Coinductive

Andreas Lochbihler
with contributions by Johannes Hölzl

February 23, 2021

Abstract

This article collects formalisations of general-purpose coinductive data types and sets. Currently, it contains:

- coinductive natural numbers,
- coinductive lists, i.e. lazy lists or streams, and a library of operations on coinductive lists,
- coinductive terminated lists, i.e. lazy lists with the stop symbol containing data,
- coinductive streams,
- coinductive resumptions, and
- numerous examples which include a version of König's lemma and the Hamming stream.

The initial theory was contributed by Paulson and Wenzel. Extensions and other coinductive formalisations of general interest are welcome.

Contents

1 Extended natural numbers as a codatatype

theory *Coinductive-Nat* **imports**
HOL−Library.Extended-Nat
HOL−Library.Complete-Partial-Order2
begin

lemma *inj-enat* [*simp*]: *inj-on enat A*
⟨*proof*⟩

lemma *Sup-range-enat* [*simp*]: *Sup* (*range enat*) = ∞
⟨*proof*⟩

lemmas *eSuc-plus* = *iadd-Suc*

lemmas *plus-enat-eq-0-conv* = *iadd-is-0*

lemma *enat-add-sub-same*:
fixes *a b* :: *enat* **shows** *a* ≠ ∞ ⟹ *a* + *b* − *a* = *b*
⟨*proof*⟩

lemma *enat-the-enat*: *n* ≠ ∞ ⟹ *enat* (*the-enat n*) = *n*
⟨*proof*⟩

lemma *enat-min-eq-0-iff*:
fixes *a b* :: *enat*
shows *min a b* = *0* ⟷ *a* = *0* ∨ *b* = *0*
⟨*proof*⟩

lemma *enat-le-plus-same*: *x* ≤ (*x* :: *enat*) + *y x* ≤ *y* + *x*
⟨*proof*⟩

lemma *the-enat-0* [*simp*]: *the-enat 0* = *0*
⟨*proof*⟩

lemma *the-enat-eSuc*: *n* ≠ ∞ ⟹ *the-enat* (*eSuc n*) = *Suc* (*the-enat n*)
⟨*proof*⟩

coinductive-set *enat-set* :: *enat set*
where *0* ∈ *enat-set*
| *n* ∈ *enat-set* ⟹ (*eSuc n*) ∈ *enat-set*

lemma *enat-set-eq-UNIV* [*simp*]: *enat-set* = *UNIV*
⟨*proof*⟩

1.1 Case operator

lemma *enat-coexhaust*:
obtains (*0*) *n* = *0*
| (*eSuc*) *n′* **where** *n* = *eSuc n′*

⟨*proof*⟩

locale *co* **begin**

free-constructors (*plugins del*: *code*) *case-enat* **for**
  0::*enat*
| *eSuc epred*
where
  epred 0 = *0*
  ⟨*proof*⟩

end

lemma *enat-cocase-0* [*simp*]: *co.case-enat z s 0* = *z*
⟨*proof*⟩

lemma *enat-cocase-eSuc* [*simp*]: *co.case-enat z s* (*eSuc n*) = *s n*
⟨*proof*⟩

lemma *neq-zero-conv-eSuc*: *n* ≠ *0* ⟷ (∃ *n'*. *n* = *eSuc n'*)
⟨*proof*⟩

lemma *enat-cocase-cert*:
  assumes *CASE* ≡ *co.case-enat c d*
  shows (*CASE 0* ≡ *c*) &&& (*CASE* (*eSuc n*) ≡ *d n*)
  ⟨*proof*⟩

lemma *enat-cosplit-asm*:
  P (*co.case-enat c d n*) = (¬ (*n* = *0* ∧ ¬ *P c* ∨ (∃ *m*. *n* = *eSuc m* ∧ ¬ *P* (*d m*))))
⟨*proof*⟩

lemma *enat-cosplit*:
  P (*co.case-enat c d n*) = ((*n* = *0* ⟶ *P c*) ∧ (∀ *m*. *n* = *eSuc m* ⟶ *P* (*d m*)))
⟨*proof*⟩

abbreviation *epred* :: *enat* => *enat* **where** *epred* ≡ *co.epred*

lemma *epred-0* [*simp*]: *epred 0* = *0* ⟨*proof*⟩
lemma *epred-eSuc* [*simp*]: *epred* (*eSuc n*) = *n* ⟨*proof*⟩
declare *co.enat.collapse*[*simp*]
lemma *epred-conv-minus*: *epred n* = *n* − *1*
⟨*proof*⟩

1.2 Corecursion for *enat*

lemma *case-enat-numeral* [*simp*]: *case-enat f i* (*numeral v*) = (*let n* = *numeral v in f n*)
⟨*proof*⟩

lemma *case-enat-0* [*simp*]: *case-enat f i 0 = f 0*
⟨*proof*⟩

lemma [*simp*]:
shows *max-eSuc-eSuc*: *max (eSuc n) (eSuc m) = eSuc (max n m)*
and *min-eSuc-eSuc*: *min (eSuc n) (eSuc m) = eSuc (min n m)*
⟨*proof*⟩

definition *epred-numeral* :: *num ⇒ enat*
where [*code del*]: *epred-numeral = enat ∘ pred-numeral*

lemma *numeral-eq-eSuc*: *numeral k = eSuc (epred-numeral k)*
⟨*proof*⟩

lemma *epred-numeral-simps* [*simp*]:
epred-numeral num.One = 0
epred-numeral (num.Bit0 k) = numeral (Num.BitM k)
epred-numeral (num.Bit1 k) = numeral (num.Bit0 k)
⟨*proof*⟩

lemma [*simp*]:
shows *eq-numeral-eSuc*: *numeral k = eSuc n ⟷ epred-numeral k = n*
and *Suc-eq-numeral*: *eSuc n = numeral k ⟷ n = epred-numeral k*
and *less-numeral-Suc*: *numeral k < eSuc n ⟷ epred-numeral k < n*
and *less-eSuc-numeral*: *eSuc n < numeral k ⟷ n < epred-numeral k*
and *le-numeral-eSuc*: *numeral k ≤ eSuc n ⟷ epred-numeral k ≤ n*
and *le-eSuc-numeral*: *eSuc n ≤ numeral k ⟷ n ≤ epred-numeral k*
and *diff-eSuc-numeral*: *eSuc n − numeral k = n − epred-numeral k*
and *diff-numeral-eSuc*: *numeral k − eSuc n = epred-numeral k − n*
and *max-eSuc-numeral*: *max (eSuc n) (numeral k) = eSuc (max n (epred-numeral k))*
and *max-numeral-eSuc*: *max (numeral k) (eSuc n) = eSuc (max (epred-numeral k) n)*
and *min-eSuc-numeral*: *min (eSuc n) (numeral k) = eSuc (min n (epred-numeral k))*
and *min-numeral-eSuc*: *min (numeral k) (eSuc n) = eSuc (min (epred-numeral k) n)*
⟨*proof*⟩

lemma *enat-cocase-numeral* [*simp*]:
co.case-enat a f (numeral v) = (let pv = epred-numeral v in f pv)
⟨*proof*⟩

lemma *enat-cocase-add-eq-if* [*simp*]:
co.case-enat a f ((numeral v) + n) = (let pv = epred-numeral v in f (pv + n))
⟨*proof*⟩

lemma [*simp*]:
  shows *epred-1*: *epred 1 = 0*
  and *epred-numeral*: *epred* (*numeral i*) = *epred-numeral i*
  and *epred-Infty*: *epred* $\infty = \infty$
  and *epred-enat*: *epred* (*enat m*) = *enat* (*m − 1*)
⟨*proof*⟩

lemmas *epred-simps* = *epred-0 epred-1 epred-numeral epred-eSuc epred-Infty epred-enat*

lemma *epred-iadd1*: $a \neq 0 \Longrightarrow$ *epred* $(a + b)$ = *epred a* + *b*
⟨*proof*⟩

lemma *epred-min* [*simp*]: *epred* (*min a b*) = *min* (*epred a*) (*epred b*)
⟨*proof*⟩

lemma *epred-le-epredI*: $n \leq m \Longrightarrow$ *epred n* ≤ *epred m*
⟨*proof*⟩

lemma *epred-minus-epred* [*simp*]:
  $m \neq 0 \Longrightarrow$ *epred n* − *epred m* = *n* − *m*
⟨*proof*⟩

lemma *eSuc-epred*: $n \neq 0 \Longrightarrow$ *eSuc* (*epred n*) = *n*
⟨*proof*⟩

lemma *epred-inject*: $[\![x \neq 0; y \neq 0]\!] \Longrightarrow$ *epred x* = *epred y* $\longleftrightarrow$ *x* = *y*
⟨*proof*⟩

lemma *monotone-fun-eSuc*[*partial-function-mono*]:
    monotone (*fun-ord* ($\lambda y\ x.\ x \leq y$)) ($\lambda y\ x.\ x \leq y$) ($\lambda f.$ *eSuc* (*f x*))
  ⟨*proof*⟩

partial-function (*gfp*) *enat-unfold* :: (*'a* ⇒ *bool*) ⇒ (*'a* ⇒ *'a*) ⇒ *'a* ⇒ *enat* **where**
  enat-unfold [*code*, *nitpick-simp*]:
  enat-unfold stop next a = (*if stop a then 0 else eSuc* (*enat-unfold stop next* (*next a*)))

lemma *enat-unfold-stop* [*simp*]: *stop a* $\Longrightarrow$ *enat-unfold stop next a* = *0*
⟨*proof*⟩

lemma *enat-unfold-next*: ¬ *stop a* $\Longrightarrow$ *enat-unfold stop next a* = *eSuc* (*enat-unfold stop next* (*next a*))
⟨*proof*⟩

lemma *enat-unfold-eq-0* [*simp*]:
  enat-unfold stop next a = *0* $\longleftrightarrow$ *stop a*
⟨*proof*⟩

lemma *epred-enat-unfold* [*simp*]:

epred (*enat-unfold stop next a*) = (*if stop a then 0 else enat-unfold stop next* (*next a*))
⟨*proof*⟩

lemma *epred-max*: *epred* (*max x y*) = *max* (*epred x*) (*epred y*)
⟨*proof*⟩

lemma *epred-Max*:
assumes *finite A A* ≠ {}
shows *epred* (*Max A*) = *Max* (*epred ' A*)
⟨*proof*⟩

lemma *finite-imageD2*: ⟦ *finite* (*f ' A*); *inj-on f* (*A* − *B*); *finite B* ⟧ ⟹ *finite A*
⟨*proof*⟩

lemma *epred-Sup*: *epred* (*Sup A*) = *Sup* (*epred ' A*)
⟨*proof*⟩

1.3 Less as greatest fixpoint

coinductive-set *Le-enat* :: (*enat* × *enat*) *set*
where
Le-enat-zero: (*0*, *n*) ∈ *Le-enat*
| *Le-enat-add*: ⟦ (*m*, *n*) ∈ *Le-enat*; *k* ≠ *0* ⟧ ⟹ (*eSuc m*, *n* + *k*) ∈ *Le-enat*

lemma *ile-into-Le-enat*:
m ≤ *n* ⟹ (*m*, *n*) ∈ *Le-enat*
⟨*proof*⟩

lemma *Le-enat-imp-ile-enat-k*:
(*m*, *n*) ∈ *Le-enat* ⟹ *n* < *enat l* ⟹ *m* < *enat l*
⟨*proof*⟩

lemma *enat-less-imp-le*:
assumes *k*: !!*k*. *n* < *enat k* ⟹ *m* < *enat k*
shows *m* ≤ *n*
⟨*proof*⟩

lemma *Le-enat-imp-ile*:
(*m*, *n*) ∈ *Le-enat* ⟹ *m* ≤ *n*
⟨*proof*⟩

lemma *Le-enat-eq-ile*:
(*m*, *n*) ∈ *Le-enat* ⟷ *m* ≤ *n*
⟨*proof*⟩

lemma *enat-leI* [*consumes 1*, *case-names Leenat*, *case-conclusion Leenat zero eSuc*]:
assumes *major*: (*m*, *n*) ∈ *X*

and *step*:
$\bigwedge m\ n.\ (m,\ n) \in X$
$\Longrightarrow m = 0 \vee (\exists\, m'\ n'\ k.\ m = eSuc\ m' \wedge n = n' + enat\ k \wedge k \neq 0 \wedge$
$((m',\ n') \in X \vee m' \leq n'))$
shows $m \leq n$
⟨*proof*⟩

lemma *enat-le-coinduct*[*consumes 1*, *case-names le*, *case-conclusion le 0 eSuc*]:
assumes *P*: $P\ m\ n$
and *step*:
$\bigwedge m\ n.\ P\ m\ n$
$\Longrightarrow (n = 0 \longrightarrow m = 0) \wedge$
$(m \neq 0 \longrightarrow n \neq 0 \longrightarrow (\exists\, k\ n'.\ P\ (epred\ m)\ n' \wedge epred\ n = n' + k) \vee epred\ m \leq epred\ n)$
shows $m \leq n$
⟨*proof*⟩

1.4 Equality as greatest fixpoint

lemma *enat-equalityI* [*consumes 1*, *case-names Eq-enat*,
case-conclusion Eq-enat zero eSuc]:
assumes *major*: $(m,\ n) \in X$
and *step*:
$\bigwedge m\ n.\ (m,\ n) \in X$
$\Longrightarrow m = 0 \wedge n = 0 \vee (\exists\, m'\ n'.\ m = eSuc\ m' \wedge n = eSuc\ n' \wedge ((m',\ n') \in X \vee m' = n'))$
shows $m = n$
⟨*proof*⟩

lemma *enat-coinduct* [*consumes 1*, *case-names Eq-enat*, *case-conclusion Eq-enat zero eSuc*]:
assumes *major*: $P\ m\ n$
and *step*: $\bigwedge m\ n.\ P\ m\ n$
$\Longrightarrow (m = 0 \longleftrightarrow n = 0) \wedge$
$(m \neq 0 \longrightarrow n \neq 0 \longrightarrow P\ (epred\ m)\ (epred\ n) \vee epred\ m = epred\ n)$
shows $m = n$
⟨*proof*⟩

lemma *enat-coinduct2* [*consumes 1*, *case-names zero eSuc*]:
$[\![\ P\ m\ n;\ \bigwedge m\ n.\ P\ m\ n \Longrightarrow m = 0 \longleftrightarrow n = 0;$
$\bigwedge m\ n.\ [\![\ P\ m\ n;\ m \neq 0;\ n \neq 0\ ]\!] \Longrightarrow P\ (epred\ m)\ (epred\ n) \vee epred\ m = epred\ n\ ]\!]$
$\Longrightarrow m = n$
⟨*proof*⟩

1.5 Uniqueness of corecursion

lemma *enat-unfold-unique*:
assumes *h*: $!!x.\ h\ x = (if\ stop\ x\ then\ 0\ else\ eSuc\ (h\ (next\ x)))$
shows $h\ x = enat\text{-}unfold\ stop\ next\ x$

⟨*proof*⟩

1.6 Setup for partial_function

lemma *enat-diff-cancel-left*: ⟦ $m \leq x$; $m \leq y$ ⟧ ⟹ $x - m = y - m \longleftrightarrow x = (y :: enat)$
⟨*proof*⟩

lemma *finite-lessThan-enatI*:
assumes $m \neq \infty$
shows *finite* $\{..<m :: enat\}$
⟨*proof*⟩

lemma *infinite-lessThan-infty*: ¬ *finite* $\{..<\infty :: enat\}$
⟨*proof*⟩

lemma *finite-lessThan-enat-iff*:
finite $\{..<m :: enat\} \longleftrightarrow m \neq \infty$
⟨*proof*⟩

lemma *enat-minus-mono1*: $x \leq y \Longrightarrow x - m \leq y - (m :: enat)$
⟨*proof*⟩

lemma *max-enat-minus1*: $max\ n\ m - k = max\ (n - k)\ (m - k :: enat)$
⟨*proof*⟩

lemma *Max-enat-minus1*:
assumes *finite* A $A \neq \{\}$
shows $Max\ A - m = Max\ ((\lambda n :: enat.\ n - m)\ `\ A)$
⟨*proof*⟩

lemma *Sup-enat-minus1*:
assumes $m \neq \infty$
shows $\bigsqcup A - m = \bigsqcup ((\lambda n :: enat.\ n - m)\ `\ A)$
⟨*proof*⟩

lemma *Sup-image-eadd1*:
assumes $Y \neq \{\}$
shows $Sup\ ((\lambda y :: enat.\ y+x)\ `\ Y) = Sup\ Y + x$
⟨*proof*⟩

lemma *Sup-image-eadd2*:
$Y \neq \{\} \Longrightarrow Sup\ ((\lambda y :: enat.\ x + y)\ `\ Y) = x + Sup\ Y$
⟨*proof*⟩

lemma *mono2mono-eSuc* [*THEN lfp.mono2mono*, *cont-intro*, *simp*]:
shows *monotone-eSuc*: $monotone\ (\leq)\ (\leq)\ eSuc$
⟨*proof*⟩

lemma *mcont2mcont-eSuc* [*THEN lfp.mcont2mcont*, *cont-intro*, *simp*]:
shows *mcont-eSuc*: *mcont Sup* (≤) *Sup* (≤) *eSuc*
⟨*proof*⟩

lemma *mono2mono-epred* [*THEN lfp.mono2mono*, *cont-intro*, *simp*]:
shows *monotone-epred*: *monotone* (≤) (≤) *epred*
⟨*proof*⟩

lemma *mcont2mcont-epred* [*THEN lfp.mcont2mcont*, *cont-intro*, *simp*]:
shows *mcont-epred*: *mcont Sup* (≤) *Sup* (≤) *epred*
⟨*proof*⟩

lemma *enat-cocase-mono* [*partial-function-mono*, *cont-intro*]:
⟦ *monotone orda ordb zero*; ⋀*n*. *monotone orda ordb* (λ*f*. *esuc f n*) ⟧
⟹ *monotone orda ordb* (λ*f*. *co.case-enat* (*zero f*) (*esuc f*) *x*)
⟨*proof*⟩

lemma *enat-cocase-mcont* [*cont-intro*, *simp*]:
⟦ *mcont luba orda lubb ordb zero*; ⋀*n*. *mcont luba orda lubb ordb* (λ*f*. *esuc f n*) ⟧
⟹ *mcont luba orda lubb ordb* (λ*f*. *co.case-enat* (*zero f*) (*esuc f*) *x*)
⟨*proof*⟩

lemma *eSuc-mono* [*partial-function-mono*]:
monotone (*fun-ord* (≤)) (≤) *f* ⟹ *monotone* (*fun-ord* (≤)) (≤) (λ*x*. *eSuc* (*f x*))
⟨*proof*⟩

lemma *mono2mono-enat-minus1* [*THEN lfp.mono2mono*, *cont-intro*, *simp*]:
shows *monotone-enat-minus1*: *monotone* (≤) (≤) (λ*n*. *n* − *m* :: *enat*)
⟨*proof*⟩

lemma *mcont2mcont-enat-minus* [*THEN lfp.mcont2mcont*, *cont-intro*, *simp*]:
shows *mcont-enat-minus*: *m* ≠ ∞ ⟹ *mcont Sup* (≤) *Sup* (≤) (λ*n*. *n* − *m* :: *enat*)
⟨*proof*⟩

lemma *monotone-eadd1*: *monotone* (≤) (≤) (λ*x*. *x* + *y* :: *enat*)
⟨*proof*⟩

lemma *monotone-eadd2*: *monotone* (≤) (≤) (λ*y*. *x* + *y* :: *enat*)
⟨*proof*⟩

lemma *mono2mono-eadd*[*THEN lfp.mono2mono2*, *cont-intro*, *simp*]:
shows *monotone-eadd*: *monotone* (*rel-prod* (≤) (≤)) (≤) (λ(*x*, *y*). *x* + *y* :: *enat*)
⟨*proof*⟩

lemma *mcont-eadd2*: *mcont Sup* (≤) *Sup* (≤) (λ*y*. *x* + *y* :: *enat*)
⟨*proof*⟩

lemma *mcont-eadd1*: *mcont Sup* (≤) *Sup* (≤) (λ*x*. *x* + *y* :: *enat*)

⟨*proof*⟩

lemma *mcont2mcont-eadd* [*cont-intro*, *simp*]:
⟦ *mcont lub ord Sup* (≤) (λ*x*. *f x*);
mcont lub ord Sup (≤) (λ*x*. *g x*) ⟧
⟹ *mcont lub ord Sup* (≤) (λ*x*. *f x* + *g x* :: *enat*)
⟨*proof*⟩

lemma *eadd-partial-function-mono* [*partial-function-mono*]:
⟦ *monotone* (*fun-ord* (≤)) (≤) *f*; *monotone* (*fun-ord* (≤)) (≤) *g* ⟧
⟹ *monotone* (*fun-ord* (≤)) (≤) (λ*x*. *f x* + *g x* :: *enat*)
⟨*proof*⟩

lemma *monotone-max-enat1*: *monotone* (≤) (≤) (λ*x*. *max x y* :: *enat*)
⟨*proof*⟩

lemma *monotone-max-enat2*: *monotone* (≤) (≤) (λ*y*. *max x y* :: *enat*)
⟨*proof*⟩

lemma *mono2mono-max-enat*[*THEN lfp.mono2mono2*, *cont-intro*, *simp*]:
shows *monotone-max-enat*: *monotone* (*rel-prod* (≤) (≤)) (≤) (λ(*x*, *y*). *max x y*
:: *enat*)
⟨*proof*⟩

lemma *max-Sup-enat2*:
assumes *Y* ≠ {}
shows *max x* (*Sup Y*) = *Sup* ((λ*y* :: *enat*. *max x y*) ‘ *Y*)
⟨*proof*⟩

lemma *max-Sup-enat1*:
Y ≠ {} ⟹ *max* (*Sup Y*) *x* = *Sup* ((λ*y* :: *enat*. *max y x*) ‘ *Y*)
⟨*proof*⟩

lemma *mcont-max-enat1*: *mcont Sup* (≤) *Sup* (≤) (λ*x*. *max x y* :: *enat*)
⟨*proof*⟩

lemma *mcont-max-enat2*: *mcont Sup* (≤) *Sup* (≤) (λ*y*. *max x y* :: *enat*)
⟨*proof*⟩

lemma *mcont2mcont-max-enat* [*cont-intro*, *simp*]:
⟦ *mcont lub ord Sup* (≤) (λ*x*. *f x*);
mcont lub ord Sup (≤) (λ*x*. *g x*) ⟧
⟹ *mcont lub ord Sup* (≤) (λ*x*. *max* (*f x*) (*g x*) :: *enat*)
⟨*proof*⟩

lemma *max-enat-partial-function-mono* [*partial-function-mono*]:
⟦ *monotone* (*fun-ord* (≤)) (≤) *f*; *monotone* (*fun-ord* (≤)) (≤) *g* ⟧
⟹ *monotone* (*fun-ord* (≤)) (≤) (λ*x*. *max* (*f x*) (*g x*) :: *enat*)
⟨*proof*⟩

lemma *chain-epredI*:
Complete-Partial-Order.chain (≤) *Y*
⟹ *Complete-Partial-Order.chain* (≤) (*epred* ‘ (*Y* ∩ {*x*. *x* ≠ *0*}))
⟨*proof*⟩

lemma *monotone-enat-le-case*:
fixes *bot*
assumes *mono*: *monotone* (≤) *ord* (λ*x*. *f x* (*eSuc x*))
and *ord*: ⋀*x*. *ord bot* (*f x* (*eSuc x*))
and *bot*: *ord bot bot*
shows *monotone* (≤) *ord* (λ*x*. *case x of 0* ⇒ *bot* | *eSuc x′* ⇒ *f x′ x*)
⟨*proof*⟩

lemma *mcont-enat-le-case*:
fixes *bot*
assumes *ccpo*: *class.ccpo lub ord* (*mk-less ord*)
and *mcont*: *mcont Sup* (≤) *lub ord* (λ*x*. *f x* (*eSuc x*))
and *ord*: ⋀*x*. *ord bot* (*f x* (*eSuc x*))
shows *mcont Sup* (≤) *lub ord* (λ*x*. *case x of 0* ⇒ *bot* | *eSuc x′* ⇒ *f x′ x*)
⟨*proof*⟩

1.7 Misc.

lemma *enat-add-mono* [*simp*]:
enat x + *y* < *enat x* + *z* ⟷ *y* < *z*
⟨*proof*⟩

lemma *enat-add1-eq* [*simp*]: *enat x* + *y* = *enat x* + *z* ⟷ *y* = *z*
⟨*proof*⟩

lemma *enat-add2-eq* [*simp*]: *y* + *enat x* = *z* + *enat x* ⟷ *y* = *z*
⟨*proof*⟩

lemma *enat-less-enat-plusI*: *x* < *y* ⟹ *enat x* < *enat y* + *z*
⟨*proof*⟩

lemma *enat-less-enat-plusI2*:
enat y < *z* ⟹ *enat* (*x* + *y*) < *enat x* + *z*
⟨*proof*⟩

lemma *min-enat1-conv-enat*: ⋀*a b*. *min* (*enat a*) *b* = *enat* (*case b of enat b′* ⇒ *min a b′* | ∞ ⇒ *a*)
and *min-enat2-conv-enat*: ⋀*a b*. *min a* (*enat b*) = *enat* (*case a of enat a′* ⇒ *min a′ b* | ∞ ⇒ *b*)
⟨*proof*⟩

lemma *eSuc-le-iff*: *eSuc x* ≤ *y* ⟷ (∃ *y′*. *y* = *eSuc y′* ∧ *x* ≤ *y′*)
⟨*proof*⟩

lemma *eSuc-eq-infinity-iff*: $eSuc\ n = \infty \longleftrightarrow n = \infty$
⟨*proof*⟩

lemma *infinity-eq-eSuc-iff*: $\infty = eSuc\ n \longleftrightarrow n = \infty$
⟨*proof*⟩

lemma *enat-cocase-inf*: $(case\ \infty\ of\ 0 \Rightarrow a \mid eSuc\ b \Rightarrow f\ b) = f\ \infty$
⟨*proof*⟩

lemma *eSuc-Inf*: $eSuc\ (Inf\ A) = Inf\ (eSuc\ `\ A)$
⟨*proof*⟩

end

2 Coinductive lists and their operations

theory *Coinductive-List*
imports
HOL−Library.Infinite-Set
HOL−Library.Sublist
HOL−Library.Simps-Case-Conv
Coinductive-Nat
begin

2.1 Auxiliary lemmata

lemma *funpow-Suc-conv* [*simp*]: $(Suc\ ^{\frown\frown}\ n)\ m = m + n$
⟨*proof*⟩

lemma *wlog-linorder-le* [*consumes 0*, *case-names le symmetry*]:
assumes *le*: $\bigwedge a\ b :: 'a :: linorder.\ a \le b \Longrightarrow P\ a\ b$
and *sym*: $P\ b\ a \Longrightarrow P\ a\ b$
shows $P\ a\ b$
⟨*proof*⟩

2.2 Type definition

codatatype (*lset*: $'a$) *llist* =
lnull: *LNil*
| *LCons* (*lhd*: $'a$) (*ltl*: $'a$ *llist*)
for
map: *lmap*
rel: *llist-all2*
where
lhd LNil = *undefined*
| *ltl LNil* = *LNil*

Coiterator setup.

primcorec *unfold-llist* :: $('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'b$ *llist*
where
$p\ a \Longrightarrow$ *unfold-llist p g21 g22 a* = *LNil* |
- $\Longrightarrow$ *unfold-llist p g21 g22 a* = *LCons* (*g21 a*) (*unfold-llist p g21 g22* (*g22 a*))

declare
unfold-llist.ctr(*1*) [*simp*]
llist.corec(*1*) [*simp*]

The following setup should be done by the BNF package.

congruence rule

declare *llist.map-cong* [*cong*]

Code generator setup

lemma *corec-llist-never-stop*: *corec-llist IS-LNIL LHD* (λ-. *False*) *MORE LTL x* = *unfold-llist IS-LNIL LHD LTL x*
⟨*proof*⟩

lemmas about generated constants

lemma *eq-LConsD*: $xs = LCons\ y\ ys \Longrightarrow xs \neq LNil \wedge lhd\ xs = y \wedge ltl\ xs = ys$
⟨*proof*⟩

lemma
shows *LNil-eq-lmap*: $LNil = lmap\ f\ xs \longleftrightarrow xs = LNil$
and *lmap-eq-LNil*: $lmap\ f\ xs = LNil \longleftrightarrow xs = LNil$
⟨*proof*⟩

declare *llist.map-sel*(*1*)[*simp*]

lemma *ltl-lmap*[*simp*]: $ltl\ (lmap\ f\ xs) = lmap\ f\ (ltl\ xs)$
⟨*proof*⟩

declare *llist.map-ident*[*simp*]

lemma *lmap-eq-LCons-conv*:
$lmap\ f\ xs = LCons\ y\ ys \longleftrightarrow$
$(\exists x\ xs'.\ xs = LCons\ x\ xs' \wedge y = f\ x \wedge ys = lmap\ f\ xs')$
⟨*proof*⟩

lemma *lmap-conv-unfold-llist*:
$lmap\ f = unfold\text{-}llist\ (\lambda xs.\ xs = LNil)\ (f \circ lhd)\ ltl$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *lmap-unfold-llist*:
$lmap\ f\ (unfold\text{-}llist\ IS\text{-}LNIL\ LHD\ LTL\ b) = unfold\text{-}llist\ IS\text{-}LNIL\ (f \circ LHD)\ LTL\ b$
⟨*proof*⟩

lemma *lmap-corec-llist*:
lmap f (*corec-llist IS-LNIL LHD endORmore TTL-end TTL-more b*) =
corec-llist IS-LNIL (*f* ∘ *LHD*) *endORmore* (*lmap f* ∘ *TTL-end*) *TTL-more b*
⟨*proof*⟩

lemma *unfold-llist-ltl-unroll*:
unfold-llist IS-LNIL LHD LTL (*LTL b*) = *unfold-llist* (*IS-LNIL* ∘ *LTL*) (*LHD* ∘ *LTL*) *LTL b*
⟨*proof*⟩

lemma *ltl-unfold-llist*:
ltl (*unfold-llist IS-LNIL LHD LTL a*) =
(*if IS-LNIL a then LNil else unfold-llist IS-LNIL LHD LTL* (*LTL a*))
⟨*proof*⟩

lemma *unfold-llist-eq-LCons* [*simp*]:
unfold-llist IS-LNIL LHD LTL b = *LCons x xs* ⟷
¬ *IS-LNIL b* ∧ *x* = *LHD b* ∧ *xs* = *unfold-llist IS-LNIL LHD LTL* (*LTL b*)
⟨*proof*⟩

lemma *unfold-llist-id* [*simp*]: *unfold-llist lnull lhd ltl xs* = *xs*
⟨*proof*⟩

lemma *lset-eq-empty* [*simp*]: *lset xs* = {} ⟷ *lnull xs*
⟨*proof*⟩

declare *llist.set-sel*(*1*)[*simp*]

lemma *lset-ltl*: *lset* (*ltl xs*) ⊆ *lset xs*
⟨*proof*⟩

lemma *in-lset-ltlD*: *x* ∈ *lset* (*ltl xs*) ⟹ *x* ∈ *lset xs*
⟨*proof*⟩

induction rules

theorem *llist-set-induct*[*consumes 1*, *case-names find step*]:
assumes *x* ∈ *lset xs* **and** ⋀*xs*. ¬ *lnull xs* ⟹ *P* (*lhd xs*) *xs*
and ⋀*xs y*. ⟦¬ *lnull xs*; *y* ∈ *lset* (*ltl xs*); *P y* (*ltl xs*)⟧ ⟹ *P y xs*
shows *P x xs*
⟨*proof*⟩

Test quickcheck setup

lemma ⋀*xs*. *xs* = *LNil*
quickcheck[*random*, *expect*=*counterexample*]
quickcheck[*exhaustive*, *expect*=*counterexample*]
⟨*proof*⟩

lemma *LCons x xs* = *LCons x xs*
quickcheck[*narrowing*, *expect*=*no-counterexample*]

⟨*proof*⟩

2.3 Properties of predefined functions

lemmas *lhd-LCons* = *llist.sel*(*1*)
lemmas *ltl-simps* = *llist.sel*(*2*,*3*)

lemmas *lhd-LCons-ltl* = *llist.collapse*(*2*)

lemma *lnull-ltlI* [*simp*]: *lnull xs* ⟹ *lnull* (*ltl xs*)
⟨*proof*⟩

lemma *neq-LNil-conv*: *xs* ≠ *LNil* ⟷ (∃ *x xs′*. *xs* = *LCons x xs′*)
⟨*proof*⟩

lemma *not-lnull-conv*: ¬ *lnull xs* ⟷ (∃ *x xs′*. *xs* = *LCons x xs′*)
⟨*proof*⟩

lemma *lset-LCons*:
lset (*LCons x xs*) = *insert x* (*lset xs*)
⟨*proof*⟩

lemma *lset-intros*:
x ∈ *lset* (*LCons x xs*)
x ∈ *lset xs* ⟹ *x* ∈ *lset* (*LCons x′ xs*)
⟨*proof*⟩

lemma *lset-cases* [*elim?*]:
assumes *x* ∈ *lset xs*
obtains *xs′* **where** *xs* = *LCons x xs′*
| *x′ xs′* **where** *xs* = *LCons x′ xs′ x* ∈ *lset xs′*
⟨*proof*⟩

lemma *lset-induct′* [*consumes 1*, *case-names find step*]:
assumes *major*: *x* ∈ *lset xs*
and *1*: ⋀*xs*. *P* (*LCons x xs*)
and *2*: ⋀*x′ xs*. ⟦ *x* ∈ *lset xs*; *P xs* ⟧ ⟹ *P* (*LCons x′ xs*)
shows *P xs*
⟨*proof*⟩

lemma *lset-induct* [*consumes 1*, *case-names find step*, *induct set*: *lset*]:
assumes *major*: *x* ∈ *lset xs*
and *find*: ⋀*xs*. *P* (*LCons x xs*)
and *step*: ⋀*x′ xs*. ⟦ *x* ∈ *lset xs*; *x* ≠ *x′*; *P xs* ⟧ ⟹ *P* (*LCons x′ xs*)
shows *P xs*
⟨*proof*⟩

lemmas *lset-LNil* = *llist.set*(*1*)

lemma *lset-lnull*: *lnull xs* $\Longrightarrow$ *lset xs* = {}
⟨*proof*⟩

Alternative definition of *lset* for nitpick

inductive *lsetp* :: *'a llist* ⇒ *'a* ⇒ *bool*
where
  lsetp (*LCons x xs*) *x*
| *lsetp xs x* $\Longrightarrow$ *lsetp* (*LCons x' xs*) *x*

lemma *lset-into-lsetp*:
  x ∈ *lset xs* $\Longrightarrow$ *lsetp xs x*
⟨*proof*⟩

lemma *lsetp-into-lset*:
  lsetp xs x $\Longrightarrow$ *x* ∈ *lset xs*
⟨*proof*⟩

lemma *lset-eq-lsetp* [*nitpick-unfold*]:
  lset xs = {*x*. *lsetp xs x*}
⟨*proof*⟩

hide-const (**open**) *lsetp*
hide-fact (**open**) *lsetp.intros lsetp.cases lsetp.induct lset-into-lsetp lset-eq-lsetp*

code setup for *lset*

definition *gen-lset* :: *'a set* ⇒ *'a llist* ⇒ *'a set*
where *gen-lset A xs* = *A* ∪ *lset xs*

lemma *gen-lset-code* [*code*]:
  gen-lset A LNil = *A*
  gen-lset A (*LCons x xs*) = *gen-lset* (*insert x A*) *xs*
⟨*proof*⟩

lemma *lset-code* [*code*]:
  lset = *gen-lset* {}
⟨*proof*⟩

definition *lmember* :: *'a* ⇒ *'a llist* ⇒ *bool*
where *lmember x xs* $\longleftrightarrow$ *x* ∈ *lset xs*

lemma *lmember-code* [*code*]:
  lmember x LNil $\longleftrightarrow$ *False*
  lmember x (*LCons y ys*) $\longleftrightarrow$ *x* = *y* ∨ *lmember x ys*
⟨*proof*⟩

lemma *lset-lmember* [*code-unfold*]:
  x ∈ *lset xs* $\longleftrightarrow$ *lmember x xs*
⟨*proof*⟩

lemmas *lset-lmap* [*simp*] = *llist.set-map*

2.4 The subset of finite lazy lists *lfinite*

inductive *lfinite* :: *'a llist* ⇒ *bool*
where
  lfinite-LNil: *lfinite LNil*
| *lfinite-LConsI*: *lfinite xs* ⟹ *lfinite* (*LCons x xs*)

declare *lfinite-LNil* [*iff*]

lemma *lnull-imp-lfinite* [*simp*]: *lnull xs* ⟹ *lfinite xs*
⟨*proof*⟩

lemma *lfinite-LCons* [*simp*]: *lfinite* (*LCons x xs*) = *lfinite xs*
⟨*proof*⟩

lemma *lfinite-ltl* [*simp*]: *lfinite* (*ltl xs*) = *lfinite xs*
⟨*proof*⟩

lemma *lfinite-code* [*code*]:
  lfinite LNil = *True*
  lfinite (*LCons x xs*) = *lfinite xs*
⟨*proof*⟩

lemma *lfinite-induct* [*consumes 1*, *case-names LNil LCons*]:
  assumes *lfinite*: *lfinite xs*
  and *LNil*: ⋀*xs*. *lnull xs* ⟹ *P xs*
  and *LCons*: ⋀*xs*. ⟦ *lfinite xs*; ¬ *lnull xs*; *P* (*ltl xs*) ⟧ ⟹ *P xs*
  shows *P xs*
⟨*proof*⟩

lemma *lfinite-imp-finite-lset*:
  assumes *lfinite xs*
  shows *finite* (*lset xs*)
⟨*proof*⟩

2.5 Concatenating two lists: *lappend*

primcorec *lappend* :: *'a llist* ⇒ *'a llist* ⇒ *'a llist*
where
  lappend xs ys = (*case xs of LNil* ⇒ *ys* | *LCons x xs'* ⇒ *LCons x* (*lappend xs' ys*))

simps-of-case *lappend-code* [*code*, *simp*, *nitpick-simp*]: *lappend.code*

lemmas *lappend-LNil-LNil* = *lappend-code*(*1*)[**where** *ys* = *LNil*]

lemma *lappend-simps* [*simp*]:
  shows *lhd-lappend*: *lhd* (*lappend xs ys*) = (*if lnull xs then lhd ys else lhd xs*)

and *ltl-lappend*: *ltl* (*lappend xs ys*) = (*if lnull xs then ltl ys else lappend* (*ltl xs*) *ys*)
⟨*proof*⟩

lemma *lnull-lappend* [*simp*]:
lnull (*lappend xs ys*) ⟷ *lnull xs* ∧ *lnull ys*
⟨*proof*⟩

lemma *lappend-eq-LNil-iff*:
lappend xs ys = *LNil* ⟷ *xs* = *LNil* ∧ *ys* = *LNil*
⟨*proof*⟩

lemma *LNil-eq-lappend-iff*:
LNil = *lappend xs ys* ⟷ *xs* = *LNil* ∧ *ys* = *LNil*
⟨*proof*⟩

lemma *lappend-LNil2* [*simp*]: *lappend xs LNil* = *xs*
⟨*proof*⟩

lemma shows *lappend-lnull1*: *lnull xs* ⟹ *lappend xs ys* = *ys*
and *lappend-lnull2*: *lnull ys* ⟹ *lappend xs ys* = *xs*
⟨*proof*⟩

lemma *lappend-assoc*: *lappend* (*lappend xs ys*) *zs* = *lappend xs* (*lappend ys zs*)
⟨*proof*⟩

lemma *lmap-lappend-distrib*:
lmap f (*lappend xs ys*) = *lappend* (*lmap f xs*) (*lmap f ys*)
⟨*proof*⟩

lemma *lappend-snocL1-conv-LCons2*:
lappend (*lappend xs* (*LCons y LNil*)) *ys* = *lappend xs* (*LCons y ys*)
⟨*proof*⟩

lemma *lappend-ltl*: ¬ *lnull xs* ⟹ *lappend* (*ltl xs*) *ys* = *ltl* (*lappend xs ys*)
⟨*proof*⟩

lemma *lfinite-lappend* [*simp*]:
lfinite (*lappend xs ys*) ⟷ *lfinite xs* ∧ *lfinite ys*
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *lappend-inf*: ¬ *lfinite xs* ⟹ *lappend xs ys* = *xs*
⟨*proof*⟩

lemma *lfinite-lmap* [*simp*]:
lfinite (*lmap f xs*) = *lfinite xs*
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *lset-lappend-lfinite* [*simp*]:
lfinite xs $\Longrightarrow$ *lset* (*lappend xs ys*) = *lset xs* $\cup$ *lset ys*
⟨*proof*⟩

lemma *lset-lappend*: *lset* (*lappend xs ys*) $\subseteq$ *lset xs* $\cup$ *lset ys*
⟨*proof*⟩

lemma *lset-lappend1*: *lset xs* $\subseteq$ *lset* (*lappend xs ys*)
⟨*proof*⟩

lemma *lset-lappend-conv*: *lset* (*lappend xs ys*) = (*if lfinite xs then lset xs* $\cup$ *lset ys else lset xs*)
⟨*proof*⟩

lemma *in-lset-lappend-iff*: *x* $\in$ *lset* (*lappend xs ys*) $\longleftrightarrow$ *x* $\in$ *lset xs* $\vee$ *lfinite xs* $\wedge$ *x* $\in$ *lset ys*
⟨*proof*⟩

lemma *split-llist-first*:
assumes *x* $\in$ *lset xs*
shows $\exists$ *ys zs*. *xs* = *lappend ys* (*LCons x zs*) $\wedge$ *lfinite ys* $\wedge$ *x* $\notin$ *lset ys*
⟨*proof*⟩

lemma *split-llist*: *x* $\in$ *lset xs* $\Longrightarrow$ $\exists$ *ys zs*. *xs* = *lappend ys* (*LCons x zs*) $\wedge$ *lfinite ys*
⟨*proof*⟩

2.6 The prefix ordering on lazy lists: *lprefix*

coinductive *lprefix* :: *'a llist* $\Rightarrow$ *'a llist* $\Rightarrow$ *bool* (**infix** $\sqsubseteq$ *65*)
where
LNil-lprefix [*simp*, *intro*!]: *LNil* $\sqsubseteq$ *xs*
| *Le-LCons*: *xs* $\sqsubseteq$ *ys* $\Longrightarrow$ *LCons x xs* $\sqsubseteq$ *LCons x ys*

lemma *lprefixI* [*consumes 1*, *case-names lprefix*,
case-conclusion lprefix LeLNil LeLCons]:
assumes *major*: (*xs*, *ys*) $\in$ *X*
and *step*:
$\bigwedge$*xs ys*. (*xs*, *ys*) $\in$ *X*
$\Longrightarrow$ *lnull xs* $\vee$ ($\exists$ *x xs' ys'*. *xs* = *LCons x xs'* $\wedge$ *ys* = *LCons x ys'* $\wedge$
((*xs'*, *ys'*) $\in$ *X* $\vee$ *xs'* $\sqsubseteq$ *ys'*))
shows *xs* $\sqsubseteq$ *ys*
⟨*proof*⟩

lemma *lprefix-coinduct* [*consumes 1*, *case-names lprefix*, *case-conclusion lprefix LNil LCons*, *coinduct pred*: *lprefix*]:
assumes *major*: *P xs ys*
and *step*: $\bigwedge$*xs ys*. *P xs ys*
$\Longrightarrow$ (*lnull ys* $\longrightarrow$ *lnull xs*) $\wedge$

