Formalization of Countable Multisets

Mathias Schack Rabing Dmitriy Traytel

June 14, 2026

Abstract

We define *countable multisets* as a generalization of finite multisets. Countable multisets are equivalently represented as functions from elements to extended natural numbers ($\mathbb{N} \cup \{\infty\}$) returning non-zero values for countably many elements, or as quotients of lazy lists modulo infinitary permutations. We register countable multisets as a bounded natural functor (BNF), enabling nested (co)recursion through them in (co)datatypes. We further define the subtype of *countably infinite multisets* and register it as a BNF, too.

Contents

1 Infinite Sums over Extended Natural Numbers

This is a theory of infinite sums of extended natural numbers defined as limits of finite sums. The goal is to make reasoning about these infinite sums almost as easy as that about finite sums.

theory *Infinite-Sums-Enat*
imports *HOL−Library.Countable-Set HOL−Library.Extended-Nat*
begin

1.1 Preliminaries

lemma *enat-pm-iff*:
$\bigwedge a\ b\ c.\ b \leq c \Longrightarrow (a::enat) + b \leq c \longleftrightarrow a \leq c - b$
$\bigwedge a\ b\ c.\ a \leq c \Longrightarrow (a::enat) + b \leq c \longleftrightarrow b \leq c - a$
$\bigwedge a\ b\ c.\ a \leq b \Longrightarrow c \leq b \Longrightarrow (a::enat) \leq b - c \longleftrightarrow c \leq b - a$
by (*smt* (*verit*) *add.commute add-diff-cancel-enat add-left-mono idiff-infinity le-iff-add nle-le plus-eq-infty-iff-enat*)+

lemma *disjoint-finite-aux*:
$\forall i{\in}I.\ \forall j{\in}I.\ i \neq j \longrightarrow A\ i \cap A\ j = \{\} \Longrightarrow B \subseteq \bigcup\ (A\ `\ I) \Longrightarrow$ *finite* $B \Longrightarrow$ *finite* $\{i \in I.\ B \cap A\ i \neq \{\}\}$
proof −
assume *disj*: $\forall i{\in}I.\ \forall j{\in}I.\ i \neq j \longrightarrow A\ i \cap A\ j = \{\}$
and *hB*: $B \subseteq \bigcup\ (A\ `\ I)$
and *hfin*: *finite* B
show *finite* $\{i \in I.\ B \cap A\ i \neq \{\}\}$
proof (*rule inj-on-finite*[*of* $\lambda i.$ *SOME* $b.\ b \in B \cap A\ i$ - B])
show *inj-on* $(\lambda i.$ *SOME* $b.\ b \in B \cap A\ i)\ \{i \in I.\ B \cap A\ i \neq \{\}\}$
unfolding *inj-on-def*
proof (*intro ballI impI*)
fix $x\ y$
assume *xI*: $x \in \{i \in I.\ B \cap A\ i \neq \{\}\}$ **and** *yI*: $y \in \{i \in I.\ B \cap A\ i \neq \{\}\}$
assume *eq*: (*SOME* $b.\ b \in B \cap A\ x$) = (*SOME* $b.\ b \in B \cap A\ y$)
have *hx*: (*SOME* $b.\ b \in B \cap A\ x$) $\in B \cap A\ x$
proof −
from *xI* **have** $B \cap A\ x \neq \{\}$ **by** *simp*
then obtain c **where** $c \in B \cap A\ x$ **by** *blast*
then show *?thesis* **by** (*rule someI*[*of* $\lambda b.\ b \in B \cap A\ x$])
qed
have *hy*: (*SOME* $b.\ b \in B \cap A\ y$) $\in B \cap A\ y$
proof −
from *yI* **have** $B \cap A\ y \neq \{\}$ **by** *simp*
then obtain c **where** $c \in B \cap A\ y$ **by** *blast*
then show *?thesis* **by** (*rule someI*[*of* $\lambda b.\ b \in B \cap A\ y$])
qed
show $x = y$
proof (*rule ccontr*)
assume $x \neq y$

with *xI yI disj* **have** $A\ x \cap A\ y = \{\}$ **by** *auto*
with *hx hy eq* **show** *False* **by** *auto*
qed
qed
show $(\lambda i.\ SOME\ b.\ b \in B \cap A\ i)\ `\ \{i \in I.\ B \cap A\ i \neq \{\}\} \subseteq B$
proof (*intro subsetI*)
fix *d* **assume** $d \in (\lambda i.\ SOME\ b.\ b \in B \cap A\ i)\ `\ \{i \in I.\ B \cap A\ i \neq \{\}\}$
then obtain *i* **where** *iI*: $i \in \{i \in I.\ B \cap A\ i \neq \{\}\}$ **and** *deq*: $d = (SOME\ b.\ b \in B \cap A\ i)$ **by** *blast*
from *iI* **have** $B \cap A\ i \neq \{\}$ **by** *simp*
then obtain *c* **where** $c \in B \cap A\ i$ **by** *blast*
then have $(SOME\ b.\ b \in B \cap A\ i) \in B \cap A\ i$ **by** (*rule someI*[*of* $\lambda b.\ b \in B \cap A\ i$])
with *deq* **show** $d \in B$ **by** *auto*
qed
show *finite B* **using** *hfin* **.**
qed
qed

lemma *incl-UNION-aux*: $B \subseteq \bigcup\ (A\ `\ I) \Longrightarrow B = \bigcup\ ((\lambda i.\ (B \cap A\ i))\ `\ \{i \in I.\ B \cap A\ i \neq \{\}\})$
by *fastforce*

lemma *incl-UNION-aux2*: $B \subseteq \bigcup\ (A\ `\ I) \longleftrightarrow B = \bigcup\ ((\lambda i.\ (B \cap A\ i))\ `\ I)$
by *fastforce*

lemma *sum-singl*[*simp*]: $sum\ f\ \{a\} = f\ a$
by *simp*

lemma *sum-two*[*simp*]: $a1 \neq a2 \Longrightarrow sum\ f\ \{a1,a2\} = f\ a1 + f\ a2$
by *simp*

lemma *sum-three*[*simp*]: $a1 \neq a2 \Longrightarrow a1 \neq a3 \Longrightarrow a2 \neq a3 \Longrightarrow$
$sum\ f\ \{a1,a2,a3\} = f\ a1 + f\ a2 + f\ a3$
by (*simp add*: *add.assoc*)

lemma *Sup-leq*:
$A \neq \{\} \Longrightarrow \forall\, a{\in}A.\ \exists\, b{\in}B.\ (a{::}enat) \leq b \Longrightarrow Sup\ A \leq Sup\ B$
by (*simp add*: *cSup-mono*)

lemma *Sup-image-leq*:
$A \neq \{\} \Longrightarrow\ \forall\, a{\in}A.\ \exists\, b{\in}B.\ (f\ a{::}enat) \leq g\ b \Longrightarrow$
$Sup\ (f\ `\ A) \leq Sup\ (g\ `\ B)$
by (*rule Sup-leq*) *auto*

lemma *Sup-cong*:
assumes $A \neq \{\} \vee B \neq \{\}\ \forall\, a{\in}A.\ \exists\, b{\in}B.\ (a{::}enat) \leq b\ \forall\, b{\in}B.\ \exists\, a{\in}A.\ (b{::}enat)$

$\leq a$
shows *Sup A = Sup B*
proof−
have $A \neq \{\} \wedge B \neq \{\}$
using *assms* **unfolding** *bdd-above-def* **using** *order.trans* **by** *blast+*
thus *?thesis* **using** *assms*
by (*meson Sup-mono order-antisym*)
qed

lemma *Sup-image-cong*:
$A \neq \{\} \vee B \neq \{\} \Longrightarrow \forall a{\in}A.\ \exists b{\in}B.\ (f\ a{::}enat) \leq g\ b \Longrightarrow \forall b{\in}B.\ \exists a{\in}A.\ (g\ b{::}enat) \leq f\ a \Longrightarrow$
Sup (*f* ‘ *A*) = *Sup* (*g* ‘ *B*)
by (*rule Sup-cong*) *auto*

lemma *Sup-congL*:
$A \neq \{\} \Longrightarrow \forall a{\in}A.\ \exists b{\in}B.\ (a{::}enat) \leq b \Longrightarrow \forall b{\in}B.\ b \leq Sup\ A \Longrightarrow Sup\ A = Sup\ B$
by (*metis Collect-empty-eq Collect-mem-eq Sup-leq cSup-least dual-order.antisym*)

lemma *Sup-image-congL*:
$A \neq \{\} \Longrightarrow \forall a{\in}A.\ \exists b{\in}B.\ (f\ a{::}enat) \leq g\ b \Longrightarrow \forall b{\in}B.\ g\ b \leq Sup\ (f\ `\ A) \Longrightarrow$
Sup (*f* ‘ *A*) = *Sup* (*g* ‘ *B*)
by (*rule Sup-congL*) *auto*

lemma *Sup-congR*:
$B \neq \{\} \Longrightarrow \forall a{\in}A.\ a \leq Sup\ B \Longrightarrow \forall b{\in}B.\ \exists a{\in}A.\ (b{::}enat) \leq a \Longrightarrow Sup\ A = Sup\ B$
by (*metis Collect-empty-eq Collect-mem-eq Sup-leq cSup-least dual-order.antisym*)

lemma *Sup-image-congR*:
$B \neq \{\} \Longrightarrow \forall a{\in}A.\ f\ a \leq Sup\ (g\ `\ B) \Longrightarrow \forall b{\in}B.\ \exists a{\in}A.\ (g\ b{::}enat) \leq f\ a \Longrightarrow$
Sup (*f* ‘ *A*) = *Sup* (*g* ‘ *B*)
by (*rule Sup-congR*) *auto*

lemma *Sup-eq-0-iff*: *Sup* (*A* :: *enat set*) = *0* ⟷ (∀ *a*∈*A*. *a* = *0*)
using *Sup-bot-conv*(*1*)[*of A*] **unfolding** *bot-enat-def* **by** *auto*

lemma *plus-Sup-commute*:
assumes *f1*: {*f1 b1* | *b1*. *φ1 b1*} ≠ {} **and**
f2: {*f2 b2* | *b2*. *φ2 b2*} ≠ {}
shows
Sup {(*f1 b1*::*enat*) | *b1* . *φ1 b1*} + *Sup* {*f2 b2* | *b2* . *φ2 b2*} =
Sup {*f1 b1* + *f2 b2* | *b1 b2*. *φ1 b1* ∧ *φ2 b2*} (**is** *?L1* + *?L2* = *?R*)
proof−
have *f*:
{*f1 b1* + *f2 b2* | *b1 b2*. *φ1 b1* ∧ *φ2 b2*} ≠ {}
using *f1 f2* **by** *auto*

```
  show ?thesis proof (rule order.antisym)
    show ?L1 + ?L2 ≤ ?R
    proof −
      obtain b1-wit where hb1-wit: φ1 b1-wit using f1 by auto
      have hL2-le-R: ?L2 ≤ ?R
      proof (intro cSup-le-iff[OF f2 bdd-above-top, THEN iffD2] ballI)
        fix x assume x ∈ {f2 b2 | b2. φ2 b2}
        then obtain b2 where hb: x = f2 b2 φ2 b2 by auto
        have mem: f1 b1-wit + f2 b2 ∈ {f1 b1 + f2 b2 | b1 b2. φ1 b1 ∧ φ2 b2}
          using hb1-wit hb(2) by blast
        have f2 b2 ≤ f1 b1-wit + f2 b2 by (simp add: le-add2)
        also have ... ≤ ?R using cSup-upper[OF mem bdd-above-top] by simp
        finally show x ≤ ?R using hb(1) by simp
      qed
      have hL1-le-R: ⋀b1. φ1 b1 ⟹ f1 b1 ≤ ?R
      proof −
        fix b1 assume hφ1: φ1 b1
        obtain b2 where hb2: φ2 b2 using f2 by auto
        have mem: f1 b1 + f2 b2 ∈ {f1 b1 + f2 b2 | b1 b2. φ1 b1 ∧ φ2 b2}
          using hφ1 hb2 by blast
        have f1 b1 ≤ f1 b1 + f2 b2 by (simp add: le-add1)
        also have ... ≤ ?R using cSup-upper[OF mem bdd-above-top] by simp
        finally show f1 b1 ≤ ?R .
      qed
      have hL1-le: ?L1 ≤ ?R − ?L2
      proof (intro cSup-le-iff[OF f1 bdd-above-top, THEN iffD2] ballI)
        fix x assume x ∈ {f1 b1 | b1. φ1 b1}
        then obtain b1 where hb: x = f1 b1 φ1 b1 by auto
        have hfb1-le-R: f1 b1 ≤ ?R using hL1-le-R hb(2) by blast
        have hL2-le-Rm: ?L2 ≤ ?R − f1 b1
        proof (intro cSup-le-iff[OF f2 bdd-above-top, THEN iffD2] ballI)
          fix y assume y ∈ {f2 b2 | b2. φ2 b2}
          then obtain b2 where hy: y = f2 b2 φ2 b2 by auto
          have mem: f1 b1 + f2 b2 ∈ {f1 b1 + f2 b2 | b1 b2. φ1 b1 ∧ φ2 b2}
            using hb(2) hy(2) by blast
          have hle: f1 b1 + f2 b2 ≤ ?R using cSup-upper[OF mem bdd-above-top]
by simp
          show y ≤ ?R − f1 b1
            using enat-pm-iff(2)[OF hfb1-le-R] hle hy(1) by simp
        qed
        show x ≤ ?R − ?L2
          using enat-pm-iff(3)[OF hfb1-le-R hL2-le-R] hL2-le-Rm hb(1) by simp
      qed
      show ?L1 + ?L2 ≤ ?R
        using enat-pm-iff(1)[OF hL2-le-R] hL1-le by simp
    qed
  next
    show ?R ≤ ?L1 + ?L2
    proof (intro cSup-le-iff[OF f bdd-above-top, THEN iffD2] ballI)
```

```
    fix x assume x ∈ {f1 b1 + f2 b2 | b1 b2. φ1 b1 ∧ φ2 b2}
    then obtain b1 b2 where hx: x = f1 b1 + f2 b2 φ1 b1 φ2 b2 by auto
    have mem1: f1 b1 ∈ {f1 b1 | b1. φ1 b1} using hx(2) by blast
    have mem2: f2 b2 ∈ {f2 b2 | b2. φ2 b2} using hx(3) by blast
    have h1: f1 b1 ≤ ?L1 using cSup-upper[OF mem1 bdd-above-top] by simp
    have h2: f2 b2 ≤ ?L2 using cSup-upper[OF mem2 bdd-above-top] by simp
    show x ≤ ?L1 + ?L2 using h1 h2 hx(1) by (simp add: add-mono)
   qed
  qed
qed

lemma plus-Sup-commute':
  assumes f1: A1 ≠ {} and f2: A2 ≠ {}
  shows Sup A1 + Sup A2 = Sup {(a1::enat) + a2 | a1 a2. a1 ∈ A1 ∧ a2 ∈
A2}
  using plus-Sup-commute[of id λa1. a1 ∈ A1 id λa2. a2 ∈ A2] assms
  by auto

lemma plus-SupR: A ≠ {} ⟹ Sup A + (b::enat) = Sup {a + b | a. a ∈ A}
proof −
  assume hA: A ≠ {}
  have Sup A + b = Sup A + Sup {b} by simp
  also have ... = Sup {a1 + a2 | a1 a2. a1 ∈ A ∧ a2 ∈ {b}}
    using plus-Sup-commute'[of A {b}] hA by auto
  also have ... = Sup {a + b | a. a ∈ A} by auto
  finally show ?thesis .
qed

lemma plus-SupL: A ≠ {} ⟹ (b::enat) + Sup A = Sup {b + a | a. a ∈ A}
proof −
  assume hA: A ≠ {}
  have b + Sup A = Sup {b} + Sup A by simp
  also have ... = Sup {a1 + a2 | a1 a2. a1 ∈ {b} ∧ a2 ∈ A}
    using plus-Sup-commute'[of {b} A] hA by auto
  also have ... = Sup {b + a | a. a ∈ A} by auto
  finally show ?thesis .
qed



lemma sum-mono3:
  finite B ⟹ A ⊆ B ⟹ (⋀a. a ∈ A ⟹ (f a::enat) ≤ g a) ⟹
 sum f A ≤ sum g B
  by (metis order-trans sum-mono sum-mono2 zero-le)

lemma sum-Sup-commute:
  fixes h :: 'a ⇒ enat
  assumes finite J and ∀ i∈J. {h b | b. φ i b} ≠ {}
  shows sum (λi. Sup {h b | b. φ i b}) J =
```

```
      Sup {sum (λi. h (b i)) J | b . ∀i∈J. φ i (b i)}
  using assms proof (induction J)
  case empty then show ?case by simp
next
  case (insert j J)
  have note1: {∑i∈J. h (b i) | b. ∀i∈J. φ i (b i)} ≠ {}
    if hyp: ∀i∈J. {h b | b. φ i b} ≠ {}
  proof −
    from hyp have ∀i∈J. ∃b. φ i b by auto
    then have ∃b. ∀i∈J. φ i (b i)
      by (intro exI[of - λi. SOME b. φ i b]) (simp add: some-eq-ex)
    then show ?thesis by blast
  qed
  have hJ-ne: ∀i∈J. {h b | b. φ i b} ≠ {} using insert.prems by auto
  have hIH: sum (λi. Sup {h b | b. φ i b}) J = Sup {sum (λi. h (b i)) J | b. ∀i∈J.
φ i (b i)}
    using insert.IH[OF hJ-ne] by simp
  have hpsc: Sup {h b | b. φ j b} + Sup {sum (λi. h (b i)) J | b. ∀i∈J. φ i (b i)}
=
            Sup {h b1 + sum (λi. h (b2 i)) J | b1 b2. φ j b1 ∧ (∀i∈J. φ i (b2 i))}
    using plus-Sup-commute[of h φ j λb. sum (λi. h (b i)) J λb2. ∀i∈J. φ i (b2
i)]
          insert.prems note1[OF hJ-ne]
    by auto
  have hset: {h b1 + sum (λi. h (b2 i)) J | b1 b2. φ j b1 ∧ (∀i∈J. φ i (b2 i))} =
            {sum (λi. h (b i)) (insert j J) | b. ∀i∈insert j J. φ i (b i)}
    using insert.hyps
  proof (intro set-eqI iffI)
    fix x
    assume x ∈ {h b1 + sum (λi. h (b2 i)) J | b1 b2. φ j b1 ∧ (∀i∈J. φ i (b2
i))}
    then obtain b1 b2 where hx: x = h b1 + sum (λi. h (b2 i)) J φ j b1 ∀i∈J.
φ i (b2 i)
      by auto
    let ?b = λj'. if j' = j then b1 else b2 j'
    have hb-J: ∀i∈J. ?b i = b2 i using insert.hyps by simp
    have hsum-eq: sum (λi. h (?b i)) J = sum (λi. h (b2 i)) J
      by (rule sum.cong) (auto simp: hb-J)
    have x = sum (λi. h (?b i)) (insert j J)
      using hx insert.hyps by (simp add: hsum-eq)
    moreover have ∀i∈insert j J. φ i (?b i)
      using hx insert.hyps by (auto simp: insertE)
    ultimately show x ∈ {sum (λi. h (b i)) (insert j J) | b. ∀i∈insert j J. φ i (b
i)}
      by auto
  next
    fix x
    assume x ∈ {sum (λi. h (b i)) (insert j J) | b. ∀i∈insert j J. φ i (b i)}
    then obtain b where hx: x = sum (λi. h (b i)) (insert j J) ∀i∈insert j J. φ
```

i (*b i*)
by *auto*
have $x = h\ (b\ j) + sum\ (\lambda i.\ h\ (b\ i))\ J$
using *insert.hyps hx*(*1*) **by** *simp*
then show $x \in \{h\ b1 + sum\ (\lambda i.\ h\ (b2\ i))\ J \mid b1\ b2.\ \varphi\ j\ b1 \wedge (\forall\, i{\in}J.\ \varphi\ i\ (b2\ i))\}$
using *hx*(*2*) **by** *auto*
qed
show *?case*
using *insert.hyps*
by (*simp add*: *hIH hpsc hset*)
qed

1.2 Infinite Summation

definition $isum :: ('a \Rightarrow enat) \Rightarrow 'a\ set \Rightarrow enat$ **where**
$isum\ f\ A \equiv Sup\ (sum\ f\ `\{B \mid B.\ B \subseteq A \wedge finite\ B\})$

lemma *isum-subset-mono*: $A \subseteq B \Longrightarrow isum\ f\ A \leq isum\ f\ B$
unfolding *isum-def image-def*
by(*auto intro*: *Sup-subset-mono*)

lemma *isum-eq-sum*:
$finite\ A \Longrightarrow isum\ f\ A = sum\ f\ A$
proof −
assume *hA*: *finite A*
show $isum\ f\ A = sum\ f\ A$
unfolding *isum-def*
proof (*rule cSup-eq-maximum*)
show $sum\ f\ A \in sum\ f\ `\{B \mid B.\ B \subseteq A \wedge finite\ B\}$ **using** *hA* **by** *blast*
show $\bigwedge x.\ x \in sum\ f\ `\{B \mid B.\ B \subseteq A \wedge finite\ B\} \Longrightarrow x \leq sum\ f\ A$
using *hA* **by** (*auto intro*: *sum-mono2*)
qed
qed

lemma *isum-cong*:
assumes $A = B$ **and** $\bigwedge x.\ x \in B \Longrightarrow g\ x = h\ x$
shows $isum\ g\ A = isum\ h\ B$
using *assms* **unfolding** *isum-def*
by (*auto intro*!: *SUP-cong comm-monoid-add-class.sum.cong*)

lemma *isum-mono*:
assumes $\bigwedge x.\ x \in A \Longrightarrow g\ x \leq h\ x$
shows *isum g A* $\leq$ *isum h A*
unfolding *isum-def*
proof (*intro cSup-mono*)
show *sum g* ‘ $\{B \mid B.\ B \subseteq A \wedge$ *finite* $B\} \neq \{\}$ **by** *auto*
show *bdd-above* (*sum h* ‘ $\{B \mid B.\ B \subseteq A \wedge$ *finite* $B\}$)
unfolding *bdd-above-def* **using** *bdd-above.unfold* **by** *blast*
fix *r* **assume** *r* $\in$ *sum g* ‘ $\{B \mid B.\ B \subseteq A \wedge$ *finite* $B\}$
then obtain *B* **where** *hB*: $B \subseteq A$ *finite B* **and** *hr*: *r* = *sum g B* **by** *blast*
show $\exists\, b \in$ *sum h* ‘ $\{B \mid B.\ B \subseteq A \wedge$ *finite* $B\}.\ r \leq b$
proof (*intro bexI*[*of* - *sum h B*])
show *r* $\leq$ *sum h B* **using** *hr assms hB* **by** (*auto intro*: *sum-mono*)
show *sum h B* $\in$ *sum h* ‘ $\{B \mid B.\ B \subseteq A \wedge$ *finite* $B\}$ **using** *hB* **by** *blast*
qed
qed

lemma *isum-mono′*:
assumes $A \subseteq B$
shows *isum g A* $\leq$ *isum g B*
using *assms* **unfolding** *isum-def*
by (*intro cSup-subset-mono*) (*auto intro*!: *exI*[*of* - ∞])

lemma *isum-empty*[*simp*]: *isum g* $\{\}$ = *0*
unfolding *isum-def* **by** *auto*

lemma *isum-const-zero*[*simp*]: *isum* ($\lambda x.\ 0$) *A* = *0*
unfolding *isum-def*
by *simp* (*metis cSUP-const empty-iff empty-subsetI finite.emptyI mem-Collect-eq*)

lemma *isum-const-zero′*: $\forall\, x \in A.\ g\ x = 0 \Longrightarrow$ *isum g A* = *0*
by (*simp add*: *isum-cong*)

lemma *isum-eq-0-iff*: *isum f A* = *0* $\longleftrightarrow$ ($\forall\, a \in A.\ f\ a = 0$)
unfolding *isum-def* **by** (*subst Sup-eq-0-iff*) *auto*

lemma *isum-reindex*: *inj-on h A* $\Longrightarrow$ *isum g* (*h* ‘ *A*) = *isum* (*g* $\circ$ *h*) *A*
unfolding *isum-def*
proof (*intro arg-cong*[*of* - - *Sup*])
assume *inj-on h A*

then have *sum* $(g \circ h)$ B = *sum* g (*image* h B) **if** $B \subseteq A$ **for** B
by (*simp add*: *inj-on-subset sum.reindex that*)
then show *sum* g ‘ $\{B$ |B. $B \subseteq h$ ‘ $A \wedge$ *finite* $B\}$ =
sum $(g \circ h)$ ‘ $\{B$ |B. $B \subseteq A \wedge$ *finite* $B\}$
by (*auto simp*: *image-iff dest*: *finite-subset-image*)
qed

lemma *isum-reindex-cong*: *inj-on* l $B \Longrightarrow A = l$ ‘ $B \Longrightarrow$
$(\bigwedge x.\ x \in B \Longrightarrow g\ (l\ x) = h\ x) \Longrightarrow$ *isum* g A = *isum* h B
by (*metis isum-cong comp-def isum-reindex*)

lemma *isum-reindex-cong′*:
assumes $(\bigwedge x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow x \neq y \Longrightarrow h\ x = h\ y \Longrightarrow g\ (h\ x) = 0)$
shows *isum* g $(h$ ‘ $A)$ = *isum* $(g \circ h)$ A
unfolding *isum-def*
proof (*safe intro*!: *arg-cong*[*of* - - *Sup*])
fix B
assume $B \subseteq h$ ‘ A *finite* B
with *assms* **show** *sum* g $B \in$ *sum* $(g \circ h)$ ‘ $\{B$ |B. $B \subseteq A \wedge$ *finite* $B\}$
by (*smt* (*verit*) *finite-subset-image image-iff in-mono mem-Collect-eq*
sum.reindex-nontrivial)
next
fix B
assume $B \subseteq A$ *finite* B
with *assms* **show** *sum* $(g \circ h)$ $B \in$ *sum* g ‘ $\{B$ |B. $B \subseteq h$ ‘ $A \wedge$ *finite* $B\}$
by (*smt* (*verit*, *ccfv-SIG*) *finite-imageI image-iff mem-Collect-eq subset-iff*
sum.reindex-nontrivial)
qed

lemma *isum-zeros-cong*:
assumes $\bigwedge i.\ i \in T - S \Longrightarrow h\ i = 0$ **and** $\bigwedge i.\ i \in S - T \Longrightarrow g\ i = 0$
and $\bigwedge x.\ x \in S \cap T \Longrightarrow g\ x = h\ x$
shows *isum* g S = *isum* h T
unfolding *isum-def*
proof (*rule Sup-image-cong*)
show $\{B$ |B. $B \subseteq S \wedge$ *finite* $B\} \neq \{\} \vee \{B$ |B. $B \subseteq T \wedge$ *finite* $B\} \neq \{\}$
by *auto*
next
show $\forall a \in \{B$ |B. $B \subseteq S \wedge$ *finite* $B\}$. $\exists b \in \{B$ |B. $B \subseteq T \wedge$ *finite* $B\}$. *sum* g
$a \leq$ *sum* h b
proof (*intro ballI*)
fix a **assume** $a \in \{B$ |B. $B \subseteq S \wedge$ *finite* $B\}$
then have *hA*: $a \subseteq S$ *finite* a **by** *blast*+
show $\exists b \in \{B$ |B. $B \subseteq T \wedge$ *finite* $B\}$. *sum* g $a \leq$ *sum* h b
proof (*intro bexI*[*of* - $a \cap T$])
show *sum* g $a \leq$ *sum* h $(a \cap T)$

by (*rule order-eq-refl*, *rule sum.mono-neutral-cong*) (*use assms hA* **in** *auto*)
show $a \cap T \in \{B \,|B.\ B \subseteq T \wedge \mathit{finite}\ B\}$
using *hA* **by** *blast*
qed
qed
next
show $\forall\, b \in \{B \,|B.\ B \subseteq T \wedge \mathit{finite}\ B\}.\ \exists\, a \in \{B \,|B.\ B \subseteq S \wedge \mathit{finite}\ B\}.\ \mathit{sum}\ h\ b \leq \mathit{sum}\ g\ a$
proof (*intro ballI*)
fix *b* **assume** $b \in \{B \,|B.\ B \subseteq T \wedge \mathit{finite}\ B\}$
then have *hB*: $b \subseteq T\ \mathit{finite}\ b$ **by** *blast+*
show $\exists\, a \in \{B \,|B.\ B \subseteq S \wedge \mathit{finite}\ B\}.\ \mathit{sum}\ h\ b \leq \mathit{sum}\ g\ a$
proof (*intro bexI*[*of* - $b \cap S$])
show $\mathit{sum}\ h\ b \leq \mathit{sum}\ g\ (b \cap S)$
by (*rule order-eq-refl*, *rule sum.mono-neutral-cong*) (*use assms hB* **in** *auto*)
show $b \cap S \in \{B \,|B.\ B \subseteq S \wedge \mathit{finite}\ B\}$
using *hB* **by** *blast*
qed
qed
qed

lemma *isum-zeros-congL*:
$S \subseteq T \Longrightarrow \forall\, i{\in}T - S.\ g\ i = 0 \Longrightarrow \mathit{isum}\ g\ S = \mathit{isum}\ g\ T$
by (*metis Diff-eq-empty-iff emptyE isum-zeros-cong*)

lemma *isum-zeros-congR*:
$S \subseteq T \Longrightarrow \forall\, i{\in}T - S.\ g\ i = 0 \Longrightarrow \mathit{isum}\ g\ T = \mathit{isum}\ g\ S$
by (*simp add*: *isum-zeros-congL*)

lemma *isum-singl*[*simp*]: $\mathit{isum}\ f\ \{a\} = f\ a$
by (*simp add*: *isum-eq-sum*)

lemma *isum-two*[*simp*]: $a1 \neq a2 \Longrightarrow \mathit{isum}\ f\ \{a1,a2\} = f\ a1 + f\ a2$
by (*simp add*: *isum-eq-sum*)

lemma *isum-three*[*simp*]: $a1 \neq a2 \Longrightarrow a1 \neq a3 \Longrightarrow a2 \neq a3 \Longrightarrow$
$\mathit{isum}\ f\ \{a1,a2,a3\} = f\ a1 + f\ a2 + f\ a3$
by (*simp add*: *isum-eq-sum*)

lemma *in-le-isum*: $a \in A \Longrightarrow f\ a \leq \mathit{isum}\ f\ A$
by (*metis isum-mono′ isum-singl singletonD subsetI*)

