bdd*: *subset-mset.bdd-above A*
show $X \subseteq\#$ *Sup A*
proof (*rule mset-subset-eqI*)
fix x
from X **have** $A \neq \{\}$ **by** *auto*
have *count X x* $\leq$ (*SUP X*:*A*. *count X x*)
by (*intro cSUP-upper X bdd-above-multiset-imp-bdd-above-count bdd*)
also from *count-Sup-multiset-nonempty*[*OF* ‹$A \neq \{\}$› *bdd*]
have (*SUP X*:*A*. *count X x*) = *count* (*Sup A*) x **by** *simp*
finally show *count X x* $\leq$ *count* (*Sup A*) x **.**
qed
next
fix X :: *'a multiset* **and** A
assume *nonempty*: $A \neq \{\}$ **and** *ge*: $\bigwedge Y.\ Y \in A \Longrightarrow Y \subseteq\# X$
from *ge* **have** *bdd*: *subset-mset.bdd-above A* **by** (*rule subset-mset.bdd-aboveI*[*of*

```
- X])
  show Sup A ⊆# X
  proof (rule mset-subset-eqI)
    fix x
    from count-Sup-multiset-nonempty[OF ‹A ≠ {}› bdd]
      have count (Sup A) x = (SUP X:A. count X x) .
    also from nonempty have ... ≤ count X x
      by (intro cSup-least) (auto intro: mset-subset-eq-count ge)
    finally show count (Sup A) x ≤ count X x .
  qed
qed

lemma set-mset-Inf:
  assumes A ≠ {}
  shows   set-mset (Inf A) = (⋂X∈A. set-mset X)
proof safe
  fix x X assume x ∈# Inf A X ∈ A
  hence nonempty: A ≠ {} by (auto simp: Inf-multiset-empty)
  from ‹x ∈# Inf A› have {#x#} ⊆# Inf A by auto
  also from ‹X ∈ A› have ... ⊆# X by (rule subset-mset.cInf-lower) simp-all
  finally show x ∈# X by simp
next
  fix x assume x: x ∈ (⋂X∈A. set-mset X)
  hence {#x#} ⊆# X if X ∈ A for X using that by auto
  from assms and this have {#x#} ⊆# Inf A by (rule subset-mset.cInf-greatest)
  thus x ∈# Inf A by simp
qed

lemma in-Inf-multiset-iff:
  assumes A ≠ {}
  shows   x ∈# Inf A ⟷ (∀X∈A. x ∈# X)
proof −
  from assms have set-mset (Inf A) = (⋂X∈A. set-mset X) by (rule set-mset-Inf)
  also have x ∈ ... ⟷ (∀X∈A. x ∈# X) by simp
  finally show ?thesis .
qed

lemma in-Inf-multisetD: x ∈# Inf A ⟹ X ∈ A ⟹ x ∈# X
  by (subst (asm) in-Inf-multiset-iff) auto

lemma set-mset-Sup:
  assumes subset-mset.bdd-above A
  shows   set-mset (Sup A) = (⋃X∈A. set-mset X)
proof safe
  fix x assume x ∈# Sup A
  hence nonempty: A ≠ {} by (auto simp: Sup-multiset-empty)
  show x ∈ (⋃X∈A. set-mset X)
  proof (rule ccontr)
    assume x: x ∉ (⋃X∈A. set-mset X)
```

```
    have count X x ≤ count (Sup A) x if X ∈ A for X x
      using that by (intro mset-subset-eq-count subset-mset.cSup-upper assms)
    with x have X ⊆# Sup A − {#x#} if X ∈ A for X
      using that by (auto simp: subseteq-mset-def algebra-simps not-in-iff)
    hence Sup A ⊆# Sup A − {#x#} by (intro subset-mset.cSup-least nonempty)
    with ‹x ∈# Sup A› show False
      by (auto simp: subseteq-mset-def count-greater-zero-iff [symmetric]
            simp del: count-greater-zero-iff dest!: spec[of - x])
  qed
next
  fix x X assume x ∈ set-mset X X ∈ A
  hence {#x#} ⊆# X by auto
  also have X ⊆# Sup A by (intro subset-mset.cSup-upper ‹X ∈ A› assms)
  finally show x ∈ set-mset (Sup A) by simp
qed

lemma in-Sup-multiset-iff:
  assumes subset-mset.bdd-above A
  shows   x ∈# Sup A ⟷ (∃ X∈A. x ∈# X)
proof −
  from assms have set-mset (Sup A) = (⋃ X∈A. set-mset X) by (rule set-mset-Sup)
  also have x ∈ ... ⟷ (∃ X∈A. x ∈# X) by simp
  finally show ?thesis .
qed

lemma in-Sup-multisetD:
  assumes x ∈# Sup A
  shows   ∃ X∈A. x ∈# X
proof −
  have subset-mset.bdd-above A
    by (rule ccontr) (insert assms, simp-all add: Sup-multiset-unbounded)
  with assms show ?thesis by (simp add: in-Sup-multiset-iff)
qed

interpretation subset-mset: distrib-lattice op ∩# op ⊆# op ⊂# op ∪#
proof
  fix A B C :: 'a multiset
  show A ∪# (B ∩# C) = A ∪# B ∩# (A ∪# C)
    by (intro multiset-eqI) simp-all
qed
```

1.4.3 Filter (with comprehension syntax)

Multiset comprehension

```
lift-definition filter-mset :: ('a ⇒ bool) ⇒ 'a multiset ⇒ 'a multiset
is λP M. λx. if P x then M x else 0
by (rule filter-preserves-multiset)

syntax (ASCII)
```

-MCollect :: *pttrn* ⇒ *'a multiset* ⇒ *bool* ⇒ *'a multiset* ((*1*{#- :# -./ -#}))
syntax
-MCollect :: *pttrn* ⇒ *'a multiset* ⇒ *bool* ⇒ *'a multiset* ((*1*{#- ∈# -./ -#}))
translations
{#*x* ∈# *M*. *P*#} == *CONST filter-mset* (λ*x*. *P*) *M*

lemma *count-filter-mset* [*simp*]:
count (*filter-mset P M*) *a* = (*if P a then count M a else 0*)
by (*simp add*: *filter-mset.rep-eq*)

lemma *set-mset-filter* [*simp*]:
set-mset (*filter-mset P M*) = {*a* ∈ *set-mset M*. *P a*}
by (*simp only*: *set-eq-iff count-greater-zero-iff* [*symmetric*] *count-filter-mset*) *simp*

lemma *filter-empty-mset* [*simp*]: *filter-mset P* {#} = {#}
by (*rule multiset-eqI*) *simp*

lemma *filter-single-mset*: *filter-mset P* {#*x*#} = (*if P x then* {#*x*#} *else* {#})
by (*rule multiset-eqI*) *simp*

lemma *filter-union-mset* [*simp*]: *filter-mset P* (*M* + *N*) = *filter-mset P M* + *filter-mset P N*
by (*rule multiset-eqI*) *simp*

lemma *filter-diff-mset* [*simp*]: *filter-mset P* (*M* − *N*) = *filter-mset P M* − *filter-mset P N*
by (*rule multiset-eqI*) *simp*

lemma *filter-inter-mset* [*simp*]: *filter-mset P* (*M* ∩# *N*) = *filter-mset P M* ∩# *filter-mset P N*
by (*rule multiset-eqI*) *simp*

lemma *filter-sup-mset*[*simp*]: *filter-mset P* (*A* ∪# *B*) = *filter-mset P A* ∪# *filter-mset P B*
by (*rule multiset-eqI*) *simp*

lemma *filter-mset-add-mset* [*simp*]:
filter-mset P (*add-mset x A*) =
(*if P x then add-mset x* (*filter-mset P A*) *else filter-mset P A*)
by (*auto simp*: *multiset-eq-iff*)

lemma *multiset-filter-subset*[*simp*]: *filter-mset f M* ⊆# *M*
by (*simp add*: *mset-subset-eqI*)

lemma *multiset-filter-mono*:
assumes *A* ⊆# *B*
shows *filter-mset f A* ⊆# *filter-mset f B*
proof −
from *assms*[*unfolded mset-subset-eq-exists-conv*]

obtain C **where** B: $B = A + C$ **by** *auto*
show *?thesis* **unfolding** B **by** *auto*
qed

lemma *filter-mset-eq-conv*:
filter-mset P $M = N \longleftrightarrow N \subseteq\# M \wedge (\forall b{\in}\#N.\ P\ b) \wedge (\forall a{\in}\#M - N.\ \neg\ P\ a)$
(**is** $?P \longleftrightarrow ?Q$)
proof
assume *?P* **then show** *?Q* **by** *auto* (*simp add*: *multiset-eq-iff in-diff-count*)
next
assume *?Q*
then obtain Q **where** M: $M = N + Q$
by (*auto simp add*: *mset-subset-eq-exists-conv*)
then have MN: $M - N = Q$ **by** *simp*
show *?P*
proof (*rule multiset-eqI*)
fix a
from ‹*?Q*› MN **have** $*$: $\neg\ P\ a \Longrightarrow a \notin\# N\ P\ a \Longrightarrow a \notin\# Q$
by *auto*
show *count* (*filter-mset* P M) a = *count* N a
proof (*cases* $a \in\# M$)
case *True*
with $*$ **show** *?thesis*
by (*simp add*: *not-in-iff* M)
next
case *False* **then have** *count* M a = *0*
by (*simp add*: *not-in-iff*)
with M **show** *?thesis* **by** *simp*
qed
qed
qed

lemma *filter-filter-mset*: *filter-mset* P (*filter-mset* Q M) = $\{\#x \in\# M.\ Q\ x \wedge P\ x\#\}$
by (*auto simp*: *multiset-eq-iff*)

lemma
filter-mset-True[*simp*]: $\{\#y \in\# M.\ True\#\} = M$ **and**
filter-mset-False[*simp*]: $\{\#y \in\# M.\ False\#\} = \{\#\}$
by (*auto simp*: *multiset-eq-iff*)

1.4.4 Size

definition *wcount* **where** *wcount* f $M = (\lambda x.\ count\ M\ x * Suc\ (f\ x))$

lemma *wcount-union*: *wcount* f $(M + N)$ a = *wcount* f M a + *wcount* f N a
by (*auto simp*: *wcount-def add-mult-distrib*)

lemma *wcount-add-mset*:

wcount f (add-mset x M) a = (if x = a then Suc (f a) else 0) + wcount f M a
unfolding *add-mset-add-single*[*of - M*] *wcount-union* **by** (*auto simp*: *wcount-def*)

definition *size-multiset* :: ('*a* ⇒ *nat*) ⇒ '*a multiset* ⇒ *nat* **where**
size-multiset f M = sum (*wcount f M*) (*set-mset M*)

lemmas *size-multiset-eq* = *size-multiset-def*[*unfolded wcount-def*]

instantiation *multiset* :: (*type*) *size*
begin

definition *size-multiset* **where**
size-multiset-overloaded-def: *size-multiset* = *Multiset.size-multiset* (λ-. *0*)
instance ..

end

lemmas *size-multiset-overloaded-eq* =
size-multiset-overloaded-def[*THEN fun-cong*, *unfolded size-multiset-eq*, *simplified*]

lemma *size-multiset-empty* [*simp*]: *size-multiset f* {#} = *0*
by (*simp add*: *size-multiset-def*)

lemma *size-empty* [*simp*]: *size* {#} = *0*
by (*simp add*: *size-multiset-overloaded-def*)

lemma *size-multiset-single* : *size-multiset f* {#*b*#} = *Suc* (*f b*)
by (*simp add*: *size-multiset-eq*)

lemma *size-single*: *size* {#*b*#} = *1*
by (*simp add*: *size-multiset-overloaded-def size-multiset-single*)

lemma *sum-wcount-Int*:
finite A ⟹ *sum* (*wcount f N*) (*A* ∩ *set-mset N*) = *sum* (*wcount f N*) *A*
by (*induct rule*: *finite-induct*)
(*simp-all add*: *Int-insert-left wcount-def count-eq-zero-iff*)

lemma *size-multiset-union* [*simp*]:
size-multiset f (*M* + *N*::'*a multiset*) = *size-multiset f M* + *size-multiset f N*
apply (*simp add*: *size-multiset-def sum-Un-nat sum.distrib sum-wcount-Int wcount-union*)
apply (*subst Int-commute*)
apply (*simp add*: *sum-wcount-Int*)
done

lemma *size-multiset-add-mset* [*simp*]:
size-multiset f (*add-mset a M*) = *Suc* (*f a*) + *size-multiset f M*
unfolding *add-mset-add-single*[*of - M*] *size-multiset-union* **by** (*auto simp*: *size-multiset-single*)

lemma *size-add-mset* [*simp*]: *size* (*add-mset a A*) = *Suc* (*size A*)

by (*simp add*: *size-multiset-overloaded-def wcount-add-mset*)

lemma *size-union* [*simp*]: *size* (*M* + *N*::'*a multiset*) = *size M* + *size N*
by (*auto simp add*: *size-multiset-overloaded-def*)

lemma *size-multiset-eq-0-iff-empty* [*iff*]:
size-multiset f M = *0* ⟷ *M* = {#}
by (*auto simp add*: *size-multiset-eq count-eq-zero-iff*)

lemma *size-eq-0-iff-empty* [*iff*]: (*size M* = *0*) = (*M* = {#})
by (*auto simp add*: *size-multiset-overloaded-def*)

lemma *nonempty-has-size*: (*S* ≠ {#}) = (*0* < *size S*)
by (*metis gr0I gr-implies-not0 size-empty size-eq-0-iff-empty*)

lemma *size-eq-Suc-imp-elem*: *size M* = *Suc n* ⟹ ∃ *a*. *a* ∈# *M*
apply (*unfold size-multiset-overloaded-eq*)
apply (*drule sum-SucD*)
apply *auto*
done

lemma *size-eq-Suc-imp-eq-union*:
assumes *size M* = *Suc n*
shows ∃ *a N*. *M* = *add-mset a N*
proof −
from *assms* **obtain** *a* **where** *a* ∈# *M*
by (*erule size-eq-Suc-imp-elem* [*THEN exE*])
then have *M* = *add-mset a* (*M* − {#*a*#}) **by** *simp*
then show *?thesis* **by** *blast*
qed

lemma *size-mset-mono*:
fixes *A B* :: '*a multiset*
assumes *A* ⊆# *B*
shows *size A* ≤ *size B*
proof −
from *assms*[*unfolded mset-subset-eq-exists-conv*]
obtain *C* **where** *B*: *B* = *A* + *C* **by** *auto*
show *?thesis* **unfolding** *B* **by** (*induct C*) *auto*
qed

lemma *size-filter-mset-lesseq*[*simp*]: *size* (*filter-mset f M*) ≤ *size M*
by (*rule size-mset-mono*[*OF multiset-filter-subset*])

lemma *size-Diff-submset*:
M ⊆# *M'* ⟹ *size* (*M'* − *M*) = *size M'* − *size*(*M*::'*a multiset*)
by (*metis add-diff-cancel-left' size-union mset-subset-eq-exists-conv*)

1.5 Induction and case splits

theorem *multiset-induct* [*case-names empty add*, *induct type*: *multiset*]:
  assumes *empty*: $P\ \{\#\}$
  assumes *add*: $\bigwedge x\ M.\ P\ M \Longrightarrow P\ (\mathit{add\text{-}mset}\ x\ M)$
  shows $P\ M$
proof (*induct* $n \equiv \mathit{size}\ M$ *arbitrary*: M)
  case *0* **thus** $P\ M$ **by** (*simp add*: *empty*)
next
  case (*Suc k*)
  obtain $N\ x$ **where** $M = \mathit{add\text{-}mset}\ x\ N$
    using ‹*Suc k* = *size M*› [*symmetric*]
    using *size-eq-Suc-imp-eq-union* **by** *fast*
  with *Suc add* **show** $P\ M$ **by** *simp*
qed

lemma *multi-nonempty-split*: $M \neq \{\#\} \Longrightarrow \exists A\ a.\ M = \mathit{add\text{-}mset}\ a\ A$
by (*induct M*) *auto*

lemma *multiset-cases* [*cases type*]:
  obtains (*empty*) $M = \{\#\}$
    | (*add*) $x\ N$ **where** $M = \mathit{add\text{-}mset}\ x\ N$
  by (*induct M*) *simp-all*

lemma *multi-drop-mem-not-eq*: $c \in\# B \Longrightarrow B - \{\#c\#\} \neq B$
by (*cases* $B = \{\#\}$) (*auto dest*: *multi-member-split*)

lemma *multiset-partition*: $M = \{\#\ x{\in}\#M.\ P\ x\ \#\} + \{\#\ x{\in}\#M.\ \neg\ P\ x\ \#\}$
apply (*subst multiset-eq-iff*)
apply *auto*
done

lemma *mset-subset-size*: $(A::'a\ \mathit{multiset}) \subset\# B \Longrightarrow \mathit{size}\ A < \mathit{size}\ B$
proof (*induct A arbitrary*: B)
  case (*empty M*)
  then have $M \neq \{\#\}$ **by** (*simp add*: *subset-mset.zero-less-iff-neq-zero*)
  then obtain $M'\ x$ **where** $M = \mathit{add\text{-}mset}\ x\ M'$
    by (*blast dest*: *multi-nonempty-split*)
  then show *?case* **by** *simp*
next
  case (*add x S T*)
  have *IH*: $\bigwedge B.\ S \subset\# B \Longrightarrow \mathit{size}\ S < \mathit{size}\ B$ **by** *fact*
  have *SxsubT*: $\mathit{add\text{-}mset}\ x\ S \subset\# T$ **by** *fact*
  then have $x \in\# T$ **and** $S \subset\# T$
    by (*auto dest*: *mset-subset-insertD*)
  then obtain T' **where** *T*: $T = \mathit{add\text{-}mset}\ x\ T'$
    by (*blast dest*: *multi-member-split*)
  then have $S \subset\# T'$ **using** *SxsubT*
    by *simp*
  then have $\mathit{size}\ S < \mathit{size}\ T'$ **using** *IH* **by** *simp*

then show *?case* **using** *T* **by** *simp*
qed

lemma *size-1-singleton-mset*: *size M = 1* $\Longrightarrow \exists\, a.\ M = \{\#a\#\}$
by (*cases M*) *auto*

1.5.1 Strong induction and subset induction for multisets

Well-foundedness of strict subset relation

lemma *wf-subset-mset-rel*: *wf* $\{(M, N :: 'a\ multiset).\ M \subset\# N\}$
apply (*rule wf-measure* [*THEN wf-subset*, **where** *f1=size*])
apply (*clarsimp simp*: *measure-def inv-image-def mset-subset-size*)
done

lemma *full-multiset-induct* [*case-names less*]:
assumes *ih*: $\bigwedge B.\ \forall (A::'a\ multiset).\ A \subset\# B \longrightarrow P\ A \Longrightarrow P\ B$
shows *P B*
apply (*rule wf-subset-mset-rel* [*THEN wf-induct*])
apply (*rule ih*, *auto*)
done

lemma *multi-subset-induct* [*consumes 2*, *case-names empty add*]:
assumes *F* $\subseteq\#$ *A*
and *empty*: *P* {#}
and *insert*: $\bigwedge a\ F.\ a \in\# A \Longrightarrow P\ F \Longrightarrow P$ (*add-mset a F*)
shows *P F*
proof −
from ‹*F* $\subseteq\#$ *A*›
show *?thesis*
proof (*induct F*)
show *P* {#} **by** *fact*
next
fix *x F*
assume *P*: *F* $\subseteq\#$ *A* $\Longrightarrow$ *P F* **and** *i*: *add-mset x F* $\subseteq\#$ *A*
show *P* (*add-mset x F*)
proof (*rule insert*)
from *i* **show** *x* $\in\#$ *A* **by** (*auto dest*: *mset-subset-eq-insertD*)
from *i* **have** *F* $\subseteq\#$ *A* **by** (*auto dest*: *mset-subset-eq-insertD*)
with *P* **show** *P F* **.**
qed
qed
qed

1.6 The fold combinator

definition *fold-mset* :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a\ multiset \Rightarrow 'b$
where
fold-mset f s M = *Finite-Set.fold* $(\lambda x.\ f\ x\ \hat{}\hat{}\ count\ M\ x)$ *s* (*set-mset M*)

lemma *fold-mset-empty* [*simp*]: *fold-mset f s* {#} = *s*
by (*simp add*: *fold-mset-def*)

context *comp-fun-commute*
begin

lemma *fold-mset-add-mset* [*simp*]: *fold-mset f s* (*add-mset x M*) = *f x* (*fold-mset f s M*)
proof −
interpret *mset*: *comp-fun-commute* λ*y*. *f y* ^^ *count M y*
by (*fact comp-fun-commute-funpow*)
interpret *mset-union*: *comp-fun-commute* λ*y*. *f y* ^^ *count* (*add-mset x M*) *y*
by (*fact comp-fun-commute-funpow*)
show *?thesis*
proof (*cases x* ∈ *set-mset M*)
case *False*
then have ∗: *count* (*add-mset x M*) *x* = *1*
by (*simp add*: *not-in-iff*)
from *False* **have** *Finite-Set.fold* (λ*y*. *f y* ^^ *count* (*add-mset x M*) *y*) *s* (*set-mset M*) =
Finite-Set.fold (λ*y*. *f y* ^^ *count M y*) *s* (*set-mset M*)
by (*auto intro*!: *Finite-Set.fold-cong comp-fun-commute-funpow*)
with *False* ∗ **show** *?thesis*
by (*simp add*: *fold-mset-def del*: *count-add-mset*)
next
case *True*
define *N* **where** *N* = *set-mset M* − {*x*}
from *N-def True* **have** ∗: *set-mset M* = *insert x N x* ∉ *N finite N* **by** *auto*
then have *Finite-Set.fold* (λ*y*. *f y* ^^ *count* (*add-mset x M*) *y*) *s N* =
Finite-Set.fold (λ*y*. *f y* ^^ *count M y*) *s N*
by (*auto intro*!: *Finite-Set.fold-cong comp-fun-commute-funpow*)
with ∗ **show** *?thesis* **by** (*simp add*: *fold-mset-def del*: *count-add-mset*) *simp*
qed
qed

corollary *fold-mset-single*: *fold-mset f s* {#*x*#} = *f x s*
by *simp*

lemma *fold-mset-fun-left-comm*: *f x* (*fold-mset f s M*) = *fold-mset f* (*f x s*) *M*
by (*induct M*) (*simp-all add*: *fun-left-comm*)

lemma *fold-mset-union* [*simp*]: *fold-mset f s* (*M* + *N*) = *fold-mset f* (*fold-mset f s M*) *N*
by (*induct M*) (*simp-all add*: *fold-mset-fun-left-comm*)

lemma *fold-mset-fusion*:
assumes *comp-fun-commute g*
and ∗: ⋀*x y*. *h* (*g x y*) = *f x* (*h y*)
shows *h* (*fold-mset g w A*) = *fold-mset f* (*h w*) *A*

proof −
interpret *comp-fun-commute g* **by** (*fact assms*)
from ∗ **show** *?thesis* **by** (*induct A*) *auto*
qed

end

lemma *union-fold-mset-add-mset*: *A* + *B* = *fold-mset add-mset A B*
proof −
interpret *comp-fun-commute add-mset*
by *standard auto*
show *?thesis*
by (*induction B*) *auto*
qed

A note on code generation: When defining some function containing a subterm *fold-mset F*, code generation is not automatic. When interpreting locale *left-commutative* with *F*, the would be code thms for *fold-mset* become thms like *fold-mset F z* {#} = *z* where *F* is not a pattern but contains defined symbols, i.e. is not a code thm. Hence a separate constant with its own code thms needs to be introduced for *F*. See the image operator below.