$(\neg\ lnull\ xs \longrightarrow \neg\ lnull\ ys \longrightarrow lhd\ xs = lhd\ ys \wedge (P\ (ltl\ xs)\ (ltl\ ys) \vee ltl\ xs \sqsubseteq ltl\ ys))$
shows $xs \sqsubseteq ys$
⟨*proof*⟩

lemma *lprefix-refl* [*intro*, *simp*]: $xs \sqsubseteq xs$
⟨*proof*⟩

lemma *lprefix-LNil* [*simp*]: $xs \sqsubseteq LNil \longleftrightarrow lnull\ xs$
⟨*proof*⟩

lemma *lprefix-lnull*: $lnull\ ys \Longrightarrow xs \sqsubseteq ys \longleftrightarrow lnull\ xs$
⟨*proof*⟩

lemma *lnull-lprefix*: $lnull\ xs \Longrightarrow lprefix\ xs\ ys$
⟨*proof*⟩

lemma *lprefix-LCons-conv*:
$xs \sqsubseteq LCons\ y\ ys \longleftrightarrow$
$xs = LNil \vee (\exists xs'.\ xs = LCons\ y\ xs' \wedge xs' \sqsubseteq ys)$
⟨*proof*⟩

lemma *LCons-lprefix-LCons* [*simp*]:
$LCons\ x\ xs \sqsubseteq LCons\ y\ ys \longleftrightarrow x = y \wedge xs \sqsubseteq ys$
⟨*proof*⟩

lemma *LCons-lprefix-conv*:
$LCons\ x\ xs \sqsubseteq ys \longleftrightarrow (\exists ys'.\ ys = LCons\ x\ ys' \wedge xs \sqsubseteq ys')$
⟨*proof*⟩

lemma *lprefix-ltlI*: $xs \sqsubseteq ys \Longrightarrow ltl\ xs \sqsubseteq ltl\ ys$
⟨*proof*⟩

lemma *lprefix-code* [*code*]:
$LNil \sqsubseteq ys \longleftrightarrow True$
$LCons\ x\ xs \sqsubseteq LNil \longleftrightarrow False$
$LCons\ x\ xs \sqsubseteq LCons\ y\ ys \longleftrightarrow x = y \wedge xs \sqsubseteq ys$
⟨*proof*⟩

lemma *lprefix-lhdD*: $[\![\ xs \sqsubseteq ys;\ \neg\ lnull\ xs\ ]\!] \Longrightarrow lhd\ xs = lhd\ ys$
⟨*proof*⟩

lemma *lprefix-lnullD*: $[\![\ xs \sqsubseteq ys;\ lnull\ ys\ ]\!] \Longrightarrow lnull\ xs$
⟨*proof*⟩

lemma *lprefix-not-lnullD*: $[\![\ xs \sqsubseteq ys;\ \neg\ lnull\ xs\ ]\!] \Longrightarrow \neg\ lnull\ ys$
⟨*proof*⟩

lemma *lprefix-expand*:

$(\neg\ lnull\ xs \Longrightarrow \neg\ lnull\ ys \wedge lhd\ xs = lhd\ ys \wedge ltl\ xs \sqsubseteq ltl\ ys) \Longrightarrow xs \sqsubseteq ys$
⟨*proof*⟩

lemma *lprefix-antisym*:
$[\![xs \sqsubseteq ys;\ ys \sqsubseteq xs]\!] \Longrightarrow xs = ys$
⟨*proof*⟩

lemma *lprefix-trans* [*trans*]:
$[\![xs \sqsubseteq ys;\ ys \sqsubseteq zs]\!] \Longrightarrow xs \sqsubseteq zs$
⟨*proof*⟩

lemma *preorder-lprefix* [*cont-intro*]:
class.preorder (⊑) (*mk-less* (⊑))
⟨*proof*⟩

lemma *lprefix-lsetD*:
assumes $xs \sqsubseteq ys$
shows $lset\ xs \subseteq lset\ ys$
⟨*proof*⟩

lemma *lprefix-lappend-sameI*:
assumes $xs \sqsubseteq ys$
shows $lappend\ zs\ xs \sqsubseteq lappend\ zs\ ys$
⟨*proof*⟩

lemma *not-lfinite-lprefix-conv-eq*:
assumes *nfin*: $\neg\ lfinite\ xs$
shows $xs \sqsubseteq ys \longleftrightarrow xs = ys$
⟨*proof*⟩

lemma *lprefix-lappend*: $xs \sqsubseteq lappend\ xs\ ys$
⟨*proof*⟩

lemma *lprefix-down-linear*:
assumes $xs \sqsubseteq zs\ ys \sqsubseteq zs$
shows $xs \sqsubseteq ys \vee ys \sqsubseteq xs$
⟨*proof*⟩

lemma *lprefix-lappend-same* [*simp*]:
$lappend\ xs\ ys \sqsubseteq lappend\ xs\ zs \longleftrightarrow (lfinite\ xs \longrightarrow ys \sqsubseteq zs)$
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

2.7 Setup for partial_function

primcorec *lSup* :: *'a llist set* ⇒ *'a llist*
where
lSup A =
(*if* ∀ *x*∈*A*. *lnull x then LNil*

else LCons (THE x. x ∈ lhd ‘ (A ∩ {xs. ¬ lnull xs})) (lSup (ltl ‘ (A ∩ {xs. ¬ lnull xs})))

declare *lSup.simps*[*simp del*]

lemma *lnull-lSup* [*simp*]: *lnull (lSup A) ⟷ (∀ x∈A. lnull x)*
⟨*proof*⟩

lemma *lhd-lSup* [*simp*]: *∃ x∈A. ¬ lnull x ⟹ lhd (lSup A) = (THE x. x ∈ lhd ‘ (A ∩ {xs. ¬ lnull xs}))*
⟨*proof*⟩

lemma *ltl-lSup* [*simp*]: *ltl (lSup A) = lSup (ltl ‘ (A ∩ {xs. ¬ lnull xs}))*
⟨*proof*⟩

lemma *lhd-lSup-eq*:
assumes *chain*: *Complete-Partial-Order.chain (⊑) Y*
shows *⟦ xs ∈ Y; ¬ lnull xs ⟧ ⟹ lhd (lSup Y) = lhd xs*
⟨*proof*⟩

lemma *lSup-empty* [*simp*]: *lSup {} = LNil*
⟨*proof*⟩

lemma *lSup-singleton* [*simp*]: *lSup {xs} = xs*
⟨*proof*⟩

lemma *LCons-image-Int-not-lnull*: *(LCons x ‘ A ∩ {xs. ¬ lnull xs}) = LCons x ‘ A*
⟨*proof*⟩

lemma *lSup-LCons*: *A ≠ {} ⟹ lSup (LCons x ‘ A) = LCons x (lSup A)*
⟨*proof*⟩

lemma *lSup-eq-LCons-iff*:
lSup Y = LCons x xs ⟷ (∃ x∈Y. ¬ lnull x) ∧ x = (THE x. x ∈ lhd ‘ (Y ∩ {xs. ¬ lnull xs})) ∧ xs = lSup (ltl ‘ (Y ∩ {xs. ¬ lnull xs}))
⟨*proof*⟩

lemma *lSup-insert-LNil*: *lSup (insert LNil Y) = lSup Y*
⟨*proof*⟩

lemma *lSup-minus-LNil*: *lSup (Y − {LNil}) = lSup Y*
⟨*proof*⟩

lemma *chain-lprefix-ltl*:
assumes *chain*: *Complete-Partial-Order.chain (⊑) A*
shows *Complete-Partial-Order.chain (⊑) (ltl ‘ (A ∩ {xs. ¬ lnull xs}))*
⟨*proof*⟩

lemma *lSup-finite-prefixes*: *lSup* {*ys*. *ys* ⊑ *xs* ∧ *lfinite ys*} = *xs* (**is** *lSup* (*?C xs*) = -)
⟨*proof*⟩

lemma *lSup-finite-gen-prefixes*:
assumes *zs* ⊑ *xs lfinite zs*
shows *lSup* {*ys*. *ys* ⊑ *xs* ∧ *zs* ⊑ *ys* ∧ *lfinite ys*} = *xs*
⟨*proof*⟩

lemma *lSup-strict-prefixes*:
¬ *lfinite xs* ⟹ *lSup* {*ys*. *ys* ⊑ *xs* ∧ *ys* ≠ *xs*} = *xs*
(**is** - ⟹ *lSup* (*?C xs*) = -)
⟨*proof*⟩

lemma *chain-lprefix-lSup*:
⟦ *Complete-Partial-Order.chain* (⊑) *A*; *xs* ∈ *A* ⟧
⟹ *xs* ⊑ *lSup A*
⟨*proof*⟩

lemma *chain-lSup-lprefix*:
⟦ *Complete-Partial-Order.chain* (⊑) *A*; ⋀*xs*. *xs* ∈ *A* ⟹ *xs* ⊑ *zs* ⟧
⟹ *lSup A* ⊑ *zs*
⟨*proof*⟩

lemma *llist-ccpo* [*simp*, *cont-intro*]: *class.ccpo lSup* (⊑) (*mk-less* (⊑))
⟨*proof*⟩

lemmas [*cont-intro*] = *ccpo.admissible-leI*[*OF llist-ccpo*]

lemma *llist-partial-function-definitions*:
partial-function-definitions (⊑) *lSup*
⟨*proof*⟩

interpretation *llist*: *partial-function-definitions* (⊑) *lSup*
rewrites *lSup* {} ≡ *LNil*
⟨*proof*⟩

abbreviation *mono-llist* ≡ *monotone* (*fun-ord* (⊑)) (⊑)

interpretation *llist-lift*: *partial-function-definitions fun-ord lprefix fun-lub lSup*
rewrites *fun-lub lSup* {} ≡ λ-. *LNil*
⟨*proof*⟩

abbreviation *mono-llist-lift* ≡ *monotone* (*fun-ord* (*fun-ord lprefix*)) (*fun-ord lprefix*)

lemma *lprefixes-chain*:
Complete-Partial-Order.chain (⊑) {*ys*. *lprefix ys xs*}
⟨*proof*⟩

lemma *llist-gen-induct*:
assumes *adm*: *ccpo.admissible lSup* (⊑) *P*
and *step*: ∃ *zs*. *zs* ⊑ *xs* ∧ *lfinite zs* ∧ (∀ *ys*. *zs* ⊑ *ys* ⟶ *ys* ⊑ *xs* ⟶ *lfinite ys* ⟶ *P ys*)
shows *P xs*
⟨*proof*⟩

lemma *llist-induct* [*case-names adm LNil LCons*, *induct type*: *llist*]:
assumes *adm*: *ccpo.admissible lSup* (⊑) *P*
and *LNil*: *P LNil*
and *LCons*: ⋀*x xs*. ⟦ *lfinite xs*; *P xs* ⟧ ⟹ *P* (*LCons x xs*)
shows *P xs*
⟨*proof*⟩

lemma *LCons-mono* [*partial-function-mono*, *cont-intro*]:
mono-llist A ⟹ *mono-llist* (λ*f*. *LCons x* (*A f*))
⟨*proof*⟩

lemma *mono2mono-LCons* [*THEN llist.mono2mono*, *simp*, *cont-intro*]:
shows *monotone-LCons*: *monotone* (⊑) (⊑) (*LCons x*)
⟨*proof*⟩

lemma *mcont2mcont-LCons* [*THEN llist.mcont2mcont*, *simp*, *cont-intro*]:
shows *mcont-LCons*: *mcont lSup* (⊑) *lSup* (⊑) (*LCons x*)
⟨*proof*⟩

lemma *mono2mono-ltl*[*THEN llist.mono2mono*, *simp*, *cont-intro*]:
shows *monotone-ltl*: *monotone* (⊑) (⊑) *ltl*
⟨*proof*⟩

lemma *cont-ltl*: *cont lSup* (⊑) *lSup* (⊑) *ltl*
⟨*proof*⟩

lemma *mcont2mcont-ltl* [*THEN llist.mcont2mcont*, *simp*, *cont-intro*]:
shows *mcont-ltl*: *mcont lSup* (⊑) *lSup* (⊑) *ltl*
⟨*proof*⟩

lemma *llist-case-mono* [*partial-function-mono*, *cont-intro*]:
assumes *lnil*: *monotone orda ordb lnil*
and *lcons*: ⋀*x xs*. *monotone orda ordb* (λ*f*. *lcons f x xs*)
shows *monotone orda ordb* (λ*f*. *case-llist* (*lnil f*) (*lcons f*) *x*)
⟨*proof*⟩

lemma *mcont-llist-case* [*cont-intro*, *simp*]:
⟦ *mcont luba orda lubb ordb* (λ*x*. *f x*); ⋀*x xs*. *mcont luba orda lubb ordb* (λ*y*. *g x xs y*) ⟧
⟹ *mcont luba orda lubb ordb* (λ*y*. *case xs of LNil* ⇒ *f y* | *LCons x xs′* ⇒ *g x xs′ y*)

⟨*proof*⟩

lemma *monotone-lprefix-case* [*cont-intro*, *simp*]:
assumes *mono*: ⋀*x*. *monotone* (⊑) (⊑) (λ*xs*. *f x xs* (*LCons x xs*))
shows *monotone* (⊑) (⊑) (λ*xs*. *case xs of LNil* ⇒ *LNil* | *LCons x xs′* ⇒ *f x xs′ xs*)
⟨*proof*⟩

lemma *mcont-lprefix-case-aux*:
fixes *f bot*
defines *g* ≡ λ*xs*. *f* (*lhd xs*) (*ltl xs*) (*LCons* (*lhd xs*) (*ltl xs*))
assumes *mcont*: ⋀*x*. *mcont lSup* (⊑) *lub ord* (λ*xs*. *f x xs* (*LCons x xs*))
and *ccpo*: *class.ccpo lub ord* (*mk-less ord*)
and *bot*: ⋀*x*. *ord bot x*
shows *mcont lSup* (⊑) *lub ord* (λ*xs*. *case xs of LNil* ⇒ *bot* | *LCons x xs′* ⇒ *f x xs′ xs*)
⟨*proof*⟩

lemma *mcont-lprefix-case* [*cont-intro*, *simp*]:
assumes ⋀*x*. *mcont lSup* (⊑) *lSup* (⊑) (λ*xs*. *f x xs* (*LCons x xs*))
shows *mcont lSup* (⊑) *lSup* (⊑) (λ*xs*. *case xs of LNil* ⇒ *LNil* | *LCons x xs′* ⇒ *f x xs′ xs*)
⟨*proof*⟩

lemma *monotone-lprefix-case-lfp* [*cont-intro*, *simp*]:
fixes *f* :: - ⇒ - :: *order-bot*
assumes *mono*: ⋀*x*. *monotone* (⊑) (≤) (λ*xs*. *f x xs* (*LCons x xs*))
shows *monotone* (⊑) (≤) (λ*xs*. *case xs of LNil* ⇒ ⊥ | *LCons x xs* ⇒ *f x xs* (*LCons x xs*))
⟨*proof*⟩

lemma *mcont-lprefix-case-lfp* [*cont-intro*, *simp*]:
fixes *f* :: - => - :: *complete-lattice*
assumes ⋀*x*. *mcont lSup* (⊑) *Sup* (≤) (λ*xs*. *f x xs* (*LCons x xs*))
shows *mcont lSup* (⊑) *Sup* (≤) (λ*xs*. *case xs of LNil* ⇒ ⊥ | *LCons x xs* ⇒ *f x xs* (*LCons x xs*))
⟨*proof*⟩

⟨*ML*⟩

2.8 Monotonicity and continuity of already defined functions

lemma fixes *f F*
defines *F* ≡ λ*lmap xs*. *case xs of LNil* ⇒ *LNil* | *LCons x xs* ⇒ *LCons* (*f x*) (*lmap xs*)
shows *lmap-conv-fixp*: *lmap f* ≡ *ccpo.fixp* (*fun-lub lSup*) (*fun-ord* (⊑)) *F* (**is** *?lhs* ≡ *?rhs*)
and *lmap-mono*: ⋀*xs*. *mono-llist* (λ*lmap*. *F lmap xs*) (**is** *PROP ?mono*)
⟨*proof*⟩

lemma *mono2mono-lmap*[*THEN llist.mono2mono*, *simp*, *cont-intro*]:
shows *monotone-lmap*: *monotone* (⊑) (⊑) (*lmap f*)
⟨*proof*⟩

lemma *mcont2mcont-lmap*[*THEN llist.mcont2mcont*, *simp*, *cont-intro*]:
shows *mcont-lmap*: *mcont lSup* (⊑) *lSup* (⊑) (*lmap f*)
⟨*proof*⟩

lemma [*partial-function-mono*]: *mono-llist F* ⟹ *mono-llist* (λ*f*. *lmap g* (*F f*))
⟨*proof*⟩

lemma *mono-llist-lappend2* [*partial-function-mono*]:
mono-llist A ⟹ *mono-llist* (λ*f*. *lappend xs* (*A f*))
⟨*proof*⟩

lemma *mono2mono-lappend2* [*THEN llist.mono2mono*, *cont-intro*, *simp*]:
shows *monotone-lappend2*: *monotone* (⊑) (⊑) (*lappend xs*)
⟨*proof*⟩

lemma *mcont2mcont-lappend2* [*THEN llist.mcont2mcont*, *cont-intro*, *simp*]:
shows *mcont-lappend2*: *mcont lSup* (⊑) *lSup* (⊑) (*lappend xs*)
⟨*proof*⟩

lemma fixes *f F*
defines *F* ≡ λ*lset xs*. *case xs of LNil* ⇒ {} | *LCons x xs* ⇒ *insert x* (*lset xs*)
shows *lset-conv-fixp*: *lset* ≡ *ccpo.fixp* (*fun-lub Union*) (*fun-ord* (⊆)) *F* (**is** - ≡ *?fixp*)
and *lset-mono*: ⋀*x*. *monotone* (*fun-ord* (⊆)) (⊆) (λ*f*. *F f x*) (**is** *PROP ?mono*)
⟨*proof*⟩

lemma *mono2mono-lset* [*THEN lfp.mono2mono*, *cont-intro*, *simp*]:
shows *monotone-lset*: *monotone* (⊑) (⊆) *lset*
⟨*proof*⟩

lemma *mcont2mcont-lset*[*THEN mcont2mcont*, *cont-intro*, *simp*]:
shows *mcont-lset*: *mcont lSup* (⊑) *Union* (⊆) *lset*
⟨*proof*⟩

lemma *lset-lSup*: *Complete-Partial-Order.chain* (⊑) *Y* ⟹ *lset* (*lSup Y*) = ⋃(*lset* ‘ *Y*)
⟨*proof*⟩

lemma *lfinite-lSupD*: *lfinite* (*lSup A*) ⟹ ∀ *xs* ∈ *A*. *lfinite xs*
⟨*proof*⟩

lemma *monotone-enat-le-lprefix-case* [*cont-intro*, *simp*]:
monotone (≤) (⊑) (λ*x*. *f x* (*eSuc x*)) ⟹ *monotone* (≤) (⊑) (λ*x*. *case x of 0* ⇒

LNil | *eSuc* $x' \Rightarrow f\ x'\ x$)
⟨*proof*⟩

lemma *mcont-enat-le-lprefix-case* [*cont-intro*, *simp*]:
  assumes *mcont Sup* (≤) *lSup* (⊑) (λ*x*. *f x* (*eSuc x*))
  shows *mcont Sup* (≤) *lSup* (⊑) (λ*x*. *case x of 0* ⇒ *LNil* | *eSuc* $x' \Rightarrow f\ x'\ x$)
⟨*proof*⟩

lemma *compact-LConsI*:
  assumes *ccpo.compact lSup* (⊑) *xs*
  shows *ccpo.compact lSup* (⊑) (*LCons x xs*)
⟨*proof*⟩

lemma *compact-LConsD*:
  assumes *ccpo.compact lSup* (⊑) (*LCons x xs*)
  shows *ccpo.compact lSup* (⊑) *xs*
⟨*proof*⟩

lemma *compact-LCons-iff* [*simp*]:
  ccpo.compact lSup (⊑) (*LCons x xs*) ⟷ *ccpo.compact lSup* (⊑) *xs*
⟨*proof*⟩

lemma *compact-lfiniteI*:
  lfinite xs ⟹ *ccpo.compact lSup* (⊑) *xs*
⟨*proof*⟩

lemma *compact-lfiniteD*:
  assumes *ccpo.compact lSup* (⊑) *xs*
  shows *lfinite xs*
⟨*proof*⟩

lemma *compact-eq-lfinite* [*simp*]: *ccpo.compact lSup* (⊑) = *lfinite*
⟨*proof*⟩

2.9 More function definitions

primcorec *iterates* :: ('*a* ⇒ '*a*) ⇒ '*a* ⇒ '*a llist*
where *iterates f x* = *LCons x* (*iterates f* (*f x*))

primrec *llist-of* :: '*a list* ⇒ '*a llist*
where
  llist-of [] = *LNil*
| *llist-of* (*x*#*xs*) = *LCons x* (*llist-of xs*)

definition *list-of* :: '*a llist* ⇒ '*a list*
where [*code del*]: *list-of xs* = (*if lfinite xs then inv llist-of xs else undefined*)

definition *llength* :: '*a llist* ⇒ *enat*
where [*code del*]:

llength = *enat-unfold lnull ltl*

primcorec *ltake* :: *enat* ⇒ *'a llist* ⇒ *'a llist*
where
n = *0* ∨ *lnull xs* ⟹ *lnull* (*ltake n xs*)
| *lhd* (*ltake n xs*) = *lhd xs*
| *ltl* (*ltake n xs*) = *ltake* (*epred n*) (*ltl xs*)

definition *ldropn* :: *nat* ⇒ *'a llist* ⇒ *'a llist*
where *ldropn n xs* = (*ltl* ^^ *n*) *xs*

context notes [[*function-internals*]]
begin

partial-function (*llist*) *ldrop* :: *enat* ⇒ *'a llist* ⇒ *'a llist*
where
ldrop n xs = (*case n of 0* ⇒ *xs* | *eSuc n'* ⇒ *case xs of LNil* ⇒ *LNil* | *LCons x xs'* ⇒ *ldrop n' xs'*)

end

primcorec *ltakeWhile* :: (*'a* ⇒ *bool*) ⇒ *'a llist* ⇒ *'a llist*
where
lnull xs ∨ ¬ *P* (*lhd xs*) ⟹ *lnull* (*ltakeWhile P xs*)
| *lhd* (*ltakeWhile P xs*) = *lhd xs*
| *ltl* (*ltakeWhile P xs*) = *ltakeWhile P* (*ltl xs*)

context fixes *P* :: *'a* ⇒ *bool*
notes [[*function-internals*]]
begin

partial-function (*llist*) *ldropWhile* :: *'a llist* ⇒ *'a llist*
where *ldropWhile xs* = (*case xs of LNil* ⇒ *LNil* | *LCons x xs'* ⇒ *if P x then ldropWhile xs' else xs*)

partial-function (*llist*) *lfilter* :: *'a llist* ⇒ *'a llist*
where *lfilter xs* = (*case xs of LNil* ⇒ *LNil* | *LCons x xs'* ⇒ *if P x then LCons x* (*lfilter xs'*) *else lfilter xs'*)

end

primrec *lnth* :: *'a llist* ⇒ *nat* ⇒ *'a*
where
lnth xs 0 = (*case xs of LNil* ⇒ *undefined* (*0* :: *nat*) | *LCons x xs'* ⇒ *x*)
| *lnth xs* (*Suc n*) = (*case xs of LNil* ⇒ *undefined* (*Suc n*) | *LCons x xs'* ⇒ *lnth xs' n*)

declare *lnth.simps* [*simp del*]

primcorec *lzip* :: *'a llist ⇒ 'b llist ⇒ ('a × 'b) llist*
where
lnull xs ∨ lnull ys ⟹ lnull (lzip xs ys)
| *lhd (lzip xs ys) = (lhd xs, lhd ys)*
| *ltl (lzip xs ys) = lzip (ltl xs) (ltl ys)*

definition *llast* :: *'a llist ⇒ 'a*
where [*nitpick-simp*]:
llast xs = (case llength xs of enat n ⇒ (case n of 0 ⇒ undefined | Suc n' ⇒ lnth xs n') | ∞ ⇒ undefined)

coinductive *ldistinct* :: *'a llist ⇒ bool*
where
LNil [*simp*]: *ldistinct LNil*
| *LCons*: *⟦ x ∉ lset xs; ldistinct xs ⟧ ⟹ ldistinct (LCons x xs)*

hide-fact (**open**) *LNil LCons*

definition *inf-llist* :: *(nat ⇒ 'a) ⇒ 'a llist*
where [*code del*]: *inf-llist f = lmap f (iterates Suc 0)*

abbreviation *repeat* :: *'a ⇒ 'a llist*
where *repeat ≡ iterates (λx. x)*

definition *lstrict-prefix* :: *'a llist ⇒ 'a llist ⇒ bool*
where [*code del*]: *lstrict-prefix xs ys ≡ xs ⊑ ys ∧ xs ≠ ys*

longest common prefix

definition *llcp* :: *'a llist ⇒ 'a llist ⇒ enat*
where [*code del*]:
llcp xs ys =
enat-unfold (λ(xs, ys). lnull xs ∨ lnull ys ∨ lhd xs ≠ lhd ys) (map-prod ltl ltl) (xs, ys)

coinductive *llexord* :: *('a ⇒ 'a ⇒ bool) ⇒ 'a llist ⇒ 'a llist ⇒ bool*
for *r* :: *'a ⇒ 'a ⇒ bool*
where
llexord-LCons-eq: *llexord r xs ys ⟹ llexord r (LCons x xs) (LCons x ys)*
| *llexord-LCons-less*: *r x y ⟹ llexord r (LCons x xs) (LCons y ys)*
| *llexord-LNil* [*simp*, *intro*!]: *llexord r LNil ys*

context notes [[*function-internals*]]
begin

partial-function (*llist*) *lconcat* :: *'a llist llist ⇒ 'a llist*
where *lconcat xss = (case xss of LNil ⇒ LNil | LCons xs xss' ⇒ lappend xs (lconcat xss'))*

end

definition *lhd′* :: *'a llist* ⇒ *'a option* **where**
lhd′ xs = (*if lnull xs then None else Some* (*lhd xs*))

lemma *lhd′-simps*[*simp*]:
lhd′ LNil = *None*
lhd′ (*LCons x xs*) = *Some x*
⟨*proof*⟩

definition *ltl′* :: *'a llist* ⇒ *'a llist option* **where**
ltl′ xs = (*if lnull xs then None else Some* (*ltl xs*))

lemma *ltl′-simps*[*simp*]:
ltl′ LNil = *None*
ltl′ (*LCons x xs*) = *Some xs*
⟨*proof*⟩

definition *lnths* :: *'a llist* ⇒ *nat set* ⇒ *'a llist*
where *lnths xs A* = *lmap fst* (*lfilter* (λ(*x*, *y*). *y* ∈ *A*) (*lzip xs* (*iterates Suc 0*)))

definition (**in** *monoid-add*) *lsum-list* :: *'a llist* ⇒ *'a*
where *lsum-list xs* = (*if lfinite xs then sum-list* (*list-of xs*) *else 0*)

2.10 Converting ordinary lists to lazy lists: *llist-of*

lemma *lhd-llist-of* [*simp*]: *lhd* (*llist-of xs*) = *hd xs*
⟨*proof*⟩

lemma *ltl-llist-of* [*simp*]: *ltl* (*llist-of xs*) = *llist-of* (*tl xs*)
⟨*proof*⟩

lemma *lfinite-llist-of* [*simp*]: *lfinite* (*llist-of xs*)
⟨*proof*⟩

lemma *lfinite-eq-range-llist-of*: *lfinite xs* ⟷ *xs* ∈ *range llist-of*
⟨*proof*⟩

lemma *lnull-llist-of* [*simp*]: *lnull* (*llist-of xs*) ⟷ *xs* = []
⟨*proof*⟩

lemma *llist-of-eq-LNil-conv*:
llist-of xs = *LNil* ⟷ *xs* = []
⟨*proof*⟩

lemma *llist-of-eq-LCons-conv*:
llist-of xs = *LCons y ys* ⟷ (∃ *xs′*. *xs* = *y* # *xs′* ∧ *ys* = *llist-of xs′*)
⟨*proof*⟩

lemma *lappend-llist-of-llist-of*:

lappend (*llist-of xs*) (*llist-of ys*) = *llist-of* (*xs* @ *ys*)
⟨*proof*⟩

lemma *lfinite-rev-induct* [*consumes 1*, *case-names Nil snoc*]:
assumes *fin*: *lfinite xs*
and *Nil*: *P LNil*
and *snoc*: ⋀*x xs*. ⟦ *lfinite xs*; *P xs* ⟧ ⟹ *P* (*lappend xs* (*LCons x LNil*))
shows *P xs*
⟨*proof*⟩

lemma *lappend-llist-of-LCons*:
lappend (*llist-of xs*) (*LCons y ys*) = *lappend* (*llist-of* (*xs* @ [*y*])) *ys*
⟨*proof*⟩

lemma *lmap-llist-of* [*simp*]:
lmap f (*llist-of xs*) = *llist-of* (*map f xs*)
⟨*proof*⟩

lemma *lset-llist-of* [*simp*]: *lset* (*llist-of xs*) = *set xs*
⟨*proof*⟩

lemma *llist-of-inject* [*simp*]: *llist-of xs* = *llist-of ys* ⟷ *xs* = *ys*
⟨*proof*⟩

lemma *inj-llist-of* [*simp*]: *inj llist-of*
⟨*proof*⟩

2.11 Converting finite lazy lists to ordinary lists: *list-of*

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

lemma *llist-of-list-of* [*simp*]: *lfinite xs* ⟹ *llist-of* (*list-of xs*) = *xs*
⟨*proof*⟩

lemma *list-of-LNil* [*simp*, *nitpick-simp*]: *list-of LNil* = []
⟨*proof*⟩

lemma *list-of-LCons* [*simp*]: *lfinite xs* ⟹ *list-of* (*LCons x xs*) = *x* # *list-of xs*
⟨*proof*⟩

lemma *list-of-LCons-conv* [*nitpick-simp*]:
list-of (*LCons x xs*) = (*if lfinite xs then x* # *list-of xs else undefined*)
⟨*proof*⟩

lemma *list-of-lappend*:
assumes *lfinite xs lfinite ys*
shows *list-of* (*lappend xs ys*) = *list-of xs* @ *list-of ys*
⟨*proof*⟩

lemma *list-of-lmap* [*simp*]:
assumes *lfinite xs*
shows *list-of* (*lmap f xs*) = *map f* (*list-of xs*)
⟨*proof*⟩

lemma *set-list-of* [*simp*]:
assumes *lfinite xs*
shows *set* (*list-of xs*) = *lset xs*
⟨*proof*⟩

lemma *hd-list-of* [*simp*]: *lfinite xs* ⟹ *hd* (*list-of xs*) = *lhd xs*
⟨*proof*⟩

lemma *tl-list-of*: *lfinite xs* ⟹ *tl* (*list-of xs*) = *list-of* (*ltl xs*)
⟨*proof*⟩

Efficient implementation via tail recursion suggested by Brian Huffman

definition *list-of-aux* :: *'a list* ⇒ *'a llist* ⇒ *'a list*
where *list-of-aux xs ys* = (*if lfinite ys then rev xs* @ *list-of ys else undefined*)

lemma *list-of-code* [*code*]: *list-of* = *list-of-aux* []
⟨*proof*⟩

lemma *list-of-aux-code* [*code*]:
list-of-aux xs LNil = *rev xs*
list-of-aux xs (*LCons y ys*) = *list-of-aux* (*y* # *xs*) *ys*
⟨*proof*⟩

2.12 The length of a lazy list: *llength*

lemma [*simp*, *nitpick-simp*]:
shows *llength-LNil*: *llength LNil* = *0*
and *llength-LCons*: *llength* (*LCons x xs*) = *eSuc* (*llength xs*)
⟨*proof*⟩

lemma *llength-eq-0* [*simp*]: *llength xs* = *0* ⟷ *lnull xs*
⟨*proof*⟩

lemma *llength-lnull* [*simp*]: *lnull xs* ⟹ *llength xs* = *0*
⟨*proof*⟩

lemma *epred-llength*:
epred (*llength xs*) = *llength* (*ltl xs*)
⟨*proof*⟩

lemmas *llength-ltl* = *epred-llength*[*symmetric*]

lemma *llength-lmap* [*simp*]: *llength* (*lmap f xs*) = *llength xs*

⟨*proof*⟩

lemma *llength-lappend* [*simp*]: *llength* (*lappend xs ys*) = *llength xs* + *llength ys*
⟨*proof*⟩

lemma *llength-llist-of* [*simp*]:
llength (*llist-of xs*) = *enat* (*length xs*)
⟨*proof*⟩

lemma *length-list-of*:
lfinite xs ⟹ *enat* (*length* (*list-of xs*)) = *llength xs*
⟨*proof*⟩

lemma *length-list-of-conv-the-enat*:
lfinite xs ⟹ *length* (*list-of xs*) = *the-enat* (*llength xs*)
⟨*proof*⟩

lemma *llength-eq-enat-lfiniteD*: *llength xs* = *enat n* ⟹ *lfinite xs*
⟨*proof*⟩

lemma *lfinite-llength-enat*:
assumes *lfinite xs*
shows ∃ *n*. *llength xs* = *enat n*
⟨*proof*⟩

lemma *lfinite-conv-llength-enat*:
lfinite xs ⟷ (∃ *n*. *llength xs* = *enat n*)
⟨*proof*⟩

lemma *not-lfinite-llength*:
¬ *lfinite xs* ⟹ *llength xs* = ∞
⟨*proof*⟩

lemma *llength-eq-infty-conv-lfinite*:
llength xs = ∞ ⟷ ¬ *lfinite xs*
⟨*proof*⟩

lemma *lfinite-finite-index*: *lfinite xs* ⟹ *finite* {*n*. *enat n* < *llength xs*}
⟨*proof*⟩

tail-recursive implementation for *llength*

definition *gen-llength* :: *nat* ⇒ *'a llist* ⇒ *enat*
where *gen-llength n xs* = *enat n* + *llength xs*

lemma *gen-llength-code* [*code*]:
gen-llength n LNil = *enat n*
gen-llength n (*LCons x xs*) = *gen-llength* (*n* + *1*) *xs*
⟨*proof*⟩

lemma *llength-code* [*code*]: *llength* = *gen-llength 0*
⟨*proof*⟩

lemma fixes *F*
defines $F \equiv \lambda llength\ xs.\ case\ xs\ of\ LNil \Rightarrow 0 \mid LCons\ x\ xs \Rightarrow eSuc\ (llength\ xs)$
shows *llength-conv-fixp*: *llength* ≡ *ccpo.fixp* (*fun-lub Sup*) (*fun-ord* (≤)) *F* (**is** - ≡ *?fixp*)
and *llength-mono*: $\bigwedge xs.\ monotone\ (fun\text{-}ord\ (\leq))\ (\leq)\ (\lambda llength.\ F\ llength\ xs)$ (**is** *PROP ?mono*)
⟨*proof*⟩

lemma *mono2mono-llength*[*THEN lfp.mono2mono*, *simp*, *cont-intro*]:
shows *monotone-llength*: *monotone* (⊑) (≤) *llength*
⟨*proof*⟩

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

2.13 Taking and dropping from lazy lists: *ltake*, *ldropn*, and *ldrop*

lemma *ltake-LNil* [*simp*, *code*, *nitpick-simp*]: *ltake n LNil* = *LNil*
⟨*proof*⟩

lemma *ltake-0* [*simp*]: *ltake 0 xs* = *LNil*
⟨*proof*⟩

lemma *ltake-eSuc-LCons* [*simp*]:
ltake (*eSuc n*) (*LCons x xs*) = *LCons x* (*ltake n xs*)
⟨*proof*⟩

lemma *ltake-eSuc*:
ltake (*eSuc n*) *xs* =
(*case xs of LNil* ⇒ *LNil* | *LCons x xs′* ⇒ *LCons x* (*ltake n xs′*))
⟨*proof*⟩

lemma *lnull-ltake* [*simp*]: *lnull* (*ltake n xs*) ⟷ *lnull xs* ∨ *n* = *0*
⟨*proof*⟩

lemma *ltake-eq-LNil-iff*: *ltake n xs* = *LNil* ⟷ *xs* = *LNil* ∨ *n* = *0*
⟨*proof*⟩

lemma *LNil-eq-ltake-iff* [*simp*]: *LNil* = *ltake n xs* ⟷ *xs* = *LNil* ∨ *n* = *0*
⟨*proof*⟩

lemma *ltake-LCons* [*code*, *nitpick-simp*]:
ltake n (*LCons x xs*) =
(*case n of 0* ⇒ *LNil* | *eSuc n′* ⇒ *LCons x* (*ltake n′ xs*))

⟨*proof*⟩

lemma *lhd-ltake* [*simp*]: $n \neq 0 \Longrightarrow lhd\ (ltake\ n\ xs) = lhd\ xs$
⟨*proof*⟩

lemma *ltl-ltake*: $ltl\ (ltake\ n\ xs) = ltake\ (epred\ n)\ (ltl\ xs)$
⟨*proof*⟩

lemmas *ltake-epred-ltl* = *ltl-ltake* [*symmetric*]

declare *ltake.sel*(*2*) [*simp del*]

lemma *ltake-ltl*: $ltake\ n\ (ltl\ xs) = ltl\ (ltake\ (eSuc\ n)\ xs)$
⟨*proof*⟩

lemma *llength-ltake* [*simp*]: $llength\ (ltake\ n\ xs) = min\ n\ (llength\ xs)$
⟨*proof*⟩

lemma *ltake-lmap* [*simp*]: $ltake\ n\ (lmap\ f\ xs) = lmap\ f\ (ltake\ n\ xs)$
⟨*proof*⟩

lemma *ltake-ltake* [*simp*]: $ltake\ n\ (ltake\ m\ xs) = ltake\ (min\ n\ m)\ xs$
⟨*proof*⟩

lemma *lset-ltake*: $lset\ (ltake\ n\ xs) \subseteq lset\ xs$
⟨*proof*⟩

lemma *ltake-all*: $llength\ xs \leq m \Longrightarrow ltake\ m\ xs = xs$
⟨*proof*⟩

lemma *ltake-llist-of* [*simp*]:
$ltake\ (enat\ n)\ (llist\text{-}of\ xs) = llist\text{-}of\ (take\ n\ xs)$
⟨*proof*⟩

lemma *lfinite-ltake* [*simp*]:
$lfinite\ (ltake\ n\ xs) \longleftrightarrow lfinite\ xs \vee n < \infty$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *ltake-lappend1*: $n \leq llength\ xs \Longrightarrow ltake\ n\ (lappend\ xs\ ys) = ltake\ n\ xs$
⟨*proof*⟩

lemma *ltake-lappend2*:
$llength\ xs \leq n \Longrightarrow ltake\ n\ (lappend\ xs\ ys) = lappend\ xs\ (ltake\ (n - llength\ xs)\ ys)$
⟨*proof*⟩

lemma *ltake-lappend*:
$ltake\ n\ (lappend\ xs\ ys) = lappend\ (ltake\ n\ xs)\ (ltake\ (n - llength\ xs)\ ys)$

⟨*proof*⟩

lemma *take-list-of*:
assumes *lfinite xs*
shows *take n* (*list-of xs*) = *list-of* (*ltake* (*enat n*) *xs*)
⟨*proof*⟩

lemma *ltake-eq-ltake-antimono*:
⟦ *ltake n xs* = *ltake n ys*; *m* ≤ *n* ⟧ ⟹ *ltake m xs* = *ltake m ys*
⟨*proof*⟩

lemma *ltake-is-lprefix* [*simp*, *intro*]: *ltake n xs* ⊑ *xs*
⟨*proof*⟩

lemma *lprefix-ltake-same* [*simp*]:
ltake n xs ⊑ *ltake m xs* ⟷ *n* ≤ *m* ∨ *llength xs* ≤ *m*
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma fixes *f F*
defines *F* ≡ λ*ltake n xs*. *case xs of LNil* ⇒ *LNil* | *LCons x xs* ⇒ *case n of 0* ⇒ *LNil* | *eSuc n* ⇒ *LCons x* (*ltake n xs*)
shows *ltake-conv-fixp*: *ltake* ≡ *curry* (*ccpo.fixp* (*fun-lub lSup*) (*fun-ord* (⊑)) (λ*ltake*. *case-prod* (*F* (*curry ltake*)))) (**is** *?lhs* ≡ *?rhs*)
and *ltake-mono*: ⋀*nxs*. *mono-llist* (λ*ltake*. *case nxs of* (*n*, *xs*) ⇒ *F* (*curry ltake*) *n xs*) (**is** *PROP ?mono*)
⟨*proof*⟩

lemma *monotone-ltake*: *monotone* (*rel-prod* (≤) (⊑)) (⊑) (*case-prod ltake*)
⟨*proof*⟩

lemma *mono2mono-ltake1*[*THEN llist.mono2mono*, *cont-intro*, *simp*]:
shows *monotone-ltake1*: *monotone* (≤) (⊑) (λ*n*. *ltake n xs*)
⟨*proof*⟩

lemma *mono2mono-ltake2*[*THEN llist.mono2mono*, *cont-intro*, *simp*]:
shows *monotone-ltake2*: *monotone* (⊑) (⊑) (*ltake n*)
⟨*proof*⟩

lemma *mcont-ltake*: *mcont* (*prod-lub Sup lSup*) (*rel-prod* (≤) (⊑)) *lSup* (⊑) (*case-prod ltake*)
⟨*proof*⟩

lemma *mcont2mcont-ltake1* [*THEN llist.mcont2mcont*, *cont-intro*, *simp*]:
shows *mcont-ltake1*: *mcont Sup* (≤) *lSup* (⊑) (λ*n*. *ltake n xs*)
⟨*proof*⟩

lemma *mcont2mcont-ltake2* [*THEN llist.mcont2mcont*, *cont-intro*, *simp*]:
shows *mcont-ltake2*: *mcont lSup* (⊑) *lSup* (⊑) (*ltake n*)

