Formalization of Countable Multisets

Mathias Schack Rabing Dmitriy Traytel

June 14, 2026

Abstract

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

Contents

1 Infinite Sums over Extended Natural Numbers

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

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

1.1 Preliminaries

lemma *enat-pm-iff*:
$\bigwedge a\ b\ c.\ b \leq c \Longrightarrow (a::enat) + b \leq c \longleftrightarrow a \leq c - b$
$\bigwedge a\ b\ c.\ a \leq c \Longrightarrow (a::enat) + b \leq c \longleftrightarrow b \leq c - a$
$\bigwedge a\ b\ c.\ a \leq b \Longrightarrow c \leq b \Longrightarrow (a::enat) \leq b - c \longleftrightarrow c \leq b - a$
⟨*proof*⟩

lemma *disjoint-finite-aux*:
$\forall i{\in}I.\ \forall j{\in}I.\ i \neq j \longrightarrow A\ i \cap A\ j = \{\} \Longrightarrow B \subseteq \bigcup\ (A\ `\ I) \Longrightarrow$ *finite* $B \Longrightarrow$ *finite* $\{i \in I.\ B \cap A\ i \neq \{\}\}$
⟨*proof*⟩

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

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

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

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

lemma *sum-three*[*simp*]: $a1 \neq a2 \Longrightarrow a1 \neq a3 \Longrightarrow a2 \neq a3 \Longrightarrow$
sum $f\ \{a1,a2,a3\} = f\ a1 + f\ a2 + f\ a3$
⟨*proof*⟩

lemma *Sup-leq*:
$A \neq \{\} \Longrightarrow \forall a{\in}A.\ \exists b{\in}B.\ (a::enat) \leq b \Longrightarrow Sup\ A \leq Sup\ B$
⟨*proof*⟩

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

⟨*proof*⟩

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*⟩

lemma *Sup-image-cong*:
$A \neq \{\} \vee B \neq \{\} \Longrightarrow \forall a{\in}A.\; \exists b{\in}B.\; (f\ a{::}enat) \leq g\ b \Longrightarrow \forall b{\in}B.\; \exists a{\in}A.\; (g\ b{::}enat) \leq f\ a \Longrightarrow$
$Sup\ (f\ `\ A) = Sup\ (g\ `\ B)$
⟨*proof*⟩

lemma *Sup-congL*:
$A \neq \{\} \Longrightarrow \forall a{\in}A.\; \exists b{\in}B.\; (a{::}enat) \leq b \Longrightarrow \forall b{\in}B.\; b \leq Sup\ A \Longrightarrow Sup\ A = Sup\ B$
⟨*proof*⟩

lemma *Sup-image-congL*:
$A \neq \{\} \Longrightarrow \forall a{\in}A.\; \exists b{\in}B.\; (f\ a{::}enat) \leq g\ b \Longrightarrow \forall b{\in}B.\; g\ b \leq Sup\ (f\ `\ A) \Longrightarrow Sup\ (f\ `\ A) = Sup\ (g\ `\ B)$
⟨*proof*⟩

lemma *Sup-congR*:
$B \neq \{\} \Longrightarrow \forall a{\in}A.\; a \leq Sup\ B \Longrightarrow \forall b{\in}B.\; \exists a{\in}A.\; (b{::}enat) \leq a \Longrightarrow Sup\ A = Sup\ B$
⟨*proof*⟩

lemma *Sup-image-congR*:
$B \neq \{\} \Longrightarrow \forall a{\in}A.\; f\ a \leq Sup\ (g\ `\ B) \Longrightarrow \forall b{\in}B.\; \exists a{\in}A.\; (g\ b{::}enat) \leq f\ a \Longrightarrow Sup\ (f\ `\ A) = Sup\ (g\ `\ B)$
⟨*proof*⟩

lemma *Sup-eq-0-iff*: $Sup\ (A :: enat\ set) = 0 \longleftrightarrow (\forall a{\in}A.\; a = 0)$
⟨*proof*⟩

lemma *plus-Sup-commute*:
assumes *f1*: $\{f1\ b1 \mid b1.\ \varphi 1\ b1\} \neq \{\}$ **and**
f2: $\{f2\ b2 \mid b2.\ \varphi 2\ b2\} \neq \{\}$
shows
$Sup\ \{(f1\ b1{::}enat) \mid b1\ .\ \varphi 1\ b1\} + Sup\ \{f2\ b2 \mid b2\ .\ \varphi 2\ b2\} =$
$Sup\ \{f1\ b1 + f2\ b2 \mid b1\ b2.\ \varphi 1\ b1 \wedge \varphi 2\ b2\}$ (**is** *?L1 + ?L2 = ?R*)
⟨*proof*⟩

lemma *plus-Sup-commute′*:
assumes *f1*: $A1 \neq \{\}$ **and** *f2*: $A2 \neq \{\}$
shows $Sup\ A1 + Sup\ A2 = Sup\ \{(a1{::}enat) + a2 \mid a1\ a2.\ a1 \in A1 \wedge a2 \in$

A2}
⟨*proof*⟩

lemma *plus-SupR*: $A \neq \{\} \Longrightarrow Sup\ A + (b::enat) = Sup\ \{a + b \mid a.\ a \in A\}$
⟨*proof*⟩

lemma *plus-SupL*: $A \neq \{\} \Longrightarrow (b::enat) + Sup\ A = Sup\ \{b + a \mid a.\ a \in A\}$
⟨*proof*⟩

lemma *sum-mono3*:
$finite\ B \Longrightarrow A \subseteq B \Longrightarrow (\bigwedge a.\ a \in A \Longrightarrow (f\ a::enat) \leq g\ a) \Longrightarrow$
$sum\ f\ A \leq sum\ g\ B$
⟨*proof*⟩

lemma *sum-Sup-commute*:
fixes $h :: 'a \Rightarrow enat$
assumes *finite* J **and** $\forall i{\in}J.\ \{h\ b \mid b.\ \varphi\ i\ b\} \neq \{\}$
shows $sum\ (\lambda i.\ Sup\ \{h\ b \mid b.\ \varphi\ i\ b\})\ J =$
$Sup\ \{sum\ (\lambda i.\ h\ (b\ i))\ J \mid b\ .\ \forall i{\in}J.\ \varphi\ i\ (b\ i)\}$
⟨*proof*⟩