1.7 Image

definition *image-mset* :: ('*a* ⇒ '*b*) ⇒ '*a multiset* ⇒ '*b multiset* **where**
image-mset f = *fold-mset* (*add-mset* ∘ *f*) {#}

lemma *comp-fun-commute-mset-image*: *comp-fun-commute* (*add-mset* ∘ *f*)
proof
qed (*simp add*: *fun-eq-iff*)

lemma *image-mset-empty* [*simp*]: *image-mset f* {#} = {#}
by (*simp add*: *image-mset-def*)

lemma *image-mset-single*: *image-mset f* {#*x*#} = {#*f x*#}
proof −
interpret *comp-fun-commute add-mset* ∘ *f*
by (*fact comp-fun-commute-mset-image*)
show *?thesis* **by** (*simp add*: *image-mset-def*)
qed

lemma *image-mset-union* [*simp*]: *image-mset f* (*M* + *N*) = *image-mset f M* + *image-mset f N*
proof −
interpret *comp-fun-commute add-mset* ∘ *f*
by (*fact comp-fun-commute-mset-image*)
show *?thesis* **by** (*induct N*) (*simp-all add*: *image-mset-def*)
qed

corollary *image-mset-add-mset* [*simp*]:
image-mset f (*add-mset a M*) = *add-mset* (*f a*) (*image-mset f M*)
unfolding *image-mset-union add-mset-add-single*[*of a M*] **by** (*simp add*: *image-mset-single*)

lemma *set-image-mset* [*simp*]: *set-mset* (*image-mset f M*) = *image f* (*set-mset M*)
by (*induct M*) *simp-all*

lemma *size-image-mset* [*simp*]: *size* (*image-mset f M*) = *size M*
by (*induct M*) *simp-all*

lemma *image-mset-is-empty-iff* [*simp*]: *image-mset f M* = {#} ⟷ *M* = {#}
by (*cases M*) *auto*

lemma *image-mset-If*:
image-mset (λ*x*. *if P x then f x else g x*) *A* =
image-mset f (*filter-mset P A*) + *image-mset g* (*filter-mset* (λ*x*. ¬*P x*) *A*)
by (*induction A*) *auto*

lemma *image-mset-Diff*:
assumes *B* ⊆# *A*
shows *image-mset f* (*A* − *B*) = *image-mset f A* − *image-mset f B*
proof −
have *image-mset f* (*A* − *B* + *B*) = *image-mset f* (*A* − *B*) + *image-mset f B*
by *simp*
also from *assms* **have** *A* − *B* + *B* = *A*
by (*simp add*: *subset-mset.diff-add*)
finally show *?thesis* **by** *simp*
qed

lemma *count-image-mset*:
count (*image-mset f A*) *x* = (∑ *y*∈*f* −‘ {*x*} ∩ *set-mset A*. *count A y*)
proof (*induction A*)
case *empty*
then show *?case* **by** *simp*
next
case (*add x A*)
moreover have ∗: (*if x* = *y then Suc n else n*) = *n* + (*if x* = *y then 1 else 0*)
for *n y*
by *simp*
ultimately show *?case*
by (*auto simp*: *sum.distrib sum.delta′ intro*!: *sum.mono-neutral-left*)
qed

lemma *image-mset-subseteq-mono*: *A* ⊆# *B* ⟹ *image-mset f A* ⊆# *image-mset f B*
by (*metis image-mset-union subset-mset.le-iff-add*)

syntax (*ASCII*)

-comprehension-mset :: *'a* ⇒ *'b* ⇒ *'b multiset* ⇒ *'a multiset* (({#-/. - :# -#}))
syntax
-comprehension-mset :: *'a* ⇒ *'b* ⇒ *'b multiset* ⇒ *'a multiset* (({#-/. - ∈# -#}))
translations
{#*e*. *x* ∈# *M*#} ⇌ *CONST image-mset* (λ*x*. *e*) *M*

syntax (*ASCII*)
-comprehension-mset' :: *'a* ⇒ *'b* ⇒ *'b multiset* ⇒ *bool* ⇒ *'a multiset* (({#-/ | -:# -./ -#}))
syntax
-comprehension-mset' :: *'a* ⇒ *'b* ⇒ *'b multiset* ⇒ *bool* ⇒ *'a multiset* (({#-/ | -∈# -./ -#}))
translations
{#*e* | *x*∈#*M*. *P*#} ↽ {#*e*. *x* ∈# {# *x*∈#*M*. *P*#}#}

This allows to write not just filters like {#*x* ∈# *M*. *x* < *c*#} but also images like {#*x* + *x*. *x* ∈# *M*#} and {#*x*+*x*|*x*∈#*M*. *x*<*c*#}, where the latter is currently displayed as {#*x* + *x*. *x* ∈# {#*x* ∈# *M*. *x* < *c*#}#}.

lemma *in-image-mset*: *y* ∈# {#*f x*. *x* ∈# *M*#} ⟷ *y* ∈ *f* ‘ *set-mset M*
by (*metis set-image-mset*)

functor *image-mset*: *image-mset*
proof −
fix *f g* **show** *image-mset f* ∘ *image-mset g* = *image-mset* (*f* ∘ *g*)
proof
fix *A*
show (*image-mset f* ∘ *image-mset g*) *A* = *image-mset* (*f* ∘ *g*) *A*
by (*induct A*) *simp-all*
qed
show *image-mset id* = *id*
proof
fix *A*
show *image-mset id A* = *id A*
by (*induct A*) *simp-all*
qed
qed

declare
image-mset.id [*simp*]
image-mset.identity [*simp*]

lemma *image-mset-id*[*simp*]: *image-mset id x* = *x*
unfolding *id-def* **by** *auto*

lemma *image-mset-cong*: (⋀*x*. *x* ∈# *M* ⟹ *f x* = *g x*) ⟹ {#*f x*. *x* ∈# *M*#} = {#*g x*. *x* ∈# *M*#}
by (*induct M*) *auto*

lemma *image-mset-cong-pair*:

$(\forall x\ y.\ (x,\ y) \in\# M \longrightarrow f\ x\ y = g\ x\ y) \Longrightarrow \{\#f\ x\ y.\ (x,\ y) \in\# M\#\} = \{\#g\ x\ y.\ (x,\ y) \in\# M\#\}$
by (*metis image-mset-cong split-cong*)

1.8 Further conversions

primrec *mset* :: *'a list* ⇒ *'a multiset* **where**
mset [] = {#} |
mset (*a* # *x*) = *add-mset a* (*mset x*)

lemma *in-multiset-in-set*:
x ∈# *mset xs* ⟷ *x* ∈ *set xs*
by (*induct xs*) *simp-all*

lemma *count-mset*:
count (*mset xs*) *x* = *length* (*filter* (λ*y*. *x* = *y*) *xs*)
by (*induct xs*) *simp-all*

lemma *mset-zero-iff*[*simp*]: (*mset x* = {#}) = (*x* = [])
by (*induct x*) *auto*

lemma *mset-zero-iff-right*[*simp*]: ({#} = *mset x*) = (*x* = [])
by (*induct x*) *auto*

lemma *mset-single-iff*[*iff*]: *mset xs* = {#*x*#} ⟷ *xs* = [*x*]
by (*cases xs*) *auto*

lemma *mset-single-iff-right*[*iff*]: {#*x*#} = *mset xs* ⟷ *xs* = [*x*]
by (*cases xs*) *auto*

lemma *set-mset-mset*[*simp*]: *set-mset* (*mset xs*) = *set xs*
by (*induct xs*) *auto*

lemma *set-mset-comp-mset* [*simp*]: *set-mset* ∘ *mset* = *set*
by (*simp add*: *fun-eq-iff*)

lemma *size-mset* [*simp*]: *size* (*mset xs*) = *length xs*
by (*induct xs*) *simp-all*

lemma *mset-append* [*simp*]: *mset* (*xs* @ *ys*) = *mset xs* + *mset ys*
by (*induct xs arbitrary*: *ys*) *auto*

lemma *mset-filter*: *mset* (*filter P xs*) = {#*x* ∈# *mset xs*. *P x* #}
by (*induct xs*) *simp-all*

lemma *mset-rev* [*simp*]:
mset (*rev xs*) = *mset xs*
by (*induct xs*) *simp-all*

lemma *surj-mset*: *surj mset*
apply (*unfold surj-def*)
apply (*rule allI*)
apply (*rule-tac M = y* **in** *multiset-induct*)
apply *auto*
apply (*rule-tac x = x # xa* **in** *exI*)
apply *auto*
done

lemma *distinct-count-atmost-1*:
distinct x = (∀ a. count (mset x) a = (if a ∈ set x then 1 else 0))
proof (*induct x*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons x xs*) **show** *?case* (**is** *?lhs ⟷ ?rhs*)
proof
assume *?lhs* **then show** *?rhs* **using** *Cons* **by** *simp*
next
assume *?rhs* **then have** *x ∉ set xs*
by (*simp split*: *if-splits*)
moreover from ‹*?rhs*› **have** (*∀ a. count (mset xs) a =*
(if a ∈ set xs then 1 else 0))
by (*auto split*: *if-splits simp add*: *count-eq-zero-iff*)
ultimately show *?lhs* **using** *Cons* **by** *simp*
qed
qed

lemma *mset-eq-setD*:
assumes *mset xs = mset ys*
shows *set xs = set ys*
proof −
from *assms* **have** *set-mset (mset xs) = set-mset (mset ys)*
by *simp*
then show *?thesis* **by** *simp*
qed

lemma *set-eq-iff-mset-eq-distinct*:
distinct x ⟹ distinct y ⟹
(set x = set y) = (mset x = mset y)
by (*auto simp*: *multiset-eq-iff distinct-count-atmost-1*)

lemma *set-eq-iff-mset-remdups-eq*:
(set x = set y) = (mset (remdups x) = mset (remdups y))
apply (*rule iffI*)
apply (*simp add*: *set-eq-iff-mset-eq-distinct*[*THEN iffD1*])
apply (*drule distinct-remdups* [*THEN distinct-remdups*
[*THEN set-eq-iff-mset-eq-distinct* [*THEN iffD2*]]])
apply *simp*
done

lemma *mset-compl-union* [*simp*]: *mset* [*x*←*xs*. *P x*] + *mset* [*x*←*xs*. ¬*P x*] = *mset xs*
by (*induct xs*) *auto*

lemma *nth-mem-mset*: *i* < *length ls* ⟹ (*ls* ! *i*) ∈# *mset ls*
proof (*induct ls arbitrary*: *i*)
case *Nil*
then show *?case* **by** *simp*
next
case *Cons*
then show *?case* **by** (*cases i*) *auto*
qed

lemma *mset-remove1*[*simp*]: *mset* (*remove1 a xs*) = *mset xs* − {#*a*#}
by (*induct xs*) (*auto simp add*: *multiset-eq-iff*)

lemma *mset-eq-length*:
assumes *mset xs* = *mset ys*
shows *length xs* = *length ys*
using *assms* **by** (*metis size-mset*)

lemma *mset-eq-length-filter*:
assumes *mset xs* = *mset ys*
shows *length* (*filter* (λ*x*. *z* = *x*) *xs*) = *length* (*filter* (λ*y*. *z* = *y*) *ys*)
using *assms* **by** (*metis count-mset*)

lemma *fold-multiset-equiv*:
assumes *f*: ⋀*x y*. *x* ∈ *set xs* ⟹ *y* ∈ *set xs* ⟹ *f x* ∘ *f y* = *f y* ∘ *f x*
and *equiv*: *mset xs* = *mset ys*
shows *List.fold f xs* = *List.fold f ys*
using *f equiv* [*symmetric*]
proof (*induct xs arbitrary*: *ys*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons x xs*)
then have ∗: *set ys* = *set* (*x* # *xs*)
by (*blast dest*: *mset-eq-setD*)
have ⋀*x y*. *x* ∈ *set ys* ⟹ *y* ∈ *set ys* ⟹ *f x* ∘ *f y* = *f y* ∘ *f x*
by (*rule Cons.prems*(*1*)) (*simp-all add*: ∗)
moreover from ∗ **have** *x* ∈ *set ys*
by *simp*
ultimately have *List.fold f ys* = *List.fold f* (*remove1 x ys*) ∘ *f x*
by (*fact fold-remove1-split*)
moreover from *Cons.prems* **have** *List.fold f xs* = *List.fold f* (*remove1 x ys*)
by (*auto intro*: *Cons.hyps*)
ultimately show *?case* **by** *simp*
qed

lemma *mset-insort* [*simp*]: *mset* (*insort x xs*) = *add-mset x* (*mset xs*)
by (*induct xs*) *simp-all*

lemma *mset-map*[*simp*]: *mset* (*map f xs*) = *image-mset f* (*mset xs*)
by (*induct xs*) *simp-all*

global-interpretation *mset-set*: *folding add-mset* {#}
defines *mset-set* = *folding.F add-mset* {#}
by *standard* (*simp add*: *fun-eq-iff*)

lemma *count-mset-set* [*simp*]:
finite A ⟹ *x* ∈ *A* ⟹ *count* (*mset-set A*) *x* = *1* (**is** *PROP ?P*)
¬ *finite A* ⟹ *count* (*mset-set A*) *x* = *0* (**is** *PROP ?Q*)
x ∉ *A* ⟹ *count* (*mset-set A*) *x* = *0* (**is** *PROP ?R*)
proof −
have ∗: *count* (*mset-set A*) *x* = *0* **if** *x* ∉ *A* **for** *A*
proof (*cases finite A*)
case *False* **then show** *?thesis* **by** *simp*
next
case *True* **from** *True* ‹*x* ∉ *A*› **show** *?thesis* **by** (*induct A*) *auto*
qed
then show *PROP ?P PROP ?Q PROP ?R*
by (*auto elim*!: *Set.set-insert*)
qed — TODO: maybe define *mset-set* also in terms of *Abs-multiset*

lemma *elem-mset-set*[*simp*, *intro*]: *finite A* ⟹ *x* ∈# *mset-set A* ⟷ *x* ∈ *A*
by (*induct A rule*: *finite-induct*) *simp-all*

lemma *mset-set-Union*:
finite A ⟹ *finite B* ⟹ *A* ∩ *B* = {} ⟹ *mset-set* (*A* ∪ *B*) = *mset-set A* + *mset-set B*
by (*induction A rule*: *finite-induct*) *auto*

lemma *filter-mset-mset-set* [*simp*]:
finite A ⟹ *filter-mset P* (*mset-set A*) = *mset-set* {*x*∈*A*. *P x*}
proof (*induction A rule*: *finite-induct*)
case (*insert x A*)
from *insert.hyps* **have** *filter-mset P* (*mset-set* (*insert x A*)) =
filter-mset P (*mset-set A*) + *mset-set* (*if P x then* {*x*} *else* {})
by *simp*
also have *filter-mset P* (*mset-set A*) = *mset-set* {*x*∈*A*. *P x*}
by (*rule insert.IH*)
also from *insert.hyps*
have ... + *mset-set* (*if P x then* {*x*} *else* {}) =
mset-set ({*x* ∈ *A*. *P x*} ∪ (*if P x then* {*x*} *else* {})) (**is** - = *mset-set ?A*)
by (*intro mset-set-Union* [*symmetric*]) *simp-all*
also from *insert.hyps* **have** *?A* = {*y*∈*insert x A*. *P y*} **by** *auto*
finally show *?case* **.**

qed *simp-all*

lemma *mset-set-Diff*:
assumes *finite A B* ⊆ *A*
shows *mset-set* (*A* − *B*) = *mset-set A* − *mset-set B*
proof −
from *assms* **have** *mset-set* ((*A* − *B*) ∪ *B*) = *mset-set* (*A* − *B*) + *mset-set B*
by (*intro mset-set-Union*) (*auto dest*: *finite-subset*)
also from *assms* **have** *A* − *B* ∪ *B* = *A* **by** *blast*
finally show *?thesis* **by** *simp*
qed

lemma *mset-set-set*: *distinct xs* ⟹ *mset-set* (*set xs*) = *mset xs*
by (*induction xs*) *simp-all*

context *linorder*
begin

definition *sorted-list-of-multiset* :: *'a multiset* ⇒ *'a list*
where
sorted-list-of-multiset M = *fold-mset insort* [] *M*

lemma *sorted-list-of-multiset-empty* [*simp*]:
sorted-list-of-multiset {#} = []
by (*simp add*: *sorted-list-of-multiset-def*)

lemma *sorted-list-of-multiset-singleton* [*simp*]:
sorted-list-of-multiset {#*x*#} = [*x*]
proof −
interpret *comp-fun-commute insort* **by** (*fact comp-fun-commute-insort*)
show *?thesis* **by** (*simp add*: *sorted-list-of-multiset-def*)
qed

lemma *sorted-list-of-multiset-insert* [*simp*]:
sorted-list-of-multiset (*add-mset x M*) = *List.insort x* (*sorted-list-of-multiset M*)
proof −
interpret *comp-fun-commute insort* **by** (*fact comp-fun-commute-insort*)
show *?thesis* **by** (*simp add*: *sorted-list-of-multiset-def*)
qed

end

lemma *mset-sorted-list-of-multiset* [*simp*]:
mset (*sorted-list-of-multiset M*) = *M*
by (*induct M*) *simp-all*

lemma *sorted-list-of-multiset-mset* [*simp*]:
sorted-list-of-multiset (*mset xs*) = *sort xs*
by (*induct xs*) *simp-all*

lemma *finite-set-mset-mset-set*[*simp*]:
finite A ⟹ *set-mset* (*mset-set A*) = *A*
by (*induct A rule*: *finite-induct*) *simp-all*

lemma *mset-set-empty-iff*: *mset-set A* = {#} ⟷ *A* = {} ∨ *infinite A*
using *finite-set-mset-mset-set* **by** *fastforce*

lemma *infinite-set-mset-mset-set*:
¬ *finite A* ⟹ *set-mset* (*mset-set A*) = {}
by *simp*

lemma *set-sorted-list-of-multiset* [*simp*]:
set (*sorted-list-of-multiset M*) = *set-mset M*
by (*induct M*) (*simp-all add*: *set-insort*)

lemma *sorted-list-of-mset-set* [*simp*]:
sorted-list-of-multiset (*mset-set A*) = *sorted-list-of-set A*
by (*cases finite A*) (*induct A rule*: *finite-induct*, *simp-all*)

lemma *mset-upt* [*simp*]: *mset* [*m*..<*n*] = *mset-set* {*m*..<*n*}
by (*induction n*) (*simp-all add*: *atLeastLessThanSuc*)

lemma *image-mset-map-of*:
distinct (*map fst xs*) ⟹ {#*the* (*map-of xs i*). *i* ∈# *mset* (*map fst xs*)#} = *mset* (*map snd xs*)
proof (*induction xs*)
case (*Cons x xs*)
have {#*the* (*map-of* (*x* # *xs*) *i*). *i* ∈# *mset* (*map fst* (*x* # *xs*))#} =
add-mset (*snd x*) {#*the* (*if i* = *fst x then Some* (*snd x*) *else map-of xs i*).
i ∈# *mset* (*map fst xs*)#} (**is** - = *add-mset* - *?A*) **by** *simp*
also from *Cons.prems* **have** *?A* = {#*the* (*map-of xs i*). *i* :# *mset* (*map fst xs*)#}
by (*cases x*, *intro image-mset-cong*) (*auto simp*: *in-multiset-in-set*)
also from *Cons.prems* **have** ... = *mset* (*map snd xs*) **by** (*intro Cons.IH*) *simp-all*
finally show *?case* **by** *simp*
qed *simp-all*

lemma *image-mset-mset-set*:
assumes *inj-on f A*
shows *image-mset f* (*mset-set A*) = *mset-set* (*f* ‘ *A*)
proof *cases*
assume *finite A*
from *this* ‹*inj-on f A*› **show** *?thesis*
by (*induct A*) *auto*
next
assume *infinite A*

from *this* ‹*inj-on f A*› **have** *infinite* (*f* ‘ *A*)
using *finite-imageD* **by** *blast*
from ‹*infinite A*› ‹*infinite* (*f* ‘ *A*)› **show** *?thesis* **by** *simp*
qed

1.9 More properties of the replicate and repeat operations

lemma *in-replicate-mset*[*simp*]: *x* ∈# *replicate-mset n y* ⟷ *n* > *0* ∧ *x* = *y*
unfolding *replicate-mset-def* **by** (*induct n*) *auto*

lemma *set-mset-replicate-mset-subset*[*simp*]: *set-mset* (*replicate-mset n x*) = (*if n* = *0 then* {} *else* {*x*})
by (*auto split*: *if-splits*)

lemma *size-replicate-mset*[*simp*]: *size* (*replicate-mset n M*) = *n*
by (*induct n*, *simp-all*)

lemma *count-le-replicate-mset-subset-eq*: *n* ≤ *count M x* ⟷ *replicate-mset n x* ⊆# *M*
by (*auto simp add*: *mset-subset-eqI*) (*metis count-replicate-mset subseteq-mset-def*)

lemma *filter-eq-replicate-mset*: {#*y* ∈# *D*. *y* = *x*#} = *replicate-mset* (*count D x*) *x*
by (*induct D*) *simp-all*

lemma *replicate-count-mset-eq-filter-eq*:
replicate (*count* (*mset xs*) *k*) *k* = *filter* (*HOL.eq k*) *xs*
by (*induct xs*) *auto*

lemma *replicate-mset-eq-empty-iff* [*simp*]:
replicate-mset n a = {#} ⟷ *n* = *0*
by (*induct n*) *simp-all*

lemma *replicate-mset-eq-iff*:
replicate-mset m a = *replicate-mset n b* ⟷
m = *0* ∧ *n* = *0* ∨ *m* = *n* ∧ *a* = *b*
by (*auto simp add*: *multiset-eq-iff*)

lemma *repeat-mset-cancel1*: *repeat-mset a A* = *repeat-mset a B* ⟷ *A* = *B* ∨ *a* = *0*
by (*auto simp*: *multiset-eq-iff*)

lemma *repeat-mset-cancel2*: *repeat-mset a A* = *repeat-mset b A* ⟷ *a* = *b* ∨ *A* = {#}
by (*auto simp*: *multiset-eq-iff*)

lemma *repeat-mset-eq-empty-iff*: *repeat-mset n A* = {#} ⟷ *n* = *0* ∨ *A* = {#}
by (*cases n*) *auto*

lemma *image-replicate-mset* [*simp*]:
  image-mset f (*replicate-mset n a*) = *replicate-mset n* (*f a*)
  by (*induct n*) *simp-all*