lemma *isum-eq-singl*:
assumes *fx*: $f\ a = x$ **and** *f*: $\forall\, a'.\ a' \neq a \longrightarrow f\ a' = 0$ **and** *a*: $a \in A$
shows $\mathit{isum}\ f\ A = x$
proof −

```
  have isum f A = isum f {a}
    by (rule isum-zeros-cong) (use assms in auto)
  then show ?thesis by (simp add: fx)
qed

lemma isum-le-singl:
  assumes fx: f a ≤ x and f: ∀ a'. a' ≠ a ⟶ f a' = 0 and a: a ∈ A
  shows isum f A ≤ x
  by (metis a f fx isum-eq-singl)

lemma isum-insert[simp]: a ∉ A ⟹ isum f (insert a A) = isum f A + f a
proof −
  assume ha: a ∉ A
  have hA: sum f ‘ {B |B. B ⊆ A ∧ finite B} ≠ {} by auto
  show ?thesis
    unfolding isum-def
  proof (subst plus-SupR[OF hA], rule Sup-cong)
    show sum f ‘ {B |B. B ⊆ insert a A ∧ finite B} ≠ {} ∨
        {x + f a |x. x ∈ sum f ‘ {B |B. B ⊆ A ∧ finite B}} ≠ {} by auto
  next
    show ∀ v ∈ sum f ‘ {B |B. B ⊆ insert a A ∧ finite B}.
        ∃ w ∈ {x + f a |x. x ∈ sum f ‘ {B |B. B ⊆ A ∧ finite B}}. v ≤ w
    proof (intro ballI)
      fix v assume v ∈ sum f ‘ {B |B. B ⊆ insert a A ∧ finite B}
      then obtain B where hB: B ⊆ insert a A finite B and hv: v = sum f B by
blast
      show ∃ w ∈ {x + f a |x. x ∈ sum f ‘ {B |B. B ⊆ A ∧ finite B}}. v ≤ w
      proof (intro bexI[of - sum f (B − {a}) + f a])
        show v ≤ sum f (B − {a}) + f a
        proof (cases a ∈ B)
          case True
          then have sum f B = f a + sum f (B − {a}) using hB by (simp add:
sum.remove)
          then show ?thesis unfolding hv by (simp add: add.commute)
        next
          case False
          then show ?thesis unfolding hv by simp
        qed
        show sum f (B − {a}) + f a ∈ {x + f a |x. x ∈ sum f ‘ {B |B. B ⊆ A ∧
finite B}}
          using hB ha by blast
      qed
    qed
  next
    show ∀ w ∈ {x + f a |x. x ∈ sum f ‘ {B |B. B ⊆ A ∧ finite B}}.
        ∃ v ∈ sum f ‘ {B |B. B ⊆ insert a A ∧ finite B}. w ≤ v
    proof (intro ballI)
      fix w assume w ∈ {x + f a |x. x ∈ sum f ‘ {B |B. B ⊆ A ∧ finite B}}
      then obtain B where hB: B ⊆ A finite B and hw: w = sum f B + f a by
```

```
  blast
      show ∃ v ∈ sum f ` {B |B. B ⊆ insert a A ∧ finite B}. w ≤ v
      proof (intro bexI[of - sum f (insert a B)])
        show w ≤ sum f (insert a B)
          unfolding hw using ha hB by force
        show sum f (insert a B) ∈ sum f ` {B |B. B ⊆ insert a A ∧ finite B}
          using hB by blast
      qed
    qed
  qed
qed
```

```
lemma isum-UNION:
  assumes dsj: ∀ i∈I. ∀ j∈I. i ≠ j ⟶ A i ∩ A j = {}
  shows isum g (⋃ (A ` I)) = isum (λi. isum g (A i)) I
proof −
  have step1: ⋀J. J ⊆ I ⟹ finite J ⟹
      (∑ i∈J. Sup {y. ∃ B⊆A i. finite B ∧ y = sum g B}) ≤
      Sup {y. ∃ B⊆⋃ (A ` I). finite B ∧ y = sum g B}
  proof −
    fix J assume J1: J ⊆ I finite J
    have hsets-ne: ∀ i∈J. {sum g B | B. B ⊆ A i ∧ finite B} ≠ {}
      by auto
    have step1a:
      sum (λi. Sup {sum g B | B. B ⊆ A i ∧ finite B}) J =
       Sup {sum (λi. sum g (B i)) J | B. ∀ i∈J. B i ⊆ A i ∧ finite (B i)}
      using sum-Sup-commute[OF J1(2) hsets-ne] by simp
    have step1b:
      Sup {sum (λi. sum g (B i)) J | B. ∀ i∈J. B i ⊆ A i ∧ finite (B i)} =
       Sup {sum g (⋃ (B ` J)) | B. ∀ i∈J. B i ⊆ A i ∧ finite (B i)}
    proof (rule arg-cong[of - - Sup], intro set-eqI iffI)
      fix x
      assume x ∈ {sum (λi. sum g (B i)) J | B. ∀ i∈J. B i ⊆ A i ∧ finite (B i)}
      then obtain B where hB: ∀ i∈J. B i ⊆ A i ∧ finite (B i)
        and hx: x = sum (λi. sum g (B i)) J by auto
      have hdisj: ∀ i∈J. ∀ j∈J. i ≠ j ⟶ B i ∩ B j = {}
      proof (intro ballI impI)
        fix i j assume i ∈ J j ∈ J i ≠ j
        have A i ∩ A j = {} using dsj ‹i ∈ J› ‹j ∈ J› ‹i ≠ j› J1(1) by auto
        moreover have B i ⊆ A i B j ⊆ A j using hB ‹i ∈ J› ‹j ∈ J› by auto
        ultimately show B i ∩ B j = {} by auto
      qed
      have x = sum g (⋃ (B ` J))
        unfolding hx
        by (rule sum.UNION-disjoint[symmetric]) (use J1(2) hB hdisj in auto)
      then show x ∈ {sum g (⋃ (B ` J)) | B. ∀ i∈J. B i ⊆ A i ∧ finite (B i)}
```

```
      using hB by auto
  next
    fix x
    assume x ∈ {sum g (⋃ (B ‘ J)) | B. ∀ i∈J. B i ⊆ A i ∧ finite (B i)}
    then obtain B where hB: ∀ i∈J. B i ⊆ A i ∧ finite (B i)
      and hx: x = sum g (⋃ (B ‘ J)) by auto
    have hdisj: ∀ i∈J. ∀ j∈J. i ≠ j ⟶ B i ∩ B j = {}
    proof (intro ballI impI)
      fix i j assume i ∈ J j ∈ J i ≠ j
      have A i ∩ A j = {} using dsj ‹i ∈ J› ‹j ∈ J› ‹i ≠ j› J1(1) by auto
      moreover have B i ⊆ A i B j ⊆ A j using hB ‹i ∈ J› ‹j ∈ J› by auto
      ultimately show B i ∩ B j = {} by auto
    qed
    have x = sum (λi. sum g (B i)) J
      unfolding hx
      by (rule sum.UNION-disjoint) (use J1(2) hB hdisj in auto)
    then show x ∈ {sum (λi. sum g (B i)) J | B. ∀ i∈J. B i ⊆ A i ∧ finite (B
i)}
      using hB by auto
  qed
  have step1c:
    Sup {sum g (⋃ (B ‘ J)) | B. ∀ i∈J. B i ⊆ A i ∧ finite (B i)} =
     Sup {sum g B | B. B ⊆ ⋃ (A ‘ J) ∧ finite B}
  proof (rule Sup-cong)
    show {sum g (⋃ (B ‘ J)) | B. ∀ i∈J. B i ⊆ A i ∧ finite (B i)} ≠ {} ∨
         {sum g B | B. B ⊆ ⋃ (A ‘ J) ∧ finite B} ≠ {} by auto
  next
    show ∀ a ∈ {sum g (⋃ (B ‘ J)) | B. ∀ i∈J. B i ⊆ A i ∧ finite (B i)}.
          ∃ b ∈ {sum g B | B. B ⊆ ⋃ (A ‘ J) ∧ finite B}. a ≤ b
    proof (intro ballI)
      fix a assume a ∈ {sum g (⋃ (B ‘ J)) | B. ∀ i∈J. B i ⊆ A i ∧ finite (B i)}
      then obtain B where hB: ∀ i∈J. B i ⊆ A i ∧ finite (B i)
        and ha: a = sum g (⋃ (B ‘ J)) by auto
      show ∃ b ∈ {sum g B | B. B ⊆ ⋃ (A ‘ J) ∧ finite B}. a ≤ b
      proof (intro bexI[of - sum g (⋃ (B ‘ J))])
        show sum g (⋃ (B ‘ J)) ∈ {sum g B | B. B ⊆ ⋃ (A ‘ J) ∧ finite B}
          using J1 hB by force
        show a ≤ sum g (⋃ (B ‘ J)) using ha by simp
      qed
    qed
  next
    show ∀ b ∈ {sum g B | B. B ⊆ ⋃ (A ‘ J) ∧ finite B}.
          ∃ a ∈ {sum g (⋃ (B ‘ J)) | B. ∀ i∈J. B i ⊆ A i ∧ finite (B i)}. b ≤ a
    proof (intro ballI)
      fix b assume b ∈ {sum g B | B. B ⊆ ⋃ (A ‘ J) ∧ finite B}
      then obtain C where hCsub: C ⊆ ⋃ (A ‘ J) finite C
        and hb: b = sum g C by auto
      show ∃ a ∈ {sum g (⋃ (B ‘ J)) | B. ∀ i∈J. B i ⊆ A i ∧ finite (B i)}. b ≤ a
      proof (intro bexI[of - sum g (⋃ ((λi. C ∩ A i) ‘ J))])
```

```
        show b ≤ sum g (⋃ ((λi. C ∩ A i) ‘ J))
          using hCsub unfolding incl-UNION-aux2 by (auto simp: hb)
        show sum g (⋃ ((λi. C ∩ A i) ‘ J)) ∈
            {sum g (⋃ (B ‘ J)) | B. ∀ i∈J. B i ⊆ A i ∧ finite (B i)}
          using J1 hCsub by blast
      qed
    qed
  qed
  have step1d:
    Sup {sum g B | B. B ⊆ ⋃ (A ‘ J) ∧ finite B} ≤
     Sup {sum g B | B. B ⊆ ⋃ (A ‘ I) ∧ finite B}
  proof (rule cSup-subset-mono)
    show {sum g B | B. B ⊆ ⋃ (A ‘ J) ∧ finite B} ≠ {} by auto
    show {sum g B | B. B ⊆ ⋃ (A ‘ J) ∧ finite B} ⊆
        {sum g B | B. B ⊆ ⋃ (A ‘ I) ∧ finite B}
      using J1 by (force intro: Union-mono)
  qed simp
  show (∑ i∈J. Sup {y. ∃ B⊆A i. finite B ∧ y = sum g B}) ≤
      Sup {y. ∃ B⊆⋃ (A ‘ I). finite B ∧ y = sum g B}
    using step1a step1b step1c step1d
    by (smt (verit, best) Collect-eqI order.trans order.refl sum-mono)
qed
show ?thesis
  unfolding isum-def
proof (intro Sup-image-congL ballI)
  show {B |B. B ⊆ ⋃ (A ‘ I) ∧ finite B} ≠ {} by auto
next
  fix B assume hB: B ∈ {B |B. B ⊆ ⋃ (A ‘ I) ∧ finite B}
  then have hBsub: B ⊆ ⋃ (A ‘ I) and hBfin: finite B by blast+
  let ?J = {i. i ∈ I ∧ B ∩ A i ≠ {}}
  show ∃ b ∈ {B |B. B ⊆ I ∧ finite B}.
      sum g B ≤ (∑ i∈b. Sup (sum g ‘ {B |B. B ⊆ A i ∧ finite B}))
  proof (intro bexI[of - ?J])
    have sum g B ≤ sum (λi. Sup {y. ∃ x∈{B |B. B ⊆ A i ∧ finite B}. y = sum
g x}) ?J
    proof −
      have hJfin: finite ?J using dsj hBsub hBfin disjoint-finite-aux by blast
      have hJsub: ?J ⊆ I by auto
      have hdecomp: B = ⋃ ((λi. B ∩ A i) ‘ ?J)
        using incl-UNION-aux[OF hBsub] by simp
      have hdisj-J: ∀ i∈?J. ∀ j∈?J. i ≠ j ⟶ (B ∩ A i) ∩ (B ∩ A j) = {}
        using dsj hJsub by auto
      have hfin-J: ∀ i∈?J. finite (B ∩ A i)
        using hBfin by auto
      have hsum-eq: sum g B = sum (λi. sum g (B ∩ A i)) ?J
        by (subst hdecomp, rule comm-monoid-add-class.sum.UNION-disjoint)
          (use hJfin hdisj-J hfin-J in auto)
      show ?thesis
        unfolding hsum-eq
```

```
      proof (rule sum-mono)
        fix i assume i ∈ ?J
        then have B ∩ A i ⊆ A i finite (B ∩ A i) using hBfin by auto
          with hBsub show sum g (B ∩ A i) ≤ Sup {y. ∃x∈{B |B. B ⊆ A i ∧
finite B}. y = sum g x}
            by (intro cSup-upper) auto
      qed
    qed
    then show sum g B ≤ (∑ i | i ∈ I ∧ B ∩ A i ≠ {}. Sup (sum g ' {B |B. B
⊆ A i ∧ finite B}))
      by (rule order-trans) (auto intro!: Sup-mono sum-mono)
    show ?J ∈ {B |B. B ⊆ I ∧ finite B}
      using dsj hBsub hBfin disjoint-finite-aux by force
  qed
 next
  fix J assume hB: J ∈ {B |B. B ⊆ I ∧ finite B}
  then have J1: J ⊆ I finite J by blast+
  show (∑ i∈J. Sup (sum g ' {B |B. B ⊆ A i ∧ finite B}))
      ≤ Sup (sum g ' {B |B. B ⊆ ⋃ (A ' I) ∧ finite B})
    using step1[OF J1] by (auto simp: image-def)
 qed
qed

lemma isum-Un[simp]:
 assumes A1 ∩ A2 = {}
 shows isum f (A1 ∪ A2) = isum f A1 + isum f A2
proof−
 have [simp]: isum (λi. isum f (case i of 0 ⇒ A1 | Suc - ⇒ A2)) {0, Suc 0} =
isum f A1 + isum f A2
   by (subst isum-two) auto
 show ?thesis using assms isum-UNION[of {0,1::nat} λi. case i of 0 ⇒ A1 |-
⇒ A2 f] by auto
qed


lemma isum-Sigma:
 isum (λ(a,b). f a b) (Sigma A Bs) = isum (λa. isum (f a) (Bs a)) A
proof−
 define g where g ≡ λ(a,b). f a b
 define Cs where Cs ≡ λa. {a} × Bs a
 have 1: ⋀a. {a} × Bs a = Pair a ' (Bs a) by auto
 have 0: Sigma A Bs = (⋃ a∈A. Cs a) unfolding Cs-def by auto
 show ?thesis unfolding 0
 proof (subst isum-UNION)
   show ∀ i∈A. ∀ j∈A. i ≠ j ⟶ Cs i ∩ Cs j = {} unfolding Cs-def by auto
 next
   show isum (λi. isum (λ(x, y). f x y) (Cs i)) A = isum (λa. isum (f a) (Bs a))
A
     unfolding Cs-def
```

```
    proof (rule isum-cong)
      show A = A ..
    next
      fix a assume a ∈ A
      show isum (λ(x, y). f x y) ({a} × Bs a) = isum (f a) (Bs a)
      proof −
        have isum (λ(x, y). f x y) ({a} × Bs a) = isum (λ(x, y). f x y) (Pair a ‘
Bs a)
          by (simp add: 1)
        also have ... = isum (f a) (Bs a)
          by (subst isum-reindex-cong′) (auto simp: o-def)
        finally show ?thesis .
      qed
    qed
  qed
qed



lemma isum-Times: isum (λ(a,b). f a b) (A × B) = isum (λa. isum (f a) B) A
  using isum-Sigma .


lemma isum-swap: isum (λa. isum (f a) B) A = isum (λb. isum (λa. f a b) A) B
(is ?L = ?R)
proof−
  have 0: A × B = (λ(a,b). (b,a)) ‘ (B × A) by auto
  have ?L = isum (λ(a,b). f a b) (A × B) using isum-Times ..
 also have ... = isum (λ(b,a). f a b) (B × A) unfolding 0 by (subst isum-reindex-cong′)
(auto simp: o-def intro!: arg-cong2[of - - - - isum])
  also have ... = ?R by (subst isum-Times) auto
  finally show ?thesis .
qed

lemma isum-plus:
  shows isum (λa. f1 a + f2 a) A = isum f1 A + isum f2 A
proof −
  let ?S = sum f1 ‘ {B |B. B ⊆ A ∧ finite B}
  let ?T = sum f2 ‘ {B |B. B ⊆ A ∧ finite B}
  have hS: ?S ≠ {} by auto
  have hT: ?T ≠ {} by auto
  have isum (λa. f1 a + f2 a) A =
    Sup {a1 + a2 | a1 a2. a1 ∈ ?S ∧ a2 ∈ ?T}
  proof (unfold isum-def, rule Sup-cong)
    show sum (λa. f1 a + f2 a) ‘ {B |B. B ⊆ A ∧ finite B} ≠ {} ∨
      {a1 + a2 |a1 a2. a1 ∈ ?S ∧ a2 ∈ ?T} ≠ {} by auto
    show ∀ a ∈ sum (λa. f1 a + f2 a) ‘ {B |B. B ⊆ A ∧ finite B}.
      ∃ b ∈ {a1 + a2 |a1 a2. a1 ∈ ?S ∧ a2 ∈ ?T}. a ≤ b
    proof (intro ballI)
```

fix *a* **assume** *a* ∈ *sum* (λ*a*. *f1 a* + *f2 a*) ‘ {*B* |*B*. *B* ⊆ *A* ∧ *finite B*}
then obtain *B* **where** *hB*: *B* ⊆ *A finite B* **and** *ha*: *a* = *sum* (λ*a*. *f1 a* + *f2 a*) *B* **by** *auto*
show ∃ *b* ∈ {*a1* + *a2* |*a1 a2*. *a1* ∈ *?S* ∧ *a2* ∈ *?T*}. *a* ≤ *b*
using *ha sum.distrib*[*of f1 f2 B*] *hB*
by (*intro bexI*[*of* - *sum f1 B* + *sum f2 B*]) *auto*
qed
show ∀ *b* ∈ {*a1* + *a2* |*a1 a2*. *a1* ∈ *?S* ∧ *a2* ∈ *?T*}.
∃ *a* ∈ *sum* (λ*a*. *f1 a* + *f2 a*) ‘ {*B* |*B*. *B* ⊆ *A* ∧ *finite B*}. *b* ≤ *a*
proof (*intro ballI*)
fix *b* **assume** *b* ∈ {*a1* + *a2* |*a1 a2*. *a1* ∈ *?S* ∧ *a2* ∈ *?T*}
then obtain *B1 B2* **where** *hB1*: *B1* ⊆ *A finite B1* **and** *hB2*: *B2* ⊆ *A finite B2*
and *hb*: *b* = *sum f1 B1* + *sum f2 B2* **by** *auto*
show ∃ *a* ∈ *sum* (λ*a*. *f1 a* + *f2 a*) ‘ {*B* |*B*. *B* ⊆ *A* ∧ *finite B*}. *b* ≤ *a*
proof (*intro bexI*[*of* - *sum* (λ*a*. *f1 a* + *f2 a*) (*B1* ∪ *B2*)])
have *h1*: *sum f1 B1* ≤ *sum f1* (*B1* ∪ *B2*) **using** *hB1 hB2* **by** (*intro sum-mono2*) *auto*
have *h2*: *sum f2 B2* ≤ *sum f2* (*B1* ∪ *B2*) **using** *hB1 hB2* **by** (*intro sum-mono2*) *auto*
show *b* ≤ *sum* (λ*a*. *f1 a* + *f2 a*) (*B1* ∪ *B2*)
using *hb h1 h2 sum.distrib*[*of f1 f2 B1* ∪ *B2*] **by** (*simp add*: *add-mono*)
show *sum* (λ*a*. *f1 a* + *f2 a*) (*B1* ∪ *B2*) ∈ *sum* (λ*a*. *f1 a* + *f2 a*) ‘ {*B* |*B*. *B* ⊆ *A* ∧ *finite B*}
using *hB1 hB2* **by** *auto*
qed
qed
qed
also have ... = *Sup ?S* + *Sup ?T*
by (*subst plus-Sup-commute*′[*symmetric*, *OF hS hT*]) *simp*
finally show *?thesis* **unfolding** *isum-def* **.**
qed

end
theory *Countable-Multiset*
imports
HOL−Library.Countable-Set-Type
HOL−Library.Extended-Nat
Coinductive.Coinductive-List
HOL−Library.BNF-Corec
Infinite-Sums-Enat
begin

2 Miscellaneous (Mostly About Lazy Lists)

lemma *bij-betw-singl-remap*: ‹*bij-betw* π *A B* ⟹ *x* ∈ *A* ⟹ *y* ∈ *B* ⟹
bij-betw (π(*inv-into A* π *y* := π *x*)) (*A* − {*x*}) (*B* − {*y*})›
by (*auto simp*: *bij-betw-def inj-on-def image-iff*)

lemma *ldropWhile-eq-LCons-iff*: ‹*ldropWhile P lxs = LCons x lxs'* ⟷ (¬ *P x* ∧
(∃ *xs*. *lxs = lappend* (*llist-of xs*) (*LCons x lxs'*) ∧ (∀ *x* ∈ *set xs*. *P x*)))›
proof −
have *False*
if ‹*ldropWhile P lxs = LCons x lxs'*› **and** ‹*P x*›
using *that* **by** (*metis lhd-LCons lhd-ldropWhile llist.disc*(*2*) *lnull-ldropWhile*)
moreover have ‹∃ *xs*. *lxs = lappend* (*llist-of xs*) (*LCons x lxs'*) ∧ (∀ *x*∈*set xs*. *P x*)›
if ‹*ldropWhile P lxs = LCons x lxs'*›
using *that* **by** (*metis eq-LConsD in-lset-lappend-iff lappend-ltakeWhile-ldropWhile ldropWhile-eq-LNil-iff llist-of-list-of lset-llist-of lset-ltakeWhileD*)
ultimately show *?thesis*
by (*auto simp*: *ldropWhile-lappend*)
qed

lemma *ltakeWhile-ldropWhile-decomp*:
assumes ‹*x* ∈ *lset lxs*›
shows ‹*lxs = lappend* (*ltakeWhile* ((≠) *x*) *lxs*) (*LCons x* (*ltl* (*ldropWhile* ((≠) *x*) *lxs*)))›
proof (*subst lappend-ltakeWhile-ldropWhile*[*symmetric*, *of lxs* ‹(≠) *x*›], *rule arg-cong*[**where** *f*=‹*lappend* -›])
from *assms* **show** ‹*ldropWhile* ((≠) *x*) *lxs = LCons x* (*ltl* (*ldropWhile* ((≠) *x*) *lxs*))›
by (*cases* ‹*ldropWhile* ((≠) *x*) *lxs*›)
(*auto simp*: *ldropWhile-eq-LNil-iff lhd-ldropWhile*[**where** *P*=‹(≠) *x*›, *simplified*, *symmetric*] *dest*!: *eq-LConsD*)
qed

lemma *lzip-lmap-same*: ‹*lzip* (*lmap f lxs*) (*lmap g lxs*) = *lmap* (λ*x*. (*f x*, *g x*)) *lxs*›
by (*coinduction arbitrary*: *lxs*) *auto*

lemma *llength-not-lnull*: ‹¬ *lnull lxs* ⟹ *llength lxs = eSuc* (*llength* (*ltl lxs*))›
by (*auto simp*: *lnull-def neq-LNil-conv*)

primrec *ltaken* :: ‹*nat* ⇒ *'a llist* ⇒ *'a list*› **where**
‹*ltaken 0 lxs* = []›
| ‹*ltaken* (*Suc i*) *lxs* = (*case lxs of LNil* ⇒ [] | *LCons x lxs* ⇒ *x* # *ltaken i lxs*)›

lemma *nth-ltaken*: ‹*m* < *n* ⟹ *n* ≤ *llength lxs* ⟹ *nth* (*ltaken n lxs*) *m* = *lnth lxs m*›
by (*induct n arbitrary*: *m lxs*) (*auto simp*: *nth-Cons' gr0-conv-Suc simp flip*: *eSuc-enat split*: *llist.splits*)

lemma *set-ltaken*: ‹*set* (*ltaken n lxs*) ⊆ *lset lxs*›
by (*induct n arbitrary*: *lxs*) (*force split*: *llist.splits*)+

lemma *length-ltaken*: ‹*length* (*ltaken n lxs*) = (*if enat n* ≤ *llength lxs then n else the-enat* (*llength lxs*))›
by (*induct n arbitrary*: *lxs*)

(*auto simp*: *enat-0 min-def not-le eSuc-enat enat-0-iff eSuc-enat-iff elim*!: *less-enatE split*: *llist.splits*)

lemma *set-ltaken-conv*: ‹$n \leq$ *llength lxs* $\Longrightarrow$ *set* (*ltaken n lxs*) = *lnth lxs* ‘ {*0*..<*n*}›
proof (*induct n arbitrary*: *lxs*)
case (*Suc n*)
then show *?case*
by (*cases lxs*)
(*force simp flip*: *eSuc-enat simp*: *image-iff lnth-LCons′ dest*: *bspec*[*of* - - ‹*Suc* -›])+
qed *simp*

lemma *ltaken-lappend*:
‹*ltaken n* (*lappend lxs lys*) = (*case llength lxs of* $\infty \Rightarrow$ *ltaken n lxs* | *enat m* $\Rightarrow$ *ltaken n lxs* @ *ltaken* (*n* − *m*) *lys*)›
by (*induct n arbitrary*: *lxs*) (*auto split*: *enat.splits simp*: *enat-0-iff eSuc-enat-iff eSuc-eq-infinity-iff split*: *llist.splits*)

lemma *ltaken-LNil*[*simp*]: ‹*ltaken i LNil* = []›
by (*cases i*) *auto*

3 enat as a Codatatype

codatatype *en* = *eZ* | *eS* (*ep*: *en*)
where ‹*ep eZ* = *eZ*›

coinductive *is-einf* **where**
‹*is-einf n* $\Longrightarrow$ *is-einf* (*eS n*)›

inductive *is-fin* **where**
‹*is-fin eZ*›
| ‹*is-fin n* $\Longrightarrow$ *is-fin* (*eS n*)›

lemma *not-is-fin-is-einf*: ‹$\neg$ *is-fin n* $\Longrightarrow$ *is-einf n*›
proof (*coinduction arbitrary*: *n*)
case *is-einf*
then show *?case*
by (*cases n*) (*auto intro*: *is-fin.intros*)
qed

lemma *is-fin-not-is-einf*: ‹*is-fin n* $\Longrightarrow$ $\neg$ *is-einf n*›
by (*induct n pred*: *is-fin*) (*auto elim*: *is-einf.cases*)

primcorec *einf* **where**
‹*einf* = *eS einf*›

lemma *einf-eS-iff*: ‹*einf* = *eS x* $\longleftrightarrow$ *x* = *einf*›
by (*subst einf.code*) *auto*

lemma *is-einf-einf*: ‹*is-einf einf*›
  by (*coinduction*; *subst einf.code*) *auto*

lemma *is-einf-eq-einf*: ‹*is-einf n* ⟹ *n* = *einf*›
  by (*coinduction arbitrary*: *n*) (*auto elim*: *is-einf.cases*)

fun *nat-to-en* **where**
  ‹*nat-to-en 0* = *eZ*›
| ‹*nat-to-en* (*Suc n*) = *eS* (*nat-to-en n*)›

lemma *is-fin-ex-nat-to-en*: ‹*is-fin n* ⟹ ∃ *m*. *n* = *nat-to-en m*›
  by (*induct n pred*: *is-fin*) (*auto intro*: *exI*[*of* - *0*] *exI*[*of* - ‹*Suc* -›])

lemma *inj-nat-to-en*: ‹*nat-to-en x* = *nat-to-en y* ⟹ *x* = *y*›
proof (*induct x arbitrary*: *y*)
  case *0*
  then show *?case*
    by (*cases y*; *simp*)
next
  case (*Suc x*)
  then show *?case*
    by (*cases y*; *simp*)
qed

lemma *nat-to-en-not-einf*: ‹*nat-to-en n* = *einf* ⟹ *False*›
  by (*induct n*; *subst* (*asm*) *einf.code*) *auto*

definition *enat-to-en* **where**
  ‹*enat-to-en n* = (*case n of enat n* ⇒ *nat-to-en n* | - ⇒ *einf*)›

lemma *inj-enat-to-en*: ‹*inj enat-to-en*›
  by (*auto simp*: *inj-on-def enat-to-en-def inj-nat-to-en*
      intro: *nat-to-en-not-einf nat-to-en-not-einf*[*OF sym*] *split*: *enat.splits*)

lemma *range-enat-to-en*: ‹*n* ∈ *range enat-to-en*›
proof (*cases* ‹*n* = *einf*›)
  case *True*
  then show *?thesis* **by** (*intro image-eqI*[*of* - - ∞]) (*auto simp*: *enat-to-en-def*)
next
  case *False*
  then have ‹*is-fin n*›
    using *is-einf-eq-einf not-is-fin-is-einf* **by** *auto*
  then obtain *m* **where** ‹*n* = *nat-to-en m*›
    using *is-fin-ex-nat-to-en* **by** *blast*
  then show *?thesis*
    by (*intro image-eqI*[*of* - - ‹*enat m*›]) (*auto simp*: *enat-to-en-def*)
qed

lemma *type-definition-enat*: ‹*type-definition enat-to-en* (*inv enat-to-en*) *UNIV*›

by *standard* (*auto intro*: *inv-f-f f-inv-into-f inj-enat-to-en simp*: *range-enat-to-en*)

setup-lifting *type-definition-enat*

lift-definition *corec-enat* :: ‹('*a* ⇒ *bool*) ⇒ ('*a* ⇒ *bool*) ⇒ ('*a* ⇒ *enat*) ⇒ ('*a* ⇒ '*a*) ⇒ '*a* ⇒ *enat*›
is *corec-en* **.**

lemma *eZ-transfer*[*transfer-rule*]: ‹*pcr-enat eZ 0*›
by (*auto simp*: *pcr-enat-def cr-enat-def enat-to-en-def enat-0*[*symmetric*])

lemma *eS-transfer*[*transfer-rule*]: ‹*rel-fun pcr-enat pcr-enat eS eSuc*›
by (*auto simp*: *pcr-enat-def cr-enat-def enat-to-en-def rel-fun-def eSuc-enat einf.code*[*symmetric*] *eq-OO split*: *enat.splits*)

lemma *ep-nat-to-en*: ‹*ep* (*nat-to-en n*) = *nat-to-en* (*n* − *Suc 0*)›
by (*induct n*) (*auto*)

lemma *ep-transfer*[*transfer-rule*]: ‹*rel-fun pcr-enat pcr-enat ep epred*›
by (*auto simp*: *pcr-enat-def cr-enat-def enat-to-en-def rel-fun-def eSuc-enat einf.code*[*symmetric*] *eq-OO ep-nat-to-en split*: *enat.splits*)

lemma *corec-enat-code*[*code*]:
‹*corec-enat zer stop end cnt a* =
(*if zer a then 0 else eSuc* (*if stop a then end a else corec-enat zer stop end cnt* (*cnt a*)))›
by *transfer* (*rule en.corec-code*)

instantiation *en* :: *comm-monoid-add*
begin

definition *zero-en* **where** ‹*zero-en* = *eZ*›

primcorec *plus-en* **where**
‹*plus-en e1 e2* = (*case e1 of eZ* ⇒ *e2* | *eS e1'* ⇒ *eS* (*plus-en e1' e2*))›

friend-of-corec *plus* :: ‹*en* ⇒ *en* ⇒ *en*› **where**
‹*e1* + *e2* = (*case e1 of eZ* ⇒ (*case e2 of eZ* ⇒ *eZ* | *eS e2'* ⇒ *eS e2'*) | *eS e1'* ⇒ *eS* (*e1'* + *e2*))›
by (*subst plus-en.code*; *simp split*: *en.splits*) *transfer-prover*

lemma *plus-einf-left*[*simp*]: ‹*einf* + *e* = *einf*›
by (*coinduction arbitrary*: *e*, *subst* (*5*) *einf.code*) (*auto split*: *en.splits*)

lemma *eZ-left-neutral*[*simp*]: ‹*eZ* + *e* = *e*›
by (*simp add*: *plus-en.code*)
lemma *eZ-right-neutral*[*simp*]: ‹*e* + *eZ* = *e*›
by (*coinduction arbitrary*: *e*) (*auto split*: *en.splits*)
lemma *plus-eS-left*[*simp*]: ‹*eS e1* + *e2* = *eS* (*e1* + *e2*)›

by (*simp add*: *plus-en.code*)
lemma *plus-eS-right*[*simp*]: ‹*e1* + *eS e2* = *eS* (*e1* + *e2*)›
by (*coinduction arbitrary*: *e1 e2 rule*: *en.coinduct-strong*) (*auto split*: *en.splits*)

instance
proof
fix *n m q* :: *en*
show ‹*n* + *m* + *q* = *n* + (*m* + *q*)›
by (*coinduction arbitrary*: *n m q rule*: *en.coinduct-strong*)
(*force split*: *en.splits intro*: *exI*[*of* - *eZ*] *exI*[*of* - ‹*eS* -›])
show ‹*n* + *m* = *m* + *n*›
by (*coinduction arbitrary*: *n m rule*: *en.coinduct-strong*)
(*force split*: *en.splits intro*: *exI*[*of* - ‹*eS* -›])
show ‹*0* + *n* = *n*›
by (*coinduction arbitrary*: *n*) (*auto simp*: *zero-en-def split*: *en.splits*)
qed

end

lemma *plus-einf-right*[*simp*]: ‹*e* + *einf* = *einf*›
by (*simp add*: *add.commute*)

lemma *nat-to-en-plus*[*simp*]: ‹*nat-to-en* (*m* + *n*) = *nat-to-en m* + *nat-to-en n*›
by (*induct m*) *auto*

lemma *ezero-transfer*[*transfer-rule*]: ‹*pcr-enat 0 0*›
by (*auto simp*: *zero-en-def eZ-transfer*)

lemma *eplus-transfer*[*transfer-rule*]: ‹*rel-fun pcr-enat* (*rel-fun pcr-enat pcr-enat*)
(+) (+)›
by (*auto simp*: *pcr-enat-def cr-enat-def enat-to-en-def rel-fun-def eSuc-enat einf.code*[*symmetric*]
eq-OO ep-nat-to-en split: *enat.splits*)

lemma *case-en-transfer*[*transfer-rule*]: ‹*rel-fun R* (*rel-fun* (*rel-fun pcr-enat R*) (*rel-fun*
pcr-enat R)) *case-en co.case-enat*›
by (*auto simp*: *pcr-enat-def cr-enat-def enat-to-en-def rel-fun-def eSuc-enat einf.code*[*symmetric*]
eq-OO ep-nat-to-en enat-0-iff enat-eSuc-iff infinity-eq-eSuc-iff einf-eS-iff split:
enat.splits en.splits co.enat.splits)

corec *lsum-en* **where**
‹*lsum-en lxs* = (*case ldropWhile* ((=) *0*) *lxs of LNil* ⇒ *0* | *LCons e lxs* ⇒ *eS* (*ep*
e + *lsum-en lxs*))›

lift-definition *lsum* :: ‹*enat llist* ⇒ *enat*›
is *lsum-en* **.**

lemmas *lsum-code*[*code*] = *lsum-en.code*[*transferred*]

lemma *lsum-code-alt*:

‹*lsum lxs* = (*case ldropWhile* ((=) *0*) *lxs of LNil* ⇒ *0* | *LCons e lxs* ⇒ *e* + *lsum lxs*)›
by (*subst lsum-code*, *cases* ‹*lhd* (*ldropWhile* ((=) *0*) *lxs*)› *rule*: *co.enat.exhaust*)
(*auto simp*: *iadd-Suc dest*: *ldropWhile-eq-LCons-iff*[*THEN iffD1*] *split*: *llist.splits*)

4 Counting in Lazy Lists

primcorec *count-llist-en* **where**
‹*count-llist-en lxs x* = (*if x* ∈ *lset lxs then eS* (*count-llist-en* (*ltl* (*ldropWhile* ((≠) *x*) *lxs*)) *x*) *else eZ*)›

lift-definition *count-llist* :: ‹'*a llist* ⇒ '*a* ⇒ *enat*›
is *count-llist-en* **.**

lemmas *count-llist-code*[*code*] = *count-llist-en.code*[*transferred*]
lemmas *count-llist-sel*[*simp*] = *count-llist-en.sel*[*transferred*]
lemmas *count-llist-ctr* = *count-llist-en.ctr*[*transferred*]
lemmas *enat-coinduct-strong*[*case-names eq-enat*, *coinduct type*] = *en.coinduct-strong*[*transferred*]

lift-definition *enat-cong* :: ‹(*enat* ⇒ *enat* ⇒ *bool*) ⇒ *enat* ⇒ *enat* ⇒ *bool*› **is** ‹*en.congclp*› **.**
lemmas *enat-coinduct-upto*[*case-names eq-enat*] = *en.coinduct-upto*[*transferred*]
lemmas *enat-cong-intros* = *en.cong-intros*[*transferred*]

lifting-update *enat.lifting*
lifting-forget *enat.lifting*

lemma *in-lset-iff-count-llist*: ‹*x* ∈ *lset lxs* ⟷ *count-llist lxs x* ≠ *0*›
by (*subst count-llist-code*) *auto*

lemma *count-llist-zero-iff*: ‹*count-llist lxs x* = *0* ⟷ *x* ∉ *lset lxs*›
by (*metis in-lset-iff-count-llist*)

lemma *count-llist-alt*: ‹*count-llist lxs x* = *llength* (*lfilter* ((=) *x*) *lxs*)›
by (*coinduction arbitrary*: *lxs*) (*auto simp*: *count-llist-zero-iff epred-llength ltl-lfilter*)

lemma *count-llist-lappend*:
‹*count-llist* (*lappend lxs lys*) *x* = *count-llist lxs x* + (*if lfinite lxs then count-llist lys x else 0*)›
proof (*coinduction arbitrary*: *lxs lys*)
case *eq-enat*
then show *?case*
by (*auto simp*: *count-llist-zero-iff epred-iadd1 lset-lappend-conv lappend-inf ldropWhile-lappend*
lfinite-ldropWhile count-llist-ctr(*1*)[*of x lxs*])
qed

lemma *count-llist-LNil*[*simp*]: ‹*count-llist LNil x* = *0*›
by (*subst count-llist-code*) *auto*

lemma *count-llist-LCons*[*simp*]: ‹*count-llist* (*LCons y lxs*) *x* =
(*if x* = *y then eSuc* (*count-llist lxs x*) *else count-llist lxs x*)›
by (*subst* (*1 3*) *count-llist-code*) (*auto simp*: *count-llist-zero-iff*)