1.2 Infinite Summation

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

lemma *isum-subset-mono*: $A \subseteq B \Longrightarrow isum\ f\ A \leq isum\ f\ B$
⟨*proof*⟩

lemma *isum-eq-sum*:
$finite\ A \Longrightarrow isum\ f\ A = sum\ f\ A$
⟨*proof*⟩

lemma *isum-cong*:
assumes $A = B$ **and** $\bigwedge x.\ x \in B \Longrightarrow g\ x = h\ x$
shows $isum\ g\ A = isum\ h\ B$
⟨*proof*⟩

lemma *isum-mono*:
assumes $\bigwedge x.\ x \in A \Longrightarrow g\ x \leq h\ x$
shows *isum g A* $\leq$ *isum h A*
⟨*proof*⟩

lemma *isum-mono'*:
assumes $A \subseteq B$
shows *isum g A* $\leq$ *isum g B*
⟨*proof*⟩

lemma *isum-empty*[*simp*]: *isum g* {} = *0*
⟨*proof*⟩

lemma *isum-const-zero*[*simp*]: *isum* ($\lambda x.\ 0$) *A* = *0*
⟨*proof*⟩

lemma *isum-const-zero'*: $\forall\, x{\in}A.\ g\ x = 0 \Longrightarrow$ *isum g A* = *0*
⟨*proof*⟩

lemma *isum-eq-0-iff*: *isum f A* = *0* $\longleftrightarrow$ ($\forall\, a{\in}A.\ f\ a = 0$)
⟨*proof*⟩

lemma *isum-reindex*: *inj-on h A* $\Longrightarrow$ *isum g* (*h* ‘ *A*) = *isum* (*g* $\circ$ *h*) *A*
⟨*proof*⟩

lemma *isum-reindex-cong*: *inj-on l B* $\Longrightarrow$ *A* = *l* ‘ *B* $\Longrightarrow$
($\bigwedge x.\ x \in B \Longrightarrow g\ (l\ x) = h\ x$) $\Longrightarrow$ *isum g A* = *isum h B*
⟨*proof*⟩

lemma *isum-reindex-cong'*:
assumes ($\bigwedge x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow x \neq y \Longrightarrow h\ x = h\ y \Longrightarrow g\ (h\ x) = 0$)
shows *isum g* (*h* ‘ *A*) = *isum* (*g* $\circ$ *h*) *A*
⟨*proof*⟩

lemma *isum-zeros-cong*:
assumes $\bigwedge i.\ i \in T - S \Longrightarrow h\ i = 0$ **and** $\bigwedge i.\ i \in S - T \Longrightarrow g\ i = 0$
and $\bigwedge x.\ x \in S \cap T \Longrightarrow g\ x = h\ x$
shows *isum g S* = *isum h T*

⟨*proof*⟩

lemma *isum-zeros-congL*:
$S \subseteq T \Longrightarrow \forall\, i{\in}T - S.\ g\ i = 0 \Longrightarrow isum\ g\ S = isum\ g\ T$
⟨*proof*⟩

lemma *isum-zeros-congR*:
$S \subseteq T \Longrightarrow \forall\, i{\in}T - S.\ g\ i = 0 \Longrightarrow isum\ g\ T = isum\ g\ S$
⟨*proof*⟩

lemma *isum-singl*[*simp*]: $isum\ f\ \{a\} = f\ a$
⟨*proof*⟩

lemma *isum-two*[*simp*]: $a1 \neq a2 \Longrightarrow isum\ f\ \{a1,a2\} = f\ a1 + f\ a2$
⟨*proof*⟩

lemma *isum-three*[*simp*]: $a1 \neq a2 \Longrightarrow a1 \neq a3 \Longrightarrow a2 \neq a3 \Longrightarrow$
$isum\ f\ \{a1,a2,a3\} = f\ a1 + f\ a2 + f\ a3$
⟨*proof*⟩

lemma *in-le-isum*: $a \in A \Longrightarrow f\ a \leq isum\ f\ A$
⟨*proof*⟩

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

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 $isum\ f\ A \leq x$
⟨*proof*⟩

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

lemma *isum-UNION*:
assumes *dsj*: $\forall\, i{\in}I.\ \forall\, j{\in}I.\ i \neq j \longrightarrow A\ i \cap A\ j = \{\}$
shows $isum\ g\ (\bigcup\ (A\ `\ I)) = isum\ (\lambda i.\ isum\ g\ (A\ i))\ I$
⟨*proof*⟩

lemma *isum-Un*[*simp*]:
assumes $A1 \cap A2 = \{\}$

shows *isum f (A1 ∪ A2) = isum f A1 + isum f A2*
⟨*proof*⟩

lemma *isum-Sigma*:
isum (λ(a,b). f a b) (Sigma A Bs) = isum (λa. isum (f a) (Bs a)) A
⟨*proof*⟩

lemma *isum-Times*: *isum (λ(a,b). f a b) (A × B) = isum (λa. isum (f a) B) A*
⟨*proof*⟩

lemma *isum-swap*: *isum (λa. isum (f a) B) A = isum (λb. isum (λa. f a b) A) B*
(**is** *?L = ?R*)
⟨*proof*⟩

lemma *isum-plus*:
shows *isum (λa. f1 a + f2 a) A = isum f1 A + isum f2 A*
⟨*proof*⟩

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})›
⟨*proof*⟩

lemma *ldropWhile-eq-LCons-iff*: ‹*ldropWhile P lxs = LCons x lxs′ ⟷* (¬ *P x* ∧
(∃ *xs. lxs = lappend* (*llist-of xs*) (*LCons x lxs′*) ∧ (∀ *x* ∈ *set xs. P x*)))›
⟨*proof*⟩

lemma *ltakeWhile-ldropWhile-decomp*:
assumes ‹*x* ∈ *lset lxs*›
shows ‹*lxs = lappend* (*ltakeWhile* ((≠) *x*) *lxs*) (*LCons x* (*ltl* (*ldropWhile* ((≠) *x*) *lxs*)))›
⟨*proof*⟩

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

⟨*proof*⟩

lemma *llength-not-lnull*: ‹¬ *lnull lxs* ⟹ *llength lxs* = *eSuc* (*llength* (*ltl lxs*))›
⟨*proof*⟩

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

lemma *nth-ltaken*: ‹*m* < *n* ⟹ *n* ≤ *llength lxs* ⟹ *nth* (*ltaken n lxs*) *m* = *lnth lxs m*›
⟨*proof*⟩

lemma *set-ltaken*: ‹*set* (*ltaken n lxs*) ⊆ *lset lxs*›
⟨*proof*⟩

lemma *length-ltaken*: ‹*length* (*ltaken n lxs*) = (*if enat n* ≤ *llength lxs then n else the-enat* (*llength lxs*))›
⟨*proof*⟩

lemma *set-ltaken-conv*: ‹*n* ≤ *llength lxs* ⟹ *set* (*ltaken n lxs*) = *lnth lxs* ‘ {*0*..<*n*}›
⟨*proof*⟩

lemma *ltaken-lappend*:
‹*ltaken n* (*lappend lxs lys*) = (*case llength lxs of* ∞ ⇒ *ltaken n lxs* | *enat m* ⇒ *ltaken n lxs* @ *ltaken* (*n* − *m*) *lys*)›
⟨*proof*⟩

lemma *ltaken-LNil*[*simp*]: ‹*ltaken i LNil* = []›
⟨*proof*⟩

3 enat as a Codatatype

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

coinductive *is-einf* **where**
‹*is-einf n* ⟹ *is-einf* (*eS n*)›

inductive *is-fin* **where**
‹*is-fin eZ*›
| ‹*is-fin n* ⟹ *is-fin* (*eS n*)›

lemma *not-is-fin-is-einf*: ‹¬ *is-fin n* ⟹ *is-einf n*›
⟨*proof*⟩

lemma *is-fin-not-is-einf*: ‹*is-fin n* ⟹ ¬ *is-einf n*›
⟨*proof*⟩

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

lemma *einf-eS-iff*: ‹*einf* = *eS x* ⟷ *x* = *einf*›
  ⟨*proof*⟩

lemma *is-einf-einf*: ‹*is-einf einf*›
  ⟨*proof*⟩

lemma *is-einf-eq-einf*: ‹*is-einf n* ⟹ *n* = *einf*›
  ⟨*proof*⟩

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

lemma *is-fin-ex-nat-to-en*: ‹*is-fin n* ⟹ ∃ *m*. *n* = *nat-to-en m*›
  ⟨*proof*⟩