1.10 Big operators

locale *comm-monoid-mset* = *comm-monoid*
begin

interpretation *comp-fun-commute f*
  by *standard* (*simp add*: *fun-eq-iff left-commute*)

interpretation *comp?*: *comp-fun-commute f* ∘ *g*
  by (*fact comp-comp-fun-commute*)

context
begin

definition *F* :: *'a multiset* ⇒ *'a*
  where *eq-fold*: *F M* = *fold-mset f* **1** *M*

lemma *empty* [*simp*]: *F* {#} = **1**
  by (*simp add*: *eq-fold*)

lemma *singleton* [*simp*]: *F* {#*x*#} = *x*
proof −
  interpret *comp-fun-commute*
    by *standard* (*simp add*: *fun-eq-iff left-commute*)
  show *?thesis* **by** (*simp add*: *eq-fold*)
qed

lemma *union* [*simp*]: *F* (*M* + *N*) = *F M* ∗ *F N*
proof −
  interpret *comp-fun-commute f*
    by *standard* (*simp add*: *fun-eq-iff left-commute*)
  show *?thesis*
    by (*induct N*) (*simp-all add*: *left-commute eq-fold*)
qed

lemma *add-mset* [*simp*]: *F* (*add-mset x N*) = *x* ∗ *F N*
  unfolding *add-mset-add-single*[*of x N*] *union* **by** (*simp add*: *ac-simps*)

lemma *insert* [*simp*]:
  shows *F* (*image-mset g* (*add-mset x A*)) = *g x* ∗ *F* (*image-mset g A*)
  by (*simp add*: *eq-fold*)

lemma *remove*:
  assumes *x* ∈# *A*
  shows *F A* = *x* ∗ *F* (*A* − {#*x*#})

using *multi-member-split*[*OF assms*] **by** *auto*

lemma *neutral*:
$\forall x{\in}\#A.\ x = \mathbf{1} \Longrightarrow F\ A = \mathbf{1}$
by (*induct A*) *simp-all*

lemma *neutral-const* [*simp*]:
F (*image-mset* (λ-. **1**) *A*) = **1**
by (*simp add*: *neutral*)

private lemma *F-image-mset-product*:
F {#*g x j* ∗ *F* {#*g i j*. *i* ∈# *A*#}. *j* ∈# *B*#} =
F (*image-mset* (*g x*) *B*) ∗ *F* {#*F* {#*g i j*. *i* ∈# *A*#}. *j* ∈# *B*#}
by (*induction B*) (*simp-all add*: *left-commute semigroup.assoc semigroup-axioms*)

lemma *commute*:
F (*image-mset* (λ*i*. *F* (*image-mset* (*g i*) *B*)) *A*) =
F (*image-mset* (λ*j*. *F* (*image-mset* (λ*i*. *g i j*) *A*)) *B*)
apply (*induction A*, *simp*)
apply (*induction B*, *auto simp add*: *F-image-mset-product ac-simps*)
done

lemma *distrib*: *F* (*image-mset* (λ*x*. *g x* ∗ *h x*) *A*) = *F* (*image-mset g A*) ∗ *F* (*image-mset h A*)
by (*induction A*) (*auto simp*: *ac-simps*)

lemma *union-disjoint*:
A ∩# *B* = {#} ⟹ *F* (*image-mset g* (*A* ∪# *B*)) = *F* (*image-mset g A*) ∗ *F* (*image-mset g B*)
by (*induction A*) (*auto simp*: *ac-simps*)

end
end

lemma *comp-fun-commute-plus-mset*[*simp*]: *comp-fun-commute* (*op* + :: *'a multi-set* ⇒ - ⇒ -)
by *standard* (*simp add*: *add-ac comp-def*)

declare *comp-fun-commute.fold-mset-add-mset*[*OF comp-fun-commute-plus-mset*, *simp*]

lemma *in-mset-fold-plus-iff*[*iff*]: *x* ∈# *fold-mset* (*op* +) *M NN* ⟷ *x* ∈# *M* ∨ (∃ *N*. *N* ∈# *NN* ∧ *x* ∈# *N*)
by (*induct NN*) *auto*

context *comm-monoid-add*
begin

sublocale *sum-mset*: *comm-monoid-mset plus 0*

defines *sum-mset* = *sum-mset.F* **..**

lemma (**in** *semiring-1*) *sum-mset-replicate-mset* [*simp*]:
sum-mset (*replicate-mset n a*) = *of-nat n* ∗ *a*
by (*induct n*) (*simp-all add*: *algebra-simps*)

lemma *sum-unfold-sum-mset*:
sum f A = *sum-mset* (*image-mset f* (*mset-set A*))
by (*cases finite A*) (*induct A rule*: *finite-induct*, *simp-all*)

lemma *sum-mset-delta*: *sum-mset* (*image-mset* (λ*x*. *if x* = *y then c else 0*) *A*) =
c ∗ *count A y*
by (*induction A*) *simp-all*

lemma *sum-mset-delta'*: *sum-mset* (*image-mset* (λ*x*. *if y* = *x then c else 0*) *A*) =
c ∗ *count A y*
by (*induction A*) *simp-all*

end

lemma *of-nat-sum-mset* [*simp*]:
of-nat (*sum-mset M*) = *sum-mset* (*image-mset of-nat M*)
by (*induction M*) *auto*

lemma *sum-mset-0-iff* [*simp*]:
sum-mset M = (*0*::*'a*::*canonically-ordered-monoid-add*)
⟷ (∀ *x* ∈ *set-mset M*. *x* = *0*)
by(*induction M*) *auto*

lemma *sum-mset-diff*:
fixes *M N* :: (*'a* :: *ordered-cancel-comm-monoid-diff*) *multiset*
shows *N* ⊆# *M* ⟹ *sum-mset* (*M* − *N*) = *sum-mset M* − *sum-mset N*
by (*metis add-diff-cancel-right' sum-mset.union subset-mset.diff-add*)

lemma *size-eq-sum-mset*: *size M* = *sum-mset* (*image-mset* (λ-. *1*) *M*)
proof (*induct M*)
case *empty* **then show** *?case* **by** *simp*
next
case (*add x M*) **then show** *?case*
by (*cases x* ∈ *set-mset M*)
(*simp-all add*: *size-multiset-overloaded-eq not-in-iff sum.If-cases Diff-eq*[*symmetric*]
sum.remove)
qed

lemma *size-mset-set* [*simp*]: *size* (*mset-set A*) = *card A*
by (*simp only*: *size-eq-sum-mset card-eq-sum sum-unfold-sum-mset*)

lemma *sum-mset-sum-list*: *sum-mset* (*mset xs*) = *sum-list xs*
by (*induction xs*) *auto*

syntax (*ASCII*)
-sum-mset-image :: *pttrn* ⇒ *'b set* ⇒ *'a* ⇒ *'a*::*comm-monoid-add* (*(3SUM -:#-. -) [0, 51, 10] 10*)
syntax
-sum-mset-image :: *pttrn* ⇒ *'b set* ⇒ *'a* ⇒ *'a*::*comm-monoid-add* (*(3∑ -∈#-. -) [0, 51, 10] 10*)
translations
∑ *i* ∈# *A*. *b* ⇌ *CONST sum-mset* (*CONST image-mset* (λ*i*. *b*) *A*)

lemma *sum-mset-distrib-left*:
fixes *f* :: *'a* ⇒ *'b*::*semiring-0*
shows *c* ∗ (∑ *x* ∈# *M*. *f x*) = (∑ *x* ∈# *M*. *c* ∗ *f*(*x*))
by (*induction M*) (*simp-all add*: *distrib-left*)

lemma *sum-mset-distrib-right*:
fixes *f* :: *'a* ⇒ *'b*::*semiring-0*
shows (∑ *b* ∈# *B*. *f b*) ∗ *a* = (∑ *b* ∈# *B*. *f b* ∗ *a*)
by (*induction B*) (*auto simp*: *distrib-right*)

lemma *sum-mset-constant* [*simp*]:
fixes *y* :: *'b*::*semiring-1*
shows ‹(∑ *x*∈#*A*. *y*) = *of-nat* (*size A*) ∗ *y*›
by (*induction A*) (*auto simp*: *algebra-simps*)

lemma (**in** *ordered-comm-monoid-add*) *sum-mset-mono*:
assumes ⋀*i*. *i* ∈# *K* ⟹ *f i* ≤ *g i*
shows *sum-mset* (*image-mset f K*) ≤ *sum-mset* (*image-mset g K*)
using *assms* **by** (*induction K*) (*simp-all add*: *local.add-mono*)

lemma *sum-mset-product*:
fixes *f* :: *'a*::{*comm-monoid-add*,*times*} ⇒ *'b*::*semiring-0*
shows (∑ *i* ∈# *A*. *f i*) ∗ (∑ *i* ∈# *B*. *g i*) = (∑ *i*∈#*A*. ∑ *j*∈#*B*. *f i* ∗ *g j*)
by (*subst sum-mset.commute*) (*simp add*: *sum-mset-distrib-left sum-mset-distrib-right*)

abbreviation *Union-mset* :: *'a multiset multiset* ⇒ *'a multiset* (⋃#- [*900*] *900*)
where ⋃# *MM* ≡ *sum-mset MM* — FIXME ambiguous notation – could likewise refer to ⨆#

lemma *set-mset-Union-mset*[*simp*]: *set-mset* (⋃# *MM*) = (⋃ *M* ∈ *set-mset MM*. *set-mset M*)
by (*induct MM*) *auto*

lemma *in-Union-mset-iff*[*iff*]: *x* ∈# ⋃# *MM* ⟷ (∃ *M*. *M* ∈# *MM* ∧ *x* ∈# *M*)
by (*induct MM*) *auto*

lemma *count-sum*:
count (*sum f A*) *x* = *sum* (λ*a*. *count* (*f a*) *x*) *A*

by (*induct A rule*: *infinite-finite-induct*) *simp-all*

lemma *sum-eq-empty-iff*:
assumes *finite A*
shows *sum f A* = {#} ⟷ (∀ *a*∈*A*. *f a* = {#})
using *assms* **by** *induct simp-all*

lemma *Union-mset-empty-conv*[*simp*]: ⋃# *M* = {#} ⟷ (∀ *i*∈#*M*. *i* = {#})
by (*induction M*) *auto*

context *comm-monoid-mult*
begin

sublocale *prod-mset*: *comm-monoid-mset times 1*
defines *prod-mset* = *prod-mset.F* **..**

lemma *prod-mset-empty*:
prod-mset {#} = *1*
by (*fact prod-mset.empty*)

lemma *prod-mset-singleton*:
prod-mset {#*x*#} = *x*
by (*fact prod-mset.singleton*)

lemma *prod-mset-Un*:
prod-mset (*A* + *B*) = *prod-mset A* ∗ *prod-mset B*
by (*fact prod-mset.union*)

lemma *prod-mset-replicate-mset* [*simp*]:
prod-mset (*replicate-mset n a*) = *a* ^ *n*
by (*induct n*) *simp-all*

lemma *prod-unfold-prod-mset*:
prod f A = *prod-mset* (*image-mset f* (*mset-set A*))
by (*cases finite A*) (*induct A rule*: *finite-induct*, *simp-all*)

lemma *prod-mset-multiplicity*:
prod-mset M = *prod* (λ*x*. *x* ^ *count M x*) (*set-mset M*)
by (*simp add*: *fold-mset-def prod.eq-fold prod-mset.eq-fold funpow-times-power comp-def*)

lemma *prod-mset-delta*: *prod-mset* (*image-mset* (λ*x*. *if x* = *y then c else 1*) *A*) = *c* ^ *count A y*
by (*induction A*) *simp-all*

lemma *prod-mset-delta′*: *prod-mset* (*image-mset* (λ*x*. *if y* = *x then c else 1*) *A*) = *c* ^ *count A y*
by (*induction A*) *simp-all*

end

syntax (*ASCII*)
-prod-mset-image :: *pttrn* ⇒ *'b set* ⇒ *'a* ⇒ *'a*::*comm-monoid-mult* ((*3PROD -:#-. -*) [*0*, *51*, *10*] *10*)
syntax
-prod-mset-image :: *pttrn* ⇒ *'b set* ⇒ *'a* ⇒ *'a*::*comm-monoid-mult* ((*3*∏ *-∈#-. -*) [*0*, *51*, *10*] *10*)
translations
∏ *i* ∈# *A*. *b* ⇌ *CONST prod-mset* (*CONST image-mset* (λ*i*. *b*) *A*)

lemma (**in** *comm-monoid-mult*) *prod-mset-subset-imp-dvd*:
assumes *A* ⊆# *B*
shows *prod-mset A dvd prod-mset B*
proof −
from *assms* **have** *B* = (*B* − *A*) + *A* **by** (*simp add*: *subset-mset.diff-add*)
also have *prod-mset* ... = *prod-mset* (*B* − *A*) ∗ *prod-mset A* **by** *simp*
also have *prod-mset A dvd* ... **by** *simp*
finally show *?thesis* **.**
qed

lemma (**in** *comm-monoid-mult*) *dvd-prod-mset*:
assumes *x* ∈# *A*
shows *x dvd prod-mset A*
using *assms prod-mset-subset-imp-dvd* [*of* {#*x*#} *A*] **by** *simp*

lemma (**in** *semidom*) *prod-mset-zero-iff* [*iff*]:
prod-mset A = *0* ⟷ *0* ∈# *A*
by (*induct A*) *auto*

lemma (**in** *semidom-divide*) *prod-mset-diff*:
assumes *B* ⊆# *A* **and** *0* ∉# *B*
shows *prod-mset* (*A* − *B*) = *prod-mset A div prod-mset B*
proof −
from *assms* **obtain** *C* **where** *A* = *B* + *C*
by (*metis subset-mset.add-diff-inverse*)
with *assms* **show** *?thesis* **by** *simp*
qed

lemma (**in** *semidom-divide*) *prod-mset-minus*:
assumes *a* ∈# *A* **and** *a* ≠ *0*
shows *prod-mset* (*A* − {#*a*#}) = *prod-mset A div a*
using *assms prod-mset-diff* [*of* {#*a*#} *A*] **by** *auto*

lemma (**in** *algebraic-semidom*) *is-unit-prod-mset-iff*:
is-unit (*prod-mset A*) ⟷ (∀ *x* ∈# *A*. *is-unit x*)
by (*induct A*) (*auto simp*: *is-unit-mult-iff*)

lemma (**in** *normalization-semidom*) *normalize-prod-mset*:

normalize (*prod-mset A*) = *prod-mset* (*image-mset normalize A*)
by (*induct A*) (*simp-all add*: *normalize-mult*)

lemma (**in** *normalization-semidom*) *normalized-prod-msetI*:
assumes ⋀*a*. *a* ∈# *A* ⟹ *normalize a* = *a*
shows *normalize* (*prod-mset A*) = *prod-mset A*
proof −
from *assms* **have** *image-mset normalize A* = *A*
by (*induct A*) *simp-all*
then show *?thesis* **by** (*simp add*: *normalize-prod-mset*)
qed

lemma *prod-mset-prod-list*: *prod-mset* (*mset xs*) = *prod-list xs*
by (*induct xs*) *auto*

1.11 Alternative representations

1.11.1 Lists

context *linorder*
begin

lemma *mset-insort* [*simp*]:
mset (*insort-key k x xs*) = *add-mset x* (*mset xs*)
by (*induct xs*) *simp-all*

lemma *mset-sort* [*simp*]:
mset (*sort-key k xs*) = *mset xs*
by (*induct xs*) *simp-all*

This lemma shows which properties suffice to show that a function f with $f\ xs = ys$ behaves like sort.

lemma *properties-for-sort-key*:
assumes *mset ys* = *mset xs*
and ⋀*k*. *k* ∈ *set ys* ⟹ *filter* (λ*x*. *f k* = *f x*) *ys* = *filter* (λ*x*. *f k* = *f x*) *xs*
and *sorted* (*map f ys*)
shows *sort-key f xs* = *ys*
using *assms*
proof (*induct xs arbitrary*: *ys*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons x xs*)
from *Cons.prems*(*2*) **have**
∀ *k* ∈ *set ys*. *filter* (λ*x*. *f k* = *f x*) (*remove1 x ys*) = *filter* (λ*x*. *f k* = *f x*) *xs*
by (*simp add*: *filter-remove1*)
with *Cons.prems* **have** *sort-key f xs* = *remove1 x ys*
by (*auto intro*!: *Cons.hyps simp add*: *sorted-map-remove1*)
moreover from *Cons.prems* **have** *x* ∈# *mset ys*
by *auto*
then have *x* ∈ *set ys*

```
    by simp
  ultimately show ?case using Cons.prems by (simp add: insort-key-remove1)
qed

lemma properties-for-sort:
  assumes multiset: mset ys = mset xs
    and sorted ys
  shows sort xs = ys
proof (rule properties-for-sort-key)
  from multiset show mset ys = mset xs .
  from ‹sorted ys› show sorted (map (λx. x) ys) by simp
  from multiset have length (filter (λy. k = y) ys) = length (filter (λx. k = x)
xs) for k
    by (rule mset-eq-length-filter)
  then have replicate (length (filter (λy. k = y) ys)) k =
    replicate (length (filter (λx. k = x) xs)) k for k
    by simp
  then show k ∈ set ys ⟹ filter (λy. k = y) ys = filter (λx. k = x) xs for k
    by (simp add: replicate-length-filter)
qed

lemma sort-key-inj-key-eq:
  assumes mset-equal: mset xs = mset ys
    and inj-on f (set xs)
    and sorted (map f ys)
  shows sort-key f xs = ys
proof (rule properties-for-sort-key)
  from mset-equal
  show mset ys = mset xs by simp
  from ‹sorted (map f ys)›
  show sorted (map f ys) .
  show [x←ys . f k = f x] = [x←xs . f k = f x] if k ∈ set ys for k
  proof −
    from mset-equal
    have set-equal: set xs = set ys by (rule mset-eq-setD)
    with that have insert k (set ys) = set ys by auto
    with ‹inj-on f (set xs)› have inj: inj-on f (insert k (set ys))
      by (simp add: set-equal)
    from inj have [x←ys . f k = f x] = filter (HOL.eq k) ys
      by (auto intro!: inj-on-filter-key-eq)
    also have ... = replicate (count (mset ys) k) k
      by (simp add: replicate-count-mset-eq-filter-eq)
    also have ... = replicate (count (mset xs) k) k
      using mset-equal by simp
    also have ... = filter (HOL.eq k) xs
      by (simp add: replicate-count-mset-eq-filter-eq)
    also have ... = [x←xs . f k = f x]
     using inj by (auto intro!: inj-on-filter-key-eq [symmetric] simp add: set-equal)
    finally show ?thesis .
```

```
  qed
qed

lemma sort-key-eq-sort-key:
  assumes mset xs = mset ys
    and inj-on f (set xs)
  shows sort-key f xs = sort-key f ys
  by (rule sort-key-inj-key-eq) (simp-all add: assms)

lemma sort-key-by-quicksort:
  sort-key f xs = sort-key f [x←xs. f x < f (xs ! (length xs div 2))]
    @ [x←xs. f x = f (xs ! (length xs div 2))]
    @ sort-key f [x←xs. f x > f (xs ! (length xs div 2))] (is sort-key f ?lhs = ?rhs)
proof (rule properties-for-sort-key)
  show mset ?rhs = mset ?lhs
    by (rule multiset-eqI) (auto simp add: mset-filter)
  show sorted (map f ?rhs)
    by (auto simp add: sorted-append intro: sorted-map-same)
next
  fix l
  assume l ∈ set ?rhs
  let ?pivot = f (xs ! (length xs div 2))
  have *: ⋀x. f l = f x ⟷ f x = f l by auto
  have [x ← sort-key f xs . f x = f l] = [x ← xs. f x = f l]
   unfolding filter-sort by (rule properties-for-sort-key) (auto intro: sorted-map-same)
  with * have **: [x ← sort-key f xs . f l = f x] = [x ← xs. f l = f x] by simp
  have ⋀x P. P (f x) ?pivot ∧ f l = f x ⟷ P (f l) ?pivot ∧ f l = f x by auto
  then have ⋀P. [x ← sort-key f xs . P (f x) ?pivot ∧ f l = f x] =
    [x ← sort-key f xs. P (f l) ?pivot ∧ f l = f x] by simp
  note *** = this [of op <] this [of op >] this [of op =]
  show [x ← ?rhs. f l = f x] = [x ← ?lhs. f l = f x]
  proof (cases f l ?pivot rule: linorder-cases)
    case less
    then have f l ≠ ?pivot and ¬ f l > ?pivot by auto
    with less show ?thesis
      by (simp add: filter-sort [symmetric] ** ***)
  next
    case equal then show ?thesis
      by (simp add: * less-le)
  next
    case greater
    then have f l ≠ ?pivot and ¬ f l < ?pivot by auto
    with greater show ?thesis
      by (simp add: filter-sort [symmetric] ** ***)
  qed
qed

lemma sort-by-quicksort:
  sort xs = sort [x←xs. x < xs ! (length xs div 2)]
```

```
  @ [x←xs. x = xs ! (length xs div 2)]
  @ sort [x←xs. x > xs ! (length xs div 2)] (is sort ?lhs = ?rhs)
 using sort-key-by-quicksort [of λx. x, symmetric] by simp
```

A stable parametrized quicksort

```
definition part :: ('b ⇒ 'a) ⇒ 'a ⇒ 'b list ⇒ 'b list × 'b list × 'b list where
 part f pivot xs = ([x ← xs. f x < pivot], [x ← xs. f x = pivot], [x ← xs. pivot <
f x])

lemma part-code [code]:
 part f pivot [] = ([], [], [])
 part f pivot (x # xs) = (let (lts, eqs, gts) = part f pivot xs; x' = f x in
   if x' < pivot then (x # lts, eqs, gts)
   else if x' > pivot then (lts, eqs, x # gts)
   else (lts, x # eqs, gts))
 by (auto simp add: part-def Let-def split-def)

lemma sort-key-by-quicksort-code [code]:
 sort-key f xs =
   (case xs of
     [] ⇒ []
   | [x] ⇒ xs
   | [x, y] ⇒ (if f x ≤ f y then xs else [y, x])
   | - ⇒
       let (lts, eqs, gts) = part f (f (xs ! (length xs div 2))) xs
       in sort-key f lts @ eqs @ sort-key f gts)
proof (cases xs)
 case Nil then show ?thesis by simp
next
 case (Cons - ys) note hyps = Cons show ?thesis
 proof (cases ys)
   case Nil with hyps show ?thesis by simp
 next
   case (Cons - zs) note hyps = hyps Cons show ?thesis
   proof (cases zs)
     case Nil with hyps show ?thesis by auto
   next
     case Cons
     from sort-key-by-quicksort [of f xs]
     have sort-key f xs = (let (lts, eqs, gts) = part f (f (xs ! (length xs div 2))) xs
       in sort-key f lts @ eqs @ sort-key f gts)
     by (simp only: split-def Let-def part-def fst-conv snd-conv)
     with hyps Cons show ?thesis by (simp only: list.cases)
   qed
 qed
qed

end
```

hide-const (**open**) *part*

lemma *mset-remdups-subset-eq*: *mset* (*remdups xs*) $\subseteq\#$ *mset xs*
by (*induct xs*) (*auto intro*: *subset-mset.order-trans*)

lemma *mset-update*:
$i <$ *length ls* $\Longrightarrow$ *mset* (*ls*[$i :=$ *v*]) = *add-mset v* (*mset ls* $-$ {#*ls* ! *i*#})
proof (*induct ls arbitrary*: *i*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons x xs*)
show *?case*
proof (*cases i*)
case *0* **then show** *?thesis* **by** *simp*
next
case (*Suc i'*)
with *Cons* **show** *?thesis*
by (*cases* ‹*x* = *xs* ! *i'*›) *auto*
qed
qed

lemma *mset-swap*:
$i <$ *length ls* $\Longrightarrow$ $j <$ *length ls* $\Longrightarrow$
mset (*ls*[$j :=$ *ls* ! *i*, $i :=$ *ls* ! *j*]) = *mset ls*
by (*cases i* = *j*) (*simp-all add*: *mset-update nth-mem-mset*)