5 Lazy Lists of Natural Numbers

primcorec *lupt* **where**
‹*lupt i j* = (*if enat i* ≥ *j then LNil else LCons i* (*lupt* (*Suc i*) *j*))›

lemma *lset-luptD*[*OF - refl*]:
assumes ‹*k* ∈ *lset lxs*› ‹*lxs* = *lupt i j*›
shows ‹*i* ≤ *k*› ‹*k* < *j*›
using *assms*
proof (*induct k lxs arbitrary*: *i rule*: *llist.set-induct*)
case (*LCons1 z1 z2*)
{
case *1*
then show *?case*
by (*subst* (*asm*) *lupt.code*) (*auto split*: *if-splits*)
next
case *2*
then show *?case*
by (*subst* (*asm*) *lupt.code*) (*auto split*: *if-splits*)
}
next
case (*LCons2 z1 z2 k*)
{
case *1*
with *LCons2*(*2*,*3*)[*of* ‹*Suc i*›] **show** *?case*
by (*subst* (*asm*) (*3*) *lupt.code*) (*auto split*: *if-splits*)
next
case *2*
with *LCons2*(*2*,*3*)[*of* ‹*Suc i*›] **show** *?case*
by (*subst* (*asm*) (*3*) *lupt.code*) (*auto split*: *if-splits*)
}
qed

lemma *lset-luptI*: ‹*i* ≤ *k* ⟹ *enat k* < *j* ⟹ *k* ∈ *lset* (*lupt i j*)›
proof (*induct* ‹*k* − *i*› *arbitrary*: *i*)
case *0*
then show *?case*
by (*subst lupt.code*) *auto*
next
case (*Suc x*)
then have ‹*enat i* < *j*›
by (*meson enat-ord-simps*(*1*) *order.strict-trans1*)
from *Suc*(*2*) **have** ‹*x* = *k* − *Suc i*›
by *force*

```
  with Suc(1)[OF this] Suc(2–4) ‹enat i < j› show ?case
    by (subst lupt.code) (auto simp: Suc-le-eq)
qed

lemma lset-lupt: ‹lset (lupt i j) = {k. i ≤ k ∧ enat k < j}›
  by (auto intro: lset-luptI dest: lset-luptD)

lemma llength-lupt[simp]: ‹llength (lupt i j) = j − i›
proof (coinduction arbitrary: i)
  case eq-enat
  then show ?case
  proof (cases j)
    case (enat nat)
    then show ?thesis
      by (subst (3) lupt.code) (auto simp: enat-0 enat-0-iff)
  next
    case infinity
    then show ?thesis
      by (subst (3) lupt.code) auto
  qed
qed

lemma lnth-lupt: ‹k < j − enat i ⟹ lnth (lupt i j) k = i + k›
proof (induct k arbitrary: i)
  case 0
  then show ?case by (cases j) (auto simp add: lnth-0-conv-lhd)
next
  case (Suc k)
  from Suc(2) have ‹enat k < j − Suc i›
    by (cases j) auto
  with Suc(1)[OF this] Suc(2) show ?case
    by (subst lupt.code; cases j) auto
qed

lemma lmap-lupt-Suc: ‹lmap f (lupt (Suc i) (eSuc j)) = lmap (f o Suc) (lupt i j)›
  by (coinduction arbitrary: i j) (auto simp: eSuc-def split: enat.splits)

lemma llist-conv-lmap-lupt:
  ‹lxs = lmap (lnth lxs) (lupt 0 (llength lxs))›
  by (coinduction arbitrary: lxs)
    (auto simp: enat-0 lnth-0-conv-lhd llength-not-lnull lmap-lupt-Suc lnth-ltl)

lemma ldistinct-lupt[simp]: ‹ldistinct (lupt i j)›
  by (coinduction arbitrary: i) (auto simp: lset-lupt)

lemma lzip-lupt: ‹lzip (lupt i j) (lupt i k) = lmap (λx. (x, x)) (lupt i (min j k))›
  by (coinduction arbitrary: i) (auto simp: min-def)
```

6 Permutations of Lazy Lists

definition ‹*lpermute* π *lxs* = *lmap* (λi. *lnth lxs* (π *i*)) (*lupt 0* (*llength lxs*))›

abbreviation ‹*lbij-on* π *lxs* ≡ *bij-betw* π {*k*. *enat k* < *llength lxs*} {*k*. *enat k* < *llength lxs*}›

lemma *lset-lpermute*:
assumes ‹*lbij-on* π *lxs*›
shows ‹*lset* (*lpermute* π *lxs*) = *lset lxs*›
proof −
have ‹∃ *m*. *lnth lxs n* = *lnth* (*lmap* (λi. *lnth lxs* (π *i*)) (*lupt 0* (*llength lxs*))) *m* ∧ *enat m* < *llength lxs*›
if ‹*enat n* < *llength lxs*›
for *n* :: *nat*
using *that assms bij-betw-inv-into-right*[*OF assms*, *of n*]
by (*intro exI*[*of* - ‹*inv-into* {*k*. *enat k* < *llength lxs*} π *n*›])
(*auto 0 3 simp*: *lnth-lupt elim*: *bij-betwE*[*OF bij-betw-inv-into*, *THEN bspec*, *elim-format*])
with *assms* **show** *?thesis*
by (*auto 0 3 simp*: *lpermute-def image-iff lset-lupt lset-conv-lnth lnth-lupt dest*: *bij-betwE*)
qed

lemma *llength-lpermute*[*simp*]: ‹*llength* (*lpermute* π *lxs*) = *llength lxs*›
by (*auto simp*: *lpermute-def*)

lemma *lnth-lpermute*[*simp*]: ‹*lbij-on* π *lxs* ⟹ *i* < *llength lxs* ⟹ *lnth* (*lpermute* π *lxs*) *i* = *lnth lxs* (π *i*)›
by (*simp add*: *lnth-lupt lpermute-def*)

lemma *lfilter-permute*: ‹*ldistinct lxs* ⟹ *ldistinct lys* ⟹ *bij-betw* π (*lset lxs*) (*lset lys*) ⟹ ∀ *x* ∈ *lset lxs*. *P x* ⟷ *Q* (π *x*) ⟹
llength (*lfilter P lxs*) = *llength* (*lfilter Q lys*)›
proof (*coinduction arbitrary*: *lxs lys* π)
case *eq-enat*
then have ‹*llength* (*lfilter P lxs*) = *0* ⟷ *llength* (*lfilter Q lys*) = *0*›
by (*force simp*: *bij-betw-iff-bijections*)
moreover
{ assume ∗: ‹*llength* (*lfilter P lxs*) ≠ *0*› ‹*llength* (*lfilter Q lys*) ≠ *0*›
then have *lfin*[*unfolded comp-def*, *simp*]: ‹*lfinite* (*ltakeWhile* (*Not o P*) *lxs*)› ‹*lfinite* (*ltakeWhile* (*Not o Q*) *lys*)›
by (*simp-all add*: *lfinite-ltakeWhile*)
let *?lxs* = ‹*lappend* (*ltakeWhile* (*Not o P*) *lxs*) (*ltl* (*ldropWhile* (*Not o P*) *lxs*))›
let *?lys* = ‹*lappend* (*ltakeWhile* (*Not o Q*) *lys*) (*ltl* (*ldropWhile* (*Not o Q*) *lys*))›
define *x* **where** ‹*x* = *lhd* (*ldropWhile* (*Not o P*) *lxs*)›
with ∗(*1*) **have** ‹*P x*› **using** *lhd-ldropWhile* **by** *force*
from ∗(*1*) **have** ‹*x* ∈ *lset lxs*› **unfolding** *x-def*
by (*metis in-lset-ldropWhileD lfilter-eq-LCons lhd-lfilter llength-eq-0 llist.exhaust-sel*

llist.set-intros(1) llist.set-sel(1))
define *y* **where** ‹*y* = *lhd* (*ldropWhile* (*Not o Q*) *lys*)›
with ∗(*2*) **have** ‹*Q y*› **using** *lhd-ldropWhile* **by** *force*
from ∗(*2*) **have** ‹*y* ∈ *lset lys*› **unfolding** *y-def*
by (*metis in-lset-ldropWhileD lfilter-eq-LCons lhd-lfilter llength-eq-0 llist.exhaust-sel llist.set-intros(1) llist.set-sel(1)*)
let *?π* = ‹*π*(*inv-into* (*lset lxs*) *π y* := *π x*)›
have
‹*epred* (*llength* (*lfilter P lxs*)) = *llength* (*lfilter P ?lxs*)›
‹*epred* (*llength* (*lfilter Q lys*)) = *llength* (*lfilter Q ?lys*)›
by (*auto simp add*: *epred-llength ltl-lfilter dest*: *lset-ltakeWhileD*)
moreover from *eq-enat(1,2)*
lappend-ltakeWhile-ldropWhile[*symmetric*, *of lxs* ‹*Not o P*›, *THEN arg-cong*, *of ldistinct*, *THEN iffD1*, *OF eq-enat(1)*]
lappend-ltakeWhile-ldropWhile[*symmetric*, *of lys* ‹*Not o Q*›, *THEN arg-cong*, *of ldistinct*, *THEN iffD1*, *OF eq-enat(2)*]
have ‹*ldistinct ?lxs*› ‹*ldistinct ?lys*›
by (*auto simp add*: *ldistinct-lappend ldistinct-ldrop ldistinct-ltlI*
ldistinct-lprefix lprefix-ltakeWhile ldropWhile-eq-ldrop dest!: *in-lset-ltlD*)
moreover
have ‹*lxs* = *lappend* (*ltakeWhile* (*Not* ∘ *P*) *lxs*) (*LCons x* (*ltl* (*ldropWhile* (*Not* ∘ *P*) *lxs*)))›
‹*lys* = *lappend* (*ltakeWhile* (*Not* ∘ *Q*) *lys*) (*LCons y* (*ltl* (*ldropWhile* (*Not* ∘ *Q*) *lys*)))›
using ∗(*1,2*) *x-def y-def* **by** *force+*
with ‹*P x*› ‹*x* ∈ *lset lxs*› ‹*Q y*› ‹*y* ∈ *lset lys*› *eq-enat(1,2)*
have ‹*lset ?lxs* = *lset lxs* − {*x*}› ‹*lset ?lys* = *lset lys* − {*y*}›
by (*smt* (*verit*, *ccfv-SIG*) *Diff-insert-absorb Un-insert-right comp-eq-dest-lhs lset-ltakeWhileD*
in-lset-lappend-iff ldistinct-LCons ldistinct-lappend lset-LCons lset-lappend-conv)+
with *eq-enat(3,4)* ‹*x* ∈ *lset lxs*› ‹*y* ∈ *lset lys*› **have** ‹*bij-betw ?π* (*lset ?lxs*) (*lset ?lys*)›
by (*auto elim*!: *bij-betw-singl-remap*)
moreover from *eq-enat(4)* ‹*P x*› ‹*Q y*› ‹*x* ∈ *lset lxs*› ‹*y* ∈ *lset lys*› **have** ‹∀ *x* ∈ *lset ?lxs*. *P x* ⟷ *Q* (*?π x*)›
using *bij-betw-inv-into-right*[*OF eq-enat(3)*, *of y*]
by (*auto simp*: *in-lset-lappend-iff lhd-ldropWhile-in-lset*
dest!: *lset-ltakeWhileD in-lset-ldropWhileD in-lset-ltlD*)
note ∗ = *calculation this*
}
ultimately show *?case*
by *blast*
qed

lemma *count-llist-lpermute*:
‹*lbij-on π lxs* ⟹ *count-llist* (*lpermute π lxs*) *x* = *count-llist lxs x*›
unfolding *count-llist-alt*
by (*subst* (*2*) *llist-conv-lmap-lupt*[*of lxs*]) (*auto simp*: *lpermute-def lfilter-lmap lset-lupt*

lfilter-permute[*of* ‹*lupt 0* (*llength lxs*)› ‹*lupt 0* (*llength lxs*)› *π* ‹*λi. x = lnth lxs* (*π i*)› ‹*λi. x = lnth lxs i*›])

lemma *lpermute-lzip*: ‹*lbij-on π lxs* ⟹ *llength lys* = *llength lxs* ⟹ *lpermute π* (*lzip lxs lys*) = *lzip* (*lpermute π lxs*) (*lpermute π lys*)›
by (*auto simp*: *lpermute-def lzip-lmap-same lzip-lupt llist.map-comp lset-lupt lnth-lzip bij-betw-iff-bijections intro*!: *llist.map-cong*)

lemma *exist-nth-occurrence-in-llist*:
‹*count-llist lxs x* > *n* ⟹ ∃ *i* < *llength lxs*. *lnth lxs i* = *x* ∧ *count-list* (*ltaken i lxs*) *x* = *n*›
proof (*induct n arbitrary*: *lxs*)
case *0*
then have *ex*: ‹∃ *i* < *llength lxs*. *lnth lxs i* = *x*›
by (*metis in-lset-conv-lnth in-lset-iff-count-llist not-gr-zero zero-enat-def*)
define *i* **where** ‹*i* = (*LEAST i*. *enat i* < *llength lxs* ∧ *lnth lxs i* = *x*)›
have *i*: ‹*enat i* < *llength lxs*› ‹*lnth lxs i* = *x*›
‹⋀*j*. *enat j* < *llength lxs* ⟹ *lnth lxs j* = *x* ⟹ *i* ≤ *j*›
unfolding *i-def* **using** *LeastI2-wellorder-ex*[*OF ex*]
by (*force simp*: *Least-le*)+
moreover have ‹*lnth lxs i* = *lnth lxs j* ⟹ *j* < *i* ⟹ *False*› **for** *j*
using *i* **by** (*metis order.strict-trans enat-ord-simps*(*2*) *less-le-not-le*)
ultimately show *?case*
by (*force simp*: *count-list-0-iff set-ltaken-conv intro*!: *exI*[*of - i*])
next
case (*Suc n*)
let *?lxs* = ‹*ltl* (*ldropWhile* ((≠) *x*) *lxs*)›
from *Suc*(*2*) **have** ‹*enat n* < *count-llist ?lxs x*›
by (*metis Suc-ile-eq count-llist-code iless-Suc-eq not-less-zero*)
with *Suc*(*1*) **obtain** *i* **where** *i*: ‹*enat i* < *llength ?lxs*›
‹*lnth ?lxs i* = *x*› ‹*count-list* (*ltaken i ?lxs*) *x* = *n*›
by *blast*
from *Suc*(*2*) **obtain** *k* **where** *k*: ‹*llength* (*ltakeWhile* ((≠) *x*) *lxs*) = *enat k*›
by (*metis* (*full-types*) *enat-the-enat in-lset-iff-count-llist llength-ltakeWhile-eq-infinity*
lset-ltakeWhileD not-less-zero)
then have *klen*: ‹*enat k* < *llength lxs*›
by (*metis gr-implies-not-zero i*(*1*) *ldropWhile-eq-ldrop ldrop-eq-LNil llength-LNil*
lprefix-llength-le lprefix-ltakeWhile ltl-simps(*1*) *order-neq-le-trans*)
let *?i* = ‹*k* + *Suc i*›
have *lxs*: ‹*lxs* = *lappend* (*ltakeWhile* ((≠) *x*) *lxs*) (*LCons x ?lxs*)›
by (*metis Suc*(*2*) *in-lset-iff-count-llist ltakeWhile-ldropWhile-decomp not-less-zero*)
have ‹*enat ?i* < *llength lxs*›
by (*subst lxs*) (*metis eSuc-enat enat-less-enat-plusI2 i*(*1*) *ileI1 iless-Suc-eq k*
llength-LCons
llength-lappend)
moreover from *k i*(*2*) **have** ‹*lnth lxs ?i* = *x*›
by (*subst lxs*) (*auto simp*: *lnth-lappend*)
moreover have *x*: ‹*x* ∉ *lset* (*ltakeWhile* ((≠) *x*) *lxs*)›
by (*metis* (*full-types*) *lset-ltakeWhileD*)

with *k llength-ltakeWhile-le*[*of* ‹(≠) *x*› *lxs*] **have** ‹*x* ∉ *set* (*ltaken k lxs*)›
by (*subst set-ltaken-conv*) (*auto simp*: *in-lset-conv-lnth ltakeWhile-nth*)
with *k klen i*(*3*) *x* **have** ‹*count-list* (*ltaken ?i lxs*) *x* = *Suc n*›
using [[*simproc del*: *list-to-set-comprehension*]]
by (*subst lxs*) (*auto simp*: *ltaken-lappend dest*!: *set-mp*[*OF set-ltaken*] *intro*!: *count-notin split*: *llist.splits*)
ultimately show *?case* **by** *blast*
qed

function *lfind-index* **where**
‹*lfind-index n x i lxs* = (*if count-llist lxs x* ≤ *enat n then i else*
if lhd lxs = *x then case n of 0* ⇒ *i* | *Suc m* ⇒ *lfind-index m x* (*Suc i*) (*ltl lxs*)
else lfind-index n x (*Suc i*) (*ltl lxs*))›
by *auto*
termination
proof (*relation* ‹*measure* (λ(*n*, *x*, -, *lxs*). *LEAST i*. *enat i* < *llength lxs* ∧ *lnth lxs i* = *x* ∧ *count-list* (*ltaken i lxs*) *x* = *n*)›)
fix *n x i* **and** *lxs* :: ‹'*a llist*› **and** *m*
assume *hlt*: ‹¬ *count-llist lxs x* ≤ *enat n*›
assume *hlhd*: ‹*lhd lxs* = *x*›
assume *hn*: ‹*n* = *Suc m*›
let *?PP* = ‹λ*x lxs k n*. *enat k* < *llength lxs* ∧ *lnth lxs k* = *x* ∧ *count-list* (*ltaken k lxs*) *x* = *n*›
let *?P* = ‹*?PP x*›
from *hlt hn* **have** *hcount*: ‹*enat* (*Suc m*) < *count-llist lxs x*›
by (*simp add*: *not-le*)
obtain *j* **where** *hj1*: ‹*enat j* < *llength lxs*› **and** *hj2*: ‹*lnth lxs j* = *x*›
and *hj3*: ‹*enat* (*count-list* (*ltaken j lxs*) *x*) = *enat* (*Suc m*)›
using *exist-nth-occurrence-in-llist*[*OF hcount*] **by** *blast*
have *hj3'*: ‹*count-list* (*ltaken j lxs*) *x* = *Suc m*›
using *hj3* **by** *simp*
have *rhs-eq*: ‹(*LEAST k*. *?P lxs k* (*Suc m*)) = *Suc* (*LEAST k*. *?P lxs* (*Suc k*) (*Suc m*))›
proof (*rule Least-Suc*)
show ‹*?P lxs j* (*Suc m*)›
using *hj1 hj2 hj3'* **by** *simp*
show ‹¬ *?P lxs 0* (*Suc m*)›
by *simp*
qed
have *lhs-eq*: ‹(*LEAST k*. *?P* (*ltl lxs*) *k m*) = (*LEAST k*. *?P lxs* (*Suc k*) (*Suc m*))›
using *hlhd* **by** (*cases lxs*) (*auto simp*: *Suc-ile-eq lnth-LCons' fun-eq-iff gr0-conv-Suc*)
have *key2*: ‹(*LEAST k*. *?P lxs k* (*Suc m*)) = *Suc* (*LEAST k*. *?P* (*ltl lxs*) *k m*)›
using *rhs-eq lhs-eq* **by** *simp*
show ‹((*m*, *x*, *Suc i*, *ltl lxs*), *n*, *x*, *i*, *lxs*) ∈ *measure* (λ(*n*, *x*, -, *lxs*). *LEAST i*. *?PP x lxs i n*)›
by (*simp add*: *hn key2*)
next
fix *n x i* **and** *lxs* :: ‹'*a llist*›
assume *hlt*: ‹¬ *count-llist lxs x* ≤ *enat n*›

assume *hlhd*: ‹*lhd lxs* ≠ *x*›
let *?PP* = ‹λ*x lxs k n*. *enat k* < *llength lxs* ∧ *lnth lxs k* = *x* ∧ *count-list* (*ltaken k lxs*) *x* = *n*›
let *?P* = ‹*?PP x*›
from *hlt* **have** *hcount*: ‹*enat n* < *count-llist lxs x*›
by (*simp add*: *not-le*)
obtain *j* **where** *hj1*: ‹*enat j* < *llength lxs*› **and** *hj2*: ‹*lnth lxs j* = *x*›
and *hj3*: ‹*enat* (*count-list* (*ltaken j lxs*) *x*) = *enat n*›
using *exist-nth-occurrence-in-llist*[*OF hcount*] **by** *blast*
have *hj3'*: ‹*count-list* (*ltaken j lxs*) *x* = *n*›
using *hj3* **by** *simp*
have *rhs-eq*: ‹(*LEAST k*. *?P lxs k n*) = *Suc* (*LEAST k*. *?P lxs* (*Suc k*) *n*)›
proof (*rule Least-Suc*)
show ‹*?P lxs j n*›
using *hj1 hj2 hj3'* **by** *simp*
show ‹¬ *?P lxs 0 n*›
using *hlhd* **by** (*cases lxs*) (*auto simp*: *lhd-conv-lnth enat-0*)
qed
have *lhs-eq*: ‹(*LEAST k*. *?P* (*ltl lxs*) *k n*) = (*LEAST k*. *?P lxs* (*Suc k*) *n*)›
using *hlhd* **by** (*cases lxs*) (*auto simp*: *Suc-ile-eq lnth-LCons' fun-eq-iff gr0-conv-Suc*)
show ‹((*n*, *x*, *Suc i*, *ltl lxs*), *n*, *x*, *i*, *lxs*) ∈ *measure* (λ(*n*, *x*, -, *lxs*). *LEAST i*. *?PP x lxs i n*)›
by (*simp add*: *lhs-eq rhs-eq less-Suc-eq-le*)
qed *simp*
declare *lfind-index.simps*[*simp del*]

lemma *lfind-index-ge*[*simp*]: ‹*lfind-index n x j lxs* ≥ *j*›
proof (*induct n x j lxs rule*: *lfind-index.induct*)
case (*1 n x i lxs*)
then show *?case*
by (*subst lfind-index.simps*) (*auto dest*: *Suc-leD split*: *nat.splits*)
qed

lemma *lfind-index-less*[*simp*]: ‹*count-llist lxs x* > *enat n* ⟹ *lfind-index n x j lxs* < *llength lxs* + *j*›
proof (*induct n x j lxs rule*: *lfind-index.induct*)
case (*1 n x i lxs*)
then show *?case*
by (*subst lfind-index.simps*; *cases lxs*)
(*auto split*: *nat.splits simp*: *enat-0 count-llist-zero-iff Suc-ile-eq eSuc-plus iadd-Suc-right*
simp flip: *eSuc-enat*)
qed

lemma *lfind-index-lnth*:
‹*count-llist lxs x* > *enat n* ⟹ *lnth lxs* (*lfind-index n x j lxs* − *j*) = *x*›
proof (*induct n x j lxs rule*: *lfind-index.induct*)
case (*1 n x i lxs*)
show *?case*

```
  proof −
    note IH1 = 1(1)
    note IH2 = 1(2)
    from 1(3) have hcount: ‹enat n < count-llist lxs x› by simp
    have hlt: ‹¬ count-llist lxs x ≤ enat n› using hcount by simp
    obtain h t where hlxs: ‹lxs = LCons h t›
      using hcount by (cases lxs) (auto simp: count-llist-zero-iff)
    have step1: ‹lfind-index n x i lxs =
     (if lhd lxs = x then case n of 0 ⇒ i | Suc m ⇒ lfind-index m x (Suc i) (ltl lxs)
      else lfind-index n x (Suc i) (ltl lxs))›
      by (subst lfind-index.simps) (simp add: hlt)
    show ?thesis
    proof (cases ‹h = x›)
      case hx: True
      show ?thesis
      proof (cases n)
        case 0
        with hx hlxs hlt step1 show ?thesis by simp
      next
        case (Suc m)
        have cnt-t: ‹enat m < count-llist t x›
          using hcount hlxs hx Suc by (simp add: Suc-ile-eq)
        have ih: ‹lnth t (lfind-index m x (Suc i) t − Suc i) = x›
          using IH1[of m] hlt hlxs hx Suc cnt-t by simp
        have ‹Suc i ≤ lfind-index m x (Suc i) t› by simp
        then have ‹¬ lfind-index m x (Suc i) t ≤ i› by linarith
        then show ?thesis
          using ih hx hlxs Suc step1 by (simp add: lnth-LCons′)
      qed
    next
      case hx: False
      have cnt-t: ‹enat n < count-llist t x›
        using hcount hlxs hx by simp
      have ih: ‹lnth t (lfind-index n x (Suc i) t − Suc i) = x›
        using IH2 hlt hlxs hx cnt-t by simp
      have ‹Suc i ≤ lfind-index n x (Suc i) t› by simp
      then have ‹¬ lfind-index n x (Suc i) t ≤ i› by linarith
      then show ?thesis
        using ih hx hlxs step1 by (simp add: lnth-LCons′)
    qed
  qed
qed

lemma ltaken-LCons:
  ‹ltaken i (LCons x lxs) = (case i of 0 ⇒ [] | Suc j ⇒ x # ltaken j lxs)›
  by (cases i; simp)

lemma lfind-index-count-list:
  ‹count-llist lxs x > n ⟹ count-list (ltaken (lfind-index n x j lxs − j) lxs) x = n›
```

```
proof (induct n x j lxs rule: lfind-index.induct)
  case (1 n x i lxs)
  show ?case
  proof −
    note IH1 = 1(1)
    note IH2 = 1(2)
    from 1(3) have hcount: ‹enat n < count-llist lxs x› by simp
    have hlt: ‹¬ count-llist lxs x ≤ enat n› using hcount by simp
    obtain h t where hlxs: ‹lxs = LCons h t›
      using hcount by (cases lxs) (auto simp: count-llist-zero-iff)
    have step1: ‹lfind-index n x i lxs =
     (if lhd lxs = x then case n of 0 ⇒ i | Suc m ⇒ lfind-index m x (Suc i) (ltl lxs)
      else lfind-index n x (Suc i) (ltl lxs))›
      by (subst lfind-index.simps) (simp add: hlt)
    show ?thesis
    proof (cases ‹h = x›)
      case hx: True
      show ?thesis
      proof (cases n)
        case 0
        with hx hlxs step1 show ?thesis by (simp add: ltaken-LCons)
      next
        case (Suc m)
        have cnt-t: ‹enat m < count-llist t x›
          using hcount hlxs hx Suc by (simp add: Suc-ile-eq)
        have ih: ‹count-list (ltaken (lfind-index m x (Suc i) t − Suc i) t) x = m›
          using IH1[of m] hlt hlxs hx Suc cnt-t by simp
        have ge: ‹Suc i ≤ lfind-index m x (Suc i) t› by simp
        have nle: ‹¬ lfind-index m x (Suc i) t ≤ i› using ge by linarith
        have diff-eq: ‹lfind-index m x (Suc i) t − i = Suc (lfind-index m x (Suc i)
t − Suc i)›
          using ge by arith
        show ?thesis
          using ih nle hx hlxs Suc step1 diff-eq by (simp add: ltaken-LCons)
      qed
    next
      case hx: False
      have cnt-t: ‹enat n < count-llist t x›
        using hcount hlxs hx by simp
      have ih: ‹count-list (ltaken (lfind-index n x (Suc i) t − Suc i) t) x = n›
        using IH2 hlt hlxs hx cnt-t by simp
      have ge: ‹Suc i ≤ lfind-index n x (Suc i) t› by simp
      have nle: ‹¬ lfind-index n x (Suc i) t ≤ i› using ge by linarith
      have diff-eq: ‹lfind-index n x (Suc i) t − i = Suc (lfind-index n x (Suc i) t −
Suc i)›
        using ge by arith
      show ?thesis
        using ih nle hx hlxs step1 diff-eq by (simp add: ltaken-LCons)
    qed
```

qed
qed

lemma *lnth-equalityI*: ‹*llength lxs = llength lys* ⟹ (⋀*i*. *enat i < llength lxs* ⟹ *lnth lxs i = lnth lys i*) ⟹ *lxs = lys*›
by (*metis* (*full-types*) *llist.rel-eq llist-all2-all-lnthI*)

lemma *lfind-index-inject*:
‹*count-llist lxs x > enat n* ⟹ *count-llist lxs y > enat m* ⟹
lfind-index n x j lxs = lfind-index m y j lxs ⟹ *n = m* ∧ *x = y*›
by (*metis lfind-index-count-list lfind-index-lnth*)

lemma *count-list-ltaken-less*:
‹*i < llength lxs* ⟹ *count-list* (*ltaken i lxs*) (*lnth lxs i*) < *count-llist lxs* (*lnth lxs i*)›
proof (*induct i arbitrary*: *lxs*)
case *0*
then show *?case*
by (*auto simp*: *enat-0 count-llist-zero-iff lnth-0-conv-lhd*)
next
case (*Suc i*)
then show *?case*
by (*cases lxs*) (*auto simp flip*: *eSuc-enat*)
qed

lemma *count-list-inject*:
‹*enat i < llength lxs* ⟹ *enat j < llength lxs* ⟹
count-list (*ltaken i lxs*) (*lnth lxs j*) = *count-list* (*ltaken j lxs*) (*lnth lxs j*) ⟹
lnth lxs i = lnth lxs j ⟹ *i = j*›
by (*induct i arbitrary*: *j lxs*)
(*auto simp*: *count-list-0-iff set-ltaken-conv image-iff ltaken-LCons Suc-ile-eq Ball-def split*: *llist.splits if-splits nat.splits*)