⟨*proof*⟩

lemma [*partial-function-mono*]: *mono-llist F* ⟹ *mono-llist* (λ*f*. *ltake n* (*F f*))
⟨*proof*⟩

lemma *llist-induct2*:
assumes *adm*: *ccpo.admissible* (*prod-lub lSup lSup*) (*rel-prod* (⊑) (⊑)) (λ*x*. *P* (*fst x*) (*snd x*))
and *LNil*: *P LNil LNil*
and *LCons1*: ⋀*x xs*. ⟦ *lfinite xs*; *P xs LNil* ⟧ ⟹ *P* (*LCons x xs*) *LNil*
and *LCons2*: ⋀*y ys*. ⟦ *lfinite ys*; *P LNil ys* ⟧ ⟹ *P LNil* (*LCons y ys*)
and *LCons*: ⋀*x xs y ys*. ⟦ *lfinite xs*; *lfinite ys*; *P xs ys* ⟧ ⟹ *P* (*LCons x xs*) (*LCons y ys*)
shows *P xs ys*
⟨*proof*⟩

lemma *ldropn-0* [*simp*]: *ldropn 0 xs* = *xs*
⟨*proof*⟩

lemma *ldropn-LNil* [*code*, *simp*]: *ldropn n LNil* = *LNil*
⟨*proof*⟩

lemma *ldropn-lnull*: *lnull xs* ⟹ *ldropn n xs* = *LNil*
⟨*proof*⟩

lemma *ldropn-LCons* [*code*]:
ldropn n (*LCons x xs*) = (*case n of 0* ⇒ *LCons x xs* | *Suc n′* ⇒ *ldropn n′ xs*)
⟨*proof*⟩

lemma *ldropn-Suc*: *ldropn* (*Suc n*) *xs* = (*case xs of LNil* ⇒ *LNil* | *LCons x xs′* ⇒ *ldropn n xs′*)
⟨*proof*⟩

lemma *ldropn-Suc-LCons* [*simp*]: *ldropn* (*Suc n*) (*LCons x xs*) = *ldropn n xs*
⟨*proof*⟩

lemma *ltl-ldropn*: *ltl* (*ldropn n xs*) = *ldropn n* (*ltl xs*)
⟨*proof*⟩

lemma *ldrop-simps* [*simp*]:
shows *ldrop-LNil*: *ldrop n LNil* = *LNil*
and *ldrop-0*: *ldrop 0 xs* = *xs*
and *ldrop-eSuc-LCons*: *ldrop* (*eSuc n*) (*LCons x xs*) = *ldrop n xs*
⟨*proof*⟩

lemma *ldrop-lnull*: *lnull xs* ⟹ *ldrop n xs* = *LNil*
⟨*proof*⟩

lemma fixes *f F*

defines $F \equiv \lambda ldropn\ xs.\ case\ xs\ of\ LNil \Rightarrow \lambda\text{-}.\ LNil \mid LCons\ x\ xs \Rightarrow \lambda n.\ if\ n = 0\ then\ LCons\ x\ xs\ else\ ldropn\ xs\ (n - 1)$
shows *ldrop-conv-fixp*: $(\lambda xs\ n.\ ldropn\ n\ xs) \equiv ccpo.fixp\ (fun\text{-}lub\ (fun\text{-}lub\ lSup))\ (fun\text{-}ord\ (fun\text{-}ord\ lprefix))\ (\lambda ldrop.\ F\ ldrop)$ (**is** *?lhs* ≡ *?rhs*)
and *ldrop-mono*: $\bigwedge xs.\ mono\text{-}llist\text{-}lift\ (\lambda ldrop.\ F\ ldrop\ xs)$ (**is** *PROP ?mono*)
⟨*proof*⟩

lemma *ldropn-fixp-case-conv*:
$(\lambda xs.\ case\ xs\ of\ LNil \Rightarrow \lambda\text{-}.\ LNil \mid LCons\ x\ xs \Rightarrow \lambda n.\ if\ n = 0\ then\ LCons\ x\ xs\ else\ f\ xs\ (n - 1)) =$
$(\lambda xs\ n.\ case\ xs\ of\ LNil \Rightarrow LNil \mid LCons\ x\ xs \Rightarrow if\ n = 0\ then\ LCons\ x\ xs\ else\ f\ xs\ (n - 1))$
⟨*proof*⟩

lemma *monotone-ldropn-aux*: $monotone\ lprefix\ (fun\text{-}ord\ lprefix)\ (\lambda xs\ n.\ ldropn\ n\ xs)$
⟨*proof*⟩

lemma *mono2mono-ldropn*[*THEN llist.mono2mono*, *cont-intro*, *simp*]:
shows *monotone-ldropn′*: $monotone\ lprefix\ lprefix\ (\lambda xs.\ ldropn\ n\ xs)$
⟨*proof*⟩

lemma *mcont-ldropn-aux*: $mcont\ lSup\ lprefix\ (fun\text{-}lub\ lSup)\ (fun\text{-}ord\ lprefix)\ (\lambda xs\ n.\ ldropn\ n\ xs)$
⟨*proof*⟩

lemma *mcont2mcont-ldropn* [*THEN llist.mcont2mcont*, *cont-intro*, *simp*]:
shows *mcont-ldropn*: $mcont\ lSup\ lprefix\ lSup\ lprefix\ (ldropn\ n)$
⟨*proof*⟩

lemma *monotone-enat-cocase* [*cont-intro*, *simp*]:
$[\![\bigwedge n.\ monotone\ (\leq)\ ord\ (\lambda n.\ f\ n\ (eSuc\ n)); \bigwedge n.\ ord\ a\ (f\ n\ (eSuc\ n)); ord\ a\ a]\!] \Longrightarrow monotone\ (\leq)\ ord\ (\lambda n.\ case\ n\ of\ 0 \Rightarrow a \mid eSuc\ n' \Rightarrow f\ n'\ n)$
⟨*proof*⟩

lemma *monotone-ldrop*: $monotone\ (rel\text{-}prod\ (=)\ (\sqsubseteq))\ (\sqsubseteq)\ (case\text{-}prod\ ldrop)$
⟨*proof*⟩

lemma *mono2mono-ldrop2* [*THEN llist.mono2mono*, *cont-intro*, *simp*]:
shows *monotone-ldrop2*: $monotone\ (\sqsubseteq)\ (\sqsubseteq)\ (ldrop\ n)$
⟨*proof*⟩

lemma *mcont-ldrop*: $mcont\ (prod\text{-}lub\ the\text{-}Sup\ lSup)\ (rel\text{-}prod\ (=)\ (\sqsubseteq))\ lSup\ (\sqsubseteq)\ (case\text{-}prod\ ldrop)$
⟨*proof*⟩

lemma *mcont2monct-ldrop2* [*THEN llist.mcont2mcont*, *cont-intro*, *simp*]:
shows *mcont-ldrop2*: $mcont\ lSup\ (\sqsubseteq)\ lSup\ (\sqsubseteq)\ (ldrop\ n)$

⟨*proof*⟩

lemma *ldrop-eSuc-conv-ltl*: *ldrop* (*eSuc n*) *xs* = *ltl* (*ldrop n xs*)
⟨*proof*⟩

lemma *ldrop-ltl*: *ldrop n* (*ltl xs*) = *ldrop* (*eSuc n*) *xs*
⟨*proof*⟩

lemma *lnull-ldropn* [*simp*]: *lnull* (*ldropn n xs*) ⟷ *llength xs* ≤ *enat n*
⟨*proof*⟩

lemma *ldrop-eq-LNil* [*simp*]: *ldrop n xs* = *LNil* ⟷ *llength xs* ≤ *n*
⟨*proof*⟩

lemma *lnull-ldrop* [*simp*]: *lnull* (*ldrop n xs*) ⟷ *llength xs* ≤ *n*
⟨*proof*⟩

lemma *ldropn-eq-LNil*: (*ldropn n xs* = *LNil*) = (*llength xs* ≤ *enat n*)
⟨*proof*⟩

lemma *ldropn-all*: *llength xs* ≤ *enat m* ⟹ *ldropn m xs* = *LNil*
⟨*proof*⟩

lemma *ldrop-all*: *llength xs* ≤ *m* ⟹ *ldrop m xs* = *LNil*
⟨*proof*⟩

lemma *ltl-ldrop*: *ltl* (*ldrop n xs*) = *ldrop n* (*ltl xs*)
⟨*proof*⟩

lemma *ldrop-eSuc*:
ldrop (*eSuc n*) *xs* = (*case xs of LNil* ⇒ *LNil* | *LCons x xs′* ⇒ *ldrop n xs′*)
⟨*proof*⟩

lemma *ldrop-LCons*:
ldrop n (*LCons x xs*) = (*case n of 0* ⇒ *LCons x xs* | *eSuc n′* ⇒ *ldrop n′ xs*)
⟨*proof*⟩

lemma *ldrop-inf* [*code*, *simp*]: *ldrop* ∞ *xs* = *LNil*
⟨*proof*⟩

lemma *ldrop-enat* [*code*]: *ldrop* (*enat n*) *xs* = *ldropn n xs*
⟨*proof*⟩

lemma *lfinite-ldropn* [*simp*]: *lfinite* (*ldropn n xs*) = *lfinite xs*
⟨*proof*⟩

lemma *lfinite-ldrop* [*simp*]:
lfinite (*ldrop n xs*) ⟷ *lfinite xs* ∨ *n* = ∞
⟨*proof*⟩

lemma *ldropn-ltl*: *ldropn n* (*ltl xs*) = *ldropn* (*Suc n*) *xs*
⟨*proof*⟩

lemmas *ldrop-eSuc-ltl* = *ldropn-ltl*[*symmetric*]

lemma *lset-ldropn-subset*: *lset* (*ldropn n xs*) ⊆ *lset xs*
⟨*proof*⟩

lemma *in-lset-ldropnD*: *x* ∈ *lset* (*ldropn n xs*) ⟹ *x* ∈ *lset xs*
⟨*proof*⟩

lemma *lset-ldrop-subset*: *lset* (*ldrop n xs*) ⊆ *lset xs*
⟨*proof*⟩

lemma *in-lset-ldropD*: *x* ∈ *lset* (*ldrop n xs*) ⟹ *x* ∈ *lset xs*
⟨*proof*⟩

lemma *lappend-ltake-ldrop*: *lappend* (*ltake n xs*) (*ldrop n xs*) = *xs*
⟨*proof*⟩

lemma *ldropn-lappend*:
ldropn n (*lappend xs ys*) =
(*if enat n* < *llength xs then lappend* (*ldropn n xs*) *ys*
else ldropn (*n* − *the-enat* (*llength xs*)) *ys*)
⟨*proof*⟩

lemma *ldropn-lappend2*:
llength xs ≤ *enat n* ⟹ *ldropn n* (*lappend xs ys*) = *ldropn* (*n* − *the-enat* (*llength xs*)) *ys*
⟨*proof*⟩

lemma *lappend-ltake-enat-ldropn* [*simp*]: *lappend* (*ltake* (*enat n*) *xs*) (*ldropn n xs*) = *xs*
⟨*proof*⟩

lemma *ldrop-lappend*:
ldrop n (*lappend xs ys*) =
(*if n* < *llength xs then lappend* (*ldrop n xs*) *ys*
else ldrop (*n* − *llength xs*) *ys*)
— cannot prove this directly using fixpoint induction, because (−) is not a least fixpoint
⟨*proof*⟩

lemma *ltake-plus-conv-lappend*:
ltake (*n* + *m*) *xs* = *lappend* (*ltake n xs*) (*ltake m* (*ldrop n xs*))
⟨*proof*⟩

lemma *ldropn-eq-LConsD*:

ldropn n xs = LCons y ys ⟹ enat n < llength xs
⟨*proof*⟩

lemma *ldrop-eq-LConsD*:
ldrop n xs = LCons y ys ⟹ n < llength xs
⟨*proof*⟩

lemma *ldropn-lmap* [*simp*]: *ldropn n (lmap f xs) = lmap f (ldropn n xs)*
⟨*proof*⟩

lemma *ldrop-lmap* [*simp*]: *ldrop n (lmap f xs) = lmap f (ldrop n xs)*
⟨*proof*⟩

lemma *ldropn-ldropn* [*simp*]:
ldropn n (ldropn m xs) = ldropn (n + m) xs
⟨*proof*⟩

lemma *ldrop-ldrop* [*simp*]:
ldrop n (ldrop m xs) = ldrop (n + m) xs
⟨*proof*⟩

lemma *llength-ldropn* [*simp*]: *llength (ldropn n xs) = llength xs − enat n*
⟨*proof*⟩

lemma *enat-llength-ldropn*:
enat n ≤ llength xs ⟹ enat (n − m) ≤ llength (ldropn m xs)
⟨*proof*⟩

lemma *ldropn-llist-of* [*simp*]: *ldropn n (llist-of xs) = llist-of (drop n xs)*
⟨*proof*⟩

lemma *ldrop-llist-of*: *ldrop (enat n) (llist-of xs) = llist-of (drop n xs)*
⟨*proof*⟩

lemma *drop-list-of*:
lfinite xs ⟹ drop n (list-of xs) = list-of (ldropn n xs)
⟨*proof*⟩

lemma *llength-ldrop*: *llength (ldrop n xs) = (if n = ∞ then 0 else llength xs − n)*
⟨*proof*⟩

lemma *ltake-ldropn*: *ltake n (ldropn m xs) = ldropn m (ltake (n + enat m) xs)*
⟨*proof*⟩

lemma *ldropn-ltake*: *ldropn n (ltake m xs) = ltake (m − enat n) (ldropn n xs)*
⟨*proof*⟩

lemma *ltake-ldrop*: *ltake n (ldrop m xs) = ldrop m (ltake (n + m) xs)*
⟨*proof*⟩

lemma *ldrop-ltake*: *ldrop n* (*ltake m xs*) = *ltake* (*m* − *n*) (*ldrop n xs*)
⟨*proof*⟩

2.14 Taking the *n*-th element of a lazy list: *lnth*

lemma *lnth-LNil*:
lnth LNil n = *undefined n*
⟨*proof*⟩

lemma *lnth-0* [*simp*]:
lnth (*LCons x xs*) *0* = *x*
⟨*proof*⟩

lemma *lnth-Suc-LCons* [*simp*]:
lnth (*LCons x xs*) (*Suc n*) = *lnth xs n*
⟨*proof*⟩

lemma *lnth-LCons*:
lnth (*LCons x xs*) *n* = (*case n of 0* ⇒ *x* | *Suc n′* ⇒ *lnth xs n′*)
⟨*proof*⟩

lemma *lnth-LCons′*: *lnth* (*LCons x xs*) *n* = (*if n* = *0 then x else lnth xs* (*n* − *1*))
⟨*proof*⟩

lemma *lhd-conv-lnth*:
¬ *lnull xs* ⟹ *lhd xs* = *lnth xs 0*
⟨*proof*⟩

lemmas *lnth-0-conv-lhd* = *lhd-conv-lnth*[*symmetric*]

lemma *lnth-ltl*: ¬ *lnull xs* ⟹ *lnth* (*ltl xs*) *n* = *lnth xs* (*Suc n*)
⟨*proof*⟩

lemma *lhd-ldropn*:
enat n < *llength xs* ⟹ *lhd* (*ldropn n xs*) = *lnth xs n*
⟨*proof*⟩

lemma *lhd-ldrop*:
assumes *n* < *llength xs*
shows *lhd* (*ldrop n xs*) = *lnth xs* (*the-enat n*)
⟨*proof*⟩

lemma *lnth-beyond*:
llength xs ≤ *enat n* ⟹ *lnth xs n* = *undefined* (*n* − (*case llength xs of enat m* ⇒ *m*))
⟨*proof*⟩

lemma *lnth-lmap* [*simp*]:

enat n < llength xs ⟹ lnth (lmap f xs) n = f (lnth xs n)
⟨*proof*⟩

lemma *lnth-ldropn* [*simp*]:
enat (n + m) < llength xs ⟹ lnth (ldropn n xs) m = lnth xs (m + n)
⟨*proof*⟩

lemma *lnth-ldrop* [*simp*]:
n + enat m < llength xs ⟹ lnth (ldrop n xs) m = lnth xs (m + the-enat n)
⟨*proof*⟩

lemma *in-lset-conv-lnth*:
x ∈ lset xs ⟷ (∃ n. enat n < llength xs ∧ lnth xs n = x)
(**is** *?lhs ⟷ ?rhs*)
⟨*proof*⟩

lemma *lset-conv-lnth*: *lset xs = {lnth xs n|n. enat n < llength xs}*
⟨*proof*⟩

lemma *lnth-llist-of* [*simp*]: *lnth (llist-of xs) = nth xs*
⟨*proof*⟩

lemma *nth-list-of* [*simp*]:
assumes *lfinite xs*
shows *nth (list-of xs) = lnth xs*
⟨*proof*⟩

lemma *lnth-lappend1*:
enat n < llength xs ⟹ lnth (lappend xs ys) n = lnth xs n
⟨*proof*⟩

lemma *lnth-lappend-llist-of*:
lnth (lappend (llist-of xs) ys) n =
(if n < length xs then xs ! n else lnth ys (n − length xs))
⟨*proof*⟩

lemma *lnth-lappend2*:
⟦ llength xs = enat k; k ≤ n ⟧ ⟹ lnth (lappend xs ys) n = lnth ys (n − k)
⟨*proof*⟩

lemma *lnth-lappend*:
lnth (lappend xs ys) n = (if enat n < llength xs then lnth xs n else lnth ys (n − the-enat (llength xs)))
⟨*proof*⟩

lemma *lnth-ltake*:
enat m < n ⟹ lnth (ltake n xs) m = lnth xs m
⟨*proof*⟩

lemma *ldropn-Suc-conv-ldropn*:
enat n < llength xs ⟹ LCons (lnth xs n) (ldropn (Suc n) xs) = ldropn n xs
⟨*proof*⟩

lemma *ltake-Suc-conv-snoc-lnth*:
enat m < llength xs ⟹ ltake (enat (Suc m)) xs = lappend (ltake (enat m) xs) (LCons (lnth xs m) LNil)
⟨*proof*⟩

lemma *lappend-eq-lappend-conv*:
assumes *len*: *llength xs = llength us*
shows *lappend xs ys = lappend us vs ⟷*
xs = us ∧ (lfinite xs ⟶ ys = vs) (**is** *?lhs ⟷ ?rhs*)
⟨*proof*⟩

2.15 iterates

lemmas *iterates* [*code*, *nitpick-simp*] = *iterates.ctr*
and *lnull-iterates* = *iterates.simps*(*1*)
and *lhd-iterates* = *iterates.simps*(*2*)
and *ltl-iterates* = *iterates.simps*(*3*)

lemma *lfinite-iterates* [*iff*]: ¬ *lfinite* (*iterates f x*)
⟨*proof*⟩

lemma *lmap-iterates*: *lmap f* (*iterates f x*) = *iterates f* (*f x*)
⟨*proof*⟩

lemma *iterates-lmap*: *iterates f x = LCons x* (*lmap f* (*iterates f x*))
⟨*proof*⟩

lemma *lappend-iterates*: *lappend* (*iterates f x*) *xs = iterates f x*
⟨*proof*⟩

lemma [*simp*]:
fixes *f* :: *'a ⇒ 'a*
shows *lnull-funpow-lmap*: *lnull* ((*lmap f* ^^ *n*) *xs*) ⟷ *lnull xs*
and *lhd-funpow-lmap*: ¬ *lnull xs ⟹ lhd* ((*lmap f* ^^ *n*) *xs*) = (*f* ^^ *n*) (*lhd xs*)
and *ltl-funpow-lmap*: ¬ *lnull xs ⟹ ltl* ((*lmap f* ^^ *n*) *xs*) = (*lmap f* ^^ *n*) (*ltl xs*)
⟨*proof*⟩

lemma *iterates-equality*:
assumes *h*: ⋀*x*. *h x = LCons x* (*lmap f* (*h x*))
shows *h = iterates f*
⟨*proof*⟩

lemma *llength-iterates* [*simp*]: *llength* (*iterates f x*) = ∞
⟨*proof*⟩

lemma *ldropn-iterates*: *ldropn n* (*iterates f x*) = *iterates f* ((*f* ^^ *n*) *x*)
⟨*proof*⟩

lemma *ldrop-iterates*: *ldrop* (*enat n*) (*iterates f x*) = *iterates f* ((*f* ^^ *n*) *x*)
⟨*proof*⟩

lemma *lnth-iterates* [*simp*]: *lnth* (*iterates f x*) *n* = (*f* ^^ *n*) *x*
⟨*proof*⟩

lemma *lset-iterates*:
lset (*iterates f x*) = {(*f* ^^ *n*) *x*|*n*. *True*}
⟨*proof*⟩

lemma *lset-repeat* [*simp*]: *lset* (*repeat x*) = {*x*}
⟨*proof*⟩

2.16 More on the prefix ordering on lazy lists: (⊑) and *lstrict-prefix*

lemma *lstrict-prefix-code* [*code*, *simp*]:
lstrict-prefix LNil LNil ⟷ *False*
lstrict-prefix LNil (*LCons y ys*) ⟷ *True*
lstrict-prefix (*LCons x xs*) *LNil* ⟷ *False*
lstrict-prefix (*LCons x xs*) (*LCons y ys*) ⟷ *x* = *y* ∧ *lstrict-prefix xs ys*
⟨*proof*⟩

lemma *lmap-lprefix*: *xs* ⊑ *ys* ⟹ *lmap f xs* ⊑ *lmap f ys*
⟨*proof*⟩

lemma *lprefix-llength-eq-imp-eq*:
⟦ *xs* ⊑ *ys*; *llength xs* = *llength ys* ⟧ ⟹ *xs* = *ys*
⟨*proof*⟩

lemma *lprefix-llength-le*: *xs* ⊑ *ys* ⟹ *llength xs* ≤ *llength ys*
⟨*proof*⟩

lemma *lstrict-prefix-llength-less*:
assumes *lstrict-prefix xs ys*
shows *llength xs* < *llength ys*
⟨*proof*⟩

lemma *lstrict-prefix-lfinite1*: *lstrict-prefix xs ys* ⟹ *lfinite xs*
⟨*proof*⟩

lemma *wfP-lstrict-prefix*: *wfP lstrict-prefix*
⟨*proof*⟩

lemma *llist-less-induct*[*case-names less*]:
(⋀*xs*. (⋀*ys*. *lstrict-prefix ys xs* ⟹ *P ys*) ⟹ *P xs*) ⟹ *P xs*
⟨*proof*⟩

lemma *ltake-enat-eq-imp-eq*: $(\bigwedge n.\ ltake\ (enat\ n)\ xs = ltake\ (enat\ n)\ ys) \Longrightarrow xs = ys$
⟨*proof*⟩

lemma *ltake-enat-lprefix-imp-lprefix*:
assumes $\bigwedge n.\ lprefix\ (ltake\ (enat\ n)\ xs)\ (ltake\ (enat\ n)\ ys)$
shows *lprefix xs ys*
⟨*proof*⟩

lemma *lprefix-conv-lappend*: $xs \sqsubseteq ys \longleftrightarrow (\exists\, zs.\ ys = lappend\ xs\ zs)$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *lappend-lprefixE*:
assumes *lappend xs ys* $\sqsubseteq$ *zs*
obtains zs' **where** $zs = lappend\ xs\ zs'$
⟨*proof*⟩

lemma *lprefix-lfiniteD*:
$[\![\ xs \sqsubseteq ys;\ lfinite\ ys\ ]\!] \Longrightarrow lfinite\ xs$
⟨*proof*⟩

lemma *lprefix-lappendD*:
assumes $xs \sqsubseteq lappend\ ys\ zs$
shows $xs \sqsubseteq ys \vee ys \sqsubseteq xs$
⟨*proof*⟩

lemma *lstrict-prefix-lappend-conv*:
$lstrict\text{-}prefix\ xs\ (lappend\ xs\ ys) \longleftrightarrow lfinite\ xs \wedge \neg\ lnull\ ys$
⟨*proof*⟩

lemma *lprefix-llist-ofI*:
$\exists\, zs.\ ys = xs\ @\ zs \Longrightarrow llist\text{-}of\ xs \sqsubseteq llist\text{-}of\ ys$
⟨*proof*⟩

lemma *lprefix-llist-of* [*simp*]: $llist\text{-}of\ xs \sqsubseteq llist\text{-}of\ ys \longleftrightarrow prefix\ xs\ ys$
⟨*proof*⟩

lemma *llimit-induct* [*case-names LNil LCons limit*]:
— The limit case is just an instance of admissibility
assumes *LNil*: *P LNil*
and *LCons*: $\bigwedge x\ xs.\ [\![\ lfinite\ xs;\ P\ xs\ ]\!] \Longrightarrow P\ (LCons\ x\ xs)$
and *limit*: $(\bigwedge ys.\ lstrict\text{-}prefix\ ys\ xs \Longrightarrow P\ ys) \Longrightarrow P\ xs$
shows *P xs*
⟨*proof*⟩

lemma *lmap-lstrict-prefix*:
$lstrict\text{-}prefix\ xs\ ys \Longrightarrow lstrict\text{-}prefix\ (lmap\ f\ xs)\ (lmap\ f\ ys)$

⟨*proof*⟩

lemma *lprefix-lnthD*:
assumes *xs* ⊑ *ys* **and** *enat n* < *llength xs*
shows *lnth xs n* = *lnth ys n*
⟨*proof*⟩

lemma *lfinite-lSup-chain*:
assumes *chain*: *Complete-Partial-Order.chain* (⊑) *A*
shows *lfinite* (*lSup A*) ⟷ *finite A* ∧ (∀ *xs* ∈ *A*. *lfinite xs*) (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

Setup for (⊑) for Nitpick

definition *finite-lprefix* :: *'a llist* ⇒ *'a llist* ⇒ *bool*
where *finite-lprefix* = (⊑)

lemma *finite-lprefix-nitpick-simps* [*nitpick-simp*]:
finite-lprefix xs LNil ⟷ *xs* = *LNil*
finite-lprefix LNil xs ⟷ *True*
finite-lprefix xs (*LCons y ys*) ⟷
xs = *LNil* ∨ (∃ *xs'*. *xs* = *LCons y xs'* ∧ *finite-lprefix xs' ys*)
⟨*proof*⟩

lemma *lprefix-nitpick-simps* [*nitpick-simp*]:
xs ⊑ *ys* = (*if lfinite xs then finite-lprefix xs ys else xs* = *ys*)
⟨*proof*⟩

hide-const (**open**) *finite-lprefix*
hide-fact (**open**) *finite-lprefix-def finite-lprefix-nitpick-simps lprefix-nitpick-simps*

2.17 Length of the longest common prefix

lemma *llcp-simps* [*simp*, *code*, *nitpick-simp*]:
shows *llcp-LNil1*: *llcp LNil ys* = *0*
and *llcp-LNil2*: *llcp xs LNil* = *0*
and *llcp-LCons*: *llcp* (*LCons x xs*) (*LCons y ys*) = (*if x* = *y then eSuc* (*llcp xs ys*) *else 0*)
⟨*proof*⟩

lemma *llcp-eq-0-iff*:
llcp xs ys = *0* ⟷ *lnull xs* ∨ *lnull ys* ∨ *lhd xs* ≠ *lhd ys*
⟨*proof*⟩

lemma *epred-llcp*:
⟦ ¬ *lnull xs*; ¬ *lnull ys*; *lhd xs* = *lhd ys* ⟧
⟹ *epred* (*llcp xs ys*) = *llcp* (*ltl xs*) (*ltl ys*)
⟨*proof*⟩

lemma *llcp-commute*: *llcp xs ys* = *llcp ys xs*

⟨*proof*⟩

lemma *llcp-same-conv-length* [*simp*]: *llcp xs xs* = *llength xs*
⟨*proof*⟩

lemma *llcp-lappend-same* [*simp*]:
llcp (*lappend xs ys*) (*lappend xs zs*) = *llength xs* + *llcp ys zs*
⟨*proof*⟩

lemma *llcp-lprefix1* [*simp*]: *xs* ⊑ *ys* ⟹ *llcp xs ys* = *llength xs*
⟨*proof*⟩

lemma *llcp-lprefix2* [*simp*]: *ys* ⊑ *xs* ⟹ *llcp xs ys* = *llength ys*
⟨*proof*⟩

lemma *llcp-le-length*: *llcp xs ys* ≤ *min* (*llength xs*) (*llength ys*)
⟨*proof*⟩

lemma *llcp-ltake1*: *llcp* (*ltake n xs*) *ys* = *min n* (*llcp xs ys*)
⟨*proof*⟩

lemma *llcp-ltake2*: *llcp xs* (*ltake n ys*) = *min n* (*llcp xs ys*)
⟨*proof*⟩

lemma *llcp-ltake* [*simp*]: *llcp* (*ltake n xs*) (*ltake m ys*) = *min* (*min n m*) (*llcp xs ys*)
⟨*proof*⟩

2.18 Zipping two lazy lists to a lazy list of pairs *lzip*

lemma *lzip-simps* [*simp*, *code*, *nitpick-simp*]:
lzip LNil ys = *LNil*
lzip xs LNil = *LNil*
lzip (*LCons x xs*) (*LCons y ys*) = *LCons* (*x*, *y*) (*lzip xs ys*)
⟨*proof*⟩

lemma *lnull-lzip* [*simp*]: *lnull* (*lzip xs ys*) ⟷ *lnull xs* ∨ *lnull ys*
⟨*proof*⟩

lemma *lzip-eq-LNil-conv*: *lzip xs ys* = *LNil* ⟷ *xs* = *LNil* ∨ *ys* = *LNil*
⟨*proof*⟩

lemmas *lhd-lzip* = *lzip.sel*(*1*)
and *ltl-lzip* = *lzip.sel*(*2*)

lemma *lzip-eq-LCons-conv*:
lzip xs ys = *LCons z zs* ⟷
(∃ *x xs′ y ys′*. *xs* = *LCons x xs′* ∧ *ys* = *LCons y ys′* ∧ *z* = (*x*, *y*) ∧ *zs* = *lzip xs′ ys′*)

⟨*proof*⟩

lemma *lzip-lappend*:
  llength xs = *llength us*
  ⟹ *lzip* (*lappend xs ys*) (*lappend us vs*) = *lappend* (*lzip xs us*) (*lzip ys vs*)
⟨*proof*⟩

lemma *llength-lzip* [*simp*]:
  llength (*lzip xs ys*) = *min* (*llength xs*) (*llength ys*)
⟨*proof*⟩

lemma *ltake-lzip*: *ltake n* (*lzip xs ys*) = *lzip* (*ltake n xs*) (*ltake n ys*)
⟨*proof*⟩

lemma *ldropn-lzip* [*simp*]:
  ldropn n (*lzip xs ys*) = *lzip* (*ldropn n xs*) (*ldropn n ys*)
⟨*proof*⟩

lemma
  fixes *F*
  defines *F* ≡ λ*lzip* (*xs*, *ys*). *case xs of LNil* ⇒ *LNil* | *LCons x xs′* ⇒ *case ys of LNil* ⇒ *LNil* | *LCons y ys′* ⇒ *LCons* (*x*, *y*) (*curry lzip xs′ ys′*)
  shows *lzip-conv-fixp*: *lzip* ≡ *curry* (*ccpo.fixp* (*fun-lub lSup*) (*fun-ord* (⊑)) *F*) (**is** *?lhs* ≡ *?rhs*)
  and *lzip-mono*: *mono-llist* (λ*lzip*. *F lzip xs*) (**is** *?mono xs*)
⟨*proof*⟩

lemma *monotone-lzip*: *monotone* (*rel-prod* (⊑) (⊑)) (⊑) (*case-prod lzip*)
⟨*proof*⟩

lemma *mono2mono-lzip1* [*THEN llist.mono2mono*, *cont-intro*, *simp*]:
  shows *monotone-lzip1*: *monotone* (⊑) (⊑) (λ*xs*. *lzip xs ys*)
⟨*proof*⟩

lemma *mono2mono-lzip2* [*THEN llist.mono2mono*, *cont-intro*, *simp*]:
  shows *monotone-lzip2*: *monotone* (⊑) (⊑) (λ*ys*. *lzip xs ys*)
⟨*proof*⟩

lemma *mcont-lzip*: *mcont* (*prod-lub lSup lSup*) (*rel-prod* (⊑) (⊑)) *lSup* (⊑) (*case-prod lzip*)
⟨*proof*⟩

lemma *mcont2mcont-lzip1* [*THEN llist.mcont2mcont*, *cont-intro*, *simp*]:
  shows *mcont-lzip1*: *mcont lSup* (⊑) *lSup* (⊑) (λ*xs*. *lzip xs ys*)
⟨*proof*⟩

lemma *mcont2mcont-lzip2* [*THEN llist.mcont2mcont*, *cont-intro*, *simp*]:
  shows *mcont-lzip2*: *mcont lSup* (⊑) *lSup* (⊑) (λ*ys*. *lzip xs ys*)
⟨*proof*⟩

lemma *ldrop-lzip* [*simp*]: *ldrop n* (*lzip xs ys*) = *lzip* (*ldrop n xs*) (*ldrop n ys*)
⟨*proof*⟩

lemma *lzip-iterates*:
lzip (*iterates f x*) (*iterates g y*) = *iterates* (λ(*x*, *y*). (*f x*, *g y*)) (*x*, *y*)
⟨*proof*⟩

lemma *lzip-llist-of* [*simp*]:
lzip (*llist-of xs*) (*llist-of ys*) = *llist-of* (*zip xs ys*)
⟨*proof*⟩

lemma *lnth-lzip*:
⟦ *enat n* < *llength xs*; *enat n* < *llength ys* ⟧
⟹ *lnth* (*lzip xs ys*) *n* = (*lnth xs n*, *lnth ys n*)
⟨*proof*⟩

lemma *lset-lzip*:
lset (*lzip xs ys*) =
{(*lnth xs n*, *lnth ys n*)|*n*. *enat n* < *min* (*llength xs*) (*llength ys*)}
⟨*proof*⟩

lemma *lset-lzipD1*: (*x*, *y*) ∈ *lset* (*lzip xs ys*) ⟹ *x* ∈ *lset xs*
⟨*proof*⟩

lemma *lset-lzipD2*: (*x*, *y*) ∈ *lset* (*lzip xs ys*) ⟹ *y* ∈ *lset ys*
⟨*proof*⟩

lemma *lset-lzip-same* [*simp*]: *lset* (*lzip xs xs*) = (λ*x*. (*x*, *x*)) ‘ *lset xs*
⟨*proof*⟩

lemma *lfinite-lzip* [*simp*]:
lfinite (*lzip xs ys*) ⟷ *lfinite xs* ∨ *lfinite ys* (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *lzip-eq-lappend-conv*:
assumes *eq*: *lzip xs ys* = *lappend us vs*
shows ∃ *xs′ xs′′ ys′ ys′′*. *xs* = *lappend xs′ xs′′* ∧ *ys* = *lappend ys′ ys′′* ∧
llength xs′ = *llength ys′* ∧ *us* = *lzip xs′ ys′* ∧
vs = *lzip xs′′ ys′′*
⟨*proof*⟩

lemma *lzip-lmap* [*simp*]:
lzip (*lmap f xs*) (*lmap g ys*) = *lmap* (λ(*x*, *y*). (*f x*, *g y*)) (*lzip xs ys*)
⟨*proof*⟩

lemma *lzip-lmap1*:
lzip (*lmap f xs*) *ys* = *lmap* (λ(*x*, *y*). (*f x*, *y*)) (*lzip xs ys*)
⟨*proof*⟩

lemma *lzip-lmap2*:
lzip xs (*lmap f ys*) = *lmap* (λ(*x*, *y*). (*x*, *f y*)) (*lzip xs ys*)
⟨*proof*⟩

lemma *lmap-fst-lzip-conv-ltake*:
lmap fst (*lzip xs ys*) = *ltake* (*min* (*llength xs*) (*llength ys*)) *xs*
⟨*proof*⟩

lemma *lmap-snd-lzip-conv-ltake*:
lmap snd (*lzip xs ys*) = *ltake* (*min* (*llength xs*) (*llength ys*)) *ys*
⟨*proof*⟩

lemma *lzip-conv-lzip-ltake-min-llength*:
lzip xs ys =
lzip (*ltake* (*min* (*llength xs*) (*llength ys*)) *xs*)
(*ltake* (*min* (*llength xs*) (*llength ys*)) *ys*)
⟨*proof*⟩

2.19 Taking and dropping from a lazy list: *ltakeWhile* and *ldropWhile*

lemma *ltakeWhile-simps* [*simp*, *code*, *nitpick-simp*]:
shows *ltakeWhile-LNil*: *ltakeWhile P LNil* = *LNil*
and *ltakeWhile-LCons*: *ltakeWhile P* (*LCons x xs*) = (*if P x then LCons x* (*ltakeWhile P xs*) *else LNil*)
⟨*proof*⟩

lemma *ldropWhile-simps* [*simp*, *code*]:
shows *ldropWhile-LNil*: *ldropWhile P LNil* = *LNil*
and *ldropWhile-LCons*: *ldropWhile P* (*LCons x xs*) = (*if P x then ldropWhile P xs else LCons x xs*)
⟨*proof*⟩

lemma fixes *f F P*
defines *F* ≡ λ*ltakeWhile xs*. *case xs of LNil* ⇒ *LNil* | *LCons x xs* ⇒ *if P x then LCons x* (*ltakeWhile xs*) *else LNil*
shows *ltakeWhile-conv-fixp*: *ltakeWhile P* ≡ *ccpo.fixp* (*fun-lub lSup*) (*fun-ord lprefix*) *F* (**is** *?lhs* ≡ *?rhs*)
and *ltakeWhile-mono*: ⋀*xs*. *mono-llist* (λ*ltakeWhile*. *F ltakeWhile xs*) (**is** *PROP ?mono*)
⟨*proof*⟩

lemma *mono2mono-ltakeWhile*[*THEN llist.mono2mono*, *cont-intro*, *simp*]:
shows *monotone-ltakeWhile*: *monotone lprefix lprefix* (*ltakeWhile P*)
⟨*proof*⟩

lemma *mcont2mcont-ltakeWhile* [*THEN llist.mcont2mcont*, *cont-intro*, *simp*]:
shows *mcont-ltakeWhile*: *mcont lSup lprefix lSup lprefix* (*ltakeWhile P*)

⟨*proof*⟩

lemma *mono-llist-ltakeWhile* [*partial-function-mono*]:
  mono-llist F ⟹ *mono-llist* (λ*f*. *ltakeWhile P* (*F f*))
⟨*proof*⟩

lemma *mono2mono-ldropWhile* [*THEN llist.mono2mono*, *cont-intro*, *simp*]:
  shows *monotone-ldropWhile*: *monotone* (⊑) (⊑) (*ldropWhile P*)
⟨*proof*⟩

lemma *mcont2mcont-ldropWhile* [*THEN llist.mcont2mcont*, *cont-intro*, *simp*]:
  shows *mcont-ldropWhile*: *mcont lSup* (⊑) *lSup* (⊑) (*ldropWhile P*)
⟨*proof*⟩

lemma *lnull-ltakeWhile* [*simp*]: *lnull* (*ltakeWhile P xs*) ⟷ (¬ *lnull xs* ⟶ ¬ *P* (*lhd xs*))
⟨*proof*⟩

lemma *ltakeWhile-eq-LNil-iff*: *ltakeWhile P xs* = *LNil* ⟷ (*xs* ≠ *LNil* ⟶ ¬ *P* (*lhd xs*))
⟨*proof*⟩

lemmas *lhd-ltakeWhile* = *ltakeWhile.sel*(*1*)

lemma *ltl-ltakeWhile*:
  ltl (*ltakeWhile P xs*) = (*if P* (*lhd xs*) *then ltakeWhile P* (*ltl xs*) *else LNil*)
⟨*proof*⟩

lemma *lprefix-ltakeWhile*: *ltakeWhile P xs* ⊑ *xs*
⟨*proof*⟩

lemma *llength-ltakeWhile-le*: *llength* (*ltakeWhile P xs*) ≤ *llength xs*
⟨*proof*⟩

lemma *ltakeWhile-nth*: *enat i* < *llength* (*ltakeWhile P xs*) ⟹ *lnth* (*ltakeWhile P xs*) *i* = *lnth xs i*
⟨*proof*⟩

lemma *ltakeWhile-all*: ∀ *x*∈*lset xs*. *P x* ⟹ *ltakeWhile P xs* = *xs*
⟨*proof*⟩

lemma *lset-ltakeWhileD*:
  assumes *x* ∈ *lset* (*ltakeWhile P xs*)
  shows *x* ∈ *lset xs* ∧ *P x*
⟨*proof*⟩

lemma *lset-ltakeWhile-subset*:
  lset (*ltakeWhile P xs*) ⊆ *lset xs* ∩ {*x*. *P x*}
⟨*proof*⟩

lemma *ltakeWhile-all-conv*: *ltakeWhile P xs = xs ⟷ lset xs ⊆ {x. P x}*
⟨*proof*⟩

lemma *llength-ltakeWhile-all*: *llength (ltakeWhile P xs) = llength xs ⟷ ltakeWhile P xs = xs*
⟨*proof*⟩

lemma *ldropWhile-eq-LNil-iff*: *ldropWhile P xs = LNil ⟷ (∀ x ∈ lset xs. P x)*
⟨*proof*⟩

lemma *lnull-ldropWhile* [*simp*]:
lnull (ldropWhile P xs) ⟷ (∀ x ∈ lset xs. P x) (**is** *?lhs ⟷ ?rhs*)
⟨*proof*⟩

lemma *lset-ldropWhile-subset*:
lset (ldropWhile P xs) ⊆ lset xs
⟨*proof*⟩

lemma *in-lset-ldropWhileD*: *x ∈ lset (ldropWhile P xs) ⟹ x ∈ lset xs*
⟨*proof*⟩

lemma *ltakeWhile-lmap*: *ltakeWhile P (lmap f xs) = lmap f (ltakeWhile (P ∘ f) xs)*
⟨*proof*⟩

lemma *ldropWhile-lmap*: *ldropWhile P (lmap f xs) = lmap f (ldropWhile (P ∘ f) xs)*
⟨*proof*⟩

lemma *llength-ltakeWhile-lt-iff*: *llength (ltakeWhile P xs) < llength xs ⟷ (∃ x∈lset xs. ¬ P x)*
(**is** *?lhs ⟷ ?rhs*)
⟨*proof*⟩

lemma *ltakeWhile-K-False* [*simp*]: *ltakeWhile (λ-. False) xs = LNil*
⟨*proof*⟩

lemma *ltakeWhile-K-True* [*simp*]: *ltakeWhile (λ-. True) xs = xs*
⟨*proof*⟩

lemma *ldropWhile-K-False* [*simp*]: *ldropWhile (λ-. False) = id*
⟨*proof*⟩

lemma *ldropWhile-K-True* [*simp*]: *ldropWhile (λ-. True) xs = LNil*
⟨*proof*⟩

lemma *lappend-ltakeWhile-ldropWhile* [*simp*]:
lappend (ltakeWhile P xs) (ldropWhile P xs) = xs