1.12 The multiset order

1.12.1 Well-foundedness

definition *mult1* :: ('*a* × '*a*) *set* ⇒ ('*a multiset* × '*a multiset*) *set* **where**
mult1 r = {(*N*, *M*). ∃ *a M0 K*. *M* = *add-mset a M0* ∧ *N* = *M0* + *K* ∧
(∀ *b*. *b* ∈# *K* ⟶ (*b*, *a*) ∈ *r*)}

definition *mult* :: ('*a* × '*a*) *set* ⇒ ('*a multiset* × '*a multiset*) *set* **where**
mult r = (*mult1 r*)$^+$

lemma *mult1I*:
assumes *M* = *add-mset a M0* **and** *N* = *M0* + *K* **and** ⋀*b*. *b* ∈# *K* $\Longrightarrow$ (*b*, *a*)
∈ *r*
shows (*N*, *M*) ∈ *mult1 r*
using *assms* **unfolding** *mult1-def* **by** *blast*

lemma *mult1E*:
assumes (*N*, *M*) ∈ *mult1 r*
obtains *a M0 K* **where** *M* = *add-mset a M0 N* = *M0* + *K* ⋀*b*. *b* ∈# *K* $\Longrightarrow$
(*b*, *a*) ∈ *r*
using *assms* **unfolding** *mult1-def* **by** *blast*

lemma *mono-mult1*:

assumes $r \subseteq r'$ **shows** *mult1* $r \subseteq$ *mult1* r'
unfolding *mult1-def* **using** *assms* **by** *blast*

lemma *mono-mult*:
assumes $r \subseteq r'$ **shows** *mult* $r \subseteq$ *mult* r'
unfolding *mult-def* **using** *mono-mult1*[*OF assms*] *trancl-mono* **by** *blast*

lemma *not-less-empty* [*iff*]: $(M, \{\#\}) \notin$ *mult1* r
by (*simp add*: *mult1-def*)

lemma *less-add*:
assumes *mult1*: $(N,$ *add-mset a M0*$) \in$ *mult1* r
shows
$(\exists M.\ (M,\ M0) \in$ *mult1* $r \wedge N =$ *add-mset a* $M) \vee$
$(\exists K.\ (\forall b.\ b \in\# K \longrightarrow (b,\ a) \in r) \wedge N = M0 + K)$
proof −
let *?r* $= \lambda K\ a.\ \forall b.\ b \in\# K \longrightarrow (b,\ a) \in r$
let *?R* $= \lambda N\ M.\ \exists a\ M0\ K.\ M =$ *add-mset a M0* $\wedge\ N = M0 + K \wedge$ *?r K a*
obtain $a'\ M0'\ K$ **where** *M0*: *add-mset a M0* $=$ *add-mset* $a'\ M0'$
and *N*: $N = M0' + K$
and *r*: *?r K* a'
using *mult1* **unfolding** *mult1-def* **by** *auto*
show *?thesis* (**is** *?case1* $\vee$ *?case2*)
proof −
from *M0* **consider** $M0 = M0'\ a = a'$
| K' **where** *M0* $=$ *add-mset* $a'\ K'\ M0' =$ *add-mset a* K'
by *atomize-elim* (*simp only*: *add-eq-conv-ex*)
then show *?thesis*
proof *cases*
case *1*
with *N r* **have** *?r K a* $\wedge\ N = M0 + K$ **by** *simp*
then have *?case2* **..**
then show *?thesis* **..**
next
case *2*
from *N 2*(*2*) **have** *n*: $N =$ *add-mset a* $(K' + K)$ **by** *simp*
with *r 2*(*1*) **have** *?R* $(K' + K)$ *M0* **by** *blast*
with *n* **have** *?case1* **by** (*simp add*: *mult1-def*)
then show *?thesis* **..**
qed
qed
qed

lemma *all-accessible*:
assumes *wf r*
shows $\forall M.\ M \in$ *Wellfounded.acc* (*mult1 r*)
proof
let *?R* $=$ *mult1 r*
let *?W* $=$ *Wellfounded.acc ?R*

```
{
  fix M M0 a
  assume M0: M0 ∈ ?W
    and wf-hyp: ⋀b. (b, a) ∈ r ⟹ (∀ M ∈ ?W. add-mset b M ∈ ?W)
    and acc-hyp: ∀ M. (M, M0) ∈ ?R ⟶ add-mset a M ∈ ?W
  have add-mset a M0 ∈ ?W
  proof (rule accI [of add-mset a M0])
    fix N
    assume (N, add-mset a M0) ∈ ?R
    then consider M where (M, M0) ∈ ?R N = add-mset a M
      | K where ∀ b. b ∈# K ⟶ (b, a) ∈ r N = M0 + K
      by atomize-elim (rule less-add)
    then show N ∈ ?W
    proof cases
      case 1
      from acc-hyp have (M, M0) ∈ ?R ⟶ add-mset a M ∈ ?W ..
      from this and ‹(M, M0) ∈ ?R› have add-mset a M ∈ ?W ..
      then show N ∈ ?W by (simp only: ‹N = add-mset a M›)
    next
      case 2
      from this(1) have M0 + K ∈ ?W
      proof (induct K)
        case empty
        from M0 show M0 + {#} ∈ ?W by simp
      next
        case (add x K)
        from add.prems have (x, a) ∈ r by simp
        with wf-hyp have ∀ M ∈ ?W. add-mset x M ∈ ?W by blast
        moreover from add have M0 + K ∈ ?W by simp
        ultimately have add-mset x (M0 + K) ∈ ?W ..
        then show M0 + (add-mset x K) ∈ ?W by simp
      qed
      then show N ∈ ?W by (simp only: 2(2))
    qed
  qed
} note tedious-reasoning = this

show M ∈ ?W for M
proof (induct M)
  show {#} ∈ ?W
  proof (rule accI)
    fix b assume (b, {#}) ∈ ?R
    with not-less-empty show b ∈ ?W by contradiction
  qed

  fix M a assume M ∈ ?W
  from ‹wf r› have ∀ M ∈ ?W. add-mset a M ∈ ?W
  proof induct
    fix a
```

assume *r*: $\bigwedge b.\ (b,\ a) \in r \Longrightarrow (\forall M \in ?W.\ add\text{-}mset\ b\ M \in ?W)$
show $\forall M \in ?W.\ add\text{-}mset\ a\ M \in ?W$
proof
fix *M* **assume** $M \in ?W$
then show *add-mset a M* $\in ?W$
by (*rule acc-induct*) (*rule tedious-reasoning* [*OF* - *r*])
qed
qed
from *this* **and** ‹$M \in ?W$› **show** *add-mset a M* $\in ?W$ **..**
qed
qed

theorem *wf-mult1*: *wf r* $\Longrightarrow$ *wf* (*mult1 r*)
by (*rule acc-wfI*) (*rule all-accessible*)

theorem *wf-mult*: *wf r* $\Longrightarrow$ *wf* (*mult r*)
unfolding *mult-def* **by** (*rule wf-trancl*) (*rule wf-mult1*)

1.12.2 Closure-free presentation

One direction.

lemma *mult-implies-one-step*:
assumes
trans: *trans r* **and**
MN: $(M,\ N) \in mult\ r$
shows $\exists I\ J\ K.\ N = I + J \wedge M = I + K \wedge J \neq \{\#\} \wedge (\forall k \in set\text{-}mset\ K.\ \exists j \in set\text{-}mset\ J.\ (k,\ j) \in r)$
using *MN* **unfolding** *mult-def mult1-def*
proof (*induction rule*: *converse-trancl-induct*)
case (*base y*)
then show *?case* **by** *force*
next
case (*step y z*) **note** *yz* = *this*(*1*) **and** *zN* = *this*(*2*) **and** *N-decomp* = *this*(*3*)
obtain *I J K* **where**
N: $N = I + J\ z = I + K\ J \neq \{\#\}\ \forall k{\in}\#K.\ \exists j{\in}\#J.\ (k,\ j) \in r$
using *N-decomp* **by** *blast*
obtain *a M0 K′* **where**
z: $z = add\text{-}mset\ a\ M0$ **and** *y*: $y = M0 + K'$ **and** *K*: $\forall b.\ b \in\#\ K' \longrightarrow (b,\ a) \in r$
using *yz* **by** *blast*
show *?case*
proof (*cases a* $\in\#$ *K*)
case *True*
moreover have $\exists j{\in}\#J.\ (k,\ j) \in r$ **if** $k \in\#\ K'$ **for** *k*
using *K N trans True* **by** (*meson that transE*)
ultimately show *?thesis*
by (*rule-tac x* = *I* **in** *exI*, *rule-tac x* = *J* **in** *exI*, *rule-tac x* = $(K - \{\#a\#\}) + K'$ **in** *exI*)
(*use z y N* **in** ‹*auto simp del*: *subset-mset.add-diff-assoc2 dest*: *in-diffD*›)

```
  next
    case False
    then have a ∈# I by (metis N(2) union-iff union-single-eq-member z)
    moreover have M0 = I + K − {#a#}
      using N(2) z by force
    ultimately show ?thesis
      by (rule-tac x = I − {#a#} in exI, rule-tac x = add-mset a J in exI,
          rule-tac x = K + K' in exI)
        (use z y N False K in ⟨auto simp: add.assoc⟩)
  qed
qed

lemma one-step-implies-mult:
  assumes
    J ≠ {#} and
    ∀ k ∈ set-mset K. ∃ j ∈ set-mset J. (k, j) ∈ r
  shows (I + K, I + J) ∈ mult r
  using assms
proof (induction size J arbitrary: I J K)
  case 0
  then show ?case by auto
next
  case (Suc n) note IH = this(1) and size-J = this(2)[THEN sym]
  obtain J' a where J: J = add-mset a J'
    using size-J by (blast dest: size-eq-Suc-imp-eq-union)
  show ?case
  proof (cases J' = {#})
    case True
    then show ?thesis
      using J Suc by (fastforce simp add: mult-def mult1-def)
  next
    case [simp]: False
    have K: K = {#x ∈# K. (x, a) ∈ r#} + {#x ∈# K. (x, a) ∉ r#}
      by (rule multiset-partition)
    have (I + K, (I + {# x ∈# K. (x, a) ∈ r #}) + J') ∈ mult r
      using IH[of J' {# x ∈# K. (x, a) ∉ r#} I + {# x ∈# K. (x, a) ∈ r#}]
        J Suc.prems K size-J by (auto simp: ac-simps)
    moreover have (I + {#x ∈# K. (x, a) ∈ r#} + J', I + J) ∈ mult r
      by (fastforce simp: J mult1-def mult-def)
    ultimately show ?thesis
      unfolding mult-def by simp
  qed
qed
```

1.13 The multiset extension is cancellative for multiset union

```
lemma mult-cancel:
  assumes trans s and irrefl s
  shows (X + Z, Y + Z) ∈ mult s ⟷ (X, Y) ∈ mult s (is ?L ⟷ ?R)
```

proof
assume *?L* **thus** *?R*
proof (*induct Z*)
case (*add z Z*)
obtain *X' Y' Z'* **where** ∗: *add-mset z X + Z = Z' + X' add-mset z Y + Z = Z' + Y' Y' ≠ {#}*
∀ x ∈ set-mset X'. ∃ y ∈ set-mset Y'. (x, y) ∈ s
using *mult-implies-one-step*[*OF* ‹*trans s*› *add*(*2*)] **by** *auto*
consider *Z2* **where** *Z' = add-mset z Z2 | X2 Y2* **where** *X' = add-mset z X2 Y' = add-mset z Y2*
using ∗(*1*,*2*) **by** (*metis add-mset-remove-trivial-If insert-iff set-mset-add-mset-insert union-iff*)
thus *?case*
proof (*cases*)
case *1* **thus** *?thesis* **using** ∗ *one-step-implies-mult*[*of Y' X' s Z2*]
by (*auto simp*: *add.commute*[*of* - {#-#}] *add.assoc intro*: *add*(*1*))
next
case *2* **then obtain** *y* **where** *y ∈ set-mset Y2* (*z*, *y*) ∈ *s* **using** ∗(*4*) ‹*irrefl s*›
by (*auto simp*: *irrefl-def*)
moreover from *this transD*[*OF* ‹*trans s*› - *this*(*2*)]
have *x' ∈ set-mset X2 ⟹ ∃ y ∈ set-mset Y2*. (*x'*, *y*) ∈ *s* **for** *x'*
using *2* ∗(*4*)[*rule-format*, *of x'*] **by** *auto*
ultimately show *?thesis* **using** ∗ *one-step-implies-mult*[*of Y2 X2 s Z'*] *2*
by (*force simp*: *add.commute*[*of* {#-#}] *add.assoc*[*symmetric*] *intro*: *add*(*1*))
qed
qed *auto*
next
assume *?R* **then obtain** *I J K*
where *Y = I + J X = I + K J ≠ {#} ∀ k ∈ set-mset K. ∃ j ∈ set-mset J.* (*k*, *j*) ∈ *s*
using *mult-implies-one-step*[*OF* ‹*trans s*›] **by** *blast*
thus *?L* **using** *one-step-implies-mult*[*of J K s I + Z*] **by** (*auto simp*: *ac-simps*)
qed

lemma *mult-cancel-max*:
assumes *trans s* **and** *irrefl s*
shows (*X*, *Y*) ∈ *mult s* ⟷ (*X − X ∩# Y*, *Y − X ∩# Y*) ∈ *mult s* (**is** *?L* ⟷ *?R*)
proof −
have *X − X ∩# Y + X ∩# Y = X Y − X ∩# Y + X ∩# Y = Y* **by** (*auto simp*: *count-inject*[*symmetric*])
thus *?thesis* **using** *mult-cancel*[*OF assms*, *of X − X ∩# Y X ∩# Y Y − X ∩# Y*] **by** *auto*
qed

1.14 Quasi-executable version of the multiset extension

Predicate variants of *mult* and the reflexive closure of *mult*, which are executable whenever the given predicate P is. Together with the standard code equations for *op* $\cap\#$ and *op* $-$ this should yield quadratic (with respect to calls to P) implementations of *multp* and *multeqp*.

definition *multp* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ multiset \Rightarrow 'a\ multiset \Rightarrow bool$ **where**
multp P N M =
($let\ Z = M \cap\# N;\ X = M - Z\ in$
$X \neq \{\#\} \wedge (let\ Y = N - Z\ in\ (\forall y \in set\text{-}mset\ Y.\ \exists x \in set\text{-}mset\ X.\ P\ y\ x)))$

definition *multeqp* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ multiset \Rightarrow 'a\ multiset \Rightarrow bool$ **where**
multeqp P N M =
($let\ Z = M \cap\# N;\ X = M - Z;\ Y = N - Z\ in$
$(\forall y \in set\text{-}mset\ Y.\ \exists x \in set\text{-}mset\ X.\ P\ y\ x))$

lemma *multp-iff*:
assumes *irrefl R* **and** *trans R* **and** [*simp*]: $\bigwedge x\ y.\ P\ x\ y \longleftrightarrow (x, y) \in R$
shows *multp P N M* $\longleftrightarrow (N, M) \in mult\ R$ (**is** *?L* $\longleftrightarrow$ *?R*)
proof −
have $*$: $M \cap\# N + (N - M \cap\# N) = N\ M \cap\# N + (M - M \cap\# N) = M$
$(M - M \cap\# N) \cap\# (N - M \cap\# N) = \{\#\}$ **by** (*auto simp*: *count-inject*[*symmetric*])
show *?thesis*
proof
assume *?L* **thus** *?R*
using *one-step-implies-mult*[*of* $M - M \cap\# N\ N - M \cap\# N\ R\ M \cap\# N$] $*$
by (*auto simp*: *multp-def Let-def*)
next
{ fix *I J K* :: *'a multiset* **assume** $(I + J) \cap\# (I + K) = \{\#\}$
then have $I = \{\#\}$ **by** (*metis inter-union-distrib-right union-eq-empty*)
} note [*dest*!] = *this*
assume *?R* **thus** *?L*
using *mult-implies-one-step*[*OF assms*(*2*), *of* $N - M \cap\# N\ M - M \cap\# N$]
mult-cancel-max[*OF assms*(*2*,*1*), *of N M*] $*$ **by** (*auto simp*: *multp-def*)
qed
qed

lemma *multeqp-iff*:
assumes *irrefl R* **and** *trans R* **and** $\bigwedge x\ y.\ P\ x\ y \longleftrightarrow (x, y) \in R$
shows *multeqp P N M* $\longleftrightarrow (N, M) \in (mult\ R)^=$
proof −
{ assume $N \neq M\ M - M \cap\# N = \{\#\}$
then obtain *y* **where** *count N y* $\neq$ *count M y* **by** (*auto simp*: *count-inject*[*symmetric*])
then have $\exists y.\ count\ M\ y < count\ N\ y$ **using** ‹$M - M \cap\# N = \{\#\}$›
by (*auto simp*: *count-inject*[*symmetric*] *dest*!: *le-neq-implies-less fun-cong*[*of* -
- *y*])
}
then have *multeqp P N M* $\longleftrightarrow$ *multp P N M* $\vee$ $N = M$
by (*auto simp*: *multeqp-def multp-def Let-def in-diff-count*)

thus *?thesis* **using** *multp-iff*[*OF assms*] **by** *simp*
qed

1.14.1 Partial-order properties

lemma (**in** *preorder*) *mult1-lessE*:
assumes $(N, M) \in mult1\ \{(a, b).\ a < b\}$
obtains *a M0 K* **where** $M = add\text{-}mset\ a\ M0\ N = M0 + K$
$a \notin\# K \bigwedge b.\ b \in\# K \Longrightarrow b < a$
proof −
from *assms* **obtain** *a M0 K* **where** $M = add\text{-}mset\ a\ M0\ N = M0 + K$ **and**
$*$: $b \in\# K \Longrightarrow b < a$ **for** *b* **by** (*blast elim*: *mult1E*)
moreover from $*$ [*of a*] **have** $a \notin\# K$ **by** *auto*
ultimately show *thesis* **by** (*auto intro*: *that*)
qed

instantiation *multiset* :: (*preorder*) *order*
begin

definition *less-multiset* :: *'a multiset* ⇒ *'a multiset* ⇒ *bool*
where $M' < M \longleftrightarrow (M', M) \in mult\ \{(x', x).\ x' < x\}$

definition *less-eq-multiset* :: *'a multiset* ⇒ *'a multiset* ⇒ *bool*
where *less-eq-multiset M' M* ⟷ $M' < M \vee M' = M$

instance
proof −
have *irrefl*: ¬ $M < M$ **for** M :: *'a multiset*
proof
assume $M < M$
then have *MM*: $(M, M) \in mult\ \{(x, y).\ x < y\}$ **by** (*simp add*: *less-multiset-def*)
have *trans* $\{(x'::'a, x).\ x' < x\}$
by (*metis* (*mono-tags*, *lifting*) *case-prodD case-prodI less-trans mem-Collect-eq transI*)
moreover note *MM*
ultimately have $\exists I\ J\ K.\ M = I + J \wedge M = I + K$
$\wedge\ J \neq \{\#\} \wedge (\forall k \in set\text{-}mset\ K.\ \exists j \in set\text{-}mset\ J.\ (k, j) \in \{(x, y).\ x < y\})$
by (*rule mult-implies-one-step*)
then obtain *I J K* **where** $M = I + J$ **and** $M = I + K$
and $J \neq \{\#\}$ **and** $(\forall k \in set\text{-}mset\ K.\ \exists j \in set\text{-}mset\ J.\ (k, j) \in \{(x, y).\ x < y\})$ **by** *blast*
then have $*$: $K \neq \{\#\}$ **and** $**$: $\forall k \in set\text{-}mset\ K.\ \exists j \in set\text{-}mset\ K.\ k < j$ **by** *auto*
have *finite* (*set-mset K*) **by** *simp*
moreover note $**$
ultimately have *set-mset K* = {}
by (*induct rule*: *finite-induct*) (*auto intro*: *order-less-trans*)
with $*$ **show** *False* **by** *simp*
qed

have *trans*: $K < M \Longrightarrow M < N \Longrightarrow K < N$ **for** $K\ M\ N$ :: *'a multiset*
unfolding *less-multiset-def mult-def* **by** (*blast intro*: *trancl-trans*)
show *OFCLASS*(*'a multiset*, *order-class*)
by *standard* (*auto simp add*: *less-eq-multiset-def irrefl dest*: *trans*)
qed
end — FIXME avoid junk stemming from type class interpretation

lemma *mset-le-irrefl* [*elim*!]:
fixes M :: *'a*::*preorder multiset*
shows $M < M \Longrightarrow R$
by *simp*

1.14.2 Monotonicity of multiset union

lemma *mult1-union*: $(B,\ D) \in$ *mult1 r* $\Longrightarrow (C + B,\ C + D) \in$ *mult1 r*
by (*force simp*: *mult1-def*)

lemma *union-le-mono2*: $B < D \Longrightarrow C + B < C +$ (D::*'a*::*preorder multiset*)
apply (*unfold less-multiset-def mult-def*)
apply (*erule trancl-induct*)
apply (*blast intro*: *mult1-union*)
apply (*blast intro*: *mult1-union trancl-trans*)
done

lemma *union-le-mono1*: $B < D \Longrightarrow B + C < D +$ (C::*'a*::*preorder multiset*)
apply (*subst add.commute* [*of B C*])
apply (*subst add.commute* [*of D C*])
apply (*erule union-le-mono2*)
done

lemma *union-less-mono*:
fixes $A\ B\ C\ D$ :: *'a*::*preorder multiset*
shows $A < C \Longrightarrow B < D \Longrightarrow A + B < C + D$
by (*blast intro*!: *union-le-mono1 union-le-mono2 less-trans*)

instantiation *multiset* :: (*preorder*) *ordered-ab-semigroup-add*
begin
instance
by *standard* (*auto simp add*: *less-eq-multiset-def intro*: *union-le-mono2*)
end

1.14.3 Termination proofs with multiset orders

lemma *multi-member-skip*: $x \in\# XS \Longrightarrow x \in\# \{\#\ y\ \#\} + XS$
and *multi-member-this*: $x \in\# \{\#\ x\ \#\} + XS$
and *multi-member-last*: $x \in\# \{\#\ x\ \#\}$
by *auto*

definition *ms-strict* = *mult pair-less*
definition *ms-weak* = *ms-strict* $\cup$ *Id*

lemma *ms-reduction-pair*: *reduction-pair* (*ms-strict*, *ms-weak*)
unfolding *reduction-pair-def ms-strict-def ms-weak-def pair-less-def*
by (*auto intro*: *wf-mult1 wf-trancl simp*: *mult-def*)

lemma *smsI*:
(*set-mset A*, *set-mset B*) ∈ *max-strict* ⟹ (*Z* + *A*, *Z* + *B*) ∈ *ms-strict*
unfolding *ms-strict-def*
by (*rule one-step-implies-mult*) (*auto simp add*: *max-strict-def pair-less-def elim*!:*max-ext.cases*)

lemma *wmsI*:
(*set-mset A*, *set-mset B*) ∈ *max-strict* ∨ *A* = {#} ∧ *B* = {#}
⟹ (*Z* + *A*, *Z* + *B*) ∈ *ms-weak*
unfolding *ms-weak-def ms-strict-def*
by (*auto simp add*: *pair-less-def max-strict-def elim*!:*max-ext.cases intro*: *one-step-implies-mult*)

inductive *pw-leq*
where
pw-leq-empty: *pw-leq* {#} {#}
| *pw-leq-step*: ⟦(*x*,*y*) ∈ *pair-leq*; *pw-leq X Y* ⟧ ⟹ *pw-leq* ({#*x*#} + *X*) ({#*y*#}
+ *Y*)

lemma *pw-leq-lstep*:
(*x*, *y*) ∈ *pair-leq* ⟹ *pw-leq* {#*x*#} {#*y*#}
by (*drule pw-leq-step*) (*rule pw-leq-empty*, *simp*)

lemma *pw-leq-split*:
assumes *pw-leq X Y*
shows ∃ *A B Z*. *X* = *A* + *Z* ∧ *Y* = *B* + *Z* ∧ ((*set-mset A*, *set-mset B*) ∈
max-strict ∨ (*B* = {#} ∧ *A* = {#}))
using *assms*
proof *induct*
case *pw-leq-empty* **thus** *?case* **by** *auto*
next
case (*pw-leq-step x y X Y*)
then obtain *A B Z* **where**
[*simp*]: *X* = *A* + *Z Y* = *B* + *Z*
and *1*[*simp*]: (*set-mset A*, *set-mset B*) ∈ *max-strict* ∨ (*B* = {#} ∧ *A* = {#})
by *auto*
from *pw-leq-step* **consider** *x* = *y* | (*x*, *y*) ∈ *pair-less*
unfolding *pair-leq-def* **by** *auto*
thus *?case*
proof *cases*
case [*simp*]: *1*
have {#*x*#} + *X* = *A* + ({#*y*#}+*Z*) ∧ {#*y*#} + *Y* = *B* + ({#*y*#}+*Z*) ∧
((*set-mset A*, *set-mset B*) ∈ *max-strict* ∨ (*B* = {#} ∧ *A* = {#}))
by *auto*
thus *?thesis* **by** *blast*
next

```
    case 2
    let ?A' = {#x#} + A and ?B' = {#y#} + B
    have {#x#} + X = ?A' + Z
      {#y#} + Y = ?B' + Z
      by auto
    moreover have
      (set-mset ?A', set-mset ?B') ∈ max-strict
      using 1 2 unfolding max-strict-def
      by (auto elim!: max-ext.cases)
    ultimately show ?thesis by blast
  qed
qed

lemma
  assumes pwleq: pw-leq Z Z'
  shows ms-strictI: (set-mset A, set-mset B) ∈ max-strict ⟹ (Z + A, Z' + B)
∈ ms-strict
    and ms-weakI1: (set-mset A, set-mset B) ∈ max-strict ⟹ (Z + A, Z' + B)
∈ ms-weak
    and ms-weakI2: (Z + {#}, Z' + {#}) ∈ ms-weak
proof −
  from pw-leq-split[OF pwleq]
  obtain A' B' Z''
    where [simp]: Z = A' + Z'' Z' = B' + Z''
    and mx-or-empty: (set-mset A', set-mset B') ∈ max-strict ∨ (A' = {#} ∧ B'
= {#})
    by blast
  {
    assume max: (set-mset A, set-mset B) ∈ max-strict
    from mx-or-empty
    have (Z'' + (A + A'), Z'' + (B + B')) ∈ ms-strict
    proof
      assume max': (set-mset A', set-mset B') ∈ max-strict
      with max have (set-mset (A + A'), set-mset (B + B')) ∈ max-strict
        by (auto simp: max-strict-def intro: max-ext-additive)
      thus ?thesis by (rule smsI)
    next
      assume [simp]: A' = {#} ∧ B' = {#}
      show ?thesis by (rule smsI) (auto intro: max)
    qed
    thus (Z + A, Z' + B) ∈ ms-strict by (simp add: ac-simps)
    thus (Z + A, Z' + B) ∈ ms-weak by (simp add: ms-weak-def)
  }
  from mx-or-empty
  have (Z'' + A', Z'' + B') ∈ ms-weak by (rule wmsI)
  thus (Z + {#}, Z' + {#}) ∈ ms-weak by (simp add: ac-simps)
qed

lemma empty-neutral: {#} + x = x x + {#} = x
```

```
and nonempty-plus: {# x #} + rs ≠ {#}
and nonempty-single: {# x #} ≠ {#}
by auto

setup ‹
  let
    fun msetT T = Type (@{type-name multiset}, [T]);

    fun mk-mset T [] = Const (@{const-abbrev Mempty}, msetT T)
      | mk-mset T [x] =
        Const (@{const-name add-mset}, T --> msetT T --> msetT T) $ x $
          Const (@{const-abbrev Mempty}, msetT T)
      | mk-mset T (x :: xs) =
        Const (@{const-name plus}, msetT T --> msetT T --> msetT T) $
          mk-mset T [x] $ mk-mset T xs

    fun mset-member-tac ctxt m i =
      if m <= 0 then
        resolve-tac ctxt @{thms multi-member-this} i ORELSE
        resolve-tac ctxt @{thms multi-member-last} i
      else
        resolve-tac ctxt @{thms multi-member-skip} i THEN mset-member-tac ctxt
(m - 1) i

    fun mset-nonempty-tac ctxt =
      resolve-tac ctxt @{thms nonempty-plus} ORELSE'
      resolve-tac ctxt @{thms nonempty-single}

    fun regroup-munion-conv ctxt =
      Function-Lib.regroup-conv ctxt @{const-abbrev Mempty} @{const-name plus}
      (map (fn t => t RS eq-reflection) (@{thms ac-simps} @ @{thms empty-neutral}))

    fun unfold-pwleq-tac ctxt i =
     (resolve-tac ctxt @{thms pw-leq-step} i THEN (fn st => unfold-pwleq-tac ctxt
(i + 1) st))
        ORELSE (resolve-tac ctxt @{thms pw-leq-lstep} i)
        ORELSE (resolve-tac ctxt @{thms pw-leq-empty} i)

    val set-mset-simps = [@{thm set-mset-empty}, @{thm set-mset-single}, @{thm
set-mset-union},
                          @{thm Un-insert-left}, @{thm Un-empty-left}]
  in
    ScnpReconstruct.multiset-setup (ScnpReconstruct.Multiset
    {
     msetT=msetT, mk-mset=mk-mset, mset-regroup-conv=regroup-munion-conv,
     mset-member-tac=mset-member-tac, mset-nonempty-tac=mset-nonempty-tac,
      mset-pwleq-tac=unfold-pwleq-tac, set-of-simps=set-mset-simps,
       smsI'= @{thm ms-strictI}, wmsI2''= @{thm ms-weakI2}, wmsI1= @{thm
ms-weakI1},
```