lemma *inj-nat-to-en*: ‹*nat-to-en x* = *nat-to-en y* ⟹ *x* = *y*›
⟨*proof*⟩

lemma *nat-to-en-not-einf*: ‹*nat-to-en n* = *einf* ⟹ *False*›
  ⟨*proof*⟩

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

lemma *inj-enat-to-en*: ‹*inj enat-to-en*›
  ⟨*proof*⟩

lemma *range-enat-to-en*: ‹*n* ∈ *range enat-to-en*›
⟨*proof*⟩

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

setup-lifting *type-definition-enat*

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

lemma *eZ-transfer*[*transfer-rule*]: ‹*pcr-enat eZ 0*›
  ⟨*proof*⟩

lemma *eS-transfer*[*transfer-rule*]: ‹*rel-fun pcr-enat pcr-enat eS eSuc*›
  ⟨*proof*⟩

lemma *ep-nat-to-en*: ‹*ep* (*nat-to-en n*) = *nat-to-en* (*n* − *Suc 0*)›
⟨*proof*⟩

lemma *ep-transfer*[*transfer-rule*]: ‹*rel-fun pcr-enat pcr-enat ep epred*›
⟨*proof*⟩

lemma *corec-enat-code*[*code*]:
‹*corec-enat zer stop end cnt a* =
(*if zer a then 0 else eSuc* (*if stop a then end a else corec-enat zer stop end cnt* (*cnt a*)))›
⟨*proof*⟩

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

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

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

friend-of-corec *plus* :: ‹*en* ⇒ *en* ⇒ *en*› **where**
‹*e1* + *e2* = (*case e1 of eZ* ⇒ (*case e2 of eZ* ⇒ *eZ* | *eS e2′* ⇒ *eS e2′*) | *eS e1′* ⇒ *eS* (*e1′* + *e2*))›
⟨*proof*⟩

lemma *plus-einf-left*[*simp*]: ‹*einf* + *e* = *einf*›
⟨*proof*⟩

lemma *eZ-left-neutral*[*simp*]: ‹*eZ* + *e* = *e*›
⟨*proof*⟩
lemma *eZ-right-neutral*[*simp*]: ‹*e* + *eZ* = *e*›
⟨*proof*⟩
lemma *plus-eS-left*[*simp*]: ‹*eS e1* + *e2* = *eS* (*e1* + *e2*)›
⟨*proof*⟩
lemma *plus-eS-right*[*simp*]: ‹*e1* + *eS e2* = *eS* (*e1* + *e2*)›
⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *plus-einf-right*[*simp*]: ‹*e* + *einf* = *einf*›
⟨*proof*⟩

lemma *nat-to-en-plus*[*simp*]: ‹*nat-to-en* (*m* + *n*) = *nat-to-en m* + *nat-to-en n*›
⟨*proof*⟩

lemma *ezero-transfer*[*transfer-rule*]: ‹*pcr-enat 0 0*›

⟨*proof*⟩

lemma *eplus-transfer*[*transfer-rule*]: ‹*rel-fun pcr-enat* (*rel-fun pcr-enat pcr-enat*) (+) (+)›
⟨*proof*⟩

lemma *case-en-transfer*[*transfer-rule*]: ‹*rel-fun R* (*rel-fun* (*rel-fun pcr-enat R*) (*rel-fun pcr-enat R*)) *case-en co.case-enat*›
⟨*proof*⟩

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

lift-definition *lsum* :: ‹*enat llist* ⇒ *enat*›
is *lsum-en* ⟨*proof*⟩

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

lemma *lsum-code-alt*:
‹*lsum lxs* = (*case ldropWhile* ((=) *0*) *lxs of LNil* ⇒ *0* | *LCons e lxs* ⇒ *e* + *lsum lxs*)›
⟨*proof*⟩

4 Counting in Lazy Lists

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

lift-definition *count-llist* :: ‹*'a llist* ⇒ *'a* ⇒ *enat*›
is *count-llist-en* ⟨*proof*⟩

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

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

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

lemma *in-lset-iff-count-llist*: ‹*x* ∈ *lset lxs* ⟷ *count-llist lxs x* ≠ *0*›
⟨*proof*⟩

lemma *count-llist-zero-iff*: ‹*count-llist lxs x = 0* ⟷ *x* ∉ *lset lxs*›
⟨*proof*⟩

lemma *count-llist-alt*: ‹*count-llist lxs x = llength* (*lfilter* ((=) *x*) *lxs*)›
⟨*proof*⟩

lemma *count-llist-lappend*:
‹*count-llist* (*lappend lxs lys*) *x* = *count-llist lxs x* + (*if lfinite lxs then count-llist lys x else 0*)›
⟨*proof*⟩

lemma *count-llist-LNil*[*simp*]: ‹*count-llist LNil x = 0*›
⟨*proof*⟩

lemma *count-llist-LCons*[*simp*]: ‹*count-llist* (*LCons y lxs*) *x* =
(*if x = y then eSuc* (*count-llist lxs x*) *else count-llist lxs x*)›
⟨*proof*⟩

5 Lazy Lists of Natural Numbers

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

lemma *lset-luptD*[*OF - refl*]:
assumes ‹*k* ∈ *lset lxs*› ‹*lxs = lupt i j*›
shows ‹*i* ≤ *k*› ‹*k* < *j*›
⟨*proof*⟩

lemma *lset-luptI*: ‹*i* ≤ *k* ⟹ *enat k* < *j* ⟹ *k* ∈ *lset* (*lupt i j*)›
⟨*proof*⟩

lemma *lset-lupt*: ‹*lset* (*lupt i j*) = {*k*. *i* ≤ *k* ∧ *enat k* < *j*}›
⟨*proof*⟩

lemma *llength-lupt*[*simp*]: ‹*llength* (*lupt i j*) = *j* − *i*›
⟨*proof*⟩

lemma *lnth-lupt*: ‹*k* < *j* − *enat i* ⟹ *lnth* (*lupt i j*) *k* = *i* + *k*›
⟨*proof*⟩

lemma *lmap-lupt-Suc*: ‹*lmap f* (*lupt* (*Suc i*) (*eSuc j*)) = *lmap* (*f o Suc*) (*lupt i j*)›
⟨*proof*⟩

lemma *llist-conv-lmap-lupt*:
‹*lxs = lmap* (*lnth lxs*) (*lupt 0* (*llength lxs*))›
⟨*proof*⟩

lemma *ldistinct-lupt*[*simp*]: ‹*ldistinct* (*lupt i j*)›
⟨*proof*⟩

lemma *lzip-lupt*: ‹*lzip* (*lupt i j*) (*lupt i k*) = *lmap* (λ*x*. (*x*, *x*)) (*lupt i* (*min j k*))›
⟨*proof*⟩