⟨*proof*⟩

lemma *ltakeWhile-lappend*:
ltakeWhile P (*lappend xs ys*) =
(*if* ∃ *x*∈*lset xs*. ¬ *P x then ltakeWhile P xs*
else lappend xs (*ltakeWhile P ys*))
⟨*proof*⟩

lemma *ldropWhile-lappend*:
ldropWhile P (*lappend xs ys*) =
(*if* ∃ *x*∈*lset xs*. ¬ *P x then lappend* (*ldropWhile P xs*) *ys*
else if lfinite xs then ldropWhile P ys else LNil)
⟨*proof*⟩

lemma *lfinite-ltakeWhile*:
lfinite (*ltakeWhile P xs*) ⟷ *lfinite xs* ∨ (∃ *x* ∈ *lset xs*. ¬ *P x*) (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *llength-ltakeWhile-eq-infinity*:
llength (*ltakeWhile P xs*) = ∞ ⟷ ¬ *lfinite xs* ∧ *ltakeWhile P xs* = *xs*
⟨*proof*⟩

lemma *llength-ltakeWhile-eq-infinity′*:
llength (*ltakeWhile P xs*) = ∞ ⟷ ¬ *lfinite xs* ∧ (∀ *x*∈*lset xs*. *P x*)
⟨*proof*⟩

lemma *lzip-ltakeWhile-fst*: *lzip* (*ltakeWhile P xs*) *ys* = *ltakeWhile* (*P* ∘ *fst*) (*lzip xs ys*)
⟨*proof*⟩

lemma *lzip-ltakeWhile-snd*: *lzip xs* (*ltakeWhile P ys*) = *ltakeWhile* (*P* ∘ *snd*) (*lzip xs ys*)
⟨*proof*⟩

lemma *ltakeWhile-lappend1*:
⟦ *x* ∈ *lset xs*; ¬ *P x* ⟧ ⟹ *ltakeWhile P* (*lappend xs ys*) = *ltakeWhile P xs*
⟨*proof*⟩

lemma *ltakeWhile-lappend2*:
lset xs ⊆ {*x*. *P x*}
⟹ *ltakeWhile P* (*lappend xs ys*) = *lappend xs* (*ltakeWhile P ys*)
⟨*proof*⟩

lemma *ltakeWhile-cong* [*cong*, *fundef-cong*]:
assumes *xs*: *xs* = *ys*
and *PQ*: ⋀*x*. *x* ∈ *lset ys* ⟹ *P x* = *Q x*
shows *ltakeWhile P xs* = *ltakeWhile Q ys*
⟨*proof*⟩

lemma *lnth-llength-ltakeWhile*:
assumes *len*: *llength* (*ltakeWhile P xs*) < *llength xs*
shows ¬ *P* (*lnth xs* (*the-enat* (*llength* (*ltakeWhile P xs*))))
⟨*proof*⟩

lemma assumes ∃ *x*∈*lset xs*. ¬ *P x*
shows *lhd-ldropWhile*: ¬ *P* (*lhd* (*ldropWhile P xs*)) (**is** *?thesis1*)
and *lhd-ldropWhile-in-lset*: *lhd* (*ldropWhile P xs*) ∈ *lset xs* (**is** *?thesis2*)
⟨*proof*⟩

lemma *ldropWhile-eq-ldrop*:
ldropWhile P xs = *ldrop* (*llength* (*ltakeWhile P xs*)) *xs*
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *ldropWhile-cong* [*cong*]:
⟦ *xs* = *ys*; ⋀*x*. *x* ∈ *lset ys* ⟹ *P x* = *Q x* ⟧ ⟹ *ldropWhile P xs* = *ldropWhile Q ys*
⟨*proof*⟩

lemma *ltakeWhile-repeat*:
ltakeWhile P (*repeat x*) = (*if P x then repeat x else LNil*)
⟨*proof*⟩

lemma *ldropWhile-repeat*: *ldropWhile P* (*repeat x*) = (*if P x then LNil else repeat x*)
⟨*proof*⟩

lemma *lfinite-ldropWhile*: *lfinite* (*ldropWhile P xs*) ⟷ (∃ *x* ∈ *lset xs*. ¬ *P x*) ⟶ *lfinite xs*
⟨*proof*⟩

lemma *llength-ldropWhile*:
llength (*ldropWhile P xs*) =
(*if* ∃ *x*∈*lset xs*. ¬ *P x then llength xs* − *llength* (*ltakeWhile P xs*) *else 0*)
⟨*proof*⟩

lemma *lhd-ldropWhile-conv-lnth*:
∃ *x*∈*lset xs*. ¬ *P x* ⟹ *lhd* (*ldropWhile P xs*) = *lnth xs* (*the-enat* (*llength* (*ltakeWhile P xs*)))
⟨*proof*⟩

2.20 *llist-all2*

lemmas *llist-all2-LNil-LNil* = *llist.rel-inject*(*1*)
lemmas *llist-all2-LNil-LCons* = *llist.rel-distinct*(*1*)
lemmas *llist-all2-LCons-LNil* = *llist.rel-distinct*(*2*)
lemmas *llist-all2-LCons-LCons* = *llist.rel-inject*(*2*)

lemma *llist-all2-LNil1* [*simp*]: *llist-all2 P LNil xs* $\longleftrightarrow$ *xs* = *LNil*
⟨*proof*⟩

lemma *llist-all2-LNil2* [*simp*]: *llist-all2 P xs LNil* $\longleftrightarrow$ *xs* = *LNil*
⟨*proof*⟩

lemma *llist-all2-LCons1*:
llist-all2 P (*LCons x xs*) *ys* $\longleftrightarrow$ ($\exists$ *y ys'*. *ys* = *LCons y ys'* $\wedge$ *P x y* $\wedge$ *llist-all2 P xs ys'*)
⟨*proof*⟩

lemma *llist-all2-LCons2*:
llist-all2 P xs (*LCons y ys*) $\longleftrightarrow$ ($\exists$ *x xs'*. *xs* = *LCons x xs'* $\wedge$ *P x y* $\wedge$ *llist-all2 P xs' ys*)
⟨*proof*⟩

lemma *llist-all2-llist-of* [*simp*]:
llist-all2 P (*llist-of xs*) (*llist-of ys*) = *list-all2 P xs ys*
⟨*proof*⟩

lemma *llist-all2-conv-lzip*:
llist-all2 P xs ys $\longleftrightarrow$ *llength xs* = *llength ys* $\wedge$ ($\forall$ (*x*, *y*) $\in$ *lset* (*lzip xs ys*). *P x y*)
⟨*proof*⟩

lemma *llist-all2-llengthD*:
llist-all2 P xs ys $\Longrightarrow$ *llength xs* = *llength ys*
⟨*proof*⟩

lemma *llist-all2-lnullD*: *llist-all2 P xs ys* $\Longrightarrow$ *lnull xs* $\longleftrightarrow$ *lnull ys*
⟨*proof*⟩

lemma *llist-all2-all-lnthI*:
⟦ *llength xs* = *llength ys*;
$\bigwedge$*n*. *enat n* < *llength xs* $\Longrightarrow$ *P* (*lnth xs n*) (*lnth ys n*) ⟧
$\Longrightarrow$ *llist-all2 P xs ys*
⟨*proof*⟩

lemma *llist-all2-lnthD*:
⟦ *llist-all2 P xs ys*; *enat n* < *llength xs* ⟧ $\Longrightarrow$ *P* (*lnth xs n*) (*lnth ys n*)
⟨*proof*⟩

lemma *llist-all2-lnthD2*:
⟦ *llist-all2 P xs ys*; *enat n* < *llength ys* ⟧ $\Longrightarrow$ *P* (*lnth xs n*) (*lnth ys n*)
⟨*proof*⟩

lemma *llist-all2-conv-all-lnth*:
llist-all2 P xs ys $\longleftrightarrow$
llength xs = *llength ys* $\wedge$
($\forall$ *n*. *enat n* < *llength ys* $\longrightarrow$ *P* (*lnth xs n*) (*lnth ys n*))

⟨*proof*⟩

lemma *llist-all2-True* [*simp*]: *llist-all2* (λ- -. *True*) *xs ys* ⟷ *llength xs* = *llength ys*
⟨*proof*⟩

lemma *llist-all2-reflI*:
(⋀*x*. *x* ∈ *lset xs* ⟹ *P x x*) ⟹ *llist-all2 P xs xs*
⟨*proof*⟩

lemma *llist-all2-lmap1*:
llist-all2 P (*lmap f xs*) *ys* ⟷ *llist-all2* (λ*x*. *P* (*f x*)) *xs ys*
⟨*proof*⟩

lemma *llist-all2-lmap2*:
llist-all2 P xs (*lmap g ys*) ⟷ *llist-all2* (λ*x y*. *P x* (*g y*)) *xs ys*
⟨*proof*⟩

lemma *llist-all2-lfiniteD*:
llist-all2 P xs ys ⟹ *lfinite xs* ⟷ *lfinite ys*
⟨*proof*⟩

lemma *llist-all2-coinduct*[*consumes 1*, *case-names LNil LCons*, *case-conclusion LCons lhd ltl*, *coinduct pred*]:
assumes *major*: *X xs ys*
and *step*:
⋀*xs ys*. *X xs ys* ⟹ *lnull xs* ⟷ *lnull ys*
⋀*xs ys*. ⟦ *X xs ys*; ¬ *lnull xs*; ¬ *lnull ys* ⟧ ⟹ *P* (*lhd xs*) (*lhd ys*) ∧ (*X* (*ltl xs*) (*ltl ys*) ∨ *llist-all2 P* (*ltl xs*) (*ltl ys*))
shows *llist-all2 P xs ys*
⟨*proof*⟩

lemma *llist-all2-cases*[*consumes 1*, *case-names LNil LCons*, *cases pred*]:
assumes *llist-all2 P xs ys*
obtains (*LNil*) *xs* = *LNil ys* = *LNil*
| (*LCons*) *x xs*′ *y ys*′
where *xs* = *LCons x xs*′ **and** *ys* = *LCons y ys*′
and *P x y* **and** *llist-all2 P xs*′ *ys*′
⟨*proof*⟩

lemma *llist-all2-mono*:
⟦ *llist-all2 P xs ys*; ⋀*x y*. *P x y* ⟹ *P*′ *x y* ⟧ ⟹ *llist-all2 P*′ *xs ys*
⟨*proof*⟩

lemma *llist-all2-left*: *llist-all2* (λ*x* -. *P x*) *xs ys* ⟷ *llength xs* = *llength ys* ∧ (∀ *x*∈*lset xs*. *P x*)
⟨*proof*⟩

lemma *llist-all2-right*: *llist-all2* (λ-. *P*) *xs ys* ⟷ *llength xs* = *llength ys* ∧ (∀ *x*∈*lset*

ys. P x)
⟨*proof*⟩

lemma *llist-all2-lsetD1*: ⟦ *llist-all2 P xs ys*; *x* ∈ *lset xs* ⟧ ⟹ ∃ *y*∈*lset ys. P x y*
⟨*proof*⟩

lemma *llist-all2-lsetD2*: ⟦ *llist-all2 P xs ys*; *y* ∈ *lset ys* ⟧ ⟹ ∃ *x*∈*lset xs. P x y*
⟨*proof*⟩

lemma *llist-all2-conj*:
llist-all2 (λ*x y. P x y* ∧ *Q x y*) *xs ys* ⟷ *llist-all2 P xs ys* ∧ *llist-all2 Q xs ys*
⟨*proof*⟩

lemma *llist-all2-lhdD*:
⟦ *llist-all2 P xs ys*; ¬ *lnull xs* ⟧ ⟹ *P* (*lhd xs*) (*lhd ys*)
⟨*proof*⟩

lemma *llist-all2-lhdD2*:
⟦ *llist-all2 P xs ys*; ¬ *lnull ys* ⟧ ⟹ *P* (*lhd xs*) (*lhd ys*)
⟨*proof*⟩

lemma *llist-all2-ltlI*:
llist-all2 P xs ys ⟹ *llist-all2 P* (*ltl xs*) (*ltl ys*)
⟨*proof*⟩

lemma *llist-all2-lappendI*:
assumes *1*: *llist-all2 P xs ys*
and *2*: ⟦ *lfinite xs*; *lfinite ys* ⟧ ⟹ *llist-all2 P xs′ ys′*
shows *llist-all2 P* (*lappend xs xs′*) (*lappend ys ys′*)
⟨*proof*⟩

lemma *llist-all2-lappend1D*:
assumes *llist-all2 P* (*lappend xs xs′*) *ys*
shows *llist-all2 P xs* (*ltake* (*llength xs*) *ys*)
and *lfinite xs* ⟹ *llist-all2 P xs′* (*ldrop* (*llength xs*) *ys*)
⟨*proof*⟩

lemma *lmap-eq-lmap-conv-llist-all2*:
lmap f xs = *lmap g ys* ⟷ *llist-all2* (λ*x y. f x* = *g y*) *xs ys* (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *llist-all2-expand*:
⟦ *lnull xs* ⟷ *lnull ys*;
⟦ ¬ *lnull xs*; ¬ *lnull ys* ⟧ ⟹ *P* (*lhd xs*) (*lhd ys*) ∧ *llist-all2 P* (*ltl xs*) (*ltl ys*) ⟧
⟹ *llist-all2 P xs ys*
⟨*proof*⟩

lemma *llist-all2-llength-ltakeWhileD*:
assumes *major*: *llist-all2 P xs ys*

and *Q*: $\bigwedge x\ y.\ P\ x\ y \Longrightarrow Q1\ x \longleftrightarrow Q2\ y$
shows *llength* (*ltakeWhile Q1 xs*) = *llength* (*ltakeWhile Q2 ys*)
⟨*proof*⟩

lemma *llist-all2-lzipI*:
⟦ *llist-all2 P xs ys*; *llist-all2 P′ xs′ ys′* ⟧
⟹ *llist-all2* (*rel-prod P P′*) (*lzip xs xs′*) (*lzip ys ys′*)
⟨*proof*⟩

lemma *llist-all2-ltakeI*:
llist-all2 P xs ys ⟹ *llist-all2 P* (*ltake n xs*) (*ltake n ys*)
⟨*proof*⟩

lemma *llist-all2-ldropnI*:
llist-all2 P xs ys ⟹ *llist-all2 P* (*ldropn n xs*) (*ldropn n ys*)
⟨*proof*⟩

lemma *llist-all2-ldropI*:
llist-all2 P xs ys ⟹ *llist-all2 P* (*ldrop n xs*) (*ldrop n ys*)
⟨*proof*⟩

lemma *llist-all2-lSupI*:
assumes *Complete-Partial-Order.chain* (*rel-prod* (⊑) (⊑)) *Y* ∀(*xs*, *ys*)∈*Y*. *llist-all2 P xs ys*
shows *llist-all2 P* (*lSup* (*fst* ‘ *Y*)) (*lSup* (*snd* ‘ *Y*))
⟨*proof*⟩

lemma *admissible-llist-all2* [*cont-intro*, *simp*]:
assumes *f*: *mcont lub ord lSup* (⊑) (λ*x*. *f x*)
and *g*: *mcont lub ord lSup* (⊑) (λ*x*. *g x*)
shows *ccpo.admissible lub ord* (λ*x*. *llist-all2 P* (*f x*) (*g x*))
⟨*proof*⟩

lemmas [*cont-intro*] =
ccpo.mcont2mcont[*OF llist-ccpo - mcont-fst*]
ccpo.mcont2mcont[*OF llist-ccpo - mcont-snd*]

lemmas *ldropWhile-fixp-parallel-induct* =
parallel-fixp-induct-1-1[*OF llist-partial-function-definitions llist-partial-function-definitions ldropWhile.mono ldropWhile.mono ldropWhile-def ldropWhile-def*, *case-names adm LNil step*]

lemma *llist-all2-ldropWhileI*:
assumes ∗: *llist-all2 P xs ys*
and *Q*: $\bigwedge x\ y.\ P\ x\ y \Longrightarrow Q1\ x \longleftrightarrow Q2\ y$
shows *llist-all2 P* (*ldropWhile Q1 xs*) (*ldropWhile Q2 ys*)
— cannot prove this with parallel induction over *xs* and *ys* because λ*x*. ¬ *llist-all2 P* (*f x*) (*g x*) is not admissible.
⟨*proof*⟩

lemma *llist-all2-same* [*simp*]: *llist-all2 P xs xs* $\longleftrightarrow$ $(\forall x \in lset\ xs.\ P\ x\ x)$
⟨*proof*⟩

lemma *llist-all2-trans*:
⟦ *llist-all2 P xs ys*; *llist-all2 P ys zs*; *transp P* ⟧
$\Longrightarrow$ *llist-all2 P xs zs*
⟨*proof*⟩

2.21 The last element *llast*

lemma *llast-LNil*: *llast LNil* = *undefined*
⟨*proof*⟩

lemma *llast-LCons*: *llast* (*LCons x xs*) = (*if lnull xs then x else llast xs*)
⟨*proof*⟩

lemma *llast-linfinite*: ¬ *lfinite xs* $\Longrightarrow$ *llast xs* = *undefined*
⟨*proof*⟩

lemma [*simp*, *code*]:
shows *llast-singleton*: *llast* (*LCons x LNil*) = *x*
and *llast-LCons2*: *llast* (*LCons x* (*LCons y xs*)) = *llast* (*LCons y xs*)
⟨*proof*⟩

lemma *llast-lappend*:
llast (*lappend xs ys*) = (*if lnull ys then llast xs else if lfinite xs then llast ys else undefined*)
⟨*proof*⟩

lemma *llast-lappend-LCons* [*simp*]:
lfinite xs $\Longrightarrow$ *llast* (*lappend xs* (*LCons y ys*)) = *llast* (*LCons y ys*)
⟨*proof*⟩

lemma *llast-ldropn*: *enat n* < *llength xs* $\Longrightarrow$ *llast* (*ldropn n xs*) = *llast xs*
⟨*proof*⟩

lemma *llast-ldrop*:
assumes *n* < *llength xs*
shows *llast* (*ldrop n xs*) = *llast xs*
⟨*proof*⟩

lemma *llast-llist-of* [*simp*]: *llast* (*llist-of xs*) = *last xs*
⟨*proof*⟩

lemma *llast-conv-lnth*: *llength xs* = *eSuc* (*enat n*) $\Longrightarrow$ *llast xs* = *lnth xs n*
⟨*proof*⟩

lemma *llast-lmap*:

assumes *lfinite xs* ¬ *lnull xs*
shows *llast* (*lmap f xs*) = *f* (*llast xs*)
⟨*proof*⟩

2.22 Distinct lazy lists *ldistinct*

inductive-simps *ldistinct-LCons* [*code*, *simp*]:
ldistinct (*LCons x xs*)

lemma *ldistinct-LNil-code* [*code*]:
ldistinct LNil = *True*
⟨*proof*⟩

lemma *ldistinct-llist-of* [*simp*]:
ldistinct (*llist-of xs*) ⟷ *distinct xs*
⟨*proof*⟩

lemma *ldistinct-coinduct* [*consumes 1*, *case-names ldistinct*, *case-conclusion ldistinct lhd ltl*, *coinduct pred*: *ldistinct*]:
assumes *X xs*
and *step*: ⋀*xs*. ⟦ *X xs*; ¬ *lnull xs* ⟧
⟹ *lhd xs* ∉ *lset* (*ltl xs*) ∧ (*X* (*ltl xs*) ∨ *ldistinct* (*ltl xs*))
shows *ldistinct xs*
⟨*proof*⟩

lemma *ldistinct-lhdD*:
⟦ *ldistinct xs*; ¬ *lnull xs* ⟧ ⟹ *lhd xs* ∉ *lset* (*ltl xs*)
⟨*proof*⟩

lemma *ldistinct-ltlI*:
ldistinct xs ⟹ *ldistinct* (*ltl xs*)
⟨*proof*⟩

lemma *ldistinct-lSup*:
⟦ *Complete-Partial-Order.chain* (⊑) *Y*; ∀ *xs*∈*Y*. *ldistinct xs* ⟧
⟹ *ldistinct* (*lSup Y*)
⟨*proof*⟩

lemma *admissible-ldistinct* [*cont-intro*, *simp*]:
assumes *mcont*: *mcont lub ord lSup* (⊑) (λ*x*. *f x*)
shows *ccpo.admissible lub ord* (λ*x*. *ldistinct* (*f x*))
⟨*proof*⟩

lemma *ldistinct-lappend*:
ldistinct (*lappend xs ys*) ⟷ *ldistinct xs* ∧ (*lfinite xs* ⟶ *ldistinct ys* ∧ *lset xs* ∩ *lset ys* = {})
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *ldistinct-lprefix*:
⟦ *ldistinct xs*; *ys* ⊑ *xs* ⟧ ⟹ *ldistinct ys*
⟨*proof*⟩

lemma *admissible-not-ldistinct*[*THEN admissible-subst*, *cont-intro*, *simp*]:
ccpo.admissible lSup (⊑) (λ*x*. ¬ *ldistinct x*)
⟨*proof*⟩

lemma *ldistinct-ltake*: *ldistinct xs* ⟹ *ldistinct* (*ltake n xs*)
⟨*proof*⟩

lemma *ldistinct-ldropn*:
ldistinct xs ⟹ *ldistinct* (*ldropn n xs*)
⟨*proof*⟩

lemma *ldistinct-ldrop*: *ldistinct xs* ⟹ *ldistinct* (*ldrop n xs*)
⟨*proof*⟩

lemma *ldistinct-conv-lnth*:
ldistinct xs ⟷ (∀ *i j*. *enat i* < *llength xs* ⟶ *enat j* < *llength xs* ⟶ *i* ≠ *j* ⟶ *lnth xs i* ≠ *lnth xs j*)
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *ldistinct-lmap* [*simp*]:
ldistinct (*lmap f xs*) ⟷ *ldistinct xs* ∧ *inj-on f* (*lset xs*)
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *ldistinct-lzipI1*: *ldistinct xs* ⟹ *ldistinct* (*lzip xs ys*)
⟨*proof*⟩

lemma *ldistinct-lzipI2*: *ldistinct ys* ⟹ *ldistinct* (*lzip xs ys*)
⟨*proof*⟩

2.23 Sortedness *lsorted*

context *ord* **begin**

coinductive *lsorted* :: *'a llist* ⇒ *bool*
where
LNil [*simp*]: *lsorted LNil*
| *Singleton* [*simp*]: *lsorted* (*LCons x LNil*)
| *LCons-LCons*: ⟦ *x* ≤ *y*; *lsorted* (*LCons y xs*) ⟧ ⟹ *lsorted* (*LCons x* (*LCons y xs*))

inductive-simps *lsorted-LCons-LCons* [*simp*]:
lsorted (*LCons x* (*LCons y xs*))

inductive-simps *lsorted-code* [*code*]:
lsorted LNil
lsorted (*LCons x LNil*)
lsorted (*LCons x* (*LCons y xs*))

lemma *lsorted-coinduct′* [*consumes 1*, *case-names lsorted*, *case-conclusion lsorted lhd ltl*, *coinduct pred*: *lsorted*]:
assumes *major*: *X xs*
and *step*: $\bigwedge xs.\ [\![\ X\ xs;\ \neg\ lnull\ xs;\ \neg\ lnull\ (ltl\ xs)\ ]\!] \Longrightarrow lhd\ xs \leq lhd\ (ltl\ xs) \wedge (X\ (ltl\ xs) \vee lsorted\ (ltl\ xs))$
shows *lsorted xs*
⟨*proof*⟩

lemma *lsorted-ltlI*: *lsorted xs* ⟹ *lsorted* (*ltl xs*)
⟨*proof*⟩

lemma *lsorted-lhdD*:
$[\![\ lsorted\ xs;\ \neg\ lnull\ xs;\ \neg\ lnull\ (ltl\ xs)\ ]\!] \Longrightarrow lhd\ xs \leq lhd\ (ltl\ xs)$
⟨*proof*⟩

lemma *lsorted-LCons′*:
lsorted (*LCons x xs*) ⟷ (¬ *lnull xs* ⟶ *x* ≤ *lhd xs* ∧ *lsorted xs*)
⟨*proof*⟩

lemma *lsorted-lSup*:
$[\![$ *Complete-Partial-Order.chain* (⊑) *Y*; ∀ *xs* ∈ *Y*. *lsorted xs* $]\!]$
⟹ *lsorted* (*lSup Y*)
⟨*proof*⟩

lemma *lsorted-lprefixD*:
$[\![\ xs \sqsubseteq ys;\ lsorted\ ys\ ]\!] \Longrightarrow lsorted\ xs$
⟨*proof*⟩

lemma *admissible-lsorted* [*cont-intro*, *simp*]:
assumes *mcont*: *mcont lub ord lSup* (⊑) (λ*x*. *f x*)
and *ccpo*: *class.ccpo lub ord* (*mk-less ord*)
shows *ccpo.admissible lub ord* (λ*x*. *lsorted* (*f x*))
⟨*proof*⟩

lemma *admissible-not-lsorted*[*THEN admissible-subst*, *cont-intro*, *simp*]:
ccpo.admissible lSup (⊑) (λ*xs*. ¬ *lsorted xs*)
⟨*proof*⟩

lemma *lsorted-ltake* [*simp*]: *lsorted xs* ⟹ *lsorted* (*ltake n xs*)
⟨*proof*⟩

lemma *lsorted-ldropn* [*simp*]: *lsorted xs* ⟹ *lsorted* (*ldropn n xs*)
⟨*proof*⟩

lemma *lsorted-ldrop* [*simp*]: *lsorted xs* $\Longrightarrow$ *lsorted* (*ldrop n xs*)
⟨*proof*⟩

end

declare
ord.lsorted-code [*code*]
ord.admissible-lsorted [*cont-intro*, *simp*]
ord.admissible-not-lsorted [*THEN admissible-subst*, *cont-intro*, *simp*]

context *preorder* **begin**

lemma *lsorted-LCons*:
lsorted (*LCons x xs*) $\longleftrightarrow$ *lsorted xs* $\wedge$ ($\forall$ *y*$\in$*lset xs*. *x* $\leq$ *y*) (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *lsorted-coinduct* [*consumes 1*, *case-names lsorted*, *case-conclusion lsorted lhd ltl*, *coinduct pred*: *lsorted*]:
assumes *major*: *X xs*
and *step*: $\bigwedge$*xs*. ⟦ *X xs*; ¬ *lnull xs* ⟧ $\Longrightarrow$ ($\forall$ *x* $\in$ *lset* (*ltl xs*). *lhd xs* $\leq$ *x*) $\wedge$ (*X* (*ltl xs*) $\vee$ *lsorted* (*ltl xs*))
shows *lsorted xs*
⟨*proof*⟩

lemma *lsortedD*: ⟦ *lsorted xs*; ¬ *lnull xs*; *y* $\in$ *lset* (*ltl xs*) ⟧ $\Longrightarrow$ *lhd xs* $\leq$ *y*
⟨*proof*⟩

end

lemma *lsorted-lmap′*:
assumes *ord.lsorted orda xs monotone orda ordb f*
shows *ord.lsorted ordb* (*lmap f xs*)
⟨*proof*⟩

lemma *lsorted-lmap*:
assumes *lsorted xs monotone* ($\leq$) ($\leq$) *f*
shows *lsorted* (*lmap f xs*)
⟨*proof*⟩

context *linorder* **begin**

lemma *lsorted-ldistinct-lset-unique*:
⟦ *lsorted xs*; *ldistinct xs*; *lsorted ys*; *ldistinct ys*; *lset xs* = *lset ys* ⟧
$\Longrightarrow$ *xs* = *ys*
⟨*proof*⟩

end

lemma *lsorted-llist-of*[*simp*]: *lsorted* (*llist-of xs*) $\longleftrightarrow$ *sorted xs*

⟨*proof*⟩

2.24 Lexicographic order on lazy lists: *llexord*

lemma *llexord-coinduct* [*consumes 1*, *case-names llexord*, *coinduct pred*: *llexord*]:
  assumes *X*: *X xs ys*
  and *step*: $\bigwedge$*xs ys*. ⟦ *X xs ys*; ¬ *lnull xs* ⟧
    ⟹ ¬ *lnull ys* ∧
      (¬ *lnull ys* ⟶ *r* (*lhd xs*) (*lhd ys*) ∨
                lhd xs = *lhd ys* ∧ (*X* (*ltl xs*) (*ltl ys*) ∨ *llexord r* (*ltl xs*) (*ltl ys*)))
  shows *llexord r xs ys*
⟨*proof*⟩

lemma *llexord-refl* [*simp*, *intro*!]:
  llexord r xs xs
⟨*proof*⟩

lemma *llexord-LCons-LCons* [*simp*]:
  llexord r (*LCons x xs*) (*LCons y ys*) ⟷ (*x* = *y* ∧ *llexord r xs ys* ∨ *r x y*)
⟨*proof*⟩

lemma *lnull-llexord* [*simp*]: *lnull xs* ⟹ *llexord r xs ys*
⟨*proof*⟩

lemma *llexord-LNil-right* [*simp*]:
  lnull ys ⟹ *llexord r xs ys* ⟷ *lnull xs*
⟨*proof*⟩

lemma *llexord-LCons-left*:
  llexord r (*LCons x xs*) *ys* ⟷
   (∃ *y ys*′. *ys* = *LCons y ys*′ ∧ (*x* = *y* ∧ *llexord r xs ys*′ ∨ *r x y*))
⟨*proof*⟩

lemma *lprefix-imp-llexord*:
  assumes *xs* ⊑ *ys*
  shows *llexord r xs ys*
⟨*proof*⟩

lemma *llexord-empty*:
  llexord (λ*x y*. *False*) *xs ys* = *xs* ⊑ *ys*
⟨*proof*⟩

lemma *llexord-append-right*:
  llexord r xs (*lappend xs ys*)
⟨*proof*⟩

lemma *llexord-lappend-leftI*:
  assumes *llexord r ys zs*
  shows *llexord r* (*lappend xs ys*) (*lappend xs zs*)

⟨*proof*⟩

lemma *llexord-lappend-leftD*:
assumes *lex*: *llexord r* (*lappend xs ys*) (*lappend xs zs*)
and *fin*: *lfinite xs*
and *irrefl*: !!*x*. *x* ∈ *lset xs* ⟹ ¬ *r x x*
shows *llexord r ys zs*
⟨*proof*⟩

lemma *llexord-lappend-left*:
⟦ *lfinite xs*; !!*x*. *x* ∈ *lset xs* ⟹ ¬ *r x x* ⟧
⟹ *llexord r* (*lappend xs ys*) (*lappend xs zs*) ⟷ *llexord r ys zs*
⟨*proof*⟩

lemma *antisym-llexord*:
assumes *r*: *antisymp r*
and *irrefl*: ⋀*x*. ¬ *r x x*
shows *antisymp* (*llexord r*)
⟨*proof*⟩

lemma *llexord-antisym*:
⟦ *llexord r xs ys*; *llexord r ys xs*;
!!*a b*. ⟦ *r a b*; *r b a* ⟧ ⟹ *False* ⟧
⟹ *xs* = *ys*
⟨*proof*⟩

lemma *llexord-trans*:
assumes *1*: *llexord r xs ys*
and *2*: *llexord r ys zs*
and *trans*: !!*a b c*. ⟦ *r a b*; *r b c* ⟧ ⟹ *r a c*
shows *llexord r xs zs*
⟨*proof*⟩

lemma *trans-llexord*:
transp r ⟹ *transp* (*llexord r*)
⟨*proof*⟩

lemma *llexord-linear*:
assumes *linear*: !!*x y*. *r x y* ∨ *x* = *y* ∨ *r y x*
shows *llexord r xs ys* ∨ *llexord r ys xs*
⟨*proof*⟩

lemma *llexord-code* [*code*]:
llexord r LNil ys = *True*
llexord r (*LCons x xs*) *LNil* = *False*
llexord r (*LCons x xs*) (*LCons y ys*) = (*r x y* ∨ *x* = *y* ∧ *llexord r xs ys*)
⟨*proof*⟩

lemma *llexord-conv*:

llexord r xs ys $\longleftrightarrow$
xs = ys $\vee$
($\exists$ *zs xs' y ys'*. *lfinite zs* $\wedge$ *xs = lappend zs xs'* $\wedge$ *ys = lappend zs* (*LCons y ys'*) $\wedge$
(*xs' = LNil* $\vee$ *r* (*lhd xs'*) *y*))
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *llexord-conv-ltake-index*:
llexord r xs ys $\longleftrightarrow$
(*llength xs* $\leq$ *llength ys* $\wedge$ *ltake* (*llength xs*) *ys = xs*) $\vee$
($\exists$ *n*. *enat n* $<$ *min* (*llength xs*) (*llength ys*) $\wedge$
ltake (*enat n*) *xs = ltake* (*enat n*) *ys* $\wedge$ *r* (*lnth xs n*) (*lnth ys n*))
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *llexord-llist-of*:
llexord r (*llist-of xs*) (*llist-of ys*) $\longleftrightarrow$
xs = ys $\vee$ (*xs*, *ys*) $\in$ *lexord* {(*x*, *y*). *r x y*}
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

2.25 The filter functional on lazy lists: *lfilter*

lemma *lfilter-code* [*simp*, *code*]:
shows *lfilter-LNil*: *lfilter P LNil = LNil*
and *lfilter-LCons*: *lfilter P* (*LCons x xs*) = (*if P x then LCons x* (*lfilter P xs*) *else lfilter P xs*)
⟨*proof*⟩

declare *lfilter.mono*[*cont-intro*]

lemma *mono2mono-lfilter*[*THEN llist.mono2mono*, *simp*, *cont-intro*]:
shows *monotone-lfilter*: *monotone* ($\sqsubseteq$) ($\sqsubseteq$) (*lfilter P*)
⟨*proof*⟩

lemma *mcont2mcont-lfilter*[*THEN llist.mcont2mcont*, *simp*, *cont-intro*]:
shows *mcont-lfilter*: *mcont lSup* ($\sqsubseteq$) *lSup* ($\sqsubseteq$) (*lfilter P*)
⟨*proof*⟩

lemma *lfilter-mono* [*partial-function-mono*]:
mono-llist A $\Longrightarrow$ *mono-llist* (λf. *lfilter P* (*A f*))
⟨*proof*⟩

lemma *lfilter-LCons-seek*: ~ (*p x*) ==> *lfilter p* (*LCons x l*) = *lfilter p l*
⟨*proof*⟩

lemma *lfilter-LCons-found*:
P x $\Longrightarrow$ *lfilter P* (*LCons x xs*) = *LCons x* (*lfilter P xs*)
⟨*proof*⟩

lemma *lfilter-eq-LNil*: *lfilter P xs* = *LNil* ⟷ (∀ *x* ∈ *lset xs*. ¬ *P x*)
⟨*proof*⟩

notepad begin
⟨*proof*⟩
end

lemma *diverge-lfilter-LNil* [*simp*]: ∀ *x*∈*lset xs*. ¬ *P x* ⟹ *lfilter P xs* = *LNil*
⟨*proof*⟩

lemmas *lfilter-False* = *diverge-lfilter-LNil*

lemma *lnull-lfilter* [*simp*]: *lnull* (*lfilter P xs*) ⟷ (∀ *x*∈*lset xs*. ¬ *P x*)
⟨*proof*⟩

lemmas *lfilter-empty-conv* = *lfilter-eq-LNil*

lemma *lhd-lfilter* [*simp*]: *lhd* (*lfilter P xs*) = *lhd* (*ldropWhile* (*Not* ∘ *P*) *xs*)
⟨*proof*⟩

lemma *ltl-lfilter*: *ltl* (*lfilter P xs*) = *lfilter P* (*ltl* (*ldropWhile* (*Not* ∘ *P*) *xs*))
⟨*proof*⟩

lemma *lfilter-eq-LCons*:
lfilter P xs = *LCons x xs*′ ⟹
∃ *xs*″. *xs*′ = *lfilter P xs*″ ∧ *ldropWhile* (*Not* ∘ *P*) *xs* = *LCons x xs*″
⟨*proof*⟩

lemma *lfilter-K-True* [*simp*]: *lfilter* (%-. *True*) *xs* = *xs*
⟨*proof*⟩

lemma *lfitler-K-False* [*simp*]: *lfilter* (λ-. *False*) *xs* = *LNil*
⟨*proof*⟩

lemma *lfilter-lappend-lfinite* [*simp*]:
lfinite xs ⟹ *lfilter P* (*lappend xs ys*) = *lappend* (*lfilter P xs*) (*lfilter P ys*)
⟨*proof*⟩

lemma *lfinite-lfilterI* [*simp*]: *lfinite xs* ⟹ *lfinite* (*lfilter P xs*)
⟨*proof*⟩

lemma *lset-lfilter* [*simp*]: *lset* (*lfilter P xs*) = {*x* ∈ *lset xs*. *P x*}
⟨*proof*⟩

notepad begin — show *lset-lfilter* by fixpoint induction
⟨*proof*⟩
end

lemma *lfilter-lfilter*: *lfilter P* (*lfilter Q xs*) = *lfilter* (λ*x*. *P x* ∧ *Q x*) *xs*
⟨*proof*⟩

notepad begin — show *lfilter-lfilter* by fixpoint induction
⟨*proof*⟩
end

lemma *lfilter-idem* [*simp*]: *lfilter P* (*lfilter P xs*) = *lfilter P xs*
⟨*proof*⟩

lemma *lfilter-lmap*: *lfilter P* (*lmap f xs*) = *lmap f* (*lfilter* (*P o f*) *xs*)
⟨*proof*⟩

lemma *lfilter-llist-of* [*simp*]:
lfilter P (*llist-of xs*) = *llist-of* (*filter P xs*)
⟨*proof*⟩

lemma *lfilter-cong* [*cong*]:
assumes *xsys*: *xs* = *ys*
and *set*: ⋀*x*. *x* ∈ *lset ys* ⟹ *P x* = *Q x*
shows *lfilter P xs* = *lfilter Q ys*
⟨*proof*⟩

lemma *llength-lfilter-ile*:
llength (*lfilter P xs*) ≤ *llength xs*
⟨*proof*⟩

lemma *lfinite-lfilter*:
lfinite (*lfilter P xs*) ⟷
lfinite xs ∨ *finite* {*n*. *enat n* < *llength xs* ∧ *P* (*lnth xs n*)}
⟨*proof*⟩

lemma *lfilter-eq-LConsD*:
assumes *lfilter P ys* = *LCons x xs*
shows ∃ *us vs*. *ys* = *lappend us* (*LCons x vs*) ∧ *lfinite us* ∧
(∀ *u*∈*lset us*. ¬ *P u*) ∧ *P x* ∧ *xs* = *lfilter P vs*
⟨*proof*⟩

lemma *lfilter-eq-lappend-lfiniteD*:
assumes *lfilter P xs* = *lappend ys zs* **and** *lfinite ys*
shows ∃ *us vs*. *xs* = *lappend us vs* ∧ *lfinite us* ∧
ys = *lfilter P us* ∧ *zs* = *lfilter P vs*
⟨*proof*⟩

lemma *ldistinct-lfilterI*: *ldistinct xs* ⟹ *ldistinct* (*lfilter P xs*)
⟨*proof*⟩

notepad begin
⟨*proof*⟩

end

lemma *ldistinct-lfilterD*:
⟦ *ldistinct* (*lfilter P xs*); *enat n* < *llength xs*; *enat m* < *llength xs*; *P a*; *lnth xs n* = *a*; *lnth xs m* = *a* ⟧ ⟹ *m* = *n*
⟨*proof*⟩

lemmas *lfilter-fixp-parallel-induct* =
parallel-fixp-induct-1-1[*OF llist-partial-function-definitions llist-partial-function-definitions lfilter.mono lfilter.mono lfilter-def lfilter-def*, *case-names adm LNil step*]

lemma *llist-all2-lfilterI*:
assumes ∗: *llist-all2 P xs ys*
and *Q*: ⋀*x y*. *P x y* ⟹ *Q1 x* ⟷ *Q2 y*
shows *llist-all2 P* (*lfilter Q1 xs*) (*lfilter Q2 ys*)
⟨*proof*⟩

lemma *distinct-filterD*:
⟦ *distinct* (*filter P xs*); *n* < *length xs*; *m* < *length xs*; *P x*; *xs* ! *n* = *x*; *xs* ! *m* = *x* ⟧ ⟹ *m* = *n*
⟨*proof*⟩

lemma *lprefix-lfilterI*:
xs ⊑ *ys* ⟹ *lfilter P xs* ⊑ *lfilter P ys*
⟨*proof*⟩

context *preorder* **begin**

lemma *lsorted-lfilterI*:
lsorted xs ⟹ *lsorted* (*lfilter P xs*)
⟨*proof*⟩

lemma *lsorted-lfilter-same*:
lsorted (*lfilter* (λ*x*. *x* = *c*) *xs*)
⟨*proof*⟩

end

lemma *lfilter-id-conv*: *lfilter P xs* = *xs* ⟷ (∀ *x*∈*lset xs*. *P x*) (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *lfilter-repeat* [*simp*]: *lfilter P* (*repeat x*) = (*if P x then repeat x else LNil*)
⟨*proof*⟩

2.26 Concatenating all lazy lists in a lazy list: *lconcat*

lemma *lconcat-simps* [*simp*, *code*]:
shows *lconcat-LNil*: *lconcat LNil* = *LNil*
and *lconcat-LCons*: *lconcat* (*LCons xs xss*) = *lappend xs* (*lconcat xss*)