7 Same-Count Equivalence Relation

definition ‹*eq-cmset lxs lys* = (∀ *x*. *count-llist lxs x = count-llist lys x*)›

lemma *lset-eq-cmset*: ‹*eq-cmset lxs lys* ⟹ *lset lxs = lset lys*›
by (*auto simp*: *eq-cmset-def in-lset-iff-count-llist*)

lemma *eq-cmset-llength*: ‹*eq-cmset lxs lys* ⟹ *llength lxs = llength lys*›
unfolding *eq-cmset-def*
proof (*coinduction arbitrary*: *lxs lys*)
case *eq-enat*
then have *heq*: ‹∀ *x*. *count-llist lxs x = count-llist lys x*› **by** *blast*
have *lnull-iff*: ‹*lnull lxs* ⟷ *lnull lys*›
using *heq* **by** (*auto simp*: *lnull-def count-llist-zero-iff*)
show *?case*
proof (*cases* ‹*lnull lxs*›)

```
    case True
    then show ?thesis using lnull-iff by (auto simp: epred-llength)
  next
    case lxs-ne: False
    then obtain h t where hlxs: ‹lxs = LCons h t› by (cases lxs) auto
    from lnull-iff lxs-ne have ‹¬ lnull lys› by simp
    then have h-in-lys: ‹h ∈ lset lys›
    proof −
      have hpos: ‹0 < count-llist lxs h› by (simp add: hlxs)
      have ‹0 < count-llist lys h› using heq hpos by simp
      then show ?thesis by (simp add: in-lset-iff-count-llist)
    qed
    let ?lys' = ‹lappend (ltakeWhile ((≠) h) lys) (ltl (ldropWhile ((≠) h) lys))›
   have lys-decomp: ‹lys = lappend (ltakeWhile ((≠) h) lys) (LCons h (ltl (ldropWhile
((≠) h) lys)))›
      using h-in-lys by (metis ltakeWhile-ldropWhile-decomp)
    have epred-lxs: ‹epred (llength lxs) = llength t›
      by (simp add: hlxs epred-llength)
    have epred-lys: ‹epred (llength lys) = llength ?lys'›
      by (subst lys-decomp) (simp add: iadd-Suc-right)
    have lfinite-take: ‹lfinite (ltakeWhile ((≠) h) lys)›
      by (simp add: lfinite-ltakeWhile) (use h-in-lys in blast)
    have h-not-in-take: ‹count-llist (ltakeWhile ((≠) h) lys) h = 0›
      by (force simp: count-llist-zero-iff dest: lset-ltakeWhileD)
    have heq': ‹∀ x. count-llist t x = count-llist ?lys' x›
    proof
      fix x
      have hx: ‹count-llist (LCons h t) x = count-llist lys x›
        using heq hlxs by blast
      have lys-x: ‹count-llist lys x =
          count-llist (ltakeWhile ((≠) h) lys) x +
          count-llist (LCons h (ltl (ldropWhile ((≠) h) lys))) x›
        by (subst lys-decomp) (simp add: count-llist-lappend lfinite-take)
      show ‹count-llist t x = count-llist ?lys' x›
        using hx lys-x h-not-in-take lfinite-take
        by (simp add: count-llist-lappend split: if-splits)
    qed
    show ?thesis
      using lnull-iff epred-lxs epred-lys heq'
      using llength-eq-0 by blast
  qed
qed

lemma eq-cmset-alt: ‹eq-cmset lxs' lxs ⟷ (∃ π. lbij-on π lxs ∧ lpermute π lxs =
lxs')›
proof
  assume h: ‹∃ π. lbij-on π lxs ∧ lpermute π lxs = lxs'›
  then obtain π where hπ: ‹lbij-on π lxs› ‹lpermute π lxs = lxs'› by blast
  then show ‹eq-cmset lxs' lxs›
```

```
  unfolding eq-cmset-def by (auto simp: count-llist-lpermute fun-eq-iff)
next
 assume h: ‹eq-cmset lxs' lxs›
 then have hlen: ‹llength lxs = llength lxs'›
  using eq-cmset-llength[of lxs' lxs, symmetric] by blast
 then have hcount: ‹∀x. count-llist lxs' x = count-llist lxs x›
  using h unfolding eq-cmset-def by (auto simp: fun-eq-iff)
 define f where ‹f = (λi. let x = lnth lxs' i in lfind-index (count-list (ltaken i
lxs') x) x 0 lxs)›
 have hgt: ‹⋀i. enat i < llength lxs ⟹
    enat (count-list (ltaken i lxs') (lnth lxs' i)) < count-llist lxs (lnth lxs' i)›
  using hlen hcount by (metis count-list-ltaken-less)
 show ‹∃π. lbij-on π lxs ∧ lpermute π lxs = lxs'›
 proof (intro exI[of - f] conjI)
  show ‹lbij-on f lxs›
  proof (rule bij-betwI'[where f = f and X = ‹{k. enat k < llength lxs}› and
Y = ‹{k. enat k < llength lxs}›])
   fix i j
   assume hi: ‹i ∈ {k. enat k < llength lxs}› and hj: ‹j ∈ {k. enat k < llength
lxs}›
   show ‹(f i = f j) = (i = j)›
   proof
    assume hfij: ‹f i = f j›
    have hi': ‹enat (count-list (ltaken i lxs') (lnth lxs' i)) < count-llist lxs (lnth
lxs' i)›
     using hgt hi by simp
    have hj': ‹enat (count-list (ltaken j lxs') (lnth lxs' j)) < count-llist lxs (lnth
lxs' j)›
     using hgt hj by simp
    have heq: ‹lfind-index (count-list (ltaken i lxs') (lnth lxs' i)) (lnth lxs' i) 0
lxs =
          lfind-index (count-list (ltaken j lxs') (lnth lxs' j)) (lnth lxs' j) 0 lxs›
     using hfij unfolding f-def Let-def by simp
    from lfind-index-inject[OF hi' hj' heq]
    have heqx: ‹lnth lxs' i = lnth lxs' j›
     and heqn: ‹count-list (ltaken i lxs') (lnth lxs' i) = count-list (ltaken j lxs')
(lnth lxs' j)›
     by auto
    show ‹i = j›
     using hi hj hlen heqx heqn by (metis count-list-inject mem-Collect-eq)
   next
    assume ‹i = j› then show ‹f i = f j› by simp
   qed
  next
   fix i assume hi: ‹i ∈ {k. enat k < llength lxs}›
   show ‹f i ∈ {k. enat k < llength lxs}›
    unfolding f-def Let-def
    using hgt[OF] hi
    by (metis add.right-neutral lfind-index-less mem-Collect-eq zero-enat-def)
```

```
    next
      fix k assume hk: ‹k ∈ {k. enat k < llength lxs}›
      show ‹∃ i ∈ {k. enat k < llength lxs}. k = f i›
        unfolding f-def Let-def
        by (metis add.right-neutral count-list-inject count-list-ltaken-less diff-zero
hcount hk
            hlen lfind-index-count-list lfind-index-less lfind-index-lnth mem-Collect-eq
zero-enat-def)
    qed
  next
    show ‹lpermute f lxs = lxs'›
    proof (intro lnth-equalityI)
      show ‹llength (lpermute f lxs) = llength lxs'›
        by (simp add: lpermute-def hlen)
    next
      fix i
      assume hi: ‹enat i < llength (lpermute f lxs)›
      then have hi': ‹enat i < llength lxs› by (simp add: lpermute-def lnth-lupt)
      have ‹lnth (lpermute f lxs) i = lnth lxs (f i)›
        by (simp add: lpermute-def lnth-lupt hi' f-def Let-def)
      also have ‹... = lnth lxs' i›
      proof −
        have hlt: ‹enat (count-list (ltaken i lxs') (lnth lxs' i)) < count-llist lxs (lnth
lxs' i)›
          using hgt hi' by blast
        from lfind-index-lnth[OF hlt, of 0] show ?thesis
          unfolding f-def Let-def by simp
      qed
      finally show ‹lnth (lpermute f lxs) i = lnth lxs' i› .
    qed
  qed
qed

lemma eq-cmset-lzip-left:
  assumes ‹eq-cmset lxs' lxs› ‹llength lys = llength lxs›
  shows ‹∃ lys'. eq-cmset (lzip lxs' lys') (lzip lxs lys) ∧ llength lys' = llength lxs'›
proof −
  from assms(1) obtain π where ‹lbij-on π lxs› ‹lpermute π lxs = lxs'› unfolding
eq-cmset-alt
    by blast
  with assms(2) show ?thesis
   by (intro exI[of - ‹lpermute π lys›]) (auto simp: lpermute-lzip eq-cmset-alt intro!:
exI[of - π])
qed

lemma eq-cmset-lmap:
  assumes ‹eq-cmset lxs lys›
  shows ‹eq-cmset (lmap f lxs) (lmap f lys)›
proof −
```

from *assms* **obtain** π **where** ‹*lbij-on* π *lys*› ‹*lpermute* π *lys* = *lxs*› **unfolding** *eq-cmset-alt*
by *blast*
then show *?thesis*
by (*force simp*: *lpermute-def llist.map-comp lset-lupt eq-cmset-alt*
dest: *bij-betwE intro*!: *llist.map-cong exI*[**where** $x = \pi$] *lnth-lmap*)
qed

lemma *llist-all2-reorder-left-invariance*:
assumes *rel*: ‹*llist-all2 R lxs lys*› **and** *ms-x*: ‹*eq-cmset lxs' lxs*›
shows ‹$\exists$ *lys'*. *llist-all2 R lxs' lys'* $\wedge$ *eq-cmset lys' lys*›
proof −
from *ms-x rel*[*THEN llist-all2-llengthD*] **obtain** *lys'* **where**
ms-xy: ‹*eq-cmset* (*lzip lxs' lys'*) (*lzip lxs lys*)› **and** *len*: ‹*llength lys'* = *llength lxs'*›
by (*metis eq-cmset-lzip-left*)
with *rel* **have** *rel'*: ‹*llist-all2 R lxs' lys'*› **unfolding** *llist-all2-conv-lzip*
by (*auto simp*: *lset-eq-cmset*)
moreover from *eq-cmset-lmap*[*OF ms-xy*, *of snd*]
have ‹*eq-cmset* (*lmap snd* (*lzip lxs' lys'*)) (*lmap snd* (*lzip lxs lys*))› **.**
with *rel*[*THEN llist-all2-llengthD*] *rel'*[*THEN llist-all2-llengthD*] **have** ‹*eq-cmset lys' lys*›
by (*auto simp*: *lmap-snd-lzip-conv-ltake ltake-all*)
ultimately show *?thesis*
by *blast*
qed

8 Countable Multisets as a Quotient

quotient-type *'a cmset* = ‹*'a llist*› / *eq-cmset*
by (*auto simp*: *eq-cmset-def intro*!: *equivpI reflpI sympI transpI*)

lift-bnf (*cmset*: *'a*) *cmset*
for *map*: *cmimage rel*: *cmrel*
proof −
fix *P* :: ‹*'a* ⇒ *'b* ⇒ *bool*› **and** *Q* :: ‹*'b* ⇒ *'c* ⇒ *bool*›
show ‹*llist-all2 P OO eq-cmset OO llist-all2 Q*
≤ *eq-cmset OO llist-all2* (*P OO Q*) *OO eq-cmset*›
proof *safe*
fix *l r l' r'*
assume ‹*llist-all2 P l l'*› ‹*eq-cmset l' r'*› ‹*llist-all2 Q r' r*›
with *llist-all2-reorder-left-invariance*[*OF this*(*3*,*2*)]
show ‹(*eq-cmset OO llist-all2* (*P OO Q*) *OO eq-cmset*) *l r*›
by (*auto intro*!: *relcomppI*[*of - - l*] *simp*: *eq-cmset-def llist.rel-compp*)
qed
next
fix *S* :: ‹*'a set set*›
assume ‹*S* ≠ {}›
then show ‹($\bigcap$ *As*∈*S*. {(*x*, *x'*). *eq-cmset x x'*} `` {*x*. *lset x* ⊆ *As*})

$\subseteq \{(x, x').\ eq\text{-}cmset\ x\ x'\}$ `` $\{x.\ lset\ x \subseteq \bigcap S\}$›
using *Inter-greatest*[*of S* ‹*lset* -›] *lset-eq-cmset*
unfolding *subset-eq Ball-def Bex-def INT-iff Image-iff mem-Collect-eq prod.case*
by (*metis cmset.abs-eq-iff*)
qed

9 Lazy List Interleaving

context notes [[*function-internals*]]
begin
partial-function (*llist*) *lflat* **where**
‹*lflat lxss* = (*case lxss of LNil* ⇒ *LNil* | *LCons xs lxss* ⇒ *lappend* (*llist-of xs*) (*lflat lxss*))›
end

lemma *ltaken-ldropn-decomp*: ‹*lxs* = *lappend* (*llist-of* (*ltaken n lxs*)) (*ldropn n lxs*)›
by (*induct n arbitrary*: *lxs*) (*auto split*: *llist.splits*)

lemma *count-llist-llist-of*[*simp*]: ‹*count-llist* (*llist-of xs*) *x* = *count-list xs x*›
by (*induct xs*) (*auto simp*: *enat-0 simp flip*: *eSuc-enat*)

lemma *lmap-lfilter-swap*: ‹∀ *x* ∈ *lset lxs*. *P x* ⟷ *Q* (*f x*) ⟹ *lmap f* (*lfilter P lxs*) = *lfilter Q* (*lmap f lxs*)›
by (*induct lxs*) *auto*

lemma *lsum-lmap-zero*: ‹*lsum* (*lmap* (λ*z*. *0*) *lxs*) = *0*›
proof −
have ‹*lset* (*lmap* (λ*z*. *0*::*enat*) *lxs*) ⊆ {*0*}› **by** *fastforce*
then have ‹*ldropWhile* ((=) *0*) (*lmap* (λ*z*. *0*) *lxs*) = *LNil*›
by (*simp add*: *ldropWhile-eq-LNil-iff*)
then show *?thesis*
by (*metis llist.case*(*1*) *lsum-code-alt*)
qed

lemma *lsum-LNil*[*simp*]: ‹*lsum LNil* = *0*›
by (*simp add*: *lsum-code-alt*)

lemma *lsum-LCons*[*simp*]: ‹*lsum* (*LCons x lxs*) = *x* + *lsum lxs*›
by (*simp add*: *lsum-code-alt*)

lemma *count-llist-lmap-const*:
‹*count-llist* (*lmap* (λ*z*. *a*) *lxs*) *x* = (*if x* = *a then llength lxs else 0*)›
unfolding *count-llist-alt*
by (*auto simp*: *lfilter-lmap*)

lemma *lsum-lmap-all-but-one-0*: ‹*x* ∈ *lset lxs* ⟹ *ldistinct lxs* ⟹ *lsum* (*lmap* (λ*z*. *if z* = *x then y else 0*) *lxs*) = *y*›
proof (*induct x lxs rule*: *llist.set-induct*)

case (*LCons1 x′ lxs*)
then have ‹*lmap* (λ*z*. *if z* = *x′ then y else 0*) *lxs* = *lmap* (λ-. *0*) *lxs*›
by (*intro llist.map-cong*) *auto*
then show *?case*
by (*auto simp*: *lsum-lmap-zero*)
next
case (*LCons2 x′ lxs x*)
then show *?case*
by *auto*
qed

lemma *ltaken-Suc*:
‹*ltaken* (*Suc i*) *lxs* = (*if i* < *llength lxs then ltaken i lxs* @ [*lnth lxs i*] *else ltaken i lxs*)›
proof (*induct i arbitrary*: *lxs*)
case *0*
then show *?case*
by (*auto simp*: *enat-0 Suc-ile-eq split*: *if-splits llist.splits*)
next
case (*Suc i*)
then show *?case*
by (*cases lxs*) (*auto simp*: *enat-0 Suc-ile-eq split*: *if-splits llist.splits*)
qed

lemma *ltaken-all-same*: ‹*enat n* ≥ *llength lxs* ⟹ *enat m* ≥ *llength lxs* ⟹ *ltaken n lxs* = *ltaken m lxs*›
by (*metis lappend-LNil2 ldropn-eq-LNil llist-of-inject ltaken-ldropn-decomp*)

lemma *lsum-mono*: ‹*lprefix lxs lys* ⟹ *lsum lxs* ≤ *lsum lys*›
proof −
assume ‹*lprefix lxs lys*›
show ‹*lsum lxs* ≤ *lsum lys*›
proof (*cases* ‹*lfinite lxs*›)
case *True*
then show *?thesis*
using ‹*lprefix lxs lys*› ‹*lfinite lxs*›
by (*induct lxs arbitrary*: *lys rule*: *lfinite.induct*) (*auto simp*: *LCons-lprefix-conv*)
next
case *False*
then show *?thesis*
using ‹*lprefix lxs lys*›
by (*simp add*: *not-lfinite-lprefix-conv-eq*)
qed
qed

lemma *Sup-enat-remove0*: ‹∃ *x* ∈ *X*. *x* > *0* ⟹ *Sup* (*X* :: *enat set*) = *Sup* (*X* − {*0*})›
proof −
assume *ex*: ‹(∃ *x* ∈ *X*. *x* > *0*)›

```
  show ‹Sup (X :: enat set) = Sup (X − {0})›
    unfolding Sup-enat-def
     by (smt (verit, ccfv-threshold) Diff-empty Diff-insert0 Max.remove empty-iff
finite.emptyI
        finite-Diff-insert max-enat-simps(3))
qed

lemma lSup-eq-lappend: ‹Complete-Partial-Order.chain lprefix Y ⟹ ∃ lys ∈ Y.
lprefix (llist-of xs) lys ⟹
  lSup Y = lappend (llist-of xs) (lSup (ldropn (length xs) ‘ Y))›
proof (induct xs arbitrary: Y)
  case (Cons a xs)
  from Cons(3) obtain lys where lys: ‹¬ lnull lys› ‹lys ∈ Y› ‹lhd lys = a› ‹lprefix
(llist-of xs) (ltl lys)›
    by (metis eq-LConsD llist-of.simps(2) llist-of-eq-LNil-conv lnull-llist-of lprefix.cases
        lprefix-not-lnullD)
  from Cons(2−) have lhd-Y: ‹lys ∈ Y ⟹ ¬ lnull lys ==> lhd lys = a› for lys
    by (metis eq-LConsD lhd-lSup-eq llist.disc(2) llist-of.simps(2) lprefix-lhdD
        lprefix-not-lnullD)
  from Cons(2−) have lhd-lSup[simp]: ‹lhd (lSup Y) = a›
   by (metis lhd-LCons lhd-lSup-eq llist.disc(2) llist-of.simps(2) lprefix-lhdD lprefix-lnull)
  moreover have ‹lSup (ltl ‘ (Y ∩ {l. ¬ lnull l})) =
          lappend (llist-of xs) (lSup (ldropn (Suc (length xs)) ‘ Y))›
  proof −
    have ‹ldropn (Suc (length xs)) ‘ (Y ∩ {l. ¬ lnull l}) ⊆ ldropn (Suc (length xs))
‘ Y›
      (is ‹?lhs ⊆ ?rhs›) by auto
    moreover have ‹?rhs − {LNil} ⊆ ?lhs›
      by auto
    ultimately have ‹lSup ?lhs = lSup ?rhs›
      by (metis Int-Diff inf.absorb-iff2 inf.order-iff lSup-minus-LNil)
    moreover have chain-ltl: ‹Complete-Partial-Order.chain lprefix (ltl ‘ (Y ∩ {l.
¬ lnull l}))›
      by (metis chain-lprefix-ltl Cons.prems(1))
    moreover have ‹Bex (ltl ‘ (Y ∩ {l. ¬ lnull l})) (lprefix (llist-of xs))›
      by (auto intro!: bexI[of - lys] simp: lys)
    ultimately show ?thesis
      by (subst Cons(1)) (auto simp: image-image ldropn-ltl)
  qed
  ultimately show ?case
    by (subst lSup.code)
      (auto simp: LCons-lprefix-conv lhd-Y lys intro!: the1-equality bexI[of - lys])
qed simp

lemma lsum-0D: ‹x ∈ lset lxs ⟹ lsum lxs = 0 ⟹ x = 0›
  by (induct x lxs rule: llist.set-induct) auto

lemma lsum-0I: ‹∀ x∈lset lxs. x = 0 ⟹ lsum lxs = 0›
  by (subst lsum-code-alt) auto
```

lemma *lsum-0-iff*: ‹*lsum lxs = 0* ⟷ (∀ *x* ∈ *lset lxs*. *x = 0*)›
by (*metis lsum-0I lsum-0D*)

lemma *lsum-lappend-lfinite*: ‹*lfinite lxs* ⟹ *lsum* (*lappend lxs lys*) = *lsum lxs* + *lsum lys*›
by (*induct lxs rule*: *lfinite.induct*) *auto*

lemma *lsum-lappend*: ‹*lsum* (*lappend lxs lys*) = (*if lfinite lxs then lsum lxs* + *lsum lys else lsum lxs*)›
by (*auto simp*: *lappend-inf lsum-lappend-lfinite*)

lemma *lsum-llist-of*[*simp*]: ‹*lsum* (*llist-of xs*) = *sum-list xs*›
by (*induct xs*) *auto*

lemma *lsum-cont*:
‹*Complete-Partial-Order.chain lprefix Y* ⟹ *Y* ≠ {} ⟹ *lsum* (*lSup Y*) = ⨆ (*lsum* ‘ *Y*)›
proof (*coinduction arbitrary*: *Y rule*: *enat-coinduct*)
case *Eq-enat*
then show *?case*
proof (*intro allI impI conjI disjI1*)
show ‹*Complete-Partial-Order.chain lprefix Y* ⟹ *Y* ≠ {} ⟹ (*lsum* (*lSup Y*) = *0*) = (⨆ (*lsum* ‘ *Y*) = *0*)›
by (*simp add*: *Sup-eq-0-iff lset-lSup lsum-0-iff*)
next
assume *hn0-lsum*: ‹*lsum* (*lSup Y*) ≠ *0*› **and** *hn0-Sup*: ‹⨆ (*lsum* ‘ *Y*) ≠ *0*›
show ‹(∃ *Y'*. *epred* (*lsum* (*lSup Y*)) = *lsum* (*lSup Y'*) ∧ *epred* (⨆ (*lsum* ‘ *Y*)) = ⨆ (*lsum* ‘ *Y'*) ∧
Complete-Partial-Order.chain lprefix Y' ∧ *Y'* ≠ {})›
proof −
have *lnth-eq-Y*: ‹*lnth lxs i* = *lnth lys i*›
if ‹*enat i* < *llength lxs*› ‹*lxs* ∈ *Y*› ‹*enat i* < *llength lys*› ‹*lys* ∈ *Y*› **for** *lxs lys i*
using *Eq-enat*(*1*) *that* **by** (*force simp*: *chain-def dest*: *lprefix-lnthD*[*of - - i*])
obtain *lys z* **where** *hlys*: ‹*lys* ∈ *Y*› **and** *hz*: ‹*z* ∈ *lset lys*› ‹*z* ≠ *0*›
using *hn0-lsum Eq-enat*(*1*) **by** (*auto simp*: *lsum-0-iff lset-lSup*)
define *A* **where** ‹*A* = {*i*. ∀ *lys* ∈ *Y*. *ltaken i lys* = *replicate* (*case llength lys of enat j* ⇒ *min i j* | - ⇒ *i*) *0*}›
define *i* **where** ‹*i* = *Max A*›
have *ne*: ‹*A* ≠ {}›
using *Eq-enat*(*2*) ‹*lys* ∈ *Y*› **by** (*auto simp*: *A-def enat-0 intro*!: *exI*[*of - 0*] *split*: *enat.splits*)
have *fin*: ‹*finite A*›
proof −
from *hn0-lsum Eq-enat*(*1*) **obtain** *lys'* **where** *hlys'*: ‹*lys'* ∈ *Y*› **and** *hn0*: ‹*lsum lys'* ≠ *0*›
by (*auto simp*: *lset-lSup lsum-0-iff*)
from *hn0* **obtain** *n* **where** *hn*: ‹*enat n* < *llength lys'*› ‹*lnth lys' n* ≠ *0*›

```
      by (auto simp: lsum-0-iff in-lset-conv-lnth dest: lsum-0D)
    show ‹finite A›
    proof (rule finite-subset[of - ‹{0..n}›])
      show ‹A ⊆ {0..n}›
      proof
        fix k assume hk: ‹k ∈ A›
       have hkprop: ‹ltaken k lys' = replicate (case llength lys' of enat j ⇒ min
k j | - ⇒ k) 0›
          using hk hlys' by (auto simp: A-def)
        have ‹n < k ⟹ False›
        proof −
          assume hnk: ‹n < k›
          have ‹nth (ltaken k lys') n = lnth lys' n›
          proof (cases ‹enat k ≤ llength lys'›)
            case True
            thus ?thesis using hnk by (simp add: nth-ltaken)
          next
            case False
            hence hlt: ‹llength lys' < enat k› by simp
            then obtain m where hm: ‹llength lys' = enat m›
              by (cases ‹llength lys'›) (auto simp: enat-def)
            have hmk: ‹m ≤ k› using hlt hm by auto
            have hnm: ‹n < m› using hn(1) hm by auto
            have ‹ltaken k lys' = ltaken m lys'›
              by (rule ltaken-all-same) (simp-all add: hm hmk)
            thus ?thesis using hnm hm by (simp add: nth-ltaken)
          qed
          moreover have ‹nth (replicate (case llength lys' of enat j ⇒ min k j |
- ⇒ k) 0) n = 0›
            using hn(1) hnk by (auto simp: min-def split: enat.splits)
          ultimately show False using hkprop hn(2) by metis
        qed
        thus ‹k ∈ {0..n}› by (force simp: not-less)
      qed
    qed simp
  qed
  have iA: ‹∀ lys ∈ Y. ltaken i lys = replicate (case llength lys of enat j ⇒ min
i j | - ⇒ i) 0›
    using Max-in[OF fin ne] unfolding i-def A-def by blast
  from iA[THEN bspec, of lys] hlys ‹z ∈ lset lys› ‹z ≠ 0› have hi: ‹enat i <
llength lys›
  proof −
    have hiA: ‹ltaken i lys = replicate (case llength lys of enat j ⇒ min i j | -
⇒ i) 0›
      using iA hlys by blast
    obtain n0 where hn0-len: ‹enat n0 < llength lys› and hn0-nth: ‹lnth lys
n0 = z›
      using ‹z ∈ lset lys› by (auto simp: in-lset-conv-lnth)
    show ‹enat i < llength lys›
```

```
      proof (rule ccontr)
        assume h: ‹¬ enat i < llength lys›
        hence hfin: ‹llength lys ≠ ∞› by (cases ‹llength lys›) auto
       then obtain m where hm: ‹llength lys = enat m› by (cases ‹llength lys›)
auto
        from h hm have him: ‹m ≤ i› by auto
        have hn0m: ‹n0 < m› using hn0-len hm by auto
        have ‹nth (ltaken i lys) n0 = lnth lys n0›
        proof (cases ‹enat i ≤ llength lys›)
          case True
          thus ?thesis using hn0m him hm by (simp add: nth-ltaken)
        next
          case False
          hence ‹ltaken i lys = ltaken m lys›
            by (rule-tac ltaken-all-same) (simp-all add: hm him)
          thus ?thesis using hn0m hm by (simp add: nth-ltaken)
        qed
        moreover have ‹nth (ltaken i lys) n0 = 0›
          using hiA hm him hn0m by (auto simp: min-def)
        ultimately show False using hn0-nth ‹z ≠ 0› by auto
      qed
    qed
    with iA hlys have hpref: ‹lprefix (llist-of (replicate i 0)) lys›
    by (metis length-ltaken length-replicate lprefix-conv-lappend ltaken-ldropn-decomp
nless-le)
    obtain n where hn: ‹∀ lys ∈ Y. enat i ≥ llength lys ∨ lnth lys i = n›
      using lnth-eq-Y by (meson linorder-not-le)
    have hn0: ‹n = 0 ⟹ Suc i ∈ A›
    proof −
      assume hn0-eq: ‹n = 0›
      show ‹Suc i ∈ A›
        unfolding A-def mem-Collect-eq
      proof
        fix lys′ assume hlys′-Y: ‹lys′ ∈ Y›
         show ‹ltaken (Suc i) lys′ = replicate (case llength lys′ of enat j ⇒ min
(Suc i) j | - ⇒ Suc i) 0›
        proof (cases ‹enat i < llength lys′›)
          case True
          hence hstep: ‹ltaken (Suc i) lys′ = ltaken i lys′ @ [lnth lys′ i]›
            by (metis ltaken-Suc)
          have ‹lnth lys′ i = 0›
            using hn hlys′-Y hn0-eq True by auto
          moreover have ‹ltaken i lys′ = replicate (case llength lys′ of enat j ⇒
min i j | - ⇒ i) 0›
            using iA hlys′-Y by blast
          ultimately show ?thesis
            using hstep ‹enat i < llength lys′›
              by (subst hstep, auto simp: min-def replicate-append-same Suc-ile-eq
split: enat.splits)
```

```
        next
          case False
          hence ‹ltaken (Suc i) lys' = ltaken i lys'›
            by (metis ltaken-Suc)
           moreover have ‹ltaken i lys' = replicate (case llength lys' of enat j ⇒
min i j | - ⇒ i) 0›
            using iA hlys'-Y by blast
          ultimately show ?thesis
            using False
            by (auto simp: min-def not-less split: enat.splits)
        qed
      qed
    qed
    have hnn: ‹n ≠ 0›
     using Max-eq-iff[OF fin ne, of i, THEN iffD1, OF i-def[symmetric], THEN
conjunct2, rule-format, of ‹Suc i›]
       hn0 by fastforce
    have ldropn-cases: ‹lys = LNil ∨ (∃ lys'. lys = LCons n lys')›
      if ‹lys ∈ ldropn i ‘ Y› for lys
      using that hn
    by (metis (mono-tags, lifting) image-iff ldropn-Suc-conv-ldropn ldropn-eq-LNil
linorder-not-le)
    have hTHE: ‹(THE x. x ∈ lhd ‘ (ldropn i ‘ Y ∩ {xs. ¬ lnull xs})) = n›
    proof (intro the-equality)
      show ‹n ∈ lhd ‘ (ldropn i ‘ Y ∩ {xs. ¬ lnull xs})›
           by (metis IntI hi hlys image-eqI ldropn-Suc-conv-ldropn lhd-LCons
linorder-not-le
          llist.disc(2) mem-Collect-eq hn)
      show ‹⋀x. x ∈ lhd ‘ (ldropn i ‘ Y ∩ {xs. ¬ lnull xs}) ⟹ x = n›
      proof −
        fix x assume ‹x ∈ lhd ‘ (ldropn i ‘ Y ∩ {xs. ¬ lnull xs})›
       then obtain lxs where hlxs: ‹lxs ∈ Y› ‹¬ lnull (ldropn i lxs)› ‹x = lhd
(ldropn i lxs)›
          by auto
        from hlxs(2) have hlen: ‹enat i < llength lxs›
          by (simp add: lnull-def ldropn-eq-LNil not-less)
        from hlxs(3) have ‹x = lnth lxs i›
          by (simp add: lhd-ldropn hlen)
        with hn hlxs(1) hlen show ‹x = n›
          by auto
      qed
    qed
    have hLCons: ‹(λx. LCons (epred n) (ltl x)) ‘ X ∩ {xs. ¬ lnull xs} =
        (λx. LCons (epred n) (ltl x)) ‘ X› for X
      by auto
    have h4: ‹ldropn i lys ∈ ldropn i ‘ Y ∩ {xs. ¬ lnull xs}›
      using hi hlys by auto
   define Y' where ‹Y' = (LCons (epred n) ∘ ltl) ‘ (ldropn i ‘ Y ∩ {xs. ¬ lnull
xs})›
```

```
    show ?thesis
    proof (intro disjI1 exI[of - Y'] conjI)
      show ‹Y' ≠ {}›
        unfolding Y'-def using h4 by force
    next
      show ‹Complete-Partial-Order.chain lprefix Y'›
        unfolding Y'-def chain-def
      proof (intro ballI)
        fix lxs' lys'
       assume hm1: ‹lxs' ∈ (LCons (epred n) ∘ ltl) ‘ (ldropn i ‘ Y ∩ {xs. ¬ lnull
xs})›
          and hm2: ‹lys' ∈ (LCons (epred n) ∘ ltl) ‘ (ldropn i ‘ Y ∩ {xs. ¬ lnull
xs})›
        from hm1 obtain lxs0 where hlxs0: ‹lxs0 ∈ ldropn i ‘ Y› ‹¬ lnull lxs0›
          ‹lxs' = LCons (epred n) (ltl lxs0)› by auto
        from hm2 obtain lys0 where hlys0: ‹lys0 ∈ ldropn i ‘ Y› ‹¬ lnull lys0›
          ‹lys' = LCons (epred n) (ltl lys0)› by auto
          from hlxs0(1) obtain lxs1 where hlxs1: ‹lxs1 ∈ Y› ‹lxs0 = ldropn i
lxs1› by auto
          from hlys0(1) obtain lys1 where hlys1: ‹lys1 ∈ Y› ‹lys0 = ldropn i
lys1› by auto
         from Eq-enat(1) hlxs1(1) hlys1(1) have ‹lprefix lxs1 lys1 ∨ lprefix lys1
lxs1›
          by (auto simp: chain-def)
        thus ‹lprefix lxs' lys' ∨ lprefix lys' lxs'›
          using hlxs0(3) hlys0(3) hlxs1(2) hlys1(2)
          by (auto intro: lprefix-ltlI monotone-ldropn'[THEN monotoneD])
      qed
    next
      let ?S = ‹ldropn i ‘ Y ∩ {xs. ¬ lnull xs}›
      have hS-ne: ‹?S ≠ {}› using h4 by force
      have lSup-ldropn: ‹lSup (ldropn i ‘ Y) = LCons n (lSup (ltl ‘ ?S))›
      proof (subst lSup.code, auto simp: hTHE h4)
        show ‹∃x∈Y. ¬ llength x ≤ enat i›
          using hlys hi by (intro bexI[OF - hlys]) auto
      qed
      have lSup-Y': ‹lSup Y' = LCons (epred n) (lSup (ltl ‘ ?S))›
      proof −
        have hne: ‹ltl ‘ ?S ≠ {}› using hS-ne by auto
        have ‹Y' = LCons (epred n) ‘ ltl ‘ ?S›
          by (simp add: Y'-def image-image)
        thus ?thesis using hne by (simp add: lSup-LCons)
      qed
      have lSup-Y-eq: ‹lSup Y = lappend (llist-of (replicate i 0)) (lSup (ldropn i
‘ Y))›
        using lSup-eq-lappend[OF Eq-enat(1), of ‹replicate i 0›] hlys hpref by
auto
      show ‹epred (lsum (lSup Y)) = lsum (lSup Y')›
        using lSup-Y-eq lSup-ldropn lSup-Y' hnn
```

```
      by (simp add: lsum-lappend sum-list-replicate epred-iadd1)
    next
      let ?S = ‹ldropn i ‘ Y ∩ {xs. ¬ lnull xs}›
      have hS-ne: ‹?S ≠ {}› using h4 by force
      have lSup-Y': ‹lSup Y' = LCons (epred n) (lSup (ltl ‘ ?S))›
      proof −
        have hne: ‹ltl ‘ ?S ≠ {}› using hS-ne by auto
        have ‹Y' = LCons (epred n) ‘ ltl ‘ ?S›
          by (simp add: Y'-def image-image)
        thus ?thesis using hne by (simp add: lSup-LCons)
      qed
      have lsum-lzs: ‹lsum lzs = lsum (ldropn i lzs)› if ‹lzs ∈ Y› for lzs
        using iA[rule-format, OF that]
        by (subst ltaken-ldropn-decomp[of - i]) (auto simp: lsum-lappend)
      have lsum-eq-sets: ‹(epred ∘ lsum) ‘ (Y ∩ {lzs. lsum lzs ≠ 0}) =
                          lsum ‘ Y'›
      proof (rule set-eqI)
        fix v
       show ‹v ∈ (epred ∘ lsum) ‘ (Y ∩ {lzs. lsum lzs ≠ 0}) ⟷ v ∈ lsum ‘ Y'›
        proof
          assume ‹v ∈ (epred ∘ lsum) ‘ (Y ∩ {lzs. lsum lzs ≠ 0})›
         then obtain lzs where hlzs: ‹lzs ∈ Y› ‹lsum lzs ≠ 0› ‹v = epred (lsum
lzs)› by auto
          have hnd: ‹ldropn i lzs ≠ LNil›
            using hlzs(1,2) lsum-lzs by (auto simp: lsum-0-iff)
          then obtain lzs' where hlzs': ‹ldropn i lzs = LCons n lzs'›
            using ldropn-cases[of ‹ldropn i lzs›] hlzs(1) by auto
          have ‹LCons (epred n) (ltl (ldropn i lzs)) ∈ Y'›
            using hlzs(1) hnd by (auto simp: Y'-def)
          moreover have ‹v = lsum (LCons (epred n) (ltl (ldropn i lzs)))›
            using hlzs(3) hlzs'(1) lsum-lzs[OF hlzs(1)] hnn
            by (simp add: epred-iadd1)
          ultimately show ‹v ∈ lsum ‘ Y'›
            by blast
        next
          assume ‹v ∈ lsum ‘ Y'›
          then obtain ya where hya: ‹ya ∈ Y'› ‹v = lsum ya› by auto
          from hya(1) obtain lzs where hlzs: ‹lzs ∈ Y› ‹ldropn i lzs ∈ ?S›
            ‹ya = LCons (epred n) (ltl (ldropn i lzs))› by (auto simp: Y'-def)
          from hlzs(2) have hnd: ‹¬ lnull (ldropn i lzs)› by auto
          then obtain lzs' where hlzs': ‹ldropn i lzs = LCons n lzs'›
            using ldropn-cases[of ‹ldropn i lzs›] hlzs(1) by fastforce
          have ‹lsum lzs = n + lsum lzs'›
            using lsum-lzs[OF hlzs(1)] hlzs' by simp
          hence ‹lsum lzs ≠ 0› using hnn by auto
          moreover have ‹v = epred (lsum lzs)›
            using hya(2) hlzs(3) hlzs' lsum-lzs[OF hlzs(1)] hnn
            by (simp add: epred-iadd1)
          ultimately show ‹v ∈ (epred ∘ lsum) ‘ (Y ∩ {lzs. lsum lzs ≠ 0})›
```

```
          using hlzs(1) by auto
        qed
      qed
      show ‹epred (⨆ (lsum ‘ Y)) = ⨆ (lsum ‘ Y')›
      proof −
        have ‹⨆ (lsum ‘ Y) = ⨆ (lsum ‘ (Y ∩ {lzs. lsum lzs ≠ 0}))›
          using hlys hz(1,2)
         by (subst Sup-enat-remove0) (auto dest: lsum-0D intro!: arg-cong[where
f=Sup])
        moreover have ‹epred (⨆ (lsum ‘ (Y ∩ {lzs. lsum lzs ≠ 0}))) =
                      ⨆ (epred ‘ (lsum ‘ (Y ∩ {lzs. lsum lzs ≠ 0})))›
          by (rule epred-Sup)
        moreover have ‹epred ‘ (lsum ‘ (Y ∩ {lzs. lsum lzs ≠ 0})) =
                      (epred ∘ lsum) ‘ (Y ∩ {lzs. lsum lzs ≠ 0})›
          by (simp add: image-image)
        ultimately show ?thesis
          using lsum-eq-sets by simp
      qed
    qed
  qed
qed

lemma mcont2mcont-lsum[THEN lfp.mcont2mcont, simp, cont-intro]:
  shows mcont-lsum: ‹mcont lSup (lprefix) Sup (≤) lsum›
  by (auto simp: mcont-def monotone-def lsum-mono cont-def lsum-cont)


lemma lsum-lfilter-nonzero: ‹lsum (lfilter ((≠) 0) lxs) = lsum lxs›
  by (induct lxs) auto

abbreviation ‹LSUM lxs f ≡ lsum (lmap f lxs)›

abbreviation ‹lenats ≡ lupt 0 ∞›

lemma LSUM-lfilter: ‹LSUM (lfilter (λx. f x ≠ 0) lxs) f = LSUM lxs f›
  by (induct lxs) auto

lemma LSUM-count-llist-lfilter:
  ‹lsum (lmap (λlxs. count-llist lxs x) (lfilter (λx. ¬ lnull x) lxss)) =
  lsum (lmap (λlxs. count-llist lxs x) lxss)›
proof (induct lxss)
  case LNil
  then show ?case by (auto simp: count-llist-zero-iff)
next
  case (LCons lxs lxss)
   then show ?case by (auto simp: count-llist-zero-iff, metis lnull-def empty-iff
lset-LNil)
qed simp
```

