

Formalization of Countable Multisets

Mathias Schack Rabing Dmitriy Traytel

June 25, 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	2
1.1	Preliminaries	2
1.2	Infinite Summation	8
2	Miscellaneous (Mostly About Lazy Lists)	18
3	enat as a Codatatype	20
4	Counting in Lazy Lists	24
5	Lazy Lists of Natural Numbers	25
6	Permutations of Lazy Lists	27
7	Same-Count Equivalence Relation	34
8	Countable Multisets as a Quotient	38
9	Lazy List Interleaving	39
10	Countable Multisets as a Subtype	68
11	Countably Infinite Multisets	95

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*:

```

 $\bigwedge a\ b\ c. b \leq c \implies (a::enat) + b \leq c \iff a \leq c - b$ 
 $\bigwedge a\ b\ c. a \leq c \implies (a::enat) + b \leq c \iff b \leq c - a$ 
 $\bigwedge a\ b\ c. a \leq b \implies c \leq b \implies (a::enat) \leq b - c \iff c \leq b - a$ 
by (smt (verit) add.commute add-diff-cancel-enat add-left-mono idiff-infinity le-iff-add
nle-le plus-eq-inf-iff-enat)+
```

lemma *disjoint-finite-aux*:

```

 $\forall i \in I. \forall j \in I. i \neq j \implies A\ i \cap A\ j = \{\}$   $\implies B \subseteq \bigcup (A\ ' I) \implies \text{finite } B \implies$ 
 $\text{finite } \{i \in I. B \cap A\ i \neq \{\}\}$ 
```

proof –

assume *disj*: $\forall i \in I. \forall j \in I. i \neq j \implies A\ i \cap A\ j = \{\}$

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* $\lambda i. \text{SOME } b. b \in B \cap A\ i - B$])

show *inj-on* ($\lambda i. \text{SOME } b. b \in B \cap A\ i$) $\{i \in I. B \cap A\ i \neq \{\}\}$

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*: $(\text{SOME } b. b \in B \cap A\ x) = (\text{SOME } b. b \in B \cap A\ y)$

have *hx*: $(\text{SOME } b. b \in B \cap A\ x) \in B \cap A\ x$

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*: $(\text{SOME } b. b \in B \cap A\ y) \in B \cap A\ y$

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  $(\lambda i. SOME\ b. b \in B \cap A\ i) \text{ ' } \{i \in I. B \cap A\ i \neq \{\}\} \subseteq B$ 
proof (intro subsetI)
  fix  $d$  assume  $d \in (\lambda i. SOME\ b. b \in B \cap A\ i) \text{ ' } \{i \in I. B \cap A\ i \neq \{\}\}$ 
  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*: $B \subseteq \bigcup (A \text{ ' } I) \implies B = \bigcup ((\lambda i. (B \cap A\ i)) \text{ ' } \{i \in I. B \cap A\ i \neq \{\}\})$
 by *fastforce*

lemma *incl-UNION-aux2*: $B \subseteq \bigcup (A \text{ ' } I) \iff B = \bigcup ((\lambda i. (B \cap A\ i)) \text{ ' } I)$
 by *fastforce*

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

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

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

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

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

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

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

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

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

lemma *Sup-eq-0-iff*: $Sup (A :: enat set) = 0 \iff (\forall a \in A. a = 0)$
using *Sup-bot-conv(1)[of A] unfolding bot-enat-def by auto*

lemma *plus-Sup-commute*:
assumes $f1: \{f1 b1 \mid b1. \varphi1 b1\} \neq \{\}$ **and**
 $f2: \{f2 b2 \mid b2. \varphi2 b2\} \neq \{\}$
shows
 $Sup \{(f1 b1 :: enat) \mid b1. \varphi1 b1\} + Sup \{f2 b2 \mid b2. \varphi2 b2\} =$
 $Sup \{f1 b1 + f2 b2 \mid b1 b2. \varphi1 b1 \wedge \varphi2 b2\}$ (**is** *?L1 + ?L2 = ?R*)
proof –
have f :
 $\{f1 b1 + f2 b2 \mid b1 b2. \varphi1 b1 \wedge \varphi2 b2\} \neq \{\}$
using $f1 f2$ **by** *auto*

```

show ?thesis proof (rule order.antisym)
  show ?L1 + ?L2 ≤ ?R
  proof –
    obtain b1-wit where hb1-wit:  $\varphi 1$  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  $x \in \{f2\ b2 \mid b2. \varphi 2\ b2\}$ 
      then obtain b2 where hb:  $x = f2\ b2\ \varphi 2\ b2$  by auto
      have mem:  $f1\ b1\text{-wit} + f2\ b2 \in \{f1\ b1 + f2\ b2 \mid b1\ b2. \varphi 1\ b1 \wedge \varphi 2\ b2\}$ 
        using hb1-wit hb(2) by blast
      have  $f2\ b2 \leq f1\ b1\text{-wit} + f2\ b2$  by (simp add: le-add2)
      also have ... ≤ ?R using cSup-upper[OF mem bdd-above-top] by simp
      finally show  $x \leq ?R$  using hb(1) by simp
    qed
    have hL1-le-R:  $\bigwedge b1. \varphi 1\ b1 \implies f1\ b1 \leq ?R$ 
    proof –
      fix b1 assume h $\varphi 1$ :  $\varphi 1\ b1$ 
      obtain b2 where hb2:  $\varphi 2\ b2$  using f2 by auto
      have mem:  $f1\ b1 + f2\ b2 \in \{f1\ b1 + f2\ b2 \mid b1\ b2. \varphi 1\ b1 \wedge \varphi 2\ b2\}$ 
        using h $\varphi 1$  hb2 by blast
      have  $f1\ b1 \leq f1\ b1 + f2\ b2$  by (simp add: le-add1)
      also have ... ≤ ?R using cSup-upper[OF mem bdd-above-top] by simp
      finally show  $f1\ b1 \leq ?R$  .
    qed
    have hL1-le: ?L1 ≤ ?R – ?L2
    proof (intro cSup-le-iff[OF f1 bdd-above-top, THEN iffD2] ballI)
      fix x assume  $x \in \{f1\ b1 \mid b1. \varphi 1\ b1\}$ 
      then obtain b1 where hb:  $x = f1\ b1\ \varphi 1\ b1$  by auto
      have hfb1-le-R:  $f1\ b1 \leq ?R$  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  $y \in \{f2\ b2 \mid b2. \varphi 2\ b2\}$ 
        then obtain b2 where hy:  $y = f2\ b2\ \varphi 2\ b2$  by auto
        have mem:  $f1\ b1 + f2\ b2 \in \{f1\ b1 + f2\ b2 \mid b1\ b2. \varphi 1\ b1 \wedge \varphi 2\ b2\}$ 
          using hb(2) hy(2) by blast
        have hle:  $f1\ b1 + f2\ b2 \leq ?R$  using cSup-upper[OF mem bdd-above-top]
      by simp
      show  $y \leq ?R - f1\ b1$ 
        using enat-pm-iff(2)[OF hfb1-le-R] hle hy(1) by simp
      qed
      show  $x \leq ?R - ?L2$ 
        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.premis 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.premis 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 (λi. h (b i)) J
  using insert.hyps hx(1) by simp
  then show x ∈ {h b1 + sum (λi. h (b2 i)) J | b1 b2. φ j b1 ∧ (∀ i ∈ J. φ 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 ⇒ enat) ⇒ 'a set ⇒ enat **where**
isum f A ≡ Sup (sum f ` {B | B . B ⊆ A ∧ finite B})

lemma *isum-subset-mono*: A ⊆ B ⇒ *isum* f A ≤ *isum* f B
unfolding *isum-def image-def*
by(*auto intro: Sup-subset-mono*)

lemma *isum-eq-sum*:
finite A ⇒ *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 ∈ sum f ` {B | B . B ⊆ A ∧ finite B} **using** hA **by** *blast*
show ∧x. x ∈ sum f ` {B | B . B ⊆ A ∧ finite B} ⇒ x ≤ sum f A
using hA **by** (*auto intro: sum-mono2*)
qed
qed

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

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

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

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

lemma *isum-const-zero'*: $\forall x \in A. g x = 0 \implies \text{isum } g A = 0$
by (*simp add: isum-cong*)

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

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

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

lemma *isum-reindex-cong*: $\text{inj-on } l B \implies A = l \text{ ` } B \implies$
 $(\bigwedge x. x \in B \implies g (l x) = h x) \implies \text{isum } g A = \text{isum } h B$
by (*metis isum-cong comp-def isum-reindex*)

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

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

```

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

```

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

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

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

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

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

lemma in-le-isum: $a \in A \implies f a \leq \text{isum } f A$
 by (metis isum-mono' isum-singl singletonD subsetI)

lemma isum-eq-singl:
 assumes $fx: f a = x$ and $f: \forall a'. a' \neq a \implies f a' = 0$ and $a: a \in A$
 shows $\text{isum } f A = x$
 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 $fx: f\ a \leq x$ **and** $f: \forall a'. a' \neq a \longrightarrow f\ a' = 0$ **and** $a: a \in A$
shows $isum\ f\ A \leq x$
by (*metis a f fx isum-eq-singl*)

lemma *isum-insert[simp]*: $a \notin A \implies isum\ f\ (insert\ a\ A) = isum\ f\ A + f\ a$
proof –

assume $ha: a \notin A$

have $hA: sum\ f\ '\{B\ |\ B \subseteq A \wedge finite\ B\} \neq \{\}$ **by** *auto*

show *?thesis*

unfolding *isum-def*

proof (*subst plus-SupR[OF ha], rule Sup-cong*)

show $sum\ f\ '\{B\ |\ B \subseteq insert\ a\ A \wedge finite\ B\} \neq \{\} \vee$

$\{x + f\ a\ |\ x. x \in sum\ f\ '\{B\ |\ B \subseteq A \wedge finite\ B\}\} \neq \{\}$ **by** *auto*

next

show $\forall v \in sum\ f\ '\{B\ |\ B \subseteq insert\ a\ A \wedge finite\ B\}.$

$\exists w \in \{x + f\ a\ |\ x. x \in sum\ f\ '\{B\ |\ B \subseteq A \wedge finite\ B\}\}. v \leq w$

proof (*intro ballI*)

fix v **assume** $v \in sum\ f\ '\{B\ |\ B \subseteq insert\ a\ A \wedge finite\ B\}$

then obtain B **where** $hB: B \subseteq insert\ a\ A$ *finite* B **and** $hv: v = sum\ f\ B$ **by**

blast

show $\exists w \in \{x + f\ a\ |\ x. x \in sum\ f\ '\{B\ |\ B \subseteq A \wedge finite\ B\}\}. v \leq w$

proof (*intro bexI[of - sum f (B - {a}) + f a]*)

show $v \leq sum\ f\ (B - \{a\}) + f\ a$

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 $sum\ f\ (B - \{a\}) + f\ a \in \{x + f\ a\ |\ x. x \in sum\ f\ '\{B\ |\ B \subseteq A \wedge finite\ B\}\}$

using $hB\ ha$ **by** *blast*

qed

qed

next

show $\forall w \in \{x + f\ a\ |\ x. x \in sum\ f\ '\{B\ |\ B \subseteq A \wedge finite\ B\}\}.$

$\exists v \in sum\ f\ '\{B\ |\ B \subseteq insert\ a\ A \wedge finite\ B\}. w \leq v$

proof (*intro ballI*)

fix w **assume** $w \in \{x + f\ a\ |\ x. x \in sum\ f\ '\{B\ |\ B \subseteq A \wedge finite\ B\}\}$

then obtain B **where** $hB: B \subseteq A$ *finite* B **and** $hw: w = sum\ f\ B + f\ a$ **by**

blast
show $\exists v \in \text{sum } f \text{ ' } \{B \mid B. B \subseteq \text{insert } a \ A \wedge \text{finite } B\}. w \leq v$
proof (*intro* *bestI*[*of - sum f (insert a B)*])
show $w \leq \text{sum } f \text{ (insert } a \ B)$
unfolding *hw* **using** *ha hB* **by** *force*
show $\text{sum } f \text{ (insert } a \ B) \in \text{sum } f \text{ ' } \{B \mid B. B \subseteq \text{insert } a \ A \wedge \text{finite } B\}$
using *hB* **by** *blast*
qed
qed
qed
qed

lemma *isum-UNION*:

assumes *dsj*: $\forall i \in I. \forall j \in I. i \neq j \longrightarrow A \ i \cap A \ j = \{\}$
shows $\text{isum } g \ (\bigcup (A \text{ ' } I)) = \text{isum } (\lambda i. \text{isum } g \ (A \ i)) \ I$

proof –

have *step1*: $\bigwedge J. J \subseteq I \implies \text{finite } J \implies$
 $(\sum_{i \in J. \text{Sup } \{y. \exists B \subseteq A \ i. \text{finite } B \wedge y = \text{sum } g \ B\}}) \leq$
 $\text{Sup } \{y. \exists B \subseteq \bigcup (A \text{ ' } I). \text{finite } B \wedge y = \text{sum } g \ B\}$

proof –

fix *J* **assume** *J1*: $J \subseteq I \text{ finite } J$

have *hsets-ne*: $\forall i \in J. \{\text{sum } g \ B \mid B. B \subseteq A \ i \wedge \text{finite } B\} \neq \{\}$
by *auto*

have *step1a*:

$\text{sum } (\lambda i. \text{Sup } \{\text{sum } g \ B \mid B. B \subseteq A \ i \wedge \text{finite } B\}) \ J =$
 $\text{Sup } \{\text{sum } (\lambda i. \text{sum } g \ (B \ i)) \ J \mid B. \forall i \in J. B \ i \subseteq A \ i \wedge \text{finite } (B \ i)\}$
using *sum-Sup-commute*[*OF J1(2) hsets-ne*] **by** *simp*

have *step1b*:

$\text{Sup } \{\text{sum } (\lambda i. \text{sum } g \ (B \ i)) \ J \mid B. \forall i \in J. B \ i \subseteq A \ i \wedge \text{finite } (B \ i)\} =$
 $\text{Sup } \{\text{sum } g \ (\bigcup (B \text{ ' } J)) \mid B. \forall i \in J. B \ i \subseteq A \ i \wedge \text{finite } (B \ i)\}$

proof (*rule arg-cong*[*of - - Sup*], *intro set-eqI iffI*)

fix *x*

assume $x \in \{\text{sum } (\lambda i. \text{sum } g \ (B \ i)) \ J \mid B. \forall i \in J. B \ i \subseteq A \ i \wedge \text{finite } (B \ i)\}$

then obtain *B* **where** *hB*: $\forall i \in J. B \ i \subseteq A \ i \wedge \text{finite } (B \ i)$

and *hx*: $x = \text{sum } (\lambda i. \text{sum } g \ (B \ i)) \ J$ **by** *auto*

have *hdisj*: $\forall i \in J. \forall j \in J. i \neq j \longrightarrow B \ i \cap B \ j = \{\}$

proof (*intro ballI impI*)

fix *i j* **assume** $i \in J \ j \in J \ i \neq j$

have $A \ i \cap A \ j = \{\}$ **using** *dsj* $\langle i \in J \rangle \langle j \in J \rangle \langle i \neq j \rangle$ *J1(1)* **by** *auto*

moreover have $B \ i \subseteq A \ i \ B \ j \subseteq A \ j$ **using** *hB* $\langle i \in J \rangle \langle j \in J \rangle$ **by** *auto*

ultimately show $B \ i \cap B \ j = \{\}$ **by** *auto*

qed

have $x = \text{sum } g \ (\bigcup (B \text{ ' } J))$

unfolding *hx*

by (*rule sum.UNION-disjoint*[*symmetric*]) (*use J1(2) hB hdisj in auto*)

then show $x \in \{\text{sum } g \ (\bigcup (B \text{ ' } J)) \mid B. \forall i \in J. B \ i \subseteq A \ i \wedge \text{finite } (B \ i)\}$

```

    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  $b \leq \text{sum } g (\bigcup ((\lambda i. C \cap A i) ' J))$ 
  using hCsub unfolding incl-UNION-aux2 by (auto simp: hb)
show  $\text{sum } g (\bigcup ((\lambda i. C \cap A i) ' J)) \in$ 
   $\{\text{sum } g (\bigcup (B ' J)) \mid B. \forall i \in J. B i \subseteq A i \wedge \text{finite } (B i)\}$ 
  using J1 hCsub by blast
qed
qed
qed
have step1d:
   $\text{Sup } \{\text{sum } g B \mid B. B \subseteq \bigcup (A ' J) \wedge \text{finite } B\} \leq$ 
   $\text{Sup } \{\text{sum } g B \mid B. B \subseteq \bigcup (A ' I) \wedge \text{finite } B\}$ 
proof (rule cSup-subset-mono)
  show  $\{\text{sum } g B \mid B. B \subseteq \bigcup (A ' J) \wedge \text{finite } B\} \neq \{\}$  by auto
  show  $\{\text{sum } g B \mid B. B \subseteq \bigcup (A ' J) \wedge \text{finite } B\} \subseteq$ 
   $\{\text{sum } g B \mid B. B \subseteq \bigcup (A ' I) \wedge \text{finite } B\}$ 
  using J1 by (force intro: Union-mono)
qed simp
show  $(\sum_{i \in J. \text{Sup } \{y. \exists B \subseteq A i. \text{finite } B \wedge y = \text{sum } g B\}}) \leq$ 
   $\text{Sup } \{y. \exists B \subseteq \bigcup (A ' I). \text{finite } B \wedge y = \text{sum } g B\}$ 
  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  $\{B \mid B. B \subseteq \bigcup (A ' I) \wedge \text{finite } B\} \neq \{\}$  by auto
next
  fix B assume hB:  $B \in \{B \mid B. B \subseteq \bigcup (A ' I) \wedge \text{finite } B\}$ 
  then have hBsub:  $B \subseteq \bigcup (A ' I)$  and hBfin: finite B by blast+
  let ?J =  $\{i. i \in I \wedge B \cap A i \neq \{\}\}$ 
  show  $\exists b \in \{B \mid B. B \subseteq I \wedge \text{finite } B\}.$ 
   $\text{sum } g B \leq (\sum_{i \in b. \text{Sup } (\text{sum } g ' \{B \mid B. B \subseteq A i \wedge \text{finite } B\}))$ 
proof (intro bexI[of - ?J])
  have  $\text{sum } g B \leq \text{sum } (\lambda i. \text{Sup } \{y. \exists x \in \{B \mid B. B \subseteq A i \wedge \text{finite } B\}. y = \text{sum}$ 
g x\}) ?J
proof –
  have hJfin: finite ?J using dsj hBsub hBfin disjoint-finite-aux by blast
  have hJsub:  $?J \subseteq I$  by auto
  have hdecomp:  $B = \bigcup ((\lambda i. B \cap A i) ' ?J)$ 
  using incl-UNION-aux[OF hBsub] by simp
  have hdisj-J:  $\forall i \in ?J. \forall j \in ?J. i \neq j \longrightarrow (B \cap A i) \cap (B \cap A j) = \{\}$ 
  using dsj hJsub by auto
  have hfin-J:  $\forall i \in ?J. \text{finite } (B \cap A i)$ 
  using hBfin by auto
  have hsum-eq:  $\text{sum } g B = \text{sum } (\lambda i. \text{sum } g (B \cap A i)) ?J$ 
  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  $i \in ?J$ 
  then have  $B \cap A \ i \subseteq A \ i$  finite ( $B \cap A \ i$ ) using  $hBfin$  by auto
  with  $hBsub$  show  $\text{sum } g \ (B \cap A \ i) \leq \text{Sup } \{y. \exists x \in \{B \mid B \subseteq A \ i \wedge \text{finite } B\}. y = \text{sum } g \ x\}$ 
  by (intro cSup-upper) auto
qed
qed
then show  $\text{sum } g \ B \leq (\sum i \mid i \in I \wedge B \cap A \ i \neq \{\}) . \text{Sup } (\text{sum } g \ ' \{B \mid B \subseteq A \ i \wedge \text{finite } B\})$ 
by (rule order-trans) (auto intro!: Sup-mono sum-mono)
show  $?J \in \{B \mid B \subseteq I \wedge \text{finite } B\}$ 
using  $dsj \ hBsub \ hBfin \ \text{disjoint-finite-aux}$  by force
qed
next
fix  $J$  assume  $hB: J \in \{B \mid B \subseteq I \wedge \text{finite } B\}$ 
then have  $J1: J \subseteq I$  finite  $J$  by blast+
show  $(\sum i \in J. \text{Sup } (\text{sum } g \ ' \{B \mid B \subseteq A \ i \wedge \text{finite } B\})) \leq \text{Sup } (\text{sum } g \ ' \{B \mid B \subseteq \bigcup (A \ ' I) \wedge \text{finite } B\})$ 
using  $step1[OF \ J1]$  by (auto simp: image-def)
qed
qed

```

```

lemma isum-Un[simp]:
  assumes  $A1 \cap A2 = \{\}$ 
  shows  $\text{isum } f \ (A1 \cup A2) = \text{isum } f \ A1 + \text{isum } f \ A2$ 
proof–
  have [simp]:  $\text{isum } (\lambda i. \text{isum } f \ (\text{case } i \ \text{of } 0 \Rightarrow A1 \mid \text{Suc } - \Rightarrow A2)) \ \{0, \text{Suc } 0\} = \text{isum } f \ A1 + \text{isum } f \ A2$ 
  by (subst isum-two) auto
  show  $?thesis$  using assms isum-UNION[of  $\{0,1::\text{nat}\}$ ]  $\lambda i. \text{case } i \ \text{of } 0 \Rightarrow A1 \mid - \Rightarrow A2 \ f]$  by auto
qed

```