reduction-pair = @{*thm ms-reduction-pair*}
})
end
›

1.15 Legacy theorem bindings

lemmas *multi-count-eq* = *multiset-eq-iff* [*symmetric*]

lemma *union-commute*: $M + N = N + (M::'a\ multiset)$
by (*fact add.commute*)

lemma *union-assoc*: $(M + N) + K = M + (N + (K::'a\ multiset))$
by (*fact add.assoc*)

lemma *union-lcomm*: $M + (N + K) = N + (M + (K::'a\ multiset))$
by (*fact add.left-commute*)

lemmas *union-ac* = *union-assoc union-commute union-lcomm add-mset-commute*

lemma *union-right-cancel*: $M + K = N + K \longleftrightarrow M = (N::'a\ multiset)$
by (*fact add-right-cancel*)

lemma *union-left-cancel*: $K + M = K + N \longleftrightarrow M = (N::'a\ multiset)$
by (*fact add-left-cancel*)

lemma *multi-union-self-other-eq*: $(A::'a\ multiset) + X = A + Y \Longrightarrow X = Y$
by (*fact add-left-imp-eq*)

lemma *mset-subset-trans*: $(M::'a\ multiset) \subset\# K \Longrightarrow K \subset\# N \Longrightarrow M \subset\# N$
by (*fact subset-mset.less-trans*)

lemma *multiset-inter-commute*: $A \cap\# B = B \cap\# A$
by (*fact subset-mset.inf.commute*)

lemma *multiset-inter-assoc*: $A \cap\# (B \cap\# C) = A \cap\# B \cap\# C$
by (*fact subset-mset.inf.assoc* [*symmetric*])

lemma *multiset-inter-left-commute*: $A \cap\# (B \cap\# C) = B \cap\# (A \cap\# C)$
by (*fact subset-mset.inf.left-commute*)

lemmas *multiset-inter-ac* =
multiset-inter-commute
multiset-inter-assoc
multiset-inter-left-commute

lemma *mset-le-not-refl*: $\neg\ M < (M::'a::preorder\ multiset)$
by (*fact less-irrefl*)

lemma *mset-le-trans*: $K < M \Longrightarrow M < N \Longrightarrow K < (N::'a::preorder\ multiset)$
by (*fact less-trans*)

lemma *mset-le-not-sym*: $M < N \Longrightarrow \neg\ N < (M::'a::preorder\ multiset)$
by (*fact less-not-sym*)

lemma *mset-le-asym*: $M < N \Longrightarrow (\neg\ P \Longrightarrow N < (M::'a::preorder\ multiset)) \Longrightarrow P$
by (*fact less-asym*)

declaration ‹
```
let
  fun multiset-postproc - maybe-name all-values (T as Type (-, [elem-T])) (Const
- $ t') =
      let
        val (maybe-opt, ps) =
          Nitpick-Model.dest-plain-fun t'
          ||> op ~~
          ||> map (apsnd (snd o HOLogic.dest-number))
        fun elems-for t =
          (case AList.lookup (op =) ps t of
            SOME n => replicate n t
          | NONE => [Const (maybe-name, elem-T --> elem-T) $ t])
      in
        (case maps elems-for (all-values elem-T) @
              (if maybe-opt then [Const (Nitpick-Model.unrep-mixfix (), elem-T)]
else []) of
          [] => Const (@{const-name zero-class.zero}, T)
        | ts =>
            foldl1 (fn (s, t) => Const (@{const-name add-mset}, elem-T -->
T --> T) $ s $ t)
              ts)
      end
  | multiset-postproc - - - - t = t
in Nitpick-Model.register-term-postprocessor @{typ 'a multiset} multiset-postproc
end
```
›

1.16 Naive implementation using lists

code-datatype *mset*

lemma [*code*]: $\{\#\} = mset\ []$
by *simp*

lemma [*code*]: $add\text{-}mset\ x\ (mset\ xs) = mset\ (x\ \#\ xs)$
by *simp*

lemma [*code*]: $Multiset.is\text{-}empty\ (mset\ xs) \longleftrightarrow List.null\ xs$

by (*simp add*: *Multiset.is-empty-def List.null-def*)

lemma *union-code* [*code*]: *mset xs* + *mset ys* = *mset* (*xs* @ *ys*)
by *simp*

lemma [*code*]: *image-mset f* (*mset xs*) = *mset* (*map f xs*)
by *simp*

lemma [*code*]: *filter-mset f* (*mset xs*) = *mset* (*filter f xs*)
by (*simp add*: *mset-filter*)

lemma [*code*]: *mset xs* − *mset ys* = *mset* (*fold remove1 ys xs*)
by (*rule sym*, *induct ys arbitrary*: *xs*) (*simp-all add*: *diff-add diff-right-commute diff-diff-add*)

lemma [*code*]:
mset xs ∩# *mset ys* =
mset (*snd* (*fold* (λ*x* (*ys*, *zs*).
if x ∈ *set ys then* (*remove1 x ys*, *x* # *zs*) *else* (*ys*, *zs*)) *xs* (*ys*, [])))
proof −
have ⋀*zs*. *mset* (*snd* (*fold* (λ*x* (*ys*, *zs*).
if x ∈ *set ys then* (*remove1 x ys*, *x* # *zs*) *else* (*ys*, *zs*)) *xs* (*ys*, *zs*))) =
(*mset xs* ∩# *mset ys*) + *mset zs*
by (*induct xs arbitrary*: *ys*)
(*auto simp add*: *inter-add-right1 inter-add-right2 ac-simps*)
then show *?thesis* **by** *simp*
qed

lemma [*code*]:
mset xs ∪# *mset ys* =
mset (*case-prod append* (*fold* (λ*x* (*ys*, *zs*). (*remove1 x ys*, *x* # *zs*)) *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*
qed

declare *in-multiset-in-set* [*code-unfold*]

lemma [*code*]: *count* (*mset xs*) *x* = *fold* (λ*y*. *if x* = *y then Suc else id*) *xs 0*
proof −
have ⋀*n*. *fold* (λ*y*. *if x* = *y then Suc else id*) *xs n* = *count* (*mset xs*) *x* + *n*
by (*induct xs*) *simp-all*
then show *?thesis* **by** *simp*
qed

declare *set-mset-mset* [*code*]

declare *sorted-list-of-multiset-mset* [*code*]

lemma [*code*]: — not very efficient, but representation-ignorant!
mset-set A = mset (*sorted-list-of-set A*)
apply (*cases finite A*)
apply *simp-all*
apply (*induct A rule*: *finite-induct*)
apply *simp-all*
done

declare *size-mset* [*code*]

fun *subset-eq-mset-impl* :: *'a list* ⇒ *'a list* ⇒ *bool option* **where**
subset-eq-mset-impl [] *ys* = *Some* (*ys* ≠ [])
| *subset-eq-mset-impl* (*Cons x xs*) *ys* = (*case List.extract* (*op* = *x*) *ys of*
None ⇒ *None*
| *Some* (*ys1*,-,*ys2*) ⇒ *subset-eq-mset-impl xs* (*ys1* @ *ys2*))

lemma *subset-eq-mset-impl*: (*subset-eq-mset-impl xs ys* = *None* ⟷ ¬ *mset xs* ⊆# *mset ys*) ∧
(*subset-eq-mset-impl xs ys* = *Some True* ⟷ *mset xs* ⊂# *mset ys*) ∧
(*subset-eq-mset-impl xs ys* = *Some False* ⟶ *mset xs* = *mset ys*)
proof (*induct xs arbitrary*: *ys*)
case (*Nil ys*)
show *?case* **by** (*auto simp*: *subset-mset.zero-less-iff-neq-zero*)
next
case (*Cons x xs ys*)
show *?case*
proof (*cases List.extract* (*op* = *x*) *ys*)
case *None*
hence *x*: *x* ∉ *set ys* **by** (*simp add*: *extract-None-iff*)
{
assume *mset* (*x* # *xs*) ⊆# *mset ys*
from *set-mset-mono*[*OF this*] *x* **have** *False* **by** *simp*
} note *nle* = *this*
moreover
{
assume *mset* (*x* # *xs*) ⊂# *mset ys*
hence *mset* (*x* # *xs*) ⊆# *mset ys* **by** *auto*
from *nle*[*OF this*] **have** *False* **.**
}
ultimately show *?thesis* **using** *None* **by** *auto*
next
case (*Some res*)
obtain *ys1 y ys2* **where** *res*: *res* = (*ys1*,*y*,*ys2*) **by** (*cases res*, *auto*)
note *Some* = *Some*[*unfolded res*]
from *extract-SomeE*[*OF Some*] **have** *ys* = *ys1* @ *x* # *ys2* **by** *simp*
hence *id*: *mset ys* = *add-mset x* (*mset* (*ys1* @ *ys2*))

```
    by auto
  show ?thesis unfolding subset-eq-mset-impl.simps
    unfolding Some option.simps split
    unfolding id
    using Cons[of ys1 @ ys2]
    unfolding subset-mset-def subseteq-mset-def by auto
 qed
qed

lemma [code]: mset xs ⊆# mset ys ⟷ subset-eq-mset-impl xs ys ≠ None
 using subset-eq-mset-impl[of xs ys] by (cases subset-eq-mset-impl xs ys, auto)

lemma [code]: mset xs ⊂# mset ys ⟷ subset-eq-mset-impl xs ys = Some True
 using subset-eq-mset-impl[of xs ys] by (cases subset-eq-mset-impl xs ys, auto)

instantiation multiset :: (equal) equal
begin

definition
 [code del]: HOL.equal A (B :: 'a multiset) ⟷ A = B
lemma [code]: HOL.equal (mset xs) (mset ys) ⟷ subset-eq-mset-impl xs ys =
Some False
 unfolding equal-multiset-def
 using subset-eq-mset-impl[of xs ys] by (cases subset-eq-mset-impl xs ys, auto)

instance
 by standard (simp add: equal-multiset-def)

end

lemma [code]: sum-mset (mset xs) = sum-list xs
 by (induct xs) simp-all

lemma [code]: prod-mset (mset xs) = fold times xs 1
proof −
 have ⋀x. fold times xs x = prod-mset (mset xs) * x
   by (induct xs) (simp-all add: ac-simps)
 then show ?thesis by simp
qed
```

Exercise for the casual reader: add implementations for $op \leq$ and $op <$ (multiset order).

Quickcheck generators

```
definition (in term-syntax)
 msetify :: 'a::typerep list × (unit ⇒ Code-Evaluation.term)
   ⇒ 'a multiset × (unit ⇒ Code-Evaluation.term) where
 [code-unfold]: msetify xs = Code-Evaluation.valtermify mset {·} xs
```

notation *fcomp* (**infixl** $\circ>$ *60*)
notation *scomp* (**infixl** $\circ\rightarrow$ *60*)

instantiation *multiset* :: (*random*) *random*
begin

definition
Quickcheck-Random.random i = *Quickcheck-Random.random i* $\circ\rightarrow$ (λ*xs*. *Pair* (*msetify xs*))

instance ..

end

no-notation *fcomp* (**infixl** $\circ>$ *60*)
no-notation *scomp* (**infixl** $\circ\rightarrow$ *60*)

instantiation *multiset* :: (*full-exhaustive*) *full-exhaustive*
begin

definition *full-exhaustive-multiset* :: ('*a multiset* $\times$ (*unit* $\Rightarrow$ *term*) $\Rightarrow$ (*bool* $\times$ *term list*) *option*) $\Rightarrow$ *natural* $\Rightarrow$ (*bool* $\times$ *term list*) *option*
where
full-exhaustive-multiset f i = *Quickcheck-Exhaustive.full-exhaustive* (λ*xs*. *f* (*msetify xs*)) *i*

instance ..

end

hide-const (**open**) *msetify*

1.17 BNF setup

definition *rel-mset* **where**
rel-mset R X Y $\longleftrightarrow$ ($\exists$ *xs ys*. *mset xs* = *X* $\wedge$ *mset ys* = *Y* $\wedge$ *list-all2 R xs ys*)

lemma *mset-zip-take-Cons-drop-twice*:
assumes *length xs* = *length ys j* $\leq$ *length xs*
shows *mset* (*zip* (*take j xs* @ *x* # *drop j xs*) (*take j ys* @ *y* # *drop j ys*)) =
add-mset (*x*,*y*) (*mset* (*zip xs ys*))
using *assms*
proof (*induct xs ys arbitrary*: *x y j rule*: *list-induct2*)
case *Nil*
thus *?case*
by *simp*
next
case (*Cons x xs y ys*)
thus *?case*

```
  proof (cases j = 0)
    case True
    thus ?thesis
      by simp
  next
    case False
    then obtain k where k: j = Suc k
      by (cases j) simp
    hence k ≤ length xs
      using Cons.prems by auto
    hence mset (zip (take k xs @ x # drop k xs) (take k ys @ y # drop k ys)) =
      add-mset (x,y) (mset (zip xs ys))
      by (rule Cons.hyps(2))
    thus ?thesis
      unfolding k by auto
  qed
qed

lemma ex-mset-zip-left:
  assumes length xs = length ys mset xs' = mset xs
  shows ∃ys'. length ys' = length xs' ∧ mset (zip xs' ys') = mset (zip xs ys)
using assms
proof (induct xs ys arbitrary: xs' rule: list-induct2)
  case Nil
  thus ?case
    by auto
next
  case (Cons x xs y ys xs')
  obtain j where j-len: j < length xs' and nth-j: xs' ! j = x
    by (metis Cons.prems in-set-conv-nth list.set-intros(1) mset-eq-setD)

  define xsa where xsa = take j xs' @ drop (Suc j) xs'
  have mset xs' = {#x#} + mset xsa
    unfolding xsa-def using j-len nth-j
   by (metis Cons-nth-drop-Suc union-mset-add-mset-right add-mset-remove-trivial
add-diff-cancel-left'
       append-take-drop-id mset.simps(2) mset-append)
  hence ms-x: mset xsa = mset xs
    by (simp add: Cons.prems)
  then obtain ysa where
    len-a: length ysa = length xsa and ms-a: mset (zip xsa ysa) = mset (zip xs ys)
    using Cons.hyps(2) by blast

  define ys' where ys' = take j ysa @ y # drop j ysa
  have xs': xs' = take j xsa @ x # drop j xsa
    using ms-x j-len nth-j Cons.prems xsa-def
   by (metis append-eq-append-conv append-take-drop-id diff-Suc-Suc Cons-nth-drop-Suc
length-Cons
      length-drop size-mset)
```

have *j-len'*: $j \leq$ *length xsa*
using *j-len xs' xsa-def*
by (*metis add-Suc-right append-take-drop-id length-Cons length-append less-eq-Suc-le not-less*)
have *length ys'* = *length xs'*
unfolding *ys'-def* **using** *Cons.prems len-a ms-x*
by (*metis add-Suc-right append-take-drop-id length-Cons length-append mset-eq-length*)
moreover have *mset* (*zip xs' ys'*) = *mset* (*zip* (*x* # *xs*) (*y* # *ys*))
unfolding *xs' ys'-def*
by (*rule trans*[*OF mset-zip-take-Cons-drop-twice*])
(*auto simp*: *len-a ms-a j-len'*)
ultimately show *?case*
by *blast*
qed

lemma *list-all2-reorder-left-invariance*:
assumes *rel*: *list-all2 R xs ys* **and** *ms-x*: *mset xs'* = *mset xs*
shows ∃ *ys'*. *list-all2 R xs' ys'* ∧ *mset ys'* = *mset ys*
proof −
have *len*: *length xs* = *length ys*
using *rel list-all2-conv-all-nth* **by** *auto*
obtain *ys'* **where**
len': *length xs'* = *length ys'* **and** *ms-xy*: *mset* (*zip xs' ys'*) = *mset* (*zip xs ys*)
using *len ms-x* **by** (*metis ex-mset-zip-left*)
have *list-all2 R xs' ys'*
using *assms*(*1*) *len' ms-xy* **unfolding** *list-all2-iff* **by** (*blast dest*: *mset-eq-setD*)
moreover have *mset ys'* = *mset ys*
using *len len' ms-xy map-snd-zip mset-map* **by** *metis*
ultimately show *?thesis*
by *blast*
qed

lemma *ex-mset*: ∃ *xs*. *mset xs* = *X*
by (*induct X*) (*simp*, *metis mset.simps*(*2*))

inductive *pred-mset* :: ('*a* ⇒ *bool*) ⇒ '*a multiset* ⇒ *bool*
where
pred-mset P {#}
| ⟦*P a*; *pred-mset P M*⟧ ⟹ *pred-mset P* (*add-mset a M*)

bnf '*a multiset*
map: *image-mset*
sets: *set-mset*
bd: *natLeq*
wits: {#}
rel: *rel-mset*
pred: *pred-mset*
proof −
show *image-mset id* = *id*

by (*rule image-mset.id*)
show *image-mset* (*g* ∘ *f*) = *image-mset g* ∘ *image-mset f* **for** *f g*
unfolding *comp-def* **by** (*rule ext*) (*simp add*: *comp-def image-mset.compositionality*)
show (⋀*z*. *z* ∈ *set-mset X* ⟹ *f z* = *g z*) ⟹ *image-mset f X* = *image-mset g X* **for** *f g X*
by (*induct X*) *simp-all*
show *set-mset* ∘ *image-mset f* = *op* ‘ *f* ∘ *set-mset* **for** *f*
by *auto*
show *card-order natLeq*
by (*rule natLeq-card-order*)
show *BNF-Cardinal-Arithmetic.cinfinite natLeq*
by (*rule natLeq-cinfinite*)
show *ordLeq3* (*card-of* (*set-mset X*)) *natLeq* **for** *X*
by *transfer*
(*auto intro*!: *ordLess-imp-ordLeq simp*: *finite-iff-ordLess-natLeq*[*symmetric*] *multiset-def*)
show *rel-mset R OO rel-mset S* ≤ *rel-mset* (*R OO S*) **for** *R S*
unfolding *rel-mset-def*[*abs-def*] *OO-def*
apply *clarify*
subgoal for *X Z Y xs ys*′ *ys zs*
apply (*drule list-all2-reorder-left-invariance* [**where** *xs* = *ys*′ **and** *ys* = *zs* **and** *xs*′ = *ys*])
apply (*auto intro*: *list-all2-trans*)
done
done
show *rel-mset R* =
(λ*x y*. ∃*z*. *set-mset z* ⊆ {(*x*, *y*). *R x y*} ∧
image-mset fst z = *x* ∧ *image-mset snd z* = *y*) **for** *R*
unfolding *rel-mset-def*[*abs-def*]
apply (*rule ext*)+
apply *safe*
apply (*rule-tac x* = *mset* (*zip xs ys*) **in** *exI*;
auto simp: *in-set-zip list-all2-iff mset-map*[*symmetric*])
apply (*rename-tac XY*)
apply (*cut-tac X* = *XY* **in** *ex-mset*)
apply (*erule exE*)
apply (*rename-tac xys*)
apply (*rule-tac x* = *map fst xys* **in** *exI*)
apply (*auto simp*: *mset-map*)
apply (*rule-tac x* = *map snd xys* **in** *exI*)
apply (*auto simp*: *mset-map list-all2I subset-eq zip-map-fst-snd*)
done
show *z* ∈ *set-mset* {#} ⟹ *False* **for** *z*
by *auto*
show *pred-mset P* = (λ*x*. *Ball* (*set-mset x*) *P*) **for** *P*
proof (*intro ext iffI*)
fix *x*
assume *pred-mset P x*
then show *Ball* (*set-mset x*) *P* **by** (*induct pred*: *pred-mset*; *simp*)

```
  next
    fix x
    assume Ball (set-mset x) P
    then show pred-mset P x by (induct x; auto intro: pred-mset.intros)
  qed
qed

inductive rel-mset'
where
  Zero[intro]: rel-mset' R {#} {#}
| Plus[intro]: ⟦R a b; rel-mset' R M N⟧ ⟹ rel-mset' R (add-mset a M) (add-mset
b N)

lemma rel-mset-Zero: rel-mset R {#} {#}
unfolding rel-mset-def Grp-def by auto

declare multiset.count[simp]
declare Abs-multiset-inverse[simp]
declare multiset.count-inverse[simp]
declare union-preserves-multiset[simp]

lemma rel-mset-Plus:
  assumes ab: R a b
    and MN: rel-mset R M N
  shows rel-mset R (add-mset a M) (add-mset b N)
proof −
  have ∃ya. add-mset a (image-mset fst y) = image-mset fst ya ∧
    add-mset b (image-mset snd y) = image-mset snd ya ∧
    set-mset ya ⊆ {(x, y). R x y}
    if R a b and set-mset y ⊆ {(x, y). R x y} for y
    using that by (intro exI[of - add-mset (a,b) y]) auto
  thus ?thesis
  using assms
  unfolding multiset.rel-compp-Grp Grp-def by blast
qed

lemma rel-mset'-imp-rel-mset: rel-mset' R M N ⟹ rel-mset R M N
  by (induct rule: rel-mset'.induct) (auto simp: rel-mset-Zero rel-mset-Plus)

lemma rel-mset-size: rel-mset R M N ⟹ size M = size N
  unfolding multiset.rel-compp-Grp Grp-def by auto

lemma multiset-induct2[case-names empty addL addR]:
  assumes empty: P {#} {#}
    and addL: ⋀a M N. P M N ⟹ P (add-mset a M) N
    and addR: ⋀a M N. P M N ⟹ P M (add-mset a N)
  shows P M N
apply(induct N rule: multiset-induct)
  apply(induct M rule: multiset-induct, rule empty, erule addL)
```