lemma *lflat-LNil*[*simp*]: ‹*lflat LNil = LNil*›
by (*subst lflat.simps*) *auto*

lemma *lflat-LCons*[*simp*]: ‹*lflat* (*LCons xs lxss*) = *lappend* (*llist-of xs*) (*lflat lxss*)›
by (*subst lflat.simps*) *auto*

lemma *count-llist-mono*:
assumes ‹*lprefix lxs lys*›
shows ‹*count-llist lxs x* ≤ *count-llist lys x*›
proof (*cases* ‹*lfinite lxs*›)
case *True*
then show *?thesis* **using** *assms*
by (*induct lxs arbitrary*: *lys pred*: *lfinite*) (*auto simp*: *LCons-lprefix-conv*)
next
case *False*
then show *?thesis* **using** *assms*
by (*simp add*: *not-lfinite-lprefix-conv-eq*)
qed

lemma *ldropWhile-not-lnull-alt*:
‹*ldropWhile* ((≠) *x*) ‘ *Y* ∩ {*lxs*. ¬ *lnull lxs*} = *ldropWhile* ((≠) *x*) ‘ {*lxs* ∈ *Y*. *x* ∈ *lset lxs*}›
by *auto*

lemma *count-llist-cont*: ‹*Complete-Partial-Order.chain lprefix Y* ⟹ *Y* ≠ {} ⟹
count-llist (*lSup Y*) *x* = (⨆ *lxs*∈*Y*. *count-llist lxs x*)›
proof (*coinduction arbitrary*: *Y*)
case (*eq-enat Y*)
then have *chain-Y*: ‹*Complete-Partial-Order.chain lprefix Y*› **and** *ne-Y*: ‹*Y* ≠ {}› **by** *auto*
show *?case*
proof (*rule conjI*)
from *chain-Y ne-Y* **show** ‹(*count-llist* (*lSup Y*) *x* = *0*) = ((⨆ *lxs*∈*Y*. *count-llist lxs x*) = *0*)›
by (*auto simp*: *count-llist-zero-iff Sup-eq-0-iff lset-lSup*)
next
show ‹*count-llist* (*lSup Y*) *x* ≠ *0* ⟶ (⨆ *lxs*∈*Y*. *count-llist lxs x*) ≠ *0* ⟶
(∃ *Z*. *epred* (*count-llist* (*lSup Y*) *x*) = *count-llist* (*lSup Z*) *x* ∧
epred (⨆ *lxs*∈*Y*. *count-llist lxs x*) = (⨆ *lxs*∈*Z*. *count-llist lxs x*) ∧
Complete-Partial-Order.chain lprefix Z ∧ *Z* ≠ {}) ∨
epred (*count-llist* (*lSup Y*) *x*) = *epred* (⨆ *lxs*∈*Y*. *count-llist lxs x*)›
proof (*intro impI disjI1*)
assume *h1*: ‹*count-llist* (*lSup Y*) *x* ≠ *0*› **and** *h2*: ‹(⨆ *lxs*∈*Y*. *count-llist lxs x*) ≠ *0*›
define *Z* **where** ‹*Z* ≡ *ltl* ‘ (*ldropWhile* ((≠) *x*) ‘ *Y* ∩ {*lxs*. ¬ *lnull lxs*})›
have *chain-Z*: ‹*Complete-Partial-Order.chain lprefix Z*›
unfolding *Z-def* **using** *chain-Y*
by (*auto simp*: *chain-def ldropWhile-not-lnull-alt*

```
      intro!: lprefix-ltlI mcont-monoD[OF mcont-ldropWhile])
  have ne-Yx: ‹{lxs∈Y. x∈lset lxs} ≠ {}›
   using chain-Y ne-Y h1 by (auto simp: lset-lSup count-llist-zero-iff)
  have ne-Z: ‹Z ≠ {}›
   unfolding Z-def
   using ne-Yx by (auto simp: ldropWhile-not-lnull-alt)
  have x-lset: ‹x ∈ lset (lSup Y)›
   using h1 by (simp add: count-llist-zero-iff)
  have drop-cont: ‹ldropWhile ((≠) x) (lSup Y) = lSup (ldropWhile ((≠) x) ‘
Y)›
   using chain-Y ne-Y by (rule mcont-contD[OF mcont-ldropWhile])
  have eq1: ‹epred (count-llist (lSup Y) x) = count-llist (lSup Z) x›
  proof −
   have ‹epred (count-llist (lSup Y) x) = count-llist (ltl (ldropWhile ((≠) x)
(lSup Y))) x›
    using x-lset by (rule count-llist-sel)
   also have ‹ltl (ldropWhile ((≠) x) (lSup Y)) = lSup Z›
    by (simp add: drop-cont image-image Z-def)
   finally show ?thesis .
  qed
  have eq2: ‹epred (⨆lxs∈Y. count-llist lxs x) = (⨆lxs∈Z. count-llist lxs x)›
  proof −
   have aux: ‹(⨆lxs∈Y. epred (count-llist lxs x)) = (⨆lxs∈{lxs∈Y. x∈lset
lxs}. count-llist (ltl (ldropWhile ((≠) x) lxs)) x)›
    proof (rule antisym)
     show ‹(⨆lxs∈Y. epred (count-llist lxs x)) ≤ (⨆lxs∈{lxs∈Y. x∈lset lxs}.
count-llist (ltl (ldropWhile ((≠) x) lxs)) x)›
      proof (rule cSUP-mono[OF ne-Y])
       show ‹bdd-above ((λlxs. count-llist (ltl (ldropWhile ((≠) x) lxs)) x) ‘
{lxs∈Y. x∈lset lxs})›
        by (auto simp: bdd-above-def)
       fix lxs assume lxs-Y: ‹lxs ∈ Y›
       show ‹∃ya ∈ {lxs∈Y. x∈lset lxs}. epred (count-llist lxs x) ≤ count-llist
(ltl (ldropWhile ((≠) x) ya)) x›
        proof (cases ‹x ∈ lset lxs›)
         case True
         with lxs-Y show ?thesis by auto
        next
         case False
         with ne-Yx obtain ya where ya-Y: ‹ya ∈ Y› and x-ya: ‹x ∈ lset ya›
by blast
        from False have ‹count-llist lxs x = 0› by (simp add: count-llist-zero-iff)
         then show ?thesis using ya-Y x-ya by auto
        qed
      qed
    show ‹(⨆lxs∈{lxs∈Y. x∈lset lxs}. count-llist (ltl (ldropWhile ((≠) x) lxs))
x) ≤ (⨆lxs∈Y. epred (count-llist lxs x))›
      proof (rule cSUP-mono[OF ne-Yx])
       show ‹bdd-above ((λlxs. epred (count-llist lxs x)) ‘ Y)›
```

```
            by (auto simp: bdd-above-def)
          fix lxs assume ‹lxs ∈ {lxs∈Y. x∈lset lxs}›
          then have lxs-Y: ‹lxs ∈ Y› and x-lxs: ‹x ∈ lset lxs› by auto
            show ‹∃ m ∈ Y. count-llist (ltl (ldropWhile ((≠) x) lxs)) x ≤ epred
(count-llist m x)›
            using lxs-Y x-lxs
            by (auto intro!: bexI[of - lxs])
        qed
      qed
      then show ?thesis
        by (simp add: epred-Sup image-image ldropWhile-not-lnull-alt Z-def)
    qed
    show ‹∃ Z. epred (count-llist (lSup Y) x) = count-llist (lSup Z) x ∧
        epred (⨆ lxs∈Y. count-llist lxs x) = (⨆ lxs∈Z. count-llist lxs x) ∧
        Complete-Partial-Order.chain lprefix Z ∧ Z ≠ {}›
      by (rule exI[of - Z]) (use eq1 eq2 chain-Z ne-Z in simp)
  qed
 qed
qed

lemma mcont2mcont-count-llist[THEN lfp.mcont2mcont, simp, cont-intro]:
 shows mcont-count-llist: ‹mcont lSup (lprefix) Sup (≤) (λlxs. count-llist lxs x)›
 by (auto simp: mcont-def cont-def monotone-def count-llist-mono count-llist-cont)

lemma mono2mono-lflat[THEN llist.mono2mono, simp, cont-intro]:
 shows mono-lflat: ‹monotone lprefix lprefix lflat›
 by (rule llist.fixp-preserves-mono1[OF lflat.mono lflat-def]) simp

lemma mcont2mcont-lflat[THEN llist.mcont2mcont, simp, cont-intro]:
 shows mcont-lflat: ‹mcont lSup lprefix lSup lprefix lflat›
 by (rule llist.fixp-preserves-mcont1[OF lflat.mono lflat-def]) simp

lemma lflat-LNil-iff: ‹lflat lxss = LNil ⟷ (∀ xs ∈ lset lxss. xs = [])›
 by (induct lxss) (auto simp: llist-of-eq-LNil-conv lappend-eq-LNil-iff)

lemma count-llist-lflat: ‹count-llist (lflat lxss) x = lsum (lmap (λxs. count-list xs
x) lxss)›
 by (induct lxss) (auto simp: count-llist-lappend)

lemma lset-lflat: ‹lset (lflat lxss) = (⋃ xs ∈ lset lxss. set xs)›
 unfolding in-lset-iff-count-llist set-eq-iff
 by (auto simp: count-llist-zero-iff count-llist-lflat lsum-0-iff count-list-0-iff enat-0-iff)

definition ‹lvertical lxss i = (if enat i < llength lxss then ltaken (Suc i) (lnth lxss
i) else [])›
definition ‹lhorizontal lxss j = List.map-filter (λlxs. if enat j < llength lxs then
Some (lnth lxs j) else None) (ltaken j lxss)›
definition lmerge where
 ‹lmerge lxss = lflat (lmap (λi. lvertical lxss i @ lhorizontal lxss i) (lupt 0 ∞))›
```

```
lemma set-ltaken-conv': ‹set (ltaken n lxs) = (case llength lxs of enat m ⇒ lnth
lxs ‘ {0..<min n m} | - ⇒ lnth lxs ‘ {0 ..< n})›
proof −
  show ?thesis
  proof −
    { fix m x
      assume ‹m < n› ‹llength lxs = enat m› ‹x ∈ set (ltaken n lxs)›
      then have ‹x ∈ lnth lxs ‘ {0..<m}›
      by (metis enat-ord-simps(1) linorder-not-le ltaken-all-same nle-le set-ltaken-conv)
    }
    moreover
    { fix m l
      assume ‹m < n› ‹llength lxs = enat m› ‹l < m›
      then have ‹lnth lxs l ∈ set (ltaken n lxs)›
      by (metis enat-ord-simps(2) in-lset-conv-lnth ltaken-ldropn-decomp linorder-not-le
            lset-llist-of order-le-less ldropn-eq-LNil lappend-LNil2)
    }
    ultimately show ?thesis
      by (auto simp: set-ltaken-conv min-def not-le split: enat.splits)
  qed
qed

lemma lset-lmerge: ‹lset (lmerge lxss) = (⋃ lxs ∈ lset lxss. lset lxs)›
proof safe
  fix x
  assume ‹x ∈ lset (lmerge lxss)›
  then show ‹x ∈ ⋃ (lset ‘ lset lxss)›
  proof −
    have vert: ‹⋀i. set (lvertical lxss i) ⊆ ⋃ (lset ‘ lset lxss)›
    proof
      fix i y assume hy: ‹y ∈ set (lvertical lxss i)›
      show ‹y ∈ ⋃ (lset ‘ lset lxss)›
      proof (cases ‹enat i < llength lxss›)
        case True
        have hsub: ‹set (lvertical lxss i) ⊆ lset (lnth lxss i)›
          using True set-ltaken[of ‹Suc i› ‹lnth lxss i›]
          by (auto simp: lvertical-def intro: lset-intros dest: set-mp[OF set-ltaken])
        have hmem: ‹lnth lxss i ∈ lset lxss›
          using True by (auto simp: in-lset-conv-lnth)
        show ?thesis using hy hsub hmem by blast
      next
        case False
        then have ‹set (lvertical lxss i) = {}› by (simp add: lvertical-def)
        then show ?thesis using hy by simp
      qed
    qed
    have horiz: ‹⋀j. set (lhorizontal lxss j) ⊆ ⋃ (lset ‘ lset lxss)›
    proof
```

```
    fix j y assume hy: ‹y ∈ set (lhorizontal lxss j)›
    then obtain n where hn: ‹enat n < llength lxss› and hjn: ‹enat j < llength
(lnth lxss n)›
        and yj: ‹y = lnth (lnth lxss n) j›
      by (auto simp: lhorizontal-def map-filter-def in-lset-conv-lnth not-less not-le
          enat-0-iff min-def split: enat.splits if-splits dest!: set-mp[OF set-ltaken])
    have hmem: ‹lnth lxss n ∈ lset lxss›
      using hn by (auto simp: in-lset-conv-lnth)
    have hin: ‹y ∈ lset (lnth lxss n)›
      using hjn yj by (auto simp: in-lset-conv-lnth)
    show ‹y ∈ ⋃ (lset ‘ lset lxss)›
      using hmem hin by blast
  qed
  from ‹x ∈ lset (lmerge lxss)›
  obtain i where ‹x ∈ set (lvertical lxss i) ∨ x ∈ set (lhorizontal lxss i)›
    unfolding lmerge-def lset-lflat
    by (auto simp: lset-lupt)
  then show ?thesis
    using vert horiz by blast
 qed
next
 fix x lxs
 assume xlxss: ‹lxs ∈ lset lxss› and xlxs: ‹x ∈ lset lxs›
 from xlxss obtain i where hi: ‹enat i < llength lxss› and lxi: ‹lnth lxss i = lxs›
   by (auto simp: in-lset-conv-lnth)
 from xlxs obtain j where hj: ‹enat j < llength lxs› and xj: ‹lnth lxs j = x›
   by (auto simp: in-lset-conv-lnth)
 show ‹x ∈ lset (lmerge lxss)›
 proof (cases ‹j ≤ i›)
   case jle: True
   have x-in-vert: ‹x ∈ set (lvertical lxss i)›
   proof (cases ‹enat (Suc i) ≤ llength lxs›)
     case leq: True
     have ji-lt: ‹j < Suc i› using le-imp-less-Suc jle by blast
     have nth-eq: ‹ltaken (Suc i) lxs ! j = lnth lxs j›
       by (rule nth-ltaken) (fact ji-lt, fact leq)
     have jlt: ‹j < length (ltaken (Suc i) lxs)›
       using leq ji-lt by (metis length-ltaken)
     show ?thesis
       using hi hj xj lxi nth-eq jlt
       by (auto simp: lvertical-def in-set-conv-nth simp del: ltaken.simps)
   next
     case nleq: False
     obtain k where fin: ‹llength lxs = enat k›
       by (meson less-enatE linorder-not-le nleq)
     have len-eq: ‹length (ltaken (Suc i) lxs) = k›
       using nleq fin by (metis length-ltaken the-enat.simps)
     have jltk: ‹j < k›
       using hj fin by simp
```

```
    have nth-eq: ‹ltaken (Suc i) lxs ! j = lnth lxs j›
      by (metis len-eq jltk lnth-lappend-llist-of ltaken-ldropn-decomp)
    show ?thesis
      using hi hj xj lxi nth-eq jltk len-eq
      by (auto simp: lvertical-def in-set-conv-nth simp del: ltaken.simps)
  qed
  show ?thesis
    unfolding lmerge-def lset-lflat
    using x-in-vert by (auto simp: lset-lupt)
 next
  case False
  have x-in-horiz: ‹x ∈ set (lhorizontal lxss j)›
    using hi hj xj lxi False
     by (auto simp: not-le lhorizontal-def map-filter-def set-ltaken-conv' split:
enat.splits)
  show ?thesis
    unfolding lmerge-def lset-lflat
    using x-in-horiz by (auto simp: lset-lupt)
 qed
qed

lemma lsum-lmap-add: ‹lsum (lmap (λx. f x + g x) lxs) = lsum (lmap f lxs) +
lsum (lmap g lxs)›
 by (induct lxs) auto

lemma count-list-alt: ‹count-list xs x = card {j. j < length xs ∧ xs ! j = x}›
proof (induct xs)
 case (Cons a xs)
 then show ?case
 proof (cases ‹a = x›)
  case True
  with Cons show ?thesis
  proof −
    have eq: ‹{j. j < Suc (length xs) ∧ (x # xs) ! j = x} = insert 0 (Suc ‘ {j. j
< length xs ∧ xs ! j = x})›
      by (auto simp: nth-Cons' less-Suc-eq-0-disj)
    have fin: ‹finite {j. j < length xs ∧ xs ! j = x}› by simp
    have notin: ‹0 ∉ Suc ‘ {j. j < length xs ∧ xs ! j = x}› by simp
    have inj: ‹inj-on Suc {j. j < length xs ∧ xs ! j = x}› by (rule inj-onI) simp
    show ?thesis
      using Cons True
      by (simp add: eq fin notin card-image inj)
  qed
 next
  case False
  with Cons show ?thesis
    by (auto simp: nth-Cons' gr0-conv-Suc intro!: bij-betw-same-card[of Suc])
 qed
qed simp
```