6 Permutations of Lazy Lists

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

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

lemma *lset-lpermute*:
assumes ‹*lbij-on* π *lxs*›
shows ‹*lset* (*lpermute* π *lxs*) = *lset lxs*›
⟨*proof*⟩

lemma *llength-lpermute*[*simp*]: ‹*llength* (*lpermute* π *lxs*) = *llength lxs*›
⟨*proof*⟩

lemma *lnth-lpermute*[*simp*]: ‹*lbij-on* π *lxs* ⟹ *i* < *llength lxs* ⟹ *lnth* (*lpermute* π *lxs*) *i* = *lnth lxs* (π *i*)›
⟨*proof*⟩

lemma *lfilter-permute*: ‹*ldistinct lxs* ⟹ *ldistinct lys* ⟹ *bij-betw* π (*lset lxs*) (*lset lys*) ⟹ ∀ *x* ∈ *lset lxs*. *P x* ⟷ *Q* (π *x*) ⟹
llength (*lfilter P lxs*) = *llength* (*lfilter Q lys*)›
⟨*proof*⟩

lemma *count-llist-lpermute*:
‹*lbij-on* π *lxs* ⟹ *count-llist* (*lpermute* π *lxs*) *x* = *count-llist lxs x*›
⟨*proof*⟩

lemma *lpermute-lzip*: ‹*lbij-on* π *lxs* ⟹ *llength lys* = *llength lxs* ⟹ *lpermute* π (*lzip lxs lys*) = *lzip* (*lpermute* π *lxs*) (*lpermute* π *lys*)›
⟨*proof*⟩

lemma *exist-nth-occurrence-in-llist*:
‹*count-llist lxs x* > *n* ⟹ ∃ *i* < *llength lxs*. *lnth lxs i* = *x* ∧ *count-list* (*ltaken i lxs*) *x* = *n*›
⟨*proof*⟩

function *lfind-index* **where**
‹*lfind-index n x i lxs* = (*if count-llist lxs x* ≤ *enat n then i else*
if lhd lxs = *x then case n of 0* ⇒ *i* | *Suc m* ⇒ *lfind-index m x* (*Suc i*) (*ltl lxs*)
else lfind-index n x (*Suc i*) (*ltl lxs*))›
⟨*proof*⟩
termination
⟨*proof*⟩
declare *lfind-index.simps*[*simp del*]

lemma *lfind-index-ge*[*simp*]: ‹*lfind-index n x j lxs* ≥ *j*›
⟨*proof*⟩

lemma *lfind-index-less*[*simp*]: ‹*count-llist lxs x* > *enat n* ⟹ *lfind-index n x j lxs* < *llength lxs* + *j*›
⟨*proof*⟩

lemma *lfind-index-lnth*:
‹*count-llist lxs x* > *enat n* ⟹ *lnth lxs* (*lfind-index n x j lxs* − *j*) = *x*›
⟨*proof*⟩

lemma *ltaken-LCons*:
‹*ltaken i* (*LCons x lxs*) = (*case i of* 0 ⇒ [] | *Suc j* ⇒ *x* # *ltaken j lxs*)›
⟨*proof*⟩

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

lemma *lnth-equalityI*: ‹*llength lxs* = *llength lys* ⟹ (⋀*i*. *enat i* < *llength lxs* ⟹ *lnth lxs i* = *lnth lys i*) ⟹ *lxs* = *lys*›
⟨*proof*⟩

lemma *lfind-index-inject*:
‹*count-llist lxs x* > *enat n* ⟹ *count-llist lxs y* > *enat m* ⟹
lfind-index n x j lxs = *lfind-index m y j lxs* ⟹ *n* = *m* ∧ *x* = *y*›
⟨*proof*⟩

lemma *count-list-ltaken-less*:
‹*i* < *llength lxs* ⟹ *count-list* (*ltaken i lxs*) (*lnth lxs i*) < *count-llist lxs* (*lnth lxs i*)›
⟨*proof*⟩

lemma *count-list-inject*:
‹*enat i* < *llength lxs* ⟹ *enat j* < *llength lxs* ⟹
count-list (*ltaken i lxs*) (*lnth lxs j*) = *count-list* (*ltaken j lxs*) (*lnth lxs j*) ⟹
lnth lxs i = *lnth lxs j* ⟹ *i* = *j*›
⟨*proof*⟩

7 Same-Count Equivalence Relation

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

lemma *lset-eq-cmset*: ‹*eq-cmset lxs lys* ⟹ *lset lxs* = *lset lys*›
⟨*proof*⟩

lemma *eq-cmset-llength*: ‹*eq-cmset lxs lys* ⟹ *llength lxs* = *llength lys*›
⟨*proof*⟩

lemma *eq-cmset-alt*: ‹*eq-cmset lxs′ lxs* ⟷ (∃ π. *lbij-on π lxs* ∧ *lpermute π lxs* = *lxs′*)›
⟨*proof*⟩

lemma *eq-cmset-lzip-left*:
assumes ‹*eq-cmset lxs′ lxs*› ‹*llength lys* = *llength lxs*›
shows ‹∃ *lys′*. *eq-cmset* (*lzip lxs′ lys′*) (*lzip lxs lys*) ∧ *llength lys′* = *llength lxs′*›
⟨*proof*⟩

lemma *eq-cmset-lmap*:
assumes ‹*eq-cmset lxs lys*›
shows ‹*eq-cmset* (*lmap f lxs*) (*lmap f lys*)›
⟨*proof*⟩

lemma *llist-all2-reorder-left-invariance*:
assumes *rel*: ‹*llist-all2 R lxs lys*› **and** *ms-x*: ‹*eq-cmset lxs′ lxs*›
shows ‹∃ *lys′*. *llist-all2 R lxs′ lys′* ∧ *eq-cmset lys′ lys*›
⟨*proof*⟩

8 Countable Multisets as a Quotient

quotient-type *'a cmset* = ‹*'a llist*› / *eq-cmset*
⟨*proof*⟩

lift-bnf (*cmset*: *'a*) *cmset*
for *map*: *cmimage rel*: *cmrel*
⟨*proof*⟩

9 Lazy List Interleaving

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

lemma *ltaken-ldropn-decomp*: ‹*lxs* = *lappend* (*llist-of* (*ltaken n lxs*)) (*ldropn n lxs*)›
⟨*proof*⟩

lemma *count-llist-llist-of*[*simp*]: ‹*count-llist* (*llist-of xs*) *x* = *count-list xs x*›
⟨*proof*⟩

lemma *lmap-lfilter-swap*: ‹∀ *x* ∈ *lset lxs*. *P x* ⟷ *Q* (*f x*) ⟹ *lmap f* (*lfilter P lxs*) = *lfilter Q* (*lmap f lxs*)›
⟨*proof*⟩

lemma *lsum-lmap-zero*: ‹*lsum* (*lmap* (λ*z*. *0*) *lxs*) = *0*›
⟨*proof*⟩

lemma *lsum-LNil*[*simp*]: ‹*lsum LNil* = *0*›
⟨*proof*⟩

lemma *lsum-LCons*[*simp*]: ‹*lsum* (*LCons x lxs*) = *x* + *lsum lxs*›
⟨*proof*⟩

lemma *count-llist-lmap-const*:
‹*count-llist* (*lmap* (λ*z*. *a*) *lxs*) *x* = (*if x* = *a then llength lxs else 0*)›
⟨*proof*⟩