apply(*induct M rule*: *multiset-induct*, *erule addR*, *erule addR*)
done

lemma *multiset-induct2-size*[*consumes 1*, *case-names empty add*]:
assumes *c*: *size M* = *size N*
and *empty*: *P* {#} {#}
and *add*: ⋀*a b M N a b*. *P M N* ⟹ *P* (*add-mset a M*) (*add-mset b N*)
shows *P M N*
using *c*
proof (*induct M arbitrary*: *N rule*: *measure-induct-rule*[*of size*])
case (*less M*)
show *?case*
proof(*cases M* = {#})
case *True* **hence** *N* = {#} **using** *less.prems* **by** *auto*
thus *?thesis* **using** *True empty* **by** *auto*
next
case *False* **then obtain** *M1 a* **where** *M*: *M* = *add-mset a M1* **by** (*metis multi-nonempty-split*)
have *N* ≠ {#} **using** *False less.prems* **by** *auto*
then obtain *N1 b* **where** *N*: *N* = *add-mset b N1* **by** (*metis multi-nonempty-split*)
have *size M1* = *size N1* **using** *less.prems* **unfolding** *M N* **by** *auto*
thus *?thesis* **using** *M N less.hyps add* **by** *auto*
qed
qed

lemma *msed-map-invL*:
assumes *image-mset f* (*add-mset a M*) = *N*
shows ∃ *N1*. *N* = *add-mset* (*f a*) *N1* ∧ *image-mset f M* = *N1*
proof −
have *f a* ∈# *N*
using *assms multiset.set-map*[*of f add-mset a M*] **by** *auto*
then obtain *N1* **where** *N*: *N* = *add-mset* (*f a*) *N1* **using** *multi-member-split* **by** *metis*
have *image-mset f M* = *N1* **using** *assms* **unfolding** *N* **by** *simp*
thus *?thesis* **using** *N* **by** *blast*
qed

lemma *msed-map-invR*:
assumes *image-mset f M* = *add-mset b N*
shows ∃ *M1 a*. *M* = *add-mset a M1* ∧ *f a* = *b* ∧ *image-mset f M1* = *N*
proof −
obtain *a* **where** *a*: *a* ∈# *M* **and** *fa*: *f a* = *b*
using *multiset.set-map*[*of f M*] **unfolding** *assms*
by (*metis image-iff union-single-eq-member*)
then obtain *M1* **where** *M*: *M* = *add-mset a M1* **using** *multi-member-split* **by** *metis*
have *image-mset f M1* = *N* **using** *assms* **unfolding** *M fa*[*symmetric*] **by** *simp*
thus *?thesis* **using** *M fa* **by** *blast*
qed

lemma *msed-rel-invL*:
assumes *rel-mset R* (*add-mset a M*) *N*
shows ∃ *N1 b*. *N* = *add-mset b N1* ∧ *R a b* ∧ *rel-mset R M N1*
proof −
obtain *K* **where** *KM*: *image-mset fst K* = *add-mset a M*
and *KN*: *image-mset snd K* = *N* **and** *sK*: *set-mset K* ⊆ {(*a*, *b*). *R a b*}
using *assms*
unfolding *multiset.rel-compp-Grp Grp-def* **by** *auto*
obtain *K1 ab* **where** *K*: *K* = *add-mset ab K1* **and** *a*: *fst ab* = *a*
and *K1M*: *image-mset fst K1* = *M* **using** *msed-map-invR*[*OF KM*] **by** *auto*
obtain *N1* **where** *N*: *N* = *add-mset* (*snd ab*) *N1* **and** *K1N1*: *image-mset snd K1* = *N1*
using *msed-map-invL*[*OF KN*[*unfolded K*]] **by** *auto*
have *Rab*: *R a* (*snd ab*) **using** *sK a* **unfolding** *K* **by** *auto*
have *rel-mset R M N1* **using** *sK K1M K1N1*
unfolding *K multiset.rel-compp-Grp Grp-def* **by** *auto*
thus *?thesis* **using** *N Rab* **by** *auto*
qed

lemma *msed-rel-invR*:
assumes *rel-mset R M* (*add-mset b N*)
shows ∃ *M1 a*. *M* = *add-mset a M1* ∧ *R a b* ∧ *rel-mset R M1 N*
proof −
obtain *K* **where** *KN*: *image-mset snd K* = *add-mset b N*
and *KM*: *image-mset fst K* = *M* **and** *sK*: *set-mset K* ⊆ {(*a*, *b*). *R a b*}
using *assms*
unfolding *multiset.rel-compp-Grp Grp-def* **by** *auto*
obtain *K1 ab* **where** *K*: *K* = *add-mset ab K1* **and** *b*: *snd ab* = *b*
and *K1N*: *image-mset snd K1* = *N* **using** *msed-map-invR*[*OF KN*] **by** *auto*
obtain *M1* **where** *M*: *M* = *add-mset* (*fst ab*) *M1* **and** *K1M1*: *image-mset fst K1* = *M1*
using *msed-map-invL*[*OF KM*[*unfolded K*]] **by** *auto*
have *Rab*: *R* (*fst ab*) *b* **using** *sK b* **unfolding** *K* **by** *auto*
have *rel-mset R M1 N* **using** *sK K1N K1M1*
unfolding *K multiset.rel-compp-Grp Grp-def* **by** *auto*
thus *?thesis* **using** *M Rab* **by** *auto*
qed

lemma *rel-mset-imp-rel-mset′*:
assumes *rel-mset R M N*
shows *rel-mset′ R M N*
using *assms* **proof**(*induct M arbitrary*: *N rule*: *measure-induct-rule*[*of size*])
case (*less M*)
have *c*: *size M* = *size N* **using** *rel-mset-size*[*OF less.prems*] **.**
show *?case*
proof(*cases M* = {#})
case *True* **hence** *N* = {#} **using** *c* **by** *simp*
thus *?thesis* **using** *True rel-mset′.Zero* **by** *auto*

next
case *False* **then obtain** *M1 a* **where** *M*: *M* = *add-mset a M1* **by** (*metis multi-nonempty-split*)
obtain *N1 b* **where** *N*: *N* = *add-mset b N1* **and** *R*: *R a b* **and** *ms*: *rel-mset R M1 N1*
using *msed-rel-invL*[*OF less.prems*[*unfolded M*]] **by** *auto*
have *rel-mset′ R M1 N1* **using** *less.hyps*[*of M1 N1*] *ms* **unfolding** *M* **by** *simp*
thus *?thesis* **using** *rel-mset′.Plus*[*of R a b*, *OF R*] **unfolding** *M N* **by** *simp*
qed
qed

lemma *rel-mset-rel-mset′*: *rel-mset R M N* = *rel-mset′ R M N*
using *rel-mset-imp-rel-mset′ rel-mset′-imp-rel-mset* **by** *auto*

The main end product for *rel-mset*: inductive characterization:

lemmas *rel-mset-induct*[*case-names empty add*, *induct pred*: *rel-mset*] =
rel-mset′.induct[*unfolded rel-mset-rel-mset′*[*symmetric*]]

1.18 Size setup

lemma *multiset-size-o-map*:
size-multiset g ∘ *image-mset f* = *size-multiset* (*g* ∘ *f*)
apply (*rule ext*)
subgoal for *x* **by** (*induct x*) *auto*
done

setup ‹
BNF-LFP-Size.register-size-global @{*type-name multiset*} @{*const-name size-multiset*}
@{*thm size-multiset-overloaded-def*}
@{*thms size-multiset-empty size-multiset-single size-multiset-union size-empty size-single*
size-union}
@{*thms multiset-size-o-map*}
›

hide-const (**open**) *wcount*

end

2 Additions to Isabelle's Main Theories

theory *Additions-to-Main*
imports ~~/*src*/*HOL*/*Library*/*Multiset*
begin

2.1 Addition to Finite-Set Theory

lemma *bound-domain-and-range-impl-finitely-many-functions*:
finite {*f*::*nat*⇒*nat*. (∀ *i*. *f i* ≤ *n*) ∧ (∀ *i*≥*m*. *f i* = *0*)}

proof (*induct m*)
case *0*
have *eq*: $\{f.\ (\forall i.\ f\ i \leq n) \wedge (\forall i.\ f\ i = 0)\} = \{(\lambda\text{-}.\ 0)\}$ **by** *auto*
from *this* **show** *?case* **by** *auto* (*subst eq*; *auto*)
next
case (*Suc m*)
let *?S* = $(\lambda(y, f).\ f(m := y))$ ‘ $(\{0..n\} \times \{f.\ (\forall i.\ f\ i \leq n) \wedge (\forall i{\geq}m.\ f\ i = 0)\})$
{
fix *g*
assume $\forall i.\ g\ i \leq n\ \forall i{\geq}Suc\ m.\ g\ i = 0$
from *this* **have** $g \in ?S$
by (*auto intro*: *image-eqI*[**where** $x{=}(g\ m,\ g(m{:=}0))$])
}
from *this* **have** $\{f.\ (\forall i.\ f\ i \leq n) \wedge (\forall i{\geq}Suc\ m.\ f\ i = 0)\} = ?S$ **by** *auto*
from *this Suc* **show** *?case* **by** *simp*
qed

2.2 Additions to Groups-Big Theory

lemma *sum-card-image*:
assumes *finite A*
assumes $\forall s{\in}A.\ \forall t{\in}A.\ s \neq t \longrightarrow (f\ s) \cap (f\ t) = \{\}$
shows *sum card* (*f* ‘ *A*) = *sum* $(\lambda a.\ card\ (f\ a))$ *A*
using *assms*
proof (*induct A*)
case *empty*
from *this* **show** *?case* **by** *simp*
next
case (*insert a A*)
show *?case*
proof *cases*
assume *f a* = {}
from *this insert* **show** *?case*
by (*subst sum.mono-neutral-right*[**where** *S*=*f* ‘ *A*]) *auto*
next
assume *f a* ≠ {}
from *this* **have** *sum card* (*insert* (*f a*) (*f* ‘ *A*)) = *card* (*f a*) + *sum card* (*f* ‘ *A*)
using *insert* **by** (*subst sum.insert*) *auto*
from *this insert* **show** *?case* **by** *simp*
qed
qed

lemma *card-Union-image*:
assumes *finite S*
assumes $\forall s{\in}f\ ‘\ S.$ *finite s*
assumes $\forall s{\in}S.\ \forall t{\in}S.\ s \neq t \longrightarrow (f\ s) \cap (f\ t) = \{\}$
shows *card* $(\bigcup(f\ ‘\ S)) = sum\ (\lambda x.\ card\ (f\ x))\ S$
proof −

have $\forall A{\in}f\ `\ S.\ \forall B{\in}f\ `\ S.\ A \neq B \longrightarrow A \cap B = \{\}$
using *assms*(*3*) **by** (*metis image-iff*)
from *this* **have** *card* $(\bigcup (f\ `\ S)) = sum\ card\ (f\ `\ S)$
using *assms*(*1*, *2*) **by** (*subst card-Union-disjoint*) *auto*
also have ... = *sum* $(\lambda x.\ card\ (f\ x))\ S$
using *assms*(*1*, *3*) **by** (*auto simp add*: *sum-card-image*)
finally show *?thesis* **.**
qed

2.3 Addition to Set-Interval Theory

lemma *sum-atMost-remove-nat*:
assumes $k \leq (n :: nat)$
shows $(\sum i{\leq}n.\ f\ i) = f\ k + (\sum i{\in}\{..n\}-\{k\}.\ f\ i)$
using *assms* **by** (*auto simp add*: *sum.remove*[**where** $x{=}k$])

2.4 Additions to Multiset Theory

lemma *set-mset-Abs-multiset*:
assumes $f \in multiset$
shows *set-mset* (*Abs-multiset f*) = $\{x.\ f\ x > 0\}$
using *assms* **unfolding** *set-mset-def* **by** *simp*

lemma *sum-mset-sum-count*:
sum-mset M = $(\sum i{\in}$*set-mset M*. *count M i* $*$ *i*$)$
proof (*induct M*)
show *sum-mset* $\{\#\}$ = $(\sum i{\in}$*set-mset* $\{\#\}$. *count* $\{\#\}$ *i* $*$ *i*$)$ **by** *simp*
next
fix *M x*
assume *hyp*: *sum-mset M* = $(\sum i{\in}$*set-mset M*. *count M i* $*$ *i*$)$
show *sum-mset* (*add-mset x M*) = $(\sum i{\in}$*set-mset* (*add-mset x M*). *count* (*add-mset x M*) *i* $*$ *i*$)$
proof (*cases x* $\in\#$ *M*)
assume *a*: $\neg$ *x* $\in\#$ *M*
from *this* **have** *count M x* = *0* **by** (*meson count-inI*)
from ‹$\neg$ *x* $\in\#$ *M*› *this hyp* **show** *?thesis*
by (*auto intro*!: *sum.cong*)
next
assume *x* $\in\#$ *M*
have *sum-mset* (*add-mset x M*) = $(\sum i{\in}$*set-mset M*. *count M i* $*$ *i*$)$ + *x*
using *hyp* **by** *simp*
also have ... = $(\sum i{\in}$*set-mset M* $-$ $\{x\}$. *count M i* $*$ *i*$)$ + *count M x* $*$ *x* + *x*
using ‹*x* $\in\#$ *M*› **by** (*simp add*: *sum.remove*[*of* - *x*])
also have ... = *count* (*add-mset x M*) *x* $*$ *x* + $(\sum i{\in}$*set-mset* (*add-mset x M*) $-$ $\{x\}$. *count* (*add-mset x M*) *i* $*$ *i*$)$
by *simp*
also have ... = $(\sum i{\in}$*set-mset* (*add-mset x M*). *count* (*add-mset x M*) *i* $*$ *i*$)$
using ‹*x* $\in\#$ *M*› **by** (*simp add*: *sum.remove*[*of* - *x*])
finally show *?thesis* **.**
qed

qed

lemma *sum-mset-eq-sum-on-supersets*:
assumes *finite A set-mset M* $\subseteq$ *A*
shows $(\sum i{\in}$*set-mset M*. *count M i* $*$ *i*$) = (\sum i{\in}$*A*. *count M i* $*$ *i*$)$
proof −
note ‹*finite A*› ‹*set-mset M* $\subseteq$ *A*›
moreover have $\forall\, i{\in}$*A* − *set-mset M*. *count M i* $*$ *i* = *0*
using *count-inI* **by** *fastforce*
ultimately show *?thesis*
by (*auto intro*: *sum.mono-neutral-cong-left*)
qed

end

3 Number Partitions

theory *Number-Partition*
imports *Additions-to-Main*
begin

3.1 Number Partitions as *nat* ⇒ *nat* Functions

definition *partitions* :: (*nat* ⇒ *nat*) ⇒ *nat* ⇒ *bool* (**infix** *partitions 50*)
where
p partitions n = $((\forall\, i.\ p\ i \neq 0 \longrightarrow 1 \leq i \wedge i \leq n) \wedge (\sum i{\leq}n.\ p\ i * i) = n)$

lemma *partitionsI*:
assumes $\bigwedge i.\ p\ i \neq 0 \Longrightarrow 1 \leq i \wedge i \leq n$
assumes $(\sum i{\leq}n.\ p\ i * i) = n$
shows *p partitions n*
using *assms* **unfolding** *partitions-def* **by** *auto*

lemma *partitionsE*:
assumes *p partitions n*
obtains $\bigwedge i.\ p\ i \neq 0 \Longrightarrow 1 \leq i \wedge i \leq n\ (\sum i{\leq}n.\ p\ i * i) = n$
using *assms* **unfolding** *partitions-def* **by** *auto*

lemma *partitions-zero*:
p partitions 0 $\longleftrightarrow$ *p* = $(\lambda i.\ 0)$
unfolding *partitions-def* **by** *auto*

lemma *partitions-one*:
p partitions (*Suc 0*) $\longleftrightarrow$ *p* = $(\lambda i.\ 0)(1 := 1)$
unfolding *partitions-def*
by (*auto split*: *if-split-asm*) (*auto simp add*: *fun-eq-iff*)

3.2 Bounds and Finiteness of Number Partitions

lemma *partitions-imp-finite-elements*:
assumes *p partitions n*
shows *finite* $\{i.\ 0 < p\ i\}$
proof −
from *assms* **have** $\{i.\ 0 < p\ i\} \subseteq \{..n\}$ **by** (*auto elim*: *partitionsE*)
from *this* **show** *?thesis*
using *rev-finite-subset* **by** *blast*
qed

lemma *partitions-imp-multiset*:
assumes *p partitions n*
shows $p \in$ *multiset*
using *assms partitions-imp-finite-elements multiset-def* **by** *auto*

lemma *partitions-bounds*:
assumes *p partitions n*
shows $p\ i \leq n$
proof −
from *assms* **have** *index-bounds*: $(\forall i.\ p\ i \neq 0 \longrightarrow 1 \leq i \wedge i \leq n)$
and *sum*: $(\sum i{\leq}n.\ p\ i * i) = n$
unfolding *partitions-def* **by** *auto*
show *?thesis*
proof (*cases* $1 \leq i \wedge i \leq n$)
case *True*
from *True* **have** $\{..n\} =$ *insert* $i\ \{i'.\ i' \leq n \wedge i' \neq i\}$ **by** *blast*
from *sum*[*unfolded this*] **have** $p\ i * i + (\sum i{\in}\{i'.\ i' \leq n \wedge i' \neq i\}.\ p\ i * i) = n$ **by** *auto*
from *this* **have** $p\ i * i \leq n$ **by** *linarith*
from *this True* **show** *?thesis* **using** *dual-order.trans* **by** *fastforce*
next
case *False*
from *this index-bounds* **show** *?thesis* **by** *fastforce*
qed
qed

lemma *partitions-parts-bounded*:
assumes *p partitions n*
shows *sum p* $\{..n\} \leq n$
proof −
{
fix *i*
assume $i \leq n$
from *assms* **have** $p\ i \leq p\ i * i$
by (*auto elim*!: *partitionsE*)
}
from *this* **have** *sum p* $\{..n\} \leq (\sum i{\leq}n.\ p\ i * i)$
by (*auto intro*: *sum-mono*)
also from *assms* **have** *n*: $(\sum i{\leq}n.\ p\ i * i) = n$

by (*auto elim*!: *partitionsE*)
finally show *?thesis* **.**
qed

lemma *finite-partitions*:
finite {*p*. *p partitions n*}
proof −
have {*p*. *p partitions n*} ⊆ {*f*. (∀ *i*. *f i* ≤ *n*) ∧ (∀ *i*. *n* + *1* ≤ *i* ⟶ *f i* = *0*)}
by (*auto elim*: *partitions-bounds*) (*auto simp add*: *partitions-def*)
from *this bound-domain-and-range-impl-finitely-many-functions*[*of n n* + *1*] **show** *?thesis*
by (*simp add*: *finite-subset*)
qed

lemma *finite-partitions-k-parts*:
finite {*p*. *p partitions n* ∧ *sum p* {..*n*} = *k*}
by (*simp add*: *finite-partitions*)

lemma *partitions-remaining-Max-part*:
assumes *p partitions n*
assumes *0* < *p k*
shows ∀ *i*. *n* − *k* < *i* ∧ *i* ≠ *k* ⟶ *p i* = *0*
proof (*clarify*)
fix *i*
assume *n* − *k* < *i i* ≠ *k*
show *p i* = *0*
proof (*cases i* ≤ *n*)
assume *i* ≤ *n*
from *assms* **have** *n*: (∑ *i*≤*n*. *p i* ∗ *i*) = *n* **and** *k* ≤ *n*
by (*auto elim*: *partitionsE*)
have (∑ *i*≤*n*. *p i* ∗ *i*) = *p k* ∗ *k* + (∑ *i*∈{..*n*}−{*k*}. *p i* ∗ *i*)
using ‹*k* ≤ *n*› *sum-atMost-remove-nat* **by** *blast*
also have ... = *p i* ∗ *i* + *p k* ∗ *k* + (∑ *i*∈{..*n*}−{*i*, *k*}. *p i* ∗ *i*)
using ‹*i* ≤ *n*› ‹*i* ≠ *k*›
by (*auto simp add*: *sum.remove*[**where** *x*=*i*]) (*metis Diff-insert*)
finally have *eq*: (∑ *i*≤*n*. *p i* ∗ *i*) = *p i* ∗ *i* + *p k* ∗ *k* + (∑ *i*∈{..*n*} − {*i*, *k*}. *p i* ∗ *i*) **.**
show *p i* = *0*
proof (*rule ccontr*)
assume *p i* ≠ *0*
have *upper-bound*: *p i* ∗ *i* + *p k* ∗ *k* ≤ *n*
using *eq n* **by** *auto*
have *lower-bound*: *p i* ∗ *i* + *p k* ∗ *k* > *n*
using ‹*n* − *k* < *i*› ‹*0* < *p k*› ‹*k* ≤ *n*› ‹*p i* ≠ *0*› *mult-eq-if not-less* **by** *auto*
from *upper-bound lower-bound* **show** *False* **by** *simp*
qed
next
assume ¬ (*i* ≤ *n*)
from *this* **show** *p i* = *0*

using *assms*(*1*) **by** (*auto elim*: *partitionsE*)
qed
qed

3.3 Operations of Number Partitions

lemma *partitions-remove1-bounds*:
assumes *partitions*: *p partitions n*
assumes *gr0*: *0 < p k*
assumes *neq*: $(p(k := p\ k - 1))\ i \neq 0$
shows $1 \leq i \wedge i \leq n - k$
proof
from *partitions neq* **show** $1 \leq i$
by (*auto elim*!: *partitionsE split*: *if-split-asm*)
next
from *partitions gr0* **have** *n*: $(\sum i{\leq}n.\ p\ i * i) = n$ **and** $k \leq n$
by (*auto elim*: *partitionsE*)
show $i \leq n - k$
proof *cases*
assume $k \leq n - k$
from ‹$k \leq n - k$› *neq* **show** *?thesis*
using *partitions-remaining-Max-part*[*OF partitions gr0*] *not-le* **by** *force*
next
assume $\neg\ k \leq n - k$
from *this* **have** $2 * k > n$ **by** *auto*
have $p\ k = 1$
proof (*rule ccontr*)
assume $p\ k \neq 1$
with *gr0* **have** $p\ k \geq 2$ **by** *auto*
from *this* **have** $p\ k * k \geq 2 * k$ **by** *simp*
with ‹$2 * k > n$› **have** $p\ k * k > n$ **by** *linarith*
from ‹$k \leq n$› *this* **have** $(\sum i{\leq}n.\ p\ i * i) > n$
by (*simp add*: *sum-atMost-remove-nat*[*of k*])
from *this n* **show** *False* **by** *auto*
qed
from *neq this* **show** *?thesis*
using *partitions-remaining-Max-part*[*OF partitions gr0*] *leI*
by (*auto split*: *if-split-asm*) *force*
qed
qed

lemma *partitions-remove1*:
assumes *partitions*: *p partitions n*
assumes *gr0*: *0 < p k*
shows $p(k := p\ k - 1)$ *partitions* $(n - k)$
proof (*rule partitionsI*)
fix *i*
assume $(p(k := p\ k - 1))\ i \neq 0$
from *this* **show** $1 \leq i \wedge i \leq n - k$ **using** *partitions-remove1-bounds partitions*

gr0 **by** *blast*
next
from *partitions gr0* **have** $k \leq n$ **by** (*auto elim*: *partitionsE*)
have $(\sum i{\leq}n - k.\ (p(k := p\ k - 1))\ i * i) = (\sum i{\leq}n.\ (p(k := p\ k - 1))\ i * i)$
using *partitions-remove1-bounds partitions gr0* **by** (*auto intro*!: *sum.mono-neutral-left*)
also have $\ldots = (p\ k - 1) * k + (\sum i{\in}\{..n\} - \{k\}.\ (p(k := p\ k - 1))\ i * i)$
using ‹$k \leq n$› **by** (*simp add*: *sum-atMost-remove-nat*[**where** $k{=}k$])
also have $\ldots = p\ k * k + (\sum i{\in}\{..n\} - \{k\}.\ p\ i * i) - k$
using *gr0* **by** (*simp add*: *diff-mult-distrib*)
also have $\ldots = (\sum i{\leq}n.\ p\ i * i) - k$
using ‹$k \leq n$› **by** (*simp add*: *sum-atMost-remove-nat*[*of k*])
also from *partitions* **have** $\ldots = n - k$
by (*auto elim*: *partitionsE*)
finally show $(\sum i{\leq}n - k.\ (p(k := p\ k - 1))\ i * i) = n - k$ **.**
qed