⟨*proof*⟩

declare *lconcat.mono*[*cont-intro*]

lemma *mono2mono-lconcat*[*THEN llist.mono2mono*, *cont-intro*, *simp*]:
  shows *monotone-lconcat*: *monotone* (⊑) (⊑) *lconcat*
⟨*proof*⟩

lemma *mcont2mcont-lconcat*[*THEN llist.mcont2mcont*, *cont-intro*, *simp*]:
  shows *mcont-lconcat*: *mcont lSup* (⊑) *lSup* (⊑) *lconcat*
⟨*proof*⟩

lemma *lconcat-eq-LNil*: *lconcat xss* = *LNil* ⟷ *lset xss* ⊆ {*LNil*} (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *lnull-lconcat* [*simp*]: *lnull* (*lconcat xss*) ⟷ *lset xss* ⊆ {*xs*. *lnull xs*}
⟨*proof*⟩

lemma *lconcat-llist-of*:
  lconcat (*llist-of* (*map llist-of xs*)) = *llist-of* (*concat xs*)
⟨*proof*⟩

lemma *lhd-lconcat* [*simp*]:
  ⟦ ¬ *lnull xss*; ¬ *lnull* (*lhd xss*) ⟧ ⟹ *lhd* (*lconcat xss*) = *lhd* (*lhd xss*)
⟨*proof*⟩

lemma *ltl-lconcat* [*simp*]:
  ⟦ ¬ *lnull xss*; ¬ *lnull* (*lhd xss*) ⟧ ⟹ *ltl* (*lconcat xss*) = *lappend* (*ltl* (*lhd xss*)) (*lconcat* (*ltl xss*))
⟨*proof*⟩

lemma *lmap-lconcat*:
  lmap f (*lconcat xss*) = *lconcat* (*lmap* (*lmap f*) *xss*)
⟨*proof*⟩

lemma *lconcat-lappend* [*simp*]:
  assumes *lfinite xss*
  shows *lconcat* (*lappend xss yss*) = *lappend* (*lconcat xss*) (*lconcat yss*)
⟨*proof*⟩

lemma *lconcat-eq-LCons-conv*:
  lconcat xss = *LCons x xs* ⟷
  (∃ *xs′ xss′ xss″*. *xss* = *lappend* (*llist-of xss′*) (*LCons* (*LCons x xs′*) *xss″*) ∧
                xs = *lappend xs′* (*lconcat xss″*) ∧ *set xss′* ⊆ {*xs*. *lnull xs*})
  (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *llength-lconcat-lfinite-conv-sum*:

assumes *lfinite xss*
shows *llength* (*lconcat xss*) = (∑ *i* | *enat i* < *llength xss*. *llength* (*lnth xss i*))
⟨*proof*⟩

lemma *lconcat-lfilter-neq-LNil*:
lconcat (*lfilter* (λ*xs*. ¬ *lnull xs*) *xss*) = *lconcat xss*
⟨*proof*⟩

lemmas *lconcat-fixp-parallel-induct* =
parallel-fixp-induct-1-1[*OF llist-partial-function-definitions llist-partial-function-definitions lconcat.mono lconcat.mono lconcat-def lconcat-def*, *case-names adm LNil step*]

lemma *llist-all2-lconcatI*:
llist-all2 (*llist-all2 A*) *xss yss*
⟹ *llist-all2 A* (*lconcat xss*) (*lconcat yss*)
⟨*proof*⟩

lemma *llength-lconcat-eqI*:
fixes *xss* :: *'a llist llist* **and** *yss* :: *'b llist llist*
assumes *llist-all2* (λ*xs ys*. *llength xs* = *llength ys*) *xss yss*
shows *llength* (*lconcat xss*) = *llength* (*lconcat yss*)
⟨*proof*⟩

lemma *lset-lconcat-lfinite*:
∀ *xs* ∈ *lset xss*. *lfinite xs* ⟹ *lset* (*lconcat xss*) = (⋃ *xs*∈*lset xss*. *lset xs*)
⟨*proof*⟩

lemma *lconcat-ltake*:
lconcat (*ltake* (*enat n*) *xss*) = *ltake* (∑ *i*<*n*. *llength* (*lnth xss i*)) (*lconcat xss*)
⟨*proof*⟩

lemma *lnth-lconcat-conv*:
assumes *enat n* < *llength* (*lconcat xss*)
shows ∃ *m n'*. *lnth* (*lconcat xss*) *n* = *lnth* (*lnth xss m*) *n'* ∧ *enat n'* < *llength* (*lnth xss m*) ∧
enat m < *llength xss* ∧ *enat n* = (∑ *i*<*m* . *llength* (*lnth xss i*)) + *enat n'*
⟨*proof*⟩

lemma *lprefix-lconcatI*:
xss ⊑ *yss* ⟹ *lconcat xss* ⊑ *lconcat yss*
⟨*proof*⟩

lemma *lnth-lconcat-ltake*:
assumes *enat w* < *llength* (*lconcat* (*ltake* (*enat n*) *xss*))
shows *lnth* (*lconcat* (*ltake* (*enat n*) *xss*)) *w* = *lnth* (*lconcat xss*) *w*
⟨*proof*⟩

lemma *lfinite-lconcat* [*simp*]:

lfinite (*lconcat xss*) ⟷ *lfinite* (*lfilter* (λ*xs*. ¬ *lnull xs*) *xss*) ∧ (∀ *xs* ∈ *lset xss*. *lfinite xs*)
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *list-of-lconcat*:
assumes *lfinite xss*
and ∀ *xs* ∈ *lset xss*. *lfinite xs*
shows *list-of* (*lconcat xss*) = *concat* (*list-of* (*lmap list-of xss*))
⟨*proof*⟩

lemma *lfilter-lconcat-lfinite*:
∀ *xs*∈*lset xss*. *lfinite xs*
⟹ *lfilter P* (*lconcat xss*) = *lconcat* (*lmap* (*lfilter P*) *xss*)
⟨*proof*⟩

lemma *lconcat-repeat-LNil* [*simp*]: *lconcat* (*repeat LNil*) = *LNil*
⟨*proof*⟩

lemma *lconcat-lmap-singleton* [*simp*]: *lconcat* (*lmap* (λ*x*. *LCons* (*f x*) *LNil*) *xs*) = *lmap f xs*
⟨*proof*⟩

lemma *lset-lconcat-subset*: *lset* (*lconcat xss*) ⊆ (⋃ *xs*∈*lset xss*. *lset xs*)
⟨*proof*⟩

lemma *ldistinct-lconcat*:
⟦ *ldistinct xss*; ⋀*ys*. *ys* ∈ *lset xss* ⟹ *ldistinct ys*;
⋀*ys zs*. ⟦ *ys* ∈ *lset xss*; *zs* ∈ *lset xss*; *ys* ≠ *zs* ⟧ ⟹ *lset ys* ∩ *lset zs* = {} ⟧
⟹ *ldistinct* (*lconcat xss*)
⟨*proof*⟩

2.27 Sublist view of a lazy list: *lnths*

lemma *lnths-empty* [*simp*]: *lnths xs* {} = *LNil*
⟨*proof*⟩

lemma *lnths-LNil* [*simp*]: *lnths LNil A* = *LNil*
⟨*proof*⟩

lemma *lnths-LCons*:
lnths (*LCons x xs*) *A* =
(*if* 0 ∈ *A then LCons x* (*lnths xs* {*n*. *Suc n* ∈ *A*}) *else lnths xs* {*n*. *Suc n* ∈ *A*})
⟨*proof*⟩

lemma *lset-lnths*:
lset (*lnths xs I*) = {*lnth xs i*|*i*. *enat i*<*llength xs* ∧ *i* ∈ *I*}
⟨*proof*⟩

lemma *lset-lnths-subset*: *lset* (*lnths xs I*) ⊆ *lset xs*
⟨*proof*⟩

lemma *lnths-singleton* [*simp*]:
lnths (*LCons x LNil*) *A* = (*if 0* : *A then LCons x LNil else LNil*)
⟨*proof*⟩

lemma *lnths-upt-eq-ltake* [*simp*]:
lnths xs {..<*n*} = *ltake* (*enat n*) *xs*
⟨*proof*⟩

lemma *lnths-llist-of* [*simp*]:
lnths (*llist-of xs*) *A* = *llist-of* (*nths xs A*)
⟨*proof*⟩

lemma *llength-lnths-ile*: *llength* (*lnths xs A*) ≤ *llength xs*
⟨*proof*⟩

lemma *lnths-lmap* [*simp*]:
lnths (*lmap f xs*) *A* = *lmap f* (*lnths xs A*)
⟨*proof*⟩

lemma *lfilter-conv-lnths*:
lfilter P xs = *lnths xs* {*n*. *enat n* < *llength xs* ∧ *P* (*lnth xs n*)}
⟨*proof*⟩

lemma *ltake-iterates-Suc*:
ltake (*enat n*) (*iterates Suc m*) = *llist-of* [*m*..<*n* + *m*]
⟨*proof*⟩

lemma *lnths-lappend-lfinite*:
assumes *len*: *llength xs* = *enat k*
shows *lnths* (*lappend xs ys*) *A* =
lappend (*lnths xs A*) (*lnths ys* {*n*. *n* + *k* ∈ *A*})
⟨*proof*⟩

lemma *lnths-split*:
lnths xs A =
lappend (*lnths* (*ltake* (*enat n*) *xs*) *A*) (*lnths* (*ldropn n xs*) {*m*. *n* + *m* ∈ *A*})
⟨*proof*⟩

lemma *lnths-cong*:
assumes *xs* = *ys* **and** *A*: ⋀*n*. *enat n* < *llength ys* ⟹ *n* ∈ *A* ⟷ *n* ∈ *B*
shows *lnths xs A* = *lnths ys B*
⟨*proof*⟩

lemma *lnths-insert*:
assumes *n*: *enat n* < *llength xs*
shows *lnths xs* (*insert n A*) =

lappend (*lnths* (*ltake* (*enat n*) *xs*) *A*) (*LCons* (*lnth xs n*)
(*lnths* (*ldropn* (*Suc n*) *xs*) {*m*. *Suc* (*n* + *m*) ∈ *A*}))
⟨*proof*⟩

lemma *lfinite-lnths* [*simp*]:
lfinite (*lnths xs A*) ⟷ *lfinite xs* ∨ *finite A*
⟨*proof*⟩

2.28 *lsum-list*

context *monoid-add* **begin**

lemma *lsum-list-0* [*simp*]: *lsum-list* (*lmap* (λ-. *0*) *xs*) = *0*
⟨*proof*⟩

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

lemma *lsum-list-lappend*: ⟦ *lfinite xs*; *lfinite ys* ⟧ ⟹ *lsum-list* (*lappend xs ys*) = *lsum-list xs* + *lsum-list ys*
⟨*proof*⟩

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

lemma *lsum-list-LCons* [*simp*]: *lfinite xs* ⟹ *lsum-list* (*LCons x xs*) = *x* + *lsum-list xs*
⟨*proof*⟩

lemma *lsum-list-inf* [*simp*]: ¬ *lfinite xs* ⟹ *lsum-list xs* = *0*
⟨*proof*⟩

end

lemma *lsum-list-mono*:
fixes *f* :: *'a* ⇒ *'b* :: {*monoid-add*, *ordered-ab-semigroup-add*}
assumes ⋀*x*. *x* ∈ *lset xs* ⟹ *f x* ≤ *g x*
shows *lsum-list* (*lmap f xs*) ≤ *lsum-list* (*lmap g xs*)
⟨*proof*⟩

2.29 Alternative view on *'a llist* as datatype with constructors *llist-of* and *inf-llist*

lemma *lnull-inf-llist* [*simp*]: ¬ *lnull* (*inf-llist f*)
⟨*proof*⟩

lemma *inf-llist-neq-LNil*: *inf-llist f* ≠ *LNil*
⟨*proof*⟩

lemmas *LNil-neq-inf-llist* = *inf-llist-neq-LNil*[*symmetric*]

lemma *lhd-inf-llist* [*simp*]: *lhd* (*inf-llist f*) = *f 0*
⟨*proof*⟩

lemma *ltl-inf-llist* [*simp*]: *ltl* (*inf-llist f*) = *inf-llist* (λ*n*. *f* (*Suc n*))

⟨*proof*⟩

lemma *inf-llist-rec* [*code*, *nitpick-simp*]:
inf-llist f = *LCons* (*f 0*) (*inf-llist* (λ*n*. *f* (*Suc n*)))
⟨*proof*⟩

lemma *lfinite-inf-llist* [*iff*]: ¬ *lfinite* (*inf-llist f*)
⟨*proof*⟩

lemma *iterates-conv-inf-llist*:
iterates f a = *inf-llist* (λ*n*. (*f* ^^ *n*) *a*)
⟨*proof*⟩

lemma *inf-llist-neq-llist-of* [*simp*]:
llist-of xs ≠ *inf-llist f*
inf-llist f ≠ *llist-of xs*
⟨*proof*⟩

lemma *lnth-inf-llist* [*simp*]: *lnth* (*inf-llist f*) *n* = *f n*
⟨*proof*⟩

lemma *inf-llist-lprefix* [*simp*]: *inf-llist f* ⊑ *xs* ⟷ *xs* = *inf-llist f*
⟨*proof*⟩

lemma *llength-inf-llist* [*simp*]: *llength* (*inf-llist f*) = ∞
⟨*proof*⟩

lemma *lset-inf-llist* [*simp*]: *lset* (*inf-llist f*) = *range f*
⟨*proof*⟩

lemma *inf-llist-inj* [*simp*]:
inf-llist f = *inf-llist g* ⟷ *f* = *g*
⟨*proof*⟩

lemma *inf-llist-lnth* [*simp*]: ¬ *lfinite xs* ⟹ *inf-llist* (*lnth xs*) = *xs*
⟨*proof*⟩

lemma *llist-exhaust*:
obtains (*llist-of*) *ys* **where** *xs* = *llist-of ys*
| (*inf-llist*) *f* **where** *xs* = *inf-llist f*
⟨*proof*⟩

lemma *lappend-inf-llist* [*simp*]: *lappend* (*inf-llist f*) *xs* = *inf-llist f*
⟨*proof*⟩

lemma *lmap-inf-llist* [*simp*]:
lmap f (*inf-llist g*) = *inf-llist* (*f o g*)
⟨*proof*⟩

lemma *ltake-enat-inf-llist* [*simp*]:
ltake (*enat n*) (*inf-llist f*) = *llist-of* (*map f* [*0*..<*n*])
⟨*proof*⟩

lemma *ldropn-inf-llist* [*simp*]:
ldropn n (*inf-llist f*) = *inf-llist* (λ*m*. *f* (*m* + *n*))
⟨*proof*⟩

lemma *ldrop-enat-inf-llist*:
ldrop (*enat n*) (*inf-llist f*) = *inf-llist* (λ*m*. *f* (*m* + *n*))
⟨*proof*⟩

lemma *lzip-inf-llist-inf-llist* [*simp*]:
lzip (*inf-llist f*) (*inf-llist g*) = *inf-llist* (λ*n*. (*f n*, *g n*))
⟨*proof*⟩

lemma *lzip-llist-of-inf-llist* [*simp*]:
lzip (*llist-of xs*) (*inf-llist f*) = *llist-of* (*zip xs* (*map f* [*0*..<*length xs*]))
⟨*proof*⟩

lemma *lzip-inf-llist-llist-of* [*simp*]:
lzip (*inf-llist f*) (*llist-of xs*) = *llist-of* (*zip* (*map f* [*0*..<*length xs*]) *xs*)
⟨*proof*⟩

lemma *llist-all2-inf-llist* [*simp*]:
llist-all2 P (*inf-llist f*) (*inf-llist g*) ⟷ (∀ *n*. *P* (*f n*) (*g n*))
⟨*proof*⟩

lemma *llist-all2-llist-of-inf-llist* [*simp*]:
¬ *llist-all2 P* (*llist-of xs*) (*inf-llist f*)
⟨*proof*⟩

lemma *llist-all2-inf-llist-llist-of* [*simp*]:
¬ *llist-all2 P* (*inf-llist f*) (*llist-of xs*)
⟨*proof*⟩

lemma (**in** *monoid-add*) *lsum-list-infllist*: *lsum-list* (*inf-llist f*) = *0*
⟨*proof*⟩

2.30 Setup for lifting and transfer

2.30.1 Relator and predicator properties

abbreviation *llist-all* == *pred-llist*

2.30.2 Transfer rules for the Transfer package

context includes *lifting-syntax*
begin

lemma *set1-pre-llist-transfer* [*transfer-rule*]:
(*rel-pre-llist A B* ===> *rel-set A*) *set1-pre-llist set1-pre-llist*
⟨*proof*⟩

lemma *set2-pre-llist-transfer* [*transfer-rule*]:
(*rel-pre-llist A B* ===> *rel-set B*) *set2-pre-llist set2-pre-llist*
⟨*proof*⟩

lemma *LNil-transfer* [*transfer-rule*]: *llist-all2 P LNil LNil*
⟨*proof*⟩

lemma *LCons-transfer* [*transfer-rule*]:
(*A* ===> *llist-all2 A* ===> *llist-all2 A*) *LCons LCons*
⟨*proof*⟩

lemma *case-llist-transfer* [*transfer-rule*]:
(*B* ===> (*A* ===> *llist-all2 A* ===> *B*) ===> *llist-all2 A* ===> *B*)
case-llist case-llist
⟨*proof*⟩

lemma *unfold-llist-transfer* [*transfer-rule*]:
((*A* ===> (=)) ===> (*A* ===> *B*) ===> (*A* ===> *A*) ===> *A* ===> *llist-all2 B*) *unfold-llist unfold-llist*
⟨*proof*⟩

lemma *llist-corec-transfer* [*transfer-rule*]:
((*A* ===> (=)) ===> (*A* ===> *B*) ===> (*A* ===> (=)) ===> (*A* ===> *llist-all2 B*) ===> (*A* ===> *A*) ===> *A* ===> *llist-all2 B*) *corec-llist corec-llist*
⟨*proof*⟩

lemma *ltl-transfer* [*transfer-rule*]:
(*llist-all2 A* ===> *llist-all2 A*) *ltl ltl*
⟨*proof*⟩

lemma *lset-transfer* [*transfer-rule*]:
(*llist-all2 A* ===> *rel-set A*) *lset lset*
⟨*proof*⟩

lemma *lmap-transfer* [*transfer-rule*]:
(*A* ===> *B*) ===> *llist-all2 A* ===> *llist-all2 B*) *lmap lmap*
⟨*proof*⟩

lemma *lappend-transfer* [*transfer-rule*]:
(*llist-all2 A* ===> *llist-all2 A* ===> *llist-all2 A*) *lappend lappend*
⟨*proof*⟩

lemma *iterates-transfer* [*transfer-rule*]:
((*A* ===> *A*) ===> *A* ===> *llist-all2 A*) *iterates iterates*
⟨*proof*⟩

lemma *lfinite-transfer* [*transfer-rule*]:
(*llist-all2 A* ===> (=)) *lfinite lfinite*
⟨*proof*⟩

lemma *llist-of-transfer* [*transfer-rule*]:
(*list-all2 A* ===> *llist-all2 A*) *llist-of llist-of*
⟨*proof*⟩

lemma *llength-transfer* [*transfer-rule*]:
(*llist-all2 A* ===> (=)) *llength llength*
⟨*proof*⟩

lemma *ltake-transfer* [*transfer-rule*]:
((=) ===> *llist-all2 A* ===> *llist-all2 A*) *ltake ltake*
⟨*proof*⟩

lemma *ldropn-transfer* [*transfer-rule*]:
((=) ===> *llist-all2 A* ===> *llist-all2 A*) *ldropn ldropn*
⟨*proof*⟩

lemma *ldrop-transfer* [*transfer-rule*]:
((=) ===> *llist-all2 A* ===> *llist-all2 A*) *ldrop ldrop*
⟨*proof*⟩

lemma *ltakeWhile-transfer* [*transfer-rule*]:
((*A* ===> (=)) ===> *llist-all2 A* ===> *llist-all2 A*) *ltakeWhile ltakeWhile*
⟨*proof*⟩

lemma *ldropWhile-transfer* [*transfer-rule*]:
((*A* ===> (=)) ===> *llist-all2 A* ===> *llist-all2 A*) *ldropWhile ldropWhile*
⟨*proof*⟩

lemma *lzip-ltransfer* [*transfer-rule*]:
(*llist-all2 A* ===> *llist-all2 B* ===> *llist-all2* (*rel-prod A B*)) *lzip lzip*
⟨*proof*⟩

lemma *inf-llist-transfer* [*transfer-rule*]:

(((=) ===> A) ===> llist-all2 A) inf-llist inf-llist
⟨proof⟩

lemma *lfilter-transfer* [*transfer-rule*]:
((A ===> (=)) ===> llist-all2 A ===> llist-all2 A) lfilter lfilter
⟨proof⟩

lemma *lconcat-transfer* [*transfer-rule*]:
(llist-all2 (llist-all2 A) ===> llist-all2 A) lconcat lconcat
⟨proof⟩

lemma *lnths-transfer* [*transfer-rule*]:
(llist-all2 A ===> (=) ===> llist-all2 A) lnths lnths
⟨proof⟩

lemma *llist-all-transfer* [*transfer-rule*]:
((A ===> (=)) ===> llist-all2 A ===> (=)) llist-all llist-all
⟨proof⟩

lemma *llist-all2-rsp*:
assumes *r*: $\forall x\ y.\ R\ x\ y \longrightarrow (\forall a\ b.\ R\ a\ b \longrightarrow S\ x\ a = T\ y\ b)$
and *l1*: *llist-all2 R x y*
and *l2*: *llist-all2 R a b*
shows *llist-all2 S x a* = *llist-all2 T y b*
⟨proof⟩

lemma *llist-all2-transfer* [*transfer-rule*]:
((R ===> R ===> (=)) ===> llist-all2 R ===> llist-all2 R ===> (=))
llist-all2 llist-all2
⟨proof⟩

end

no-notation *lprefix* (**infix** $\sqsubseteq$ *65*)

end

3 Instantiation of the order type classes for lazy lists

theory *Coinductive-List-Prefix* **imports**
Coinductive-List
HOL−Library.Prefix-Order
begin

3.1 Instantiation of the order type class

instantiation *llist* :: (*type*) *order* **begin**

definition [*code-unfold*]: $xs \leq ys = lprefix\ xs\ ys$

definition [*code-unfold*]: $xs < ys = lstrict\text{-}prefix\ xs\ ys$

instance
⟨*proof*⟩

end

lemma *le-llist-conv-lprefix* [*iff*]: $(\leq) = lprefix$
⟨*proof*⟩

lemma *less-llist-conv-lstrict-prefix* [*iff*]: $(<) = lstrict\text{-}prefix$
⟨*proof*⟩

instantiation *llist* :: (*type*) *order-bot* **begin**

definition $bot = LNil$

instance
⟨*proof*⟩

end

lemma *llist-of-lprefix-llist-of* [*simp*]:
$lprefix\ (llist\text{-}of\ xs)\ (llist\text{-}of\ ys) \longleftrightarrow xs \leq ys$
⟨*proof*⟩

3.2 Prefix ordering as a lower semilattice

instantiation *llist* :: (*type*) *semilattice-inf* **begin**

definition [*code del*]:
$inf\ xs\ ys =$
$unfold\text{-}llist\ (\lambda(xs, ys).\ xs \neq LNil \longrightarrow ys \neq LNil \longrightarrow lhd\ xs \neq lhd\ ys)$
$(lhd \circ snd)\ (map\text{-}prod\ ltl\ ltl)\ (xs, ys)$

lemma *llist-inf-simps* [*simp*, *code*, *nitpick-simp*]:
$inf\ LNil\ xs = LNil$
$inf\ xs\ LNil = LNil$
$inf\ (LCons\ x\ xs)\ (LCons\ y\ ys) = (if\ x = y\ then\ LCons\ x\ (inf\ xs\ ys)\ else\ LNil)$
⟨*proof*⟩

lemma *llist-inf-eq-LNil* [*simp*]:
$lnull\ (inf\ xs\ ys) \longleftrightarrow (xs \neq LNil \longrightarrow ys \neq LNil \longrightarrow lhd\ xs \neq lhd\ ys)$
⟨*proof*⟩

lemma [*simp*]: **assumes** $xs \neq LNil\ ys \neq LNil\ lhd\ xs = lhd\ ys$

shows *lhd-llist-inf*: *lhd* (*inf xs ys*) = *lhd ys*
and *ltl-llist-inf*: *ltl* (*inf xs ys*) = *inf* (*ltl xs*) (*ltl ys*)
⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *llength-inf* [*simp*]: *llength* (*inf xs ys*) = *llcp xs ys*
⟨*proof*⟩

instantiation *llist* :: (*type*) *ccpo*
begin

definition *Sup A* = *lSup A*

instance
⟨*proof*⟩

end

end

4 Infinite lists as a codatatype

theory *Coinductive-Stream*
imports
HOL−Library.Stream
HOL−Library.Linear-Temporal-Logic-on-Streams
Coinductive-List
begin

lemma *eq-onpI*: *P x* ⟹ *eq-onp P x x*
⟨*proof*⟩

primcorec *unfold-stream* :: ('*a* ⇒ '*b*) ⇒ ('*a* ⇒ '*a*) ⇒ '*a* ⇒ '*b stream* **where**
unfold-stream g1 g2 a = *g1 a* ## *unfold-stream g1 g2* (*g2 a*)

The following setup should be done by the BNF package.

congruence rule

declare *stream.map-cong* [*cong*]

lemmas about generated constants

lemma *eq-SConsD*: *xs* = *SCons y ys* ⟹ *shd xs* = *y* ∧ *stl xs* = *ys*
⟨*proof*⟩

declare *stream.map-ident*[*simp*]

lemma *smap-eq-SCons-conv*:
smap f xs = y ## ys ⟷
(∃ x xs′. xs = x ## xs′ ∧ y = f x ∧ ys = smap f xs′)
⟨*proof*⟩

lemma *smap-unfold-stream*:
smap f (unfold-stream SHD STL b) = unfold-stream (f ∘ SHD) STL b
⟨*proof*⟩

lemma *smap-corec-stream*:
smap f (corec-stream SHD endORmore STL-end STL-more b) =
corec-stream (f ∘ SHD) endORmore (smap f ∘ STL-end) STL-more b
⟨*proof*⟩

lemma *unfold-stream-ltl-unroll*:
unfold-stream SHD STL (STL b) = unfold-stream (SHD ∘ STL) STL b
⟨*proof*⟩

lemma *unfold-stream-eq-SCons* [*simp*]:
unfold-stream SHD STL b = x ## xs ⟷
x = SHD b ∧ xs = unfold-stream SHD STL (STL b)
⟨*proof*⟩

lemma *unfold-stream-id* [*simp*]: *unfold-stream shd stl xs = xs*
⟨*proof*⟩

lemma *sset-neq-empty* [*simp*]: *sset xs ≠ {}*
⟨*proof*⟩

declare *stream.set-sel(1)*[*simp*]

lemma *sset-stl*: *sset (stl xs) ⊆ sset xs*
⟨*proof*⟩

induction rules

lemmas *stream-set-induct = sset-induct*

4.1 Lemmas about operations from *HOL−Library.Stream*

lemma *szip-iterates*:
szip (siterate f a) (siterate g b) = siterate (map-prod f g) (a, b)
⟨*proof*⟩

lemma *szip-smap1*: *szip (smap f xs) ys = smap (apfst f) (szip xs ys)*
⟨*proof*⟩

lemma *szip-smap2*: *szip xs (smap g ys) = smap (apsnd g) (szip xs ys)*
⟨*proof*⟩

lemma *szip-smap* [*simp*]: *szip* (*smap f xs*) (*smap g ys*) = *smap* (*map-prod f g*) (*szip xs ys*)
⟨*proof*⟩

lemma *smap-fst-szip* [*simp*]: *smap fst* (*szip xs ys*) = *xs*
⟨*proof*⟩

lemma *smap-snd-szip* [*simp*]: *smap snd* (*szip xs ys*) = *ys*
⟨*proof*⟩

lemma *snth-shift*: *snth* (*shift xs ys*) *n* = (*if n* < *length xs then xs* ! *n else snth ys* (*n* − *length xs*))
⟨*proof*⟩

declare *szip-unfold* [*simp*, *nitpick-simp*]

lemma *szip-shift*:
length xs = *length us*
⟹ *szip* (*xs* @− *ys*) (*us* @− *zs*) = *zip xs us* @− *szip ys zs*
⟨*proof*⟩

4.2 Link *'a stream* to *'a llist*

definition *llist-of-stream* :: *'a stream* ⇒ *'a llist*
where *llist-of-stream* = *unfold-llist* (λ-. *False*) *shd stl*

definition *stream-of-llist* :: *'a llist* ⇒ *'a stream*
where *stream-of-llist* = *unfold-stream lhd ltl*

lemma *lnull-llist-of-stream* [*simp*]: ¬ *lnull* (*llist-of-stream xs*)
⟨*proof*⟩

lemma *ltl-llist-of-stream* [*simp*]: *ltl* (*llist-of-stream xs*) = *llist-of-stream* (*stl xs*)
⟨*proof*⟩

lemma *stl-stream-of-llist* [*simp*]: *stl* (*stream-of-llist xs*) = *stream-of-llist* (*ltl xs*)
⟨*proof*⟩

lemma *shd-stream-of-llist* [*simp*]: *shd* (*stream-of-llist xs*) = *lhd xs*
⟨*proof*⟩

lemma *lhd-llist-of-stream* [*simp*]: *lhd* (*llist-of-stream xs*) = *shd xs*
⟨*proof*⟩

lemma *stream-of-llist-llist-of-stream* [*simp*]:
stream-of-llist (*llist-of-stream xs*) = *xs*
⟨*proof*⟩

lemma *llist-of-stream-stream-of-llist* [*simp*]:
  ¬ *lfinite xs* ⟹ *llist-of-stream* (*stream-of-llist xs*) = *xs*
⟨*proof*⟩

lemma *lfinite-llist-of-stream* [*simp*]: ¬ *lfinite* (*llist-of-stream xs*)
⟨*proof*⟩

lemma *stream-from-llist*: *type-definition llist-of-stream stream-of-llist* {*xs*. ¬ *lfinite xs*}
⟨*proof*⟩

interpretation *stream*: *type-definition llist-of-stream stream-of-llist* {*xs*. ¬ *lfinite xs*}
⟨*proof*⟩

declare *stream.exhaust*[*cases type*: *stream*]

locale *stream-from-llist-setup*
begin
setup-lifting *stream-from-llist*
end

context
begin

interpretation *stream-from-llist-setup* ⟨*proof*⟩

lemma *cr-streamI*: ¬ *lfinite xs* ⟹ *cr-stream xs* (*stream-of-llist xs*)
⟨*proof*⟩

lemma *llist-of-stream-unfold-stream* [*simp*]:
  llist-of-stream (*unfold-stream SHD STL x*) = *unfold-llist* (λ-. *False*) *SHD STL x*
⟨*proof*⟩

lemma *llist-of-stream-corec-stream* [*simp*]:
  llist-of-stream (*corec-stream SHD endORmore STL-more STL-end x*) =
   corec-llist (λ-. *False*) *SHD endORmore* (*llist-of-stream* ∘ *STL-more*) *STL-end x*
⟨*proof*⟩

lemma *LCons-llist-of-stream* [*simp*]: *LCons x* (*llist-of-stream xs*) = *llist-of-stream* (*x* ## *xs*)
⟨*proof*⟩

lemma *lmap-llist-of-stream* [*simp*]:
  lmap f (*llist-of-stream xs*) = *llist-of-stream* (*smap f xs*)
⟨*proof*⟩

lemma *lset-llist-of-stream* [*simp*]: *lset* (*llist-of-stream xs*) = *sset xs* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *lnth-list-of-stream* [*simp*]:
lnth (*llist-of-stream xs*) = *snth xs*
⟨*proof*⟩

lemma *llist-of-stream-siterates* [*simp*]: *llist-of-stream* (*siterate f x*) = *iterates f x*
⟨*proof*⟩

lemma *lappend-llist-of-stream-conv-shift* [*simp*]:
lappend (*llist-of xs*) (*llist-of-stream ys*) = *llist-of-stream* (*xs* @− *ys*)
⟨*proof*⟩

lemma *lzip-llist-of-stream* [*simp*]:
lzip (*llist-of-stream xs*) (*llist-of-stream ys*) = *llist-of-stream* (*szip xs ys*)
⟨*proof*⟩

context includes *lifting-syntax*
begin

lemma *lmap-infinite-transfer* [*transfer-rule*]:
((=) ===> *eq-onp* (λ*xs*. ¬ *lfinite xs*) ===> *eq-onp* (λ*xs*. ¬ *lfinite xs*)) *lmap*
lmap
⟨*proof*⟩

lemma *lset-infinite-transfer* [*transfer-rule*]:
(*eq-onp* (λ*xs*. ¬ *lfinite xs*) ===> (=)) *lset lset*
⟨*proof*⟩

lemma *unfold-stream-transfer* [*transfer-rule*]:
((=) ===> (=) ===> (=) ===> *pcr-stream* (=)) (*unfold-llist* (λ-. *False*))
unfold-stream
⟨*proof*⟩

lemma *corec-stream-transfer* [*transfer-rule*]:
((=) ===> (=) ===> ((=) ===> *pcr-stream* (=)) ===> (=) ===> (=)
===> *pcr-stream* (=))
(*corec-llist* (λ-. *False*)) *corec-stream*
⟨*proof*⟩

lemma *shd-transfer* [*transfer-rule*]: (*pcr-stream A* ===> *A*) *lhd shd*
⟨*proof*⟩

lemma *stl-transfer* [*transfer-rule*]: (*pcr-stream A* ===> *pcr-stream A*) *ltl stl*
⟨*proof*⟩

lemma *llist-of-stream-transfer* [*transfer-rule*]: (*pcr-stream* (=) ===> (=)) *id llist-of-stream*
⟨*proof*⟩

lemma *stream-of-llist-transfer* [*transfer-rule*]:

(*eq-onp* (λ*xs*. ¬ *lfinite xs*) ===> *pcr-stream* (=)) (λ*xs*. *xs*) *stream-of-llist*
⟨*proof*⟩

lemma *SCons-transfer* [*transfer-rule*]:
(*A* ===> *pcr-stream A* ===> *pcr-stream A*) *LCons* (##)
⟨*proof*⟩

lemma *sset-transfer* [*transfer-rule*]: (*pcr-stream A* ===> *rel-set A*) *lset sset*
⟨*proof*⟩

lemma *smap-transfer* [*transfer-rule*]:
((*A* ===> *B*) ===> *pcr-stream A* ===> *pcr-stream B*) *lmap smap*
⟨*proof*⟩

lemma *snth-transfer* [*transfer-rule*]: (*pcr-stream* (=) ===> (=)) *lnth snth*
⟨*proof*⟩

lemma *siterate-transfer* [*transfer-rule*]:
((=) ===> (=) ===> *pcr-stream* (=)) *iterates siterate*
⟨*proof*⟩

context
fixes *xs*
assumes *inf*: ¬ *lfinite xs*
notes [*transfer-rule*] = *eq-onpI*[**where** *P*=λ*xs*. ¬ *lfinite xs*, *OF inf*]
begin

lemma *smap-stream-of-llist* [*simp*]:
shows *smap f* (*stream-of-llist xs*) = *stream-of-llist* (*lmap f xs*)
⟨*proof*⟩

lemma *sset-stream-of-llist* [*simp*]:
assumes ¬ *lfinite xs*
shows *sset* (*stream-of-llist xs*) = *lset xs*
⟨*proof*⟩

end

lemma *llist-all2-llist-of-stream* [*simp*]:
llist-all2 P (*llist-of-stream xs*) (*llist-of-stream ys*) = *stream-all2 P xs ys*
⟨*proof*⟩

lemma *stream-all2-transfer* [*transfer-rule*]:
((=) ===> *pcr-stream* (=) ===> *pcr-stream* (=) ===> (=)) *llist-all2 stream-all2*
⟨*proof*⟩

lemma *stream-all2-coinduct*:
assumes *X xs ys*
and ⋀*xs ys*. *X xs ys* ⟹ *P* (*shd xs*) (*shd ys*) ∧ (*X* (*stl xs*) (*stl ys*) ∨ *stream-all2*

P (*stl xs*) (*stl ys*))
shows *stream-all2 P xs ys*
⟨*proof*⟩

lemma *shift-transfer* [*transfer-rule*]:
((=) ===> *pcr-stream* (=) ===> *pcr-stream* (=)) (*lappend* ∘ *llist-of*) *shift*
⟨*proof*⟩

lemma *szip-transfer* [*transfer-rule*]:
(*pcr-stream* (=) ===> *pcr-stream* (=) ===> *pcr-stream* (=)) *lzip szip*
⟨*proof*⟩

4.3 Link *'a stream* with *nat* ⇒ *'a*

lift-definition *of-seq* :: (*nat* ⇒ *'a*) ⇒ *'a stream* **is** *inf-llist* ⟨*proof*⟩

lemma *of-seq-rec* [*code*]: *of-seq f* = *f 0* ## *of-seq* (*f* ∘ *Suc*)
⟨*proof*⟩

lemma *snth-of-seq* [*simp*]: *snth* (*of-seq f*) = *f*
⟨*proof*⟩

lemma *snth-SCons*: *snth* (*x* ## *xs*) *n* = (*case n of 0* ⇒ *x* | *Suc n'* ⇒ *snth xs n'*)
⟨*proof*⟩

lemma *snth-SCons-simps* [*simp*]:
shows *snth-SCons-0*: (*x* ## *xs*) !! *0* = *x*
and *snth-SCons-Suc*: (*x* ## *xs*) !! *Suc n* = *xs* !! *n*
⟨*proof*⟩

lemma *of-seq-snth* [*simp*]: *of-seq* (*snth xs*) = *xs*
⟨*proof*⟩

lemma *shd-of-seq* [*simp*]: *shd* (*of-seq f*) = *f 0*
⟨*proof*⟩

lemma *stl-of-seq* [*simp*]: *stl* (*of-seq f*) = *of-seq* (λ*n*. *f* (*Suc n*))
⟨*proof*⟩

lemma *sset-of-seq* [*simp*]: *sset* (*of-seq f*) = *range f*
⟨*proof*⟩

lemma *smap-of-seq* [*simp*]: *smap f* (*of-seq g*) = *of-seq* (*f* ∘ *g*)
⟨*proof*⟩
end

4.4 Function iteration *siterate* and *sconst*

lemmas *siterate* [*nitpick-simp*] = *siterate.code*

lemma *smap-iterates*: *smap f* (*siterate f x*) = *siterate f* (*f x*)
⟨*proof*⟩

lemma *siterate-smap*: *siterate f x* = *x* ## (*smap f* (*siterate f x*))
⟨*proof*⟩

lemma *siterate-conv-of-seq*: *siterate f a* = *of-seq* (λ*n*. (*f* ^^ *n*) *a*)
⟨*proof*⟩

lemma *sconst-conv-of-seq*: *sconst a* = *of-seq* (λ-. *a*)
⟨*proof*⟩

lemma *szip-sconst1* [*simp*]: *szip* (*sconst a*) *xs* = *smap* (*Pair a*) *xs*
⟨*proof*⟩

lemma *szip-sconst2* [*simp*]: *szip xs* (*sconst b*) = *smap* (λ*x*. (*x*, *b*)) *xs*
⟨*proof*⟩

end

4.5 Counting elements

partial-function (*lfp*) *scount* :: ('*s stream* ⇒ *bool*) ⇒ '*s stream* ⇒ *enat* **where**
scount P ω = (*if P ω then eSuc* (*scount P* (*stl ω*)) *else scount P* (*stl ω*))

lemma *scount-simps*:
P ω ⟹ *scount P ω* = *eSuc* (*scount P* (*stl ω*))
¬ *P ω* ⟹ *scount P ω* = *scount P* (*stl ω*)
⟨*proof*⟩

lemma *scount-eq-0I*: *alw* (*not P*) *ω* ⟹ *scount P ω* = *0*
⟨*proof*⟩

lemma *scount-eq-0D*: *scount P ω* = *0* ⟹ *alw* (*not P*) *ω*
⟨*proof*⟩

lemma *scount-eq-0-iff*: *scount P ω* = *0* ⟷ *alw* (*not P*) *ω*
⟨*proof*⟩

lemma
assumes *ev* (*alw* (*not P*)) *ω*
shows *scount-eq-card*: *scount P ω* = *enat* (*card* {*i*. *P* (*sdrop i ω*)})
and *ev-alw-not-HLD-finite*: *finite* {*i*. *P* (*sdrop i ω*)}
⟨*proof*⟩

lemma *scount-finite*: *ev* (*alw* (*not P*)) *ω* ⟹ *scount P ω* < ∞
⟨*proof*⟩

lemma *scount-infinite*:

alw (ev P) ω ⟹ scount P ω = ∞
⟨*proof*⟩

lemma *scount-infinite-iff*: *scount P ω = ∞ ⟷ alw (ev P) ω*
⟨*proof*⟩

lemma *scount-eq*:
scount P ω = (if alw (ev P) ω then ∞ else enat (card {i. P (sdrop i ω)}))
⟨*proof*⟩

4.6 First index of an element

partial-function (*gfp*) *sfirst* :: (*'s stream ⇒ bool*) *⇒ 's stream ⇒ enat* **where**
sfirst P ω = (if P ω then 0 else eSuc (sfirst P (stl ω)))

lemma *sfirst-eq-0*: *sfirst P ω = 0 ⟷ P ω*
⟨*proof*⟩

lemma *sfirst-0*[*simp*]: *P ω ⟹ sfirst P ω = 0*
⟨*proof*⟩

lemma *sfirst-eSuc*[*simp*]: *¬ P ω ⟹ sfirst P ω = eSuc (sfirst P (stl ω))*
⟨*proof*⟩

lemma *less-sfirstD*:
fixes *n* :: *nat*
assumes *enat n < sfirst P ω* **shows** *¬ P (sdrop n ω)*
⟨*proof*⟩

lemma *sfirst-finite*: *sfirst P ω < ∞ ⟷ ev P ω*
⟨*proof*⟩

lemma *sfirst-Stream*: *sfirst P (s ## x) = (if P (s ## x) then 0 else eSuc (sfirst P x))*
⟨*proof*⟩

lemma *less-sfirst-iff*: *(not P until (alw P)) ω ⟹ enat n < sfirst P ω ⟷ ¬ P (sdrop n ω)*
⟨*proof*⟩

lemma *sfirst-eq-Inf*: *sfirst P ω = Inf {enat i | i. P (sdrop i ω)}*
⟨*proof*⟩

lemma *sfirst-eq-enat-iff*: *sfirst P ω = enat n ⟷ ev-at P n ω*
⟨*proof*⟩

4.7 *stakeWhile*

definition *stakeWhile* :: *('a ⇒ bool) ⇒ 'a stream ⇒ 'a llist*
where *stakeWhile P xs = ltakeWhile P (llist-of-stream xs)*

lemma *stakeWhile-SCons* [*simp*]:
stakeWhile P (*x ## xs*) = (*if P x then LCons x* (*stakeWhile P xs*) *else LNil*)
⟨*proof*⟩

lemma *lnull-stakeWhile* [*simp*]: *lnull* (*stakeWhile P xs*) ⟷ ¬ *P* (*shd xs*)
⟨*proof*⟩

lemma *lhd-stakeWhile* [*simp*]: *P* (*shd xs*) ⟹ *lhd* (*stakeWhile P xs*) = *shd xs*
⟨*proof*⟩

lemma *ltl-stakeWhile* [*simp*]:
ltl (*stakeWhile P xs*) = (*if P* (*shd xs*) *then stakeWhile P* (*stl xs*) *else LNil*)
⟨*proof*⟩

lemma *stakeWhile-K-False* [*simp*]: *stakeWhile* (λ-. *False*) *xs* = *LNil*
⟨*proof*⟩

lemma *stakeWhile-K-True* [*simp*]: *stakeWhile* (λ-. *True*) *xs* = *llist-of-stream xs*
⟨*proof*⟩

lemma *stakeWhile-smap*: *stakeWhile P* (*smap f xs*) = *lmap f* (*stakeWhile* (*P* ∘ *f*) *xs*)
⟨*proof*⟩

lemma *lfinite-stakeWhile* [*simp*]: *lfinite* (*stakeWhile P xs*) ⟷ (∃ *x*∈*sset xs*. ¬ *P x*)
⟨*proof*⟩

end

5 Terminated coinductive lists and their operations

theory *TLList* **imports**
Coinductive-List
begin

Terminated coinductive lists (′*a*, ′*b*) *tllist* are the codatatype defined by the construtors *TNil* of type ′*b* ⇒ (′*a*, ′*b*) *tllist* and *TCons* of type ′*a* ⇒ (′*a*, ′*b*) *tllist* ⇒ (′*a*, ′*b*) *tllist*.