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

lemma *ltaken-Suc*:
‹*ltaken* (*Suc i*) *lxs* = (*if i* < *llength lxs then ltaken i lxs* @ [*lnth lxs i*] *else ltaken i lxs*)›
⟨*proof*⟩

lemma *ltaken-all-same*: ‹*enat n* ≥ *llength lxs* ⟹ *enat m* ≥ *llength lxs* ⟹ *ltaken n lxs* = *ltaken m lxs*›
⟨*proof*⟩

lemma *lsum-mono*: ‹*lprefix lxs lys* ⟹ *lsum lxs* ≤ *lsum lys*›
⟨*proof*⟩

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

lemma *lSup-eq-lappend*: ‹*Complete-Partial-Order.chain lprefix Y* ⟹ ∃ *lys* ∈ *Y*. *lprefix* (*llist-of xs*) *lys* ⟹
lSup Y = *lappend* (*llist-of xs*) (*lSup* (*ldropn* (*length xs*) ‘ *Y*))›
⟨*proof*⟩

lemma *lsum-0D*: ‹*x* ∈ *lset lxs* ⟹ *lsum lxs* = *0* ⟹ *x* = *0*›
⟨*proof*⟩

lemma *lsum-0I*: ‹∀ *x*∈*lset lxs*. *x* = *0* ⟹ *lsum lxs* = *0*›
⟨*proof*⟩

lemma *lsum-0-iff*: ‹*lsum lxs* = *0* ⟷ (∀ *x* ∈ *lset lxs*. *x* = *0*)›
⟨*proof*⟩

lemma *lsum-lappend-lfinite*: ‹*lfinite lxs* ⟹ *lsum* (*lappend lxs lys*) = *lsum lxs* +

lsum lys›
⟨*proof*⟩

lemma *lsum-lappend*: ‹*lsum* (*lappend lxs lys*) = (*if lfinite lxs then lsum lxs* + *lsum lys else lsum lxs*)›
⟨*proof*⟩

lemma *lsum-llist-of*[*simp*]: ‹*lsum* (*llist-of xs*) = *sum-list xs*›
⟨*proof*⟩

lemma *lsum-cont*:
‹*Complete-Partial-Order.chain lprefix Y* ⟹ *Y* ≠ {} ⟹ *lsum* (*lSup Y*) = ⨆ (*lsum* ‘ *Y*)›
⟨*proof*⟩

lemma *mcont2mcont-lsum*[*THEN lfp.mcont2mcont*, *simp*, *cont-intro*]:
shows *mcont-lsum*: ‹*mcont lSup* (*lprefix*) *Sup* (≤) *lsum*›
⟨*proof*⟩

lemma *lsum-lfilter-nonzero*: ‹*lsum* (*lfilter* ((≠) 0) *lxs*) = *lsum lxs*›
⟨*proof*⟩

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

abbreviation ‹*lenats* ≡ *lupt* 0 ∞›

lemma *LSUM-lfilter*: ‹*LSUM* (*lfilter* (λ*x*. *f x* ≠ 0) *lxs*) *f* = *LSUM lxs f*›
⟨*proof*⟩

lemma *LSUM-count-llist-lfilter*:
‹*lsum* (*lmap* (λ*lxs*. *count-llist lxs x*) (*lfilter* (λ*x*. ¬ *lnull x*) *lxss*)) =
lsum (*lmap* (λ*lxs*. *count-llist lxs x*) *lxss*)›
⟨*proof*⟩

lemma *lflat-LNil*[*simp*]: ‹*lflat LNil* = *LNil*›
⟨*proof*⟩

lemma *lflat-LCons*[*simp*]: ‹*lflat* (*LCons xs lxss*) = *lappend* (*llist-of xs*) (*lflat lxss*)›
⟨*proof*⟩

lemma *count-llist-mono*:
assumes ‹*lprefix lxs lys*›
shows ‹*count-llist lxs x* ≤ *count-llist lys x*›
⟨*proof*⟩

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

⟨*proof*⟩

lemma *count-llist-cont*: ‹*Complete-Partial-Order.chain lprefix Y* ⟹ *Y* ≠ {} ⟹
count-llist (*lSup Y*) *x* = (⨆ *lxs*∈*Y*. *count-llist lxs x*)›
⟨*proof*⟩

lemma *mcont2mcont-count-llist*[*THEN lfp.mcont2mcont*, *simp*, *cont-intro*]:
shows *mcont-count-llist*: ‹*mcont lSup* (*lprefix*) *Sup* (≤) (λ*lxs*. *count-llist lxs x*)›
⟨*proof*⟩

lemma *mono2mono-lflat*[*THEN llist.mono2mono*, *simp*, *cont-intro*]:
shows *mono-lflat*: ‹*monotone lprefix lprefix lflat*›
⟨*proof*⟩

lemma *mcont2mcont-lflat*[*THEN llist.mcont2mcont*, *simp*, *cont-intro*]:
shows *mcont-lflat*: ‹*mcont lSup lprefix lSup lprefix lflat*›
⟨*proof*⟩

lemma *lflat-LNil-iff*: ‹*lflat lxss* = *LNil* ⟷ (∀ *xs* ∈ *lset lxss*. *xs* = [])›
⟨*proof*⟩

lemma *count-llist-lflat*: ‹*count-llist* (*lflat lxss*) *x* = *lsum* (*lmap* (λ*xs*. *count-list xs x*) *lxss*)›
⟨*proof*⟩

lemma *lset-lflat*: ‹*lset* (*lflat lxss*) = (⋃ *xs* ∈ *lset lxss*. *set xs*)›
⟨*proof*⟩

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

lemma *set-ltaken-conv′*: ‹*set* (*ltaken n lxs*) = (*case llength lxs of enat m* ⇒ *lnth lxs* ‘ {*0*..<*min n m*} | - ⇒ *lnth lxs* ‘ {*0* ..< *n*})›
⟨*proof*⟩

lemma *lset-lmerge*: ‹*lset* (*lmerge lxss*) = (⋃ *lxs* ∈ *lset lxss*. *lset lxs*)›
⟨*proof*⟩

lemma *lsum-lmap-add*: ‹*lsum* (*lmap* (λ*x*. *f x* + *g x*) *lxs*) = *lsum* (*lmap f lxs*) + *lsum* (*lmap g lxs*)›
⟨*proof*⟩

lemma *count-list-alt*: ‹*count-list xs x* = *card* {*j*. *j* < *length xs* ∧ *xs* ! *j* = *x*}›
⟨*proof*⟩

lemma *count-list-lvertical*:
‹*count-list* (*lvertical lxss i*) *x* = (*if i* < *llength lxss then card* {*j*. *j* ≤ *i* ∧ *j* < *llength* (*lnth lxss i*) ∧ *lnth* (*lnth lxss i*) *j* = *x*} *else* 0)›
⟨*proof*⟩

lemma *llength-less-lfinite*[*simp*]: ‹*llength lxss* < *enat j* ⟹ *lfinite lxss*›
⟨*proof*⟩

lemma *ltaken-all*: ‹*llength lxss* < *enat j* ⟹ *ltaken j lxss* = *list-of lxss*›
⟨*proof*⟩

lemma *count-list-map-filter*:
‹*count-list* (*List.map-filter f xs*) *x* = *card* {*i*. *i* < *length xs* ∧ *f* (*xs* ! *i*) = *Some x*}›
⟨*proof*⟩