```

lemma isum-Sigma:
   $\text{isum } (\lambda(a,b). f \ a \ b) \ (\text{Sigma } A \ Bs) = \text{isum } (\lambda a. \text{isum } (f \ a) \ (Bs \ a)) \ A$ 
proof–
  define  $g$  where  $g \equiv \lambda(a,b). f \ a \ b$ 
  define  $Cs$  where  $Cs \equiv \lambda a. \{a\} \times Bs \ a$ 
  have  $1: \bigwedge a. \{a\} \times Bs \ a = \text{Pair } a \ ' (Bs \ a)$  by auto
  have  $0: \text{Sigma } A \ Bs = (\bigcup a \in A. Cs \ a)$  unfolding Cs-def by auto
  show  $?thesis$  unfolding  $0$ 
proof (subst isum-UNION)
  show  $\forall i \in A. \forall j \in A. i \neq j \longrightarrow Cs \ i \cap Cs \ j = \{\}$  unfolding Cs-def by auto
next
  show  $\text{isum } (\lambda i. \text{isum } (\lambda(x,y). f \ x \ y) \ (Cs \ i)) \ A = \text{isum } (\lambda a. \text{isum } (f \ a) \ (Bs \ a)) \ A$ 
  unfolding Cs-def

```

```

proof (rule isum-cong)
  show  $A = A$  ..
next
  fix  $a$  assume  $a \in A$ 
  show  $\text{isum } (\lambda(x, y). f x y) (\{a\} \times Bs a) = \text{isum } (f a) (Bs a)$ 
  proof -
    have  $\text{isum } (\lambda(x, y). f x y) (\{a\} \times Bs a) = \text{isum } (\lambda(x, y). f x y) (\text{Pair } a \text{ '}$ 
     $Bs a)$ 
      by (simp add: 1)
    also have  $\dots = \text{isum } (f a) (Bs a)$ 
      by (subst isum-reindex-cong') (auto simp: o-def)
    finally show ?thesis .
  qed
qed
qed
qed

```

```

lemma isum-Times:  $\text{isum } (\lambda(a, b). f a b) (A \times B) = \text{isum } (\lambda a. \text{isum } (f a) B) A$ 
using isum-Sigma .

```

```

lemma isum-swap:  $\text{isum } (\lambda a. \text{isum } (f a) B) A = \text{isum } (\lambda b. \text{isum } (\lambda a. f a b) A) B$ 
(is ?L = ?R)
proof -
  have  $0: A \times B = (\lambda(a, b). (b, a)) \text{ ' } (B \times A)$  by auto
  have  $?L = \text{isum } (\lambda(a, b). f a b) (A \times B)$  using isum-Times ..
  also have  $\dots = \text{isum } (\lambda(b, a). f a b) (B \times A)$  unfolding 0 by (subst isum-reindex-cong')
(auto simp: o-def intro!: arg-cong2[of - - - isum])
  also have  $\dots = ?R$  by (subst isum-Times) auto
  finally show ?thesis .
qed

```

```

lemma isum-plus:
  shows  $\text{isum } (\lambda a. f1 a + f2 a) A = \text{isum } f1 A + \text{isum } f2 A$ 
proof -
  let  $?S = \text{sum } f1 \text{ ' } \{B \mid B \subseteq A \wedge \text{finite } B\}$ 
  let  $?T = \text{sum } f2 \text{ ' } \{B \mid B \subseteq A \wedge \text{finite } B\}$ 
  have  $hS: ?S \neq \{\}$  by auto
  have  $hT: ?T \neq \{\}$  by auto
  have  $\text{isum } (\lambda a. f1 a + f2 a) A =$ 
     $\text{Sup } \{a1 + a2 \mid a1 a2. a1 \in ?S \wedge a2 \in ?T\}$ 
  proof (unfold isum-def, rule Sup-cong)
    show  $\text{sum } (\lambda a. f1 a + f2 a) \text{ ' } \{B \mid B \subseteq A \wedge \text{finite } B\} \neq \{\} \vee$ 
     $\{a1 + a2 \mid a1 a2. a1 \in ?S \wedge a2 \in ?T\} \neq \{\}$  by auto
    show  $\forall a \in \text{sum } (\lambda a. f1 a + f2 a) \text{ ' } \{B \mid B \subseteq A \wedge \text{finite } B\}.$ 
     $\exists b \in \{a1 + a2 \mid a1 a2. a1 \in ?S \wedge a2 \in ?T\}. a \leq b$ 
  proof (intro ballI)

```

```

    fix a assume a ∈ sum (λa. f1 a + f2 a) ‘ {B |B. B ⊆ A ∧ finite B}
    then obtain B where hB: B ⊆ A finite B and ha: a = sum (λa. 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 (λa. 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 (λa. f1 a + f2 a) ‘ {B |B. B ⊆ A ∧ finite B}. b ≤ a
      proof (intro bexI[of - sum (λa. 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 (λa. f1 a + f2 a) (B1 ∪ B2)
          using hb h1 h2 sum.distrib[of f1 f2 B1 ∪ B2] by (simp add: add-mono)
        show sum (λa. f1 a + f2 a) (B1 ∪ B2) ∈ sum (λa. f1 a + f2 a) ‘ {B |B.
B ⊆ A ∧ finite B}
          using hB1 hB2 by auto
      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*: $\langle \text{ldropWhile } P \text{ } lxs = LCons \ x \ lxs' \longleftrightarrow (\neg P \ x \wedge (\exists xs. lxs = \text{lappend } (\text{lList-of } xs) (LCons \ x \ lxs') \wedge (\forall x \in \text{set } xs. P \ x))) \rangle$

proof –

have *False*

if $\langle \text{ldropWhile } P \text{ } lxs = LCons \ x \ lxs' \rangle$ **and** $\langle P \ x \rangle$

using that by (*metis lhd-LCons lhd-ldropWhile llist.disc(2) lnull-ldropWhile*)

moreover have $\langle \exists xs. lxs = \text{lappend } (\text{lList-of } xs) (LCons \ x \ lxs') \wedge (\forall x \in \text{set } xs. P \ x) \rangle$

if $\langle \text{ldropWhile } P \text{ } lxs = LCons \ x \ lxs' \rangle$

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 $\langle x \in \text{lset } lxs \rangle$

shows $\langle lxs = \text{lappend } (\text{ltakeWhile } ((\neq) \ x) \ lxs) (LCons \ x \ (\text{ltl } (\text{ldropWhile } ((\neq) \ x) \ lxs))) \rangle$

proof (*subst lappend-ltakeWhile-ldropWhile[symmetric, of lxs $\langle (\neq) \ x \rangle$], rule arg-cong[where $f = \langle \text{lappend } \rightarrow \rangle$]*)

from *assms* **show** $\langle \text{ldropWhile } ((\neq) \ x) \ lxs = LCons \ x \ (\text{ltl } (\text{ldropWhile } ((\neq) \ x) \ lxs)) \rangle$

by (*cases $\langle \text{ldropWhile } ((\neq) \ x) \ lxs \rangle$*)

(*auto simp: ldropWhile-eq-LNil-iff lhd-ldropWhile[where $P = \langle (\neq) \ x \rangle$, simplified, symmetric] dest!: eq-LConsD*)

qed

lemma *lzip-lmap-same*: $\langle \text{lzip } (\text{lmap } f \ lxs) (\text{lmap } g \ lxs) = \text{lmap } (\lambda x. (f \ x, \ g \ x)) \ lxs \rangle$

by (*coinduction arbitrary: lxs auto*)

lemma *llength-not-nullable*: $\langle \neg \text{lnull } lxs \implies \text{llength } lxs = \text{eSuc } (\text{llength } (\text{ltl } lxs)) \rangle$

by (*auto simp: lnull-def neq-LNil-conv*)

primrec *ltaken* :: $\langle \text{nat} \Rightarrow 'a \ \text{lList} \Rightarrow 'a \ \text{list} \rangle$ **where**

$\langle \text{ltaken } 0 \ lxs = [] \rangle$

$| \langle \text{ltaken } (\text{Suc } i) \ lxs = (\text{case } lxs \ \text{of } LNil \Rightarrow [] \ | \ LCons \ x \ lxs \Rightarrow x \ \# \ \text{ltaken } i \ lxs) \rangle$

lemma *nth-ltaken*: $\langle m < n \implies n \leq \text{llength } lxs \implies \text{nth } (\text{ltaken } n \ lxs) \ m = \text{lnth } lxs \ m \rangle$

by (*induct n arbitrary: m lxs (auto simp: nth-Cons' gr0-conv-Suc simp flip: eSuc-enat split: llist.splits)*)

lemma *set-ltaken*: $\langle \text{set } (\text{ltaken } n \ lxs) \subseteq \text{lset } lxs \rangle$

by (*induct n arbitrary: lxs (force split: llist.splits)+*)

lemma *length-ltaken*: $\langle \text{length } (\text{ltaken } n \ lxs) = (\text{if } \text{enat } n \leq \text{llength } lxs \ \text{then } n \ \text{else } \text{the-enat } (\text{llength } lxs)) \rangle$

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*: $\langle n \leq \text{llength } lxs \implies \text{set } (\text{ltaken } n \ lxs) = \text{lnth } lxs \ \{0..<n\}\rangle$

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*:

$\langle \text{ltaken } n \ (\text{lappend } lxs \ lys) = (\text{case } \text{llength } lxs \ \text{of } \infty \implies \text{ltaken } n \ lxs \mid \text{enat } m \implies \text{ltaken } n \ lxs \ @ \ \text{ltaken } (n - m) \ lys)\rangle$

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]*: $\langle \text{ltaken } i \ \text{LNil} = []\rangle$

by (*cases i*) *auto*

3 enat as a Codatatype

codatatype *en* = *eZ* | *eS* (*ep*: *en*)

where $\langle \text{ep } eZ = eZ \rangle$

coinductive *is-einf* **where**

$\langle \text{is-einf } n \implies \text{is-einf } (eS \ n)\rangle$

inductive *is-fin* **where**

$\langle \text{is-fin } eZ \rangle$

| $\langle \text{is-fin } n \implies \text{is-fin } (eS \ n)\rangle$

lemma *not-is-fin-is-einf*: $\langle \neg \text{is-fin } n \implies \text{is-einf } n \rangle$

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*: $\langle \text{is-fin } n \implies \neg \text{is-einf } n \rangle$

by (*induct n pred: is-fin*) (*auto elim: is-einf.cases*)

primcorec *einf* **where**

$\langle \text{einf} = eS \ \text{einf} \rangle$

lemma *einf-eS-iff*: $\langle \text{einf} = eS \ x \longleftrightarrow x = \text{einf} \rangle$

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  $\implies$  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  $\implies$   $\exists m. n = \text{nat-to-en } m$ ⟩
  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  $\implies$  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  $\implies$  False⟩
  by (induct n; subst (asm) einf.code) auto

definition enat-to-en where
  ⟨enat-to-en n = (case n of enat n  $\Rightarrow$  nat-to-en n | -  $\Rightarrow$  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  $\in$  range enat-to-en⟩
proof (cases ⟨n = einf⟩)
  case True
  then show ?thesis by (intro image-eqI[of - -  $\infty$ ]) (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-inv-into-f inj-enat-to-en simp: range-enat-to-en)

setup-lifting *type-definition-enat*

lift-definition *corec-enat* :: $\langle ('a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow \text{enat}) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow \text{enat} \rangle$
is *corec-en* .

lemma *eZ-transfer*[*transfer-rule*]: $\langle \text{pcr-enat } eZ\ 0 \rangle$
by (auto simp: pcr-enat-def cr-enat-def enat-to-en-def enat-0[symmetric])

lemma *eS-transfer*[*transfer-rule*]: $\langle \text{rel-fun pcr-enat pcr-enat } eS\ eSuc \rangle$
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*: $\langle \text{ep } (\text{nat-to-en } n) = \text{nat-to-en } (n - \text{Suc } 0) \rangle$
by (induct n) (auto)

lemma *ep-transfer*[*transfer-rule*]: $\langle \text{rel-fun pcr-enat pcr-enat } ep\ epred \rangle$
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*]:
 $\langle \text{corec-enat zer stop end cnt } a =$
(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** $\langle \text{zero-en} = eZ \rangle$

primcorec *plus-en* **where**
 $\langle \text{plus-en } e1\ e2 = (\text{case } e1 \text{ of } eZ \Rightarrow e2 \mid eS\ e1' \Rightarrow eS\ (\text{plus-en } e1'\ e2)) \rangle$

friend-of-corec *plus* :: $\langle \text{en} \Rightarrow \text{en} \Rightarrow \text{en} \rangle$ **where**
 $\langle e1 + e2 = (\text{case } e1 \text{ of } eZ \Rightarrow (\text{case } e2 \text{ of } eZ \Rightarrow eZ \mid eS\ e2' \Rightarrow eS\ e2') \mid eS\ e1' \Rightarrow eS\ (e1' + e2)) \rangle$
by (subst plus-en.code; simp split: en.splits) *transfer-prover*

lemma *plus-einf-left*[*simp*]: $\langle \text{einf} + e = \text{einf} \rangle$
by (coinduction arbitrary: e, subst (5) einf.code) (auto split: en.splits)

lemma *eZ-left-neutral*[*simp*]: $\langle eZ + e = e \rangle$
by (simp add: plus-en.code)

lemma *eZ-right-neutral*[*simp*]: $\langle e + eZ = e \rangle$
by (coinduction arbitrary: e) (auto split: en.splits)

lemma *plus-eS-left*[*simp*]: $\langle eS\ e1 + e2 = eS\ (e1 + e2) \rangle$

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

instance

proof

fix $n\ m\ q :: en$
 show $\langle n + m + q = n + (m + q) \rangle$
 by (*coinduction arbitrary: n m q rule: en.coinduct-strong*)
 (*force split: en.splits intro: exI[of - eZ] exI[of - eS ->]*)
 show $\langle n + m = m + n \rangle$
 by (*coinduction arbitrary: n m rule: en.coinduct-strong*)
 (*force split: en.splits intro: exI[of - eS ->]*)
 show $\langle 0 + n = n \rangle$
 by (*coinduction arbitrary: n*) (*auto simp: zero-en-def split: en.splits*)

qed

end

lemma *plus-einf-right*[*simp*]: $\langle e + einf = einf \rangle$
 by (*simp add: add.commute*)

lemma *nat-to-en-plus*[*simp*]: $\langle nat-to-en\ (m + n) = nat-to-en\ m + nat-to-en\ n \rangle$
 by (*induct m*) *auto*

lemma *ezero-transfer*[*transfer-rule*]: $\langle pcr-enat\ 0\ 0 \rangle$
 by (*auto simp: zero-en-def eZ-transfer*)

lemma *eplus-transfer*[*transfer-rule*]: $\langle rel-fun\ pcr-enat\ (rel-fun\ pcr-enat\ pcr-enat)$
 $(+)\ (+)\ \rangle$
 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*]: $\langle rel-fun\ R\ (rel-fun\ (rel-fun\ pcr-enat\ R)\ (rel-fun\ pcr-enat\ R))\ case-en\ co.case-enat \rangle$
 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**

$\langle lsum-en\ lxs = (case\ ldropWhile\ ((=)\ 0)\ lxs\ of\ LNil \Rightarrow 0 \mid LCons\ e\ lxs \Rightarrow eS\ (ep\ e + lsum-en\ lxs)) \rangle$

lift-definition *lsum* :: $\langle enat\ llist \Rightarrow enat \rangle$
 is *lsum-en* .

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

lemma *lsum-code-alt*:

$\langle lsum\ lxs = (case\ ldropWhile\ ((=)\ 0)\ lxs\ of\ LNil\ \Rightarrow\ 0\ |\ LCons\ e\ lxs\ \Rightarrow\ e + lsum\ lxs) \rangle$
by (*subst lsum-code, cases* $\langle lhd\ (ldropWhile\ ((=)\ 0)\ lxs) \rangle$ *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*

$\langle count-llist-en\ lxs\ x = (if\ x \in lset\ lxs\ then\ eS\ (count-llist-en\ (ltl\ (ldropWhile\ ((\neq)\ x)\ lxs))\ x)\ else\ eZ) \rangle$

lift-definition *count-llist* :: $\langle 'a\ llist \Rightarrow 'a \Rightarrow enat \rangle$
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* :: $\langle (enat \Rightarrow enat \Rightarrow bool) \Rightarrow enat \Rightarrow enat \Rightarrow bool \rangle$ **is**
 $\langle en.congclp \rangle$.

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*: $\langle x \in lset\ lxs \longleftrightarrow count-llist\ lxs\ x \neq 0 \rangle$
by (*subst count-llist-code*) *auto*

lemma *count-llist-zero-iff*: $\langle count-llist\ lxs\ x = 0 \longleftrightarrow x \notin lset\ lxs \rangle$
by (*metis in-lset-iff-count-llist*)

lemma *count-llist-alt*: $\langle count-llist\ lxs\ x = llength\ (lfilter\ ((=)\ x)\ lxs) \rangle$
by (*coinduction arbitrary: lxs*) (*auto simp: count-llist-zero-iff epred-llength ltl-lfilter*)

lemma *count-llist-lappend*:

$\langle count-llist\ (lappend\ lxs\ lys)\ x = count-llist\ lxs\ x + (if\ lfinite\ lxs\ then\ count-llist\ lys\ x\ else\ 0) \rangle$

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]: $\langle count-llist\ LNil\ x = 0 \rangle$
by (*subst count-llist-code*) *auto*

lemma *count-llist-LCons*[simp]: $\langle \text{count-llist } (LCons\ y\ lxs)\ x =$
 $(\text{if } x = y \text{ then } eSuc\ (\text{count-llist } lxs\ x) \text{ else } \text{count-llist } lxs\ x) \rangle$
by (*subst* (1 3) *count-llist-code*) (*auto simp: count-llist-zero-iff*)

5 Lazy Lists of Natural Numbers

primcorec *lupt* **where**

$\langle \text{lupt } i\ j = (\text{if } \text{enat } i \geq j \text{ then } LNil \text{ else } LCons\ i\ (\text{lupt } (Suc\ i)\ j)) \rangle$

lemma *lset-luptD*[*OF - refl*]:

assumes $\langle k \in \text{lset } lxs \rangle \langle lxs = \text{lupt } i\ j \rangle$

shows $\langle i \leq k \rangle \langle k < j \rangle$

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*: $\langle i \leq k \implies \text{enat } k < j \implies k \in \text{lset } (\text{lupt } i\ j) \rangle$

proof (*induct <k - i> arbitrary: i*)

case 0

then show *?case*

by (*subst lupt.code*) *auto*

next

case (*Suc x*)

then have $\langle \text{enat } i < j \rangle$

by (*meson enat-ord-simps(1) order.strict-trans1*)

from *Suc(2)* **have** $\langle x = k - Suc\ i \rangle$

by *force*

with $Suc(1)[OF\ this]\ Suc(2-4)\ \langle enat\ i < j \rangle$ **show** $?case$
by $(subst\ lupt.code)\ (auto\ simp:\ Suc-le-eq)$
qed

lemma $lset-lupt$: $\langle lset\ (lupt\ i\ j) = \{k.\ i \leq k \wedge enat\ k < j\} \rangle$
by $(auto\ intro:\ lset-luptI\ dest:\ lset-luptD)$

lemma $llength-lupt[simp]$: $\langle llength\ (lupt\ i\ j) = j - i \rangle$
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 $lnth-lupt$: $\langle k < j - enat\ i \implies lnth\ (lupt\ i\ j)\ k = i + k \rangle$
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** $\langle enat\ k < j - Suc\ i \rangle$
by $(cases\ j)\ auto$
with $Suc(1)[OF\ this]\ Suc(2)$ **show** $?case$
by $(subst\ lupt.code;\ cases\ j)\ auto$
qed

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

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

lemma $ldistinct-lupt[simp]$: $\langle ldistinct\ (lupt\ i\ j) \rangle$
by $(coinduction\ arbitrary:\ i)\ (auto\ simp:\ lset-lupt)$

lemma $lzip-lupt$: $\langle lzip\ (lupt\ i\ j)\ (lupt\ i\ k) = lmap\ (\lambda x.\ (x,\ x))\ (lupt\ i\ (\min\ j\ k)) \rangle$
by $(coinduction\ arbitrary:\ i)\ (auto\ simp:\ min-def)$

6 Permutations of Lazy Lists

definition $\langle \text{lpermute } \pi \text{ } lxs = \text{lmap } (\lambda i. \text{lnth } lxs (\pi \ i)) (\text{lupt } 0 (\text{llength } lxs)) \rangle$

abbreviation $\langle \text{lbij-on } \pi \text{ } lxs \equiv \text{bij-betw } \pi \ \{k. \text{enat } k < \text{llength } lxs\} \ \{k. \text{enat } k < \text{llength } lxs\} \rangle$

lemma *lset-lpermute*:

assumes $\langle \text{lbij-on } \pi \text{ } lxs \rangle$

shows $\langle \text{lset } (\text{lpermute } \pi \text{ } lxs) = \text{lset } lxs \rangle$

proof –

have $\langle \exists m. \text{lnth } lxs \ n = \text{lnth } (\text{lmap } (\lambda i. \text{lnth } lxs (\pi \ i)) (\text{lupt } 0 (\text{llength } lxs))) \ m \wedge \text{enat } m < \text{llength } lxs \rangle$

if $\langle \text{enat } n < \text{llength } lxs \rangle$

for $n :: \text{nat}$

using *that* *assms* *bij-betw-inv-into-right*[*OF* *assms*, *of* *n*]

by (*intro* *exI*[*of* - *inv-into* $\{k. \text{enat } k < \text{llength } lxs\} \ \pi \ n$])