5.1 Auxiliary lemmas

lemma *split-fst*: *R* (*fst p*) = (∀ *x y*. *p* = (*x*, *y*) ⟶ *R x*)
⟨*proof*⟩

lemma *split-fst-asm*: *R* (*fst p*) ⟷ (¬ (∃ *x y*. *p* = (*x*, *y*) ∧ ¬ *R x*))

⟨*proof*⟩

5.2 Type definition

consts *terminal0* :: *'a*

codatatype (*tset*: *'a*, *'b*) *tllist* =
  TNil (*terminal* : *'b*)
| *TCons* (*thd* : *'a*) (*ttl* : (*'a*, *'b*) *tllist*)
for
  map: *tmap*
  rel: *tllist-all2*
where
  thd (*TNil* -) = *undefined*
| *ttl* (*TNil b*) = *TNil b*
| *terminal* (*TCons* - *xs*) = *terminal0 xs*

overloading
  terminal0 == *terminal0*::(*'a*, *'b*) *tllist* ⇒ *'b*
begin

partial-function (*tailrec*) *terminal0*
where *terminal0 xs* = (*if is-TNil xs then case-tllist id undefined xs else terminal0* (*ttl xs*))

end

lemma *terminal0-terminal* [*simp*]: *terminal0* = *terminal*
⟨*proof*⟩

lemmas *terminal-TNil* [*code*, *nitpick-simp*] = *tllist.sel*(*1*)

lemma *terminal-TCons* [*simp*, *code*, *nitpick-simp*]: *terminal* (*TCons x xs*) = *terminal xs*
⟨*proof*⟩

declare *tllist.sel*(*2*) [*simp del*]

primcorec *unfold-tllist* :: (*'a* ⇒ *bool*) ⇒ (*'a* ⇒ *'b*) ⇒ (*'a* ⇒ *'c*) ⇒ (*'a* ⇒ *'a*) ⇒ *'a* ⇒ (*'c*, *'b*) *tllist* **where**
  p a ⟹ *unfold-tllist p g1 g21 g22 a* = *TNil* (*g1 a*) |
  - ⟹ *unfold-tllist p g1 g21 g22 a* =
    TCons (*g21 a*) (*unfold-tllist p g1 g21 g22* (*g22 a*))

declare
  unfold-tllist.ctr(*1*) [*simp*]
  tllist.corec(*1*) [*simp*]

5.3 Code generator setup

Test quickcheck setup

lemma *xs = TNil x*
quickcheck[*random*, *expect=counterexample*]
quickcheck[*exhaustive*, *expect=counterexample*]
⟨*proof*⟩

lemma *TCons x xs = TCons x xs*
quickcheck[*narrowing*, *expect=no-counterexample*]
⟨*proof*⟩

More lemmas about generated constants

lemma *ttl-unfold-tllist*:
ttl (*unfold-tllist IS-TNIL TNIL THD TTL a*) =
(*if IS-TNIL a then TNil* (*TNIL a*) *else unfold-tllist IS-TNIL TNIL THD TTL* (*TTL a*))
⟨*proof*⟩

lemma *is-TNil-ttl* [*simp*]: *is-TNil xs* ⟹ *is-TNil* (*ttl xs*)
⟨*proof*⟩

lemma *terminal-ttl* [*simp*]: *terminal* (*ttl xs*) = *terminal xs*
⟨*proof*⟩

lemma *unfold-tllist-eq-TNil* [*simp*]:
unfold-tllist IS-TNIL TNIL THD TTL a = TNil b ⟷ *IS-TNIL a* ∧ *b = TNIL a*
⟨*proof*⟩

lemma *TNil-eq-unfold-tllist* [*simp*]:
TNil b = unfold-tllist IS-TNIL TNIL THD TTL a ⟷ *IS-TNIL a* ∧ *b = TNIL a*
⟨*proof*⟩

lemma *tmap-is-TNil*: *is-TNil xs* ⟹ *tmap f g xs = TNil* (*g* (*terminal xs*))
⟨*proof*⟩

declare *tllist.map-sel*(*2*)[*simp*]

lemma *ttl-tmap* [*simp*]: *ttl* (*tmap f g xs*) = *tmap f g* (*ttl xs*)
⟨*proof*⟩

lemma *tmap-eq-TNil-conv*:
tmap f g xs = TNil y ⟷ (∃ *y'*. *xs = TNil y'* ∧ *g y' = y*)
⟨*proof*⟩

lemma *TNil-eq-tmap-conv*:
TNil y = tmap f g xs ⟷ (∃ *y'*. *xs = TNil y'* ∧ *g y' = y*)

⟨*proof*⟩

declare *tllist.set-sel*(*1*)[*simp*]

lemma *tset-ttl*: *tset* (*ttl xs*) ⊆ *tset xs*
⟨*proof*⟩

lemma *in-tset-ttlD*: *x* ∈ *tset* (*ttl xs*) ⟹ *x* ∈ *tset xs*
⟨*proof*⟩

theorem *tllist-set-induct*[*consumes 1*, *case-names find step*]:
assumes *x* ∈ *tset xs* **and** ⋀*xs*. ¬ *is-TNil xs* ⟹ *P* (*thd xs*) *xs*
and ⋀*xs y*. ⟦¬ *is-TNil xs*; *y* ∈ *tset* (*ttl xs*); *P y* (*ttl xs*)⟧ ⟹ *P y xs*
shows *P x xs*
⟨*proof*⟩

theorem *set2-tllist-induct*[*consumes 1*, *case-names find step*]:
assumes *x* ∈ *set2-tllist xs* **and** ⋀*xs*. *is-TNil xs* ⟹ *P* (*terminal xs*) *xs*
and ⋀*xs y*. ⟦¬ *is-TNil xs*; *y* ∈ *set2-tllist* (*ttl xs*); *P y* (*ttl xs*)⟧ ⟹ *P y xs*
shows *P x xs*
⟨*proof*⟩

5.4 Connection with *'a llist*

context fixes *b* :: *'b* **begin**
primcorec *tllist-of-llist* :: *'a llist* ⇒ (*'a*, *'b*) *tllist* **where**
tllist-of-llist xs = (*case xs of LNil* ⇒ *TNil b* | *LCons x xs'* ⇒ *TCons x* (*tllist-of-llist xs'*))
end

primcorec *llist-of-tllist* :: (*'a*, *'b*) *tllist* ⇒ *'a llist*
where *llist-of-tllist xs* = (*case xs of TNil* - ⇒ *LNil* | *TCons x xs'* ⇒ *LCons x* (*llist-of-tllist xs'*))

simps-of-case *tllist-of-llist-simps* [*simp*, *code*, *nitpick-simp*]: *tllist-of-llist.code*

lemmas *tllist-of-llist-LNil* = *tllist-of-llist-simps*(*1*)
and *tllist-of-llist-LCons* = *tllist-of-llist-simps*(*2*)

lemma *terminal-tllist-of-llist-lnull* [*simp*]:
lnull xs ⟹ *terminal* (*tllist-of-llist b xs*) = *b*
⟨*proof*⟩

declare *tllist-of-llist.sel*(*1*)[*simp del*]

lemma *lhd-LNil*: *lhd LNil* = *undefined*
⟨*proof*⟩

lemma *thd-TNil*: *thd* (*TNil b*) = *undefined*

⟨*proof*⟩

lemma *thd-tllist-of-llist* [*simp*]: *thd* (*tllist-of-llist b xs*) = *lhd xs*
⟨*proof*⟩

lemma *ttl-tllist-of-llist* [*simp*]: *ttl* (*tllist-of-llist b xs*) = *tllist-of-llist b* (*ltl xs*)
⟨*proof*⟩

lemma *llist-of-tllist-eq-LNil*:
llist-of-tllist xs = *LNil* ⟷ *is-TNil xs*
⟨*proof*⟩

simps-of-case *llist-of-tllist-simps* [*simp*, *code*, *nitpick-simp*]: *llist-of-tllist.code*

lemmas *llist-of-tllist-TNil* = *llist-of-tllist-simps*(*1*)
and *llist-of-tllist-TCons* = *llist-of-tllist-simps*(*2*)

declare *llist-of-tllist.sel* [*simp del*]

lemma *lhd-llist-of-tllist* [*simp*]: ¬ *is-TNil xs* ⟹ *lhd* (*llist-of-tllist xs*) = *thd xs*
⟨*proof*⟩

lemma *ltl-llist-of-tllist* [*simp*]:
ltl (*llist-of-tllist xs*) = *llist-of-tllist* (*ttl xs*)
⟨*proof*⟩

lemma *tllist-of-llist-cong* [*cong*]:
assumes *xs* = *xs′ lfinite xs′* ⟹ *b* = *b′*
shows *tllist-of-llist b xs* = *tllist-of-llist b′ xs′*
⟨*proof*⟩

lemma *llist-of-tllist-inverse* [*simp*]:
tllist-of-llist (*terminal b*) (*llist-of-tllist b*) = *b*
⟨*proof*⟩

lemma *tllist-of-llist-eq* [*simp*]: *tllist-of-llist b′ xs* = *TNil b* ⟷ *b* = *b′* ∧ *xs* = *LNil*
⟨*proof*⟩

lemma *TNil-eq-tllist-of-llist* [*simp*]: *TNil b* = *tllist-of-llist b′ xs* ⟷ *b* = *b′* ∧ *xs* = *LNil*
⟨*proof*⟩

lemma *tllist-of-llist-inject* [*simp*]:
tllist-of-llist b xs = *tllist-of-llist c ys* ⟷ *xs* = *ys* ∧ (*lfinite ys* ⟶ *b* = *c*)
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *tllist-of-llist-inverse* [*simp*]:
llist-of-tllist (*tllist-of-llist b xs*) = *xs*

⟨*proof*⟩

definition *cr-tllist* :: ('*a llist* × '*b*) ⇒ ('*a*, '*b*) *tllist* ⇒ *bool*
  where *cr-tllist* ≡ (λ(*xs*, *b*) *ys*. *tllist-of-llist b xs* = *ys*)

lemma *Quotient-tllist*:
  Quotient (λ(*xs*, *a*) (*ys*, *b*). *xs* = *ys* ∧ (*lfinite ys* ⟶ *a* = *b*))
    (λ(*xs*, *a*). *tllist-of-llist a xs*) (λ*ys*. (*llist-of-tllist ys*, *terminal ys*)) *cr-tllist*
⟨*proof*⟩

lemma *reflp-tllist*: *reflp* (λ(*xs*, *a*) (*ys*, *b*). *xs* = *ys* ∧ (*lfinite ys* ⟶ *a* = *b*))
⟨*proof*⟩

setup-lifting *Quotient-tllist reflp-tllist*

context includes *lifting-syntax*
begin

lemma *TNil-transfer* [*transfer-rule*]:
  (*B* ===> *pcr-tllist A B*) (*Pair LNil*) *TNil*
⟨*proof*⟩

lemma *TCons-transfer* [*transfer-rule*]:
  (*A* ===> *pcr-tllist A B* ===> *pcr-tllist A B*) (*apfst* ∘ *LCons*) *TCons*
⟨*proof*⟩

lemma *tmap-tllist-of-llist*:
  tmap f g (*tllist-of-llist b xs*) = *tllist-of-llist* (*g b*) (*lmap f xs*)
⟨*proof*⟩

lemma *tmap-transfer* [*transfer-rule*]:
  ((=) ===> (=) ===> *pcr-tllist* (=) (=) ===> *pcr-tllist* (=) (=)) (*map-prod* ∘ *lmap*) *tmap*
⟨*proof*⟩

lemma *lset-llist-of-tllist* [*simp*]:
  lset (*llist-of-tllist xs*) = *tset xs* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *tset-tllist-of-llist* [*simp*]:
  tset (*tllist-of-llist b xs*) = *lset xs*
⟨*proof*⟩

lemma *tset-transfer* [*transfer-rule*]:
  (*pcr-tllist* (=) (=) ===> (=)) (*lset* ∘ *fst*) *tset*
⟨*proof*⟩

lemma *is-TNil-transfer* [*transfer-rule*]:
  (*pcr-tllist* (=) (=) ===> (=)) (λ(*xs*, *b*). *lnull xs*) *is-TNil*

⟨*proof*⟩

lemma *thd-transfer* [*transfer-rule*]:
(*pcr-tllist* (=) (=) ===> (=)) (*lhd* ∘ *fst*) *thd*
⟨*proof*⟩

lemma *ttl-transfer* [*transfer-rule*]:
(*pcr-tllist A B* ===> *pcr-tllist A B*) (*apfst ltl*) *ttl*
⟨*proof*⟩

lemma *llist-of-tllist-transfer* [*transfer-rule*]:
(*pcr-tllist* (=) *B* ===> (=)) *fst llist-of-tllist*
⟨*proof*⟩

lemma *tllist-of-llist-transfer* [*transfer-rule*]:
((=) ===> (=) ===> *pcr-tllist* (=) (=)) (λ*b xs*. (*xs*, *b*)) *tllist-of-llist*
⟨*proof*⟩

lemma *terminal-tllist-of-llist-lfinite* [*simp*]:
lfinite xs ⟹ *terminal* (*tllist-of-llist b xs*) = *b*
⟨*proof*⟩

lemma *set2-tllist-tllist-of-llist* [*simp*]:
set2-tllist (*tllist-of-llist b xs*) = (*if lfinite xs then* {*b*} *else* {})
⟨*proof*⟩

lemma *set2-tllist-transfer* [*transfer-rule*]:
(*pcr-tllist A B* ===> *rel-set B*) (λ(*xs*, *b*). *if lfinite xs then* {*b*} *else* {}) *set2-tllist*
⟨*proof*⟩

lemma *tllist-all2-transfer* [*transfer-rule*]:
((=) ===> (=) ===> *pcr-tllist* (=) (=) ===> *pcr-tllist* (=) (=) ===> (=))
(λ*P Q* (*xs*, *b*) (*ys*, *b′*). *llist-all2 P xs ys* ∧ (*lfinite xs* ⟶ *Q b b′*)) *tllist-all2*
⟨*proof*⟩

5.5 Library function definitions

We lift the constants from *′a llist* to (*′a*, *′b*) *tllist* using the lifting package. This way, many results are transferred easily.

lift-definition *tappend* :: (*′a*, *′b*) *tllist* ⇒ (*′b* ⇒ (*′a*, *′c*) *tllist*) ⇒ (*′a*, *′c*) *tllist*
is λ(*xs*, *b*) *f*. *apfst* (*lappend xs*) (*f b*)
⟨*proof*⟩

lift-definition *lappendt* :: *′a llist* ⇒ (*′a*, *′b*) *tllist* ⇒ (*′a*, *′b*) *tllist*
is *apfst* ∘ *lappend*
⟨*proof*⟩

lift-definition *tfilter* :: *′b* ⇒ (*′a* ⇒ *bool*) ⇒ (*′a*, *′b*) *tllist* ⇒ (*′a*, *′b*) *tllist*
is λ*b P* (*xs*, *b′*). (*lfilter P xs*, *if lfinite xs then b′ else b*)

⟨*proof*⟩

lift-definition *tconcat* :: *'b* ⇒ (*'a llist*, *'b*) *tllist* ⇒ (*'a*, *'b*) *tllist*
is λ*b* (*xss*, *b'*). (*lconcat xss*, *if lfinite xss then b' else b*)
⟨*proof*⟩

lift-definition *tnth* :: (*'a*, *'b*) *tllist* ⇒ *nat* ⇒ *'a*
is *lnth* ∘ *fst* ⟨*proof*⟩

lift-definition *tlength* :: (*'a*, *'b*) *tllist* ⇒ *enat*
is *llength* ∘ *fst* ⟨*proof*⟩

lift-definition *tdropn* :: *nat* ⇒ (*'a*, *'b*) *tllist* ⇒ (*'a*, *'b*) *tllist*
is *apfst* ∘ *ldropn* ⟨*proof*⟩

abbreviation *tfinite* :: (*'a*, *'b*) *tllist* ⇒ *bool*
where *tfinite xs* ≡ *lfinite* (*llist-of-tllist xs*)

5.6 *tfinite*

lemma *tfinite-induct* [*consumes 1*, *case-names TNil TCons*]:
  assumes *tfinite xs*
  and ⋀*y*. *P* (*TNil y*)
  and ⋀*x xs*. ⟦*tfinite xs*; *P xs*⟧ ⟹ *P* (*TCons x xs*)
  shows *P xs*
⟨*proof*⟩

lemma *is-TNil-tfinite* [*simp*]: *is-TNil xs* ⟹ *tfinite xs*
⟨*proof*⟩

5.7 The terminal element *terminal*

lemma *terminal-tinfinite*:
  assumes ¬ *tfinite xs*
  shows *terminal xs* = *undefined*
⟨*proof*⟩

lemma *terminal-tllist-of-llist*:
  terminal (*tllist-of-llist y xs*) = (*if lfinite xs then y else undefined*)
⟨*proof*⟩

lemma *terminal-transfer* [*transfer-rule*]:
  (*pcr-tllist A* (=) ===> (=)) (λ(*xs*, *b*). *if lfinite xs then b else undefined*) *terminal*
⟨*proof*⟩

lemma *terminal-tmap* [*simp*]: *tfinite xs* ⟹ *terminal* (*tmap f g xs*) = *g* (*terminal xs*)
⟨*proof*⟩

5.8 *tmap*

lemma *tmap-eq-TCons-conv*:
tmap f g xs = TCons y ys $\longleftrightarrow$
($\exists$ *z zs. xs = TCons z zs* $\wedge$ *f z = y* $\wedge$ *tmap f g zs = ys)*
⟨*proof*⟩

lemma *TCons-eq-tmap-conv*:
TCons y ys = tmap f g xs $\longleftrightarrow$
($\exists$ *z zs. xs = TCons z zs* $\wedge$ *f z = y* $\wedge$ *tmap f g zs = ys)*
⟨*proof*⟩

5.9 Appending two terminated lazy lists *tappend*

lemma *tappend-TNil* [*simp*, *code*, *nitpick-simp*]:
tappend (TNil b) f = f b
⟨*proof*⟩

lemma *tappend-TCons* [*simp*, *code*, *nitpick-simp*]:
tappend (TCons a tr) f = TCons a (tappend tr f)
⟨*proof*⟩

lemma *tappend-TNil2* [*simp*]:
tappend xs TNil = xs
⟨*proof*⟩

lemma *tappend-assoc*: *tappend (tappend xs f) g = tappend xs (*λ*b. tappend (f b) g)*
⟨*proof*⟩

lemma *terminal-tappend*:
terminal (tappend xs f) = (if tfinite xs then terminal (f (terminal xs)) else terminal xs)
⟨*proof*⟩

lemma *tfinite-tappend*: *tfinite (tappend xs f)* $\longleftrightarrow$ *tfinite xs* $\wedge$ *tfinite (f (terminal xs))*
⟨*proof*⟩

lift-definition *tcast* :: *('a, 'b) tllist* $\Rightarrow$ *('a, 'c) tllist*
is λ*(xs, a). (xs, undefined)* ⟨*proof*⟩

lemma *tappend-inf*: $\neg$ *tfinite xs* $\Longrightarrow$ *tappend xs f = tcast xs*
⟨*proof*⟩

tappend is the monadic bind on *('a, 'b) tllist*

lemmas *tllist-monad = tappend-TNil tappend-TNil2 tappend-assoc*

5.10 Appending a terminated lazy list to a lazy list *lappendt*

lemma *lappendt-LNil* [*simp*, *code*, *nitpick-simp*]: *lappendt LNil tr = tr*

⟨*proof*⟩

lemma *lappendt-LCons* [*simp*, *code*, *nitpick-simp*]:
lappendt (*LCons x xs*) *tr* = *TCons x* (*lappendt xs tr*)
⟨*proof*⟩

lemma *terminal-lappendt-lfinite* [*simp*]:
lfinite xs ⟹ *terminal* (*lappendt xs ys*) = *terminal ys*
⟨*proof*⟩

lemma *tllist-of-llist-eq-lappendt-conv*:
tllist-of-llist a xs = *lappendt ys zs* ⟷
(∃ *xs′ a′*. *xs* = *lappend ys xs′* ∧ *zs* = *tllist-of-llist a′ xs′* ∧ (*lfinite ys* ⟶ *a* = *a′*))
⟨*proof*⟩

lemma *tset-lappendt-lfinite* [*simp*]:
lfinite xs ⟹ *tset* (*lappendt xs ys*) = *lset xs* ∪ *tset ys*
⟨*proof*⟩

5.11 Filtering terminated lazy lists *tfilter*

lemma *tfilter-TNil* [*simp*]:
tfilter b′ P (*TNil b*) = *TNil b*
⟨*proof*⟩

lemma *tfilter-TCons* [*simp*]:
tfilter b P (*TCons a tr*) = (*if P a then TCons a* (*tfilter b P tr*) *else tfilter b P tr*)
⟨*proof*⟩

lemma *is-TNil-tfilter*[*simp*]:
is-TNil (*tfilter y P xs*) ⟷ (∀ *x* ∈ *tset xs*. ¬ *P x*)
⟨*proof*⟩

lemma *tfilter-empty-conv*:
tfilter y P xs = *TNil y′* ⟷ (∀ *x* ∈ *tset xs*. ¬ *P x*) ∧ (*if tfinite xs then terminal xs* = *y′ else y* = *y′*)
⟨*proof*⟩

lemma *tfilter-eq-TConsD*:
tfilter a P ys = *TCons x xs* ⟹
∃ *us vs*. *ys* = *lappendt us* (*TCons x vs*) ∧ *lfinite us* ∧ (∀ *u*∈*lset us*. ¬ *P u*) ∧ *P x* ∧ *xs* = *tfilter a P vs*
⟨*proof*⟩

Use a version of *tfilter* for code generation that does not evaluate the first argument

definition *tfilter′* :: (*unit* ⇒ *′b*) ⇒ (*′a* ⇒ *bool*) ⇒ (*′a*, *′b*) *tllist* ⇒ (*′a*, *′b*) *tllist*
where [*simp*, *code del*]: *tfilter′ b* = *tfilter* (*b* ())

lemma *tfilter-code* [*code*, *code-unfold*]:
  tfilter = (λ*b*. *tfilter′* (λ-. *b*))
⟨*proof*⟩

lemma *tfilter′-code* [*code*]:
  tfilter′ b′ P (*TNil b*) = *TNil b*
  tfilter′ b′ P (*TCons a tr*) = (*if P a then TCons a* (*tfilter′ b′ P tr*) *else tfilter′ b′ P tr*)
⟨*proof*⟩

end

hide-const (**open**) *tfilter′*

5.12 Concatenating a terminated lazy list of lazy lists *tconcat*

lemma *tconcat-TNil* [*simp*]: *tconcat b* (*TNil b′*) = *TNil b′*
⟨*proof*⟩

lemma *tconcat-TCons* [*simp*]: *tconcat b* (*TCons a tr*) = *lappendt a* (*tconcat b tr*)
⟨*proof*⟩

Use a version of *tconcat* for code generation that does not evaluate the first argument

definition *tconcat′* :: (*unit* ⇒ *′b*) ⇒ (*′a llist*, *′b*) *tllist* ⇒ (*′a*, *′b*) *tllist*
where [*simp*, *code del*]: *tconcat′ b* = *tconcat* (*b* ())

lemma *tconcat-code* [*code*, *code-unfold*]: *tconcat* = (λ*b*. *tconcat′* (λ-. *b*))
⟨*proof*⟩

lemma *tconcat′-code* [*code*]:
  tconcat′ b (*TNil b′*) = *TNil b′*
  tconcat′ b (*TCons a tr*) = *lappendt a* (*tconcat′ b tr*)
⟨*proof*⟩

hide-const (**open**) *tconcat′*

5.13 *tllist-all2*

lemmas *tllist-all2-TNil* = *tllist.rel-inject*(*1*)
lemmas *tllist-all2-TCons* = *tllist.rel-inject*(*2*)

lemma *tllist-all2-TNil1*: *tllist-all2 P Q* (*TNil b*) *ts* ⟷ (∃ *b′*. *ts* = *TNil b′* ∧ *Q b b′*)
⟨*proof*⟩

lemma *tllist-all2-TNil2*: *tllist-all2 P Q ts* (*TNil b′*) ⟷ (∃ *b*. *ts* = *TNil b* ∧ *Q b b′*)
⟨*proof*⟩

lemma *tllist-all2-TCons1*:
tllist-all2 P Q (TCons x ts) ts' ⟷ (∃ x' ts''. ts' = TCons x' ts'' ∧ P x x' ∧ tllist-all2 P Q ts ts'')
⟨*proof*⟩

lemma *tllist-all2-TCons2*:
tllist-all2 P Q ts' (TCons x ts) ⟷ (∃ x' ts''. ts' = TCons x' ts'' ∧ P x' x ∧ tllist-all2 P Q ts'' ts)
⟨*proof*⟩

lemma *tllist-all2-coinduct* [*consumes 1*, *case-names tllist-all2*, *case-conclusion tllist-all2 is-TNil TNil TCons*, *coinduct pred*: *tllist-all2*]:
assumes *X xs ys*
and ⋀*xs ys*. *X xs ys* ⟹
(*is-TNil xs ⟷ is-TNil ys*) ∧
(*is-TNil xs ⟶ is-TNil ys ⟶ R* (*terminal xs*) (*terminal ys*)) ∧
(¬ *is-TNil xs ⟶* ¬ *is-TNil ys ⟶ P* (*thd xs*) (*thd ys*) ∧ (*X* (*ttl xs*) (*ttl ys*) ∨ *tllist-all2 P R* (*ttl xs*) (*ttl ys*)))
shows *tllist-all2 P R xs ys*
⟨*proof*⟩

lemma *tllist-all2-cases*[*consumes 1*, *case-names TNil TCons*, *cases pred*]:
assumes *tllist-all2 P Q xs ys*
obtains (*TNil*) *b b'* **where** *xs = TNil b ys = TNil b' Q b b'*
| (*TCons*) *x xs' y ys'*
where *xs = TCons x xs'* **and** *ys = TCons y ys'*
and *P x y* **and** *tllist-all2 P Q xs' ys'*
⟨*proof*⟩

lemma *tllist-all2-tmap1*:
tllist-all2 P Q (*tmap f g xs*) *ys ⟷ tllist-all2* (λ*x*. *P* (*f x*)) (λ*x*. *Q* (*g x*)) *xs ys*
⟨*proof*⟩

lemma *tllist-all2-tmap2*:
tllist-all2 P Q xs (*tmap f g ys*) ⟷ *tllist-all2* (λ*x y*. *P x* (*f y*)) (λ*x y*. *Q x* (*g y*)) *xs ys*
⟨*proof*⟩

lemma *tllist-all2-mono*:
⟦ *tllist-all2 P Q xs ys*; ⋀*x y*. *P x y* ⟹ *P' x y*; ⋀*x y*. *Q x y* ⟹ *Q' x y* ⟧
⟹ *tllist-all2 P' Q' xs ys*
⟨*proof*⟩

lemma *tllist-all2-tlengthD*: *tllist-all2 P Q xs ys* ⟹ *tlength xs = tlength ys*
⟨*proof*⟩

lemma *tllist-all2-tfiniteD*: *tllist-all2 P Q xs ys* ⟹ *tfinite xs = tfinite ys*
⟨*proof*⟩

lemma *tllist-all2-tfinite1-terminalD*:
⟦ *tllist-all2 P Q xs ys*; *tfinite xs* ⟧ ⟹ *Q* (*terminal xs*) (*terminal ys*)
⟨*proof*⟩

lemma *tllist-all2-tfinite2-terminalD*:
⟦ *tllist-all2 P Q xs ys*; *tfinite ys* ⟧ ⟹ *Q* (*terminal xs*) (*terminal ys*)
⟨*proof*⟩

lemma *tllist-all2D-llist-all2-llist-of-tllist*:
tllist-all2 P Q xs ys ⟹ *llist-all2 P* (*llist-of-tllist xs*) (*llist-of-tllist ys*)
⟨*proof*⟩

lemma *tllist-all2-is-TNilD*:
tllist-all2 P Q xs ys ⟹ *is-TNil xs* ⟷ *is-TNil ys*
⟨*proof*⟩

lemma *tllist-all2-thdD*:
⟦ *tllist-all2 P Q xs ys*; ¬ *is-TNil xs* ∨ ¬ *is-TNil ys* ⟧ ⟹ *P* (*thd xs*) (*thd ys*)
⟨*proof*⟩

lemma *tllist-all2-ttlI*:
⟦ *tllist-all2 P Q xs ys*; ¬ *is-TNil xs* ∨ ¬ *is-TNil ys* ⟧ ⟹ *tllist-all2 P Q* (*ttl xs*)
(*ttl ys*)
⟨*proof*⟩

lemma *tllist-all2-refl*:
tllist-all2 P Q xs xs ⟷ (∀ *x* ∈ *tset xs*. *P x x*) ∧ (*tfinite xs* ⟶ *Q* (*terminal xs*)
(*terminal xs*))
⟨*proof*⟩

lemma *tllist-all2-reflI*:
⟦ ⋀*x*. *x* ∈ *tset xs* ⟹ *P x x*; *tfinite xs* ⟹ *Q* (*terminal xs*) (*terminal xs*) ⟧
⟹ *tllist-all2 P Q xs xs*
⟨*proof*⟩

lemma *tllist-all2-conv-all-tnth*:
tllist-all2 P Q xs ys ⟷
tlength xs = *tlength ys* ∧
(∀ *n*. *enat n* < *tlength xs* ⟶ *P* (*tnth xs n*) (*tnth ys n*)) ∧
(*tfinite xs* ⟶ *Q* (*terminal xs*) (*terminal ys*))
⟨*proof*⟩

lemma *tllist-all2-tnthD*:
⟦ *tllist-all2 P Q xs ys*; *enat n* < *tlength xs* ⟧
⟹ *P* (*tnth xs n*) (*tnth ys n*)
⟨*proof*⟩

lemma *tllist-all2-tnthD2*:

⟦ *tllist-all2 P Q xs ys*; *enat n* < *tlength ys* ⟧
⟹ *P* (*tnth xs n*) (*tnth ys n*)
⟨*proof*⟩

lemmas *tllist-all2-eq* = *tllist.rel-eq*

lemma *tmap-eq-tmap-conv-tllist-all2*:
tmap f g xs = *tmap f′ g′ ys* ⟷
tllist-all2 (λ*x y*. *f x* = *f′ y*) (λ*x y*. *g x* = *g′ y*) *xs ys*
⟨*proof*⟩

lemma *tllist-all2-trans*:
⟦ *tllist-all2 P Q xs ys*; *tllist-all2 P Q ys zs*; *transp P*; *transp Q* ⟧
⟹ *tllist-all2 P Q xs zs*
⟨*proof*⟩

lemma *tllist-all2-tappendI*:
⟦ *tllist-all2 P Q xs ys*;
⟦ *tfinite xs*; *tfinite ys*; *Q* (*terminal xs*) (*terminal ys*) ⟧
⟹ *tllist-all2 P R* (*xs′* (*terminal xs*)) (*ys′* (*terminal ys*)) ⟧
⟹ *tllist-all2 P R* (*tappend xs xs′*) (*tappend ys ys′*)
⟨*proof*⟩

lemma *llist-all2-tllist-of-llistI*:
tllist-all2 A B xs ys ⟹ *llist-all2 A* (*llist-of-tllist xs*) (*llist-of-tllist ys*)
⟨*proof*⟩

lemma *tllist-all2-tllist-of-llist* [*simp*]:
tllist-all2 A B (*tllist-of-llist b xs*) (*tllist-of-llist c ys*) ⟷
llist-all2 A xs ys ∧ (*lfinite xs* ⟶ *B b c*)
⟨*proof*⟩

5.14 From a terminated lazy list to a lazy list *llist-of-tllist*

lemma *llist-of-tllist-tmap* [*simp*]:
llist-of-tllist (*tmap f g xs*) = *lmap f* (*llist-of-tllist xs*)
⟨*proof*⟩

lemma *llist-of-tllist-tappend*:
llist-of-tllist (*tappend xs f*) = *lappend* (*llist-of-tllist xs*) (*llist-of-tllist* (*f* (*terminal xs*)))
⟨*proof*⟩

lemma *llist-of-tllist-lappendt* [*simp*]:
llist-of-tllist (*lappendt xs tr*) = *lappend xs* (*llist-of-tllist tr*)
⟨*proof*⟩

lemma *llist-of-tllist-tfilter* [*simp*]:
llist-of-tllist (*tfilter b P tr*) = *lfilter P* (*llist-of-tllist tr*)

⟨*proof*⟩

lemma *llist-of-tllist-tconcat*:
llist-of-tllist (*tconcat b trs*) = *lconcat* (*llist-of-tllist trs*)
⟨*proof*⟩

lemma *llist-of-tllist-eq-lappend-conv*:
llist-of-tllist xs = *lappend us vs* ⟷
(∃ *ys*. *xs* = *lappendt us ys* ∧ *vs* = *llist-of-tllist ys* ∧ *terminal xs* = *terminal ys*)
⟨*proof*⟩

5.15 The nth element of a terminated lazy list *tnth*

lemma *tnth-TNil* [*nitpick-simp*]:
tnth (*TNil b*) *n* = *undefined n*
⟨*proof*⟩

lemma *tnth-TCons*:
tnth (*TCons x xs*) *n* = (*case n of 0* ⇒ *x* | *Suc n′* ⇒ *tnth xs n′*)
⟨*proof*⟩

lemma *tnth-code* [*simp*, *nitpick-simp*, *code*]:
shows *tnth-0*: *tnth* (*TCons x xs*) *0* = *x*
and *tnth-Suc-TCons*: *tnth* (*TCons x xs*) (*Suc n*) = *tnth xs n*
⟨*proof*⟩

lemma *lnth-llist-of-tllist* [*simp*]:
lnth (*llist-of-tllist xs*) = *tnth xs*
⟨*proof*⟩

lemma *tnth-tmap* [*simp*]: *enat n* < *tlength xs* ⟹ *tnth* (*tmap f g xs*) *n* = *f* (*tnth xs n*)
⟨*proof*⟩

5.16 The length of a terminated lazy list *tlength*

lemma [*simp*, *nitpick-simp*]:
shows *tlength-TNil*: *tlength* (*TNil b*) = *0*
and *tlength-TCons*: *tlength* (*TCons x xs*) = *eSuc* (*tlength xs*)
⟨*proof*⟩

lemma *llength-llist-of-tllist* [*simp*]: *llength* (*llist-of-tllist xs*) = *tlength xs*
⟨*proof*⟩

lemma *tlength-tmap* [*simp*]: *tlength* (*tmap f g xs*) = *tlength xs*
⟨*proof*⟩

definition *gen-tlength* :: *nat* ⇒ (′*a*, ′*b*) *tllist* ⇒ *enat*
where *gen-tlength n xs* = *enat n* + *tlength xs*

lemma *gen-tlength-code* [*code*]:
gen-tlength n (*TNil b*) = *enat n*
gen-tlength n (*TCons x xs*) = *gen-tlength* (*n* + *1*) *xs*
⟨*proof*⟩

lemma *tlength-code* [*code*]: *tlength* = *gen-tlength 0*
⟨*proof*⟩

5.17 *tdropn*

lemma *tdropn-0* [*simp*, *code*, *nitpick-simp*]: *tdropn 0 xs* = *xs*
⟨*proof*⟩

lemma *tdropn-TNil* [*simp*, *code*]: *tdropn n* (*TNil b*) = (*TNil b*)
⟨*proof*⟩

lemma *tdropn-Suc-TCons* [*simp*, *code*]: *tdropn* (*Suc n*) (*TCons x xs*) = *tdropn n xs*
⟨*proof*⟩

lemma *tdropn-Suc* [*nitpick-simp*]: *tdropn* (*Suc n*) *xs* = (*case xs of TNil b* ⇒ *TNil b* | *TCons x xs′* ⇒ *tdropn n xs′*)
⟨*proof*⟩

lemma *lappendt-ltake-tdropn*:
lappendt (*ltake* (*enat n*) (*llist-of-tllist xs*)) (*tdropn n xs*) = *xs*
⟨*proof*⟩

lemma *llist-of-tllist-tdropn* [*simp*]:
llist-of-tllist (*tdropn n xs*) = *ldropn n* (*llist-of-tllist xs*)
⟨*proof*⟩

lemma *tdropn-Suc-conv-tdropn*:
enat n < *tlength xs* ⟹ *TCons* (*tnth xs n*) (*tdropn* (*Suc n*) *xs*) = *tdropn n xs*
⟨*proof*⟩

lemma *tlength-tdropn* [*simp*]: *tlength* (*tdropn n xs*) = *tlength xs* − *enat n*
⟨*proof*⟩

lemma *tnth-tdropn* [*simp*]: *enat* (*n* + *m*) < *tlength xs* ⟹ *tnth* (*tdropn n xs*) *m* = *tnth xs* (*m* + *n*)
⟨*proof*⟩

5.18 *tset*

lemma *tset-induct* [*consumes 1*, *case-names find step*]:
assumes *x* ∈ *tset xs*
and ⋀*xs*. *P* (*TCons x xs*)
and ⋀*x′ xs*. ⟦ *x* ∈ *tset xs*; *x* ≠ *x′*; *P xs* ⟧ ⟹ *P* (*TCons x′ xs*)
shows *P xs*

⟨*proof*⟩

lemma *tset-conv-tnth*: *tset xs* = {*tnth xs n*|*n* . *enat n* < *tlength xs*}
⟨*proof*⟩

lemma *in-tset-conv-tnth*: *x* ∈ *tset xs* ⟷ (∃ *n*. *enat n* < *tlength xs* ∧ *tnth xs n* = *x*)
⟨*proof*⟩

5.19 Setup for Lifting/Transfer

5.19.1 Relator and predicator properties

abbreviation *tllist-all* == *pred-tllist*

5.19.2 Transfer rules for the Transfer package

context includes *lifting-syntax*
begin

lemma *set1-pre-tllist-transfer* [*transfer-rule*]:
(*rel-pre-tllist A B C* ===> *rel-set A*) *set1-pre-tllist set1-pre-tllist*
⟨*proof*⟩

lemma *set2-pre-tllist-transfer* [*transfer-rule*]:
(*rel-pre-tllist A B C* ===> *rel-set B*) *set2-pre-tllist set2-pre-tllist*
⟨*proof*⟩

lemma *set3-pre-tllist-transfer* [*transfer-rule*]:
(*rel-pre-tllist A B C* ===> *rel-set C*) *set3-pre-tllist set3-pre-tllist*
⟨*proof*⟩

lemma *TNil-transfer2* [*transfer-rule*]: (*B* ===> *tllist-all2 A B*) *TNil TNil*
⟨*proof*⟩
declare *TNil-transfer* [*transfer-rule*]

lemma *TCons-transfer2* [*transfer-rule*]:
(*A* ===> *tllist-all2 A B* ===> *tllist-all2 A B*) *TCons TCons*
⟨*proof*⟩
declare *TCons-transfer* [*transfer-rule*]

lemma *case-tllist-transfer* [*transfer-rule*]:
((*B* ===> *C*) ===> (*A* ===> *tllist-all2 A B* ===> *C*) ===> *tllist-all2 A B* ===> *C*)
case-tllist case-tllist
⟨*proof*⟩

lemma *unfold-tllist-transfer* [*transfer-rule*]:
((*A* ===> (=)) ===> (*A* ===> *B*) ===> (*A* ===> *C*) ===> (*A* ===> *A*) ===> *A* ===> *tllist-all2 C B*) *unfold-tllist unfold-tllist*

⟨*proof*⟩

lemma *corec-tllist-transfer* [*transfer-rule*]:
(((*A* ===> (=)) ===> (*A* ===> *B*) ===> (*A* ===> *C*) ===> (*A* ===> (=)) ===> (*A* ===> *tllist-all2 C B*) ===> (*A* ===> *A*) ===> *A* ===> *tllist-all2 C B*) *corec-tllist corec-tllist*
⟨*proof*⟩

lemma *ttl-transfer2* [*transfer-rule*]:
(*tllist-all2 A B* ===> *tllist-all2 A B*) *ttl ttl*
⟨*proof*⟩
declare *ttl-transfer* [*transfer-rule*]

lemma *tset-transfer2* [*transfer-rule*]:
(*tllist-all2 A B* ===> *rel-set A*) *tset tset*
⟨*proof*⟩

lemma *tmap-transfer2* [*transfer-rule*]:
((*A* ===> *B*) ===> (*C* ===> *D*) ===> *tllist-all2 A C* ===> *tllist-all2 B D*) *tmap tmap*
⟨*proof*⟩
declare *tmap-transfer* [*transfer-rule*]

lemma *is-TNil-transfer2* [*transfer-rule*]:
(*tllist-all2 A B* ===> (=)) *is-TNil is-TNil*
⟨*proof*⟩
declare *is-TNil-transfer* [*transfer-rule*]

lemma *tappend-transfer* [*transfer-rule*]:
(*tllist-all2 A B* ===> (*B* ===> *tllist-all2 A C*) ===> *tllist-all2 A C*) *tappend tappend*
⟨*proof*⟩
declare *tappend.transfer* [*transfer-rule*]

lemma *lappendt-transfer* [*transfer-rule*]:
(*llist-all2 A* ===> *tllist-all2 A B* ===> *tllist-all2 A B*) *lappendt lappendt*
⟨*proof*⟩
declare *lappendt.transfer* [*transfer-rule*]

lemma *llist-of-tllist-transfer2* [*transfer-rule*]:
(*tllist-all2 A B* ===> *llist-all2 A*) *llist-of-tllist llist-of-tllist*
⟨*proof*⟩
declare *llist-of-tllist-transfer* [*transfer-rule*]

lemma *tllist-of-llist-transfer2* [*transfer-rule*]:
(*B* ===> *llist-all2 A* ===> *tllist-all2 A B*) *tllist-of-llist tllist-of-llist*
⟨*proof*⟩
declare *tllist-of-llist-transfer* [*transfer-rule*]

lemma *tlength-transfer* [*transfer-rule*]:
(*tllist-all2 A B* ===> (=)) *tlength tlength*
⟨*proof*⟩
declare *tlength.transfer* [*transfer-rule*]

lemma *tdropn-transfer* [*transfer-rule*]:
((=) ===> *tllist-all2 A B* ===> *tllist-all2 A B*) *tdropn tdropn*
⟨*proof*⟩
declare *tdropn.transfer* [*transfer-rule*]

lemma *tfilter-transfer* [*transfer-rule*]:
(*B* ===> (*A* ===> (=)) ===> *tllist-all2 A B* ===> *tllist-all2 A B*) *tfilter tfilter*
⟨*proof*⟩
declare *tfilter.transfer* [*transfer-rule*]

lemma *tconcat-transfer* [*transfer-rule*]:
(*B* ===> *tllist-all2* (*llist-all2 A*) *B* ===> *tllist-all2 A B*) *tconcat tconcat*
⟨*proof*⟩
declare *tconcat.transfer* [*transfer-rule*]

lemma *tllist-all2-rsp*:
assumes *R1*: ∀ *x y*. *R1 x y* ⟶ (∀ *a b*. *R1 a b* ⟶ *S x a* = *T y b*)
and *R2*: ∀ *x y*. *R2 x y* ⟶ (∀ *a b*. *R2 a b* ⟶ *S′ x a* = *T′ y b*)
and *xsys*: *tllist-all2 R1 R2 xs ys*
and *xs′ys′*: *tllist-all2 R1 R2 xs′ ys′*
shows *tllist-all2 S S′ xs xs′* = *tllist-all2 T T′ ys ys′*
⟨*proof*⟩

lemma *tllist-all2-transfer2* [*transfer-rule*]:
((*R1* ===> *R1* ===> (=)) ===> (*R2* ===> *R2* ===> (=)) ===>
tllist-all2 R1 R2 ===> *tllist-all2 R1 R2* ===> (=)) *tllist-all2 tllist-all2*
⟨*proof*⟩
declare *tllist-all2-transfer* [*transfer-rule*]

end

Delete lifting rules for ('*a*, '*b*) *tllist* because the parametricity rules take precedence over most of the transfer rules. They can be restored by including the bundle *tllist.lifting*.