lemma *count-list-lhorizontal*:
‹*count-list* (*lhorizontal lxss j*) *x* = *card* {*i*. *i* < *j* ∧ *i* < *llength lxss* ∧ *j* < *llength* (*lnth lxss i*) ∧ *lnth* (*lnth lxss i*) *j* = *x*}›
⟨*proof*⟩

lemma *LSUM-extend*: **assumes** ‹*lprefix lxs lys*›
‹∀ *i* < *llength lxs*. *f* (*lnth lxs i*) = *g* (*lnth lys i*)›
‹∀ *i* < *llength lys*. *i* ≥ *llength lxs* ⟶ *g* (*lnth lys i*) = 0›
shows ‹*LSUM lxs f* = *LSUM lys g*›
⟨*proof*⟩

lemma *lupt-0*[*simp*]: ‹*lupt i* 0 = *LNil*›
⟨*proof*⟩

lemma *lprefix-lupt*: ‹*j* ≤ *k* ⟹ *lprefix* (*lupt i j*) (*lupt i k*)›
⟨*proof*⟩

lemma *if-card-else-0*: ‹(*if P z then card* {*x*. *Q x z*} *else* 0) = *card* {*x*. *P z* ∧ *Q x z*}›
⟨*proof*⟩

lemma *LSUM-llist-of*[*simp*]: ‹*LSUM* (*llist-of xs*) *f* = *sum-list* (*map f xs*)›
⟨*proof*⟩

lemma *card-sum-list*: ‹(⋀*i*. *P i* ⟹ *i* ≤ *n*) ⟹
card {*i* :: *nat*. *P i*} = *sum-list* (*map* (λ*i*. *if P i then* 1 *else* 0) [0 ..< *Suc n*])›
⟨*proof*⟩

lemma *llist-of-lupt*: ‹*llist-of* [*i* ..< *j*] = *lupt i j*›
⟨*proof*⟩

lemma *enat-sum-list*: ‹*enat* (*sum-list xs*) = *sum-list* (*map enat xs*)›
⟨*proof*⟩

lemma *card-LSUM*:
assumes ‹$(\bigwedge i.\ P\ i \Longrightarrow i \leq n)$›
shows ‹$enat\ (card\ \{i :: nat.\ P\ i\}) = LSUM\ lenats\ (\lambda i.\ if\ P\ i\ then\ 1\ else\ 0)$›
⟨*proof*⟩

lemma *LSUM-extend-lenats*:
‹$LSUM\ lxs\ f = \ LSUM\ lenats\ (\lambda i.\ if\ enat\ i < llength\ lxs\ then\ f\ (lnth\ lxs\ i)\ else\ 0)$›
⟨*proof*⟩

lemma *LCons-eq-lmap-conv*:
‹$(LCons\ y\ lys = lmap\ f\ lxs) = (\exists x\ lxs'.\ lxs = LCons\ x\ lxs' \wedge y = f\ x \wedge lys = lmap\ f\ lxs')$›
⟨*proof*⟩

lemma *lmap-eq-lappend*: ‹$lfinite\ lys \Longrightarrow lmap\ f\ lxs = lappend\ lys\ lzs \longleftrightarrow$
$(\exists lus\ lvs.\ lxs = lappend\ lus\ lvs \wedge lys = lmap\ f\ lus \wedge lzs = lmap\ f\ lvs)$›
⟨*proof*⟩

lemma *llist-of-eq-lmap-conv*:
‹$llist\text{-}of\ xs = lmap\ f\ lxs \longleftrightarrow (\exists ys.\ lxs = llist\text{-}of\ ys \wedge map\ f\ ys = xs)$›
⟨*proof*⟩

lemma *neq-LCons-conv*: ‹$(\forall x\ lxs.\ lys \neq LCons\ x\ lxs) \longleftrightarrow lys = LNil$›
⟨*proof*⟩

lemma *epred-iadd*: ‹$epred\ (a + b) = (if\ a = 0\ then\ epred\ b\ else\ epred\ a + b)$›
⟨*proof*⟩

lemma *LSUM-isum*: ‹$ldistinct\ lxs \Longrightarrow LSUM\ lxs\ f = isum\ f\ (lset\ lxs)$›
⟨*proof*⟩

lemma *set-partition-subset*: ‹$A \subseteq B \Longrightarrow B = A \cup (B - A)$›
⟨*proof*⟩

lemma *count-llist-isum*: ‹$count\text{-}llist\ lxs\ x = isum\ (\lambda i.\ if\ enat\ i < llength\ lxs \wedge lnth\ lxs\ i = x\ then\ 1\ else\ 0)\ UNIV$›
⟨*proof*⟩

lemma *count-llist-lmerge*: ‹$count\text{-}llist\ (lmerge\ lxss)\ x = lsum\ (lmap\ (\lambda lxs.\ count\text{-}llist\ lxs\ x)\ lxss)$›
⟨*proof*⟩

10 Countable Multisets as a Subtype

lift-definition *cmcount* :: ‹$'a\ cmset \Rightarrow 'a \Rightarrow enat$› **is** *count-llist*
⟨*proof*⟩

lift-definition *cmempty* :: ‹*'a cmset*› **is** *LNil* ⟨*proof*⟩

lemma *cmcount-cmempty*[*simp*]: ‹*cmcount cmempty x = 0*›
⟨*proof*⟩

lemma *countable-nonzero-cmcount*:
fixes *M* :: ‹*'a cmset*›
shows ‹*countable* {*x*. *cmcount M x ≠ 0*}›
⟨*proof*⟩

lemma *countable-exists-llist*:
assumes ‹*countable X*›
shows ‹∃ *lxs*. *lset lxs = X ∧ ldistinct lxs*›
⟨*proof*⟩

lemma *in-range-cmcount*:
assumes ‹*countable* {*x* :: *'a*. *f x ≠ 0*}›
shows ‹*f ∈ range cmcount*›
⟨*proof*⟩

lemma *inj-cmcount*: ‹*inj cmcount*›
⟨*proof*⟩

lemma *type-definition-cmset*: ‹*type-definition cmcount* (*inv cmcount*) {*f*. *countable* {*x*. *f x ≠ 0*}}›
⟨*proof*⟩

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

lemma *linterleave-LNil*[*simp*]:
‹*linterleave LNil lys = lys*›
‹*linterleave lys LNil = lys*›
⟨*proof*⟩

lemma *linterleave-LCons1*[*simp*]:
‹*linterleave* (*LCons x lxs*) *lys* = *LCons x* (*linterleave lys lxs*)›
⟨*proof*⟩

lemma *lset-linterleave1*:
‹*x ∈ lset* (*linterleave lxs lys*) ⟹
x ∈ lset lxs ∪ lset lys›
⟨*proof*⟩

lemma *lset-linterleave2*:
‹$x \in lset\ lxs \Longrightarrow x \in lset\ (linterleave\ lxs\ lys)$›
⟨*proof*⟩

lemma *lset-linterleave3*:
‹$x \in lset\ lys \Longrightarrow$
$x \in lset\ (linterleave\ lxs\ lys)$›
⟨*proof*⟩

lemma *lset-linterleave*[*simp*]:
‹$lset\ (linterleave\ lxs\ lys) = lset\ lxs \cup lset\ lys$›
⟨*proof*⟩