lemma *count-list-lvertical*:
  ‹*count-list* (*lvertical lxss i*) *x* = (*if i* < *llength lxss then card* {*j*. *j* ≤ *i* ∧ *j* < *llength* (*lnth lxss i*) ∧ *lnth* (*lnth lxss i*) *j* = *x*} *else 0*)›
proof (*cases* ‹*enat i* < *llength lxss*›)
  case *False*
  then show *?thesis* **by** (*simp add*: *lvertical-def*)
next
  case *ili*: *True*
  have *lv*: ‹*lvertical lxss i* = *ltaken* (*Suc i*) (*lnth lxss i*)›
    using *ili* **by** (*simp add*: *lvertical-def*)
  show *?thesis*
  proof (*cases* ‹*enat* (*Suc i*) ≤ *llength* (*lnth lxss i*)›)
    case *leq*: *True*
    have *len*: ‹*length* (*ltaken* (*Suc i*) (*lnth lxss i*)) = *Suc i*›
      by (*simp del*: *ltaken.simps add*: *length-ltaken leq*)
    have *nth*: ‹⋀*j*. *j* < *Suc i* ⟹ *ltaken* (*Suc i*) (*lnth lxss i*) ! *j* = *lnth* (*lnth lxss i*) *j*›
      by (*metis leq nth-ltaken*)
    have *li*: ‹⋀*j*. *j* ≤ *i* ⟹ *enat j* < *llength* (*lnth lxss i*)›
    proof −
      fix *j* **assume** *hj*: ‹*j* ≤ *i*›
      have ‹*enat* (*Suc j*) ≤ *enat* (*Suc i*)› **using** *hj* **by** *simp*
       also have ‹*enat* (*Suc i*) ≤ *llength* (*lnth lxss i*)› **using** *leq* **by** (*simp add*: *Suc-ile-eq*)
      finally show ‹*enat j* < *llength* (*lnth lxss i*)› **by** (*simp add*: *Suc-ile-eq*)
    qed
    have *set-eq*: ‹{*j*. *j* < *length* (*ltaken* (*Suc i*) (*lnth lxss i*)) ∧ *ltaken* (*Suc i*) (*lnth lxss i*) ! *j* = *x*}
      = {*j*. *j* ≤ *i* ∧ *enat j* < *llength* (*lnth lxss i*) ∧ *lnth* (*lnth lxss i*) *j* = *x*}›
      by (*auto simp del*: *ltaken.simps simp add*: *len nth li less-Suc-eq-le*)
    have *key*: ‹*count-list* (*ltaken* (*Suc i*) (*lnth lxss i*)) *x* =
       card {*j*. *j* ≤ *i* ∧ *enat j* < *llength* (*lnth lxss i*) ∧ *lnth* (*lnth lxss i*) *j* = *x*}›
      by (*simp only*: *count-list-alt set-eq*)
    have ‹*count-list* (*lvertical lxss i*) *x* = *count-list* (*ltaken* (*Suc i*) (*lnth lxss i*)) *x*›
      by (*simp only*: *lv*)
    also have ‹... = *card* {*j*. *j* ≤ *i* ∧ *enat j* < *llength* (*lnth lxss i*) ∧ *lnth* (*lnth lxss i*) *j* = *x*}›
      by (*rule key*)
    also have ‹... = (*if enat i* < *llength lxss*
       then card {*j*. *j* ≤ *i* ∧ *enat j* < *llength* (*lnth lxss i*) ∧ *lnth* (*lnth lxss i*) *j* = *x*} *else 0*)›
      by (*simp add*: *ili*)
    finally show *?thesis* **.**
  next
    case *nleq*: *False*
    then obtain *k* **where** *fin*: ‹*llength* (*lnth lxss i*) = *enat k*›
      by (*meson less-enatE linorder-not-le*)
    from *nleq fin* **have** *klt*: ‹*k* < *Suc i*› **by** (*simp add*: *Suc-ile-eq*)

```
    have len-eq: ‹length (ltaken (Suc i) (lnth lxss i)) = k›
      using nleq fin by (simp del: ltaken.simps add: length-ltaken)
    have nth-eq: ‹⋀m. m < k ⟹ ltaken (Suc i) (lnth lxss i) ! m = lnth (lnth lxss
i) m›
      by (metis len-eq lnth-lappend-llist-of ltaken-ldropn-decomp)
    have li: ‹⋀j. j < k ⟹ enat j < llength (lnth lxss i)›
      by (simp add: fin)
    have le-i: ‹k ≤ i› using klt by simp
    have set-eq: ‹{j. j < length (ltaken (Suc i) (lnth lxss i)) ∧ ltaken (Suc i) (lnth
lxss i) ! j = x}
      = {j. j ≤ i ∧ enat j < llength (lnth lxss i) ∧ lnth (lnth lxss i) j = x}›
      by (auto simp del: ltaken.simps simp add: len-eq nth-eq li fin
          dest: order-less-le-trans[OF - le-i])
    have key: ‹count-list (ltaken (Suc i) (lnth lxss i)) x =
       card {j. j ≤ i ∧ enat j < llength (lnth lxss i) ∧ lnth (lnth lxss i) j = x}›
      by (simp only: count-list-alt set-eq)
    have ‹count-list (lvertical lxss i) x = count-list (ltaken (Suc i) (lnth lxss i)) x›
      by (simp only: lv)
    also have ‹... = card {j. j ≤ i ∧ enat j < llength (lnth lxss i) ∧ lnth (lnth lxss
i) j = x}›
      by (rule key)
    also have ‹... = (if enat i < llength lxss
        then card {j. j ≤ i ∧ enat j < llength (lnth lxss i) ∧ lnth (lnth lxss i) j =
x} else 0)›
      by (simp add: ili)
    finally show ?thesis .
  qed
qed

lemma llength-less-lfinite[simp]: ‹llength lxss < enat j ⟹ lfinite lxss›
  using enat-iless lfinite-conv-llength-enat by blast

lemma ltaken-all: ‹llength lxss < enat j ⟹ ltaken j lxss = list-of lxss›
  by (metis enat-iless length-ltaken linorder-not-le list-of-llist-of
      llength-llist-of lnth-equalityI lnth-lappend1 ltaken-ldropn-decomp
      the-enat.simps)

lemma count-list-map-filter:
  ‹count-list (List.map-filter f xs) x = card {i. i < length xs ∧ f (xs ! i) = Some
x}›
proof (induct xs)
  case Nil
  show ?case by simp
next
  case (Cons a xs)
  note IH = Cons.hyps
  show ?case
  proof (cases ‹f a›)
    case None
```

```
    have set-eq: ‹{i. i < Suc (length xs) ∧ f ((a # xs) ! i) = Some x} =
        Suc ` {i. i < length xs ∧ f (xs ! i) = Some x}›
      by (auto simp: None nth-Cons' gr0-conv-Suc)
     have ‹count-list (List.map-filter f (a # xs)) x = count-list (List.map-filter f
xs) x›
      by (simp add: None)
    also have ‹... = card {i. i < length xs ∧ f (xs ! i) = Some x}› by (rule IH)
    also have ‹... = card {i. i < Suc (length xs) ∧ f ((a # xs) ! i) = Some x}›
      using set-eq by (simp add: card-image)
    finally show ?thesis by simp
  next
    case (Some y)
    show ?thesis
    proof (cases ‹y = x›)
      case True
      have set-eq: ‹{i. i < Suc (length xs) ∧ f ((a # xs) ! i) = Some x} =
          insert 0 (Suc ` {i. i < length xs ∧ f (xs ! i) = Some x})›
        by (auto simp: True Some nth-Cons' less-Suc-eq-0-disj)
      have fin: ‹finite {i. i < length xs ∧ f (xs ! i) = Some x}› by simp
      have notin: ‹0 ∉ Suc ` {i. i < length xs ∧ f (xs ! i) = Some x}› by simp
      have ‹count-list (List.map-filter f (a # xs)) x =
          Suc (count-list (List.map-filter f xs) x)›
        by (simp add: Some True)
       also have ‹... = Suc (card {i. i < length xs ∧ f (xs ! i) = Some x})› by
(simp add: IH)
      also have ‹... = card (insert 0 (Suc ` {i. i < length xs ∧ f (xs ! i) = Some
x}))›
        by (simp add: fin notin card-image)
      also have ‹... = card {i. i < Suc (length xs) ∧ f ((a # xs) ! i) = Some x}›
        by (simp only: set-eq)
      finally show ?thesis by simp
    next
      case False
      have set-eq: ‹{i. i < Suc (length xs) ∧ f ((a # xs) ! i) = Some x} =
          Suc ` {i. i < length xs ∧ f (xs ! i) = Some x}›
        by (auto simp: Some False nth-Cons' gr0-conv-Suc)
       have ‹count-list (List.map-filter f (a # xs)) x = count-list (List.map-filter f
xs) x›
        by (simp add: Some False)
      also have ‹... = card {i. i < length xs ∧ f (xs ! i) = Some x}› by (rule IH)
      also have ‹... = card {i. i < Suc (length xs) ∧ f ((a # xs) ! i) = Some x}›
        using set-eq by (simp add: card-image)
      finally show ?thesis by simp
    qed
  qed
qed

lemma count-list-lhorizontal:
  ‹count-list (lhorizontal lxss j) x = card {i. i < j ∧ i < llength lxss ∧ j < llength
```

```
(lnth lxss i) ∧ lnth (lnth lxss i) j = x}›
  unfolding lhorizontal-def count-list-map-filter
proof (rule arg-cong[where f=card])
  show ‹{i. i < length (ltaken j lxss) ∧ (if enat j < llength (ltaken j lxss ! i) then
Some (lnth (ltaken j lxss ! i) j) else None) = Some x} =
        {i. i < j ∧ enat i < llength lxss ∧ enat j < llength (lnth lxss i) ∧ lnth (lnth
lxss i) j = x}›
  proof (rule set-eqI, rule iffI)
    fix i
    assume ‹i ∈ {i. i < length (ltaken j lxss) ∧ (if enat j < llength (ltaken j lxss !
i) then Some (lnth (ltaken j lxss ! i) j) else None) = Some x}›
    hence h-lt: ‹i < length (ltaken j lxss)› and h-if: ‹enat j < llength (ltaken j lxss
! i)›
      and h-x: ‹lnth (ltaken j lxss ! i) j = x›
      by (auto split: if-splits)
    have h-i-lt: ‹enat i < llength lxss›
    proof (cases ‹enat j ≤ llength lxss›)
      case True
      with h-lt have ‹i < j› by (simp add: length-ltaken)
      with True show ?thesis by (simp add: order-less-le-subst2)
    next
      case False
      with h-lt have ‹i < the-enat (llength lxss)› by (simp add: length-ltaken)
      then show ?thesis
       by (metis False enat-iless enat-ord-simps(2) linorder-not-less the-enat.simps)
    qed
    have h-key: ‹ltaken j lxss ! i = lnth lxss i›
     by (metis h-lt length-ltaken linorder-not-le llength-less-lfinite ltaken-all nth-list-of
nth-ltaken)
    show ‹i ∈ {i. i < j ∧ enat i < llength lxss ∧ enat j < llength (lnth lxss i) ∧
lnth (lnth lxss i) j = x}›
      using h-lt h-if h-x h-i-lt h-key
      by (smt (verit, del-insts) order.strict-trans enat-ord-simps(2) length-ltaken
          linorder-less-linear mem-Collect-eq order.strict-iff-order)
  next
    fix i
    assume ‹i ∈ {i. i < j ∧ enat i < llength lxss ∧ enat j < llength (lnth lxss i)
∧ lnth (lnth lxss i) j = x}›
    hence h-j: ‹i < j› and h-lt: ‹enat i < llength lxss›
      and h-llength: ‹enat j < llength (lnth lxss i)› and h-x: ‹lnth (lnth lxss i) j =
x›
      by auto
    have h-key: ‹ltaken j lxss ! i = lnth lxss i›
     by (metis h-j llength-less-lfinite ltaken-all not-le-imp-less nth-list-of nth-ltaken)
    show ‹i ∈ {i. i < length (ltaken j lxss) ∧ (if enat j < llength (ltaken j lxss ! i)
then Some (lnth (ltaken j lxss ! i) j) else None) = Some x}›
      using h-j h-lt h-llength h-x h-key
    by (smt (verit, best) enat-ord-simps(2) length-list-of length-ltaken linorder-less-linear
          llength-less-lfinite ltaken-all mem-Collect-eq order-le-less)
```

```
  qed
qed

lemma LSUM-extend: assumes ‹lprefix lxs lys›
  ‹∀ i < llength lxs. f (lnth lxs i) = g (lnth lys i)›
  ‹∀ i < llength lys. i ≥ llength lxs ⟶ g (lnth lys i) = 0›
  shows ‹LSUM lxs f = LSUM lys g›
proof −
  have key: ‹LSUM lxs f = LSUM lys g› if ‹lfinite lxs›
    using that assms
  proof (induct lxs arbitrary: lys rule: lfinite.induct)
    case (lfinite-LNil lys)
    have ‹∀ x ∈ lset lys. g x = 0›
      using lfinite-LNil(3) by (auto simp: in-lset-conv-lnth)
    then have ‹LSUM lys g = 0› by (simp add: lsum-0-iff)
    then show ?case by simp
  next
    case (lfinite-LConsI xs x lys-outer)
    obtain ys' where ys-eq: ‹lys-outer = LCons x ys'› and pre: ‹lprefix xs ys'›
      using lfinite-LConsI(3) by (auto simp: LCons-lprefix-conv)
    have eq-x: ‹f x = g x›
      using spec[OF lfinite-LConsI(4), of 0] unfolding ys-eq
      by (simp add: zero-enat-def[symmetric])
    have IH5: ‹∀ i. enat i < llength xs ⟶ f (lnth xs i) = g (lnth ys' i)›
    proof (intro allI impI)
      fix i assume hlt: ‹enat i < llength xs›
     then have hi: ‹enat (Suc i) < llength (LCons x xs)› by (simp add: Suc-ile-eq)
      have h5i: ‹enat (Suc i) < llength (LCons x xs) ⟶
          f (lnth (LCons x xs) (Suc i)) = g (lnth lys-outer (Suc i))›
        by (rule spec[OF lfinite-LConsI(4), of ‹Suc i›])
      show ‹f (lnth xs i) = g (lnth ys' i)›
        using h5i hi unfolding ys-eq by simp
    qed
    have IH6: ‹∀ i. enat i < llength ys' ⟶ llength xs ≤ enat i ⟶ g (lnth ys' i)
= 0›
    proof (intro allI impI)
      fix i assume hi: ‹enat i < llength ys'› and hi2: ‹llength xs ≤ enat i›
      have h1: ‹enat (Suc i) < llength lys-outer›
        unfolding ys-eq by (simp add: Suc-ile-eq hi)
      have h2: ‹llength (LCons x xs) ≤ enat (Suc i)›
        by (simp add: eSuc-enat[symmetric] hi2)
      have h6i: ‹enat (Suc i) < llength lys-outer ⟶
          llength (LCons x xs) ≤ enat (Suc i) ⟶ g (lnth lys-outer (Suc i)) = 0›
        by (rule spec[OF lfinite-LConsI(5), of ‹Suc i›])
      show ‹g (lnth ys' i) = 0›
        using h6i h1 h2 unfolding ys-eq by simp
    qed
    have eq-tails: ‹LSUM xs f = LSUM ys' g›
      by (rule lfinite-LConsI(2)[of ys', OF pre IH5 IH6])
```

```
    show ?case unfolding ys-eq using eq-x eq-tails by simp
  qed
  show ?thesis
  proof (cases ‹lfinite lxs›)
    case True show ?thesis using key[OF True] .
  next
    case False then show ?thesis using assms
      by (metis in-lset-conv-lnth llist.map-cong not-lfinite-lprefix-conv-eq)
  qed
qed

lemma lupt-0[simp]: ‹lupt i 0 = LNil›
  by (cases i) auto

lemma lprefix-lupt: ‹j ≤ k ⟹ lprefix (lupt i j) (lupt i k)›
proof (cases ‹enat i ≤ j›)
  case False
  then show ?thesis by (simp add: lnull-lprefix)
next
  case True
  assume jk: ‹j ≤ k›
  show ?thesis
  proof (cases j)
    case (enat m)
    have split: ‹lupt i k = lappend (lupt i (enat m)) (lupt m k)›
      using True[unfolded enat] jk[unfolded enat]
      by (coinduction arbitrary: i m k)
         (force simp: Suc-ile-eq not-le order-less-le-subst2 dual-order.strict-trans1
lappend-lnull1)
    then show ?thesis
      unfolding enat lprefix-conv-lappend by blast
  next
    case infinity
    then show ?thesis using True jk by simp
  qed
qed

lemma if-card-else-0: ‹(if P z then card {x. Q x z} else 0) = card {x. P z ∧ Q x
z}›
  by auto

lemma LSUM-llist-of[simp]: ‹LSUM (llist-of xs) f = sum-list (map f xs)›
  by (induct xs) auto

lemma card-sum-list: ‹(⋀i. P i ⟹ i ≤ n) ⟹
  card {i :: nat. P i} = sum-list (map (λi. if P i then 1 else 0) [0 ..< Suc n])›
proof (induct n arbitrary: P)
  case 0
  then show ?case by (auto simp: card-eq-0-iff card-1-singleton-iff)
```

```
next
  case (Suc n P)
  have bound: ‹⋀i. (P i ∧ i ≠ Suc n) ⟹ i ≤ n›
    using Suc.prems by (force simp: le-Suc-eq)
  have IH: ‹card {i. P i ∧ i ≠ Suc n} =
      sum-list (map (λi. if P i ∧ i ≠ Suc n then 1 else 0) [0..<Suc n])›
    by (rule Suc.hyps[of ‹λi. P i ∧ i ≠ Suc n›, OF bound])
  have sum-eq: ‹sum-list (map (λi. if P i ∧ i ≠ Suc n then 1 else 0) [0..<Suc n])
=
      sum-list (map (λi. if P i then 1 else 0) [0..<Suc n])›
    by (intro arg-cong[where f=sum-list] map-cong refl)
       (auto simp: le-Suc-eq dest: Suc.prems)
  show ?case
  proof (cases ‹P (Suc n)›)
    case True
    have fin: ‹finite {i. P i ∧ i ≠ Suc n}›
      by (rule finite-subset[of - ‹{0..<Suc n}›])
         (auto simp: le-Suc-eq dest: Suc.prems)
    have notin: ‹Suc n ∉ {i. P i ∧ i ≠ Suc n}› by simp
    have insert-eq: ‹{i. P i} = insert (Suc n) {i. P i ∧ i ≠ Suc n}›
      using True by auto
    have ‹card {i. P i} = Suc (card {i. P i ∧ i ≠ Suc n})›
      unfolding insert-eq using fin notin by simp
    also have ‹... = Suc (sum-list (map (λi. if P i then 1 else 0) [0..<Suc n]))›
      by (metis (mono-tags, lifting) IH sum-eq)
    also have ‹... = sum-list (map (λi. if P i then 1 else 0) [0..<Suc (Suc n)])›
      using True by simp
    finally show ?thesis .
  next
    case False
    have eq: ‹{i. P i} = {i. P i ∧ i ≠ Suc n}› using False by auto
    have ‹card {i. P i} = sum-list (map (λi. if P i then 1 else 0) [0..<Suc n])›
      unfolding eq by (metis (mono-tags, lifting) IH sum-eq)
    also have ‹... = sum-list (map (λi. if P i then 1 else 0) [0..<Suc (Suc n)])›
      using False by simp
    finally show ?thesis .
  qed
qed

lemma llist-of-lupt: ‹llist-of [i ..< j] = lupt i j›
  by (coinduction arbitrary: i j) auto

lemma enat-sum-list: ‹enat (sum-list xs) = sum-list (map enat xs)›
  by (induct xs) (auto simp: enat-0 simp flip: plus-enat-simps)

lemma card-LSUM:
  assumes ‹(⋀i. P i ⟹ i ≤ n)›
  shows ‹enat (card {i :: nat. P i}) = LSUM lenats (λi. if P i then 1 else 0)›
proof −
```

```
  have ‹enat (card {i :: nat. P i}) =
      enat (∑ i←[0..<Suc n]. if P i then 1 else 0)›
    using card-sum-list[OF assms] by simp
  also have ‹... = sum-list (map (enat ∘ (λi. if P i then 1 else 0)) [0..<Suc n])›
    by (simp only: enat-sum-list list.map-comp)
  also have ‹... = LSUM (llist-of [0..<Suc n]) (enat ∘ (λi. if P i then 1 else 0))›
    by (simp only: LSUM-llist-of)
  also have ‹... = LSUM lenats (λi. if P i then 1 else 0)›
  proof (rule LSUM-extend)
    show ‹lprefix (llist-of [0..<Suc n]) lenats›
      by (simp add: llist-of-lupt lprefix-lupt del: upt.simps)
    show ‹∀ i < llength (llist-of [0..<Suc n]).
        (enat ∘ (λi. if P i then 1 else 0)) (lnth (llist-of [0..<Suc n]) i) =
        (λi. if P i then 1 else 0) (lnth lenats i)›
      by (simp add: lnth-lupt one-enat-def enat-0 del: upt.simps)
    show ‹∀ i < llength lenats.
        llength (llist-of [0..<Suc n]) ≤ enat i ⟶
        (λi. if P i then 1 else 0) (lnth lenats i) = 0›
      by (simp add: lnth-lupt del: upt.simps) (metis assms not-less-eq-eq)
  qed
  finally show ?thesis .
qed

lemma LSUM-extend-lenats:
  ‹LSUM lxs f =  LSUM lenats (λi. if enat i < llength lxs then f (lnth lxs i) else
0)›
proof −
  have step1: ‹LSUM (lupt 0 (llength lxs)) (f ∘ lnth lxs) =
    LSUM lenats (λi. if enat i < llength lxs then f (lnth lxs i) else 0)›
  proof (rule LSUM-extend)
    show ‹lprefix (lupt 0 (llength lxs)) lenats›
      by (simp add: lprefix-lupt)
  next
    show ‹∀ i. enat i < llength (lupt 0 (llength lxs)) ⟶
      (f ∘ lnth lxs) (lnth (lupt 0 (llength lxs)) i) =
      (λi. if enat i < llength lxs then f (lnth lxs i) else 0) (lnth lenats i)›
      by (simp add: lnth-lupt)
  next
    show ‹∀ i. enat i < llength lenats ⟶ llength (lupt 0 (llength lxs)) ≤ enat i
⟶
      (λi. if enat i < llength lxs then f (lnth lxs i) else 0) (lnth lenats i) = 0›
      by (fastforce simp: lnth-lupt)
  qed
  show ?thesis
    by (subst llist-conv-lmap-lupt) (simp add: llist.map-comp flip: step1)
qed

lemma LCons-eq-lmap-conv:
  ‹(LCons y lys = lmap f lxs) = (∃ x lxs'. lxs = LCons x lxs' ∧ y = f x ∧ lys =
```

```
lmap f lxs')›
  using lmap-eq-LCons-conv by fastforce

lemma lmap-eq-lappend: ‹lfinite lys ⟹ lmap f lxs = lappend lys lzs ⟷
  (∃ lus lvs. lxs = lappend lus lvs ∧ lys = lmap f lus ∧ lzs = lmap f lvs)›
proof (induct lys arbitrary: lxs rule: lfinite-induct)
  case LNil
  then show ?case
    by (auto simp: lnull-def LNil-eq-lmap)
next
  case (LCons lys lxs)
  from LCons(3)[of ‹ltl lxs›] LCons(1,2) show ?case
    by (cases lxs; cases lys)
    (fastforce simp: lnull-def lmap-eq-LCons-conv LCons-eq-lmap-conv lmap-lappend-distrib
        intro: exI[of - ‹LCons - -›])+
qed

lemma llist-of-eq-lmap-conv:
  ‹llist-of xs = lmap f lxs ⟷ (∃ ys. lxs = llist-of ys ∧ map f ys = xs)›
 by (induct xs arbitrary: lxs) (force simp: LNil-eq-lmap LCons-eq-lmap-conv map-eq-Cons-conv)+

lemma neq-LCons-conv: ‹(∀ x lxs. lys ≠ LCons x lxs) ⟷ lys = LNil›
  by (cases lys; simp)

lemma epred-iadd: ‹epred (a + b) = (if a = 0 then epred b else epred a + b)›
  by (cases a rule: co.enat.exhaust) (auto simp: epred-iadd1)

lemma LSUM-isum: ‹ldistinct lxs ⟹ LSUM lxs f = isum f (lset lxs)›
proof (coinduction arbitrary: lxs f)
  case eq-enat
  show ?case
  proof (intro conjI impI disjI1)
    show ‹(LSUM lxs f = 0) = (isum f (lset lxs) = 0)›
      by (auto simp: lsum-0-iff isum-eq-0-iff)
  next
    assume hne1: ‹LSUM lxs f ≠ 0› and hne2: ‹isum f (lset lxs) ≠ 0›
    have hexists: ‹∃ x ∈ lset (lmap f lxs). x ≠ 0›
      using hne1 by (auto simp: lsum-0-iff)
    have hfin: ‹lfinite (ltakeWhile ((=) 0) (lmap f lxs))›
      using hexists by (auto simp: lfinite-ltakeWhile)
    have hnotnil: ‹ldropWhile ((=) 0) (lmap f lxs) ≠ LNil›
      using hexists by (auto simp: ldropWhile-eq-LNil-iff)
    obtain z lzs where hlws: ‹ldropWhile ((=) 0) (lmap f lxs) = LCons z lzs›
      using hnotnil by (cases ‹ldropWhile ((=) 0) (lmap f lxs)›) auto
    have hnz: ‹z ≠ 0›
      using hexists hlws by (simp add: ldropWhile-eq-LCons-iff)
    define zs where ‹zs = list-of (ltakeWhile ((=) 0) (lmap f lxs))›
    have hlmap: ‹lmap f lxs = lappend (llist-of zs) (LCons z lzs)›
      unfolding zs-def using hfin hlws
```

```
    by (metis lappend-ltakeWhile-ldropWhile llist-of-list-of)
  have hzeros: ‹∀x ∈ set zs. x = 0›
    using hfin lset-ltakeWhileD zs-def by fastforce
  let ?f = ‹λy. if y = lnth lxs (length zs) then epred (f y) else f y›
  have hep1: ‹epred (LSUM lxs f) = LSUM lxs ?f›
  proof −
    { fix ys :: ‹'a list› and x :: ‹'a› and lxs' :: ‹'a llist›
      assume hall: ‹∀z∈set ys. f z = 0› and hfx: ‹f x ≠ 0›
        and hnotinys: ‹x ∉ set ys› and hnotinlxs': ‹x ∉ lset lxs'›
      have hsum0: ‹sum-list (map f ys) = 0›
        using hall by simp
      have hsum-if0: ‹(∑z←ys. if z = x then epred (f z) else f z) = 0›
        using hnotinys hall by (auto intro!: sum-list-0 split: if-splits)
      have hLSUM-eq: ‹LSUM lxs' (λz. if z = x then epred (f z) else f z) = LSUM
lxs' f›
        by (rule arg-cong[where f=‹lsum›], rule llist.map-cong0) (use hnotinlxs'
in auto)
      have ‹epred (sum-list (map f ys) + (f x + LSUM lxs' f)) =
        (∑z←ys. if z = x then epred (f z) else f z) + (epred (f x) + LSUM lxs'
(λz. if z = x then epred (f z) else f z))›
        using hfx hsum0 hsum-if0 hLSUM-eq by (simp add: epred-iadd1 hsum0
hsum-if0)
    }
    thus ?thesis
      using hlmap hzeros hnz eq-enat
    by (auto simp: lmap-eq-lappend lmap-lappend-distrib lsum-lappend LCons-eq-lmap-conv
        llist-of-eq-lmap-conv ldistinct-lappend lnth-lappend2)
  qed
  have hep2: ‹epred (isum f (lset lxs)) = isum ?f (lset lxs)›
  proof −
    { fix ys :: ‹'a list› and x :: ‹'a› and lxs' :: ‹'a llist›
      assume hall: ‹∀z∈set ys. f z = 0› and hfx: ‹f x ≠ 0›
        and hnotinys: ‹x ∉ set ys› and hnotinlxs': ‹x ∉ lset lxs'›
        and hdist: ‹distinct ys› and hsep: ‹set ys ∩ lset lxs' = {}›
      have hisum-ys0: ‹isum f (set ys) = 0›
        using hall by (simp add: isum-eq-0-iff)
      have hisum-ys-if0: ‹isum (λy. if y = x then epred (f y) else f y) (set ys) =
0›
        using hnotinys hall by (auto simp: isum-eq-0-iff)
      have hisum-lxs'-eq: ‹isum (λy. if y = x then epred (f y) else f y) (lset lxs')
= isum f (lset lxs')›
        using hnotinlxs' by (intro isum-cong) auto
      have ‹epred (isum f (set ys) + isum f (lset lxs') + f x) =
        isum (λy. if y = x then epred (f y) else f y) (set ys) +
        isum (λy. if y = x then epred (f y) else f y) (lset lxs') + epred (f x)›
        using hfx hisum-ys0 hisum-ys-if0 hisum-lxs'-eq
        by (simp add: epred-iadd hisum-ys0 hisum-ys-if0 add.commute add.assoc)
    }
    thus ?thesis
```

```
      using hlmap hzeros hnz eq-enat
    by (auto simp: lmap-eq-lappend lmap-lappend-distrib lsum-lappend LCons-eq-lmap-conv
        llist-of-eq-lmap-conv ldistinct-lappend lnth-lappend2)
  qed
  show ‹∃(lxs' :: 'a llist) f'. epred (LSUM lxs f) = LSUM lxs' f' ∧
    epred (isum f (lset lxs)) = isum f' (lset lxs') ∧ ldistinct lxs'›
    by (intro disjI1 exI[of - lxs] exI[of - ‹?f›]) (auto simp: hep1 hep2 eq-enat)
 qed
qed

lemma set-partition-subset: ‹A ⊆ B ⟹ B = A ∪ (B − A)›
 by auto

lemma count-llist-isum: ‹count-llist lxs x = isum (λi. if enat i < llength lxs ∧ lnth
lxs i = x then 1 else 0) UNIV›
proof (coinduction arbitrary: lxs)
 case eq-enat
 show ?case
 proof −
  have h0: ‹(count-llist lxs x = 0) = (isum (λi. if enat i < llength lxs ∧ lnth lxs
i = x then 1 else 0) UNIV = 0)›
    by (auto simp: count-llist-zero-iff isum-eq-0-iff in-lset-conv-lnth)
  show ?thesis
  proof (cases ‹count-llist lxs x = 0›)
   case True
   then show ?thesis using h0 by simp
  next
   case False
   have hx-lset: ‹x ∈ lset lxs›
    using False by (simp add: count-llist-zero-iff)
   have htw-fin: ‹lfinite (ltakeWhile ((≠) x) lxs)›
    using hx-lset by (simp add: lfinite-ltakeWhile)
   obtain n where hlen-enat: ‹llength (ltakeWhile ((≠) x) lxs) = enat n›
    using lfinite-llength-enat[OF htw-fin] by blast
   have hlt-tw: ‹llength (ltakeWhile ((≠) x) lxs) < llength lxs›
    using hx-lset by (simp add: llength-ltakeWhile-lt-iff)
   have hx-at-n: ‹lnth lxs n = x›
   proof −
    have h1: ‹¬ ((≠) x) (lnth lxs (the-enat (llength (ltakeWhile ((≠) x) lxs))))›
     by (rule lnth-llength-ltakeWhile[OF hlt-tw])
    show ?thesis using h1 hlen-enat by simp
   qed
   have hn-lt: ‹enat n < llength lxs›
    using hlt-tw hlen-enat by auto
   have hdrop-eq: ‹ldropWhile ((≠) x) lxs = ldropn n lxs›
    by (simp add: ldropWhile-eq-ldrop hlen-enat ldrop-enat)
   define lxs' where hlxs': ‹lxs' = ltl (ldropWhile ((≠) x) lxs)›
   have hlxs'-eq: ‹lxs' = ldropn (Suc n) lxs›
   proof −
```

```
    have ‹ldropn n lxs = LCons (lnth lxs n) (ldropn (Suc n) lxs)›
      using ldropn-Suc-conv-ldropn[OF hn-lt] by simp
    then show ?thesis by (simp add: hlxs' hdrop-eq)
  qed
  define f where f-def: ‹f = (λi. if enat i < llength lxs ∧ lnth lxs i = x then
1 else (0 :: enat))›
  define g where g-def: ‹g = (λi. if enat i < llength lxs' ∧ lnth lxs' i = x then
1 else (0 :: enat))›
  have hsel: ‹epred (count-llist lxs x) = count-llist lxs' x›
    unfolding hlxs' by (rule count-llist-sel[OF hx-lset])
  have hfbefore: ‹∀ i < n. f i = 0›
  proof (intro allI impI)
    fix i assume hi: ‹i < n›
    have hilt: ‹enat i < llength (ltakeWhile ((≠) x) lxs)›
      using hlen-enat hi by simp
    have hmem: ‹lnth (ltakeWhile ((≠) x) lxs) i ∈ lset (ltakeWhile ((≠) x) lxs)›
      by (rule iffD2[OF in-lset-conv-lnth]) (use hilt in auto)
    have hne: ‹lnth (ltakeWhile ((≠) x) lxs) i ≠ x›
      using lset-ltakeWhileD[OF hmem] by auto
    have heq2: ‹lnth (ltakeWhile ((≠) x) lxs) i = lnth lxs i›
      using ltakeWhile-nth[OF hilt] .
    show ‹f i = 0›
      unfolding f-def using hne heq2 hn-lt hlen-enat by simp
  qed
  have hfn: ‹f n = 1›
    unfolding f-def using hn-lt hx-at-n by simp
  have hfUNIV: ‹isum f UNIV = isum f {0..<Suc n} + isum f {Suc n..}›
  proof −
    have huniv: ‹UNIV = {0..<Suc n} ∪ ({Suc n..} :: nat set)› by auto
    show ?thesis by (subst huniv, rule isum-Un) auto
  qed
  have hfin-n: ‹isum f {0..<Suc n} = 1›
  proof −
    have ‹isum f {0..<Suc n} = sum f {0..<Suc n}›
      by (rule isum-eq-sum) simp
    also have ‹... = sum f {0..<n} + f n›
      by (rule sum.atLeastLessThan-Suc) simp
    also have ‹sum f {0..<n} = 0›
      by (rule sum.neutral) (use hfbefore in auto)
    finally show ?thesis using hfn by simp
  qed
  have hfg-len: ‹enat (k + Suc n) < llength lxs ⟷ enat k < llength lxs'› for k
    by (metis hlxs'-eq ldropn-Suc-conv-ldropn ldropn-eq-LConsD ldropn-ldropn)
  have hfg-lnth: ‹lnth lxs (k + Suc n) = lnth lxs' k› if ‹enat k < llength lxs'›
for k
    unfolding hlxs'-eq using that hfg-len[of k]
    by (subst lnth-ldropn) (auto simp: add.commute)
  have hfg-eq: ‹f (k + Suc n) = g k› for k
    using hfg-lnth hfg-len by (auto simp: f-def g-def)
```

```
    have hfg: ‹isum f {Suc n..} = isum g UNIV›
    proof (rule isum-reindex-cong[where l = ‹λk. k + Suc n›])
      show ‹inj-on (λk. k + Suc n) UNIV› by (simp add: inj-on-def)
      show ‹{Suc n..} = (λk. k + Suc n) ‘ UNIV›
        by (metis (no-types, lifting) ext add.commute add.right-neutral atLeast-0
image-add-atLeast)
      fix k show ‹f (k + Suc n) = g k› using hfg-eq by simp
    qed
    have heq: ‹epred (isum f UNIV) = isum g UNIV›
      by (simp add: hfUNIV hfin-n hfg epred-iadd1)
    then show ?thesis using h0 False hsel heq
      by (auto simp: f-def g-def intro: exI[of - lxs'])
  qed
 qed
qed

lemma count-llist-lmerge: ‹count-llist (lmerge lxss) x = lsum (lmap (λlxs. count-llist
lxs x) lxss)›
proof −
 let ?LSUM = ‹λP. LSUM lenats (λi. if P i then 1 else 0)›
 let ?isum = ‹λP. isum (λi. if P i then 1 else 0) UNIV›
 have vert: ‹enat (card {xa. enat z < llength lxss ∧ xa ≤ z ∧ enat xa < llength
(lnth lxss z) ∧ lnth (lnth lxss z) xa = x}) =
   ?LSUM (λi. enat z < llength lxss ∧ i ≤ z ∧ enat i < llength (lnth lxss z) ∧
lnth (lnth lxss z) i = x)› for z
   by (rule card-LSUM) blast
 have horiz: ‹enat (card {i. i < z ∧ enat i < llength lxss ∧ enat z < llength (lnth
lxss i) ∧ lnth (lnth lxss i) z = x}) =
   ?LSUM (λi. i < z ∧ enat i < llength lxss ∧ enat z < llength (lnth lxss i) ∧
lnth (lnth lxss i) z = x)› for z
   by (rule card-LSUM) (rule less-imp-le, blast)
 have lset-lenats: ‹lset lenats = (UNIV :: nat set)›
   by (simp add: lset-lupt)
 have LSUM-len-isum: ‹LSUM lenats f = isum f UNIV› for f :: ‹nat ⇒ enat›
   by (simp only: LSUM-isum[OF ldistinct-lupt] lset-lenats)
 have rhs-eq: ‹lsum (lmap (λlxs. count-llist lxs x) lxss) =
    isum (λa. if enat a < llength lxss then
           isum (λb. if enat b < llength (lnth lxss a) ∧ lnth (lnth lxss a) b = x
then 1 else 0) UNIV
         else 0) UNIV›
   by (subst LSUM-extend-lenats)
    (auto simp add: LSUM-len-isum count-llist-isum intro: isum-cong)
 have lhs-step1: ‹LSUM lenats (λi. count-list (lvertical lxss i @ lhorizontal lxss i)
x) =
    LSUM lenats (λz.
      ?LSUM (λi. enat z < llength lxss ∧ i ≤ z ∧ enat i < llength (lnth lxss z)
∧ lnth (lnth lxss z) i = x) +
      ?LSUM (λi. i < z ∧ enat i < llength lxss ∧ enat z < llength (lnth lxss i) ∧
lnth (lnth lxss i) z = x))›
```

unfolding *plus-enat-simps(1)*[*symmetric*] *lsum-lmap-add count-list-append count-list-lvertical count-list-lhorizontal*
by (*subst if-card-else-0*) (*auto simp*: *vert horiz*)
also have *lhs-step2*: ‹... =
isum (λ*z*.
?isum (λ*i*. *enat z* < *llength lxss* ∧ *i* ≤ *z* ∧ *enat i* < *llength* (*lnth lxss z*) ∧ *lnth* (*lnth lxss z*) *i* = *x*) +
?isum (λ*i*. *i* < *z* ∧ *enat i* < *llength lxss* ∧ *enat z* < *llength* (*lnth lxss i*) ∧ *lnth* (*lnth lxss i*) *z* = *x*)) *UNIV*›
by (*simp only*: *LSUM-len-isum*)
also have *lhs-step3*: ‹... =
isum (λ*z*. *?isum* (λ*i*. *enat z* < *llength lxss* ∧ *i* ≤ *z* ∧ *enat i* < *llength* (*lnth lxss z*) ∧ *lnth* (*lnth lxss z*) *i* = *x*)) *UNIV* +
isum (λ*z*. *?isum* (λ*i*. *i* < *z* ∧ *enat i* < *llength lxss* ∧ *enat z* < *llength* (*lnth lxss i*) ∧ *lnth* (*lnth lxss i*) *z* = *x*)) *UNIV*›
by (*rule isum-plus*)
also have *lhs-step4*: ‹*isum* (λ*z*. *?isum* (λ*i*. *i* < *z* ∧ *enat i* < *llength lxss* ∧ *enat z* < *llength* (*lnth lxss i*) ∧ *lnth* (*lnth lxss i*) *z* = *x*)) *UNIV* =
isum (λ*a*. *?isum* (λ*z*. *a* < *z* ∧ *enat a* < *llength lxss* ∧ *enat z* < *llength* (*lnth lxss a*) ∧ *lnth* (*lnth lxss a*) *z* = *x*)) *UNIV*›
by (*rule isum-swap*)
finally have *lhs-eq*: ‹*LSUM lenats* (λ*i*. *count-list* (*lvertical lxss i* @ *lhorizontal lxss i*) *x*) =
isum (λ*a*. *if enat a* < *llength lxss then*
isum (λ*b*. *if enat b* < *llength* (*lnth lxss a*) ∧ *lnth* (*lnth lxss a*) *b* = *x then 1 else 0*) *UNIV*
else 0) *UNIV*›
by (*unfold isum-plus*[*symmetric*]) (*auto intro*!: *isum-cong*)
show *?thesis*
unfolding *lmerge-def count-llist-lflat*
using *lhs-eq rhs-eq* **by** (*auto simp*: *llist.map-comp*)
qed

10 Countable Multisets as a Subtype

lift-definition *cmcount* :: ‹'*a cmset* ⇒ '*a* ⇒ *enat*› **is** *count-llist*
by (*auto simp*: *eq-cmset-def*)

lift-definition *cmempty* :: ‹'*a cmset*› **is** *LNil* **.**

lemma *cmcount-cmempty*[*simp*]: ‹*cmcount cmempty x* = *0*›
by *transfer auto*

lemma *countable-nonzero-cmcount*:
fixes *M* :: ‹'*a cmset*›
shows ‹*countable* {*x*. *cmcount M x* ≠ *0*}›
proof (*transfer*)
fix *lxs* :: ‹'*a llist*›
show ‹*countable* {*x*. *count-llist lxs x* ≠ *0*}›

```
  proof (rule countable-subset[of - ‹lset lxs›])
    show ‹{x. count-llist lxs x ≠ 0} ⊆ lset lxs›
      by (auto simp: count-llist-zero-iff)
    have ‹countable {i. enat i < llength lxs}›
      by simp
    from countable-image[OF this, where f=‹lnth lxs›] show ‹countable (lset lxs)›
      by (elim countable-subset[rotated]) (auto simp: lset-conv-lnth)
  qed
qed

lemma countable-exists-llist:
  assumes ‹countable X›
  shows ‹∃lxs. lset lxs = X ∧ ldistinct lxs›
proof (cases ‹finite X›)
  case True
  then show ?thesis
  proof (induct X)
    case empty
    then show ?case
      by (auto intro: exI[of - LNil])
  next
    case (insert x F)
    then show ?case
      by (auto intro: exI[of - ‹LCons x -›])
  qed
next
  case False
  with assms obtain g where g: ‹bij-betw (g :: nat ⇒ 'a) UNIV X›
    using bij-betw-from-nat-into by blast
  then show ?thesis
    by (intro exI[of - ‹lmap g lenats›])
      (auto simp: bij-betw-def image-iff inj-on-def lset-lupt)
qed

lemma in-range-cmcount:
  assumes ‹countable {x :: 'a. f x ≠ 0}›
  shows ‹f ∈ range cmcount›
proof −
  from assms obtain lxs where lxs: ‹ldistinct lxs› ‹lset lxs = {x :: 'a. f x ≠ 0}›
    using countable-exists-llist by blast
  define lys where ‹lys = lmerge (lmap (λx. lmap (λ-. x) (lupt 0 (f x))) lxs)›
  have ‹count-llist lys x = f x› for x
    unfolding lys-def count-llist-lmerge
   by (cases ‹f x = 0›) (auto simp: llist.map-comp LSUM-isum lxs count-llist-lmap-const
isum-eq-0-iff
       intro: isum-eq-singl cong: if-cong)
  then show ?thesis
    by (intro image-eqI[of - - ‹abs-cmset lys›]) (auto simp: cmcount.abs-eq)
qed
```

lemma *inj-cmcount*: ‹*inj cmcount*›
unfolding *inj-on-def*
by *transfer* (*auto simp*: *eq-cmset-def*)

lemma *type-definition-cmset*: ‹*type-definition cmcount* (*inv cmcount*) {*f*. *countable* {*x*. *f x* ≠ *0*}}›
by *standard* (*auto simp*: *inj-cmcount in-range-cmcount inv-f-f f-inv-into-f countable-nonzero-cmcount*)

corec (*friend*) *linterleave* **where**
‹*linterleave lxs lys* = (*case* (*lxs*, *lys*) *of*
(*LCons x lxs'*, *LCons y lys'*) ⇒ *LCons x* (*LCons y* (*linterleave lxs' lys'*))
| (*LCons x lxs'*, *LNil*) ⇒ *LCons x lxs'*
| (*LNil*, *LCons y lys'*) ⇒ *LCons y lys'*
| (*LNil*, *LNil*) ⇒ *LNil*)›
simps-of-case *linterleave-simps*[*simp*]: *linterleave.code*[*unfolded prod.case*]

lemma *linterleave-LNil*[*simp*]:
‹*linterleave LNil lys* = *lys*›
‹*linterleave lys LNil* = *lys*›
by (*cases lys*; *simp*)+

lemma *linterleave-LCons1*[*simp*]:
‹*linterleave* (*LCons x lxs*) *lys* = *LCons x* (*linterleave lys lxs*)›
proof (*coinduction arbitrary*: *x lxs lys rule*: *llist.coinduct-upto*)
case *Eq-llist*
then show *?case*
by (*subst* (*9 10*) *linterleave.code*)
(*auto 5 0 split*: *llist.splits intro*: *llist.cong-intros*)
qed

lemma *lset-linterleave1*:
‹*x* ∈ *lset* (*linterleave lxs lys*) ⟹
x ∈ *lset lxs* ∪ *lset lys*›
proof (*induct* ‹*linterleave lxs lys*› *arbitrary*: *lxs lys rule*: *lset-induct*)
case (*find lxs' lxs*)
then show *?case*
by (*cases lxs*) *auto*
next
case (*step x' lxs' lxs*)
then show *?case*
by (*cases lxs*) *auto*
qed

lemma *lset-linterleave2*:
‹*x* ∈ *lset lxs* ⟹ *x* ∈ *lset* (*linterleave lxs lys*)›
proof (*induct lxs arbitrary*: *lys rule*: *lset-induct*)
case (*find lxs*)
then show *?case*

```
    by auto
next
  case (step x' lxs')
  then show ?case
    by (cases lys) (auto split: llist.splits)
qed

lemma lset-linterleave3:
  ‹x ∈ lset lys ⟹
   x ∈ lset (linterleave lxs lys)›
proof (induct lys arbitrary: lxs rule: lset-induct)
  case (find lys)
  then show ?case
    by (cases lxs) auto
next
  case (step x' lxs')
  then show ?case
    by (cases lxs) (auto split: llist.splits)
qed

lemma lset-linterleave[simp]:
  ‹lset (linterleave lxs lys) = lset lxs ∪ lset lys›
  by (auto dest: lset-linterleave1 lset-linterleave2 lset-linterleave3)

lemma ldistinct-linterleave: ‹ldistinct lxs ⟹ ldistinct lys ⟹ lset lxs ∩ lset lys =
{} ⟹ ldistinct (linterleave lxs lys)›
proof (coinduction arbitrary: lxs lys)
  case (ldistinct lxs lys)
  then show ?case
    by (cases lxs; cases lys; force intro!: linterleave-LCons1[symmetric])
qed

coinductive linfinite where
  ‹linfinite lxs ⟹ linfinite (LCons x lxs)›

inductive linfinite-cong for R where
  ‹R lxs ⟹ linfinite-cong R lxs›
| ‹linfinite lxs ⟹ linfinite-cong R lxs›
| ‹linfinite-cong R lxs ⟹ linfinite-cong R (LCons x lxs)›

lemma linfinite-coinduct-upto[consumes 1, case-names linfinite]:
  assumes ‹X lxs› ‹⋀lys. X lys ⟹ ∃lxs x. lys = LCons x lxs ∧ linfinite-cong X
lxs›
  shows ‹linfinite lxs›
proof (rule linfinite.coinduct[of ‹linfinite-cong X›])
  show ‹linfinite-cong X lxs› by (rule linfinite-cong.intros(1)) (rule assms(1))
next
  fix lxs
  assume ‹linfinite-cong X lxs›
```

then show ‹∃ *lxs'* *x*. *lxs* = *LCons x lxs'* ∧ (*linfinite-cong X lxs'* ∨ *linfinite lxs'*)›
proof (*induct lxs rule*: *linfinite-cong.induct*)
case (*1 lxs*)
from *assms*(*2*)[*OF 1*] **show** *?case* **by** (*auto intro*: *linfinite-cong.intros*(*3*))
next
case (*2 lxs*)
then show *?case* **by** (*auto elim*: *linfinite.cases intro*: *linfinite-cong.intros*(*2*))
next
case (*3 lxs y*)
then show *?case* **by** (*auto intro*: *linfinite-cong.intros*)
qed
qed

inductive-cases *linfinite-LNilE*[*elim*!]: ‹*linfinite LNil*›
inductive-cases *linfinite-LConsE*[*elim*!]: ‹*linfinite* (*LCons x lxs*)›

lemma *linfinite-linterleaveL*: ‹*linfinite lxs* ⟹ *linfinite* (*linterleave lxs lys*)›
proof (*coinduction arbitrary*: *lxs lys rule*: *linfinite-coinduct-upto*)
case (*linfinite lxs lys*)
then show *?case* **by** (*cases lxs*; *cases lys*; *auto intro*: *linfinite-cong.intros*)
qed