lifting-update *tllist.lifting*
lifting-forget *tllist.lifting*

end

6 Setup for Isabelle's quotient package for lazy lists

theory *Quotient-Coinductive-List* **imports**
HOL−Library.Quotient-List
HOL−Library.Quotient-Set
Coinductive-List
begin

6.1 Rules for the Quotient package

declare *llist.rel-eq*[*id-simps*]

lemma *transpD*: ⟦ *transp R*; *R a b*; *R b c* ⟧ ⟹ *R a c*
⟨*proof*⟩

lemma *id-respect* [*quot-respect*]:
(*R* ===> *R*) *id id*
⟨*proof*⟩

lemma *id-preserve* [*quot-preserve*]:
assumes *Quotient3 R Abs Rep*
shows (*Rep* −−−> *Abs*) *id* = *id*
⟨*proof*⟩

functor *lmap*: *lmap*
⟨*proof*⟩

declare *llist.map-id0* [*id-simps*]

lemma *reflp-llist-all2*: *reflp R* ⟹ *reflp* (*llist-all2 R*)
⟨*proof*⟩

lemma *symp-llist-all2*: *symp R* ⟹ *symp* (*llist-all2 R*)
⟨*proof*⟩

lemma *transp-llist-all2*: *transp R* ⟹ *transp* (*llist-all2 R*)
⟨*proof*⟩

lemma *llist-equivp* [*quot-equiv*]:
equivp R ⟹ *equivp* (*llist-all2 R*)
⟨*proof*⟩

lemma *unfold-llist-preserve* [*quot-preserve*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
and *q2*: *Quotient3 R2 Abs2 Rep2*
shows ((*Abs1* −−−> *id*) −−−> (*Abs1* −−−> *Rep2*) −−−> (*Abs1* −−−> *Rep1*) −−−> *Rep1* −−−> *lmap Abs2*) *unfold-llist* = *unfold-llist*
(**is** *?lhs* = *?rhs*)

⟨*proof*⟩

lemma *Quotient-lmap-Abs-Rep*:
  Quotient3 R Abs Rep ⟹ *lmap Abs* (*lmap Rep a*) = *a*
  ⟨*proof*⟩

lemma *llist-all2-rel*:
  assumes *Quotient3 R Abs Rep*
  shows *llist-all2 R r s* ⟷ *llist-all2 R r r* ∧ *llist-all2 R s s* ∧ (*lmap Abs r* = *lmap Abs s*)
  (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *Quotient-llist-all2-lmap-Rep*:
  Quotient3 R Abs Rep ⟹ *llist-all2 R* (*lmap Rep a*) (*lmap Rep a*)
⟨*proof*⟩

lemma *llist-quotient* [*quot-thm*]:
  Quotient3 R Abs Rep ⟹ *Quotient3* (*llist-all2 R*) (*lmap Abs*) (*lmap Rep*)
⟨*proof*⟩

declare [[*mapQ3 llist* = (*llist-all2*, *llist-quotient*)]]

lemma *LCons-preserve* [*quot-preserve*]:
  assumes *Quotient3 R Abs Rep*
  shows (*Rep* −−−> (*lmap Rep*) −−−> (*lmap Abs*)) *LCons* = *LCons*
⟨*proof*⟩

lemmas *LCons-respect* [*quot-respect*] = *LCons-transfer*

lemma *LNil-preserve* [*quot-preserve*]:
  lmap Abs LNil = *LNil*
⟨*proof*⟩

lemmas *LNil-respect* [*quot-respect*] = *LNil-transfer*

lemma *lmap-preserve* [*quot-preserve*]:
  assumes *a*: *Quotient3 R1 abs1 rep1*
  and *b*: *Quotient3 R2 abs2 rep2*
  shows ((*abs1* −−−> *rep2*) −−−> (*lmap rep1*) −−−> (*lmap abs2*)) *lmap* = *lmap*
  and ((*abs1* −−−> *id*) −−−> *lmap rep1* −−−> *id*) *lmap* = *lmap*
⟨*proof*⟩

lemma *lmap-respect* [*quot-respect*]:
  shows ((*R1* ===> *R2*) ===> (*llist-all2 R1*) ===> *llist-all2 R2*) *lmap lmap*
  and ((*R1* ===> (=)) ===> (*llist-all2 R1*) ===> (=)) *lmap lmap*
⟨*proof*⟩

lemmas *llist-all2-respect* [*quot-respect*] = *llist-all2-transfer*

lemma *llist-all2-preserve* [*quot-preserve*]:
assumes *Quotient3 R Abs Rep*
shows ((*Abs* −−−> *Abs* −−−> *id*) −−−> *lmap Rep* −−−> *lmap Rep* −−−> *id*) *llist-all2* = *llist-all2*
⟨*proof*⟩

lemma *llist-all2-preserve2* [*quot-preserve*]:
assumes *Quotient3 R Abs Rep*
shows (*llist-all2* ((*Rep* −−−> *Rep* −−−> *id*) *R*) *l m*) = (*l* = *m*)
⟨*proof*⟩

lemma *corec-llist-preserve* [*quot-preserve*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
and *q2*: *Quotient3 R2 Abs2 Rep2*
shows ((*Abs1* −−−> *id*) −−−> (*Abs1* −−−> *Rep2*) −−−> (*Abs1* −−−> *id*) −−−>
(*Abs1* −−−> *lmap Rep2*) −−−> (*Abs1* −−−> *Rep1*) −−−> *Rep1* −−−> *lmap Abs2*) *corec-llist* = *corec-llist*
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

end

7 Setup for Isabelle's quotient package for terminated lazy lists

theory *Quotient-TLList* **imports**
TLList
HOL−Library.Quotient-Product
HOL−Library.Quotient-Sum
HOL−Library.Quotient-Set
begin

7.1 Rules for the Quotient package

lemma *tmap-id-id* [*id-simps*]:
tmap id id = *id*
⟨*proof*⟩

declare *tllist-all2-eq*[*id-simps*]

lemma *case-sum-preserve* [*quot-preserve*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
and *q2*: *Quotient3 R2 Abs2 Rep2*
and *q3*: *Quotient3 R3 Abs3 Rep3*
shows ((*Abs1* −−−> *Rep2*) −−−> (*Abs3* −−−> *Rep2*) −−−> *map-sum Rep1*

Rep3 −−−> *Abs2*) *case-sum* = *case-sum*
⟨*proof*⟩

lemma *case-sum-preserve2* [*quot-preserve*]:
assumes *q*: *Quotient3 R Abs Rep*
shows ((*id* −−−> *Rep*) −−−> (*id* −−−> *Rep*) −−−> *id* −−−> *Abs*) *case-sum* = *case-sum*
⟨*proof*⟩

lemma *case-prod-preserve* [*quot-preserve*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
and *q2*: *Quotient3 R2 Abs2 Rep2*
and *q3*: *Quotient3 R3 Abs3 Rep3*
shows ((*Abs1* −−−> *Abs2* −−−> *Rep3*) −−−> *map-prod Rep1 Rep2* −−−> *Abs3*) *case-prod* = *case-prod*
⟨*proof*⟩

lemma *case-prod-preserve2* [*quot-preserve*]:
assumes *q*: *Quotient3 R Abs Rep*
shows ((*id* −−−> *id* −−−> *Rep*) −−−> *id* −−−> *Abs*) *case-prod* = *case-prod*
⟨*proof*⟩

lemma *id-preserve* [*quot-preserve*]:
assumes *Quotient3 R Abs Rep*
shows (*Rep* −−−> *Abs*) *id* = *id*
⟨*proof*⟩

functor *tmap*: *tmap*
⟨*proof*⟩

lemma *reflp-tllist-all2*:
assumes *R*: *reflp R* **and** *Q*: *reflp Q*
shows *reflp* (*tllist-all2 R Q*)
⟨*proof*⟩

lemma *symp-tllist-all2*: ⟦ *symp R*; *symp S* ⟧ ⟹ *symp* (*tllist-all2 R S*)
⟨*proof*⟩

lemma *transp-tllist-all2*: ⟦ *transp R*; *transp S* ⟧ ⟹ *transp* (*tllist-all2 R S*)
⟨*proof*⟩

lemma *tllist-equivp* [*quot-equiv*]:
⟦ *equivp R*; *equivp S* ⟧ ⟹ *equivp* (*tllist-all2 R S*)
⟨*proof*⟩

declare *tllist-all2-eq* [*simp*, *id-simps*]

lemma *tmap-preserve* [*quot-preserve*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*

and *q2*: *Quotient3 R2 Abs2 Rep2*
and *q3*: *Quotient3 R3 Abs3 Rep3*
and *q4*: *Quotient3 R4 Abs4 Rep4*
shows ((*Abs1* −−−> *Rep2*) −−−> (*Abs3* −−−> *Rep4*) −−−> *tmap Rep1 Rep3* −−−> *tmap Abs2 Abs4*) *tmap* = *tmap*
and ((*Abs1* −−−> *id*) −−−> (*Abs2* −−−> *id*) −−−> *tmap Rep1 Rep2* −−−> *id*) *tmap* = *tmap*
⟨*proof*⟩

lemmas *tmap-respect* [*quot-respect*] = *tmap-transfer2*

lemma *Quotient3-tmap-Abs-Rep*:
⟦*Quotient3 R1 Abs1 Rep1*; *Quotient3 R2 Abs2 Rep2*⟧
⟹ *tmap Abs1 Abs2* (*tmap Rep1 Rep2 ts*) = *ts*
⟨*proof*⟩

lemma *Quotient3-tllist-all2-tmap-tmapI*:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
and *q2*: *Quotient3 R2 Abs2 Rep2*
shows *tllist-all2 R1 R2* (*tmap Rep1 Rep2 ts*) (*tmap Rep1 Rep2 ts*)
⟨*proof*⟩

lemma *tllist-all2-rel*:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
and *q2*: *Quotient3 R2 Abs2 Rep2*
shows *tllist-all2 R1 R2 r s* ⟷ (*tllist-all2 R1 R2 r r* ∧ *tllist-all2 R1 R2 s s* ∧ *tmap Abs1 Abs2 r* = *tmap Abs1 Abs2 s*)
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *tllist-quotient* [*quot-thm*]:
⟦ *Quotient3 R1 Abs1 Rep1*; *Quotient3 R2 Abs2 Rep2* ⟧
⟹ *Quotient3* (*tllist-all2 R1 R2*) (*tmap Abs1 Abs2*) (*tmap Rep1 Rep2*)
⟨*proof*⟩

declare [[*mapQ3 tllist* = (*tllist-all2*, *tllist-quotient*)]]

lemma *TCons-preserve* [*quot-preserve*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
and *q2*: *Quotient3 R2 Abs2 Rep2*
shows (*Rep1* −−−> (*tmap Rep1 Rep2*) −−−> (*tmap Abs1 Abs2*)) *TCons* = *TCons*
⟨*proof*⟩

lemmas *TCons-respect* [*quot-respect*] = *TCons-transfer2*

lemma *TNil-preserve* [*quot-preserve*]:
assumes *Quotient3 R2 Abs2 Rep2*
shows (*Rep2* −−−> *tmap Abs1 Abs2*) *TNil* = *TNil*

⟨*proof*⟩

lemmas *TNil-respect* [*quot-respect*] = *TNil-transfer2*

lemmas *tllist-all2-respect* [*quot-respect*] = *tllist-all2-transfer*

lemma *tllist-all2-prs*:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
and *q2*: *Quotient3 R2 Abs2 Rep2*
shows *tllist-all2* ((*Abs1* −−−> *Abs1* −−−> *id*) *P*) ((*Abs2* −−−> *Abs2* −−−> *id*) *Q*)
(*tmap Rep1 Rep2 ts*) (*tmap Rep1 Rep2 ts′*)
⟷ *tllist-all2 P Q ts ts′*
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *tllist-all2-preserve* [*quot-preserve*]:
assumes *Quotient3 R1 Abs1 Rep1*
and *Quotient3 R2 Abs2 Rep2*
shows ((*Abs1* −−−> *Abs1* −−−> *id*) −−−> (*Abs2* −−−> *Abs2* −−−> *id*) −−−>
tmap Rep1 Rep2 −−−> *tmap Rep1 Rep2* −−−> *id*) *tllist-all2* = *tllist-all2*
⟨*proof*⟩

lemma *tllist-all2-preserve2* [*quot-preserve*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
and *q2*: *Quotient3 R2 Abs2 Rep2*
shows (*tllist-all2* ((*Rep1* −−−> *Rep1* −−−> *id*) *R1*) ((*Rep2* −−−> *Rep2* −−−> *id*) *R2*)) = (=)
⟨*proof*⟩

lemma *corec-tllist-preserve* [*quot-preserve*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
and *q2*: *Quotient3 R2 Abs2 Rep2*
and *q3*: *Quotient3 R3 Abs3 Rep3*
shows ((*Abs1* −−−> *id*) −−−> (*Abs1* −−−> *Rep2*) −−−> (*Abs1* −−−> *Rep3*) −−−> (*Abs1* −−−> *id*) −−−> (*Abs1* −−−> *tmap Rep3 Rep2*) −−−> (*Abs1* −−−> *Rep1*) −−−> *Rep1* −−−> *tmap Abs3 Abs2*) *corec-tllist* = *corec-tllist*
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

end

theory *Coinductive* **imports**
Coinductive-List-Prefix
Coinductive-Stream
TLList
Quotient-Coinductive-List
Quotient-TLList

begin

end

8 Code generator setup to implement lazy lists lazily

theory *Lazy-LList* **imports**
Coinductive-List
begin

8.1 Lazy lists

code-identifier code-module *Lazy-LList* ⇀
(*SML*) *Coinductive-List* **and**
(*OCaml*) *Coinductive-List* **and**
(*Haskell*) *Coinductive-List* **and**
(*Scala*) *Coinductive-List*

definition *Lazy-llist* :: (*unit* ⇒ ('*a* × '*a llist*) *option*) ⇒ '*a llist*
where [*simp*]:
Lazy-llist xs = (*case xs* () *of None* ⇒ *LNil* | *Some* (*x*, *ys*) ⇒ *LCons x ys*)

definition *force* :: '*a llist* ⇒ ('*a* × '*a llist*) *option*
where [*simp*, *code del*]: *force xs* = (*case xs of LNil* ⇒ *None* | *LCons x ys* ⇒ *Some* (*x*, *ys*))

code-datatype *Lazy-llist*

declare — Restore consistency in code equations between *partial-term-of* and *narrowing* for '*a llist*
[[*code drop*: *partial-term-of* :: - *llist itself* => -]]

lemma *partial-term-of-llist-code* [*code*]:
fixes *tytok* :: '*a* :: *partial-term-of llist itself* **shows**
partial-term-of tytok (*Quickcheck-Narrowing.Narrowing-variable p tt*) ≡
Code-Evaluation.Free (*STR* ''-'') (*Typerep.typerep TYPE*('*a llist*))
partial-term-of tytok (*Quickcheck-Narrowing.Narrowing-constructor 0* []) ≡
Code-Evaluation.Const (*STR* ''*Coinductive-List.llist.LNil*'') (*Typerep.typerep TYPE*('*a llist*))
partial-term-of tytok (*Quickcheck-Narrowing.Narrowing-constructor 1* [*head*, *tail*]) ≡
Code-Evaluation.App
(*Code-Evaluation.App*
(*Code-Evaluation.Const*
(*STR* ''*Coinductive-List.llist.LCons*'')
(*Typerep.typerep TYPE*('*a* ⇒ '*a llist* ⇒ '*a llist*)))
(*partial-term-of TYPE*('*a*) *head*))

(*partial-term-of TYPE('a llist) tail*)
⟨*proof*⟩

declare *option.splits* [*split*]

lemma *Lazy-llist-inject* [*simp*]:
Lazy-llist xs = *Lazy-llist ys* ⟷ *xs* = *ys*
⟨*proof*⟩

lemma *Lazy-llist-inverse* [*code*, *simp*]:
force (*Lazy-llist xs*) = *xs* ()
⟨*proof*⟩

lemma *force-inverse* [*simp*]:
Lazy-llist (λ-. *force xs*) = *xs*
⟨*proof*⟩

lemma *LNil-Lazy-llist* [*code*]: *LNil* = *Lazy-llist* (λ-. *None*)
⟨*proof*⟩

lemma *LCons-Lazy-llist* [*code*, *code-unfold*]: *LCons x xs* = *Lazy-llist* (λ-. *Some* (*x*, *xs*))
⟨*proof*⟩

lemma *lnull-lazy* [*code*]: *lnull* = *Option.is-none* ∘ *force*
⟨*proof*⟩

declare [[*code drop*: *equal-class.equal* :: 'a :: *equal llist* ⇒ -]]

lemma *equal-llist-Lazy-llist* [*code*]:
equal-class.equal (*Lazy-llist xs*) (*Lazy-llist ys*) ⟷
(*case xs* () *of None* ⇒ (*case ys* () *of None* ⇒ *True* | - ⇒ *False*)
| *Some* (*x*, *xs'*) ⇒
(*case ys* () *of None* ⇒ *False*
| *Some* (*y*, *ys'*) ⇒ *if x* = *y then equal-class.equal xs' ys' else False*))
⟨*proof*⟩

declare [[*code drop*: *corec-llist*]]

lemma *corec-llist-Lazy-llist* [*code*]:
corec-llist IS-LNIL LHD endORmore LTL-end LTL-more b =
Lazy-llist (λ-. *if IS-LNIL b then None*
else Some (*LHD b*,
if endORmore b then LTL-end b
else corec-llist IS-LNIL LHD endORmore LTL-end LTL-more (*LTL-more b*)))
⟨*proof*⟩

declare [[*code drop*: *unfold-llist*]]

lemma *unfold-llist-Lazy-llist* [*code*]:
unfold-llist IS-LNIL LHD LTL b =
Lazy-llist (λ-. *if IS-LNIL b then None else Some* (*LHD b*, *unfold-llist IS-LNIL LHD LTL* (*LTL b*)))
⟨*proof*⟩

declare [[*code drop*: *case-llist*]]

lemma *case-llist-Lazy-llist* [*code*]:
case-llist n c (*Lazy-llist xs*) = (*case xs* () *of None* ⇒ *n* | *Some* (*x*, *ys*) ⇒ *c x ys*)
⟨*proof*⟩

declare [[*code drop*: *lappend*]]

lemma *lappend-Lazy-llist* [*code*]:
lappend (*Lazy-llist xs*) *ys* =
Lazy-llist (λ-. *case xs* () *of None* ⇒ *force ys* | *Some* (*x*, *xs*′) ⇒ *Some* (*x*, *lappend xs*′ *ys*))
⟨*proof*⟩

declare [[*code drop*: *lmap*]]

lemma *lmap-Lazy-llist* [*code*]:
lmap f (*Lazy-llist xs*) = *Lazy-llist* (λ-. *map-option* (*map-prod f* (*lmap f*)) (*xs* ()))
⟨*proof*⟩

declare [[*code drop*: *lfinite*]]

lemma *lfinite-Lazy-llist* [*code*]:
lfinite (*Lazy-llist xs*) = (*case xs* () *of None* ⇒ *True* | *Some* (*x*, *ys*) ⇒ *lfinite ys*)
⟨*proof*⟩

declare [[*code drop*: *list-of-aux*]]

lemma *list-of-aux-Lazy-llist* [*code*]:
list-of-aux xs (*Lazy-llist ys*) =
(*case ys* () *of None* ⇒ *rev xs* | *Some* (*y*, *ys*) ⇒ *list-of-aux* (*y* # *xs*) *ys*)
⟨*proof*⟩

declare [[*code drop*: *gen-llength*]]

lemma *gen-llength-Lazy-llist* [*code*]:
gen-llength n (*Lazy-llist xs*) = (*case xs* () *of None* ⇒ *enat n* | *Some* (-, *ys*) ⇒ *gen-llength* (*n* + *1*) *ys*)
⟨*proof*⟩

declare [[*code drop*: *ltake*]]

lemma *ltake-Lazy-llist* [*code*]:

ltake n (Lazy-llist xs) =
Lazy-llist (λ-. if n = 0 then None else case xs () of None ⇒ None | Some (x, ys) ⇒ Some (x, ltake (n − 1) ys))
⟨*proof*⟩

declare [[*code drop*: *ldropn*]]

lemma *ldropn-Lazy-llist* [*code*]:
ldropn n (Lazy-llist xs) =
Lazy-llist (λ-. if n = 0 then xs () else
case xs () of None ⇒ None | Some (x, ys) ⇒ force (ldropn (n − 1) ys))
⟨*proof*⟩

declare [[*code drop*: *ltakeWhile*]]

lemma *ltakeWhile-Lazy-llist* [*code*]:
ltakeWhile P (Lazy-llist xs) =
Lazy-llist (λ-. case xs () of None ⇒ None | Some (x, ys) ⇒ if P x then Some (x, ltakeWhile P ys) else None)
⟨*proof*⟩

declare [[*code drop*: *ldropWhile*]]

lemma *ldropWhile-Lazy-llist* [*code*]:
ldropWhile P (Lazy-llist xs) =
Lazy-llist (λ-. case xs () of None ⇒ None | Some (x, ys) ⇒ if P x then force (ldropWhile P ys) else Some (x, ys))
⟨*proof*⟩

declare [[*code drop*: *lzip*]]

lemma *lzip-Lazy-llist* [*code*]:
lzip (Lazy-llist xs) (Lazy-llist ys) =
Lazy-llist (λ-. Option.bind (xs ()) (λ(x, xs′). map-option (λ(y, ys′). ((x, y), lzip xs′ ys′)) (ys ())))
⟨*proof*⟩

declare [[*code drop*: *gen-lset*]]

lemma *lset-Lazy-llist* [*code*]:
gen-lset A (Lazy-llist xs) =
(case xs () of None ⇒ A | Some (y, ys) ⇒ gen-lset (insert y A) ys)
⟨*proof*⟩

declare [[*code drop*: *lmember*]]

lemma *lmember-Lazy-llist* [*code*]:
lmember x (Lazy-llist xs) =

(*case xs () of None ⇒ False | Some (y, ys) ⇒ x = y ∨ lmember x ys*)
⟨*proof*⟩

declare [[*code drop*: *llist-all2*]]

lemma *llist-all2-Lazy-llist* [*code*]:
llist-all2 P (*Lazy-llist xs*) (*Lazy-llist ys*) =
(*case xs () of None ⇒ ys () = None*
| *Some* (*x*, *xs'*) ⇒ (*case ys () of None ⇒ False*
| *Some* (*y*, *ys'*) ⇒ *P x y* ∧ *llist-all2 P xs' ys'*))
⟨*proof*⟩

declare [[*code drop*: *lhd*]]

lemma *lhd-Lazy-llist* [*code*]:
lhd (*Lazy-llist xs*) = (*case xs () of None ⇒ undefined | Some* (*x*, *xs'*) ⇒ *x*)
⟨*proof*⟩

declare [[*code drop*: *ltl*]]

lemma *ltl-Lazy-llist* [*code*]:
ltl (*Lazy-llist xs*) = *Lazy-llist* (λ-. *case xs () of None ⇒ None | Some* (*x*, *ys*) ⇒ *force ys*)
⟨*proof*⟩

declare [[*code drop*: *llast*]]

lemma *llast-Lazy-llist* [*code*]:
llast (*Lazy-llist xs*) =
(*case xs () of*
None ⇒ undefined
| *Some* (*x*, *xs'*) ⇒
(*case force xs' of None ⇒ x | Some* (*x'*, *xs''*) ⇒ *llast* (*LCons x' xs''*)))
⟨*proof*⟩

declare [[*code drop*: *ldistinct*]]

lemma *ldistinct-Lazy-llist* [*code*]:
ldistinct (*Lazy-llist xs*) =
(*case xs () of None ⇒ True | Some* (*x*, *ys*) ⇒ *x* ∉ *lset ys* ∧ *ldistinct ys*)
⟨*proof*⟩

declare [[*code drop*: *lprefix*]]

lemma *lprefix-Lazy-llist* [*code*]:
lprefix (*Lazy-llist xs*) (*Lazy-llist ys*) =
(*case xs () of*
None ⇒ True
| *Some* (*x*, *xs'*) ⇒

(*case ys () of None ⇒ False | Some (y, ys′) ⇒ x = y ∧ lprefix xs′ ys′*))
⟨*proof*⟩

declare [[*code drop*: *lstrict-prefix*]]

lemma *lstrict-prefix-Lazy-llist* [*code*]:
lstrict-prefix (*Lazy-llist xs*) (*Lazy-llist ys*) ⟷
(*case ys () of*
None ⇒ False
| *Some* (*y*, *ys′*) ⇒
(*case xs () of None ⇒ True | Some* (*x*, *xs′*) ⇒ *x* = *y* ∧ *lstrict-prefix xs′ ys′*))
⟨*proof*⟩

declare [[*code drop*: *llcp*]]

lemma *llcp-Lazy-llist* [*code*]:
llcp (*Lazy-llist xs*) (*Lazy-llist ys*) =
(*case xs () of None ⇒ 0*
| *Some* (*x*, *xs′*) ⇒ (*case ys () of None ⇒ 0*
| *Some* (*y*, *ys′*) ⇒ *if x* = *y then eSuc* (*llcp xs′ ys′*) *else 0*))
⟨*proof*⟩

declare [[*code drop*: *llexord*]]

lemma *llexord-Lazy-llist* [*code*]:
llexord r (*Lazy-llist xs*) (*Lazy-llist ys*) ⟷
(*case xs () of*
None ⇒ True
| *Some* (*x*, *xs′*) ⇒
(*case ys () of None ⇒ False | Some* (*y*, *ys′*) ⇒ *r x y* ∨ *x* = *y* ∧ *llexord r xs′ ys′*))
⟨*proof*⟩

declare [[*code drop*: *lfilter*]]

lemma *lfilter-Lazy-llist* [*code*]:
lfilter P (*Lazy-llist xs*) =
Lazy-llist (λ-. *case xs () of None ⇒ None*
| *Some* (*x*, *ys*) ⇒ *if P x then Some* (*x*, *lfilter P ys*) *else force* (*lfilter P ys*))
⟨*proof*⟩

declare [[*code drop*: *lconcat*]]

lemma *lconcat-Lazy-llist* [*code*]:
lconcat (*Lazy-llist xss*) =
Lazy-llist (λ-. *case xss () of None ⇒ None | Some* (*xs*, *xss′*) ⇒ *force* (*lappend xs* (*lconcat xss′*)))
⟨*proof*⟩

declare *option.splits* [*split del*]
declare *Lazy-llist-def* [*simp del*]

Simple ML test for laziness

⟨*ML*⟩

hide-const (**open**) *force*

end

9 Code generator setup to implement terminated lazy lists lazily

theory *Lazy-TLList* **imports**
TLList
Lazy-LList
begin

code-identifier code-module *Lazy-TLList* ⇀
(*SML*) *TLList* **and**
(*OCaml*) *TLList* **and**
(*Haskell*) *TLList* **and**
(*Scala*) *TLList*

definition *Lazy-tllist* :: (*unit* ⇒ *'a* × (*'a*, *'b*) *tllist* + *'b*) ⇒ (*'a*, *'b*) *tllist*
where [*code del*]:
Lazy-tllist xs = (*case xs* () *of Inl* (*x*, *ys*) ⇒ *TCons x ys* | *Inr b* ⇒ *TNil b*)

definition *force* :: (*'a*, *'b*) *tllist* ⇒ *'a* × (*'a*, *'b*) *tllist* + *'b*
where [*simp*, *code del*]: *force xs* = (*case xs of TNil b* ⇒ *Inr b* | *TCons x ys* ⇒ *Inl* (*x*, *ys*))

code-datatype *Lazy-tllist*

declare — Restore consistency in code equations between *partial-term-of* and *narrowing* for (*'a*, *'b*) *tllist*
[[*code drop*: *partial-term-of* :: (-, -) *tllist itself* => -]]

lemma *partial-term-of-tllist-code* [*code*]:
fixes *tytok* :: (*'a* :: *partial-term-of*, *'b* :: *partial-term-of*) *tllist itself* **shows**
partial-term-of tytok (*Quickcheck-Narrowing.Narrowing-variable p tt*) ≡
Code-Evaluation.Free (*STR ''-''*) (*Typerep.typerep TYPE*((*'a*, *'b*) *tllist*))
partial-term-of tytok (*Quickcheck-Narrowing.Narrowing-constructor 0* [*b*]) ≡
Code-Evaluation.App
(*Code-Evaluation.Const* (*STR ''TLList.tllist.TNil''*) (*Typerep.typerep TYPE*(*'b* ⇒ (*'a*, *'b*) *tllist*)))
(*partial-term-of TYPE*(*'b*) *b*)

partial-term-of tytok (*Quickcheck-Narrowing.Narrowing-constructor 1* [*head*, *tail*])
≡
Code-Evaluation.App
(*Code-Evaluation.App*
(*Code-Evaluation.Const*
(*STR ''TLList.tllist.TCons''*)
(*Typerep.typerep TYPE*('*a* ⇒ ('*a*, '*b*) *tllist* ⇒ ('*a*, '*b*) *tllist*)))
(*partial-term-of TYPE*('*a*) *head*))
(*partial-term-of TYPE*(('*a*, '*b*) *tllist*) *tail*)
⟨*proof*⟩

declare *Lazy-tllist-def* [*simp*]
declare *sum.splits* [*split*]

lemma *TNil-Lazy-tllist* [*code*]:
TNil b = *Lazy-tllist* (λ-. *Inr b*)
⟨*proof*⟩

lemma *TCons-Lazy-tllist* [*code*, *code-unfold*]:
TCons x xs = *Lazy-tllist* (λ-. *Inl* (*x*, *xs*))
⟨*proof*⟩

lemma *Lazy-tllist-inverse* [*simp*, *code*]:
force (*Lazy-tllist xs*) = *xs* ()
⟨*proof*⟩

declare [[*code drop*: *equal-class.equal* :: (-, -) *tllist* ⇒ -]]

lemma *equal-tllist-Lazy-tllist* [*code*]:
equal-class.equal (*Lazy-tllist xs*) (*Lazy-tllist ys*) =
(*case xs* () *of*
Inr b ⇒ (*case ys* () *of Inr b'* ⇒ *b* = *b'* | - ⇒ *False*)
| *Inl* (*x*, *xs'*) ⇒
(*case ys* () *of Inr b'* ⇒ *False* | *Inl* (*y*, *ys'*) ⇒ *if x* = *y then equal-class.equal xs' ys' else False*))
⟨*proof*⟩

declare
[[*code drop*: *thd ttl*]]
thd-def [*code*]
ttl-def [*code*]

declare [[*code drop*: *is-TNil*]]

lemma *is-TNil-code* [*code*]:
is-TNil (*Lazy-tllist xs*) ⟷
(*case xs* () *of Inl* - ⇒ *False* | *Inr* - ⇒ *True*)
⟨*proof*⟩

declare [[*code drop*: *corec-tllist*]]

lemma *corec-tllist-Lazy-tllist* [*code*]:
corec-tllist IS-TNIL TNIL THD endORmore TTL-end TTL-more b = *Lazy-tllist*
(λ-. *if IS-TNIL b then Inr* (*TNIL b*)
else Inl (*THD b*, *if endORmore b then TTL-end b else corec-tllist IS-TNIL TNIL THD endORmore TTL-end TTL-more* (*TTL-more b*)))
⟨*proof*⟩

declare [[*code drop*: *unfold-tllist*]]

lemma *unfold-tllist-Lazy-tllist* [*code*]:
unfold-tllist IS-TNIL TNIL THD TTL b = *Lazy-tllist*
(λ-. *if IS-TNIL b then Inr* (*TNIL b*)
else Inl (*THD b*, *unfold-tllist IS-TNIL TNIL THD TTL* (*TTL b*)))
⟨*proof*⟩

declare [[*code drop*: *case-tllist*]]

lemma *case-tllist-Lazy-tllist* [*code*]:
case-tllist n c (*Lazy-tllist xs*) =
(*case xs* () *of Inl* (*x*, *ys*) ⇒ *c x ys* | *Inr b* ⇒ *n b*)
⟨*proof*⟩

declare [[*code drop*: *tllist-of-llist*]]

lemma *tllist-of-llist-Lazy-llist* [*code*]:
tllist-of-llist b (*Lazy-llist xs*) =
Lazy-tllist (λ-. *case xs* () *of None* ⇒ *Inr b* | *Some* (*x*, *ys*) ⇒ *Inl* (*x*, *tllist-of-llist b ys*))
⟨*proof*⟩

declare [[*code drop*: *terminal*]]

lemma *terminal-Lazy-tllist* [*code*]:
terminal (*Lazy-tllist xs*) =
(*case xs* () *of Inl* (-, *ys*) ⇒ *terminal ys* | *Inr b* ⇒ *b*)
⟨*proof*⟩

declare [[*code drop*: *tmap*]]

lemma *tmap-Lazy-tllist* [*code*]:
tmap f g (*Lazy-tllist xs*) =
Lazy-tllist (λ-. *case xs* () *of Inl* (*x*, *ys*) ⇒ *Inl* (*f x*, *tmap f g ys*) | *Inr b* ⇒ *Inr* (*g b*))
⟨*proof*⟩

declare [[*code drop*: *tappend*]]

lemma *tappend-Lazy-tllist* [*code*]:
tappend (*Lazy-tllist xs*) *ys* =
Lazy-tllist (λ-. *case xs* () *of Inl* (*x*, *xs'*) ⇒ *Inl* (*x*, *tappend xs' ys*) | *Inr b* ⇒ *force* (*ys b*))
⟨*proof*⟩

declare [[*code drop*: *lappendt*]]

lemma *lappendt-Lazy-llist* [*code*]:
lappendt (*Lazy-llist xs*) *ys* =
Lazy-tllist (λ-. *case xs* () *of None* ⇒ *force ys* | *Some* (*x*, *xs'*) ⇒ *Inl* (*x*, *lappendt xs' ys*))
⟨*proof*⟩

declare [[*code drop*: *TLList.tfilter'*]]

lemma *tfilter'-Lazy-tllist* [*code*]:
TLList.tfilter' b P (*Lazy-tllist xs*) =
Lazy-tllist (λ-. *case xs* () *of Inl* (*x*, *xs'*) ⇒ *if P x then Inl* (*x*, *TLList.tfilter' b P xs'*) *else force* (*TLList.tfilter' b P xs'*) | *Inr b'* ⇒ *Inr b'*)
⟨*proof*⟩

declare [[*code drop*: *TLList.tconcat'*]]

lemma *tconcat-Lazy-tllist* [*code*]:
TLList.tconcat' b (*Lazy-tllist xss*) =
Lazy-tllist (λ-. *case xss* () *of Inr b'* ⇒ *Inr b'* | *Inl* (*xs*, *xss'*) ⇒ *force* (*lappendt xs* (*TLList.tconcat' b xss'*)))
⟨*proof*⟩

declare [[*code drop*: *tllist-all2*]]

lemma *tllist-all2-Lazy-tllist* [*code*]:
tllist-all2 P Q (*Lazy-tllist xs*) (*Lazy-tllist ys*) ⟷
(*case xs* () *of*
Inr b ⇒ (*case ys* () *of Inr b'* ⇒ *Q b b'* | *Inl* - ⇒ *False*)
| *Inl* (*x*, *xs'*) ⇒ (*case ys* () *of Inr* - ⇒ *False* | *Inl* (*y*, *ys'*) ⇒ *P x y* ∧ *tllist-all2 P Q xs' ys'*))
⟨*proof*⟩

declare [[*code drop*: *llist-of-tllist*]]

lemma *llist-of-tllist-Lazy-tllist* [*code*]:
llist-of-tllist (*Lazy-tllist xs*) =
Lazy-llist (λ-. *case xs* () *of Inl* (*x*, *ys*) ⇒ *Some* (*x*, *llist-of-tllist ys*) | *Inr b* ⇒ *None*)
⟨*proof*⟩

declare [[*code drop*: *tnth*]]

lemma *tnth-Lazy-tllist* [*code*]:
tnth (*Lazy-tllist xs*) *n* =
(*case xs* () *of Inr b* ⇒ *undefined n* | *Inl* (*x*, *ys*) ⇒ *if n* = *0 then x else tnth ys* (*n* − *1*))
⟨*proof*⟩

declare [[*code drop*: *gen-tlength*]]

lemma *gen-tlength-Lazy-tllist* [*code*]:
gen-tlength n (*Lazy-tllist xs*) =
(*case xs* () *of Inr b* ⇒ *enat n* | *Inl* (-, *xs′*) ⇒ *gen-tlength* (*n* + *1*) *xs′*)
⟨*proof*⟩

declare [[*code drop*: *tdropn*]]

lemma *tdropn-Lazy-tllist* [*code*]:
tdropn n (*Lazy-tllist xs*) =
Lazy-tllist (λ-. *if n* = *0 then xs* () *else case xs* () *of Inr b* ⇒ *Inr b* | *Inl* (*x*, *xs′*) ⇒ *force* (*tdropn* (*n* − *1*) *xs′*))
⟨*proof*⟩

declare *Lazy-tllist-def* [*simp del*]
declare *sum.splits* [*split del*]

Simple ML test for laziness

⟨*ML*⟩

hide-const (**open**) *force*

end

10 CCPO topologies

theory *CCPO-Topology*
imports
HOL−Analysis.Extended-Real-Limits
../Coinductive-Nat
begin

lemma *dropWhile-append*:
dropWhile P (*xs* @ *ys*) = (*if* ∀ *x*∈*set xs*. *P x then dropWhile P ys else dropWhile P xs* @ *ys*)
⟨*proof*⟩

lemma *dropWhile-False*: (⋀*x*. *x* ∈ *set xs* ⟹ *P x*) ⟹ *dropWhile P xs* = []
⟨*proof*⟩

abbreviation (**in** *order*) *chain* ≡ *Complete-Partial-Order.chain* (≤)

lemma (**in** *linorder*) *chain-linorder*: *chain C*
⟨*proof*⟩

lemma *continuous-add-ereal*:
assumes $0 \leq t$
shows *continuous-on* $\{-\infty::ereal <..\}$ $(\lambda x.\ t + x)$
⟨*proof*⟩

lemma *tendsto-add-ereal*:
$0 \leq x \Longrightarrow 0 \leq y \Longrightarrow (f \longrightarrow y)\ F \Longrightarrow ((\lambda z.\ x + f\ z :: ereal) \longrightarrow x + y)\ F$
⟨*proof*⟩

lemma *tendsto-LimI*: $(f \longrightarrow y)\ F \Longrightarrow (f \longrightarrow Lim\ F\ f)\ F$
⟨*proof*⟩

10.1 The filter at'

abbreviation (**in** *ccpo*) *compact-element* ≡ *ccpo.compact Sup* (≤)

lemma *tendsto-unique-eventually*:
fixes $x\ x' :: 'a :: t2\text{-}space$
shows $F \neq bot \Longrightarrow eventually\ (\lambda x.\ f\ x = g\ x)\ F \Longrightarrow (f \longrightarrow x)\ F \Longrightarrow (g \longrightarrow x')\ F \Longrightarrow x = x'$
⟨*proof*⟩

lemma (**in** *ccpo*) *ccpo-Sup-upper2*: *chain C* ⟹ $x \in C \Longrightarrow y \leq x \Longrightarrow y \leq Sup\ C$
⟨*proof*⟩

lemma *tendsto-open-vimage*: $(\bigwedge B.\ open\ B \Longrightarrow open\ (f\ -`\ B)) \Longrightarrow f\ -l\to f\ l$
⟨*proof*⟩

lemma *open-vimageI*: $(\bigwedge x.\ f\ -x\to f\ x) \Longrightarrow open\ A \Longrightarrow open\ (f\ -`\ A)$
⟨*proof*⟩

lemma *principal-bot*: $principal\ x = bot \longleftrightarrow x = \{\}$
⟨*proof*⟩

definition $at'\ x = (if\ open\ \{x\}\ then\ principal\ \{x\}\ else\ at\ x)$

lemma *at'-bot*: $at'\ x \neq bot$
⟨*proof*⟩

lemma *tendsto-id-at'*[*simp*, *intro*]: $((\lambda x.\ x) \longrightarrow x)\ (at'\ x)$
⟨*proof*⟩

lemma *cont-at'*: $(f \longrightarrow f\ x)\ (at'\ x) \longleftrightarrow f\ -x\to f\ x$
⟨*proof*⟩

10.2 The type class *ccpo-topology*

Temporarily relax type constraints for *open*.