lemma *ldistinct-linterleave*: ‹$ldistinct\ lxs \Longrightarrow ldistinct\ lys \Longrightarrow lset\ lxs \cap lset\ lys = \{\} \Longrightarrow ldistinct\ (linterleave\ lxs\ lys)$›
⟨*proof*⟩

coinductive *linfinite* **where**
‹$linfinite\ lxs \Longrightarrow linfinite\ (LCons\ x\ lxs)$›

inductive *linfinite-cong* **for** R **where**
‹$R\ lxs \Longrightarrow linfinite\text{-}cong\ R\ lxs$›
| ‹$linfinite\ lxs \Longrightarrow linfinite\text{-}cong\ R\ lxs$›
| ‹$linfinite\text{-}cong\ R\ lxs \Longrightarrow linfinite\text{-}cong\ R\ (LCons\ x\ lxs)$›

lemma *linfinite-coinduct-upto*[*consumes 1*, *case-names linfinite*]:
assumes ‹$X\ lxs$› ‹$\bigwedge lys.\ X\ lys \Longrightarrow \exists\, lxs\ x.\ lys = LCons\ x\ lxs \wedge linfinite\text{-}cong\ X\ lxs$›
shows ‹$linfinite\ lxs$›
⟨*proof*⟩

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

lemma *linfinite-linterleaveL*: ‹$linfinite\ lxs \Longrightarrow linfinite\ (linterleave\ lxs\ lys)$›
⟨*proof*⟩

lemma *linfinite-linterleaveR*: ‹$linfinite\ lys \Longrightarrow linfinite\ (linterleave\ lxs\ lys)$›
⟨*proof*⟩

lemma *lfinite-imp-not-linfinite*: ‹$lfinite\ lxs \Longrightarrow \neg\ linfinite\ lxs$›
⟨*proof*⟩

lemma *not-lfinite-imp-linfinite*: ‹$\neg\ lfinite\ lxs \Longrightarrow linfinite\ lxs$›
⟨*proof*⟩

lemma *linfinite-eq-not-lfinite*: ‹$linfinite\ lxs \longleftrightarrow \neg\ lfinite\ lxs$›
⟨*proof*⟩

lemma *linfinite-eq-llength*: ‹*linfinite lxs* ⟷ *llength lxs* = ∞›
⟨*proof*⟩

lemma *llength-linterleave*[*simp*]: ‹*llength* (*linterleave lxs lys*) = *llength lxs* + *llength lys*›
⟨*proof*⟩

lemma *lnth-linterleave-swap*:
‹*lnth* (*linterleave lxs lys*) *i* ∉ *lset lys* ⟹ *i* < *llength* (*linterleave lxs lys*) ⟹
∃ *j* < *min* (*Suc i*) (*llength lxs*). *lnth* (*linterleave lxs lys*) *i* = *lnth lxs j*›
⟨*proof*⟩

lemma *ldropWhile-eq-ldropn*:
‹∃ *x* ∈ *lset lxs*. ¬ *P x* ⟹ ∃ *n*. *enat n* = *llength* (*ltakeWhile P lxs*) ∧ *ldropWhile P lxs* = *ldropn n lxs*›
⟨*proof*⟩

lemma *ltl-linterleave*[*simp*]: ‹¬*lnull lxs* ⟹ *ltl* (*linterleave lxs lys*) = *linterleave lys* (*ltl lxs*)›
⟨*proof*⟩

lemma *linfinite-ltl*[*simp*]: ‹*linfinite lxs* ⟹ *linfinite* (*ltl lxs*)›
⟨*proof*⟩

lemma *linfinite-ldropn*[*simp*]: ‹*linfinite lxs* ⟹ *linfinite* (*ldropn n lxs*)›
⟨*proof*⟩

lemma *linfinite-not-lnull*: ‹*linfinite lxs* ⟹ ¬ *lnull lxs*›
⟨*proof*⟩

lemma *ldropn-linterleave*: ‹*linfinite lxs* ⟹ *linfinite lys* ⟹ *ldropn n* (*linterleave lxs lys*) =
(*if* ∃ *m*. *n* = 2 ∗ *m then linterleave* (*ldropn* (*n div* 2) *lxs*) (*ldropn* (*n div* 2) *lys*)
else
linterleave (*ldropn* (*n div* 2) *lys*) (*ldropn* (*Suc* (*n div* 2)) *lxs*))›
⟨*proof*⟩

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*⟩

lemma *llength-ltakeWhile-linterleave-eq*:
‹*enat* (2 ∗ *m*) = *llength* (*ltakeWhile* ((≠) *x*) (*linterleave lxs lys*)) ⟹
x ∉ *set* (*ltaken m lxs*) ∧ *x* ∉ *set* (*ltaken m lys*)›
⟨*proof*⟩

lemma *llength-ltakeWhile-linterleave-ge-Suc*:
‹*linfinite lxs* ⟹ *linfinite lys* ⟹

*enat (Suc (2 * m)) ≤ llength (ltakeWhile ((≠) x) (linterleave lxs lys)) ⟹*
x ∉ set (ltaken (Suc m) lxs) ∧ x ∉ set (ltaken m lys)›
⟨*proof*⟩

lemma *llength-ltakeWhile-linterleave-eq-Suc*:
‹*linfinite lxs ⟹ linfinite lys ⟹*
*enat (Suc (2 * m)) = llength (ltakeWhile ((≠) x) (linterleave lxs lys)) ⟹*
x ∉ set (ltaken (Suc m) lxs) ∧ x ∉ set (ltaken m lys)›
⟨*proof*⟩

lemma *count-llist-linterleave*:
‹*count-llist (linterleave lxs lys) x = count-llist lxs x + count-llist lys x*›
⟨*proof*⟩

lemma *lfinite-linterleave*[*simp*]: ‹*lfinite (linterleave lxs lys) ⟷ lfinite lxs ∧ lfinite lys*›
⟨*proof*⟩

lift-definition *cmadd* :: ‹*'a cmset ⇒ 'a cmset ⇒ 'a cmset*› **is** *linterleave*
⟨*proof*⟩

lemma *cmcount-cmadd*[*simp*]: ‹*cmcount (cmadd M N) x = cmcount M x + cmcount N x*›
⟨*proof*⟩

lemma *count-llist-lmap*: ‹*count-llist (lmap f lxs) b = isum (count-llist lxs) {a. f a = b}*›
⟨*proof*⟩

lemma *cmcount-cmimage*[*simp*]: ‹*cmcount (cmimage f M) b = isum (cmcount M) {a. f a = b}*›
⟨*proof*⟩

lift-definition *cminfinite* :: ‹*'a cmset ⇒ bool*› **is** *linfinite*
⟨*proof*⟩

lemma *Sup-enat-eq-inf*: ‹*Sup (A :: enat set) = ∞ ⟷ ∞ ∈ A ∨ infinite A*›
⟨*proof*⟩