lemma *partitions-insert1*:
assumes *p*: *p partitions n*
assumes $k > 0$
shows $(p(k := p\ k + 1))$ *partitions* $(n + k)$
proof (*rule partitionsI*)
fix *i*
assume $(p(k := p\ k + 1))\ i \neq 0$
from *p this* ‹$k > 0$› **show** $1 \leq i \wedge i \leq n + k$
by (*auto elim*!: *partitionsE*)
next
have $(\sum i{\leq}n + k.\ (p(k := p\ k + 1))\ i * i) = p\ k * k + (\sum i{\in}\{..n + k\} - \{k\}.$
$p\ i * i) + k$
by (*simp add*: *sum-atMost-remove-nat*[*of k*])
also have $\ldots = p\ k * k + (\sum i{\in}\{..n\} - \{k\}.\ p\ i * i) + k$
using *p* **by** (*auto intro*!: *sum.mono-neutral-right elim*!: *partitionsE*)
also have $\ldots = (\sum i{\leq}n.\ p\ i * i) + k$
using *p* **by** (*cases* $k \leq n$) (*auto simp add*: *sum-atMost-remove-nat*[*of k*] *elim*:
partitionsE)
also have $\ldots = n + k$
using *p* **by** (*auto elim*: *partitionsE*)
finally show $(\sum i{\leq}n + k.\ (p(k := p\ k + 1))\ i * i) = n + k$ **.**
qed

lemma *count-remove1*:
assumes *p partitions n*
assumes $0 < p\ k$
shows $(\sum i{\leq}n - k.\ (p(k := p\ k - 1))\ i) = (\sum i{\leq}n.\ p\ i) - 1$
proof −
have $k \leq n$ **using** *assms* **by** (*auto elim*: *partitionsE*)
have $(\sum i{\leq}n - k.\ (p(k := p\ k - 1))\ i) = (\sum i{\leq}n.\ (p(k := p\ k - 1))\ i)$
using *partitions-remove1-bounds assms* **by** (*auto intro*!: *sum.mono-neutral-left*)
also have $(\sum i{\leq}n.\ (p(k := p\ k - 1))\ i) = p\ k + (\sum i{\in}\{..n\} - \{k\}.\ p\ i) - 1$
using ‹$0 < p\ k$› ‹$k \leq n$› **by** (*simp add*: *sum-atMost-remove-nat*[*of k*])

```
  also have ... = (∑ i∈{..n}. p i) - 1
    using ‹k ≤ n› by (simp add: sum-atMost-remove-nat[of k])
  finally show ?thesis .
qed

lemma count-insert1:
  assumes p partitions n
  shows sum (p(k := p k + 1)) {..n + k} = (∑ i≤n. p i) + 1
proof -
  have (∑ i≤n + k. (p(k := p k + 1)) i) = p k + (∑ i∈{..n + k} - {k}. p i)
+ 1
    by (simp add: sum-atMost-remove-nat[of k])
  also have ... = p k + (∑ i∈{..n} - {k}. p i) + 1
    using assms by (auto intro!: sum.mono-neutral-right elim!: partitionsE)
  also have ... = (∑ i≤n. p i) + 1
    using assms by (cases k ≤ n) (auto simp add: sum-atMost-remove-nat[of k]
elim: partitionsE)
  finally show ?thesis .
qed

lemma partitions-decrease1:
  assumes p: p partitions m
  assumes sum: sum p {..m} = k
  assumes p 1 = 0
  shows (λi. p (i + 1)) partitions m - k
proof -
  from p have p 0 = 0 by (auto elim!: partitionsE)
  {
    fix i
    assume neq: p (i + 1) ≠ 0
    from p this ‹p 1 = 0› have 1 ≤ i
      by (fastforce elim!: partitionsE simp add: le-Suc-eq)
    moreover have i ≤ m - k
    proof (rule ccontr)
      assume i-greater: ¬ i ≤ m - k
      from p have s: (∑ i≤m. p i * i) = m
        by (auto elim!: partitionsE)
      from p sum have k ≤ m
        using partitions-parts-bounded by fastforce
      from neq p have i + 1 ≤ m by (auto elim!: partitionsE)
      from i-greater have i > m - k by simp
      have ineq1: i + 1 > (m - k) + 1
        using i-greater by simp
      have ineq21: (∑ j≤m. (p(i + 1 := p (i + 1) - 1)) j * j) ≥ (∑ j≤m. (p(i
+ 1 := p (i + 1) - 1)) j)
        using ‹p 0 = 0› not-less by (fastforce intro!: sum-mono)
      have ineq22a: (∑ j≤m. (p(i + 1 := p (i + 1) - 1)) j) = (∑ j≤m. p j) - 1
        using ‹i + 1 ≤ m› neq by (simp add: sum.remove[where x=i + 1])
      have ineq22: (∑ j≤m. (p(i + 1 := p (i + 1) - 1)) j) ≥ k - 1
```

```
      using sum neq ineq22a by auto
    have ineq2: (∑j≤m. (p(i + 1 := p (i + 1) − 1)) j * j) ≥ k − 1
      using ineq21 ineq22 by auto
    have (∑ i≤m. p i * i) = p (i + 1) * (i + 1) + (∑ i∈{..m} − {i + 1}. p i
* i)
      using ‹i + 1 ≤ m› neq
      by (subst sum.remove[where x=i + 1]) auto
    also have ... = (i + 1) + (∑j≤m. (p(i + 1 := p (i + 1) − 1)) j * j)
      using ‹i + 1 ≤ m› neq
      by (subst sum.remove[where x=i + 1 and g=λj. (p(i + 1 := p (i + 1)
− 1)) j * j])
        (auto simp add: mult-eq-if)
    finally have (∑ i≤m. p i * i) = i + 1 + (∑j≤m. (p(i + 1 := p (i + 1)
− 1)) j * j) .
     moreover have ... > m using ineq1 ineq2 ‹k ≤ m› ‹p (i + 1) ≠ 0› by
linarith
    ultimately have (∑ i≤m. p i * i) > m by simp
    from s this show False by simp
  qed
  ultimately have 1 ≤ i ∧ i ≤ m − k ..
 } note bounds = this
 show (λi. p (i + 1)) partitions m − k
 proof (rule partitionsI)
  fix i
  assume p (i + 1) ≠ 0
  from bounds this show 1 ≤ i ∧ i ≤ m − k .
 next
  have geq: ∀ i. p i * i ≥ p i
    using ‹p 0 = 0› not-less by fastforce
  have (∑ i≤m − k. p (i + 1) * i) = (∑ i≤m. p (i + 1) * i)
    using bounds by (auto intro: sum.mono-neutral-left)
  also have ... = (∑ i∈Suc ‘ {..m}. p i * (i − 1))
    by (auto simp add: sum.reindex)
  also have ... = (∑ i≤Suc m. p i * (i − 1))
    using ‹p 0 = 0› by (simp add: Iic-Suc-eq-insert-0 zero-notin-Suc-image)
  also have ... = (∑ i≤m. p i * (i − 1))
    using p by (auto elim!: partitionsE)
  also have ... = (∑ i≤m. p i * i − p i)
    by (simp add: diff-mult-distrib2)
  also have ... = (∑ i≤m. p i * i) − (∑ i≤m. p i)
    using geq by (simp only: sum-subtractf-nat)
  also have ... = m − k using sum p by (auto elim!: partitionsE)
  finally show (∑ i≤m − k. p (i + 1) * i) = m − k .
 qed
qed

lemma partitions-increase1:
 assumes partitions: p partitions m − k
 assumes k: sum p {..m − k} = k
```

shows $(\lambda i.\ p\ (i - 1))$ *partitions* m
proof (*rule partitionsI*)
fix i
assume $p\ (i - 1) \neq 0$
from *partitions this k* **show** $1 \leq i \wedge i \leq m$
by (*cases k*) (*auto elim*!: *partitionsE*)
next
from k *partitions* **have** $k \leq m$
using *linear partitions-zero* **by** *force*
have *eq-0*: $\forall i{>}m - k.\ p\ i = 0$ **using** *partitions* **by** (*auto elim*!: *partitionsE*)
from *partitions* **have** *s*: $(\sum i{\leq}m - k.\ p\ i * i) = m - k$ **by** (*auto elim*!: *partitionsE*)
have $(\sum i{\leq}m.\ p\ (i - 1) * i) = (\sum i{\leq}Suc\ m.\ p\ (i - 1) * i)$
using *partitions k* **by** (*cases k*) (*auto elim*!: *partitionsE*)
also have $(\sum i{\leq}Suc\ m.\ p\ (i - 1) * i) = (\sum i{\leq}m.\ p\ i * (i + 1))$
by (*subst sum-atMost-Suc-shift*) *simp*
also have $... = (\sum i{\leq}m - k.\ p\ i * (i + 1))$
using *eq-0* **by** (*auto intro*: *sum.mono-neutral-right*)
also have $... = (\sum i{\leq}m - k.\ p\ i * i) + (\sum i{\leq}m - k.\ p\ i)$ **by** (*simp add*: *sum.distrib*)
also have $... = m - k + k$ **using** *s k* **by** *simp*
also have $... = m$ **using** ‹$k \leq m$› **by** *simp*
finally show $(\sum i{\leq}m.\ p\ (i - 1) * i) = m$ **.**
qed

lemma *count-decrease1*:
assumes *p*: *p partitions m*
assumes *sum*: *sum p* $\{..m\} = k$
assumes *p 1 = 0*
shows *sum* $(\lambda i.\ p\ (i + 1))\ \{..m - k\} = k$
proof −
from *p* **have** *p 0 = 0* **by** (*auto elim*!: *partitionsE*)
have *sum* $(\lambda i.\ p\ (i + 1))\ \{..m - k\} =$ *sum* $(\lambda i.\ p\ (i + 1))\ \{..m\}$
using *partitions-decrease1*[*OF assms*]
by (*auto intro*: *sum.mono-neutral-left elim*!: *partitionsE*)
also have $\ldots =$ *sum* $(\lambda i.\ p\ (i + 1))\ \{0..m\}$ **by** (*simp add*: *atLeast0AtMost*)
also have $\ldots =$ *sum* $(\lambda i.\ p\ i)\ \{Suc\ 0..\ Suc\ m\}$
by (*simp only*: *One-nat-def add-Suc-right add-0-right sum-shift-bounds-cl-Suc-ivl*)
also have $\ldots =$ *sum* $(\lambda i.\ p\ i)\ \{..\ Suc\ m\}$
using ‹*p 0 = 0*› **by** (*simp add*: *atLeast0AtMost sum-shift-lb-Suc0-0*)
also have $\ldots =$ *sum* $(\lambda i.\ p\ i)\ \{..\ m\}$
using *p* **by** (*auto elim*!: *partitionsE*)
also have $\ldots = k$
using *sum* **by** *simp*
finally show *?thesis* **.**
qed

lemma *count-increase1*:
assumes *partitions*: *p partitions* $m - k$

```
  assumes k: sum p {..m − k} = k
  shows (∑ i≤m. p (i − 1)) = k
proof −
  have p 0 = 0 using partitions by (auto elim!: partitionsE)
  have (∑ i≤m. p (i − 1)) = (∑ i∈{1..m}. p (i − 1))
    using ‹p 0 = 0› by (auto intro: sum.mono-neutral-cong-right)
  also have (∑ i∈{1..m}. p (i − 1)) = (∑ i≤m − 1. p i)
  proof (cases m)
    case 0
    from this show ?thesis using ‹p 0 = 0› by simp
  next
    case (Suc m')
    {
      fix x assume Suc 0 ≤ x x ≤ m
      from this Suc have x ∈ Suc ‘ {..m'}
        by (auto intro!: image-eqI[where x=x − 1])
    }
    from this Suc show ?thesis
      by (intro sum.reindex-cong[of Suc]) auto
  qed
  also have (∑ i≤m − 1. p i) = (∑ i≤m. p i)
  proof −
    {
      fix i
      assume 0 < p i i ≤ m
      from assms this have i ≤ m − 1
        using ‹p 0 = 0› partitions-increase1 by (cases k) (auto elim!: partitionsE)
    }
    from this show ?thesis
      by (auto intro: sum.mono-neutral-cong-left)
  qed
  also have ... = (∑ i≤m − k. p i)
    using partitions by (auto intro: sum.mono-neutral-right elim!: partitionsE)
  also have ... = k using k by auto
  finally show ?thesis .
qed
```

3.4 Number Partitions as Multisets on Natural Numbers

```
definition number-partition :: nat ⇒ nat multiset ⇒ bool
where
  number-partition n N = (sum-mset N = n ∧ 0 ∉# N)
```

3.4.1 Relationship to Definition on Functions

```
lemma count-partitions-iff:
  count N partitions n ⟷ number-partition n N
proof
  assume count N partitions n
```

```
  from this have (∀ i. count N i ≠ 0 ⟶ 1 ≤ i ∧ i ≤ n) (∑ i≤n. count N i * i)
= n
    unfolding Number-Partition.partitions-def by auto
  moreover from this have set-mset N ⊆ {..n} by auto
  moreover have finite {..n} by auto
  ultimately have sum-mset N = n
    using sum-mset-sum-count sum-mset-eq-sum-on-supersets by presburger
  moreover have 0 ∉# N
    using ‹∀ i. count N i ≠ 0 ⟶ 1 ≤ i ∧ i ≤ n› by auto
  ultimately show number-partition n N
    unfolding number-partition-def by auto
next
  assume number-partition n N
  from this have sum-mset N = n and 0 ∉# N
    unfolding number-partition-def by auto
  {
    fix i
    assume count N i ≠ 0
    have 1 ≤ i ∧ i ≤ n
    proof
      from ‹0 ∉# N› ‹count N i ≠ 0› show 1 ≤ i
        using Suc-le-eq by auto
      from ‹sum-mset N = n› ‹count N i ≠ 0› show i ≤ n
        using multi-member-split by fastforce
    qed
  }
  moreover from ‹sum-mset N = n› have (∑ i≤n. count N i * i) = n
   by (metis atMost-iff calculation finite-atMost not-in-iff subsetI sum-mset-eq-sum-on-supersets
sum-mset-sum-count)
  ultimately show count N partitions n
    by (rule partitionsI) auto
qed

lemma partitions-iff-Abs-multiset:
  p partitions n ⟷ finite {x. 0 < p x} ∧ number-partition n (Abs-multiset p)
proof
  assume p partitions n
  from this have bounds: (∀ i. p i ≠ 0 ⟶ 1 ≤ i ∧ i ≤ n)
    and sum: (∑ i≤n. p i * i) = n
  unfolding partitions-def by auto
  from ‹p partitions n› have p ∈ multiset by (rule partitions-imp-multiset)
  from ‹p partitions n› have finite {x. 0 < p x}
    by (rule partitions-imp-finite-elements)
  moreover from ‹p ∈ multiset› bounds have ¬ 0 ∈# Abs-multiset p
    using count-eq-zero-iff by force
  moreover from ‹p ∈ multiset› this sum have sum-mset (Abs-multiset p) = n
  proof −
    have (∑ i∈{x. 0 < p x}. p i * i) = (∑ i≤n. p i * i)
      using bounds by (auto intro: sum.mono-neutral-cong-left)
```

```
    from ‹p ∈ multiset› this sum show sum-mset (Abs-multiset p) = n
      by (simp add: sum-mset-sum-count set-mset-Abs-multiset)
  qed
  ultimately show finite {x. 0 < p x} ∧ number-partition n (Abs-multiset p)
    unfolding number-partition-def by auto
next
  assume finite {x. 0 < p x} ∧ number-partition n (Abs-multiset p)
  from this have finite {x. 0 < p x} 0 ∉# Abs-multiset p sum-mset (Abs-multiset
p) = n
    unfolding number-partition-def by auto
  from ‹finite {x. 0 < p x}› have p ∈ multiset by (simp add: multiset-def)
  from ‹p ∈ multiset› have (∑ i∈{x. 0 < p x}. p i * i) = n
    using ‹ sum-mset (Abs-multiset p) = n›
    by (simp add: sum-mset-sum-count set-mset-Abs-multiset)
  have bounds: ⋀i. p i ≠ 0 ⟹ 1 ≤ i ∧ i ≤ n
  proof
    fix i
    assume p i ≠ 0
    from ‹¬ 0 ∈# Abs-multiset p› ‹p ∈ multiset› have p 0 = 0
      using count-inI by force
    from this ‹p i ≠ 0› show 1 ≤ i
      by (metis One-nat-def leI less-Suc0)
    show i ≤ n
    proof (rule ccontr)
      assume ¬ i ≤ n
      from this have i > n
        using le-less-linear by blast
      from this ‹p i ≠ 0› have p i * i > n
        by (auto simp add: less-le-trans)
      from ‹p i ≠ 0› have (∑ i∈{x. 0 < p x}. p i * i) = p i * i + (∑ i∈{x. 0 <
p x} − {i}. p i * i)
        using ‹finite {x. 0 < p x}›
        by (subst sum.insert-remove[symmetric]) (auto simp add: insert-absorb)
      also from ‹p i * i > n› have ... > n by auto
      finally show False using ‹(∑ i∈{x. 0 < p x}. p i * i) = n› by blast
    qed
  qed
  moreover have (∑ i≤n. p i * i) = n
  proof −
    have (∑ i≤n. p i * i) = (∑ i∈{x. 0 < p x}. p i * i)
      using bounds by (auto intro: sum.mono-neutral-cong-right)
    from this show ?thesis
      using ‹(∑ i∈{x. 0 < p x}. p i * i) = n› by simp
  qed
  ultimately show p partitions n by (auto intro: partitionsI)
qed


lemma size-nat-multiset-eq:
```

fixes N :: *nat multiset*
assumes *number-partition n N*
shows *size N* = *sum* (*count N*) {*..n*}
proof −
have *set-mset N* ⊆ {*..sum-mset N*}
by (*auto dest*: *multi-member-split*)
have *size N* = *sum* (*count N*) (*set-mset N*)
by (*rule size-multiset-overloaded-eq*)
also have ... = *sum* (*count N*) {*..sum-mset N*}
using ‹*set-mset N* ⊆ {*..sum-mset N*}›
by (*auto intro*: *sum.mono-neutral-cong-left count-inI*)
finally show *?thesis*
using ‹*number-partition n N*›
unfolding *number-partition-def* **by** *auto*
qed

end

4 Euler's Partition Theorem

theory *Euler-Partition*
imports
Main
../Card-Number-Partitions/Number-Partition
begin

4.1 Preliminaries

4.1.1 Additions to Divides Theory

lemma *power-div-nat*:
assumes $c \leq b$
assumes $a > 0$
shows $(a :: nat)\ \hat{}\ b\ div\ a\ \hat{}\ c = a\ \hat{}\ (b - c)$
by (*metis assms nonzero-mult-div-cancel-right le-add-diff-inverse2 less-not-refl2 power-add power-not-zero*)

4.1.2 Additions to Groups-Big Theory

lemma *sum-div*:
assumes *finite A*
assumes $\bigwedge a.\ a \in A \Longrightarrow (b::'b::semiring\text{-}div)\ dvd\ f\ a$
shows $(\sum a{\in}A.\ f\ a)\ div\ b = (\sum a{\in}A.\ (f\ a)\ div\ b)$
using *assms*
proof (*induct*)
case *insert* **from** *this* **show** *?case* **by** *auto* (*subst div-add*; *auto intro*!: *dvd-sum*)
qed (*auto*)

lemma *sum-mod*:

assumes *finite A*
assumes $\bigwedge a.\ a \in A \Longrightarrow f\ a\ mod\ b = (0::'b::\{semiring\text{-}div\})$
shows $(\sum a{\in}A.\ f\ a)\ mod\ b = 0$
using *assms* **by** *induct* (*auto simp add*: *mod-add-eq*)

4.1.3 Additions to Set-Interval Theory

lemma *geometric-sum-2nat*:
$(\sum i{<}n.\ (2::nat)\ \hat{}\ i) = (2\ \hat{}\ n - 1)$
by (*induct n*) *auto*

4.1.4 Additions to Nat Theory or Power Theory

lemma *n-leq-2-pow-n*:
$n \leq 2\ \hat{}\ n$
proof (*induct n*)
case (*Suc n*)
from *this* **have** *approx*: $2 * n \leq 2\ \hat{}\ Suc\ n$ **by** *auto*
show *?case*
proof (*cases n*)
case *0*
from *this* **show** *?thesis* **by** *simp*
next
case *Suc*
from *this* **show** *?thesis* **by** (*intro le-trans*[*OF - approx*]) *simp*
qed
qed (*simp*)

4.1.5 Additions to Finite-Set Theory

lemma *finite-exponents*:
finite $\{i.\ 2\ \hat{}\ i \leq (n::nat)\}$
proof −
have $\{i::nat.\ 2\ \hat{}\ i \leq n\} \subseteq \{..n\}$
using *dual-order.trans n-leq-2-pow-n* **by** *auto*
from *this* **show** *?thesis* **by** (*simp add*: *finite-subset*)
qed

4.2 Binary Encoding of Natural Numbers

definition *bitset* :: *nat* ⇒ *nat set*
where
bitset n = $\{i.\ odd\ (n\ div\ (2\ \hat{}\ i))\}$

lemma *in-bitset-bound*:
$b \in bitset\ n \Longrightarrow 2\ \hat{}\ b \leq n$
unfolding *bitset-def* **using** *not-less* **by** *fastforce*

lemma *in-bitset-bound-weak*:
$b \in bitset\ n \Longrightarrow b \leq n$

by (*auto dest*: *in-bitset-bound intro*: *le-trans n-leq-2-pow-n*)

lemma *finite-bitset*:
  finite (*bitset n*)
proof −
  have *bitset n* ⊆ {*..n*} **by** (*auto dest*: *in-bitset-bound-weak*)
  from *this* **show** *?thesis* **using** *finite-subset* **by** *auto*
qed

lemma *bitset-0*:
  bitset 0 = {}
unfolding *bitset-def* **by** *auto*

lemma *binary-induct* [*case-names zero even odd*]:
  assumes *P* (*0* :: *nat*)
  assumes ⋀*n*. *P n* ⟹ *P* (*2* ∗ *n*)
  assumes ⋀*n*. *P n* ⟹ *P* (*2* ∗ *n* + *1*)
  shows ⋀*n*. *P n*
using *assms parity-induct* **by** *auto*

lemma *bitset-2n*: *bitset* (*2* ∗ *n*) = *Suc* ‘ (*bitset n*)
proof (*rule set-eqI*)
  fix *x*
  show (*x* ∈ *bitset* (*2* ∗ *n*)) = (*x* ∈ *Suc* ‘ *bitset n*)
    unfolding *bitset-def* **by** (*cases x*) *auto*
qed

lemma *bitset-Suc*:
  assumes *even n*
  shows *bitset* (*n* + *1*) = *insert 0* (*bitset n*)
proof (*rule set-eqI*)
  fix *x*
  from *assms* **show** (*x* ∈ *bitset* (*n* + *1*)) = (*x* ∈ *insert 0* (*bitset n*))
    unfolding *bitset-def* **by** (*cases x*) (*auto simp add*: *Divides.div-mult2-eq*)
qed

lemma *bitset-2n1*:
  bitset (*2* ∗ *n* + *1*) = *insert 0* (*Suc* ‘ (*bitset n*))
by (*subst bitset-Suc*) (*auto simp add*: *bitset-2n*)

lemma *sum-bitset*:
  (∑ *i*∈*bitset n*. *2* ^ *i*) = *n*
proof (*induct rule*: *binary-induct*)
  case *zero*
  show *?case* **by** (*auto simp add*: *bitset-0*)
next
  case (*even n*)
  from *this* **show** *?case*
    by (*simp add*: *bitset-2n sum.reindex sum-distrib-left*[*symmetric*])

next
case (*odd n*)
have ($\sum i{\in}bitset\ (2 * n + 1).\ 2\ \hat{}\ i) = (\sum i{\in}insert\ 0\ (Suc\ `\ bitset\ n).\ 2\ \hat{}\ i)$
by (*simp only*: *bitset-2n1*)
also have ... = $2\ \hat{}\ 0 + (\sum i{\in}Suc\ `\ bitset\ n.\ 2\ \hat{}\ i)$
by (*subst sum.insert*) (*auto simp add*: *finite-bitset*)
also have ... = $2 * n + 1$
using *odd* **by** (*simp add*: *sum.reindex sum-distrib-left*[*symmetric*])
finally show *?case* **.**
qed