(*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*]: $\langle \text{llength } (\text{lpermute } \pi \text{ } lxs) = \text{llength } lxs \rangle$

by (*auto simp: lpermute-def*)

lemma *lnth-lpermute*[*simp*]: $\langle \text{lbij-on } \pi \text{ } lxs \implies i < \text{llength } lxs \implies \text{lnth } (\text{lpermute } \pi \text{ } lxs) \ i = \text{lnth } lxs (\pi \ i) \rangle$

by (*simp add: lnth-lupt lpermute-def*)

lemma *lfilter-permute*: $\langle \text{ldistinct } lxs \implies \text{ldistinct } lys \implies \text{bij-betw } \pi \text{ } (\text{lset } lxs) (\text{lset } lys) \implies \forall x \in \text{lset } lxs. P \ x \longleftrightarrow Q \ (\pi \ x) \implies$

$\text{llength } (\text{lfilter } P \text{ } lxs) = \text{llength } (\text{lfilter } Q \text{ } lys) \rangle$

proof (*coinduction arbitrary: lxs lys π*)

case *eq-enat*

then have $\langle \text{llength } (\text{lfilter } P \text{ } lxs) = 0 \longleftrightarrow \text{llength } (\text{lfilter } Q \text{ } lys) = 0 \rangle$

by (*force simp: bij-betw-iff-bijections*)

moreover

{ assume $\langle \text{llength } (\text{lfilter } P \text{ } lxs) \neq 0 \rangle \langle \text{llength } (\text{lfilter } Q \text{ } lys) \neq 0 \rangle$

then have *lfin*[*unfolded comp-def, simp*]: $\langle \text{lfinite } (\text{ltakeWhile } (\text{Not } o \ P) \text{ } lxs) \rangle$
 $\langle \text{lfinite } (\text{ltakeWhile } (\text{Not } o \ Q) \text{ } lys) \rangle$

by (*simp-all add: lfinite-ltakeWhile*)

let $?lxs = \langle \text{lappend } (\text{ltakeWhile } (\text{Not } o \ P) \text{ } lxs) (\text{tl } (\text{ldropWhile } (\text{Not } o \ P) \text{ } lxs)) \rangle$

let $?lys = \langle \text{lappend } (\text{ltakeWhile } (\text{Not } o \ Q) \text{ } lys) (\text{tl } (\text{ldropWhile } (\text{Not } o \ Q) \text{ } lys)) \rangle$

define x **where** $\langle x = \text{lhd } (\text{ldropWhile } (\text{Not } o \ P) \text{ } lxs) \rangle$

with $\ast(1)$ **have** $\langle P \ x \rangle$ **using** *lhd-ldropWhile* **by** *force*

from $\ast(1)$ **have** $\langle x \in \text{lset } lxs \rangle$ **unfolding** *x-def*

by (*metis in-lset-ldropWhileD lfilter-eq-LCons lhd-lfilter llength-eq-0 llist.exhaust-sel*)

```

lset-intros(1) lset-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 lset-exhaust-sel
lset-intros(1) lset-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 o P) lxs) (LCons x (ltl (ldropWhile (Not
o P) lxs)))⟩
    ⟨lys = lappend (ltakeWhile (Not o Q) lys) (LCons y (ltl (ldropWhile (Not o
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-lset-lpermute:*

```

⟨bij-on π lxs ⟹ count-lset (lpermute π lxs) x = count-lset lxs x⟩
  unfolding count-lset-alt
  by (subst (2) lset-conv-lmap-lupt[of lxs]) (auto simp: lpermute-def lfilter-lmap
lset-lupt)

```

$\langle \text{lfilter-permute}[\text{of } \langle \text{lupt } 0 \text{ (llength } lxs) \rangle \langle \text{lupt } 0 \text{ (llength } lxs) \rangle \pi \langle \lambda i. x = \text{lnth } lxs (\pi i) \rangle \langle \lambda i. x = \text{lnth } lxs i \rangle] \rangle$

lemma *lpermute-lzip*: $\langle \text{lbij-on } \pi \text{ } lxs \implies \text{llength } lys = \text{llength } lxs \implies \text{lpermute } \pi \text{ (lzip } lxs \text{ } lys) = \text{lzip (lpermute } \pi \text{ } lxs) \text{ (lpermute } \pi \text{ } lys) \rangle$

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*:

$\langle \text{count-llist } lxs \text{ } x > n \implies \exists i < \text{llength } lxs. \text{lnth } lxs \text{ } i = x \wedge \text{count-list (ltaken } i \text{ } lxs) \text{ } x = n \rangle$

proof (*induct n arbitrary: lxs*)

case 0

then have *ex*: $\langle \exists i < \text{llength } lxs. \text{lnth } lxs \text{ } i = x \rangle$

by (*metis in-lset-conv-lnth in-lset-iff-count-llist not-gr-zero zero-enat-def*)

define *i* **where** $\langle i = (\text{LEAST } i. \text{enat } i < \text{llength } lxs \wedge \text{lnth } lxs \text{ } i = x) \rangle$

have *i*: $\langle \text{enat } i < \text{llength } lxs \rangle \langle \text{lnth } lxs \text{ } i = x \rangle$

$\langle \bigwedge j. \text{enat } j < \text{llength } lxs \implies \text{lnth } lxs \text{ } j = x \implies i \leq j \rangle$

unfolding *i-def* **using** *LeastI2-wellorder-ex*[*OF ex*]

by (*force simp: Least-le*)⁺

moreover have $\langle \text{lnth } lxs \text{ } i = \text{lnth } lxs \text{ } j \implies j < i \implies \text{False} \rangle$ **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* = $\langle \text{ltl (ldropWhile ((\neq) x) lxs) \rangle$

from *Suc(2)* **have** $\langle \text{enat } n < \text{count-llist } ?lxs \text{ } x \rangle$

by (*metis Suc-ile-eq count-llist-code iless-Suc-eq not-less-zero*)

with *Suc(1)* **obtain** *i* **where** *i*: $\langle \text{enat } i < \text{llength } ?lxs \rangle$

$\langle \text{lnth } ?lxs \text{ } i = x \rangle \langle \text{count-list (ltaken } i \text{ } ?lxs) \text{ } x = n \rangle$

by *blast*

from *Suc(2)* **obtain** *k* **where** *k*: $\langle \text{llength (ltakeWhile ((\neq) x) lxs) = enat } k \rangle$

by (*metis (full-types) enat-the-enat in-lset-iff-count-llist llength-ltakeWhile-eq-infinity lset-ltakeWhileD not-less-zero*)

then have *klen*: $\langle \text{enat } k < \text{llength } lxs \rangle$

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* = $\langle k + \text{Suc } i \rangle$

have *lxs*: $\langle lxs = \text{lappend (ltakeWhile ((\neq) x) lxs) (LCons x ?lxs) \rangle$

by (*metis Suc(2) in-lset-iff-count-llist ltakeWhile-ldropWhile-decomp not-less-zero*)

have $\langle \text{enat } ?i < \text{llength } lxs \rangle$

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** $\langle \text{lnth } lxs \text{ } ?i = x \rangle$

by (*subst lxs*) (*auto simp: lnth-lappend*)

moreover have *x*: $\langle x \notin \text{lset (ltakeWhile ((\neq) x) lxs) \rangle$

by (*metis (full-types) lset-ltakeWhileD*)

with k $l\text{length-}l\text{takeWhile-le}$ [of $\langle (\neq) x \rangle lxs$] **have** $\langle x \notin \text{set } (l\text{taken } k \text{ } lxs) \rangle$
by $(\text{subst set-}l\text{taken-conv})$ $(\text{auto simp: in-lset-conv-lnth } l\text{takeWhile-nth})$
with k $klen$ $i(\beta)$ x **have** $\langle \text{count-list } (l\text{taken } ?i \text{ } lxs) \ x = \text{Suc } n \rangle$
using $[[\text{simproc del: list-to-set-comprehension}]]$
by $(\text{subst } lxs)$ $(\text{auto simp: } l\text{taken-lappend dest!: set-mp}[OF \text{set-}l\text{taken}])$ **intro!**:
 $\text{count-notin split: llist.splits}$
ultimately show $?case$ **by** blast
qed

function $l\text{find-index}$ **where**

$\langle l\text{find-index } n \ x \ i \ lxs = (\text{if } \text{count-llist } lxs \ x \leq \text{enat } n \ \text{then } i \ \text{else}$
 $\ \ \ \ \ \text{if } lhd \ lxs = x \ \text{then } \text{case } n \ \text{of } 0 \Rightarrow i \mid \text{Suc } m \Rightarrow l\text{find-index } m \ x \ (\text{Suc } i) \ (l\text{tl } lxs)$
 $\ \ \ \ \ \text{else } l\text{find-index } n \ x \ (\text{Suc } i) \ (l\text{tl } lxs)) \rangle$

by auto

termination

proof $(\text{relation } \langle \text{measure } (\lambda(n, x, -, lxs). \text{LEAST } i. \text{enat } i < l\text{length } lxs \wedge l\text{nth } lxs$
 $i = x \wedge \text{count-list } (l\text{taken } i \text{ } lxs) \ x = n) \rangle)$

fix $n \ x \ i$ **and** $lxs :: \langle 'a \ \text{llist} \rangle$ **and** m

assume $hlt: \langle \neg \text{count-llist } lxs \ x \leq \text{enat } n \rangle$

assume $hlhd: \langle lhd \ lxs = x \rangle$

assume $hn: \langle n = \text{Suc } m \rangle$

let $?PP = \langle \lambda x \ lxs \ k \ n. \text{enat } k < l\text{length } lxs \wedge l\text{nth } lxs \ k = x \wedge \text{count-list } (l\text{taken}$
 $k \ lxs) \ x = n \rangle$

let $?P = \langle ?PP \ x \rangle$

from $hlt \ hn$ **have** $hcount: \langle \text{enat } (\text{Suc } m) < \text{count-llist } lxs \ x \rangle$

by $(\text{simp add: not-le})$

obtain j **where** $hj1: \langle \text{enat } j < l\text{length } lxs \rangle$ **and** $hj2: \langle l\text{nth } lxs \ j = x \rangle$

and $hj3: \langle \text{enat } (\text{count-list } (l\text{taken } j \text{ } lxs) \ x) = \text{enat } (\text{Suc } m) \rangle$

using $\text{exist-nth-occurrence-in-llist}[OF \ hcount]$ **by** blast

have $hj3': \langle \text{count-list } (l\text{taken } j \text{ } lxs) \ x = \text{Suc } m \rangle$

using $hj3$ **by** simp

have $\text{rhs-eq}: \langle (\text{LEAST } k. ?P \ lxs \ k \ (\text{Suc } m)) = \text{Suc } (\text{LEAST } k. ?P \ lxs \ (\text{Suc } k)$
 $(\text{Suc } m)) \rangle$

proof (rule Least-Suc)

show $\langle ?P \ lxs \ j \ (\text{Suc } m) \rangle$

using $hj1 \ hj2 \ hj3'$ **by** simp

show $\langle \neg ?P \ lxs \ 0 \ (\text{Suc } m) \rangle$

by simp

qed

have $\text{lhs-eq}: \langle (\text{LEAST } k. ?P \ (l\text{tl } lxs) \ k \ m) = (\text{LEAST } k. ?P \ lxs \ (\text{Suc } k) \ (\text{Suc } m)) \rangle$

using $hlhd$ **by** $(\text{cases } lxs)$ $(\text{auto simp: Suc-ile-eq lnth-LCons' fun-eq-iff gr0-conv-Suc})$

have $\text{key2}: \langle (\text{LEAST } k. ?P \ lxs \ k \ (\text{Suc } m)) = \text{Suc } (\text{LEAST } k. ?P \ (l\text{tl } lxs) \ k \ m) \rangle$

using rhs-eq lhs-eq **by** simp

show $\langle ((m, x, \text{Suc } i, l\text{tl } lxs), n, x, i, lxs) \in \text{measure } (\lambda(n, x, -, lxs). \text{LEAST } i.$
 $?PP \ x \ lxs \ i \ n) \rangle$

by $(\text{simp add: hn key2})$

next

fix $n \ x \ i$ **and** $lxs :: \langle 'a \ \text{llist} \rangle$

assume $hlt: \langle \neg \text{count-llist } lxs \ x \leq \text{enat } n \rangle$

```

assume hlhd:  $\langle \text{lhd } lxs \neq x \rangle$ 
let ?PP =  $\langle \lambda x lxs k n. \text{enat } k < \text{llength } lxs \wedge \text{lnth } lxs k = x \wedge \text{count-list } (\text{ltaken } k lxs) x = n \rangle$ 
let ?P =  $\langle ?PP x \rangle$ 
from hlt have hcount:  $\langle \text{enat } n < \text{count-llist } lxs x \rangle$ 
  by (simp add: not-le)
obtain j where hj1:  $\langle \text{enat } j < \text{llength } lxs \rangle$  and hj2:  $\langle \text{lnth } lxs j = x \rangle$ 
  and hj3:  $\langle \text{enat } (\text{count-list } (\text{ltaken } j lxs) x) = \text{enat } n \rangle$ 
  using exist-nth-occurrence-in-llist[OF hcount] by blast
have hj3':  $\langle \text{count-list } (\text{ltaken } j lxs) x = n \rangle$ 
  using hj3 by simp
have rhs-eq:  $\langle (\text{LEAST } k. ?P lxs k n) = \text{Suc } (\text{LEAST } k. ?P lxs (\text{Suc } k) n) \rangle$ 
proof (rule Least-Suc)
  show  $\langle ?P lxs j n \rangle$ 
  using hj1 hj2 hj3' by simp
  show  $\langle \neg ?P lxs 0 n \rangle$ 
  using hlhd by (cases lxs) (auto simp: lhd-conv-lnth enat-0)
qed
have lhs-eq:  $\langle (\text{LEAST } k. ?P (\text{tl } lxs) k n) = (\text{LEAST } k. ?P lxs (\text{Suc } k) n) \rangle$ 
  using hlhd by (cases lxs) (auto simp: Suc-ile-eq lnth-LCons' fun-eq-iff gr0-conv-Suc)
  show  $\langle ((n, x, \text{Suc } i, \text{tl } lxs), n, x, i, lxs) \in \text{measure } (\lambda(n, x, -, lxs). \text{LEAST } i. ?PP x lxs i n) \rangle$ 
  by (simp add: lhs-eq rhs-eq less-Suc-eq-le)
qed simp
declare lfind-index.simps[simp del]

lemma lfind-index-ge[simp]:  $\langle \text{lfind-index } n x j lxs \geq j \rangle$ 
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]:  $\langle \text{count-llist } lxs x > \text{enat } n \implies \text{lfind-index } n x j lxs < \text{llength } lxs + j \rangle$ 
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:
   $\langle \text{count-llist } lxs x > \text{enat } n \implies \text{lnth } lxs (\text{lfind-index } n x j lxs - j) = x \rangle$ 
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
        qed
      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*: $\langle \text{llength } lxs = \text{llength } lys \implies (\bigwedge i. \text{enat } i < \text{llength } lxs \implies \text{lnth } lxs \ i = \text{lnth } lys \ i) \implies lxs = lys \rangle$
by (*metis (full-types) llist.rel-eq llist-all2-all-lnthI*)

lemma *lfind-index-inject*:
 $\langle \text{count-llist } lxs \ x > \text{enat } n \implies \text{count-llist } lxs \ y > \text{enat } m \implies \text{lfind-index } n \ x \ j \ lxs = \text{lfind-index } m \ y \ j \ lxs \implies n = m \wedge x = y \rangle$
by (*metis lfind-index-count-list lfind-index-lnth*)

lemma *count-list-ltaken-less*:
 $\langle i < \text{llength } lxs \implies \text{count-list } (\text{ltaken } i \ lxs) \ (\text{lnth } lxs \ i) < \text{count-llist } lxs \ (\text{lnth } lxs \ i) \rangle$
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*:
 $\langle \text{enat } i < \text{llength } lxs \implies \text{enat } j < \text{llength } lxs \implies \text{count-list } (\text{ltaken } i \ lxs) \ (\text{lnth } lxs \ j) = \text{count-list } (\text{ltaken } j \ lxs) \ (\text{lnth } lxs \ j) \implies \text{lnth } lxs \ i = \text{lnth } lxs \ j \implies i = j \rangle$
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 $\langle \text{eq-cmset } lxs \ lys = (\forall x. \text{count-llist } lxs \ x = \text{count-llist } lys \ x) \rangle$

lemma *lset-eq-cmset*: $\langle \text{eq-cmset } lxs \ lys \implies \text{lset } lxs = \text{lset } lys \rangle$
by (*auto simp: eq-cmset-def in-lset-iff-count-llist*)

lemma *eq-cmset-llength*: $\langle \text{eq-cmset } lxs \ lys \implies \text{llength } lxs = \text{llength } lys \rangle$
unfolding *eq-cmset-def*
proof (*coinduction arbitrary: lxs lys*)
case *eq-enat*
then have *heq*: $\langle \forall x. \text{count-llist } lxs \ x = \text{count-llist } lys \ x \rangle$ **by** *blast*
have *lnull-iff*: $\langle \text{lnull } lxs \longleftrightarrow \text{lnull } lys \rangle$
using *heq* **by** (*auto simp: lnnull-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 -] 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 xs}⟩
  show ⟨∃ i ∈ {k. enat k < llength xs}. 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 xs = xs'⟩
  proof (intro lnth-equalityI)
    show ⟨llength (lpermute f xs) = llength xs'⟩
      by (simp add: lpermute-def hlen)
  next
    fix i
    assume hi: ⟨enat i < llength (lpermute f xs)⟩
    then have hi': ⟨enat i < llength xs⟩ by (simp add: lpermute-def lnth-lupt)
    have ⟨lnth (lpermute f xs) i = lnth xs (f i)⟩
      by (simp add: lpermute-def lnth-lupt hi' f-def Let-def)
    also have ⟨... = lnth xs' i⟩
    proof -
      have hlt: ⟨enat (count-list (ltaken i xs') (lnth xs' i)) < count-llist xs (lnth
xs' 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 xs) i = lnth xs' i⟩ .
  qed
qed
qed

```

lemma *eq-cmset-lzip-left*:

```

  assumes ⟨eq-cmset xs' xs⟩ ⟨llength lys = llength xs⟩
  shows ⟨∃ lys'. eq-cmset (lzip xs' lys') (lzip xs lys) ∧ llength lys' = llength xs'⟩
  proof -
    from assms(1) obtain π where ⟨lbij-on π xs⟩ ⟨lpermute π xs = xs'⟩ 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 xs lys⟩
  shows ⟨eq-cmset (lmap f xs) (lmap f lys)⟩
  proof -

```

from *assms* **obtain** π **where** $\langle \text{bij-on } \pi \text{ lys} \rangle \langle \text{lpermute } \pi \text{ lys} = \text{lys} \rangle$ **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 = π] lnth-lmap)
qed

lemma *lalist-all2-reorder-left-invariance*:

assumes $\text{rel} : \langle \text{lalist-all2 } R \text{ lys} \text{ lys}' \rangle$ **and** $\text{ms-x} : \langle \text{eq-cmset } \text{lys}' \text{ lys} \rangle$
shows $\langle \exists \text{lys}' . \text{lalist-all2 } R \text{ lys}' \text{ lys}' \wedge \text{eq-cmset } \text{lys}' \text{ lys} \rangle$

proof –

from ms-x rel [*THEN lalist-all2-llengthD*] **obtain** lys' **where**
 $\text{ms-xy} : \langle \text{eq-cmset } (\text{lzip } \text{lys}' \text{ lys}') (\text{lzip } \text{lys} \text{ lys}) \rangle$ **and** $\text{len} : \langle \text{llength } \text{lys}' = \text{llength } \text{lys}' \rangle$
by (*metis eq-cmset-lzip-left*)
with rel **have** $\text{rel}' : \langle \text{lalist-all2 } R \text{ lys}' \text{ lys}' \rangle$ **unfolding** *lalist-all2-conv-lzip*
by (*auto simp: lset-eq-cmset*)
moreover from *eq-cmset-lmap[OF ms-xy, of snd]*
have $\langle \text{eq-cmset } (\text{lmap } \text{snd } (\text{lzip } \text{lys}' \text{ lys}')) (\text{lmap } \text{snd } (\text{lzip } \text{lys} \text{ lys})) \rangle$.
with rel [*THEN lalist-all2-llengthD*] rel' [*THEN lalist-all2-llengthD*] **have** $\langle \text{eq-cmset } \text{lys}' \text{ lys} \rangle$
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* = $\langle 'a \text{ llist} \rangle / \text{eq-cmset}$

by (*auto simp: eq-cmset-def intro!: equivpI reflpI sympI transpI*)

lift-bnf (*cmset*: $'a$) *cmset*

for *map*: *cmimage* *rel*: *cmrel*

proof –

fix $P :: \langle 'a \Rightarrow 'b \Rightarrow \text{bool} \rangle$ **and** $Q :: \langle 'b \Rightarrow 'c \Rightarrow \text{bool} \rangle$

show $\langle \text{lalist-all2 } P \text{ OO eq-cmset OO lalist-all2 } Q$
 $\leq \text{eq-cmset OO lalist-all2 } (P \text{ OO } Q) \text{ OO eq-cmset} \rangle$

proof *safe*

fix $l \ r \ l' \ r'$

assume $\langle \text{lalist-all2 } P \ l \ l' \rangle \langle \text{eq-cmset } l' \ r' \rangle \langle \text{lalist-all2 } Q \ r' \ r \rangle$

with *lalist-all2-reorder-left-invariance[OF this(3,2)]*

show $\langle (\text{eq-cmset OO lalist-all2 } (P \text{ OO } Q) \text{ OO eq-cmset}) \ l \ r \rangle$

by (*auto intro!: relcomppI[of - - l] simp: eq-cmset-def llist-rel-compp*)

qed

next

fix $S :: \langle 'a \text{ set set} \rangle$

assume $\langle S \neq \{\} \rangle$

then show $\langle (\bigcap As \in S . \{ (x, x') . \text{eq-cmset } x \ x' \}) \text{ “ } \{ x . \text{lset } x \subseteq As \} \rangle$

$\subseteq \{(x, x'). \text{eq-cmset } x \ x'\} \text{ “ } \{x. \text{lset } x \subseteq \bigcap S\}$
using *Inter-greatest*[of $S \ \langle \text{lset } \rightarrow \rangle$] *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 $[[\text{function-internals}]]$

begin

partial-function (*lflat*) *lflat* **where**

$\langle \text{lflat } \text{lxs} = (\text{case } \text{lxs} \text{ of } \text{LNil} \Rightarrow \text{LNil} \mid \text{LCons } x \ \text{lxs} \Rightarrow \text{lappend } (\text{lflat } \text{xs}) \ (\text{lflat } \text{lxs})) \rangle$

end

lemma *ltaken-ldropn-decomp*: $\langle \text{lxs} = \text{lappend } (\text{lflat } (\text{ltaken } n \ \text{lxs})) \ (\text{ldropn } n \ \text{lxs}) \rangle$
by (*induct* n *arbitrary*: lxs) (*auto* *split*: *lflat.splits*)

lemma *count-llist-llist-of[simp]*: $\langle \text{count-llist } (\text{lflat } \text{xs}) \ x = \text{count-list } \text{xs} \ x \rangle$
by (*induct* xs) (*auto* *simp*: *enat-0* *simp* *flip*: *eSuc-enat*)

lemma *lmap-lfilter-swap*: $\langle \forall x \in \text{lset } \text{lxs}. P \ x \longleftrightarrow Q \ (f \ x) \implies \text{lmap } f \ (\text{lfilter } P \ \text{lxs}) = \text{lfilter } Q \ (\text{lmap } f \ \text{lxs}) \rangle$
by (*induct* lxs) *auto*

lemma *lsum-lmap-zero*: $\langle \text{lsum } (\text{lmap } (\lambda z. 0) \ \text{lxs}) = 0 \rangle$

proof –

have $\langle \text{lset } (\text{lmap } (\lambda z. 0) \ \text{lxs}) \subseteq \{0\} \rangle$ **by** *fastforce*
then have $\langle \text{ldropWhile } ((=) 0) \ (\text{lmap } (\lambda z. 0) \ \text{lxs}) = \text{LNil} \rangle$
by (*simp* *add*: *ldropWhile-eq-LNil-iff*)
then show *?thesis*
by (*metis* *lflat.case(1)* *lsum-code-alt*)

qed

lemma *lsum-LNil[simp]*: $\langle \text{lsum } \text{LNil} = 0 \rangle$
by (*simp* *add*: *lsum-code-alt*)

lemma *lsum-LCons[simp]*: $\langle \text{lsum } (\text{LCons } x \ \text{lxs}) = x + \text{lsum } \text{lxs} \rangle$
by (*simp* *add*: *lsum-code-alt*)

lemma *count-llist-lmap-const*:

$\langle \text{count-llist } (\text{lmap } (\lambda z. a) \ \text{lxs}) \ x = (\text{if } x = a \ \text{then } \text{llength } \text{lxs} \ \text{else } 0) \rangle$

unfolding *count-llist-alt*

by (*auto* *simp*: *lfilter-lmap*)

lemma *lsum-lmap-all-but-one-0*: $\langle x \in \text{lset } \text{lxs} \implies \text{ldistinct } \text{lxs} \implies \text{lsum } (\text{lmap } (\lambda z. \text{if } z = x \ \text{then } y \ \text{else } 0) \ \text{lxs}) = y \rangle$

proof (*induct* $x \ \text{lxs}$ *rule*: *lflat.set-induct*)

```

case (LCons1 x' lxs)
then have ⟨lmap (λz. 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 $\langle \text{Sup } (X :: \text{enat set}) = \text{Sup } (X - \{0\}) \rangle$
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*: $\langle \text{Complete-Partial-Order.chain lprefix } Y \implies \exists \text{lys} \in Y. \text{lprefix (lList-of xs) lys} \implies \text{lSup } Y = \text{lappend (lList-of xs) (lSup (ldropn (length xs) ' Y))} \rangle$
proof (*induct xs arbitrary: Y*)
case (*Cons a xs*)
from *Cons(3)* **obtain** *lys where* $\langle \neg \text{lNull lys} \rangle \langle \text{lys} \in Y \rangle \langle \text{lhd lys} = a \rangle \langle \text{lprefix (lList-of xs) (ltl lys)} \rangle$
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*: $\langle \text{lys} \in Y \implies \neg \text{lNull lys} \implies \text{lhd lys} = a \rangle$ **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]*: $\langle \text{lhd (lSup } Y) = a \rangle$
by (*metis lhd-LCons lhd-lSup-eq lList.disc(2) lList-of.simps(2) lprefix-lhdD lprefix-lNull*)
moreover **have** $\langle \text{lSup (ltl ' (Y} \cap \{l. \neg \text{lNull } l\}) = \text{lappend (lList-of xs) (lSup (ldropn (Suc (length xs)) ' Y))} \rangle$
proof –
have $\langle \text{ldropn (Suc (length xs)) ' (Y} \cap \{l. \neg \text{lNull } l\}) \subseteq \text{ldropn (Suc (length xs)) ' Y} \rangle$
(is $\langle ?lhs \subseteq ?rhs \rangle$) **by** *auto*
moreover **have** $\langle ?rhs - \{LNil\} \subseteq ?lhs \rangle$
by *auto*
ultimately **have** $\langle \text{lSup } ?lhs = \text{lSup } ?rhs \rangle$
by (*metis Int-Diff inf.absorb-iff2 inf.order-iff lSup-minus-LNil*)
moreover **have** *chain-ltl*: $\langle \text{Complete-Partial-Order.chain lprefix (ltl ' (Y} \cap \{l. \neg \text{lNull } l\}) \rangle$
by (*metis chain-lprefix-ltl Cons.prem(1)*)
moreover **have** $\langle \text{Bex (ltl ' (Y} \cap \{l. \neg \text{lNull } l\}) (lprefix (lList-of xs)) \rangle$
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*: $\langle x \in \text{lset } lxs \implies \text{lsum } lxs = 0 \implies x = 0 \rangle$
by (*induct x lxs rule: lList.set-induct*) *auto*

lemma *lsum-0I*: $\langle \forall x \in \text{lset } lxs. x = 0 \implies \text{lsum } lxs = 0 \rangle$
by (*subst lsum-code-alt*) *auto*

lemma *lsum-0-iff*: $\langle lsum\ lxs = 0 \longleftrightarrow (\forall x \in lset\ lxs.\ x = 0) \rangle$
by (*metis lsum-0I lsum-0D*)

lemma *lsum-lappend-lfinite*: $\langle lfinite\ lxs \implies lsum\ (lappend\ lxs\ lys) = lsum\ lxs + lsum\ lys \rangle$
by (*induct lxs rule: lfinite.induct*) *auto*

lemma *lsum-lappend*: $\langle lsum\ (lappend\ lxs\ lys) = (if\ lfinite\ lxs\ then\ lsum\ lxs + lsum\ lys\ else\ lsum\ lxs) \rangle$
by (*auto simp: lappend-inf lsum-lappend-lfinite*)

lemma *lsum-llist-of[simp]*: $\langle lsum\ (llist-of\ xs) = sum-list\ xs \rangle$
by (*induct xs*) *auto*

lemma *lsum-cont*:

$\langle Complete-Partial-Order.chain\ lprefix\ Y \implies Y \neq \{\} \implies lsum\ (lSup\ Y) = \bigsqcup (lsum\ ` Y) \rangle$

proof (*coinduction arbitrary: Y rule: enat-coinduct*)

case *Eq-enat*

then show *?case*

proof (*intro allI impI conjI disjI1*)

show $\langle Complete-Partial-Order.chain\ lprefix\ Y \implies Y \neq \{\} \implies (lsum\ (lSup\ Y) = 0) = (\bigsqcup (lsum\ ` Y) = 0) \rangle$

by (*simp add: Sup-eq-0-iff lset-lSup lsum-0-iff*)

next

assume *hn0-lsum*: $\langle lsum\ (lSup\ Y) \neq 0 \rangle$ **and** *hn0-Sup*: $\langle \bigsqcup (lsum\ ` Y) \neq 0 \rangle$

show $\langle (\exists Y'.\ epred\ (lsum\ (lSup\ Y)) = lsum\ (lSup\ Y') \wedge epred\ (\bigsqcup (lsum\ ` Y)) = \bigsqcup (lsum\ ` Y') \wedge$

$Complete-Partial-Order.chain\ lprefix\ Y' \wedge Y' \neq \{\}) \rangle$

proof –

have *lnth-eq-Y*: $\langle lnth\ lxs\ i = lnth\ lys\ i \rangle$

if $\langle enat\ i < llength\ lxs \rangle \langle lxs \in Y \rangle \langle enat\ i < llength\ lys \rangle \langle lys \in Y \rangle$ **for** *lxs lys i*

using *Eq-enat(1)* **that by** (*force simp: chain-def dest: lprefix-lnthD[of - - i]*)

obtain *lys z* **where** *hlys*: $\langle lys \in Y \rangle$ **and** *hz*: $\langle z \in lset\ lys \rangle \langle z \neq 0 \rangle$

using *hn0-lsum Eq-enat(1)* **by** (*auto simp: lsum-0-iff lset-lSup*)

define *A* **where** $\langle A = \{i.\ \forall lys \in Y.\ ltaken\ i\ lys = replicate\ (case\ llength\ lys\ of\ enat\ j \Rightarrow min\ i\ j\ | - \Rightarrow i)\ 0\} \rangle$

define *i* **where** $\langle i = Max\ A \rangle$

have *ne*: $\langle A \neq \{\} \rangle$

using *Eq-enat(2)* $\langle lys \in Y \rangle$ **by** (*auto simp: A-def enat-0 intro!: exI[of - 0] split: enat.splits*)

have *fin*: $\langle finite\ A \rangle$

proof –

from *hn0-lsum Eq-enat(1)* **obtain** *lys'* **where** *hlys'*: $\langle lys' \in Y \rangle$ **and** *hn0*: $\langle lsum\ lys' \neq 0 \rangle$

by (*auto simp: lset-lSup lsum-0-iff*)

from *hn0* **obtain** *n* **where** *hn*: $\langle enat\ n < llength\ lys' \rangle \langle lnth\ lys'\ n \neq 0 \rangle$

```

    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  $\langle \text{ltaken } (\text{Suc } i) \text{ lys}' = \text{ltaken } i \text{ lys}' \rangle$ 
    by (metis ltaken-Suc)
  moreover have  $\langle \text{ltaken } i \text{ lys}' = \text{replicate } (\text{case } \text{llength } \text{lys}' \text{ of } \text{enat } j \Rightarrow$ 
min } i \text{ } j \mid - \Rightarrow i) \text{ } 0 \rangle
    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:  $\langle n \neq 0 \rangle$ 
  using Max-eq-iff[OF fin ne, of i, THEN iffD1, OF i-def[symmetric], THEN
conjunct2, rule-format, of  $\langle \text{Suc } i \rangle$ ]
  hn0 by fastforce
have ldroprn-cases:  $\langle \text{lys} = \text{LNil} \vee (\exists \text{lys}'. \text{lys} = \text{LCons } n \text{ lys}') \rangle$ 
  if  $\langle \text{lys} \in \text{ldroprn } i \text{ ' } Y \rangle$  for lys
  using that hn
  by (metis (mono-tags, lifting) image-iff ldroprn-Suc-conv-ldroprn ldroprn-eq-LNil
linorder-not-le)
have hTHE:  $\langle (\text{THE } x. x \in \text{lhd } \text{' } (\text{ldroprn } i \text{ ' } Y \cap \{xs. \neg \text{lnull } xs\})) = n \rangle$ 
proof (intro the-equality)
  show  $\langle n \in \text{lhd } \text{' } (\text{ldroprn } i \text{ ' } Y \cap \{xs. \neg \text{lnull } xs\}) \rangle$ 
    by (metis IntI hi hlys image-eqI ldroprn-Suc-conv-ldroprn lhd-LCons
linorder-not-le
    llist.disc(2) mem-Collect-eq hn)
show  $\langle \bigwedge x. x \in \text{lhd } \text{' } (\text{ldroprn } i \text{ ' } Y \cap \{xs. \neg \text{lnull } xs\}) \implies x = n \rangle$ 
proof -
  fix x assume  $\langle x \in \text{lhd } \text{' } (\text{ldroprn } i \text{ ' } Y \cap \{xs. \neg \text{lnull } xs\}) \rangle$ 
  then obtain lxs where hlx:  $\langle \text{lxs} \in Y \rangle \langle \neg \text{lnull } (\text{ldroprn } i \text{ } \text{lxs}) \rangle \langle x = \text{lhd}$ 
(ldroprn } i \text{ } \text{lxs}) \rangle
    by auto
  from hlxs(2) have hlen:  $\langle \text{enat } i < \text{llength } \text{lxs} \rangle$ 
    by (simp add: lnull-def ldroprn-eq-LNil not-less)
  from hlxs(3) have  $\langle x = \text{lnth } \text{lxs } i \rangle$ 
    by (simp add: lhd-ldroprn hlen)
  with hn hlx(1) hlen show  $\langle x = n \rangle$ 
    by auto
  qed
qed
have hLCons:  $\langle (\lambda x. \text{LCons } (\text{epred } n) (\text{ttl } x)) \text{ ' } X \cap \{xs. \neg \text{lnull } xs\} =$ 
( $\lambda x. \text{LCons } (\text{epred } n) (\text{ttl } x) \text{ ' } X \rangle$  for } X
  by auto
have h4:  $\langle \text{ldroprn } i \text{ } \text{lys} \in \text{ldroprn } i \text{ ' } Y \cap \{xs. \neg \text{lnull } xs\} \rangle$ 
  using hi hlys by auto
define Y' where  $\langle Y' = (\text{LCons } (\text{epred } n) \circ \text{ttl}) \text{ ' } (\text{ldroprn } i \text{ ' } Y \cap \{xs. \neg \text{lnull}$ 
xs}) \rangle

```

```

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 xs' lys'
  assume hm1: ⟨xs' ∈ (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 xs0 where hxs0: ⟨xs0 ∈ ldropn i ' Y⟩ ⟨¬ lnull xs0⟩
  ⟨xs' = LCons (epred n) (ltl xs0)⟩ by auto
  from hm2 obtain lys0 where hlys0: ⟨lys0 ∈ ldropn i ' Y⟩ ⟨¬ lnull lys0⟩
  ⟨lys' = LCons (epred n) (ltl lys0)⟩ by auto
  from hxs0(1) obtain xs1 where hxs1: ⟨xs1 ∈ Y⟩ ⟨xs0 = ldropn i
xs1⟩ by auto
  from hlys0(1) obtain lys1 where hlys1: ⟨lys1 ∈ Y⟩ ⟨lys0 = ldropn i
lys1⟩ by auto
  from Eq-enat(1) hxs1(1) hlys1(1) have ⟨lprefix xs1 lys1 ∨ lprefix lys1
xs1⟩
  by (auto simp: chain-def)
  thus ⟨lprefix xs' lys' ∨ lprefix lys' xs'⟩
  using hxs0(3) hlys0(3) hxs1(2) hlys1(2)
  by (auto intro: lprefix-ltI 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-rotate 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 wherehya: ⟨ya ∈ Y'⟩ ⟨v = lsum ya⟩ by auto
  fromhya(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)⟩
    usinghya(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
  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) lxs)) =
  lsum (lmap (λlxs. count-llist lxs x) lxs)⟩

```

```

proof (induct lxs)

```

```

  case LNil

```

```

  then show ?case by (auto simp: count-llist-zero-iff)

```

```

next

```

```

  case (LCons lxs lxs)

```

```

  then show ?case by (auto simp: count-llist-zero-iff, metis lnull-def empty-iff
lset-LNil)

```

```

qed simp

```

```

lemma lflat-LNil[simp]:  $\langle \text{lflat } LNil = LNil \rangle$ 
  by (subst lflat.simps) auto

lemma lflat-LCons[simp]:  $\langle \text{lflat } (LCons\ xs\ lxs) = \text{lappend } (\text{llist-of } xs) (\text{lflat } lxs) \rangle$ 
  by (subst lflat.simps) auto

lemma count-llist-mono:
  assumes  $\langle \text{lprefix } lxs\ lys \rangle$ 
  shows  $\langle \text{count-llist } lxs\ x \leq \text{count-llist } lys\ x \rangle$ 
proof (cases  $\langle \text{lfinite } lxs \rangle$ )
  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:
   $\langle \text{ldropWhile } ((\neq)\ x) \text{ ' } Y \cap \{lxs. \neg \text{lnull } lxs\} = \text{ldropWhile } ((\neq)\ x) \text{ ' } \{lxs \in Y. x \in \text{lset } lxs\} \rangle$ 
  by auto

lemma count-llist-cont:  $\langle \text{Complete-Partial-Order.chain lprefix } Y \implies Y \neq \{\} \implies \text{count-llist } (\text{lSup } Y)\ x = (\bigsqcup_{lxs \in Y. \text{count-llist } lxs\ x}) \rangle$ 
proof (coinduction arbitrary: Y)
  case (eq-enat Y)
  then have chain-Y:  $\langle \text{Complete-Partial-Order.chain lprefix } Y \rangle$  and ne-Y:  $\langle Y \neq \{\} \rangle$  by auto
  show ?case
  proof (rule conjI)
  from chain-Y ne-Y show  $\langle (\text{count-llist } (\text{lSup } Y)\ x = 0) = ((\bigsqcup_{lxs \in Y. \text{count-llist } lxs\ x}) = 0) \rangle$ 
    by (auto simp: count-llist-zero-iff Sup-eq-0-iff lset-lSup)
  next
  show  $\langle \text{count-llist } (\text{lSup } Y)\ x \neq 0 \implies (\bigsqcup_{lxs \in Y. \text{count-llist } lxs\ x}) \neq 0 \implies (\exists Z. \text{epred } (\text{count-llist } (\text{lSup } Y)\ x) = \text{count-llist } (\text{lSup } Z)\ x \wedge \text{epred } (\bigsqcup_{lxs \in Y. \text{count-llist } lxs\ x}) = (\bigsqcup_{lxs \in Z. \text{count-llist } lxs\ x}) \wedge \text{Complete-Partial-Order.chain lprefix } Z \wedge Z \neq \{\}) \vee \text{epred } (\text{count-llist } (\text{lSup } Y)\ x) = \text{epred } (\bigsqcup_{lxs \in Y. \text{count-llist } lxs\ x}) \rangle$ 
    proof (intro impI disjI1)
    assume h1:  $\langle \text{count-llist } (\text{lSup } Y)\ x \neq 0 \rangle$  and h2:  $\langle (\bigsqcup_{lxs \in Y. \text{count-llist } lxs\ x}) \neq 0 \rangle$ 
    define Z where  $\langle Z \equiv \text{ltl ' } (\text{ldropWhile } ((\neq)\ x) \text{ ' } Y \cap \{lxs. \neg \text{lnull } lxs\}) \rangle$ 
    have chain-Z:  $\langle \text{Complete-Partial-Order.chain lprefix } Z \rangle$ 
    unfolding Z-def using chain-Y
    by (auto simp: chain-def ldropWhile-not-lnull-alt)

```

```

      intro!: lprefix-ltl 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** $\langle lxs \in \{lxs \in Y. x \in lset\ lxs\} \rangle$
then have $lxs \in Y$ **and** $x \in lset\ lxs$ **by** *auto*
show $\langle \exists m \in Y. count_llist\ (ltl\ (ldropWhile\ ((\neq)\ x)\ lxs))\ x \leq epred\ (count_llist\ m\ x) \rangle$
using $lxs \in Y\ x \in lxs$
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 $\langle \exists Z. epred\ (count_llist\ (lSup\ Y)\ x) = count_llist\ (lSup\ Z)\ x \wedge epred\ (\bigsqcup_{lxs \in Y}. count_llist\ lxs\ x) = (\bigsqcup_{lxs \in Z}. count_llist\ lxs\ x) \wedge Complete_Partial_Order.chain\ lprefix\ Z \wedge Z \neq \{\} \rangle$
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 $mcont_count_llist: \langle mcont\ lSup\ (lprefix)\ Sup\ (\leq)\ (\lambda lxs. count_llist\ lxs\ x) \rangle$
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 $mono_lflat: \langle monotone\ lprefix\ lprefix\ lflat \rangle$
by (*rule llist.fixp-preserves-mono1[OF lflat.mono lflat-def]*) *simp*

lemma *mcont2mcont-lflat*[*THEN llist.mcont2mcont, simp, cont-intro*]:
shows $mcont_lflat: \langle mcont\ lSup\ lprefix\ lSup\ lprefix\ lflat \rangle$
by (*rule llist.fixp-preserves-mcont1[OF lflat.mono lflat-def]*) *simp*

lemma *lflat-LNil-iff*: $\langle lflat\ lxs = LNil \iff (\forall xs \in lset\ lxs. xs = []) \rangle$
by (*induct lxs*) (*auto simp: llist-of-eq-LNil-conv lappend-eq-LNil-iff*)

lemma *count-llist-lflat*: $\langle count_llist\ (lflat\ lxs)\ x = lsum\ (lmap\ (\lambda xs. count_list\ xs\ x)\ lxs) \rangle$
by (*induct lxs*) (*auto simp: count-llist-lappend*)

lemma *lset-lflat*: $\langle lset\ (lflat\ lxs) = (\bigcup_{xs \in lset\ lxs}. set\ xs) \rangle$
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 $\langle lvertical\ lxs\ i = (if\ enat\ i < llength\ lxs\ then\ ltaken\ (Suc\ i)\ (lnth\ lxs\ i)\ else\ []) \rangle$

definition $\langle lhorizontal\ lxs\ j = List.map_filter\ (\lambda lxs. if\ enat\ j < llength\ lxs\ then\ Some\ (lnth\ lxs\ j)\ else\ None)\ (ltaken\ j\ lxs) \rangle$

definition *lmerge* **where**

$\langle lmerge\ lxs = lflat\ (lmap\ (\lambda i. lvertical\ lxs\ i\ @\ lhorizontal\ lxs\ i)\ (lupt\ 0\ \infty)) \rangle$