lemma *linfinite-linterleaveR*: ‹*linfinite lys* ⟹ *linfinite* (*linterleave lxs lys*)›
proof (*coinduction arbitrary*: *lxs lys rule*: *linfinite-coinduct-upto*)
case (*linfinite lxs lys*)
then show *?case* **by** (*cases lxs*; *cases lys*; *auto intro*: *linfinite-cong.intros*)
qed

lemma *lfinite-imp-not-linfinite*: ‹*lfinite lxs* ⟹ ¬ *linfinite lxs*›
by (*induct lxs rule*: *lfinite-induct*) (*auto simp*: *lnull-def neq-LNil-conv*)

lemma *not-lfinite-imp-linfinite*: ‹¬ *lfinite lxs* ⟹ *linfinite lxs*›
proof (*coinduction arbitrary*: *lxs*)
case (*linfinite lxs*)
then show *?case* **by** (*cases lxs*) *auto*
qed

lemma *linfinite-eq-not-lfinite*: ‹*linfinite lxs* ⟷ ¬ *lfinite lxs*›
using *lfinite-imp-not-linfinite not-lfinite-imp-linfinite* **by** *blast*
lemma *linfinite-eq-llength*: ‹*linfinite lxs* ⟷ *llength lxs* = ∞›
using *lfinite-imp-not-linfinite llength-eq-infty-conv-lfinite not-lfinite-imp-linfinite*
by *blast*

lemma *llength-linterleave*[*simp*]: ‹*llength* (*linterleave lxs lys*) = *llength lxs* + *llength lys*›
proof (*cases* ‹*linfinite lxs*›)
case *True*
then show *?thesis*

```
  proof (cases ‹linfinite lys›)
    case True
    then show ?thesis
    by (metis ‹linfinite lxs› linfinite-eq-llength linfinite-linterleaveL plus-enat-simps(3))
  next
    case False
    then show ?thesis
    by (metis ‹linfinite lxs› linfinite-eq-llength linfinite-linterleaveL plus-enat-simps(2))
  qed
next
  case False
  then have hlxs: ‹lfinite lxs› by (simp add: linfinite-eq-not-lfinite)
  show ?thesis
  proof (cases ‹linfinite lys›)
    case True
   then show ?thesis by (metis linfinite-eq-llength linfinite-linterleaveR plus-enat-simps(3))
  next
    case False
    then have hlys: ‹lfinite lys› by (simp add: linfinite-eq-not-lfinite)
    from hlxs hlys show ?thesis
    proof (induct lxs arbitrary: lys rule: lfinite-induct)
      case (LNil xs)
      then show ?case by (auto simp: lnull-def neq-LNil-conv)
    next
      case (LCons lxs)
       note outer-IH = ‹⋀lys. lfinite lys ⟹ llength (linterleave (ltl lxs) lys) =
llength (ltl lxs) + llength lys›
      from ‹lfinite lys› show ?case
      proof (induct lys rule: lfinite-induct)
        case LNil
          with ‹¬ lnull lxs› show ?case by (auto simp: lnull-def neq-LNil-conv
add.commute iadd-Suc-right)
      next
        case (LCons lys)
        show ?case
        proof −
          from ‹¬ lnull lxs› obtain x xs′ where lxs-eq: ‹lxs = LCons x xs′›
            by (cases lxs) (auto simp: lnull-def)
          from ‹¬ lnull lys› obtain y lys′ where lys-eq: ‹lys = LCons y lys′›
            by (cases lys) (auto simp: lnull-def)
          from ‹lfinite lys› have fin-lys′: ‹lfinite lys′›
            by (simp add: lys-eq)
           from fin-lys′ have ih-lys′: ‹llength (linterleave xs′ lys′) = llength xs′ +
llength lys′›
            using outer-IH[of lys′] lxs-eq by simp
          show ?case
               by (simp only: lxs-eq lys-eq linterleave-simps llength-LCons ih-lys′
iadd-Suc-right add.commute)
        qed
```

```
    qed
  qed
 qed
qed

lemma lnth-linterleave-swap:
 ‹lnth (linterleave lxs lys) i ∉ lset lys ⟹ i < llength (linterleave lxs lys) ⟹
  ∃j < min (Suc i) (llength lxs). lnth (linterleave lxs lys) i = lnth lxs j›
proof (induct i arbitrary: lxs lys rule: less-induct)
 case (less i lxs lys)
 show ?case
 proof (cases i)
  case 0
  with less.prems show ?thesis
   by (cases lxs; cases lys) (auto simp: enat-0)
 next
  case (Suc j)
  with less.prems less.hyps show ?thesis
  proof (cases lxs)
   case LNil
   then show ?thesis
   proof (cases lys)
    case LNil
    with less.prems ‹lxs = LNil› show ?thesis by (auto simp: Suc-ile-eq)
   next
    case (LCons b lys')
    with less.prems ‹lxs = LNil› ‹i = Suc j› show ?thesis
     by (auto simp: in-lset-conv-lnth lnth-LCons' Suc-ile-eq)
   qed
  next
   case (LCons a lxs')
   show ?thesis
   proof (cases lys)
    case LNil
    with less.prems less.hyps ‹lxs = LCons a lxs'› show ?thesis
         by (auto simp: Suc-ile-eq in-lset-conv-lnth lnth-LCons' gr0-conv-Suc
less-Suc-eq-le)
   next
    case (LCons b lys')
    note lxs-eq = ‹lxs = LCons a lxs'› and lys-eq = ‹lys = LCons b lys'›
    show ?thesis
    proof (cases j)
     case 0
     with less.prems lxs-eq lys-eq ‹i = Suc j› show ?thesis
      by (auto simp: Suc-ile-eq in-lset-conv-lnth lnth-LCons')
    next
     case (Suc m)
     with less.prems less.hyps lxs-eq lys-eq ‹i = Suc j› have
      IH: ‹lnth (linterleave lxs' lys') m ∉ lset lys' ⟹
```

enat m < llength (linterleave lxs' lys') ⟹
∃ j'. enat j' < min (enat (Suc m)) (llength lxs') ∧
lnth (linterleave lxs' lys') m = lnth lxs' j'›
by (*auto simp*: *Suc-ile-eq less-Suc-eq-le*)
from *less.prems lxs-eq lys-eq ‹i = Suc j› ‹j = Suc m›*
have *notin*: *‹lnth (linterleave lxs' lys') m ∉ lset lys'›*
by (*auto simp*: *Suc-ile-eq in-lset-conv-lnth lnth-LCons'*)
from *less.prems lxs-eq lys-eq ‹i = Suc j› ‹j = Suc m›*
have *len*: *‹enat m < llength (linterleave lxs' lys')›*
by (*auto simp*: *Suc-ile-eq*)
from *IH*[*OF notin len*] **obtain** *j'* **where**
j'-bound: *‹enat j' < min (enat (Suc m)) (llength lxs')›*
and *j'-eq*: *‹lnth (linterleave lxs' lys') m = lnth lxs' j'›*
by *blast*
from *j'-bound j'-eq ‹i = Suc j› ‹j = Suc m› lxs-eq lys-eq less.prems*
show *?thesis*
by (*auto simp*: *Suc-ile-eq in-lset-conv-lnth lnth-LCons' intro*!: *exI*[*of* - *‹Suc j'›*])
qed
qed
qed
qed
qed

lemma *ldropWhile-eq-ldropn*:
‹∃ x ∈ lset lxs. ¬ P x ⟹ ∃ n. enat n = llength (ltakeWhile P lxs) ∧ ldropWhile P lxs = ldropn n lxs›
by (*metis ldropWhile-eq-ldrop ldrop-enat lfinite-llength-enat lfinite-ltakeWhile*)

lemma *ltl-linterleave*[*simp*]: *‹¬lnull lxs ⟹ ltl (linterleave lxs lys) = linterleave lys (ltl lxs)›*
by (*cases lxs*) *auto*

lemma *linfinite-ltl*[*simp*]: *‹linfinite lxs ⟹ linfinite (ltl lxs)›*
by (*cases lxs*) *auto*

lemma *linfinite-ldropn*[*simp*]: *‹linfinite lxs ⟹ linfinite (ldropn n lxs)›*
by (*induct n arbitrary*: *lxs*) (*auto simp*: *ldropn-Suc split*: *llist.splits*)

lemma *linfinite-not-lnull*: *‹linfinite lxs ⟹ ¬ lnull lxs›*
by (*cases lxs*) *auto*

lemma *ldropn-linterleave*: *‹linfinite lxs ⟹ linfinite lys ⟹ ldropn n (linterleave lxs lys) =*
*(if ∃ m. n = 2 * m then linterleave (ldropn (n div 2) lxs) (ldropn (n div 2) lys) else*
linterleave (ldropn (n div 2) lys) (ldropn (Suc (n div 2)) lxs))›
proof (*induct n arbitrary*: *lxs lys*)
case *0*

```
  then show ?case by simp
next
  case (Suc n lxs lys)
  from Suc.prems(1) obtain a lxs' where lxs-eq: ‹lxs = LCons a lxs'› and linf-lxs':
‹linfinite lxs'›
    by (cases lxs) (auto dest: linfinite-not-lnull)
  have step: ‹ldropn (Suc n) (linterleave lxs lys) = ldropn n (linterleave lys lxs')›
    unfolding lxs-eq by (simp add: ldropn-Suc)
  have IH: ‹ldropn n (linterleave lys lxs') =
   (if ∃ m. n = 2 * m then linterleave (ldropn (n div 2) lys) (ldropn (n div 2) lxs')
    else linterleave (ldropn (n div 2) lxs') (ldropn (Suc (n div 2)) lys))›
    by (rule Suc.hyps[OF Suc.prems(2) linf-lxs'])
  show ?case
    unfolding step lxs-eq linterleave-LCons1 ldropn-Suc-LCons IH
   by (metis even-Suc even-Suc-div-two even-mult-iff even-two-times-div-two ldropn-Suc-LCons
odd-Suc-div-two)
qed

lemma llength-ltakeWhile-linterleave-ge:
  ‹enat (2 * m) ≤ llength (ltakeWhile ((≠) x) (linterleave lxs lys)) ⟹
  x ∉ set (ltaken m lxs) ∧ x ∉ set (ltaken m lys)›
proof (induct m arbitrary: lxs lys)
  case 0
  then show ?case by simp
next
  case (Suc m lxs lys)
  show ?case
  proof (cases lxs)
    case LNil
    then have hlen-lys: ‹enat (2 * Suc m) ≤ llength (ltakeWhile ((≠) x) lys)›
      using Suc.prems by simp
   have xnot: ‹x ∉ lset (ltakeWhile ((≠) x) lys)› by (metis (full-types) lset-ltakeWhileD)
    with hlen-lys llength-ltakeWhile-le[of ‹(≠) x› lys] have ‹x ∉ set (ltaken (Suc
m) lys)›
    proof −
      have hlys-len: ‹enat (Suc m) ≤ llength lys›
        by (rule order-trans[OF - llength-ltakeWhile-le[of ‹(≠) x› lys]])
           (rule order-trans[OF - hlen-lys], simp)
      have neq: ‹⋀i. i < Suc m ⟹ lnth lys i ≠ x›
      proof (intro notI)
        fix i assume h-i: ‹i < Suc m› ‹lnth lys i = x›
        have ‹enat i < enat (2 * Suc m)› using h-i(1) by simp
        hence hi-lt: ‹enat i < llength (ltakeWhile ((≠) x) lys)›
          using hlen-lys by (rule order-less-le-trans)
        hence ‹lnth lys i ∈ lset (ltakeWhile ((≠) x) lys)›
          by (auto simp: in-lset-conv-lnth ltakeWhile-nth)
        then have ‹lnth lys i ≠ x› using lset-ltakeWhileD by fastforce
        with h-i(2) show False by simp
      qed
```

```
    show ?thesis
      unfolding set-ltaken-conv[OF hlys-len]
      by (clarsimp; use neq in fastforce)
  qed
  then show ?thesis using LNil by simp
 next
  case (LCons a lxs') note lxs-eq = this
  show ?thesis
  proof (cases lys)
    case LNil
    then have hlen-lxs: ‹enat (2 * Suc m) ≤ llength (ltakeWhile ((≠) x) lxs)›
      using Suc.prems lxs-eq by simp
   have xnot: ‹x ∉ lset (ltakeWhile ((≠) x) lxs)› by (metis (full-types) lset-ltakeWhileD)
    with hlen-lxs llength-ltakeWhile-le[of ‹(≠) x› lxs] have ‹x ∉ set (ltaken (Suc
m) lxs)›
    proof −
      have hlxs-len: ‹enat (Suc m) ≤ llength lxs›
        by (rule order-trans[OF - llength-ltakeWhile-le[of ‹(≠) x› lxs]])
          (rule order-trans[OF - hlen-lxs], simp)
      have neq: ‹⋀i. i < Suc m ⟹ lnth lxs i ≠ x›
      proof (intro notI)
        fix i assume h-i: ‹i < Suc m› ‹lnth lxs i = x›
        have ‹enat i < enat (2 * Suc m)› using h-i(1) by simp
        hence hi-lt: ‹enat i < llength (ltakeWhile ((≠) x) lxs)›
          using hlen-lxs by (rule order-less-le-trans)
        hence ‹lnth lxs i ∈ lset (ltakeWhile ((≠) x) lxs)›
          by (auto simp: in-lset-conv-lnth ltakeWhile-nth)
        then have ‹lnth lxs i ≠ x› using lset-ltakeWhileD by fastforce
        with h-i(2) show False by simp
      qed
      show ?thesis
        unfolding set-ltaken-conv[OF hlxs-len]
        by (clarsimp; use neq in fastforce)
    qed
    then show ?thesis using LNil lxs-eq by simp
  next
    case (LCons b lys') note lys-eq = this
    have hlen: ‹enat (2 * Suc m) ≤ llength (ltakeWhile ((≠) x)
        (LCons a (LCons b (linterleave lxs' lys'))))›
      using Suc.prems unfolding lxs-eq lys-eq by simp
    have ha: ‹a ≠ x›
      using hlen by (auto simp: Suc-ile-eq enat-0-iff split: if-splits)
    have hlen': ‹enat (2 * Suc m − 1) ≤ llength (ltakeWhile ((≠) x)
        (LCons b (linterleave lxs' lys'))))›
      using hlen ha by (simp add: Suc-ile-eq)
    have hb: ‹b ≠ x›
      using hlen' by (auto simp: Suc-ile-eq enat-0-iff split: if-splits)
     have hlen'': ‹enat (2 * m) ≤ llength (ltakeWhile ((≠) x) (linterleave lxs'
lys'))›
```

```
      using hlen' hb by (simp add: Suc-ile-eq)
    note IH = Suc.hyps[OF hlen'']
    show ?thesis using ha hb IH unfolding lxs-eq lys-eq by simp
   qed
  qed
qed

lemma llength-ltakeWhile-linterleave-eq:
  ‹enat (2 * m) = llength (ltakeWhile ((≠) x) (linterleave lxs lys)) ⟹
  x ∉ set (ltaken m lxs) ∧ x ∉ set (ltaken m lys)›
  by (rule llength-ltakeWhile-linterleave-ge) auto

lemma llength-ltakeWhile-linterleave-ge-Suc:
  ‹linfinite lxs ⟹ linfinite lys ⟹
   enat (Suc (2 * m)) ≤ llength (ltakeWhile ((≠) x) (linterleave lxs lys)) ⟹
  x ∉ set (ltaken (Suc m) lxs) ∧ x ∉ set (ltaken m lys)›
proof (induct m arbitrary: lxs lys)
  case 0
  then show ?case by (auto simp: enat-0-iff Suc-ile-eq split: if-splits)
next
  case (Suc m lxs lys)
  from Suc.prems(1) obtain a lxs' where lxs-eq: ‹lxs = LCons a lxs'›
    by (cases lxs) (auto dest: linfinite-not-lnull)
  from Suc.prems(2) obtain b lys' where lys-eq: ‹lys = LCons b lys'›
    by (cases lys) (auto dest: linfinite-not-lnull)
  from Suc.prems lxs-eq lys-eq have linf-tails: ‹linfinite lxs'› ‹linfinite lys'›
    by auto
  from Suc.prems lxs-eq lys-eq have prems': ‹x ≠ a› ‹x ≠ b›
    ‹enat (Suc (2 * m)) ≤ llength (ltakeWhile ((≠) x) (linterleave lxs' lys'))›
    by (auto simp: Suc-ile-eq split: if-splits)
  from Suc.hyps[OF linf-tails(1) linf-tails(2) prems'(3)]
  have IH: ‹x ∉ set (ltaken (Suc m) lxs') ∧ x ∉ set (ltaken m lys')› .
  from IH prems' lxs-eq lys-eq show ?case
    by (auto simp: Suc-ile-eq split: if-splits)
qed

lemma llength-ltakeWhile-linterleave-eq-Suc:
  ‹linfinite lxs ⟹ linfinite lys ⟹
   enat (Suc (2 * m)) = llength (ltakeWhile ((≠) x) (linterleave lxs lys)) ⟹
  x ∉ set (ltaken (Suc m) lxs) ∧ x ∉ set (ltaken m lys)›
  by (rule llength-ltakeWhile-linterleave-ge-Suc) auto

lemma count-llist-linterleave:
  ‹count-llist (linterleave lxs lys) x = count-llist lxs x + count-llist lys x›
proof ((cases ‹lfinite lxs›; cases ‹lfinite lys›), goal-cases FF FI IF II)
  case FF
  then show ?case
  proof (induct lxs arbitrary: lys)
    case (lfinite-LConsI lxs x)
```

```
    then show ?case
      by (cases lys) (auto simp: iadd-Suc-right ac-simps)
  qed simp
next
  case FI
  then show ?case
  proof (induct lxs arbitrary: lys)
    case (lfinite-LConsI lxs x)
    then show ?case
      by (cases lys) (auto simp: iadd-Suc-right ac-simps)
  qed simp
next
  case IF
  from IF(2,1) show ?case
  proof (induct lys arbitrary: lxs)
    case (lfinite-LConsI lxs x)
    then show ?case
      by (cases lxs) (auto simp: iadd-Suc-right ac-simps)
  qed simp
next
  case II
  then show ?case
  proof (coinduction arbitrary: lxs lys)
    case eq-enat
    show ?case
    proof (cases ‹x ∈ lset (linterleave lxs lys)›)
      case False
      then show ?thesis
        using eq-enat by (auto simp: count-llist-zero-iff linfinite-eq-not-lfinite)
    next
      case True
      then have hin: ‹x ∈ lset (linterleave lxs lys)› .
      obtain n where
        h-len: ‹enat n = llength (ltakeWhile ((≠) x) (linterleave lxs lys))› and
        h-drop: ‹ldropWhile ((≠) x) (linterleave lxs lys) = ldropn n (linterleave lxs
lys)›
        using ldropWhile-eq-ldropn[of ‹linterleave lxs lys› ‹(≠) x›] hin by auto
      have h-lhd: ‹lhd (ldropWhile ((≠) x) (linterleave lxs lys)) = x›
      proof −
          have hlt: ‹llength (ltakeWhile ((≠) x) (linterleave lxs lys)) < llength
(linterleave lxs lys)›
        by (subst llength-ltakeWhile-lt-iff) (use hin in auto)
        have h1: ‹lhd (ldropWhile ((≠) x) (linterleave lxs lys)) =
          lnth (linterleave lxs lys) (the-enat (llength (ltakeWhile ((≠) x) (linterleave
lxs lys))))›
        by (rule lhd-ldropWhile-conv-lnth) (use hin in auto)
        have h2: ‹¬ ((≠) x) (lnth (linterleave lxs lys)
          (the-enat (llength (ltakeWhile ((≠) x) (linterleave lxs lys)))))›
        by (rule lnth-llength-ltakeWhile[OF hlt])
```

```
  from h1 h2 show ?thesis by simp
qed

show ?thesis
proof (cases ‹∃ m. n = 2 * m›)
  case True
  then obtain m :: nat where hm: ‹n = 2 * m› by auto
  have h-drop2: ‹ldropWhile ((≠) x) (linterleave lxs lys) =
      linterleave (ldropn m lxs) (ldropn m lys)›
    using h-drop hm eq-enat
    by (simp add: ldropn-linterleave linfinite-eq-not-lfinite)
  have h-hd-lxs: ‹lhd (ldropn m lxs) = x›
  proof −
    have hnn: ‹¬ lnull (ldropn m lxs)›
   by (metis eq-enat(1) linfinite-ldropn linfinite-not-lnull linfinite-eq-not-lfinite)
    have ‹lhd (linterleave (ldropn m lxs) (ldropn m lys)) = lhd (ldropn m lxs)›
      by (cases ‹ldropn m lxs›) (use hnn in auto)
    with h-lhd h-drop2 show ?thesis by simp
  qed
  have h-taken: ‹x ∉ set (ltaken m lxs) ∧ x ∉ set (ltaken m lys)›
  by (rule llength-ltakeWhile-linterleave-eq[of m]) (simp add: h-len[symmetric]
hm)
  have h-lxs-eq: ‹ldropn m lxs = LCons x (ltl (ldropn m lxs))›
  proof −
    have hnn: ‹¬ lnull (ldropn m lxs)›
   by (metis eq-enat(1) linfinite-ldropn linfinite-not-lnull linfinite-eq-not-lfinite)
    from lhd-LCons-ltl[OF hnn] h-hd-lxs show ?thesis by simp
  qed
  have h-lxs-cnt: ‹count-llist lxs x = eSuc (count-llist (ltl (ldropn m lxs)) x)›
  proof −
    have heq-lxs: ‹lxs = lappend (llist-of (ltaken m lxs)) (ldropn m lxs)›
      by (rule ltaken-ldropn-decomp)
    have hcnt-lxs: ‹count-llist lxs x = count-llist (ldropn m lxs) x›
      by (subst heq-lxs, simp only: count-llist-lappend lfinite-llist-of if-True,
          simp add: h-taken count-list-0-iff zero-enat-def)
    show ?thesis
      by (simp only: hcnt-lxs, subst h-lxs-eq, simp)
  qed
  have h-lys-cnt: ‹count-llist lys x = count-llist (ldropn m lys) x›
  proof −
    have heq-lys: ‹lys = lappend (llist-of (ltaken m lys)) (ldropn m lys)›
      by (rule ltaken-ldropn-decomp)
    show ?thesis
      by (subst heq-lys, simp only: count-llist-lappend lfinite-llist-of if-True,
          simp add: h-taken count-list-0-iff zero-enat-def)
  qed
  have h-inter-cnt: ‹count-llist (linterleave lxs lys) x =
      eSuc (count-llist (linterleave (ldropn m lys) (ltl (ldropn m lxs))) x)›
  proof −
```

```
      have heq: ‹ldropWhile ((≠) x) (linterleave lxs lys) =
          LCons x (linterleave (ldropn m lys) (ltl (ldropn m lxs)))›
        by (subst h-drop2, subst h-lxs-eq, simp)
      have h-tl: ‹ltl (ldropWhile ((≠) x) (linterleave lxs lys)) =
          linterleave (ldropn m lys) (ltl (ldropn m lxs))›
        by (simp add: heq)
      from count-llist-ctr(2)[OF hin] h-tl show ?thesis
        by simp
    qed
    show ?thesis
    proof (rule conjI)
      show ‹(count-llist (linterleave lxs lys) x = 0) = (count-llist lxs x +
count-llist lys x = 0)›
        by (simp add: h-inter-cnt h-lxs-cnt)
    next
    show ‹count-llist (linterleave lxs lys) x ≠ 0 ⟶ count-llist lxs x + count-llist
lys x ≠ 0 ⟶
      (∃ lxs' lys'. epred (count-llist (linterleave lxs lys) x) = count-llist (linterleave
lxs' lys') x ∧
                    epred (count-llist lxs x + count-llist lys x) = count-llist lxs' x
+ count-llist lys' x ∧
                    ¬ lfinite lxs' ∧ ¬ lfinite lys') ∨
       epred (count-llist (linterleave lxs lys) x) = epred (count-llist lxs x +
count-llist lys x)›
      proof (intro impI disjI1)
        show ‹∃ lxs' lys'. epred (count-llist (linterleave lxs lys) x) = count-llist
(linterleave lxs' lys') x ∧
                    epred (count-llist lxs x + count-llist lys x) = count-llist lxs' x
+ count-llist lys' x ∧
                    ¬ lfinite lxs' ∧ ¬ lfinite lys'›
        proof (rule exI[of - ‹ldropn m lys›], rule exI[of - ‹ltl (ldropn m lxs)›],
intro conjI)
        show ‹epred (count-llist (linterleave lxs lys) x) = count-llist (linterleave
(ldropn m lys) (ltl (ldropn m lxs))) x›
          by (simp add: h-inter-cnt)
        show ‹epred (count-llist lxs x + count-llist lys x) = count-llist (ldropn
m lys) x + count-llist (ltl (ldropn m lxs)) x›
          by (simp add: h-lxs-cnt h-lys-cnt iadd-Suc-right add.commute)
        show ‹¬ lfinite (ldropn m lys)›
          using eq-enat by (simp add: linfinite-eq-not-lfinite)
        show ‹¬ lfinite (ltl (ldropn m lxs))›
          using eq-enat by (simp add: linfinite-eq-not-lfinite)
        qed
      qed
    qed
  next
    case False
    have hodd: ‹¬ (∃ m. n = 2 * m)› using False .
    have hm: ‹n = 2 * (n div 2) + 1›
```

```
      using hodd by arith
    have h-drop2: ‹ldropWhile ((≠) x) (linterleave lxs lys) =
      linterleave (ldropn (n div 2) lys) (ldropn (Suc (n div 2)) lxs)›
      using h-drop hodd eq-enat
      by (simp add: ldropn-linterleave linfinite-eq-not-lfinite)
    have h-hd-lys: ‹lhd (ldropn (n div 2) lys) = x›
    proof −
      have hnn: ‹¬ lnull (ldropn (n div 2) lys)›
     using eq-enat by (metis linfinite-ldropn linfinite-not-lnull linfinite-eq-not-lfinite)
      have ‹lhd (linterleave (ldropn (n div 2) lys) (ldropn (Suc (n div 2)) lxs))
=
        lhd (ldropn (n div 2) lys)›
       by (cases ‹ldropn (n div 2) lys›) (use hnn in auto)
      with h-lhd h-drop2 show ?thesis by simp
    qed
    have h-taken: ‹x ∉ set (ltaken (Suc (n div 2)) lxs) ∧ x ∉ set (ltaken (n div
2) lys)›
    proof −
    have hlinf-lxs: ‹linfinite lxs› using eq-enat by (simp add: linfinite-eq-not-lfinite)
    have hlinf-lys: ‹linfinite lys› using eq-enat by (simp add: linfinite-eq-not-lfinite)
        have hlen: ‹enat (Suc (2 * (n div 2))) = llength (ltakeWhile ((≠) x)
(linterleave lxs lys))›
        using h-len hm by simp
     from llength-ltakeWhile-linterleave-eq-Suc[OF hlinf-lxs hlinf-lys hlen] show
?thesis .
    qed
    have h-lys-eq: ‹ldropn (n div 2) lys = LCons x (ltl (ldropn (n div 2) lys))›
    proof −
      have hnn: ‹¬ lnull (ldropn (n div 2) lys)›
     using eq-enat by (metis linfinite-ldropn linfinite-not-lnull linfinite-eq-not-lfinite)
      from lhd-LCons-ltl[OF hnn] h-hd-lys show ?thesis by simp
    qed
     have h-lys-cnt: ‹count-llist lys x = eSuc (count-llist (ltl (ldropn (n div 2)
lys)) x)›
    proof −
      have heq: ‹lys = lappend (llist-of (ltaken (n div 2) lys)) (ldropn (n div 2)
lys)›
        by (rule ltaken-ldropn-decomp)
      have hcnt: ‹count-llist lys x = count-llist (ldropn (n div 2) lys) x›
        by (subst heq, simp only: count-llist-lappend lfinite-llist-of if-True,
            simp add: h-taken count-list-0-iff zero-enat-def)
      show ?thesis
        by (simp only: hcnt, subst h-lys-eq, simp)
    qed
    have h-lxs-cnt: ‹count-llist lxs x = count-llist (ldropn (Suc (n div 2)) lxs) x›
     proof −
      have hx: ‹x ∉ set (ltaken (Suc (n div 2)) lxs)› using h-taken by blast
     have heq: ‹lxs = lappend (llist-of (ltaken (Suc (n div 2)) lxs)) (ldropn (Suc
(n div 2)) lxs)›
```

by (*rule ltaken-ldropn-decomp*)
have *hA*: ‹*count-llist* (*llist-of* (*ltaken* (*Suc* (*n div 2*)) *lxs*)) *x* = *0*›
by (*simp del*: *ltaken.simps add*: *count-list-0-iff hx zero-enat-def*)
show *?thesis*
by (*subst heq*, *simp only*: *count-llist-lappend lfinite-llist-of if-True hA add-0-left*)
qed
have *h-inter-cnt*: ‹*count-llist* (*linterleave lxs lys*) *x* =
eSuc (*count-llist* (*linterleave* (*ldropn* (*Suc* (*n div 2*)) *lxs*) (*ltl* (*ldropn* (*n div 2*) *lys*))) *x*)›
proof −
have *heq*: ‹*ldropWhile* ((≠) *x*) (*linterleave lxs lys*) =
LCons x (*linterleave* (*ldropn* (*Suc* (*n div 2*)) *lxs*) (*ltl* (*ldropn* (*n div 2*) *lys*)))›
by (*subst h-drop2*, *subst h-lys-eq*, *simp*)
have *h-tl*: ‹*ltl* (*ldropWhile* ((≠) *x*) (*linterleave lxs lys*)) =
linterleave (*ldropn* (*Suc* (*n div 2*)) *lxs*) (*ltl* (*ldropn* (*n div 2*) *lys*))›
by (*simp add*: *heq*)
from *count-llist-ctr*(*2*)[*OF hin*] *h-tl* **show** *?thesis*
by *simp*
qed
show *?thesis*
proof (*rule conjI*)
show ‹(*count-llist* (*linterleave lxs lys*) *x* = *0*) = (*count-llist lxs x* + *count-llist lys x* = *0*)›
by (*simp add*: *h-inter-cnt h-lys-cnt*)
next
show ‹*count-llist* (*linterleave lxs lys*) *x* ≠ *0* ⟶ *count-llist lxs x* + *count-llist lys x* ≠ *0* ⟶
(∃ *lxs′ lys′*. *epred* (*count-llist* (*linterleave lxs lys*) *x*) = *count-llist* (*linterleave lxs′ lys′*) *x* ∧
epred (*count-llist lxs x* + *count-llist lys x*) = *count-llist lxs′ x* + *count-llist lys′ x* ∧
¬ *lfinite lxs′* ∧ ¬ *lfinite lys′*) ∨
epred (*count-llist* (*linterleave lxs lys*) *x*) = *epred* (*count-llist lxs x* + *count-llist lys x*)›
proof (*intro impI disjI1*)
show ‹∃ *lxs′ lys′*. *epred* (*count-llist* (*linterleave lxs lys*) *x*) = *count-llist* (*linterleave lxs′ lys′*) *x* ∧
epred (*count-llist lxs x* + *count-llist lys x*) = *count-llist lxs′ x* + *count-llist lys′ x* ∧
¬ *lfinite lxs′* ∧ ¬ *lfinite lys′*›
proof (*rule exI*[*of* - ‹*ldropn* (*Suc* (*n div 2*)) *lxs*›],
rule exI[*of* - ‹*ltl* (*ldropn* (*n div 2*) *lys*)›], *intro conjI*)
show ‹*epred* (*count-llist* (*linterleave lxs lys*) *x*) = *count-llist* (*linterleave* (*ldropn* (*Suc* (*n div 2*)) *lxs*) (*ltl* (*ldropn* (*n div 2*) *lys*))) *x*›
by (*simp add*: *h-inter-cnt*)
show ‹*epred* (*count-llist lxs x* + *count-llist lys x*) = *count-llist* (*ldropn* (*Suc* (*n div 2*)) *lxs*) *x* + *count-llist* (*ltl* (*ldropn* (*n div 2*) *lys*)) *x*›

```
            by (simp add: h-lxs-cnt h-lys-cnt iadd-Suc-right)
          show ‹¬ lfinite (ldropn (Suc (n div 2)) lxs)›
            using eq-enat by (simp add: linfinite-eq-not-lfinite)
          show ‹¬ lfinite (ltl (ldropn (n div 2) lys))›
            using eq-enat by (simp add: linfinite-eq-not-lfinite)
        qed
      qed
    qed
  qed
 qed
qed
```

lemma *lfinite-linterleave*[*simp*]: ‹*lfinite* (*linterleave lxs lys*) ⟷ *lfinite lxs* ∧ *lfinite lys*›
by (*metis enat-add-left-cancel llength-eq-infty-conv-lfinite llength-linterleave plus-enat-simps*(*3*))

lift-definition *cmadd* :: ‹'*a cmset* ⇒ '*a cmset* ⇒ '*a cmset*› **is** *linterleave*
by (*auto simp*: *eq-cmset-def count-llist-linterleave*)

lemma *cmcount-cmadd*[*simp*]: ‹*cmcount* (*cmadd M N*) *x* = *cmcount M x* + *cmcount N x*›
by *transfer* (*auto simp*: *count-llist-linterleave*)

lemma *count-llist-lmap*: ‹*count-llist* (*lmap f lxs*) *b* = *isum* (*count-llist lxs*) {*a*. *f a* = *b*}›
proof −
have *h-ind*: ‹⋀*i*. *isum* (λ*a*. *if enat i* < *llength lxs* ∧ *lnth lxs i* = *a then 1 else 0*) {*a*. *f a* = *b*} =
(*if enat i* < *llength lxs* ∧ *f* (*lnth lxs i*) = *b then 1 else 0*)›
proof −
fix *i*
show ‹*isum* (λ*a*. *if enat i* < *llength lxs* ∧ *lnth lxs i* = *a then 1 else 0*) {*a*. *f a* = *b*} =
(*if enat i* < *llength lxs* ∧ *f* (*lnth lxs i*) = *b then 1 else 0*)›
proof (*cases* ‹*enat i* < *llength lxs* ∧ *f* (*lnth lxs i*) = *b*›)
case *True*
then have *himem*: ‹*lnth lxs i* ∈ {*a*. *f a* = *b*}› **by** *simp*
have ‹*isum* (λ*a*. *if lnth lxs i* = *a then 1 else 0*) {*a*. *f a* = *b*} =
isum (λ*a*. *if lnth lxs i* = *a then 1 else 0*) ({*a*. *f a* = *b*} − {*lnth lxs i*}) + *1*›
by (*subst insert-Diff*[*OF himem*, *symmetric*], *subst isum-insert*) (*auto simp*: *isum-eq-0-iff*)
also have ‹*isum* (λ*a*. *if lnth lxs i* = *a then 1 else 0*) ({*a*. *f a* = *b*} − {*lnth lxs i*}) = *0*›
by (*auto simp*: *isum-eq-0-iff*)