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

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

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

lemma *in-lset-mk-chainD*: ‹*X* ∈ *lset* (*mk-chain A C*) ⟹ *infinite A* ⟹ ∃ *B* ⊆ *A*. *finite B* ∧ *X* = *C* ∪ *B*›
⟨*proof*⟩

lemma *linfinite-mk-chain*[*simp*]: ‹*linfinite* (*mk-chain A C*)›
⟨*proof*⟩

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

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

lemma *chain-mk-chain*: ‹*infinite A* ⟹ *A* ∩ *C* = {} ⟹ *chain* (⊂) (*mk-chain A C*)›
⟨*proof*⟩

lemma *chainD*:
‹*chain R lxs* ⟹ *enat* (*Suc i*) < *llength lxs* ⟹ *R* (*lnth lxs i*) (*lnth lxs* (*Suc i*))›
⟨*proof*⟩

lemma *chain-transD*:
assumes ‹*transp R*›
shows ‹*chain R lxs* ⟹ *i* < *j* ⟹ *enat j* < *llength lxs* ⟹ *R* (*lnth lxs i*) (*lnth lxs j*)›
⟨*proof*⟩

lemma *chain-cpo-chain*: ‹*transp R* ⟹ *chain R lxs* ⟹ *Complete-Partial-Order.chain* (*reflclp R*) (*lset lxs*)›
⟨*proof*⟩

lemma *reflclp-subset*: ‹*reflclp* (⊂) = (⊆)›
⟨*proof*⟩

lemma *sum-strict-mono2-enat*:
fixes *f* :: ‹*'a* ⇒ *enat*›
assumes ‹*finite B*› ‹*A* ⊆ *B*› ‹*b* ∈ *B*−*A*› ‹*f b* > *0*› ‹∞ ∉ *f* ‘ *B*›
shows ‹*sum f A* < *sum f B*›
⟨*proof*⟩

lemma *infinite-lset-chain*:
assumes ‹*linfinite lxs*› ‹*chain R lxs*› ‹*irreflp R*› ‹*transp R*›
shows ‹*infinite* (*lset lxs*)›

⟨*proof*⟩

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

lemma *cminfinite-alt*: ‹*cminfinite M* = ((∃ *x*. *cmcount M x* = ∞) ∨ *infinite* {*x*. *cmcount M x* ≠ *0*})›
⟨*proof*⟩

lemma *cmset-alt*: ‹*cmset M* = {*a*. *cmcount M a* ≠ *0*}›
⟨*proof*⟩

lift-definition *cmconst* :: ‹*'a* ⇒ *'a cmset*› **is** ‹*repeat*› ⟨*proof*⟩

lemma *cminfinite-cmconst*[*simp*]: ‹*cminfinite* (*cmconst a*)›
⟨*proof*⟩

lemma *cminfinite-cmimage*: ‹*cminfinite* (*cmimage f M*) ⟷ *cminfinite M*›
⟨*proof*⟩

locale *cmset-as-Quotient*
begin

setup-lifting *Quotient-cmset*

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

end

locale *cmset-as-typedef*
begin

setup-lifting *type-definition-cmset*

lemma *cmempty-transfer*[*transfer-rule*]: ‹*pcr-cmset R* (λ*-*. *0*) *cmempty*›
⟨*proof*⟩

lemma *cmadd-transfer*[*transfer-rule*]: ‹*rel-fun* (*pcr-cmset R*) (*rel-fun* (*pcr-cmset R*) (*pcr-cmset R*)) (λ*M N x*. *M x* + *N x*) *cmadd*›
⟨*proof*⟩

lemma *cmimage-transfer*[*transfer-rule*]: ‹*rel-fun* (=) (*rel-fun* (*pcr-cmset* (=)) (*pcr-cmset* (=))) (λ*f M b*. *isum M* {*a*. *f a* = *b*}) *cmimage*›

⟨*proof*⟩

lemma *cmset-transfer*[*transfer-rule*]: ‹*rel-fun* (*pcr-cmset* (=)) (=) (λ*M*. {*a*. *M a* ≠ *0*}) *cmset*›
⟨*proof*⟩

lemma *cminfinite-transfer*[*transfer-rule*]: ‹*rel-fun* (*pcr-cmset* (=)) (=) (λ*M*. (∃ *x*. *M x* = ∞) ∨ *infinite* {*x*. *M x* ≠ *0*}) *cminfinite*›
⟨*proof*⟩

lift-definition *cmreplace* :: ‹'*a cmset* ⇒ '*a* ⇒ '*a* ⇒ '*a cmset*› **is**
‹λ*f a b*. *f*(*a* := *epred* (*f a*), *b* := *eSuc* (*f b*))›
⟨*proof*⟩

lemma *cminfinite-cmreplace*: ‹*cminfinite M* ⟹ *cminfinite* (*cmreplace M a b*)›
⟨*proof*⟩
end

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

end

11 Countably Infinite Multisets

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

typedef '*a cinfmset* = ‹{*M* :: '*a cmset*. *cminfinite M*}›
⟨*proof*⟩

setup-lifting *type-definition-cinfmset*

lift-bnf (*no-warn-wits*) '*a cinfmset*
for *map*: *cinfmimage rel*: *cinfmrel*
⟨*proof*⟩

lift-definition *cinfmcount* :: ‹'*a cinfmset* ⇒ '*a* ⇒ *enat*› **is** *cmcount* ⟨*proof*⟩

context begin
interpretation *cmset-as-typedef* ⟨*proof*⟩
lift-definition *cinfmreplace* :: ‹'*a cinfmset* ⇒ '*a* ⇒ '*a* ⇒ '*a cinfmset*› **is** *cmreplace*
⟨*proof*⟩

lemma *set-infmset-alt*: ‹*set-cinfmset xs* = {*a*. *cinfmcount xs a* ≠ *0*}›
⟨*proof*⟩

lemma *set-cinfset-cinfmreplace*:

‹*set-cinfmset* (*cinfmreplace M x y*) ⊆ *set-cinfmset M* ∪ {*y*}›
including *cmset.lifting*
⟨*proof*⟩
end

lemma *inj-cinfmcount*: ‹*inj cinfmcount*›
⟨*proof*⟩

lemma *in-range-cinfmcount*:
‹*countable* {*x*. *f x* ≠ *0*} ⟹ (∃ *x*. *f x* = ∞) ∨ *infinite* {*x*. *f x* ≠ *0*} ⟹ *f* ∈ *range cinfmcount*›
⟨*proof*⟩

lemma *cinfmcount-cinfmimage*[*simp*]:
‹*cinfmcount* (*cinfmimage f M*) *b* = *isum* (*cinfmcount M*) {*a*. *f a* = *b*}›
⟨*proof*⟩

lemma *countable-nonzero-cinfmcount*: ‹*countable* {*x*. *cinfmcount M x* ≠ *0*}›
⟨*proof*⟩

lemma *infinite-nonzero-cinfmcount*: ‹(∃ *x*. *cinfmcount M x* = ∞) ∨ *infinite* {*x*. *cinfmcount M x* ≠ *0*}›
⟨*proof*⟩

lemma *type-definition-cinfmset*: ‹*type-definition cinfmcount* (*inv cinfmcount*) {*f*. *countable* {*x*. *f x* ≠ *0*} ∧ ((∃ *x*. *f x* = ∞) ∨ *infinite* {*x*. *f x* ≠ *0*})}›
⟨*proof*⟩

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

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

context includes *cinfmset.lifting* **begin**
interpretation *cmset-as-Quotient* ⟨*proof*⟩
lemma *get-cinfmset-inject*:
‹∀ *i*. *get-cinfmset M i* = *get-cinfmset N i* ⟹ *M* = *N*›
⟨*proof*⟩

end

end