```

lemma set-ltaken-conv':  $\langle \text{set } (\text{ltaken } n \text{ } lxs) = (\text{case } \text{llength } lxs \text{ of } \text{enat } m \Rightarrow \text{lnth } lxs \text{ ' } \{0..<\text{min } n \text{ } m\} \mid - \Rightarrow \text{lnth } lxs \text{ ' } \{0 ..<n\}) \rangle$ 
proof -
  show ?thesis
  proof -
    { fix m x
      assume  $\langle m < n \rangle \langle \text{llength } lxs = \text{enat } m \rangle \langle x \in \text{set } (\text{ltaken } n \text{ } lxs) \rangle$ 
      then have  $\langle x \in \text{lnth } lxs \text{ ' } \{0..<m\} \rangle$ 
      by (metis enat-ord-simps(1) linorder-not-le ltaken-all-same nle-le set-ltaken-conv)
    }
  moreover
    { fix m l
      assume  $\langle m < n \rangle \langle \text{llength } lxs = \text{enat } m \rangle \langle l < m \rangle$ 
      then have  $\langle \text{lnth } lxs \text{ } l \in \text{set } (\text{ltaken } n \text{ } lxs) \rangle$ 
      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:  $\langle \text{lset } (\text{lmerge } lxs) = (\bigcup lxs \in \text{lset } lxs. \text{lset } lxs) \rangle$ 
proof safe
  fix x
  assume  $\langle x \in \text{lset } (\text{lmerge } lxs) \rangle$ 
  then show  $\langle x \in \bigcup (\text{lset ' } \text{lset } lxs) \rangle$ 
  proof -
    have vert:  $\langle \bigwedge i. \text{set } (\text{lvertical } lxs \text{ } i) \subseteq \bigcup (\text{lset ' } \text{lset } lxs) \rangle$ 
    proof
      fix i y assume hy:  $\langle y \in \text{set } (\text{lvertical } lxs \text{ } i) \rangle$ 
      show  $\langle y \in \bigcup (\text{lset ' } \text{lset } lxs) \rangle$ 
      proof (cases enat i < llength lxs)
        case True
          have hsub:  $\langle \text{set } (\text{lvertical } lxs \text{ } i) \subseteq \text{lset } (\text{lnth } lxs \text{ } i) \rangle$ 
          using True set-ltaken[of Suc i lnth lxs i]
          by (auto simp: lvertical-def intro: lset-intros dest: set-mp[OF set-ltaken])
          have hmem:  $\langle \text{lnth } lxs \text{ } i \in \text{lset } lxs \rangle$ 
          using True by (auto simp: in-lset-conv-lnth)
          show ?thesis using hy hsub hmem by blast
        next
          case False
          then have  $\langle \text{set } (\text{lvertical } lxs \text{ } i) = \{\} \rangle$  by (simp add: lvertical-def)
          then show ?thesis using hy by simp
      qed
    qed
  have horiz:  $\langle \bigwedge j. \text{set } (\text{lhorizontal } lxs \text{ } j) \subseteq \bigcup (\text{lset ' } \text{lset } lxs) \rangle$ 
  proof

```

```

    fix j y assume hy: ⟨y ∈ set (lhorizontal lxs j)⟩
    then obtain n where hn: ⟨enat n < llength lxs⟩ and hjn: ⟨enat j < llength
(lnth lxs n)⟩
      and yj: ⟨y = lnth (lnth lxs 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 lxs n ∈ lset lxs⟩
      using hn by (auto simp: in-lset-conv-lnth)
    have hin: ⟨y ∈ lset (lnth lxs n)⟩
      using hjn yj by (auto simp: in-lset-conv-lnth)
    show ⟨y ∈ ⋃ (lset ' lset lxs)⟩
      using hmem hin by blast
  qed
  from ⟨x ∈ lset (lmerge lxs)⟩
  obtain i where ⟨x ∈ set (lvertical lxs i) ∨ x ∈ set (lhorizontal lxs 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 xlxs: ⟨lxs ∈ lset lxs⟩ and xlx: ⟨x ∈ lset lxs⟩
  from xlxs obtain i where hi: ⟨enat i < llength lxs⟩ and lxi: ⟨lnth lxs i = lxs⟩
    by (auto simp: in-lset-conv-lnth)
  from xlx 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 lxs)⟩
  proof (cases ⟨j ≤ i⟩)
    case jle: True
      have x-in-vert: ⟨x ∈ set (lvertical lxs 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)
        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
      end
  end

```

```

    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 lxs 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
  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*:

$\langle \text{count-list } (\text{lvertical } \text{lxss } i) \ x = (\text{if } i < \text{llength } \text{lxss} \text{ then } \text{card } \{j. j \leq i \wedge j < \text{llength } (\text{lnth } \text{lxss } i) \wedge \text{lnth } (\text{lnth } \text{lxss } i) \ j = x\} \text{ else } 0) \rangle$

proof (*cases* $\langle \text{enat } i < \text{llength } \text{lxss} \rangle$)

case *False*

then show *?thesis* **by** (*simp add: lvertical-def*)

next

case *ili: True*

have *lv*: $\langle \text{lvertical } \text{lxss } i = \text{ltaken } (\text{Suc } i) (\text{lnth } \text{lxss } i) \rangle$

using *ili* **by** (*simp add: lvertical-def*)

show *?thesis*

proof (*cases* $\langle \text{enat } (\text{Suc } i) \leq \text{llength } (\text{lnth } \text{lxss } i) \rangle$)

case *leq: True*

have *len*: $\langle \text{length } (\text{ltaken } (\text{Suc } i) (\text{lnth } \text{lxss } i)) = \text{Suc } i \rangle$

by (*simp del: ltaken.simps add: length-ltaken leq*)

have *nth*: $\langle \bigwedge j. j < \text{Suc } i \implies \text{ltaken } (\text{Suc } i) (\text{lnth } \text{lxss } i) ! j = \text{lnth } (\text{lnth } \text{lxss } i) \ j \rangle$

by (*metis leq nth-ltaken*)

have *li*: $\langle \bigwedge j. j \leq i \implies \text{enat } j < \text{llength } (\text{lnth } \text{lxss } i) \rangle$

proof –

fix *j* **assume** *hj*: $\langle j \leq i \rangle$

have $\langle \text{enat } (\text{Suc } j) \leq \text{enat } (\text{Suc } i) \rangle$ **using** *hj* **by** *simp*

also have $\langle \text{enat } (\text{Suc } i) \leq \text{llength } (\text{lnth } \text{lxss } i) \rangle$ **using** *leq* **by** (*simp add: Suc-ile-eq*)

finally show $\langle \text{enat } j < \text{llength } (\text{lnth } \text{lxss } i) \rangle$ **by** (*simp add: Suc-ile-eq*)

qed

have *set-eq*: $\langle \{j. j < \text{length } (\text{ltaken } (\text{Suc } i) (\text{lnth } \text{lxss } i)) \wedge \text{ltaken } (\text{Suc } i) (\text{lnth } \text{lxss } i) ! j = x\} = \{j. j \leq i \wedge \text{enat } j < \text{llength } (\text{lnth } \text{lxss } i) \wedge \text{lnth } (\text{lnth } \text{lxss } i) \ j = x\} \rangle$

by (*auto simp del: ltaken.simps simp add: len nth li less-Suc-eq-le*)

have *key*: $\langle \text{count-list } (\text{ltaken } (\text{Suc } i) (\text{lnth } \text{lxss } i)) \ x = \text{card } \{j. j \leq i \wedge \text{enat } j < \text{llength } (\text{lnth } \text{lxss } i) \wedge \text{lnth } (\text{lnth } \text{lxss } i) \ j = x\} \rangle$

by (*simp only: count-list-alt set-eq*)

have $\langle \text{count-list } (\text{lvertical } \text{lxss } i) \ x = \text{count-list } (\text{ltaken } (\text{Suc } i) (\text{lnth } \text{lxss } i)) \ x \rangle$

by (*simp only: lv*)

also have $\langle \dots = \text{card } \{j. j \leq i \wedge \text{enat } j < \text{llength } (\text{lnth } \text{lxss } i) \wedge \text{lnth } (\text{lnth } \text{lxss } i) \ j = x\} \rangle$

by (*rule key*)

also have $\langle \dots = (\text{if } \text{enat } i < \text{llength } \text{lxss} \text{ then } \text{card } \{j. j \leq i \wedge \text{enat } j < \text{llength } (\text{lnth } \text{lxss } i) \wedge \text{lnth } (\text{lnth } \text{lxss } i) \ j = x\} \text{ else } 0) \rangle$

by (*simp add: ili*)

finally show *?thesis* .

next

case *nleq: False*

then obtain *k* **where** *fin*: $\langle \text{llength } (\text{lnth } \text{lxss } i) = \text{enat } k \rangle$

by (*meson less-enatE linorder-not-le*)

from *nleq fin* **have** *kl*: $\langle k < \text{Suc } i \rangle$ **by** (*simp add: Suc-ile-eq*)

have *len-eq*: $\langle \text{length } (\text{ltaken } (\text{Suc } i) (\text{lth } \text{lxss } i)) = k \rangle$
using *nleq fin* **by** (*simp del: ltaken.simps add: length-ltaken*)
have *nth-eq*: $\langle \bigwedge m. m < k \implies \text{ltaken } (\text{Suc } i) (\text{lth } \text{lxss } i) ! m = \text{lth } (\text{lth } \text{lxss } i) m \rangle$
by (*metis len-eq lth-lappend-llist-of ltaken-ldropn-decomp*)
have *li*: $\langle \bigwedge j. j < k \implies \text{enat } j < \text{llength } (\text{lth } \text{lxss } i) \rangle$
by (*simp add: fin*)
have *le-i*: $\langle k \leq i \rangle$ **using** *klt* **by** *simp*
have *set-eq*: $\langle \{j. j < \text{length } (\text{ltaken } (\text{Suc } i) (\text{lth } \text{lxss } i)) \wedge \text{ltaken } (\text{Suc } i) (\text{lth } \text{lxss } i) ! j = x \}$
 $= \{j. j \leq i \wedge \text{enat } j < \text{llength } (\text{lth } \text{lxss } i) \wedge \text{lth } (\text{lth } \text{lxss } i) j = x \} \rangle$
by (*auto simp del: ltaken.simps simp add: len-eq nth-eq li fin*
dest: order-less-le-trans[OF - le-i])
have *key*: $\langle \text{count-list } (\text{ltaken } (\text{Suc } i) (\text{lth } \text{lxss } i)) x =$
 $\text{card } \{j. j \leq i \wedge \text{enat } j < \text{llength } (\text{lth } \text{lxss } i) \wedge \text{lth } (\text{lth } \text{lxss } i) j = x \} \rangle$
by (*simp only: count-list-alt set-eq*)
have $\langle \text{count-list } (\text{lvertical } \text{lxss } i) x = \text{count-list } (\text{ltaken } (\text{Suc } i) (\text{lth } \text{lxss } i)) x \rangle$
by (*simp only: lv*)
also have $\langle \dots = \text{card } \{j. j \leq i \wedge \text{enat } j < \text{llength } (\text{lth } \text{lxss } i) \wedge \text{lth } (\text{lth } \text{lxss } i) j = x \} \rangle$
by (*rule key*)
also have $\langle \dots = (\text{if } \text{enat } i < \text{llength } \text{lxss}$
 $\text{then } \text{card } \{j. j \leq i \wedge \text{enat } j < \text{llength } (\text{lth } \text{lxss } i) \wedge \text{lth } (\text{lth } \text{lxss } i) j =$
 $x \} \text{ else } 0) \rangle$
by (*simp add: ili*)
finally show *?thesis* .
qed
qed

lemma *llength-less-lfinite[simp]*: $\langle \text{llength } \text{lxss} < \text{enat } j \implies \text{lfinite } \text{lxss} \rangle$
using *enat-iless lfinite-conv-llength-enat* **by** *blast*

lemma *ltaken-all*: $\langle \text{llength } \text{lxss} < \text{enat } j \implies \text{ltaken } j \text{ lxss} = \text{list-of } \text{lxss} \rangle$
by (*metis enat-iless length-ltaken linorder-not-le list-of-llist-of*
llength-llist-of lth-equalityI lth-lappend1 ltaken-ldropn-decomp
the-enat.simps)

lemma *count-list-map-filter*:
 $\langle \text{count-list } (\text{List.map-filter } f \text{ xs}) x = \text{card } \{i. i < \text{length } \text{xs} \wedge f (\text{xs } ! i) = \text{Some } x \} \rangle$
proof (*induct xs*)
case *Nil*
show *?case* **by** *simp*
next
case (*Cons a xs*)
note *IH = Cons.hyps*
show *?case*
proof (*cases* $\langle f a \rangle$)
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 lxs j) x = card {i. i < j ∧ i < llength lxs ∧ j < llength

$(\text{lnth } \text{lxss } i) \wedge \text{lnth } (\text{lnth } \text{lxss } i) j = x \rangle$
unfolding *horizontal-def count-list-map-filter*
proof (rule *arg-cong*[**where** $f = \text{card}$])
show $\langle \{i. i < \text{length } (\text{ltaken } j \text{ lxss}) \wedge (\text{if } \text{enat } j < \text{llength } (\text{ltaken } j \text{ lxss } ! i) \text{ then } \text{Some } (\text{lnth } (\text{ltaken } j \text{ lxss } ! i) j) \text{ else } \text{None}) = \text{Some } x \} = \{i. i < j \wedge \text{enat } i < \text{llength } \text{lxss} \wedge \text{enat } j < \text{llength } (\text{lnth } \text{lxss } i) \wedge \text{lnth } (\text{lnth } \text{lxss } i) j = x \} \rangle$
proof (rule *set-eqI*, rule *iffI*)
fix i
assume $\langle i \in \{i. i < \text{length } (\text{ltaken } j \text{ lxss}) \wedge (\text{if } \text{enat } j < \text{llength } (\text{ltaken } j \text{ lxss } ! i) \text{ then } \text{Some } (\text{lnth } (\text{ltaken } j \text{ lxss } ! i) j) \text{ else } \text{None}) = \text{Some } x \} \rangle$
hence $h\text{-lt}$: $\langle i < \text{length } (\text{ltaken } j \text{ lxss}) \rangle$ **and** $h\text{-if}$: $\langle \text{enat } j < \text{llength } (\text{ltaken } j \text{ lxss } ! i) \rangle$
and $h\text{-x}$: $\langle \text{lnth } (\text{ltaken } j \text{ lxss } ! i) j = x \rangle$
by (*auto split: if-splits*)
have $h\text{-i-lt}$: $\langle \text{enat } i < \text{llength } \text{lxss} \rangle$
proof (*cases* $\langle \text{enat } j \leq \text{llength } \text{lxss} \rangle$)
case *True*
with $h\text{-lt}$ **have** $\langle i < j \rangle$ **by** (*simp add: length-ltaken*)
with *True* **show** *?thesis* **by** (*simp add: order-less-le-subst2*)
next
case *False*
with $h\text{-lt}$ **have** $\langle i < \text{the-enat } (\text{llength } \text{lxss}) \rangle$ **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 $h\text{-key}$: $\langle \text{ltaken } j \text{ lxss } ! i = \text{lnth } \text{lxss } i \rangle$
by (*metis h-lt length-ltaken linorder-not-le llength-less-lfinite ltaken-all nth-list-of nth-ltaken*)
show $\langle i \in \{i. i < j \wedge \text{enat } i < \text{llength } \text{lxss} \wedge \text{enat } j < \text{llength } (\text{lnth } \text{lxss } i) \wedge \text{lnth } (\text{lnth } \text{lxss } i) j = x \} \rangle$
using $h\text{-lt } h\text{-if } h\text{-x } h\text{-i-lt } h\text{-key}$
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 $\langle i \in \{i. i < j \wedge \text{enat } i < \text{llength } \text{lxss} \wedge \text{enat } j < \text{llength } (\text{lnth } \text{lxss } i) \wedge \text{lnth } (\text{lnth } \text{lxss } i) j = x \} \rangle$
hence $h\text{-j}$: $\langle i < j \rangle$ **and** $h\text{-lt}$: $\langle \text{enat } i < \text{llength } \text{lxss} \rangle$
and $h\text{-llength}$: $\langle \text{enat } j < \text{llength } (\text{lnth } \text{lxss } i) \rangle$ **and** $h\text{-x}$: $\langle \text{lnth } (\text{lnth } \text{lxss } i) j = x \rangle$
by *auto*
have $h\text{-key}$: $\langle \text{ltaken } j \text{ lxss } ! i = \text{lnth } \text{lxss } i \rangle$
by (*metis h-j llength-less-lfinite ltaken-all not-le-imp-less nth-list-of nth-ltaken*)
show $\langle i \in \{i. i < \text{length } (\text{ltaken } j \text{ lxss}) \wedge (\text{if } \text{enat } j < \text{llength } (\text{ltaken } j \text{ lxss } ! i) \text{ then } \text{Some } (\text{lnth } (\text{ltaken } j \text{ lxss } ! i) j) \text{ else } \text{None}) = \text{Some } x \} \rangle$
then $\text{Some } (\text{lnth } (\text{ltaken } j \text{ lxss } ! i) j) \text{ else } \text{None} = \text{Some } x \rangle$
using $h\text{-j } h\text{-lt } h\text{-llength } h\text{-x } h\text{-key}$
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** $\langle \text{lprefix } lxs \text{ } lys \rangle$
 $\langle \forall i < \text{llength } lxs. f (\text{lnth } lxs \ i) = g (\text{lnth } lys \ i) \rangle$
 $\langle \forall i < \text{llength } lys. i \geq \text{llength } lxs \longrightarrow g (\text{lnth } lys \ i) = 0 \rangle$
shows $\langle \text{LSUM } lxs \ f = \text{LSUM } lys \ g \rangle$
proof –
have *key*: $\langle \text{LSUM } lxs \ f = \text{LSUM } lys \ g \rangle$ **if** $\langle \text{lfinite } lxs \rangle$
using *that assms*
proof (*induct lxs arbitrary: lys rule: lfinite.induct*)
case (*lfinite-LNil lys*)
have $\langle \forall x \in \text{lset } lys. g \ x = 0 \rangle$
using *lfinite-LNil(3)* **by** (*auto simp: in-lset-conv-lnth*)
then have $\langle \text{LSUM } lys \ g = 0 \rangle$ **by** (*simp add: lsum-0-iff*)
then show *?case* **by** *simp*
next
case (*lfinite-LConsI xs x lys-outer*)
obtain *ys'* **where** *ys-eq*: $\langle \text{lys-outer} = \text{LCons } x \ \text{ys}' \rangle$ **and** *pre*: $\langle \text{lprefix } xs \ \text{ys}' \rangle$
using *lfinite-LConsI(3)* **by** (*auto simp: LCons-lprefix-conv*)
have *eq-x*: $\langle f \ x = g \ x \rangle$
using *spec[OF lfinite-LConsI(4), of 0]* **unfolding** *ys-eq*
by (*simp add: zero-enat-def[symmetric]*)
have *IH5*: $\langle \forall i. \text{enat } i < \text{llength } xs \longrightarrow f (\text{lnth } xs \ i) = g (\text{lnth } \text{ys}' \ i) \rangle$
proof (*intro allI impI*)
fix *i* **assume** *hlt*: $\langle \text{enat } i < \text{llength } xs \rangle$
then have *hi*: $\langle \text{enat } (\text{Suc } i) < \text{llength } (\text{LCons } x \ xs) \rangle$ **by** (*simp add: Suc-ile-eq*)
have *h5i*: $\langle \text{enat } (\text{Suc } i) < \text{llength } (\text{LCons } x \ xs) \longrightarrow$
 $f (\text{lnth } (\text{LCons } x \ xs) \ (\text{Suc } i)) = g (\text{lnth } \text{lys-outer} \ (\text{Suc } i)) \rangle$
by (*rule spec[OF lfinite-LConsI(4), of 'Suc i']*)
show $\langle f (\text{lnth } xs \ i) = g (\text{lnth } \text{ys}' \ i) \rangle$
using *h5i hi* **unfolding** *ys-eq* **by** *simp*
qed
have *IH6*: $\langle \forall i. \text{enat } i < \text{llength } \text{ys}' \longrightarrow \text{llength } xs \leq \text{enat } i \longrightarrow g (\text{lnth } \text{ys}' \ i) = 0 \rangle$
proof (*intro allI impI*)
fix *i* **assume** *hi*: $\langle \text{enat } i < \text{llength } \text{ys}' \rangle$ **and** *hi2*: $\langle \text{llength } xs \leq \text{enat } i \rangle$
have *h1*: $\langle \text{enat } (\text{Suc } i) < \text{llength } \text{lys-outer} \rangle$
unfolding *ys-eq* **by** (*simp add: Suc-ile-eq hi*)
have *h2*: $\langle \text{llength } (\text{LCons } x \ xs) \leq \text{enat } (\text{Suc } i) \rangle$
by (*simp add: eSuc-enat[symmetric] hi2*)
have *h6i*: $\langle \text{enat } (\text{Suc } i) < \text{llength } \text{lys-outer} \longrightarrow$
 $\text{llength } (\text{LCons } x \ xs) \leq \text{enat } (\text{Suc } i) \longrightarrow g (\text{lnth } \text{lys-outer} \ (\text{Suc } i)) = 0 \rangle$
by (*rule spec[OF lfinite-LConsI(5), of 'Suc i']*)
show $\langle g (\text{lnth } \text{ys}' \ i) = 0 \rangle$
using *h6i h1 h2* **unfolding** *ys-eq* **by** *simp*
qed
have *eq-tails*: $\langle \text{LSUM } xs \ f = \text{LSUM } \text{ys}' \ g \rangle$
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)
  end

```