finally show *?thesis* **using** *True* **by** *simp*
next
case *False*
then show *?thesis* **by** (*auto simp*: *isum-eq-0-iff*)
qed
qed
have *step1*: ‹*count-llist* (*lmap f lxs*) *b* = *isum* (λ*i*. *if enat i* < *llength lxs* ∧ *f* (*lnth lxs i*) = *b then 1 else 0*) *UNIV*›
by (*subst count-llist-isum*, *rule isum-cong*) *auto*
have *step2*: ‹*isum* (*count-llist lxs*) {*a*. *f a* = *b*} =
isum (λ*i*. *if enat i* < *llength lxs* ∧ *f* (*lnth lxs i*) = *b then 1 else 0*) *UNIV*›
proof −
have ‹*isum* (*count-llist lxs*) {*a*. *f a* = *b*} =
isum (λ*a*. *isum* (λ*i*. *if enat i* < *llength lxs* ∧ *lnth lxs i* = *a then 1 else 0*) *UNIV*) {*a*. *f a* = *b*}›
by (*rule isum-cong*) (*auto simp*: *count-llist-isum*)
also have ‹. . . = *isum* (λ*i*. *isum* (λ*a*. *if enat i* < *llength lxs* ∧ *lnth lxs i* = *a then 1 else 0*) {*a*. *f a* = *b*}) *UNIV*›
by (*rule isum-swap*)
also have ‹. . . = *isum* (λ*i*. *if enat i* < *llength lxs* ∧ *f* (*lnth lxs i*) = *b then 1 else 0*) *UNIV*›
proof (*rule isum-cong*, *simp*)
fix *x*
show ‹*isum* (λ*a*. *if enat x* < *llength lxs* ∧ *lnth lxs x* = *a then 1 else 0*) {*a*. *f a* = *b*} =
(*if enat x* < *llength lxs* ∧ *f* (*lnth lxs x*) = *b then 1 else 0*)›
by (*rule h-ind*)
qed
finally show *?thesis* **.**
qed
show *?thesis* **by** (*simp only*: *step1 step2*)
qed

lemma *cmcount-cmimage*[*simp*]: ‹*cmcount* (*cmimage f M*) *b* = *isum* (*cmcount M*) {*a*. *f a* = *b*}›
by *transfer* (*simp add*: *count-llist-lmap*)

lift-definition *cminfinite* :: ‹'*a cmset* ⇒ *bool*› **is** *linfinite*
by (*metis eq-cmset-def eq-cmset-llength linfinite-eq-llength*)

lemma *Sup-enat-eq-inf*: ‹*Sup* (*A* :: *enat set*) = ∞ ⟷ ∞ ∈ *A* ∨ *infinite A*›
by (*auto simp*: *Sup-enat-def Max-eqI dest*: *Max-in*)

lemma *sum-eq-inf*: ‹*finite A* ⟹ *sum* (*f* :: - ⇒ *enat*) *A* = ∞ ⟷ (∃ *a* ∈ *A*. *f a* = ∞)›
by (*induct A rule*: *finite-induct*) *auto*

lemma *inf-eq-sum*: ‹*finite A* ⟹ ∞ = *sum* (*f* :: - ⇒ *enat*) *A* ⟷ (∃ *a* ∈ *A*. *f a* = ∞)›

by (*metis sum-eq-inf*)

primcorec *mk-chain* **where**
‹*mk-chain A C* = (*let a* = (*SOME a*. *a* ∈ *A*) *in LCons C* (*mk-chain* (*A* − {*a*}) (*insert a C*)))›

lemma *in-lset-mk-chainD*: ‹*X* ∈ *lset* (*mk-chain A C*) ⟹ *infinite A* ⟹ ∃ *B* ⊆ *A*. *finite B* ∧ *X* = *C* ∪ *B*›
proof (*induct X* ‹*mk-chain A C*› *arbitrary*: *A C rule*: *llist.set-induct*)
case (*LCons1 A C*)
then show *?case*
by (*subst* (*asm*) *mk-chain.code*) *blast*
next
case (*LCons2 x lxs X A C*)
let *?a* = ‹*SOME a*. *a* ∈ *A*›
have *a-in-A*: ‹*?a* ∈ *A*› **using** *LCons2.prems* **by** (*auto simp*: *some-in-eq*)
have *lxs-eq*: ‹*lxs* = *mk-chain* (*A* − {*?a*}) (*insert ?a C*)›
using *LCons2.hyps*(*1*,*3*) **by** (*metis llist.inject mk-chain.code*)
have *inf'*: ‹*infinite* (*A* − {*?a*})› **using** *LCons2.prems* **by** *simp*
obtain *B* **where** *B-sub*: ‹*B* ⊆ *A* − {*?a*}› **and** *B-fin*: ‹*finite B*› **and** *X-eq*: ‹*X* = *insert ?a C* ∪ *B*›
using *LCons2.hyps*(*2*)[*unfolded lxs-eq*, *OF - inf'*] **by** *blast*
show *?case* **using** *B-sub*
by (*intro exI*[*of - ‹insert ?a B›*]) (*auto simp*: *a-in-A B-fin X-eq*)
qed

lemma *linfinite-mk-chain*[*simp*]: ‹*linfinite* (*mk-chain A C*)›
proof (*coinduction arbitrary*: *A C*)
case (*linfinite A C*)
then show *?case*
by (*subst mk-chain.code*) *blast*
qed

coinductive *chain* **for** *R* **where**
‹*chain R LNil*›
| ‹*chain R* (*LCons x LNil*)›
| ‹*R x y* ⟹ *chain R* (*LCons y lxs*) ⟹ *chain R* (*LCons x* (*LCons y lxs*))›

inductive-cases *chain-LConsE*: ‹*chain R* (*LCons x lxs*)›

lemma *chain-mk-chain*: ‹*infinite A* ⟹ *A* ∩ *C* = {} ⟹ *chain* (⊂) (*mk-chain A C*)›
proof (*coinduction arbitrary*: *A C*)
case (*chain A C*)
let *?a* = ‹*SOME a*. *a* ∈ *A*›
have *ha*: ‹*?a* ∈ *A*›
using ‹*infinite A*› *infinite-imp-nonempty* **by** (*metis some-in-eq*)
have *ha-notin*: ‹*?a* ∉ *C*›
using ‹*A* ∩ *C* = {}› *ha* **by** *blast*

```
  let ?a' = ‹SOME a. a ∈ A − {?a}›
  have expand1: ‹mk-chain A C = LCons C (mk-chain (A − {?a}) (insert ?a C))›
    by (subst mk-chain.code) simp
  have expand2: ‹mk-chain (A − {?a}) (insert ?a C) =
       LCons (insert ?a C) (mk-chain (A − {?a} − {?a'}) (insert ?a' (insert ?a
C)))›
    by (subst mk-chain.code) simp
  show ?case
  proof (intro disjI2)
    show ‹∃ x y lxs. mk-chain A C = LCons x (LCons y lxs) ∧ x ⊂ y ∧
           ((∃ A C. LCons y lxs = mk-chain A C ∧ infinite A ∧ A ∩ C = {}) ∨
chain (⊂) (LCons y lxs))›
    proof (intro exI conjI)
      show ‹mk-chain A C =
         LCons C (LCons (insert ?a C) (mk-chain (A − {?a} − {?a'}) (insert ?a'
(insert ?a C))))›
        by (simp only: expand1 expand2)
      show ‹C ⊂ insert ?a C›
        using ha-notin by blast
      show ‹(∃ A' C'. LCons (insert ?a C) (mk-chain (A − {?a} − {?a'}) (insert
?a' (insert ?a C))) =
             mk-chain A' C' ∧ infinite A' ∧ A' ∩ C' = {}) ∨
            chain (⊂) (LCons (insert ?a C) (mk-chain (A − {?a} − {?a'}) (insert
?a' (insert ?a C))))›
      proof (intro disjI1 exI conjI)
        show ‹LCons (insert ?a C) (mk-chain (A − {?a} − {?a'}) (insert ?a'
(insert ?a C))) =
           mk-chain (A − {?a}) (insert ?a C)›
          by (rule expand2[symmetric])
        show ‹infinite (A − {?a})›
          using ‹infinite A› by simp
        show ‹(A − {?a}) ∩ (insert ?a C) = {}›
          using ‹A ∩ C = {}› ha by (auto simp: disjoint-iff)
      qed
    qed
  qed
qed

lemma chainD:
  ‹chain R lxs ⟹ enat (Suc i) < llength lxs ⟹ R (lnth lxs i) (lnth lxs (Suc i))›
proof (induct i arbitrary: lxs)
  case 0
  then show ?case by (auto simp add: zero-enat-def elim: chain.cases)
next
  case (Suc i)
  then show ?case by (fastforce simp add: zero-enat-def Suc-ile-eq elim: chain.cases)
qed

lemma chain-transD:
```

```
  assumes ‹transp R›
  shows ‹chain R lxs ⟹ i < j ⟹ enat j < llength lxs ⟹ R (lnth lxs i) (lnth
lxs j)›
proof (induct ‹j − i› arbitrary: i j lxs)
  case 0
  then show ?case by simp
next
  case (Suc x)
  note IH = Suc.hyps(1)
  show ?case
  proof (cases j)
    case 0
    with Suc.prems show ?thesis by simp
  next
    case (Suc j')
    show ?thesis
    proof (cases ‹i < j'›)
      case True
      have xeq: ‹x = j' − i› using Suc.hyps(2) Suc by simp
      have lt: ‹enat j' < llength lxs›
        using Suc.prems(3) Suc by (metis enat-ord-simps(2) lessI less-trans)
      have mid: ‹R (lnth lxs i) (lnth lxs j')›
        using IH[of j' i lxs] xeq Suc.prems(1) True lt by blast
      have step: ‹R (lnth lxs j') (lnth lxs (Suc j'))›
       using chainD[OF Suc.prems(1)] Suc.prems(3) Suc by (simp add: Suc-ile-eq)
      with mid show ?thesis using assms Suc unfolding transp-def by blast
    next
      case False
      then have ‹i = j'› using Suc.prems(2) Suc by simp
      with Suc show ?thesis
        using chainD[OF Suc.prems(1)] Suc.prems(3) by (simp add: Suc-ile-eq)
    qed
  qed
qed

lemma chain-cpo-chain: ‹transp R ⟹ chain R lxs ⟹ Complete-Partial-Order.chain
(reflclp R) (lset lxs)›
  unfolding Complete-Partial-Order.chain-def in-lset-conv-lnth
  by (metis (full-types) chain-transD in-lset-conv-lnth linorder-cases sup2CI)

lemma reflclp-subset: ‹reflclp (⊂) = (⊆)›
  by auto

lemma sum-strict-mono2-enat:
  fixes f :: ‹'a ⇒ enat›
  assumes ‹finite B› ‹A ⊆ B› ‹b ∈ B−A› ‹f b > 0› ‹∞ ∉ f ‘ B›
  shows ‹sum f A < sum f B›
proof −
  have ‹B − A ≠ {}›
```

```
    using assms(3) by blast
  have ‹sum f (B−A) > 0›
    by (rule sum-pos2) (use assms in auto)
  moreover have ‹sum f B = sum f (B−A) + sum f A›
    by (rule sum.subset-diff) (use assms in auto)
  ultimately show ?thesis
     by (metis add.commute add-diff-cancel-enat assms(1,5) enat-le-plus-same(2)
idiff-self image-iff nless-le sum-eq-inf)
qed

lemma infinite-lset-chain:
  assumes ‹linfinite lxs› ‹chain R lxs› ‹irreflp R› ‹transp R›
  shows ‹infinite (lset lxs)›
proof −
  have inj: ‹inj-on (lnth lxs) UNIV›
  proof (rule inj-onI)
    fix m n :: nat assume eq: ‹lnth lxs m = lnth lxs n›
    show ‹m = n›
    proof (rule antisym)
      show ‹m ≤ n›
      proof (rule ccontr)
        assume ‹¬ m ≤ n›
        then have ‹n < m› by simp
        with assms have ‹R (lnth lxs n) (lnth lxs m)›
          using chain-transD[OF assms(4) assms(2)] linfinite-eq-llength
          by (metis enat-ord-code(4) assms(1))
        with assms(3) eq show False by (metis irreflpD)
      qed
    next
      show ‹n ≤ m›
      proof (rule ccontr)
        assume ‹¬ n ≤ m›
        then have ‹m < n› by simp
        with assms have ‹R (lnth lxs m) (lnth lxs n)›
          using chain-transD[OF assms(4) assms(2)] linfinite-eq-llength
          by (metis enat-ord-code(4) assms(1))
        with assms(3) eq show False by (metis irreflpD)
      qed
    qed
  qed
  have ‹infinite (lnth lxs ‘ UNIV)›
    using finite-image-iff[OF inj] by simp
  with assms(1) show ?thesis
    by (auto simp: lset-conv-lnth linfinite-eq-llength full-SetCompr-eq)
qed

lemma isum-eq-inf: ‹isum f A = ∞ ⟷ infinite {a ∈ A. f a ≠ 0} ∨ (∃ a ∈ A. f
a = ∞)›
proof (rule iffI)
```

assume *h*: ‹*isum f A* = ∞›
then have *h2*: ‹∞ ∈ *sum f* ‘ {*B* |*B*. *B* ⊆ *A* ∧ *finite B*} ∨ *infinite* (*sum f* ‘ {*B* |*B*. *B* ⊆ *A* ∧ *finite B*})›
unfolding *isum-def Sup-enat-eq-inf* **by** *simp*
show ‹*infinite* {*a* ∈ *A*. *f a* ≠ *0*} ∨ (∃ *a* ∈ *A*. *f a* = ∞)›
proof (*rule ccontr*)
assume *hc*: ‹¬ (*infinite* {*a* ∈ *A*. *f a* ≠ *0*} ∨ (∃ *a* ∈ *A*. *f a* = ∞))›
then have *hfin*: ‹*finite* {*a* ∈ *A*. *f a* ≠ *0*}› **and** *hno-inf*: ‹∀ *a* ∈ *A*. *f a* ≠ ∞› **by** *auto*
have *surj*: ‹*sum f* ‘ {*B* |*B*. *B* ⊆ *A* ∧ *finite B*} ⊆ *sum f* ‘ *Pow* {*a* ∈ *A*. *f a* ≠ *0*}›
proof
fix *s* **assume** ‹*s* ∈ *sum f* ‘ {*B* |*B*. *B* ⊆ *A* ∧ *finite B*}›
then obtain *B* **where** *hB*: ‹*B* ⊆ *A*› ‹*finite B*› **and** *seq*: ‹*s* = *sum f B*› **by** *auto*
have *eq*: ‹*sum f B* = *sum f* {*a* ∈ *B*. *f a* ≠ *0*}›
by (*rule sum.mono-neutral-right*) (*use hB* **in** *auto*)
show ‹*s* ∈ *sum f* ‘ *Pow* {*a* ∈ *A*. *f a* ≠ *0*}›
using *hB eq seq* **by** (*auto simp*: *Pow-def*)
qed
have ‹*finite* (*sum f* ‘ {*B* |*B*. *B* ⊆ *A* ∧ *finite B*})›
by (*rule finite-subset*[*OF surj*]) (*rule finite-imageI*, *rule finite-Pow-iff*[*THEN iffD2*, *OF hfin*])
moreover have ‹∞ ∉ *sum f* ‘ {*B* |*B*. *B* ⊆ *A* ∧ *finite B*}›
proof
assume ‹∞ ∈ *sum f* ‘ {*B* |*B*. *B* ⊆ *A* ∧ *finite B*}›
then obtain *B* **where** *hB*: ‹*B* ⊆ *A*› ‹*finite B*› **and** *seq*: ‹*sum f B* = ∞› **by** *auto*
then obtain *a* **where** *ha*: ‹*a* ∈ *B*› **and** *hfa*: ‹*f a* = ∞›
using *inf-eq-sum*[*of B f*] **by** *auto*
have ‹*a* ∈ *A*› **using** *ha hB*(*1*) **by** *auto*
with *hfa hno-inf* **show** *False* **by** *auto*
qed
ultimately show *False* **using** *h2* **by** *simp*
qed
next
assume *h*: ‹*infinite* {*a* ∈ *A*. *f a* ≠ *0*} ∨ (∃ *a* ∈ *A*. *f a* = ∞)›
show ‹*isum f A* = ∞›
proof (*cases* ‹∃ *a* ∈ *A*. *f a* = ∞›)
case *True*
then obtain *a* **where** *ha*: ‹*a* ∈ *A*› **and** *hfa*: ‹*f a* = ∞› **by** *auto*
have ‹∞ ∈ *sum f* ‘ {*B* |*B*. *B* ⊆ *A* ∧ *finite B*}›
by (*rule image-eqI*[**where** *x*=‹{*a*}›]) (*simp-all add*: *ha hfa*)
then show *?thesis* **unfolding** *isum-def Sup-enat-eq-inf* **by** *simp*
next
case *False*
with *h* **have** *hinf*: ‹*infinite* {*a* ∈ *A*. *f a* ≠ *0*}› **and** *hno-inf*: ‹∞ ∉ *f* ‘ *A*› **by** *auto*
show *?thesis*

proof −
let *?Q* = ‹{*a* ∈ *A*. *f a* ≠ *0*}›
let *?𝔄* = ‹*lset* (*mk-chain ?Q* {})›
have *h-chain-strict*: ‹*chain* (⊂) (*mk-chain ?Q* {})›
by (*rule chain-mk-chain*) (*use hinf* **in** *simp-all*)
have *h-chain*: ‹*Complete-Partial-Order.chain* (⊆) *?𝔄*›
by (*rule chain-cpo-chain*[*of* ‹(⊂)›, *unfolded reflclp-subset*])
(*simp*, *rule h-chain-strict*)
have *h-inf-𝔄*: ‹*infinite ?𝔄*›
by (*rule infinite-lset-chain*[**where** *R*=‹(⊂)›])
(*rule linfinite-mk-chain*, *rule h-chain-strict*, *simp-all*)
have *h-sub-𝔄*: ‹∀ *C* ∈ *?𝔄*. *C* ⊆ *?Q*›
by (*auto dest*!: *in-lset-mk-chainD*[*OF* - *hinf*])
have *h-mem-𝔄*: ‹∀ *C* ∈ *?𝔄*. *C* ∈ {*B* |*B*. *B* ⊆ *A* ∧ *finite B*}›
by (*auto dest*!: *in-lset-mk-chainD*[*OF* - *hinf*])
have *h-inj*: ‹*inj-on* (*sum f*) *?𝔄*›
proof (*rule inj-onI*)
fix *B C* **assume** *hB*: ‹*B* ∈ *?𝔄*› **and** *hC*: ‹*C* ∈ *?𝔄*› **and** *heq*: ‹*sum f B* = *sum f C*›
from *h-chain* **have** *chain-rel*: ‹∀ *B* ∈ *?𝔄*. ∀ *C* ∈ *?𝔄*. *B* ⊆ *C* ∨ *C* ⊆ *B*›
unfolding *Complete-Partial-Order.chain-def* **by** *auto*
from *chain-rel*[*rule-format*, *OF hB hC*] **have** ‹*B* ⊆ *C* ∨ *C* ⊆ *B*› **.**
moreover {
assume *hBC*: ‹*B* ⊂ *C*›
from *h-mem-𝔄*[*rule-format*, *OF hC*] **have** *hCA*: ‹*C* ⊆ *A*› **and** *hfinC*: ‹*finite C*› **by** *simp+*
from *h-mem-𝔄*[*rule-format*, *OF hB*] **have** *hfinB*: ‹*finite B*› **by** *simp+*
obtain *c* **where** *hcC*: ‹*c* ∈ *C* − *B*› **using** *hBC* **by** *auto*
have *hfcpos*: ‹*0* < *f c*›
using *hcC h-sub-𝔄*[*rule-format*, *OF hC*] **by** (*auto simp*: *zero-less-iff-neq-zero*)
have *hno-infC*: ‹∞ ∉ *f* ‘ *C*› **using** *hCA hno-inf* **by** *auto*
have ‹*sum f B* < *sum f C*›
using *sum-strict-mono2-enat*[**where** *B*=*C* **and** *A*=*B* **and** *b*=*c* **and** *f*=*f*,
OF hfinC - *hcC hfcpos hno-infC*]
using *hBC* **by** *auto*
with *heq* **have** *False* **by** *simp*
}
moreover {
assume *hCB*: ‹*C* ⊂ *B*›
from *h-mem-𝔄*[*rule-format*, *OF hB*] **have** *hBA*: ‹*B* ⊆ *A*› **and** *hfinB*: ‹*finite B*› **by** *simp+*
from *h-mem-𝔄*[*rule-format*, *OF hC*] **have** *hCA*: ‹*C* ⊆ *A*› **and** *hfinC*: ‹*finite C*› **by** *simp+*
obtain *c* **where** *hcB*: ‹*c* ∈ *B* − *C*› **using** *hCB* **by** *auto*
have *hfcpos*: ‹*0* < *f c*›
using *hcB h-sub-𝔄*[*rule-format*, *OF hB*] **by** (*auto simp*: *zero-less-iff-neq-zero*)
have *hno-infB*: ‹∞ ∉ *f* ‘ *B*› **using** *hBA hno-inf* **by** *auto*
have ‹*sum f C* < *sum f B*›
using *sum-strict-mono2-enat*[**where** *B*=*B* **and** *A*=*C* **and** *b*=*c* **and** *f*=*f*,

```
            OF hfinB - hcB hfcpos hno-infB]
        using hCB by auto
      with heq have False by simp
    }
    ultimately show ‹B = C› by (cases ‹B = C›) auto
  qed
  have h-inf-image: ‹infinite (sum f ‘ ?𝔄)›
    by (metis finite-imageD h-inj h-inf-𝔄)
  have ‹sum f ‘ ?𝔄 ⊆ sum f ‘ {B |B. B ⊆ A ∧ finite B}›
    using h-mem-𝔄 by auto
  from infinite-super[OF this h-inf-image]
  show ?thesis unfolding isum-def Sup-enat-eq-inf by simp
 qed
qed

lemma cminfinite-alt: ‹cminfinite M = ((∃x. cmcount M x = ∞) ∨ infinite {x.
cmcount M x ≠ 0})›
proof (transfer)
 fix M :: ‹'a llist›
 show ‹linfinite M = ((∃x. count-llist M x = ∞) ∨ infinite {x. count-llist M x ≠
0})›
 proof (cases ‹lfinite M›)
  case True
  then have ‹¬ linfinite M› by (simp add: linfinite-eq-not-lfinite)
  moreover have ‹∀x. count-llist M x ≠ ∞›
   using True
  proof (induct M rule: lfinite.induct)
   case lfinite-LNil
   then show ?case by simp
  next
   case (lfinite-LConsI a M')
   then show ?case
   by (auto simp: eSuc-eq-infinity-iff eSuc-enat dest!: spec[of - M'] split: if-splits)
  qed
  moreover have ‹finite {x. count-llist M x ≠ 0}›
   using True
  proof (induct M rule: lfinite.induct)
   case lfinite-LNil
   then show ?case by simp
  next
   case (lfinite-LConsI a M')
   then have ‹{x. count-llist (LCons M' a) x ≠ 0} ⊆ insert M' {x. count-llist
a x ≠ 0}›
    by auto
   with lfinite-LConsI(2) show ?case
    by (meson finite-insert rev-finite-subset)
  qed
  ultimately show ?thesis by simp
```

```
  next
    case False
    then have linf: ‹linfinite M› by (simp add: linfinite-eq-not-lfinite)
    have count-inf: ‹count-llist (lmap (λ-. undefined) M) undefined = ∞›
      using False by (simp add: count-llist-lmap-const llength-eq-infty-conv-lfinite)
    then have ‹(∃ x. count-llist M x = ∞) ∨ infinite {a ∈ UNIV. count-llist M a
≠ 0}›
      unfolding count-llist-lmap isum-eq-inf by fastforce
    with linf show ?thesis by simp
  qed
qed

lemma cmset-alt: ‹cmset M = {a. cmcount M a ≠ 0}›
proof (transfer)
  fix M :: ‹'a llist›
 show ‹(⋂(mx :: (unit + 'a) llist)∈Collect (eq-cmset (lmap Inr M)). ⋃ (Basic-BNFs.setr
‘ lset mx)) = {a. count-llist M a ≠ 0}›
  proof −
    { fix x
    assume ‹∀ xa. (∀ x. count-llist (lmap Inr M) x = count-llist xa x) ⟶ (∃ xa∈lset
xa. x ∈ Basic-BNFs.setr xa)›
      then have ‹x ∈ lset M› by force
    }
    moreover
    { fix x :: 'a and lxs :: ‹(unit + 'a) llist›
      assume ‹x ∈ lset M› and ‹∀ x. count-llist (lmap Inr M) x = count-llist lxs
x›
      then have ‹∃ z∈lset lxs. x ∈ Basic-BNFs.setr z›
        by (metis count-llist-zero-iff imageI lset-lmap setr.intros)
    }
    ultimately show ?thesis
      by (fastforce simp: count-llist-zero-iff eq-cmset-def)
  qed
qed

lift-definition cmconst :: ‹'a ⇒ 'a cmset› is ‹repeat› .

lemma cminfinite-cmconst[simp]: ‹cminfinite (cmconst a)›
  by transfer (auto simp: linfinite-eq-not-lfinite)

lemma cminfinite-cmimage: ‹cminfinite (cmimage f M) ⟷ cminfinite M›
  by transfer (auto simp: linfinite-eq-not-lfinite)


locale cmset-as-Quotient
begin

setup-lifting Quotient-cmset
```

declare *cmempty.transfer*[*transfer-rule*]
declare *cmadd.transfer*[*transfer-rule*]
declare *cmset.map-transfer*[*transfer-rule*]
declare *cmset.set-transfer*[*transfer-rule*]
declare *cminfinite.transfer*[*transfer-rule*]

end

locale *cmset-as-typedef*
begin

setup-lifting *type-definition-cmset*

lemma *cmempty-transfer*[*transfer-rule*]: ‹*pcr-cmset R* (λ-. *0*) *cmempty*›
  by (*auto simp*: *pcr-cmset-def cr-cmset-def rel-fun-def relcompp-apply*)

lemma *cmadd-transfer*[*transfer-rule*]: ‹*rel-fun* (*pcr-cmset R*) (*rel-fun* (*pcr-cmset R*) (*pcr-cmset R*)) (λ*M N x*. *M x* + *N x*) *cmadd*›
  by (*auto simp*: *pcr-cmset-def cr-cmset-def rel-fun-def relcompp-apply*)

lemma *cmimage-transfer*[*transfer-rule*]: ‹*rel-fun* (=) (*rel-fun* (*pcr-cmset* (=)) (*pcr-cmset* (=))) (λ*f M b*. *isum M* {*a*. *f a* = *b*}) *cmimage*›
  by (*auto simp*: *pcr-cmset-def cr-cmset-def rel-fun-def relcompp-apply intro*: *isum-cong*)

lemma *cmset-transfer*[*transfer-rule*]: ‹*rel-fun* (*pcr-cmset* (=)) (=) (λ*M*. {*a*. *M a* ≠ *0*}) *cmset*›
  by (*auto simp*: *pcr-cmset-def cr-cmset-def rel-fun-def cmset-alt*)

lemma *cminfinite-transfer*[*transfer-rule*]: ‹*rel-fun* (*pcr-cmset* (=)) (=) (λ*M*. (∃ *x*. *M x* = ∞) ∨ *infinite* {*x*. *M x* ≠ *0*}) *cminfinite*›
  by (*auto simp*: *pcr-cmset-def cr-cmset-def rel-fun-def cminfinite-alt*)

lift-definition *cmreplace* :: ‹*'a cmset* ⇒ *'a* ⇒ *'a* ⇒ *'a cmset*› **is**
  ‹λ*f a b*. *f*(*a* := *epred* (*f a*), *b* := *eSuc* (*f b*))›
  by (*erule countable-subset*[*rotated*, *OF countable-insert*]) *auto*

lemma *cminfinite-cmreplace*: ‹*cminfinite M* ⟹ *cminfinite* (*cmreplace M a b*)›
proof (*transfer fixing*: *a b*)
  fix *M* :: ‹*'a* ⇒ *enat*›
  assume *M*: ‹*countable* {*x*. *M x* ≠ *0*}› ‹(∃ *x*. *M x* = ∞) ∨ *infinite* {*x*. *M x* ≠ *0*}›
  show ‹(∃ *x*. (*M*(*a* := *epred* (*M a*), *b* := *eSuc* (*M b*))) *x* = ∞) ∨ *infinite* {*x*. (*M*(*a* := *epred* (*M a*), *b* := *eSuc* (*M b*))) *x* ≠ *0*}›
  proof (*cases* ‹∃ *x*. *M x* = ∞›)
    case *True*
    then obtain *x* **where** ‹*M x* = ∞› **..**
    show *?thesis*
      by (*metis* ‹*M x* = ∞› *eSuc-infinity epred-Infty fun-upd-other fun-upd-same*)
  next
    case *False*

with *M* **have** ‹*infinite* {*x*. *M x* ≠ *0*}› **by** *blast*
then have ‹*infinite* {*x*. (*M*(*a* := *epred* (*M a*), *b* := *eSuc* (*M b*))) *x* ≠ *0*}›
by (*rule contrapos-nn*, *elim finite-subset*[*rotated*, *OF finite-insert*[*THEN iffD2*, *of - a*]]) *auto*
then show *?thesis*
by *blast*
qed
qed
end

lifting-update *cmset.lifting*
lifting-forget *cmset.lifting*

end

11 Countably Infinite Multisets

theory *Countably-Infinite-Multiset*
imports *Countable-Multiset*
begin

typedef *'a cinfmset* = ‹{*M* :: *'a cmset*. *cminfinite M*}›
by (*auto intro*!: *exI*[*of - ‹cmconst -›*])

setup-lifting *type-definition-cinfmset*

lift-bnf (*no-warn-wits*) *'a cinfmset*
for *map*: *cinfmimage rel*: *cinfmrel*
by (*auto simp*: *cminfinite-cmimage*)

lift-definition *cinfmcount* :: ‹*'a cinfmset* ⇒ *'a* ⇒ *enat*› **is** *cmcount* **.**

context begin
interpretation *cmset-as-typedef* **.**
lift-definition *cinfmreplace* :: ‹*'a cinfmset* ⇒ *'a* ⇒ *'a* ⇒ *'a cinfmset*› **is** *cmreplace*
by (*rule cminfinite-cmreplace*)

lemma *set-infmset-alt*: ‹*set-cinfmset xs* = {*a*. *cinfmcount xs a* ≠ *0*}›
unfolding *set-cinfmset-def cinfmcount-def cmset-alt*
by *auto*

lemma *set-cinfset-cinfmreplace*:
‹*set-cinfmset* (*cinfmreplace M x y*) ⊆ *set-cinfmset M* ∪ {*y*}›
including *cmset.lifting*
proof *transfer*
fix *M* :: ‹*'a cmset*› **and** *x y*
assume ‹*cminfinite M*›
show ‹*cmset* (*cmreplace M x y*) ⊆ *cmset M* ∪ {*y*}›
by *transfer auto*

qed
end

lemma *inj-cinfmcount*: ‹*inj cinfmcount*›
unfolding *inj-on-def cinfmcount-def map-fun-def o-apply id-apply*
by (*simp add*: *Rep-cinfmset-inject inj-cmcount inj-eq*)

lemma *in-range-cinfmcount*:
‹*countable* {*x*. *f x* ≠ *0*} ⟹ (∃ *x*. *f x* = ∞) ∨ *infinite* {*x*. *f x* ≠ *0*} ⟹ *f* ∈ *range cinfmcount*›
by (*drule in-range-cmcount*) (*smt* (*verit*, *best*) *Collect-cong Rep-cinfmset-cases UNIV-I cinfmcount.rep-eq cminfinite-alt image-iff mem-Collect-eq*)

lemma *cinfmcount-cinfmimage*[*simp*]:
‹*cinfmcount* (*cinfmimage f M*) *b* = *isum* (*cinfmcount M*) {*a*. *f a* = *b*}›
by *transfer* (*rule cmcount-cmimage*)

lemma *countable-nonzero-cinfmcount*: ‹*countable* {*x*. *cinfmcount M x* ≠ *0*}›
by *transfer* (*rule countable-nonzero-cmcount*)

lemma *infinite-nonzero-cinfmcount*: ‹(∃ *x*. *cinfmcount M x* = ∞) ∨ *infinite* {*x*. *cinfmcount M x* ≠ *0*}›
by *transfer* (*simp add*: *cminfinite-alt*)

lemma *type-definition-cinfmset*: ‹*type-definition cinfmcount* (*inv cinfmcount*) {*f*. *countable* {*x*. *f x* ≠ *0*} ∧ ((∃ *x*. *f x* = ∞) ∨ *infinite* {*x*. *f x* ≠ *0*})}›
by *standard* (*auto simp*: *inj-cinfmcount in-range-cinfmcount inv-f-f f-inv-into-f countable-nonzero-cinfmcount infinite-nonzero-cinfmcount*)

lifting-update *cinfmset.lifting*
lifting-forget *cinfmset.lifting*

definition *get-cinfmset* :: ‹'*a cinfmset* ⇒ *nat* ⇒ '*a*› **where**
‹*get-cinfmset M i* = *lnth* (*rep-cmset* (*Rep-cinfmset M*)) *i*›

context includes *cinfmset.lifting* **begin**
interpretation *cmset-as-Quotient* **.**
lemma *get-cinfmset-inject*:
‹∀ *i*. *get-cinfmset M i* = *get-cinfmset N i* ⟹ *M* = *N*›
unfolding *get-cinfmset-def*
proof *transfer*
fix *M N* :: ‹'*a cmset*›
assume ‹*cminfinite M*› ‹*cminfinite N*›
‹∀ *i*. *lnth* (*rep-cmset M*) *i* = *lnth* (*rep-cmset N*) *i*›
then have ‹*rep-cmset M* = *rep-cmset N*›
by (*simp add*: *cminfinite.rep-eq linfinite-eq-llength lnth-equalityI*)
then show ‹*M* = *N*›
by (*metis UNIV-I cmcount.rep-eq inj-cmcount inj-on-def*)
qed

end

end