lemma *binarysum-div*:
assumes *finite B*
shows $(\sum i{\in}B.\ (2{::}nat)\ \hat{}\ i)\ div\ 2\ \hat{}\ j = (\sum i{\in}B.\ if\ i < j\ then\ 0\ else\ 2\ \hat{}\ (i - j))$
(**is** - = $(\sum i{\in}\text{-}.\ ?f\ i)$)
proof −
have *split-B*: $B = \{i{\in}B.\ i < j\} \cup \{i{\in}B.\ j \le i\}$ **by** *auto*
have *bound*: $(\sum i \mid i \in B \wedge i < j.\ (2{::}nat)\ \hat{}\ i) < 2\ \hat{}\ j$
proof (*rule order.strict-trans1*)
show $(\sum i \mid i \in B \wedge i < j.\ (2{::}nat)\ \hat{}\ i) \le (\sum i{<}j.\ 2\ \hat{}\ i)$ **by** (*auto intro*: *sum-mono2*)
show ... < $2\ \hat{}\ j$ **by** (*simp add*: *geometric-sum-2nat*)
qed
from *this* **have** *zero*: $(\sum i \mid i \in B \wedge i < j.\ (2{::}nat)\ \hat{}\ i)\ div\ (2\ \hat{}\ j) = 0$ **by** (*elim div-less*)
from *assms* **have** *mod0*: $(\sum i \mid i \in B \wedge j \le i.\ (2{::}nat)\ \hat{}\ i)\ mod\ 2\ \hat{}\ j = 0$
by (*auto intro*!: *sum-mod simp add*: *le-imp-power-dvd*)
from *assms* **have** $(\sum i{\in}B.\ (2{::}nat)\ \hat{}\ i)\ div\ (2\ \hat{}\ j) = ((\sum i \mid i \in B \wedge i < j.\ 2\ \hat{}\ i) + (\sum i \mid i \in B \wedge j \le i.\ 2\ \hat{}\ i))\ div\ 2\ \hat{}\ j$
by (*subst sum.union-disjoint*[*symmetric*]) (*auto simp add*: *split-B*[*symmetric*])
also have ... = $(\sum i \mid i \in B \wedge j \le i.\ 2\ \hat{}\ i)\ div\ 2\ \hat{}\ j$
by (*simp add*: *div-add1-eq zero mod0*)
also have ... = $(\sum i \mid i \in B \wedge j \le i.\ 2\ \hat{}\ i\ div\ 2\ \hat{}\ j)$
using *assms* **by** (*subst sum-div*) (*auto simp add*: *sum-div le-imp-power-dvd*)
also have ... = $(\sum i \mid i \in B \wedge j \le i.\ 2\ \hat{}\ (i - j))$
by (*rule sum.cong*[*OF refl*]) (*auto simp add*: *power-div-nat*)
also have ... = $(\sum i{\in}B.\ ?f\ i)$
using *assms* **by** (*subst split-B*; *subst sum.union-disjoint*) *auto*
finally show *?thesis* **.**
qed

lemma *odd-iff*:
assumes *finite B*
shows *odd* $(\sum i{\in}B.\ if\ i < x\ then\ (0{::}nat)\ else\ 2\ \hat{}\ (i - x)) = (x \in B)$ (**is** *odd* $(\sum i{\in}\text{-}.\ ?s\ i) = \text{-}$)
proof −
from *assms* **have** *even*: *even* $(\sum i{\in}B - \{x\}.\ ?s\ i)$
by (*subst dvd-sum*) *auto*

show *?thesis*
proof
assume *odd* $(\sum i{\in}B.\ ?s\ i)$
from *this even* **show** $x \in B$ **by** (*cases* $x \in B$) *auto*
next
assume $x \in B$
from *assms this* **have** $(\sum i{\in}B.\ ?s\ i) = 1 + (\sum i{\in}B{-}\{x\}.\ ?s\ i)$
by (*auto simp add*: *sum.remove*)
from *assms this even* **show** *odd* $(\sum i{\in}B.\ ?s\ i)$ **by** *auto*
qed
qed

lemma *bitset-sum*:
assumes *finite B*
shows *bitset* $(\sum i{\in}B.\ 2\ \hat{}\ i) = B$
using *assms* **unfolding** *bitset-def* **by** (*simp add*: *binarysum-div odd-iff*)

4.3 Decomposition of a Number into a Power of Two and an Odd Number

function (*sequential*) *index* :: *nat* ⇒ *nat*
where
index 0 = 0
| *index n* = (*if odd n then 0 else Suc* (*index* (*n div 2*)))
by (*pat-completeness*) *auto*

termination
proof
show *wf* {(*x*::*nat*, *y*). *x* < *y*} **by** (*simp add*: *wf*)
next
fix *n* **show** (*Suc n div 2*, *Suc n*) ∈ {(*x*, *y*). *x* < *y*} **by** *simp*
qed

function (*sequential*) *oddpart* :: *nat* ⇒ *nat*
where
oddpart 0 = 0
| *oddpart n* = (*if odd n then n else oddpart* (*n div 2*))
by *pat-completeness auto*

termination
proof
show *wf* {(*x*::*nat*, *y*). *x* < *y*} **by** (*simp add*: *wf*)
next
fix *n* **show** (*Suc n div 2*, *Suc n*) ∈ {(*x*, *y*). *x* < *y*} **by** *simp*
qed

lemma *odd-oddpart*:
odd (*oddpart n*) ⟷ *n* ≠ *0*
by (*induct n rule*: *index.induct*) *auto*

lemma *index-oddpart-decomposition*:
  $n = 2 \hat{} (index\ n) * oddpart\ n$
proof (*induct n rule*: *index.induct*)
  case (*2 n*)
  from *this* **show** $Suc\ n = 2 \hat{} index\ (Suc\ n) * oddpart\ (Suc\ n)$
    by (*simp add*: *mult.assoc*)
qed (*simp*)

lemma *oddpart-leq*:
  $oddpart\ n \leq n$
by (*induct n rule*: *index.induct*) (*simp*, *metis div-le-dividend le-Suc-eq le-trans oddpart.simps*(*2*))

lemma *index-oddpart-unique*:
  assumes *odd* $(m :: nat)$ *odd* m'
  shows $(2 \hat{} i * m = 2 \hat{} i' * m') \longleftrightarrow (i = i' \wedge m = m')$
proof (*induct i arbitrary*: i')
  case *0*
  from *assms* **show** *?case* **by** *auto*
next
  case (*Suc - i'*)
  from *assms this* **show** *?case* **by** (*cases i'*) *auto*
qed

lemma *index-oddpart*:
  assumes *odd m*
  shows $index\ (2 \hat{} i * m) = i\ oddpart\ (2 \hat{} i * m) = m$
using *index-oddpart-unique*[**where** $i{=}i$ **and** $m{=}m$ **and** $m'{=}oddpart\ (2 \hat{} i * m)$ **and** $i'{=}index\ (2 \hat{} i * m)$]
  assms odd-oddpart index-oddpart-decomposition **by** *force+*

4.4 Partitions With Only Distinct and Only Odd Parts

definition *odd-of-distinct* :: $(nat \Rightarrow nat) \Rightarrow nat \Rightarrow nat$
where
  $odd\text{-}of\text{-}distinct\ p = (\lambda i.\ if\ odd\ i\ then\ (\sum j \mid p\ (2 \hat{} j * i) = 1.\ 2 \hat{} j)\ else\ 0)$

definition *distinct-of-odd* :: $(nat \Rightarrow nat) \Rightarrow nat \Rightarrow nat$
where
  $distinct\text{-}of\text{-}odd\ p = (\lambda i.\ if\ index\ i \in bitset\ (p\ (oddpart\ i))\ then\ 1\ else\ 0)$

lemma *odd*:
  $odd\text{-}of\text{-}distinct\ p\ i \neq 0 \Longrightarrow odd\ i$
unfolding *odd-of-distinct-def* **by** *auto*

lemma *distinct-distinct-of-odd*:
  $distinct\text{-}of\text{-}odd\ p\ i \leq 1$
unfolding *distinct-of-odd-def* **by** *auto*

lemma *odd-of-distinct*:
assumes *odd-of-distinct p i* $\neq$ *0*
assumes $\bigwedge i.\ p\ i \neq 0 \Longrightarrow i \leq n$
shows $1 \leq i \wedge i \leq n$
proof
from *assms*(*1*) **show** $1 \leq i$ **by** (*metis One-nat-def Suc-leI even-zero gr0I odd*)
next
from *assms*(*1*) **obtain** *j* **where** *p* (*2* ^ *j* * *i*) $\neq$ *0*
unfolding *odd-of-distinct-def* **by** (*auto split*: *if-split-asm*) *fastforce*
from *assms*(*2*)[*OF this*] **show** $i \leq n$
by (*metis div-le-dividend nonzero-mult-div-cancel-left le-trans power-not-zero zero-not-eq-two*)
qed

lemma *distinct-of-odd*:
assumes $\bigwedge i.\ p\ i * i \leq n$ $\bigwedge i.\ p\ i \neq 0 \Longrightarrow$ *odd i*
assumes *distinct-of-odd p i* $\neq$ *0*
shows $1 \leq i \wedge i \leq n$
proof
from *assms*(*3*) **have** *index*: *index i* $\in$ *bitset* (*p* (*oddpart i*))
unfolding *distinct-of-odd-def* **by** (*auto split*: *if-split-asm*)
have *i* $\neq$ *0*
proof
assume *zero*: *i* = *0*
from *assms*(*2*) **have** *p 0* = *0* **by** *auto*
from *index zero this* **show** *False* **by** (*auto simp add*: *bitset-0*)
qed
from *this* **show** $1 \leq i$ **by** *auto*
from *assms*(*1*) **have** *leq-n*: *p* (*oddpart i*) * *oddpart i* $\leq n$ **by** *auto*
from *index* **have** *2* ^ *index i* $\leq$ *p* (*oddpart i*) **by** (*rule in-bitset-bound*)
from *this leq-n* **show** $i \leq n$
by (*subst index-oddpart-decomposition*[*of i*]) (*meson dual-order.trans eq-imp-le mult-le-mono*)
qed

lemma *odd-distinct*:
assumes $\bigwedge i.\ p\ i \neq 0 \Longrightarrow$ *odd i*
shows *odd-of-distinct* (*distinct-of-odd p*) = *p*
using *assms* **unfolding** *odd-of-distinct-def distinct-of-odd-def*
by (*auto simp add*: *fun-eq-iff index-oddpart sum-bitset*)

lemma *distinct-odd*:
assumes $\bigwedge i.\ p\ i \neq 0 \Longrightarrow 1 \leq i \wedge i \leq n$ $\bigwedge i.\ p\ i \leq 1$
shows *distinct-of-odd* (*odd-of-distinct p*) = *p*
proof −
from *assms* **have** {*i*. *p i* = *1*} $\subseteq$ {..*n*} **by** *auto*
from *this* **have** *finite*: *finite* {*i*. *p i* = *1*} **by** (*simp add*: *finite-subset*)
have $\bigwedge x\ j.\ x > 0 \Longrightarrow$ *p* (*2* ^ *j* * *oddpart x*) = *1* $\Longrightarrow$

```
    index (2 ^ j * oddpart x) ∈ index ‘ {i. p i = 1 ∧ oddpart x = oddpart i}
    by (rule imageI) (auto intro: imageI simp add: index-oddpart odd-oddpart)
  from this have eq: ⋀x. x > 0 ⟹ {j. p (2 ^ j * oddpart x) = 1} = index ‘ {i.
p i = 1 ∧ oddpart x = oddpart i}
   by (auto simp add: index-oddpart odd-oddpart index-oddpart-decomposition[symmetric])
  from finite have all-finite: ⋀x. x > 0 ⟹ finite {j. p (2 ^ j * oddpart x) = 1}
    unfolding eq by auto
  show ?thesis
  proof
    fix x
    from assms(1) have p0: p 0 = 0 by auto
    show distinct-of-odd (odd-of-distinct p) x = p x
    proof (cases x > 0)
      case False
      from this p0 show ?thesis
        unfolding odd-of-distinct-def distinct-of-odd-def
        by (auto simp add: odd-oddpart bitset-0)
    next
      case True
      from p0 assms(2)[of x] all-finite[OF True] show ?thesis
        unfolding odd-of-distinct-def distinct-of-odd-def
      by (auto simp add: odd-oddpart bitset-0 bitset-sum index-oddpart-decomposition[symmetric])
    qed
  qed
qed

lemma sum-distinct-of-odd:
  assumes ⋀i. p i ≠ 0 ⟹ 1 ≤ i ∧ i ≤ n
  assumes ⋀i. p i * i ≤ n
  assumes ⋀i. p i ≠ 0 ⟹ odd i
  shows (∑ i≤n. distinct-of-odd p i * i) = (∑ i≤n. p i * i)
proof −
  {
    fix m
    assume odd: odd (m :: nat)
     have finite: finite {k. 2 ^ k * m ≤ n ∧ k ∈ bitset (p m)} by (simp add:
finite-bitset)
    have (∑ i | ∃ k. i = 2 ^ k * m ∧ i ≤ n. distinct-of-odd p i * i) =
      (∑ i | ∃ k. i = 2 ^ k * m ∧ i ≤ n. if index i ∈ bitset (p (oddpart i)) then i
else 0)
      unfolding distinct-of-odd-def by (auto intro: sum.cong)
    also have ... = (∑ i | ∃ k. i = 2 ^ k * m ∧ k ∈ bitset (p m) ∧ i ≤ n. i)
    using odd by (intro sum.mono-neutral-cong-right) (auto simp add: index-oddpart)
    also have ... = (∑ k | 2 ^ k * m ≤ n ∧ k ∈ bitset (p m). 2 ^ k * m)
      using odd by (auto intro!: sum.reindex-cong[OF - - refl] inj-onI)
    also have ... = (∑ k∈bitset (p m). 2 ^ k * m)
      using assms(2)[of m] finite dual-order.trans in-bitset-bound
      by (fastforce intro!: sum.mono-neutral-cong-right)
    also have ... = (∑ k∈bitset (p m). 2 ^ k) * m
```

```
    by (subst sum-distrib-right) auto
  also have ... = p m * m
    by (auto simp add: sum-bitset)
  finally have (∑ i | ∃ k. i = 2 ^ k * m ∧ i ≤ n. distinct-of-odd p i * i) = p m
* m .
 } note inner-eq = this

 have set-eq: {i. 1 ≤ i ∧ i ≤ n} = ⋃((λm. {i. ∃ k. i = (2 ^ k) * m ∧ i ≤ n}) ‘
{m. m ≤ n ∧ odd m})
 proof −
  {
   fix x
   assume 1 ≤ x x ≤ n
   from this oddpart-leq[of x] have oddpart x ≤ n ∧ odd (oddpart x) ∧ (∃ k. 2
^ index x * oddpart x = 2 ^ k * oddpart x)
    by (auto simp add: odd-oddpart)
   from this have ∃ m≤n. odd m ∧ (∃ k. x = 2 ^ k * m)
    by (auto simp add: index-oddpart-decomposition[symmetric])
  }
  from this show ?thesis by (auto simp add: Suc-leI odd-pos)
 qed
 let ?S = (λm. {i. ∃ k. i = 2 ^ k * m ∧ i ≤ n}) ‘ {m. m ≤ n ∧ odd m}
 have no-overlap: ∀ A∈?S. ∀ B∈?S. A ≠ B ⟶ A ∩ B = {}
  by (auto simp add: index-oddpart-unique)
 have inj: inj-on (λm. {i. (∃ k. i = 2 ^ k * m) ∧ i ≤ n}) {m. m ≤ n ∧ odd m}
  unfolding inj-on-def by auto (force simp add: index-oddpart-unique)
 have reindex: ⋀F. (∑ i | 1 ≤ i ∧ i ≤ n. F i) = (∑ m | m ≤ n ∧ odd m. (∑ i
| ∃ k. i = 2 ^ k * m ∧ i ≤ n. F i))
   unfolding set-eq by (subst sum.Union-disjoint) (auto simp add: no-overlap
intro: sum.reindex-cong[OF inj])
 have (∑ i≤n. distinct-of-odd p i * i) =  (∑ i | 1 ≤ i ∧ i ≤ n. distinct-of-odd
p i * i)
  by (auto intro: sum.mono-neutral-right)
 also have ... = (∑ m | m ≤ n ∧ odd m. ∑ i | ∃ k. i = 2 ^ k * m ∧ i ≤ n.
distinct-of-odd p i * i)
  by (simp only: reindex)
 also have ... = (∑ i | i ≤ n ∧ odd i. p i * i)
  by (rule sum.cong[OF refl]; subst inner-eq) auto
 also have ... = (∑ i≤n. p i * i)
  using assms(3) by (auto intro: sum.mono-neutral-left)
 finally show ?thesis .
qed

lemma leq-n:
 assumes ∀ i. 0 < p i ⟶ 1 ≤ i ∧ i ≤ (n::nat)
 assumes (∑ i≤n. p i * i) = n
 shows p i * i ≤ n
proof (rule ccontr)
 assume ¬ p i * i ≤ n
```

```
  from this have gr-n: p i * i > n by auto
  from this assms(1) have 1 ≤ i ∧ i ≤ n by force
  from this have (∑j≤n. p j * j) = p i * i + (∑j | j ≤ n ∧ j ≠ i. p j * j)
    by (subst sum.insert[symmetric]) (auto intro: sum.cong simp del: sum.insert)
  from this gr-n assms(2) show False by simp
qed

lemma distinct-of-odd-in-distinct-partitions:
  assumes p ∈ {p. p partitions n ∧ (∀ i. p i ≠ 0 ⟶ odd i)}
  shows distinct-of-odd p ∈ {p. p partitions n ∧ (∀ i. p i ≤ 1)}
proof
  have distinct-of-odd p partitions n
  proof (rule partitionsI)
    fix i assume distinct-of-odd p i ≠ 0
    from this assms show 1 ≤ i ∧ i ≤ n
    unfolding partitions-def
    by (rule-tac distinct-of-odd) (auto simp add: leq-n)
  next
    from assms show (∑ i≤n. distinct-of-odd p i * i) = n
      by (subst  sum-distinct-of-odd) (auto simp add: distinct-distinct-of-odd leq-n
elim: partitionsE)
  qed
  moreover have ∀ i. distinct-of-odd p i ≤ 1
    by (intro allI distinct-distinct-of-odd)
  ultimately show distinct-of-odd p partitions n ∧ (∀ i. distinct-of-odd p i ≤ 1)
by simp
qed

lemma odd-of-distinct-in-odd-partitions:
  assumes p ∈ {p. p partitions n ∧ (∀ i. p i ≤ 1)}
  shows odd-of-distinct p ∈ {p. p partitions n ∧ (∀ i. p i ≠ 0 ⟶ odd i)}
proof
  from assms have distinct: ⋀i. p i = 0 ∨ p i = 1
    using le-imp-less-Suc less-Suc-eq-0-disj by fastforce
  from assms have set-eq: {x. p x = 1} = {x ∈ {..n}. p x = 1}
     unfolding partitions-def by auto
  from assms have sum: (∑ i≤n. p i * i) = n
     unfolding partitions-def by auto
  {
    fix i
    assume i: odd (i :: nat)
    have 3: inj-on index {x. p x = 1 ∧ oddpart x = i}
      unfolding inj-on-def by auto (metis index-oddpart-decomposition)
    {
      fix j assume p (2 ^ j * i) = 1
      from this i have j ∈ index ‘ {x. p x = 1 ∧ oddpart x = i}
        by (auto simp add: index-oddpart(1, 2) intro!: image-eqI[where x=2 ^ j *
i])
    }
```

from *i this* **have** $\{j.\ p\ (2\ \hat{}\ j * i) = 1\} = index\ `\ \{x.\ p\ x = 1 \wedge oddpart\ x = i\}$
by (*auto simp add*: *index-oddpart-decomposition*[*symmetric*])
from *3 this* **have** $(\sum j \mid p\ (2\ \hat{}\ j * i) = 1.\ 2\ \hat{}\ j) * i = (\sum x \mid p\ x = 1 \wedge oddpart\ x = i.\ 2\ \hat{}\ index\ x) * i$
by (*auto intro*: *sum.reindex-cong*[**where** *l* = *index*])
also have ... = $(\sum x \mid p\ x = 1 \wedge oddpart\ x = i.\ 2\ \hat{}\ index\ x * oddpart\ x)$
by (*auto simp add*: *sum-distrib-right*)
also have ... = $(\sum x \mid p\ x = 1 \wedge oddpart\ x = i.\ x)$
by (*simp only*: *index-oddpart-decomposition*[*symmetric*])
also have ... ≤ $(\sum x \mid p\ x = 1.\ x)$
using *set-eq* **by** (*intro sum-mono2*) *auto*
also have ... = $(\sum x{\le}n.\ p\ x * x)$
using *distinct* **by** (*subst set-eq*) (*force intro*!: *sum.mono-neutral-cong-left*)
also have ... = *n* **using** *sum* **.**
finally have $(\sum j \mid p\ (2\ \hat{}\ j * i) = 1.\ 2\ \hat{}\ j) * i \le n$ **.**
}
from *this* **have** *less-n*: $\bigwedge i.\ odd\text{-}of\text{-}distinct\ p\ i * i \le n$
unfolding *odd-of-distinct-def* **by** *auto*
have *odd-of-distinct p partitions n*
proof (*rule partitionsI*)
fix *i* **assume** *odd-of-distinct p i* ≠ *0*
from *this assms* **show** *1* ≤ *i* ∧ *i* ≤ *n*
by (*elim CollectE conjE partitionsE odd-of-distinct*) *auto*
next
have $(\sum i{\le}n.\ odd\text{-}of\text{-}distinct\ p\ i * i) = (\sum i{\le}n.\ distinct\text{-}of\text{-}odd\ (odd\text{-}of\text{-}distinct\ p)\ i * i)$
using *assms less-n* **by** (*subst sum-distinct-of-odd*) (*auto elim*!: *partitionsE odd-of-distinct simp only*: *odd*)
also have ... = $(\sum i{\le}n.\ p\ i * i)$ **using** *assms*
by (*auto elim*!: *partitionsE simp only*:) (*subst distinct-odd*, *auto*)
also with *assms* **have** ... = *n* **by** (*auto elim*: *partitionsE*)
finally show $(\sum i{\le}n.\ odd\text{-}of\text{-}distinct\ p\ i * i) = n$ **.**
qed
moreover have $\forall i.\ odd\text{-}of\text{-}distinct\ p\ i \neq 0 \longrightarrow odd\ i$
by (*intro allI impI odd*)
ultimately show *odd-of-distinct p partitions n* ∧ $(\forall i.\ odd\text{-}of\text{-}distinct\ p\ i \neq 0 \longrightarrow odd\ i)$ **by** *simp*
qed

4.5 Euler's Partition Theorem

theorem *Euler-partition-theorem*:
card $\{p.\ p\ partitions\ n \wedge (\forall i.\ p\ i \le 1)\} = card\ \{p.\ p\ partitions\ n \wedge (\forall i.\ p\ i \neq 0 \longrightarrow odd\ i)\}$
(**is** *card ?distinct-partitions* = *card ?odd-partitions*)
proof (*rule card-bij-eq*)
from *odd-of-distinct-in-odd-partitions* **show**
odd-of-distinct ` ?distinct-partitions ⊆ *?odd-partitions* **by** *auto*

moreover from *distinct-of-odd-in-distinct-partitions* **show**
distinct-of-odd ‘ ?odd-partitions ⊆ *?distinct-partitions* **by** *auto*
moreover have ∀ *p*∈*?distinct-partitions*. *distinct-of-odd* (*odd-of-distinct p*) = *p*
by *auto* (*subst distinct-odd*; *auto simp add*: *partitions-def*)
moreover have ∀ *p*∈*?odd-partitions*. *odd-of-distinct* (*distinct-of-odd p*) = *p*
by *auto* (*subst odd-distinct*; *auto simp add*: *partitions-def*)
ultimately show *inj-on odd-of-distinct ?distinct-partitions*
inj-on distinct-of-odd ?odd-partitions
by (*intro bij-betw-imp-inj-on bij-betw-byWitness*; *auto*)+
show *finite ?distinct-partitions finite ?odd-partitions*
by (*simp add*: *finite-partitions*)+
qed

end

References

[1] J. Harrison. Euler's partition theorem and other elementary partition theorems. https://github.com/jrh13/hol-light/blob/master/100/euler.ml.

[2] G. Musiker. Course 18.312: Algebraic combinatorics, 2009. http://ocw.mit.edu/courses/mathematics/18-312-algebraic-combinatorics-spring-2009/readings-and-lecture-notes/MIT18_312S09_lec10_Patitio.pdf.