```

next
case (Suc n P)
have bound: ⟨ $\bigwedge i. (P i \wedge i \neq \text{Suc } n) \implies i \leq n$ ⟩
  using Suc.premis by (force simp: le-Suc-eq)
have IH: ⟨card {i. P i  $\wedge$  i  $\neq$  Suc n} =
  sum-list (map (λi. if P i  $\wedge$  i  $\neq$  Suc n then 1 else 0) [0.. $\text{Suc } n$ ])⟩
  by (rule Suc.hyps[of ⟨λi. P i  $\wedge$  i  $\neq$  Suc n⟩, OF bound])
have sum-eq: ⟨sum-list (map (λi. if P i  $\wedge$  i  $\neq$  Suc n then 1 else 0) [0.. $\text{Suc } n$ ])
=
  sum-list (map (λi. if P i then 1 else 0) [0.. $\text{Suc } n$ ])⟩
  by (intro arg-cong[where f=sum-list] map-cong refl)
  (auto simp: le-Suc-eq dest: Suc.premis)
show ?case
proof (cases ⟨P (Suc n)⟩)
case True
  have fin: ⟨finite {i. P i  $\wedge$  i  $\neq$  Suc n}⟩
    by (rule finite-subset[of - ⟨{0.. $\text{Suc } n$ }⟩])
    (auto simp: le-Suc-eq dest: Suc.premis)
  have notin: ⟨Suc n  $\notin$  {i. P i  $\wedge$  i  $\neq$  Suc n}⟩ by simp
  have insert-eq: ⟨{i. P i} = insert (Suc n) {i. P i  $\wedge$  i  $\neq$  Suc n}⟩
    using True by auto
  have ⟨card {i. P i} = Suc (card {i. P i  $\wedge$  i  $\neq$  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.. $\text{Suc } n$ ]))⟩
    by (metis (mono-tags, lifting) IH sum-eq)
  also have ⟨... = sum-list (map (λi. if P i then 1 else 0) [0.. $\text{Suc } (\text{Suc } n)$ ])⟩
    using True by simp
  finally show ?thesis .
case False
  have eq: ⟨{i. P i} = {i. P i  $\wedge$  i  $\neq$  Suc n}⟩ using False by auto
  have ⟨card {i. P i} = sum-list (map (λi. if P i then 1 else 0) [0.. $\text{Suc } n$ ])⟩
    unfolding eq by (metis (mono-tags, lifting) IH sum-eq)
  also have ⟨... = sum-list (map (λi. if P i then 1 else 0) [0.. $\text{Suc } (\text{Suc } n)$ ])⟩
    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 ⟨ $\bigwedge i. P i \implies i \leq n$ ⟩
  shows ⟨enat (card {i :: nat. P i}) = LSUM lenats (λi. if P i then 1 else 0)⟩
proof -

```

have $\langle \text{enat } (\text{card } \{i :: \text{nat. } P \ i\}) =$
 $\text{enat } (\sum i \leftarrow [0..< \text{Suc } n]. \text{ if } P \ i \text{ then } 1 \text{ else } 0) \rangle$
using $\text{card-sum-list}[OF \ \text{assms}] \text{ by } \text{simp}$
also have $\langle \dots = \text{sum-list } (\text{map } (\text{enat } \circ (\lambda i. \text{ if } P \ i \text{ then } 1 \text{ else } 0))) [0..< \text{Suc } n] \rangle$
by $(\text{simp only: enat-sum-list list.map-comp})$
also have $\langle \dots = \text{LSUM } (\text{llist-of } [0..< \text{Suc } n]) (\text{enat } \circ (\lambda i. \text{ if } P \ i \text{ then } 1 \text{ else } 0)) \rangle$
by $(\text{simp only: LSUM-llist-of})$
also have $\langle \dots = \text{LSUM lenats } (\lambda i. \text{ if } P \ i \text{ then } 1 \text{ else } 0) \rangle$
proof $(\text{rule LSUM-extend})$
show $\langle \text{lprefix } (\text{llist-of } [0..< \text{Suc } n]) \text{ lenats} \rangle$
by $(\text{simp add: llist-of-lupt lprefix-lupt del: upt.simps})$
show $\langle \forall i < \text{llength } (\text{llist-of } [0..< \text{Suc } n]).$
 $(\text{enat } \circ (\lambda i. \text{ if } P \ i \text{ then } 1 \text{ else } 0)) (\text{lnth } (\text{llist-of } [0..< \text{Suc } n]) \ i) =$
 $(\lambda i. \text{ if } P \ i \text{ then } 1 \text{ else } 0) (\text{lnth } \text{lenats } \ i) \rangle$
by $(\text{simp add: lnth-lupt one-enat-def enat-0 del: upt.simps})$
show $\langle \forall i < \text{llength } \text{lenats}.$
 $\text{llength } (\text{llist-of } [0..< \text{Suc } n]) \leq \text{enat } \ i \longrightarrow$
 $(\lambda i. \text{ if } P \ i \text{ then } 1 \text{ else } 0) (\text{lnth } \text{lenats } \ i) = 0 \rangle$
by $(\text{simp add: lnth-lupt del: upt.simps}) (\text{metis assms not-less-eq-eq})$
qed
finally show $?thesis .$
qed

lemma *LSUM-extend-lenats*:

$\langle \text{LSUM } \text{lx} \ f = \text{LSUM } \text{lenats } (\lambda i. \text{ if } \text{enat } \ i < \text{llength } \ \text{lx} \ \text{then } f \ (\text{lnth } \ \text{lx} \ \ i) \ \text{else } 0) \rangle$

proof $-$

have $\text{step1: } \langle \text{LSUM } (\text{lupt } 0 \ (\text{llength } \ \text{lx})) \ (f \ \circ \ \text{lnth } \ \text{lx}) =$
 $\text{LSUM } \text{lenats } (\lambda i. \text{ if } \text{enat } \ i < \text{llength } \ \text{lx} \ \text{then } f \ (\text{lnth } \ \text{lx} \ \ i) \ \text{else } 0) \rangle$

proof $(\text{rule LSUM-extend})$

show $\langle \text{lprefix } (\text{lupt } 0 \ (\text{llength } \ \text{lx})) \ \text{lenats} \rangle$

by $(\text{simp add: lprefix-lupt})$

next

show $\langle \forall i. \text{enat } \ i < \text{llength } (\text{lupt } 0 \ (\text{llength } \ \text{lx})) \longrightarrow$
 $(f \ \circ \ \text{lnth } \ \text{lx}) (\text{lnth } (\text{lupt } 0 \ (\text{llength } \ \text{lx})) \ i) =$
 $(\lambda i. \text{ if } \text{enat } \ i < \text{llength } \ \text{lx} \ \text{then } f \ (\text{lnth } \ \text{lx} \ \ i) \ \text{else } 0) (\text{lnth } \ \text{lenats } \ i) \rangle$
by $(\text{simp add: lnth-lupt})$

next

show $\langle \forall i. \text{enat } \ i < \text{llength } \ \text{lenats} \longrightarrow \text{llength } (\text{lupt } 0 \ (\text{llength } \ \text{lx})) \leq \text{enat } \ i$
 \longrightarrow
 $(\lambda i. \text{ if } \text{enat } \ i < \text{llength } \ \text{lx} \ \text{then } f \ (\text{lnth } \ \text{lx} \ \ i) \ \text{else } 0) (\text{lnth } \ \text{lenats } \ i) = 0 \rangle$
by $(\text{fastforce simp: lnth-lupt})$

qed

show $?thesis$

by $(\text{subst llist-conv-lmap-lupt}) (\text{simp add: llist.map-comp flip: step1})$

qed

lemma *LCons-eq-lmap-conv*:

$\langle (\text{LCons } \ y \ \text{lys} = \text{lmap } \ f \ \text{lx}) = (\exists x \ \text{lx}' . \ \text{lx} = \text{LCons } \ x \ \text{lx}' \wedge \ y = f \ x \wedge \ \text{lys} =$

$\text{lmap } f \text{ } lxs \rangle$
using $\text{lmap-eq-LCons-conv}$ **by** fastforce

lemma lmap-eq-lappend : $\langle \text{lfinite } lys \implies \text{lmap } f \text{ } lxs = \text{lappend } lys \text{ } lzs \iff$
 $(\exists lus \text{ } lvs. \text{ } lxs = \text{lappend } lus \text{ } lvs \wedge lys = \text{lmap } f \text{ } lus \wedge lzs = \text{lmap } f \text{ } lvs) \rangle$

proof ($\text{induct } lys$ *arbitrary*: lxs *rule*: lfinite-induct)
case $LNil$
then show $?case$
by ($\text{auto simp: lnull-def LNil-eq-lmap}$)

next
case ($LCons \text{ } lys \text{ } lxs$)
from $LCons(3)[\text{of } \langle \text{ltl } lxs \rangle] LCons(1,2)$ **show** $?case$
by ($\text{cases } lxs; \text{cases } lys$)
 $(\text{fastforce simp: lnull-def lmap-eq-LCons-conv LCons-eq-lmap-conv lmap-lappend-distrib}$
 $\text{intro: exI}[\text{of } - \langle LCons \text{ } - \rangle])$

qed

lemma $\text{llist-of-eq-lmap-conv}$:
 $\langle \text{llist-of } xs = \text{lmap } f \text{ } lxs \iff (\exists ys. \text{ } lxs = \text{llist-of } ys \wedge \text{map } f \text{ } ys = xs) \rangle$
by ($\text{induct } xs$ *arbitrary*: lxs) ($\text{force simp: LNil-eq-lmap LCons-eq-lmap-conv map-eq-Cons-conv}$)

lemma neq-LCons-conv : $\langle (\forall x \text{ } lxs. \text{ } lys \neq LCons \text{ } x \text{ } lxs) \iff lys = LNil \rangle$
by ($\text{cases } lys; \text{simp}$)

lemma epred-iadd : $\langle \text{epred } (a + b) = (\text{if } a = 0 \text{ then } \text{epred } b \text{ else } \text{epred } a + b) \rangle$
by ($\text{cases } a$ *rule*: co.enat.exhaust) ($\text{auto simp: epred-iadd1}$)

lemma LSUM-isum : $\langle \text{ldistinct } lxs \implies \text{LSUM } lxs \text{ } f = \text{isum } f \text{ } (\text{lset } lxs) \rangle$

proof ($\text{coinduction arbitrary: } lxs \text{ } f$)

case eq-enat

show $?case$

proof ($\text{intro conjI impI disjI1}$)

show $\langle (\text{LSUM } lxs \text{ } f = 0) = (\text{isum } f \text{ } (\text{lset } lxs) = 0) \rangle$

by ($\text{auto simp: lsum-0-iff isum-eq-0-iff}$)

next

assume $\text{hne1: } \langle \text{LSUM } lxs \text{ } f \neq 0 \rangle$ **and** $\text{hne2: } \langle \text{isum } f \text{ } (\text{lset } lxs) \neq 0 \rangle$

have $\text{hexists: } \langle \exists x \in \text{lset } (\text{lmap } f \text{ } lxs). \text{ } x \neq 0 \rangle$

using hne1 **by** ($\text{auto simp: lsum-0-iff}$)

have $\text{hfin: } \langle \text{lfinite } (\text{ltakeWhile } ((=) \text{ } 0) (\text{lmap } f \text{ } lxs)) \rangle$

using hexists **by** ($\text{auto simp: lfinite-ltakeWhile}$)

have $\text{hnotnil: } \langle \text{ldropWhile } ((=) \text{ } 0) (\text{lmap } f \text{ } lxs) \neq LNil \rangle$

using hexists **by** ($\text{auto simp: ldropWhile-eq-LNil-iff}$)

obtain $z \text{ } lzs$ **where** $\text{hlws: } \langle \text{ldropWhile } ((=) \text{ } 0) (\text{lmap } f \text{ } lxs) = LCons \text{ } z \text{ } lzs \rangle$

using hnotnil **by** ($\text{cases } \langle \text{ldropWhile } ((=) \text{ } 0) (\text{lmap } f \text{ } lxs) \rangle$) auto

have $\text{hnz: } \langle z \neq 0 \rangle$

using hexists hlws **by** ($\text{simp add: ldropWhile-eq-LCons-iff}$)

define zs **where** $\langle zs = \text{list-of } (\text{ltakeWhile } ((=) \text{ } 0) (\text{lmap } f \text{ } lxs)) \rangle$

have $\text{hmap: } \langle \text{lmap } f \text{ } lxs = \text{lappend } (\text{llist-of } zs) (LCons \text{ } z \text{ } lzs) \rangle$

unfolding $zs\text{-def}$ **using** hfin hlws

```

    by (metis lappend-ltakeWhile-ldropWhile llist-of-list-of)
  have hzeros:  $\langle \forall x \in \text{set } zs. x = 0 \rangle$ 
    using hfin lset-ltakeWhileD zs-def by fastforce
  let ?f =  $\langle \lambda y. \text{if } y = \text{lnth } lxs \text{ (length } zs) \text{ then } \text{epred } (f y) \text{ else } f y \rangle$ 
  have hep1:  $\langle \text{epred } (LSUM lxs f) = LSUM lxs ?f \rangle$ 
  proof -
    { fix ys ::  $\langle 'a \text{ list} \rangle$  and x ::  $\langle 'a \rangle$  and lxs' ::  $\langle 'a \text{ llist} \rangle$ 
      assume hall:  $\langle \forall z \in \text{set } ys. f z = 0 \rangle$  and hfx:  $\langle f x \neq 0 \rangle$ 
        and hnotinys:  $\langle x \notin \text{set } ys \rangle$  and hnotinlxs':  $\langle x \notin \text{lset } lxs' \rangle$ 
      have hsum0:  $\langle \text{sum-list } (map f ys) = 0 \rangle$ 
        using hall by simp
      have hsum-if0:  $\langle (\sum z \leftarrow ys. \text{if } z = x \text{ then } \text{epred } (f z) \text{ else } f z) = 0 \rangle$ 
        using hnotinys hall by (auto intro!: sum-list-0 split: if-splits)
      have hLSUM-eq:  $\langle LSUM lxs' (\lambda z. \text{if } z = x \text{ then } \text{epred } (f z) \text{ else } f z) = LSUM$ 
 $lxs' f \rangle$ 
        by (rule arg-cong[where f= $\langle lsum \rangle$ ], rule llist.map-cong0) (use hnotinlxs'
    in auto)
      have  $\langle \text{epred } (\text{sum-list } (map f ys) + (f x + LSUM lxs' f)) =$ 
 $(\sum z \leftarrow ys. \text{if } z = x \text{ then } \text{epred } (f z) \text{ else } f z) + (\text{epred } (f x) + LSUM lxs'$ 
 $(\lambda z. \text{if } z = x \text{ then } \text{epred } (f z) \text{ else } f z)) \rangle$ 
        using hfx hsum0 hsum-if0 hLSUM-eq by (simp add: epred-iadd1 hsum0
    hsum-if0)
      }
    thus ?thesis
      using hmap 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:  $\langle \text{epred } (\text{isum } f (\text{lset } lxs)) = \text{isum } ?f (\text{lset } lxs) \rangle$ 
  proof -
    { fix ys ::  $\langle 'a \text{ list} \rangle$  and x ::  $\langle 'a \rangle$  and lxs' ::  $\langle 'a \text{ llist} \rangle$ 
      assume hall:  $\langle \forall z \in \text{set } ys. f z = 0 \rangle$  and hfx:  $\langle f x \neq 0 \rangle$ 
        and hnotinys:  $\langle x \notin \text{set } ys \rangle$  and hnotinlxs':  $\langle x \notin \text{lset } lxs' \rangle$ 
        and hdist:  $\langle \text{distinct } ys \rangle$  and hsep:  $\langle \text{set } ys \cap \text{lset } lxs' = \{\} \rangle$ 
      have hisum-ys0:  $\langle \text{isum } f (\text{set } ys) = 0 \rangle$ 
        using hall by (simp add: isum-eq-0-iff)
      have hisum-ys-if0:  $\langle \text{isum } (\lambda y. \text{if } y = x \text{ then } \text{epred } (f y) \text{ else } f y) (\text{set } ys) =$ 
 $0 \rangle$ 
        using hnotinys hall by (auto simp: isum-eq-0-iff)
      have hisum-lxs'-eq:  $\langle \text{isum } (\lambda y. \text{if } y = x \text{ then } \text{epred } (f y) \text{ else } f y) (\text{lset } lxs')$ 
 $= \text{isum } f (\text{lset } lxs') \rangle$ 
        using hnotinlxs' by (intro isum-cong) auto
      have  $\langle \text{epred } (\text{isum } f (\text{set } ys) + \text{isum } f (\text{lset } lxs') + f x) =$ 
 $\text{isum } (\lambda y. \text{if } y = x \text{ then } \text{epred } (f y) \text{ else } f y) (\text{set } ys) +$ 
 $\text{isum } (\lambda y. \text{if } y = x \text{ then } \text{epred } (f y) \text{ else } f y) (\text{lset } lxs') + \text{epred } (f x) \rangle$ 
        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 hmap 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 = ldroprn n lxs⟩
      by (simp add: ldropWhile-eq-ldroprn hlen-enat ldroprn-eq)
      define lxs' where hlxs': ⟨lxs' = ltl (ldropWhile ((≠) x) lxs)⟩
      have hlxs'-eq: ⟨lxs' = ldroprn (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 lxs) x = lsum (lmap (λlxs. count-llist lxs x) lxs)⟩

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 lxs ∧ xa ≤ z ∧ enat xa < llength
(lnth lxs z) ∧ lnth (lnth lxs z) xa = x}) =
  ?LSUM (λi. enat z < llength lxs ∧ i ≤ z ∧ enat i < llength (lnth lxs z) ∧
lnth (lnth lxs z) i = x)⟩ for z
  by (rule card-LSUM) blast
have horiz: ⟨enat (card {i. i < z ∧ enat i < llength lxs ∧ enat z < llength (lnth
lxs i) ∧ lnth (lnth lxs i) z = x}) =
  ?LSUM (λi. i < z ∧ enat i < llength lxs ∧ enat z < llength (lnth lxs i) ∧
lnth (lnth lxs 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) lxs) =
  isum (λa. if enat a < llength lxs then
    isum (λb. if enat b < llength (lnth lxs a) ∧ lnth (lnth lxs 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 lxs i @ lhorizontal lxs i)
x) =
  LSUM lenats (λz.
    ?LSUM (λi. enat z < llength lxs ∧ i ≤ z ∧ enat i < llength (lnth lxs z)
∧ lnth (lnth lxs z) i = x) +
    ?LSUM (λi. i < z ∧ enat i < llength lxs ∧ enat z < llength (lnth lxs i) ∧
lnth (lnth lxs 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 (λz.
?isum (λi. enat z < llength lxss ∧ i ≤ z ∧ enat i < llength (lnth lxss z) ∧
lnth (lnth lxss z) i = x) +
?isum (λi. 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 (λz. ?isum (λi. enat z < llength lxss ∧ i ≤ z ∧ enat i < llength (lnth
lxss z) ∧ lnth (lnth lxss z) i = x)) UNIV +
isum (λz. ?isum (λi. 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 (λz. ?isum (λi. i < z ∧ enat i < llength lxss ∧ enat*
z < llength (lnth lxss i) ∧ lnth (lnth lxss i) z = x)) UNIV =
isum (λa. ?isum (λz. 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 (λi. count-list (lvertical lxss i @ lhorizontal*
lxss i) x) =
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 (*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*: $\langle \text{inj cmcount} \rangle$

unfolding *inj-on-def*

by *transfer* (*auto simp: eq-cmset-def*)

lemma *type-definition-cmset*: $\langle \text{type-definition cmcount (inv cmcount) } \{f. \text{countable } \{x. f x \neq 0\}\} \rangle$

by *standard* (*auto simp: inj-cmcount in-range-cmcount inv-f-f-inv-into-f countable-nonzero-cmcount*)

corec (*friend*) *linterleave* **where**

$\langle \text{linterleave } lxs \ lys = (\text{case } (lxs, lys) \text{ of}$

$(LCons \ x \ lxs', LCons \ y \ lys') \Rightarrow LCons \ x \ (LCons \ y \ (\text{linterleave } lxs' \ lys'))$

$| (LCons \ x \ lxs', LNil) \Rightarrow LCons \ x \ lxs'$

$| (LNil, LCons \ y \ lys') \Rightarrow LCons \ y \ lys'$

$| (LNil, LNil) \Rightarrow LNil \rangle$

simps-of-case *linterleave-simps*[*simp*]: *linterleave.code*[*unfolded prod.case*]

lemma *linterleave-LNil*[*simp*]:

$\langle \text{linterleave } LNil \ lys = lys \rangle$

$\langle \text{linterleave } lys \ LNil = lys \rangle$

by (*cases* *lys*; *simp*)⁺

lemma *linterleave-LCons1*[*simp*]:

$\langle \text{linterleave } (LCons \ x \ lxs) \ lys = LCons \ x \ (\text{linterleave } lys \ lxs) \rangle$

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*:

$\langle x \in \text{lset } (\text{linterleave } lxs \ lys) \Longrightarrow$

$x \in \text{lset } lxs \cup \text{lset } lys \rangle$

proof (*induct* $\langle \text{linterleave } lxs \ lys \rangle$ *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*:

$\langle x \in \text{lset } lxs \Longrightarrow x \in \text{lset } (\text{linterleave } lxs \ lys) \rangle$

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  $\langle \exists xs' x. xs = LCons x xs' \wedge (lfinite\text{-cong } X xs' \vee lfinite\ xs') \rangle$ 
proof (induct xs rule: lfinite-cong.induct)
  case (1 xs)
  from assms(2)[OF 1] show ?case by (auto intro: lfinite-cong.intros(3))
next
  case (2 xs)
  then show ?case by (auto elim: lfinite.cases intro: lfinite-cong.intros(2))
next
  case (3 xs y)
  then show ?case by (auto intro: lfinite-cong.intros)
qed
qed

```

```

inductive-cases lfinite-LNilE[elim!]:  $\langle lfinite\ LNil \rangle$ 
inductive-cases lfinite-LConsE[elim!]:  $\langle lfinite\ (LCons\ x\ xs) \rangle$ 

```

```

lemma lfinite-linterleaveL:  $\langle lfinite\ xs \implies lfinite\ (linterleave\ xs\ lys) \rangle$ 
proof (coinduction arbitrary: xs lys rule: lfinite-coinduct-upto)
  case (lfinite xs lys)
  then show ?case by (cases xs; cases lys; auto intro: lfinite-cong.intros)
qed

```

```

lemma lfinite-linterleaveR:  $\langle lfinite\ lys \implies lfinite\ (linterleave\ xs\ lys) \rangle$ 
proof (coinduction arbitrary: xs lys rule: lfinite-coinduct-upto)
  case (lfinite xs lys)
  then show ?case by (cases xs; cases lys; auto intro: lfinite-cong.intros)
qed

```

```

lemma lfinite-imp-not-lfinite:  $\langle lfinite\ xs \implies \neg lfinite\ xs \rangle$ 
by (induct xs rule: lfinite-induct) (auto simp: lnull-def neq-LNil-conv)

```

```

lemma not-lfinite-imp-lfinite:  $\langle \neg lfinite\ xs \implies lfinite\ xs \rangle$ 
proof (coinduction arbitrary: xs)
  case (lfinite xs)
  then show ?case by (cases xs) auto
qed

```

```

lemma lfinite-eq-not-lfinite:  $\langle lfinite\ xs \longleftrightarrow \neg lfinite\ xs \rangle$ 
using lfinite-imp-not-lfinite not-lfinite-imp-lfinite by blast
lemma lfinite-eq-llength:  $\langle lfinite\ xs \longleftrightarrow llength\ xs = \infty \rangle$ 
using lfinite-imp-not-lfinite llength-eq-infty-conv-lfinite not-lfinite-imp-lfinite
by blast

```

```

lemma llength-linterleave[simp]:  $\langle llength\ (linterleave\ xs\ lys) = llength\ xs + llength\ lys \rangle$ 
proof (cases  $\langle lfinite\ xs \rangle$ )
  case True
  then show ?thesis

```

```

proof (cases ⟨lfinite lys⟩)
  case True
  then show ?thesis
  by (metis ⟨lfinite lxs⟩ lfinite-eq-llength lfinite-linterleaveL plus-enat-simps(3))
next
  case False
  then show ?thesis
  by (metis ⟨lfinite lxs⟩ lfinite-eq-llength lfinite-linterleaveL plus-enat-simps(2))
qed
next
  case False
  then have hlxs: ⟨lfinite lxs⟩ by (simp add: lfinite-eq-not-lfinite)
  show ?thesis
  proof (cases ⟨lfinite lys⟩)
    case True
    then show ?thesis by (metis lfinite-eq-llength lfinite-linterleaveR plus-enat-simps(3))
  next
    case False
    then have hlys: ⟨lfinite lys⟩ by (simp add: lfinite-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 = ⟨ $\bigwedge$ lys. lfinite lys  $\implies$  llength (linterleave (ltl lxs) lys) =
llength (ltl lxs) + llength lys⟩
      from ⟨lfinite lys⟩ show ?case
      proof (induct lys rule: lfinite-induct)
        case LNil
        with ⟨ $\neg$  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 ⟨ $\neg$  lnull lxs⟩ obtain x xs' where lxs-eq: ⟨lxs = LCons x xs'⟩
          by (cases lxs) (auto simp: lnull-def)
          from ⟨ $\neg$  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*:

$\langle \text{lnth } (\text{linterleave } lxs \ lys) \ i \notin \text{lset } lys \implies i < \text{llength } (\text{linterleave } lxs \ lys) \implies \exists j < \min (\text{Suc } i) (\text{llength } lxs). \text{lnth } (\text{linterleave } lxs \ lys) \ i = \text{lnth } lxs \ j \rangle$

proof (*induct i arbitrary: lxs lys rule: less-induct*)

case (*less i lxs lys*)

show *?thesis*

proof (*cases i*)

case *0*

with *less.prem*s **show** *?thesis*

by (*cases lxs; cases lys*) (*auto simp: enat-0*)

next

case (*Suc j*)

with *less.prem*s *less.hyps* **show** *?thesis*

proof (*cases lxs*)

case *LNil*

then show *?thesis*

proof (*cases lys*)

case *LNil*

with *less.prem*s $\langle lxs = LNil \rangle$ **show** *?thesis* **by** (*auto simp: Suc-ile-eq*)

next

case (*LCons b lys'*)

with *less.prem*s $\langle lxs = LNil \rangle \langle i = \text{Suc } j \rangle$ **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.prem*s *less.hyps* $\langle lxs = LCons \ a \ lxs' \rangle$ **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* = $\langle lxs = LCons \ a \ lxs' \rangle$ **and** *lys-eq* = $\langle lys = LCons \ b \ lys' \rangle$

show *?thesis*

proof (*cases j*)

case *0*

with *less.prem*s *lxs-eq lys-eq* $\langle i = \text{Suc } j \rangle$ **show** *?thesis*

by (*auto simp: Suc-ile-eq in-lset-conv-lnth lnth-LCons'*)

next

case (*Suc m*)

with *less.prem*s *less.hyps lxs-eq lys-eq* $\langle i = \text{Suc } j \rangle$ **have**

IH: $\langle \text{lnth } (\text{linterleave } lxs' \ lys') \ m \notin \text{lset } lys' \implies$

$$\text{enat } m < \text{llength } (\text{linterleave } \text{xs}' \text{ ys}') \implies$$

$$\exists j'. \text{enat } j' < \min (\text{enat } (\text{Suc } m)) (\text{llength } \text{xs}') \wedge$$

$$\text{lnth } (\text{linterleave } \text{xs}' \text{ ys}') m = \text{lnth } \text{xs}' j'$$

by (*auto simp: Suc-ile-eq less-Suc-eq-le*)
from *less.premis xs-eq lys-eq* $\langle i = \text{Suc } j \rangle \langle j = \text{Suc } m \rangle$
have *notin*: $\langle \text{lnth } (\text{linterleave } \text{xs}' \text{ ys}') m \notin \text{lset } \text{lys}' \rangle$
by (*auto simp: Suc-ile-eq in-lset-conv-lnth lnth-LCons'*)
from *less.premis xs-eq lys-eq* $\langle i = \text{Suc } j \rangle \langle j = \text{Suc } m \rangle$
have *len*: $\langle \text{enat } m < \text{llength } (\text{linterleave } \text{xs}' \text{ ys}') \rangle$
by (*auto simp: Suc-ile-eq*)
from *IH[OF notin len]* **obtain** *j'* **where**
j'-bound: $\langle \text{enat } j' < \min (\text{enat } (\text{Suc } m)) (\text{llength } \text{xs}') \rangle$
j'-eq: $\langle \text{lnth } (\text{linterleave } \text{xs}' \text{ ys}') m = \text{lnth } \text{xs}' j' \rangle$
by *blast*
from *j'-bound j'-eq* $\langle i = \text{Suc } j \rangle \langle j = \text{Suc } m \rangle$ *xs-eq lys-eq less.premis*
show *?thesis*
by (*auto simp: Suc-ile-eq in-lset-conv-lnth lnth-LCons' intro!: exI[of -*
 $\langle \text{Suc } j' \rangle$
qed
qed
qed
qed
qed

lemma *ldropWhile-eq-ldropn*:
 $\langle \exists x \in \text{lset } \text{xs}. \neg P x \implies \exists n. \text{enat } n = \text{llength } (\text{ltakeWhile } P \text{ xs}) \wedge \text{ldropWhile } P \text{ xs} = \text{ldropn } n \text{ xs} \rangle$
by (*metis ldropWhile-eq-ldrop ldrop-enat lfinite-llength-enat lfinite-ltakeWhile*)

lemma *ltl-linterleave[simp]*: $\langle \neg \text{lnull } \text{xs} \implies \text{ltl } (\text{linterleave } \text{xs } \text{lys}) = \text{linterleave } \text{lys } (\text{ltl } \text{xs}) \rangle$
by (*cases xs*) *auto*

lemma *lfinite-ltl[simp]*: $\langle \text{lfinite } \text{xs} \implies \text{lfinite } (\text{ltl } \text{xs}) \rangle$
by (*cases xs*) *auto*

lemma *lfinite-ldropn[simp]*: $\langle \text{lfinite } \text{xs} \implies \text{lfinite } (\text{ldropn } n \text{ xs}) \rangle$
by (*induct n arbitrary: xs*) (*auto simp: ldropn-Suc split: llist.splits*)

lemma *lfinite-not-lnull*: $\langle \text{lfinite } \text{xs} \implies \neg \text{lnull } \text{xs} \rangle$
by (*cases xs*) *auto*

lemma *ldropn-linterleave*: $\langle \text{lfinite } \text{xs} \implies \text{lfinite } \text{lys} \implies \text{ldropn } n (\text{linterleave } \text{xs } \text{lys}) =$
 $(\text{if } \exists m. n = 2 * m \text{ then } \text{linterleave } (\text{ldropn } (n \text{ div } 2) \text{ xs}) (\text{ldropn } (n \text{ div } 2) \text{ lys})$
 else
 $\text{linterleave } (\text{ldropn } (n \text{ div } 2) \text{ lys}) (\text{ldropn } (\text{Suc } (n \text{ div } 2)) \text{ xs})) \rangle$
proof (*induct n arbitrary: xs lys*)
case *0*

```

then show ?case by simp
next
case (Suc n lxs lys)
from Suc.prem1 obtain a lxs' where lxs-eq: ⟨lxs = LCons a lxs'⟩ and lxf-lxs':
⟨lxfinite lxs'⟩
by (cases lxs) (auto dest: lxfinite-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.prem2] lxf-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.prem1 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 neg: ⟨∧ 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.premis 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
  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.premis 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*:
 $\langle \text{enat } (2 * m) = \text{llength } (\text{ltakeWhile } ((\neq) x) (\text{linterleave } lxs \ lys)) \implies$
 $x \notin \text{set } (\text{ltaken } m \ lxs) \wedge x \notin \text{set } (\text{ltaken } m \ lys) \rangle$
by (rule *llength-ltakeWhile-linterleave-ge*) *auto*

lemma *llength-ltakeWhile-linterleave-ge-Suc*:
 $\langle \text{lfinite } lxs \implies \text{lfinite } lys \implies$
 $\text{enat } (\text{Suc } (2 * m)) \leq \text{llength } (\text{ltakeWhile } ((\neq) x) (\text{linterleave } lxs \ lys)) \implies$
 $x \notin \text{set } (\text{ltaken } (\text{Suc } m) \ lxs) \wedge x \notin \text{set } (\text{ltaken } m \ lys) \rangle$
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.prem1* **obtain** *a lxs'* **where** *lxs-eq*: $\langle lxs = \text{LCons } a \ lxs' \rangle$
 by (cases *lxs*) (auto dest: *lfinite-not-lnull*)
 from *Suc.prem2* **obtain** *b lys'* **where** *lys-eq*: $\langle lys = \text{LCons } b \ lys' \rangle$
 by (cases *lys*) (auto dest: *lfinite-not-lnull*)
 from *Suc.prem3* *lxs-eq lys-eq* **have** *linf-tails*: $\langle \text{lfinite } lxs' \ \langle \text{lfinite } lys' \rangle$
 by *auto*
 from *Suc.prem4* *lxs-eq lys-eq* **have** *prems'*: $\langle x \neq a \ \langle x \neq b \rangle$
 $\langle \text{enat } (\text{Suc } (2 * m)) \leq \text{llength } (\text{ltakeWhile } ((\neq) x) (\text{linterleave } lxs' \ lys')) \rangle$
 by (auto simp: *Suc-ile-eq split: if-splits*)
 from *Suc.hyps*[OF *linf-tails*(1) *linf-tails*(2) *prems'*(3)]
have *IH*: $\langle x \notin \text{set } (\text{ltaken } (\text{Suc } m) \ lxs') \wedge x \notin \text{set } (\text{ltaken } m \ lys') \rangle$.
 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*:
 $\langle \text{lfinite } lxs \implies \text{lfinite } lys \implies$
 $\text{enat } (\text{Suc } (2 * m)) = \text{llength } (\text{ltakeWhile } ((\neq) x) (\text{linterleave } lxs \ lys)) \implies$
 $x \notin \text{set } (\text{ltaken } (\text{Suc } m) \ lxs) \wedge x \notin \text{set } (\text{ltaken } m \ lys) \rangle$
by (rule *llength-ltakeWhile-linterleave-ge-Suc*) *auto*

lemma *count-llist-linterleave*:
 $\langle \text{count-llist } (\text{linterleave } lxs \ lys) \ x = \text{count-llist } lxs \ x + \text{count-llist } lys \ x \rangle$
proof ((cases $\langle \text{lfinite } lxs \rangle$; cases $\langle \text{lfinite } lys \rangle$), 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-llnull 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-llnull 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 lxs) (ltl (ldropn m lxs)))) x)⟩
  proof -

```

```

have heq: ⟨ldropWhile ((≠) x) (linterleave lxs lys) =
```

$$LCons\ x\ (linterleave\ (ldropn\ m\ lys)\ (ltl\ (ldropn\ m\ lxs)))\rangle$$

```

by (subst h-drop2, subst h-lxs-eq, simp)
have h-tl: ⟨ltl (ldropWhile ((≠) x) (linterleave lxs lys)) =
```

$$linterleave\ (ldropn\ m\ lys)\ (ltl\ (ldropn\ m\ lxs))\rangle$$

```

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 +
```

$$count-llist\ lys\ x = 0)\rangle$$

```

by (simp add: h-inter-cnt h-lxs-cnt)
next
show ⟨count-llist (linterleave lxs lys) x ≠ 0 → count-llist lxs x + count-llist
```

$$lys\ x \neq 0 \rightarrow$$

```

(∃ lxs' lys'. epred (count-llist (linterleave lxs lys) x) = count-llist (linterleave
```

$$lxs'\ lys')\ x \wedge$$

```

epred (count-llist lxs x + count-llist lys x) = count-llist lxs' x
```

$$+ count-llist\ lys'\ x \wedge$$

```

¬ lfinite lxs' ∧ ¬ lfinite lys'⟩ ∨
```

```

epred (count-llist (linterleave lxs lys) x) = epred (count-llist lxs x +
```

$$count-llist\ lys\ x)\rangle$$

```

proof (intro impI disjI1)
show ⟨∃ lxs' lys'. epred (count-llist (linterleave lxs lys) x) = count-llist
```

$$(linterleave\ lxs'\ lys')\ x \wedge$$

```

epred (count-llist lxs x + count-llist lys x) = count-llist lxs' x
```

$$+ count-llist\ lys'\ x \wedge$$

```

¬ 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
```

$$(ldropn\ m\ lys)\ (ltl\ (ldropn\ m\ lxs)))\ x\rangle$$

```

by (simp add: h-inter-cnt)
show ⟨epred (count-llist lxs x + count-llist lys x) = count-llist (ldropn
```

$$m\ lys)\ x + count-llist\ (ltl\ (ldropn\ m\ lxs))\ x\rangle$$

```

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-llnull 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-llnull 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: $\langle \text{count-llist (linterleave (ltaken (Suc (n div 2)) lxs)) } x = 0 \rangle$
by (simp del: ltaken.simps add: count-list-0-iff h α 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: $\langle \text{count-llist (linterleave lxs lys) } x =$
 $e\text{Suc (count-llist (linterleave (ldropn (Suc (n div 2)) lxs) (ltl (ldropn (n$
div 2) lys))) } x \rangle
proof –
have heq: $\langle \text{ldropWhile } ((\neq) x) \text{ (linterleave lxs lys) } =$
 $L\text{Cons } x \text{ (linterleave (ldropn (Suc (n div 2)) lxs) (ltl (ldropn (n div 2)$
lys)))} \rangle
by (subst h-drop2, subst h-lys-eq, simp)
have h-tl: $\langle \text{ltl (ldropWhile } ((\neq) x) \text{ (linterleave lxs lys)) } =$
 $\text{linterleave (ldropn (Suc (n div 2)) lxs) (ltl (ldropn (n div 2) lys))} \rangle$
by (simp add: heq)
from count-llist-ctr(2)[OF hin] h-tl **show** ?thesis
by simp
qed
show ?thesis
proof (rule conjI)
show $\langle (\text{count-llist (linterleave lxs lys) } x = 0) = (\text{count-llist lxs } x +$
count-llist lys $x = 0) \rangle$
by (simp add: h-inter-cnt h-lys-cnt)
next
show $\langle \text{count-llist (linterleave lxs lys) } x \neq 0 \longrightarrow \text{count-llist lxs } x + \text{count-llist}$
lys $x \neq 0 \longrightarrow$
 $(\exists \text{ lxs' lys'. } e\text{pred (count-llist (linterleave lxs lys) } x) = \text{count-llist (linterleave}$
lxs' lys') $x \wedge$
 $e\text{pred (count-llist lxs } x + \text{count-llist lys } x) = \text{count-llist lxs' } x$
 $+ \text{count-llist lys' } x \wedge$
 $\neg \text{lfinite lxs' } \wedge \neg \text{lfinite lys'} \vee$
 $e\text{pred (count-llist (linterleave lxs lys) } x) = e\text{pred (count-llist lxs } x +$
count-llist lys $x) \rangle$
proof (intro impI disjI1)
show $\langle \exists \text{ lxs' lys'. } e\text{pred (count-llist (linterleave lxs lys) } x) = \text{count-llist}$
(linterleave lxs' lys') $x \wedge$
 $e\text{pred (count-llist lxs } x + \text{count-llist lys } x) = \text{count-llist lxs' } x$
 $+ \text{count-llist lys' } x \wedge$
 $\neg \text{lfinite lxs' } \wedge \neg \text{lfinite lys'} \rangle$
proof (rule exI[of - $\langle \text{ldropn (Suc (n div 2)) lxs} \rangle$],
rule exI[of - $\langle \text{ltl (ldropn (n div 2) lys)} \rangle$], intro conjI)
show $\langle e\text{pred (count-llist (linterleave lxs lys) } x) = \text{count-llist (linterleave}$
(ldropn (Suc (n div 2)) lxs) (ltl (ldropn (n div 2) lys))) $x \rangle$
by (simp add: h-inter-cnt)
show $\langle e\text{pred (count-llist lxs } x + \text{count-llist lys } x) = \text{count-llist (ldropn}$
(Suc (n div 2)) lxs) $x + \text{count-llist (ltl (ldropn (n div 2) lys)) } x \rangle$

```

      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
qed

```

lemma *lfinite-linterleave[simp]*: $\langle \text{lfinite} (\text{linterleave } lxs \ lys) \longleftrightarrow \text{lfinite } lxs \wedge \text{lfinite } lys \rangle$
by (*metis enat-add-left-cancel llength-eq-infty-conv-lfinite llength-linterleave plus-enat-simps(3)*)

lift-definition *cmadd* :: $\langle 'a \text{ cmset} \Rightarrow 'a \text{ cmset} \Rightarrow 'a \text{ cmset} \rangle$ **is** *linterleave*
by (*auto simp: eq-cmset-def count-llist-linterleave*)

lemma *cmcount-cmadd[simp]*: $\langle \text{cmcount} (\text{cmadd } M \ N) \ x = \text{cmcount } M \ x + \text{cmcount } N \ x \rangle$
by *transfer (auto simp: count-llist-linterleave)*

lemma *count-llist-lmap*: $\langle \text{count-llist} (\text{lmap } f \ lxs) \ b = \text{isum} (\text{count-llist } lxs) \ \{a. f \ a = b\} \rangle$

proof –

have *h-ind*: $\langle \bigwedge i. \text{isum} (\lambda a. \text{if } \text{enat } i < \text{llength } lxs \wedge \text{lth } lxs \ i = a \text{ then } 1 \text{ else } 0) \ \{a. f \ a = b\} =$

$(\text{if } \text{enat } i < \text{llength } lxs \wedge f (\text{lth } lxs \ i) = b \text{ then } 1 \text{ else } 0) \rangle$

proof –

fix *i*

show $\langle \text{isum} (\lambda a. \text{if } \text{enat } i < \text{llength } lxs \wedge \text{lth } lxs \ i = a \text{ then } 1 \text{ else } 0) \ \{a. f \ a = b\} =$

$(\text{if } \text{enat } i < \text{llength } lxs \wedge f (\text{lth } lxs \ i) = b \text{ then } 1 \text{ else } 0) \rangle$

proof (*cases* $\langle \text{enat } i < \text{llength } lxs \wedge f (\text{lth } lxs \ i) = b \rangle$)

case *True*

then have *himem*: $\langle \text{lth } lxs \ i \in \{a. f \ a = b\} \rangle$ **by** *simp*

have $\langle \text{isum} (\lambda a. \text{if } \text{lth } lxs \ i = a \text{ then } 1 \text{ else } 0) \ \{a. f \ a = b\} =$

$\text{isum} (\lambda a. \text{if } \text{lth } lxs \ i = a \text{ then } 1 \text{ else } 0) (\{a. f \ a = b\} - \{\text{lth } lxs \ i\}) + 1 \rangle$

by (*subst insert-Diff[OF himem, symmetric], subst isum-insert*) (*auto simp: isum-eq-0-iff*)

also have $\langle \text{isum} (\lambda a. \text{if } \text{lth } lxs \ i = a \text{ then } 1 \text{ else } 0) (\{a. f \ a = b\} - \{\text{lth } lxs \ i\}) = 0 \rangle$

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 (λi. 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 (λi. 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 (λa. isum (λi. 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 (λi. isum (λa. 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 (λi. if enat i < llength lxs ∧ f (lnth lxs i) = b then 1
else 0) UNIV⟩
  proof (rule isum-cong, simp)
    fix x
    show ⟨isum (λa. if enat x < llength lxs ∧ lnth lxs x = a then 1 else 0) {a. f
a = b} =
(isum (λi. if enat i < llength lxs ∧ f (lnth lxs i) = 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 . $a \in A$) in $LCons$ C (*mk-chain* ($A - \{a\}$) (insert a C)))⟩

lemma *in-lset-mk-chainD*: ⟨ $X \in lset$ (*mk-chain* A C) \implies infinite $A \implies \exists B \subseteq A$. finite $B \wedge X = C \cup B$ ⟩

proof (induct X ⟨*mk-chain* A C ⟩ arbitrary: A C rule: *lset.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 . $a \in A$ ⟩

have *a-in-A*: ⟨? $a \in A$ ⟩ using $LCons2.prem$ s by (auto simp: *some-in-eq*)

have *lxs-eq*: ⟨ $lxs = mk-chain$ ($A - \{?a\}$) (insert ? a C)⟩

using $LCons2.hyps(1,3)$ by (metis *lset.inject mk-chain.code*)

have *inf'*: ⟨infinite ($A - \{?a\}$)⟩ using $LCons2.prem$ s by simp

obtain B where *B-sub*: ⟨ $B \subseteq A - \{?a\}$ ⟩ and *B-fin*: ⟨finite B ⟩ and *X-eq*: ⟨ $X = insert$? a $C \cup B$ ⟩

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 $y \implies chain$ R ($LCons$ y lxs) $\implies chain$ R ($LCons$ x ($LCons$ y lxs))⟩

inductive-cases *chain-LConsE*: ⟨*chain* R ($LCons$ x lxs)⟩

lemma *chain-mk-chain*: ⟨infinite $A \implies A \cap C = \{\} \implies chain$ (\subseteq) (*mk-chain* A C)⟩

proof (coinduction arbitrary: A C)

case (*chain* A C)

let ? a = ⟨SOME a . $a \in A$ ⟩

have *ha*: ⟨? $a \in A$ ⟩

using ⟨infinite A ⟩ *infinite-imp-nonempty* by (metis *some-in-eq*)

have *ha-notin*: ⟨? $a \notin C$ ⟩

using ⟨ $A \cap C = \{\}$ ⟩ *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  $\implies i < j \implies \text{enat } j < \text{llength } lxs \implies R (\text{lnth } lxs \ i) (\text{lnth } lxs \ j)$ ⟩
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* $\implies \text{chain } R \ lxs \implies \text{Complete-Partial-Order.chain}$ (*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* (\subset) = (\subseteq)⟩

by *auto*

lemma *sum-strict-mono2-enat*:

fixes *f* :: ⟨*'a* $\Rightarrow \text{enat}$ ⟩

assumes ⟨*finite B*⟩ ⟨*A* $\subseteq B$ ⟩ ⟨*b* $\in B - A$ ⟩ ⟨*f b* > 0 ⟩ ⟨ $\infty \notin f \ 'B$ ⟩

shows ⟨*sum f A* $< \text{sum } f \ B$ ⟩

proof –

have ⟨*B - A* $\neq \{\}$ ⟩

using $assms(3)$ **by** *blast*
have $\langle sum\ f\ (B-A) > 0 \rangle$
by (*rule sum-pos2*) (*use assms in auto*)
moreover have $\langle sum\ f\ B = sum\ f\ (B-A) + sum\ f\ A \rangle$
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 $\langle linfinite\ lxs \rangle$ $\langle chain\ R\ lxs \rangle$ $\langle irreflp\ R \rangle$ $\langle transp\ R \rangle$
shows $\langle infinite\ (lset\ lxs) \rangle$

proof –

have *inj*: $\langle inj\ on\ (lnth\ lxs)\ UNIV \rangle$

proof (*rule inj-onI*)

fix $m\ n :: nat$ **assume** *eq*: $\langle lnth\ lxs\ m = lnth\ lxs\ n \rangle$

show $\langle m = n \rangle$

proof (*rule antisym*)

show $\langle m \leq n \rangle$

proof (*rule ccontr*)

assume $\langle \neg m \leq n \rangle$

then have $\langle n < m \rangle$ **by** *simp*

with *assms* **have** $\langle R\ (lnth\ lxs\ n)\ (lnth\ lxs\ m) \rangle$

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 $\langle n \leq m \rangle$

proof (*rule ccontr*)

assume $\langle \neg n \leq m \rangle$

then have $\langle m < n \rangle$ **by** *simp*

with *assms* **have** $\langle R\ (lnth\ lxs\ m)\ (lnth\ lxs\ n) \rangle$

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 $\langle infinite\ (lnth\ lxs\ UNIV) \rangle$

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*: $\langle isum\ f\ A = \infty \iff infinite\ \{a \in A. f\ a \neq 0\} \vee (\exists a \in A. f\ a = \infty) \rangle$

proof (*rule iffI*)

```

assume  $h$ :  $\langle isum\ f\ A = \infty \rangle$ 
then have  $h2$ :  $\langle \infty \in sum\ f\ ' \{B\ | B. B \subseteq A \wedge finite\ B\} \vee infinite\ (sum\ f\ ' \{B\ | B. B \subseteq A \wedge finite\ B\}) \rangle$ 
  unfolding  $isum-def\ Sup-enat-eq-inf$  by  $simp$ 
show  $\langle infinite\ \{a \in A. f\ a \neq 0\} \vee (\exists a \in A. f\ a = \infty) \rangle$ 
proof ( $rule\ ccontr$ )
  assume  $hc$ :  $\langle \neg (infinite\ \{a \in A. f\ a \neq 0\} \vee (\exists a \in A. f\ a = \infty)) \rangle$ 
  then have  $hfin$ :  $\langle finite\ \{a \in A. f\ a \neq 0\} \rangle$  and  $hno-inf$ :  $\langle \forall a \in A. f\ a \neq \infty \rangle$  by
 $auto$ 
  have  $surj$ :  $\langle sum\ f\ ' \{B\ | B. B \subseteq A \wedge finite\ B\} \subseteq sum\ f\ ' Pow\ \{a \in A. f\ a \neq 0\} \rangle$ 
proof
  fix  $s$  assume  $\langle s \in sum\ f\ ' \{B\ | B. B \subseteq A \wedge finite\ B\} \rangle$ 
  then obtain  $B$  where  $hB$ :  $\langle B \subseteq A \rangle$   $\langle finite\ B \rangle$  and  $seq$ :  $\langle s = sum\ f\ B \rangle$  by
 $auto$ 
  have  $eq$ :  $\langle sum\ f\ B = sum\ f\ \{a \in B. f\ a \neq 0\} \rangle$ 
  by ( $rule\ sum.mono-neutral-right$ ) ( $use\ hB$  in  $auto$ )
  show  $\langle s \in sum\ f\ ' Pow\ \{a \in A. f\ a \neq 0\} \rangle$ 
  using  $hB\ eq\ seq$  by ( $auto\ simp: Pow-def$ )
qed
have  $\langle finite\ (sum\ f\ ' \{B\ | B. B \subseteq A \wedge finite\ B\}) \rangle$ 
by ( $rule\ finite-subset[OF\ surj]$ ) ( $rule\ finite-imageI, rule\ finite-Pow-iff[THEN\ iffD2, OF\ hfin]$ )
moreover have  $\langle \infty \notin sum\ f\ ' \{B\ | B. B \subseteq A \wedge finite\ B\} \rangle$ 
proof
  assume  $\langle \infty \in sum\ f\ ' \{B\ | B. B \subseteq A \wedge finite\ B\} \rangle$ 
  then obtain  $B$  where  $hB$ :  $\langle B \subseteq A \rangle$   $\langle finite\ B \rangle$  and  $seq$ :  $\langle sum\ f\ B = \infty \rangle$  by
 $auto$ 
  then obtain  $a$  where  $ha$ :  $\langle a \in B \rangle$  and  $hfa$ :  $\langle f\ a = \infty \rangle$ 
  using  $inf-eq-sum[of\ B\ f]$  by  $auto$ 
  have  $\langle a \in A \rangle$  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  $h$ :  $\langle infinite\ \{a \in A. f\ a \neq 0\} \vee (\exists a \in A. f\ a = \infty) \rangle$ 
show  $\langle isum\ f\ A = \infty \rangle$ 
proof ( $cases\ \langle \exists a \in A. f\ a = \infty \rangle$ )
  case  $True$ 
  then obtain  $a$  where  $ha$ :  $\langle a \in A \rangle$  and  $hfa$ :  $\langle f\ a = \infty \rangle$  by  $auto$ 
  have  $\langle \infty \in sum\ f\ ' \{B\ | B. B \subseteq A \wedge finite\ B\} \rangle$ 
  by ( $rule\ image-eqI[where\ x = \langle \{a\} \rangle]$ ) ( $simp-all\ add: ha\ hfa$ )
  then show  $?thesis$  unfolding  $isum-def\ Sup-enat-eq-inf$  by  $simp$ 
next
  case  $False$ 
  with  $h$  have  $hinf$ :  $\langle infinite\ \{a \in A. f\ a \neq 0\} \rangle$  and  $hno-inf$ :  $\langle \infty \notin f\ ' A \rangle$  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
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:  $\langle \text{linfinite } M \rangle$  by (simp add: linfinite-eq-not-lfinite)
  have count-inf:  $\langle \text{count-llist } (\text{lmap } (\lambda-. \text{undefined}) M) \text{ undefined} = \infty \rangle$ 
  using False by (simp add: count-llist-lmap-const llength-eq-infty-conv-lfinite)
  then have  $\langle (\exists x. \text{count-llist } M x = \infty) \vee \text{infinite } \{a \in \text{UNIV}. \text{count-llist } M a \neq 0\} \rangle$ 
  unfolding count-llist-lmap isum-eq-inf by fastforce
  with linf show ?thesis by simp
qed
qed

lemma cmset-alt:  $\langle \text{cmset } M = \{a. \text{cmcount } M a \neq 0\} \rangle$ 
proof (transfer)
  fix M ::  $\langle 'a \text{ llist} \rangle$ 
  show  $\langle (\bigcap (mx :: (\text{unit} + 'a) \text{ llist}) \in \text{Collect } (\text{eq-cmset } (\text{lmap } \text{Inr } M)). \bigcup (\text{Basic-BNFs.setr } 'lset mx)) = \{a. \text{count-llist } M a \neq 0\} \rangle$ 
  proof -
    { fix x
      assume  $\langle \forall xa. (\forall x. \text{count-llist } (\text{lmap } \text{Inr } M) x = \text{count-llist } xa x) \longrightarrow (\exists xa \in \text{lset } xa. x \in \text{Basic-BNFs.setr } xa) \rangle$ 
      then have  $\langle x \in \text{lset } M \rangle$  by force
    }
    moreover
    { fix x ::  $'a$  and lxs ::  $\langle (\text{unit} + 'a) \text{ llist} \rangle$ 
      assume  $\langle x \in \text{lset } M \rangle$  and  $\langle \forall x. \text{count-llist } (\text{lmap } \text{Inr } M) x = \text{count-llist } lxs x \rangle$ 
      then have  $\langle \exists z \in \text{lset } lxs. x \in \text{Basic-BNFs.setr } z \rangle$ 
      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 ::  $\langle 'a \Rightarrow 'a \text{ cmset} \rangle$  is repeat .

lemma cmfinite-cmconst[simp]:  $\langle \text{cmfinite } (\text{cmconst } a) \rangle$ 
  by transfer (auto simp: linfinite-eq-not-lfinite)

lemma cmfinite-cmimage:  $\langle \text{cmfinite } (\text{cmimage } f M) \longleftrightarrow \text{cmfinite } M \rangle$ 
  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 cmfinite.transfer[transfer-rule]

end

locale cmset-as-typedef
begin

setup-lifting type-definition-cmset

lemma cmempty-transfer[transfer-rule]:  $\langle \text{pcr-cmset } R (\lambda-. 0) \text{ cmempty} \rangle$ 
  by (auto simp: pcr-cmset-def cr-cmset-def rel-fun-def relcompp-apply)

lemma cmadd-transfer[transfer-rule]:  $\langle \text{rel-fun } (\text{pcr-cmset } R) (\text{rel-fun } (\text{pcr-cmset } R) (\text{pcr-cmset } R)) (\lambda M N x. M x + N x) \text{ cmadd} \rangle$ 
  by (auto simp: pcr-cmset-def cr-cmset-def rel-fun-def relcompp-apply)

lemma cmimage-transfer[transfer-rule]:  $\langle \text{rel-fun } (=) (\text{rel-fun } (\text{pcr-cmset } (=)) (\text{pcr-cmset } (=))) (\lambda f M b. \text{isum } M \{a. f a = b\}) \text{ cmimage} \rangle$ 
  by (auto simp: pcr-cmset-def cr-cmset-def rel-fun-def relcompp-apply intro: isum-cong)

lemma cmset-transfer[transfer-rule]:  $\langle \text{rel-fun } (\text{pcr-cmset } (=)) (=) (\lambda M. \{a. M a \neq 0\}) \text{ cmset} \rangle$ 
  by (auto simp: pcr-cmset-def cr-cmset-def rel-fun-def cmset-alt)

lemma cmfinite-transfer[transfer-rule]:  $\langle \text{rel-fun } (\text{pcr-cmset } (=)) (=) (\lambda M. (\exists x. M x = \infty) \vee \text{infinite } \{x. M x \neq 0\}) \text{ cmfinite} \rangle$ 
  by (auto simp: pcr-cmset-def cr-cmset-def rel-fun-def cmfinite-alt)

lift-definition cmreplace ::  $\langle 'a \text{ cmset} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a \text{ cmset} \rangle$  is
   $\langle \lambda f a b. f(a := \text{epred } (f a), b := \text{eSuc } (f b)) \rangle$ 
  by (erule countable-subset[rotated, OF countable-insert]) auto

lemma cmfinite-cmreplace:  $\langle \text{cmfinite } M \Longrightarrow \text{cmfinite } (\text{cmreplace } M a b) \rangle$ 
proof (transfer fixing: a b)
  fix M ::  $\langle 'a \Rightarrow \text{enat} \rangle$ 
  assume M:  $\langle \text{countable } \{x. M x \neq 0\} \rangle \langle (\exists x. M x = \infty) \vee \text{infinite } \{x. M x \neq 0\} \rangle$ 
  show  $\langle (\exists x. (M(a := \text{epred } (M a), b := \text{eSuc } (M b))) x = \infty) \vee \text{infinite } \{x. (M(a := \text{epred } (M a), b := \text{eSuc } (M b))) x \neq 0\} \rangle$ 
  proof (cases  $\langle \exists x. M x = \infty \rangle$ )
    case True
    then obtain x where  $\langle M x = \infty \rangle$  ..
    show ?thesis
    by (metis  $\langle M x = \infty \rangle$  eSuc-infinity epred-Infty fun-upd-other fun-upd-same)
  next
  case False

```

```

with  $M$  have  $\langle \text{infinite } \{x. M\ x \neq 0\} \rangle$  by blast
then have  $\langle \text{infinite } \{x. (M(a := \text{epred } (M\ a), b := \text{eSuc } (M\ b)))\ x \neq 0\} \rangle$ 
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  $'a\ \text{cinfmset} = \langle \{M :: 'a\ \text{cmset}. \text{cminfinite } M\} \rangle$ 
by (auto intro!: exI[of - <cmconst ->])

```

```

setup-lifting type-definition-cinfmset

```

```

lift-bnf (no-warn-wits)  $'a\ \text{cinfmset}$ 
for map: cinfmimage rel: cinfmrel
by (auto simp: cminfinite-cmimage)

```

```

lift-definition  $\text{cinfmcount} :: \langle 'a\ \text{cinfmset} \Rightarrow 'a \Rightarrow \text{enat} \rangle$  is cmcount .

```

```

context begin

```

```

interpretation cmset-as-typedef .

```

```

lift-definition  $\text{cinfmreplace} :: \langle 'a\ \text{cinfmset} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a\ \text{cinfmset} \rangle$  is cmreplace
by (rule cminfinite-cmreplace)

```

```

lemma set-cinfmset-alt:  $\langle \text{set-cinfmset } xs = \{a. \text{cinfmcount } xs\ a \neq 0\} \rangle$ 
unfolding set-cinfmset-def cinfmcount-def cmset-alt
by auto

```

```

lemma set-cinfset-cinfmreplace:

```

```

 $\langle \text{set-cinfmset } (\text{cinfmreplace } M\ x\ y) \subseteq \text{set-cinfmset } M \cup \{y\} \rangle$ 

```

```

including cmset.lifting

```

```

proof transfer

```

```

fix  $M :: \langle 'a\ \text{cmset} \rangle$  and  $x\ y$ 

```

```

assume  $\langle \text{cminfinite } M \rangle$ 

```

```

show  $\langle \text{cmset } (\text{cmreplace } M\ x\ y) \subseteq \text{cmset } M \cup \{y\} \rangle$ 

```

```

by transfer auto

```

qed
end

lemma *inj-cinfmtcount*: $\langle \text{inj cinfmtcount} \rangle$
unfolding *inj-on-def cinfmtcount-def map-fun-def o-apply id-apply*
by (*simp add: Rep-cinfmtset-inject inj-cmcount inj-eq*)

lemma *in-range-cinfmtcount*:
 $\langle \text{countable } \{x. f x \neq 0\} \implies (\exists x. f x = \infty) \vee \text{infinite } \{x. f x \neq 0\} \implies f \in \text{range cinfmtcount} \rangle$
by (*drule in-range-cmcount*) (*smt (verit, best) Collect-cong Rep-cinfmtset-cases UNIV-I cinfmtcount.rep-eq cminfinite-alt image-iff mem-Collect-eq*)

lemma *cinfmtcount-cinfmtimage[simp]*:
 $\langle \text{cinfmtcount } (\text{cinfmtimage } f M) b = \text{isum } (\text{cinfmtcount } M) \{a. f a = b\} \rangle$
by *transfer (rule cmcount-cmimage)*

lemma *countable-nonzero-cinfmtcount*: $\langle \text{countable } \{x. \text{cinfmtcount } M x \neq 0\} \rangle$
by *transfer (rule countable-nonzero-cmcount)*

lemma *infinite-nonzero-cinfmtcount*: $\langle (\exists x. \text{cinfmtcount } M x = \infty) \vee \text{infinite } \{x. \text{cinfmtcount } M x \neq 0\} \rangle$
by *transfer (simp add: cminfinite-alt)*

lemma *type-definition-cinfmtset*: $\langle \text{type-definition cinfmtcount } (\text{inv cinfmtcount}) \{f. \text{countable } \{x. f x \neq 0\} \wedge ((\exists x. f x = \infty) \vee \text{infinite } \{x. f x \neq 0\})\} \rangle$
by *standard (auto simp: inj-cinfmtcount in-range-cinfmtcount inv-f-f f-inv-into-f countable-nonzero-cinfmtcount infinite-nonzero-cinfmtcount)*

lifting-update *cinfmtset.lifting*
lifting-forget *cinfmtset.lifting*

definition *get-cinfmtset* :: $\langle 'a \text{ cinfmtset} \Rightarrow \text{nat} \Rightarrow 'a \rangle$ **where**
 $\langle \text{get-cinfmtset } M i = \text{lth } (\text{rep-cmset } (\text{Rep-cinfmtset } M)) i \rangle$

context includes *cinfmtset.lifting* **begin**

interpretation *cmset-as-Quotient* .

lemma *get-cinfmtset-inject*:
 $\langle \forall i. \text{get-cinfmtset } M i = \text{get-cinfmtset } N i \implies M = N \rangle$
unfolding *get-cinfmtset-def*

proof *transfer*

fix $M N :: \langle 'a \text{ cmset} \rangle$

assume $\langle \text{cminfinite } M \rangle \langle \text{cminfinite } N \rangle$

$\langle \forall i. \text{lth } (\text{rep-cmset } M) i = \text{lth } (\text{rep-cmset } N) i \rangle$

then have $\langle \text{rep-cmset } M = \text{rep-cmset } N \rangle$

by (*simp add: cminfinite.rep-eq linfinite-eq-llength lth-equalityI*)

then show $\langle M = N \rangle$

by (*metis UNIV-I cmcount.rep-eq inj-cmcount inj-on-def*)

qed

end

end