⟨*ML*⟩

class *ccpo-topology* = *open* + *ccpo* +
  assumes *open-ccpo*: *open A* ⟷ (∀ *C*. *chain C* ⟶ *C* ≠ {} ⟶ *Sup C* ∈ *A* ⟶ *C* ∩ *A* ≠ {})
begin

lemma *open-ccpoD*:
  assumes *open A chain C C* ≠ {} *Sup C* ∈ *A*
  shows ∃ *c*∈*C*. ∀ *c'*∈*C*. *c* ≤ *c'* ⟶ *c'* ∈ *A*
⟨*proof*⟩

lemma *open-ccpo-Iic*: *open* {.. *b*}
  ⟨*proof*⟩

subclass *topological-space*
⟨*proof*⟩

lemma *closed-ccpo*: *closed A* ⟷ (∀ *C*. *chain C* ⟶ *C* ≠ {} ⟶ *C* ⊆ *A* ⟶ *Sup C* ∈ *A*)
  ⟨*proof*⟩

lemma *closed-admissible*: *closed* {*x*. *P x*} ⟷ *ccpo.admissible Sup* (≤) *P*
  ⟨*proof*⟩

lemma *open-singletonI-compact*: *compact-element x* ⟹ *open* {*x*}
  ⟨*proof*⟩

lemma *closed-Ici*: *closed* {.. *b*}
  ⟨*proof*⟩

lemma *closed-Iic*: *closed* {*b* ..}
  ⟨*proof*⟩

*ccpo-topology*s are also *t2-space*s. This is necessary to have a unique continuous extension.

subclass *t2-space*
⟨*proof*⟩

end

lemma *tendsto-le-ccpo*:
  fixes *f g* :: *'a* ⇒ *'b*::*ccpo-topology*
  assumes *F*: ¬ *trivial-limit F*
  assumes *x*: (*f* ⟶⟶ *x*) *F* **and** *y*: (*g* ⟶⟶ *y*) *F*
  assumes *ev*: *eventually* (λ*x*. *g x* ≤ *f x*) *F*

shows $y \leq x$
⟨*proof*⟩

lemma *tendsto-ccpoI*:
fixes $f :: 'a::ccpo\text{-}topology \Rightarrow 'b::ccpo\text{-}topology$
shows $(\bigwedge C.\ chain\ C \Longrightarrow C \neq \{\} \Longrightarrow chain\ (f\ `\ C) \wedge f\ (Sup\ C) = Sup\ (f`C)) \Longrightarrow f\ -x\rightarrow\ f\ x$
⟨*proof*⟩

lemma *tendsto-mcont*:
assumes *mcont*: $mcont\ Sup\ (\leq)\ Sup\ (\leq)\ (f :: 'a :: ccpo\text{-}topology \Rightarrow 'b :: ccpo\text{-}topology)$
shows $f\ -l\rightarrow\ f\ l$
⟨*proof*⟩

10.3 Instances for *ccpo-topology*s and continuity theorems

instantiation *set* :: (*type*) *ccpo-topology*
begin

definition *open-set* :: $'a\ set\ set \Rightarrow bool$ **where**
$open\text{-}set\ A \longleftrightarrow (\forall C.\ chain\ C \longrightarrow C \neq \{\} \longrightarrow Sup\ C \in A \longrightarrow C \cap A \neq \{\})$

instance
⟨*proof*⟩

end

instantiation *enat* :: *ccpo-topology*
begin

instance
⟨*proof*⟩

end

lemmas *tendsto-inf2*[*THEN tendsto-compose*, *tendsto-intros*] =
tendsto-mcont[*OF mcont-inf2*]

lemma *isCont-inf2*[*THEN isCont-o2*[*rotated*]]:
$isCont\ (\lambda x.\ x \sqcap y)\ (z :: - :: \{ccpo\text{-}topology, complete\text{-}distrib\text{-}lattice\})$
⟨*proof*⟩

lemmas *tendsto-sup1*[*THEN tendsto-compose*, *tendsto-intros*] =
tendsto-mcont[*OF mcont-sup1*]

lemma *isCont-If*: $isCont\ f\ x \Longrightarrow isCont\ g\ x \Longrightarrow isCont\ (\lambda x.\ if\ Q\ then\ f\ x\ else\ g\ x)\ x$
⟨*proof*⟩

lemma *isCont-enat-case*: *isCont* (*f* (*epred n*)) *x* ⟹ *isCont g x* ⟹ *isCont* (λ*x*. *co.case-enat* (*g x*) (λ*n*. *f n x*) *n*) *x*
⟨*proof*⟩

end

11 A CCPO topology on lazy lists with examples

theory *LList-CCPO-Topology* **imports**
CCPO-Topology
../Coinductive-List-Prefix
begin

lemma *closed-Collect-eq-isCont*:
fixes *f g* :: *'a* :: *t2-space* ⇒ *'b*::*t2-space*
assumes *f*: ⋀*x*. *isCont f x* **and** *g*: ⋀*x*. *isCont g x*
shows *closed* {*x*. *f x* = *g x*}
⟨*proof*⟩

instantiation *llist* :: (*type*) *ccpo-topology*
begin

definition *open-llist* :: *'a llist set* ⇒ *bool* **where**
open-llist A ⟷ (∀ *C*. *chain C* ⟶ *C* ≠ {} ⟶ *Sup C* ∈ *A* ⟶ *C* ∩ *A* ≠ {})

instance
⟨*proof*⟩

end

11.1 Continuity and closedness of predefined constants

lemma *tendsto-mcont-llist*: *mcont lSup lprefix lSup lprefix f* ⟹ *f* −*l*→ *f l*
⟨*proof*⟩

lemma *tendsto-ltl*[*THEN tendsto-compose*, *tendsto-intros*]: *ltl* −*l*→ *ltl l*
⟨*proof*⟩

lemma *tendsto-lappend2*[*THEN tendsto-compose*, *tendsto-intros*]: *lappend l* −*l'*→ *lappend l l'*
⟨*proof*⟩

lemma *tendsto-LCons*[*THEN tendsto-compose*, *tendsto-intros*]: *LCons x* −*l*→ *LCons x l*
⟨*proof*⟩

lemma *tendsto-lmap*[*THEN tendsto-compose*, *tendsto-intros*]: *lmap f* −*l*→ *lmap f l*

⟨*proof*⟩

lemma *tendsto-llength*[*THEN tendsto-compose*, *tendsto-intros*]: *llength* −*l*→ *llength l*
⟨*proof*⟩

lemma *tendsto-lset*[*THEN tendsto-compose*, *tendsto-intros*]: *lset* −*l*→ *lset l*
⟨*proof*⟩

lemma *open-lhd*: *open* {*l*. ¬ *lnull l* ∧ *lhd l* = *x*}
⟨*proof*⟩

lemma *open-LCons′*: **assumes** *A*: *open A* **shows** *open* (*LCons x* ‘ *A*)
⟨*proof*⟩

lemma *open-Ici*: *lfinite xs* ⟹ *open* {*xs* ..}
⟨*proof*⟩

lemma *open-lfinite*[*simp*]: *lfinite x* ⟹ *open* {*x*}
⟨*proof*⟩

lemma *open-singleton-iff-lfinite*: *open* {*x*} ⟷ *lfinite x*
⟨*proof*⟩

lemma *closure-eq-lfinite*:
assumes *closed-Q*: *closed* {*xs*. *Q xs*}
assumes *downwards-Q*: ⋀*xs ys*. *Q xs* ⟹ *lprefix ys xs* ⟹ *Q ys*
shows {*xs*. *Q xs*} = *closure* {*xs*. *lfinite xs* ∧ *Q xs*}
⟨*proof*⟩

lemma *closure-lfinite*: *closure* {*xs*. *lfinite xs*} = *UNIV*
⟨*proof*⟩

lemma *closed-ldistinct*: *closed* {*xs*. *ldistinct xs*}
⟨*proof*⟩

lemma *ldistinct-closure*: {*xs*. *ldistinct xs*} = *closure* {*xs*. *lfinite xs* ∧ *ldistinct xs*}
⟨*proof*⟩

lemma *closed-ldistinct′*: (⋀*x*. *isCont f x*) ⟹ *closed* {*xs*. *ldistinct* (*f xs*)}
⟨*proof*⟩

lemma *closed-lsorted*: *closed* {*xs*. *lsorted xs*}
⟨*proof*⟩

lemma *lsorted-closure*: {*xs*. *lsorted xs*} = *closure* {*xs*. *lfinite xs* ∧ *lsorted xs*}
⟨*proof*⟩

lemma *closed-lsorted′*: (⋀*x*. *isCont f x*) ⟹ *closed* {*xs*. *lsorted* (*f xs*)}

⟨*proof*⟩

lemma *closed-in-lset*: *closed* {*l*. *x* ∈ *lset l*}
⟨*proof*⟩

lemma *closed-llist-all2*:
closed {(*x*, *y*). *llist-all2 R x y*}
⟨*proof*⟩

lemma *closed-list-all2*:
fixes *f g* :: *'b*::*t2-space* ⇒ *'a llist*
assumes *f*: ⋀*x*. *isCont f x* **and** *g*: ⋀*x*. *isCont g x*
shows *closed* {*x*. *llist-all2 R* (*f x*) (*g x*)}
⟨*proof*⟩

lemma *at-botI-lfinite*[*simp*]: *lfinite l* ⟹ *at l* = *bot*
⟨*proof*⟩

lemma *at-eq-lfinite*: *at l* = (*if lfinite l then bot else at*′ *l*)
⟨*proof*⟩

lemma *eventually-lfinite*: *eventually lfinite* (*at*′ *x*)
⟨*proof*⟩

lemma *eventually-nhds-llist*:
eventually P (*nhds l*) ⟷ (∃ *xs*≤*l*. *lfinite xs* ∧ (∀ *ys*≥*xs*. *ys* ≤ *l* ⟶ *P ys*))
⟨*proof*⟩

lemma *nhds-lfinite*: *lfinite l* ⟹ *nhds l* = *principal* {*l*}
⟨*proof*⟩

lemma *eventually-at*′*-llist*:
eventually P (*at*′ *l*) ⟷ (∃ *xs*≤*l*. *lfinite xs* ∧ (∀ *ys*≥*xs*. *lfinite ys* ⟶ *ys* ≤ *l* ⟶ *P ys*))
⟨*proof*⟩

lemma *eventually-at*′*-llistI*: (⋀*xs*. *lfinite xs* ⟹ *xs* ≤ *l* ⟹ *P xs*) ⟹ *eventually P* (*at*′ *l*)
⟨*proof*⟩

lemma *Lim-at*′*-lfinite*: *lfinite xs* ⟹ *Lim* (*at*′ *xs*) *f* = *f xs*
⟨*proof*⟩

lemma *filterlim-at*′*-list*:
(*f* ⟶ *y*) (*at*′ (*x*::*'a llist*)) ⟹ *f* −*x*→ *y*
⟨*proof*⟩

lemma *tendsto-mcont-llist*′: *mcont lSup lprefix lSup lprefix f* ⟹ (*f* ⟶ *f x*) (*at*′ (*x* :: *'a llist*))

⟨*proof*⟩

lemma *tendsto-closed*:
assumes *eq*: *closed* {*x*. *P x*}
assumes *ev*: ⋀*ys*. *lfinite ys* ⟹ *ys* ≤ *x* ⟹ *P ys*
shows *P x*
⟨*proof*⟩

lemma *tendsto-Sup-at'*:
fixes *f* :: *'a llist* ⇒ *'b*::*ccpo-topology*
assumes *f*: ⋀*x y*. *x* ≤ *y* ⟹ *lfinite x* ⟹ *lfinite y* ⟹ *f x* ≤ *f y*
shows (*f* ⟶ (*Sup* (*f*‘{*xs*. *lfinite xs* ∧ *xs* ≤ *l*}))) (*at' l*)
⟨*proof*⟩

lemma *tendsto-Lim-at'*:
fixes *f* :: *'a llist* ⇒ *'b*::*ccpo-topology*
assumes *f*: ⋀*l*. *f l* = *Lim* (*at' l*) *f'*
assumes *mono*: ⋀*x y*. *x* ≤ *y* ⟹ *lfinite x* ⟹ *lfinite y* ⟹ *f' x* ≤ *f' y*
shows (*f* ⟶ *f l*) (*at' l*)
⟨*proof*⟩

lemma *isCont-LCons*[*THEN isCont-o2*[*rotated*]]: *isCont* (*LCons x*) *l*
⟨*proof*⟩

lemma *isCont-lmap*[*THEN isCont-o2*[*rotated*]]: *isCont* (*lmap f*) *l*
⟨*proof*⟩

lemma *isCont-lappend*[*THEN isCont-o2*[*rotated*]]: *isCont* (*lappend xs*) *ys*
⟨*proof*⟩

lemma *isCont-lset*[*THEN isCont-o2*[*rotated*]]: *isCont lset xs*
⟨*proof*⟩

11.2 Define *lfilter* as continuous extension

definition *lfilter' P l* = *Lim* (*at' l*) (λ*xs*. *llist-of* (*filter P* (*list-of xs*)))

lemma *tendsto-lfilter*: (*lfilter' P* ⟶ *lfilter' P xs*) (*at' xs*)
⟨*proof*⟩

lemma *isCont-lfilter*[*THEN isCont-o2*[*rotated*]]: *isCont* (*lfilter' P*) *l*
⟨*proof*⟩

lemma *lfilter'-lfinite*[*simp*]: *lfinite xs* ⟹ *lfilter' P xs* = *llist-of* (*filter P* (*list-of xs*))
⟨*proof*⟩

lemma *lfilter'-LNil*: *lfilter' P LNil = LNil*
⟨*proof*⟩

lemma *lfilter'-LCons* [*simp*]: *lfilter' P (LCons a xs) = (if P a then LCons a (lfilter' P xs) else lfilter' P xs)*
⟨*proof*⟩

lemma *ldistinct-lfilter'*: *ldistinct l ⟹ ldistinct (lfilter' P l)*
⟨*proof*⟩

lemma *lfilter'-lmap*: *lfilter' P (lmap f xs) = lmap f (lfilter' (P ∘ f) xs)*
⟨*proof*⟩

lemma *lfilter'-lfilter'*: *lfilter' P (lfilter' Q xs) = lfilter' (λx. Q x ∧ P x) xs*
⟨*proof*⟩

lemma *lfilter'-LNil-I*[*simp*]: *(∀ x ∈ lset xs. ¬ P x) ⟹ lfilter' P xs = LNil*
⟨*proof*⟩

lemma *lset-lfilter'*: *lset (lfilter' P xs) = lset xs ∩ {x. P x}*
⟨*proof*⟩

lemma *lfilter'-eq-LNil-iff*: *lfilter' P xs = LNil ⟷ (∀ x∈lset xs. ¬ P x)*
⟨*proof*⟩

lemma *lfilter'-eq-lfilter*: *lfilter' P xs = lfilter P xs*
⟨*proof*⟩

11.3 Define *lconcat* as continuous extension

definition *lconcat' xs = Lim (at' xs) (λxs. foldr lappend (list-of xs) LNil)*

lemma *tendsto-lconcat'*: *(lconcat' ⟶ lconcat' xss) (at' xss)*
⟨*proof*⟩

lemma *isCont-lconcat'*[*THEN isCont-o2*[*rotated*]]: *isCont lconcat' l*
⟨*proof*⟩

lemma *lconcat'-lfinite*[*simp*]: *lfinite xs ⟹ lconcat' xs = foldr lappend (list-of xs) LNil*
⟨*proof*⟩

lemma *lconcat'-LNil*: *lconcat' LNil = LNil*
⟨*proof*⟩

lemma *lconcat'-LCons* [*simp*]: *lconcat' (LCons l xs) = lappend l (lconcat' xs)*
⟨*proof*⟩

lemma *lmap-lconcat*: *lmap f (lconcat' xss) = lconcat' (lmap (lmap f) (xss::'a llist*

llist))
⟨*proof*⟩

lemmas *tendsto-Sup*[*THEN tendsto-compose*, *tendsto-intros*] =
mcont-SUP[*OF mcont-id*′ *mcont-const*, *THEN tendsto-mcont*]

lemma
assumes *fin*: ∀ *xs*∈*lset xss*. *lfinite xs*
shows *lset* (*lconcat*′ *xss*) = (⋃ *xs*∈*lset xss*. *lset xs*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

11.4 Define *ldropWhile* as continuous extension

definition *ldropWhile*′ *P xs* = *Lim* (*at*′ *xs*) (λ*xs*. *llist-of* (*dropWhile P* (*list-of xs*)))

lemma *tendsto-ldropWhile*′:
(*ldropWhile*′ *P* ⟶ *ldropWhile*′ *P xs*) (*at*′ *xs*)
⟨*proof*⟩

lemma *isCont-ldropWhile*′[*THEN isCont-o2*[*rotated*]]: *isCont* (*ldropWhile*′ *P*) *l*
⟨*proof*⟩

lemma *ldropWhile*′-*lfinite*[*simp*]: *lfinite xs* ⟹ *ldropWhile*′ *P xs* = *llist-of* (*dropWhile P* (*list-of xs*))
⟨*proof*⟩

lemma *ldropWhile*′-*LNil*: *ldropWhile*′ *P LNil* = *LNil*
⟨*proof*⟩

lemma *ldropWhile*′-*LCons* [*simp*]: *ldropWhile*′ *P* (*LCons l xs*) = (*if P l then ldropWhile*′ *P xs else LCons l xs*)
⟨*proof*⟩

lemma *ldropWhile*′ *P* (*lmap f xs*) = *lmap f* (*ldropWhile*′ (*P* ∘ *f*) *xs*)
⟨*proof*⟩

lemma *ldropWhile*′-*LNil-I*[*simp*]: ∀ *x* ∈ *lset xs*. *P x* ⟹ *ldropWhile*′ *P xs* = *LNil*
⟨*proof*⟩

lemma *lnull-ldropWhile*′: *lnull* (*ldropWhile*′ *P xs*) ⟷ (∀ *x* ∈ *lset xs*. *P x*) (**is** *?lhs* ⟷ -)
⟨*proof*⟩

lemma *lhd-lfilter*′: *lhd* (*lfilter*′ *P xs*) = *lhd* (*ldropWhile*′ (*Not* ∘ *P*) *xs*)
⟨*proof*⟩

11.5 Define *ldrop* as continuous extension

primrec *edrop* **where**

edrop n [] = []
| *edrop n (x # xs) = (case n of eSuc n ⇒ edrop n xs | 0 ⇒ x # xs)*

lemma *mono-edrop*: *edrop n xs ≤ edrop n (xs @ ys)*
⟨*proof*⟩

lemma *edrop-mono*: *xs ≤ ys ⟹ edrop n xs ≤ edrop n ys*
⟨*proof*⟩

definition *ldrop′ n xs = Lim (at′ xs) (llist-of ∘ edrop n ∘ list-of)*

lemma *ldrop′-lfinite*[*simp*]: *lfinite xs ⟹ ldrop′ n xs = llist-of (edrop n (list-of xs))*
⟨*proof*⟩

lemma *tendsto-ldrop′*: *(ldrop′ n ⟶ ldrop′ n l) (at′ l)*
⟨*proof*⟩

lemma *isCont-ldrop′*[*THEN isCont-o2*[*rotated*]]: *isCont (ldrop′ n) l*
⟨*proof*⟩

lemma *ldrop′ n LNil = LNil*
⟨*proof*⟩

lemma *ldrop′ n (LCons x xs) = (case n of 0 ⇒ LCons x xs | eSuc n ⇒ ldrop′ n xs)*
⟨*proof*⟩

primrec *up* :: *′a* :: *order* ⇒ *′a list* ⇒ *′a list* **where**
up a [] = []
| *up a (x # xs) = (if a < x then x # up x xs else up a xs)*

lemma *set-upD*: *x ∈ set (up y xs) ⟹ x ∈ set xs ∧ y < x*
⟨*proof*⟩

lemma *prefix-up*: *prefix (up a xs) (up a (xs @ ys))*
⟨*proof*⟩

lemma *mono-up*: *xs ≤ ys ⟹ up a xs ≤ up a ys*
⟨*proof*⟩

lemma *sorted-up*: *sorted (up a xs)*
⟨*proof*⟩

11.6 Define more functions on lazy lists as continuous extensions

definition *lup a xs = Lim (at′ xs) (λxs. llist-of (up a (list-of xs)))*

lemma *tendsto-lup*: *(lup a ⟶ lup a xs) (at′ xs)*

⟨*proof*⟩

lemma *isCont-lup*[*THEN isCont-o2*[*rotated*]]: *isCont* (*lup a*) *l*
⟨*proof*⟩

lemma *lup-lfinite*[*simp*]: *lfinite xs* ⟹ *lup a xs* = *llist-of* (*up a* (*list-of xs*))
⟨*proof*⟩

lemma *lup-LNil*: *lup a LNil* = *LNil*
⟨*proof*⟩

lemma *lup-LCons* [*simp*]: *lup a* (*LCons x xs*) = (*if a* < *x then LCons x* (*lup x xs*) *else lup a xs*)
⟨*proof*⟩

lemma *lset-lup*: *lset* (*lup x xs*) ⊆ *lset xs* ∩ {*y*. *x* < *y*}
⟨*proof*⟩

lemma *lsorted-lup*: *lsorted* (*lup* (*a*::′*a*::*linorder*) *l*)
⟨*proof*⟩

context notes [[*function-internals*]]
begin

partial-function (*llist*) *lup*′ :: ′*a* :: *ord* ⇒ ′*a llist* ⇒ ′*a llist* **where**
lup′ *a xs* = (*case xs of LNil* ⇒ *LNil* | *LCons x xs* ⇒ *if a* < *x then LCons x* (*lup*′ *x xs*) *else lup*′ *a xs*)

end

declare *lup*′.*mono*[*cont-intro*]

lemma *monotone-lup*′: *monotone* (*rel-prod* (=) *lprefix*) *lprefix* (λ(*a*, *xs*). *lup*′ *a xs*)
⟨*proof*⟩

lemma *mono2mono-lup*′*2*[*THEN llist.mono2mono*, *simp*, *cont-intro*]:
shows *monotone-lup*′*2*: *monotone lprefix lprefix* (*lup*′ *a*)
⟨*proof*⟩

lemma *mcont-lup*′: *mcont* (*prod-lub the-Sup lSup*) (*rel-prod* (=) *lprefix*) *lSup lprefix* (λ(*a*, *xs*). *lup*′ *a xs*)
⟨*proof*⟩

lemma *mcont2mcont-lup*′*2*[*THEN llist.mcont2mcont*, *simp*, *cont-intro*]:
shows *mcont-lup*′*2*: *mcont lSup lprefix lSup lprefix* (*lup*′ *a*)
⟨*proof*⟩

simps-of-case *lup*′*-simps* [*simp*]: *lup*′.*simps*

lemma *lset-lup′-subset*:
fixes x :: - :: *preorder*
shows *lset* (*lup′ x xs*) ⊆ *lset xs* ∩ {*y*. $x < y$}
⟨*proof*⟩

lemma *in-lset-lup′D*:
fixes x :: - :: *preorder*
assumes $y \in$ *lset* (*lup′ x xs*)
shows $y \in$ *lset xs* ∧ $x < y$
⟨*proof*⟩

lemma *lsorted-lup′*:
fixes x :: - :: *preorder*
shows *lsorted* (*lup′ x xs*)
⟨*proof*⟩

lemma *ldistinct-lup′*:
fixes x :: - :: *preorder*
shows *ldistinct* (*lup′ x xs*)
⟨*proof*⟩

context fixes f :: $'a \Rightarrow 'a$ **begin**

partial-function (*llist*) *iterate* :: $'a \Rightarrow 'a$ *llist*
where *iterate x* = *LCons x* (*iterate* (*f x*))

lemma *lmap-iterate*: *lmap f* (*iterate x*) = *iterate* (*f x*)
⟨*proof*⟩

end

fun *extup extdown* :: *int* ⇒ *int list* ⇒ *int list* **where**
extup i [] = []
| *extup i* (*x* # *xs*) = (*if i* ≤ *x then extup x xs else i* # *extdown x xs*)
| *extdown i* [] = []
| *extdown i* (*x* # *xs*) = (*if i* ≥ *x then extdown x xs else i* # *extup x xs*)

lemma *prefix-ext*:
prefix (*extup a xs*) (*extup a* (*xs* @ *ys*))
prefix (*extdown a xs*) (*extdown a* (*xs* @ *ys*))
⟨*proof*⟩

lemma *mono-ext*: **assumes** *xs* ≤ *ys* **shows** *extup a xs* ≤ *extup a ys extdown a xs* ≤ *extdown a ys*
⟨*proof*⟩

lemma *set-ext*: *set* (*extup a xs*) ⊆ {*a*} ∪ *set xs set* (*extdown a xs*) ⊆ {*a*} ∪ *set xs*
⟨*proof*⟩

definition *lextup i l* = *Lim* (*at' l*) (*llist-of* ∘ *extup i* ∘ *list-of*)
definition *lextdown i l* = *Lim* (*at' l*) (*llist-of* ∘ *extdown i* ∘ *list-of*)

lemma *tendsto-lextup*[*tendsto-intros*]: (*lextup i* ⟶ *lextup i xs*) (*at' xs*)
⟨*proof*⟩

lemma *tendsto-lextdown*[*tendsto-intros*]: (*lextdown i* ⟶ *lextdown i xs*) (*at' xs*)
⟨*proof*⟩

lemma *isCont-lextup*[*THEN isCont-o2*[*rotated*]]: *isCont* (*lextup a*) *l*
⟨*proof*⟩

lemma *isCont-lextdown*[*THEN isCont-o2*[*rotated*]]: *isCont* (*lextdown a*) *l*
⟨*proof*⟩

lemma *lextup-lfinite*[*simp*]: *lfinite xs* ⟹ *lextup i xs* = *llist-of* (*extup i* (*list-of xs*))
⟨*proof*⟩

lemma *lextdown-lfinite*[*simp*]: *lfinite xs* ⟹ *lextdown i xs* = *llist-of* (*extdown i* (*list-of xs*))
⟨*proof*⟩

lemma *lextup i LNil* = *LNil lextdown i LNil* = *LNil*
⟨*proof*⟩

lemma *lextup i* (*LCons x xs*) = (*if i* ≤ *x then lextup x xs else LCons i* (*lextdown x xs*))
⟨*proof*⟩

lemma *lextdown i* (*LCons x xs*) = (*if x* ≤ *i then lextdown x xs else LCons i* (*lextup x xs*))
⟨*proof*⟩

lemma *lset* (*lextup a xs*) ⊆ {*a*} ∪ *lset xs*
⟨*proof*⟩

lemma *lset* (*lextdown a xs*) ⊆ {*a*} ∪ *lset xs*
⟨*proof*⟩

lemma *distinct-ext*:
assumes *distinct xs a* ∉ *set xs*
shows *distinct* (*extup a xs*) *distinct* (*extdown a xs*)
⟨*proof*⟩

lemma *ldistinct xs* ⟹ *a* ∉ *lset xs* ⟹ *ldistinct* (*lextup a xs*)
⟨*proof*⟩

definition *esum-list* :: *ereal llist* ⇒ *ereal* **where**
esum-list xs = *Lim* (*at' xs*) (*sum-list* ∘ *list-of*)

lemma *esum-list-lfinite*[*simp*]: *lfinite xs* $\Longrightarrow$ *esum-list xs* = *sum-list* (*list-of xs*)
⟨*proof*⟩

lemma *esum-list-LNil*: *esum-list LNil* = *0*
⟨*proof*⟩

context
fixes *xs* :: *ereal llist*
assumes *xs*: $\bigwedge x.\ x \in lset\ xs \Longrightarrow 0 \leq x$
begin

lemma *esum-list-tendsto-SUP*:
((*sum-list*∘*list-of*) ⟶ (*SUP ys* ∈ {*ys*. *lfinite ys* ∧ *ys* ≤ *xs*}. *esum-list ys*)) (*at*′ *xs*)
(**is** (- ⟶ *?y*) -)
⟨*proof*⟩

lemma *tendsto-esum-list*: (*esum-list* ⟶ *esum-list xs*) (*at*′ *xs*)
⟨*proof*⟩

lemma *isCont-esum-list*: *isCont esum-list xs*
⟨*proof*⟩

end

lemma *esum-list-nonneg*:
($\bigwedge x.\ x \in lset\ xs \Longrightarrow 0 \leq x$) $\Longrightarrow$ *0* ≤ *esum-list xs*
⟨*proof*⟩

lemma *esum-list-LCons*:
assumes *x*: *0* ≤ *x* $\bigwedge x.\ x \in lset\ xs \Longrightarrow 0 \leq x$ **shows** *esum-list* (*LCons x xs*) = *x* + *esum-list xs*
⟨*proof*⟩

lemma *esum-list-lfilter*′:
assumes *nn*: $\bigwedge x.\ x \in lset\ xs \Longrightarrow 0 \leq x$ **shows** *esum-list* (*lfilter*′ (λ*x*. *x* ≠ *0*) *xs*) = *esum-list xs*
⟨*proof*⟩

function *f*:: *nat list* ⇒ *nat list* **where**
f [] = []
| *f* (*x*#*xs*) = (*x* ∗ *2*) # *f* (*f xs*)
⟨*proof*⟩

termination *f*
⟨*proof*⟩

lemma *length-f*[*simp*]: *length* (*f xs*) = *length xs*

⟨*proof*⟩

lemma *f-mono'*: ∃ *ys'*. *f* (*xs* @ *ys*) = *f xs* @ *ys'*
⟨*proof*⟩

lemma *f-mono*: *xs* ≤ *ys* ⟹ *f xs* ≤ *f ys*
⟨*proof*⟩

definition *f'* *l* = *Lim* (*at'* *l*) (λ*l*. *llist-of* (*f* (*list-of* *l*)))

lemma *f'-lfinite*[*simp*]: *lfinite xs* ⟹ *f'* *xs* = *llist-of* (*f* (*list-of xs*))
⟨*proof*⟩

lemma *tendsto-f'*: (*f'* ⟶ *f'* *l*) (*at'* *l*)
⟨*proof*⟩

lemma *isCont-f'*[*THEN isCont-o2*[*rotated*]]: *isCont f'* *l*
⟨*proof*⟩

lemma *f'* *LNil* = *LNil*
⟨*proof*⟩

lemma *f'* (*LCons x xs*) = *LCons* (*x* ∗ *2*) (*f'* (*f'* *xs*))
⟨*proof*⟩

end

12 Ccpo structure for terminated lazy lists

theory *TLList-CCPO* **imports** *TLList* **begin**

lemma *Set-is-empty-parametric* [*transfer-rule*]:
includes *lifting-syntax*
shows (*rel-set A* ===> (=)) *Set.is-empty Set.is-empty*
⟨*proof*⟩

lemma *monotone-comp*: ⟦ *monotone orda ordb g*; *monotone ordb ordc f* ⟧ ⟹ *monotone orda ordc* (*f* ∘ *g*)
⟨*proof*⟩

lemma *cont-comp*: ⟦ *mcont luba orda lubb ordb g*; *cont lubb ordb lubc ordc f* ⟧ ⟹ *cont luba orda lubc ordc* (*f* ∘ *g*)
⟨*proof*⟩

lemma *mcont-comp*: ⟦ *mcont luba orda lubb ordb g*; *mcont lubb ordb lubc ordc f* ⟧ ⟹ *mcont luba orda lubc ordc* (*f* ∘ *g*)
⟨*proof*⟩

context includes *lifting-syntax*

begin

lemma *monotone-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-total A*
shows ((*A* ===> *A* ===> (=)) ===> (*B* ===> *B* ===> (=)) ===> (*A* ===> *B*) ===> (=)) *monotone monotone*
⟨*proof*⟩

lemma *cont-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-total A bi-unique B*
shows ((*rel-set A* ===> *A*) ===> (*A* ===> *A* ===> (=)) ===> (*rel-set B* ===> *B*) ===> (*B* ===> *B* ===> (=)) ===> (*A* ===> *B*) ===> (=)) *cont cont*
⟨*proof*⟩

lemma *mcont-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-total A bi-unique B*
shows ((*rel-set A* ===> *A*) ===> (*A* ===> *A* ===> (=)) ===> (*rel-set B* ===> *B*) ===> (*B* ===> *B* ===> (=)) ===> (*A* ===> *B*) ===> (=)) *mcont mcont*
⟨*proof*⟩

end

lemma (**in** *ccpo*) *Sup-Un-less*:
assumes *chain*: *Complete-Partial-Order.chain* (≤) (*A* ∪ *B*)
and *AB*: ∀ *x*∈*A*. ∃ *y*∈*B*. *x* ≤ *y*
shows *Sup* (*A* ∪ *B*) = *Sup B*
⟨*proof*⟩

12.1 The ccpo structure

context includes *tllist.lifting* **fixes** *b* :: *'b* **begin**

lift-definition *tllist-ord* :: (*'a*, *'b*) *tllist* ⇒ (*'a*, *'b*) *tllist* ⇒ *bool*
is λ(*xs1*, *b1*) (*xs2*, *b2*). *if lfinite xs1 then b1 = b* ∧ *lprefix xs1 xs2* ∨ *xs1 = xs2* ∧ *flat-ord b b1 b2 else xs1 = xs2*
⟨*proof*⟩

lift-definition *tSup* :: (*'a*, *'b*) *tllist set* ⇒ (*'a*, *'b*) *tllist*
is λ*A*. (*lSup* (*fst* ‘ *A*), *flat-lub b* (*snd* ‘ (*A* ∩ {(*xs*, -). *lfinite xs*}))
⟨*proof*⟩

lemma *tllist-ord-simps* [*simp*, *code*]:
shows *tllist-ord-TNil-TNil*: *tllist-ord* (*TNil b1*) (*TNil b2*) ⟷ *flat-ord b b1 b2*
and *tllist-ord-TNil-TCons*: *tllist-ord* (*TNil b1*) (*TCons y ys*) ⟷ *b1 = b*
and *tllist-ord-TCons-TNil*: *tllist-ord* (*TCons x xs*) (*TNil b2*) ⟷ *False*
and *tllist-ord-TCons-TCons*: *tllist-ord* (*TCons x xs*) (*TCons y ys*) ⟷ *x = y* ∧ *tllist-ord xs ys*

⟨*proof*⟩

lemma *tllist-ord-refl* [*simp*]: *tllist-ord xs xs*
⟨*proof*⟩

lemma *tllist-ord-antisym*: ⟦ *tllist-ord xs ys*; *tllist-ord ys xs* ⟧ ⟹ *xs* = *ys*
⟨*proof*⟩

lemma *tllist-ord-trans* [*trans*]: ⟦ *tllist-ord xs ys*; *tllist-ord ys zs* ⟧ ⟹ *tllist-ord xs zs*
⟨*proof*⟩

lemma *chain-tllist-llist-of-tllist*:
assumes *Complete-Partial-Order.chain tllist-ord A*
shows *Complete-Partial-Order.chain lprefix* (*llist-of-tllist* ‘ *A*)
⟨*proof*⟩

lemma *chain-tllist-terminal*:
assumes *Complete-Partial-Order.chain tllist-ord A*
shows *Complete-Partial-Order.chain* (*flat-ord b*) {*terminal xs*|*xs*. *xs* ∈ *A* ∧ *tfinite xs*}
⟨*proof*⟩

lemma *flat-ord-chain-finite*:
assumes *Complete-Partial-Order.chain* (*flat-ord b*) *A*
shows *finite A*
⟨*proof*⟩

lemma *tSup-empty* [*simp*]: *tSup* {} = *TNil b*
⟨*proof*⟩

lemma *is-TNil-tSup* [*simp*]: *is-TNil* (*tSup A*) ⟷ (∀ *x*∈*A*. *is-TNil x*)
⟨*proof*⟩

lemma *chain-tllist-ord-tSup*:
assumes *chain*: *Complete-Partial-Order.chain tllist-ord A*
and *A*: *xs* ∈ *A*
shows *tllist-ord xs* (*tSup A*)
⟨*proof*⟩

lemma *chain-tSup-tllist-ord*:
assumes *chain*: *Complete-Partial-Order.chain tllist-ord A*
and *lub*: ⋀*xs*′. *xs*′ ∈ *A* ⟹ *tllist-ord xs*′ *xs*
shows *tllist-ord* (*tSup A*) *xs*
⟨*proof*⟩

lemma *tllist-ord-ccpo* [*simp*, *cont-intro*]:
class.ccpo tSup tllist-ord (*mk-less tllist-ord*)
⟨*proof*⟩

lemma *tllist-ord-partial-function-definitions*: *partial-function-definitions tllist-ord tSup*
⟨*proof*⟩

interpretation *tllist*: *partial-function-definitions tllist-ord tSup*
⟨*proof*⟩

lemma *admissible-mcont-is-TNil* [*THEN admissible-subst*, *cont-intro*, *simp*]:
shows *admissible-is-TNil*: *ccpo.admissible tSup tllist-ord is-TNil*
⟨*proof*⟩

lemma *terminal-tSup*:
∀ *xs*∈ *Y*. *is-TNil xs* ⟹ *terminal* (*tSup Y*) = *flat-lub b* (*terminal* ‘ *Y*)
including *tllist.lifting* ⟨*proof*⟩

lemma *thd-tSup*:
∃ *xs* ∈ *Y*. ¬ *is-TNil xs*
⟹ *thd* (*tSup Y*) = (*THE x*. *x* ∈ *thd* ‘ (*Y* ∩ {*xs*. ¬ *is-TNil xs*}))
⟨*proof*⟩

lemma *ex-TCons-raw-parametric*:
includes *lifting-syntax*
shows (*rel-set* (*rel-prod* (*llist-all2 A*) *B*) ===> (=)) (λ*Y*. ∃ (*xs*, *b*) ∈ *Y*. ¬ *lnull xs*) (λ*Y*. ∃ (*xs*, *b*) ∈ *Y*. ¬ *lnull xs*)
⟨*proof*⟩

lift-definition *ex-TCons* :: ('*a*, '*b*) *tllist set* ⇒ *bool*
is λ*Y*. ∃ (*xs*, *b*) ∈ *Y*. ¬ *lnull xs* **parametric** *ex-TCons-raw-parametric*
⟨*proof*⟩

lemma *ex-TCons-iff*: *ex-TCons Y* ⟷ (∃ *xs* ∈ *Y*. ¬ *is-TNil xs*)
⟨*proof*⟩

lemma *retain-TCons-raw-parametric*:
includes *lifting-syntax*
shows (*rel-set* (*rel-prod* (*llist-all2 A*) *B*) ===> *rel-set* (*rel-prod* (*llist-all2 A*) *B*))
(λ*A*. *A* ∩ {(*xs*, *b*). ¬ *lnull xs*}) (λ*A*. *A* ∩ {(*xs*, *b*). ¬ *lnull xs*})
⟨*proof*⟩

lift-definition *retain-TCons* :: ('*a*, '*b*) *tllist set* ⇒ ('*a*, '*b*) *tllist set*
is λ*A*. *A* ∩ {(*xs*, *b*). ¬ *lnull xs*} **parametric** *retain-TCons-raw-parametric*
⟨*proof*⟩

lemma *retain-TCons-conv*: *retain-TCons A* = *A* ∩ {*xs*. ¬ *is-TNil xs*}
⟨*proof*⟩

lemma *ttl-tSup*:

⟦ *Complete-Partial-Order.chain tllist-ord Y*; ∃ *xs* ∈ *Y*. ¬ *is-TNil xs* ⟧
⟹ *ttl* (*tSup Y*) = *tSup* (*ttl* ‘ (*Y* ∩ {*xs*. ¬ *is-TNil xs*}))
⟨*proof*⟩

lemma *tSup-TCons*: *A* ≠ {} ⟹ *tSup* (*TCons x* ‘ *A*) = *TCons x* (*tSup A*)
⟨*proof*⟩

lemma *tllist-ord-terminalD*:
⟦ *tllist-ord xs ys*; *is-TNil ys* ⟧ ⟹ *flat-ord b* (*terminal xs*) (*terminal ys*)
⟨*proof*⟩

lemma *tllist-ord-bot* [*simp*]: *tllist-ord* (*TNil b*) *xs*
⟨*proof*⟩

lemma *tllist-ord-ttlI*:
tllist-ord xs ys ⟹ *tllist-ord* (*ttl xs*) (*ttl ys*)
⟨*proof*⟩

lemma *not-is-TNil-conv*: ¬ *is-TNil xs* ⟷ (∃ *x xs*′. *xs* = *TCons x xs*′)
⟨*proof*⟩

12.2 Continuity of predefined constants

lemma *mono-tllist-ord-case*:
fixes *bot*
assumes *mono*: ⋀*x*. *monotone tllist-ord ord* (λ*xs*. *f x xs* (*TCons x xs*))
and *ord*: *class.preorder ord* (*mk-less ord*)
and *bot*: ⋀*x*. *ord* (*bot b*) *x*
shows *monotone tllist-ord ord* (λ*xs*. *case xs of TNil b* ⇒ *bot b* | *TCons x xs*′ ⇒ *f x xs*′ *xs*)
⟨*proof*⟩

lemma *mcont-lprefix-case-aux*:
fixes *f bot ord*
defines *g* ≡ λ*xs*. *f* (*thd xs*) (*ttl xs*) (*TCons* (*thd xs*) (*ttl xs*))
assumes *mcont*: ⋀*x*. *mcont tSup tllist-ord lub ord* (λ*xs*. *f x xs* (*TCons x xs*))
and *ccpo*: *class.ccpo lub ord* (*mk-less ord*)
and *bot*: ⋀*x*. *ord* (*bot b*) *x*
and *cont-bot*: *cont* (*flat-lub b*) (*flat-ord b*) *lub ord bot*
shows *mcont tSup tllist-ord lub ord* (λ*xs*. *case xs of TNil b* ⇒ *bot b* | *TCons x xs*′ ⇒ *f x xs*′ *xs*)
(**is** *mcont - - - - ?f*)
⟨*proof*⟩

lemma *cont-TNil* [*simp*, *cont-intro*]: *cont* (*flat-lub b*) (*flat-ord b*) *tSup tllist-ord TNil*
⟨*proof*⟩

lemma *monotone-TCons*: *monotone tllist-ord tllist-ord* (*TCons x*)

⟨*proof*⟩

lemmas *mono2mono-TCons*[*cont-intro*] = *monotone-TCons*[*THEN tllist.mono2mono*]

lemma *mcont-TCons*: *mcont tSup tllist-ord tSup tllist-ord* (*TCons x*)
⟨*proof*⟩

lemmas *mcont2mcont-TCons*[*cont-intro*] = *mcont-TCons*[*THEN tllist.mcont2mcont*]

lemmas [*transfer-rule del*] = *tllist-ord.transfer tSup.transfer*

lifting-update *tllist.lifting*
lifting-forget *tllist.lifting*

lemmas [*transfer-rule*] = *tllist-ord.transfer tSup.transfer*

lemma *mono2mono-tset*[*THEN lfp.mono2mono*, *cont-intro*]:
shows *smonotone-tset*: *monotone tllist-ord* (⊆) *tset*
including *tllist.lifting*
⟨*proof*⟩

lemma *mcont2mcont-tset* [*THEN lfp.mcont2mcont*, *cont-intro*]:
shows *mcont-tset*: *mcont tSup tllist-ord Union* (⊆) *tset*
including *tllist.lifting*
⟨*proof*⟩

end

context includes *lifting-syntax*
begin

lemma *rel-fun-lift*:
(⋀*x*. *A* (*f x*) (*g x*)) ⟹ ((=) ===> *A*) *f g*
⟨*proof*⟩

lemma *tllist-ord-transfer* [*transfer-rule*]:
((=) ===> *pcr-tllist* (=) (=) ===> *pcr-tllist* (=) (=) ===> (=))
(λ*b* (*xs1*, *b1*) (*xs2*, *b2*). *if lfinite xs1 then b1* = *b* ∧ *lprefix xs1 xs2* ∨ *xs1* = *xs2*
∧ *flat-ord b b1 b2 else xs1* = *xs2*)
tllist-ord
⟨*proof*⟩

lemma *tSup-transfer* [*transfer-rule*]:
((=) ===> *rel-set* (*pcr-tllist* (=) (=)) ===> *pcr-tllist* (=) (=))
(λ*b A*. (*lSup* (*fst* ‘ *A*), *flat-lub b* (*snd* ‘ (*A* ∩ {(*xs*, -). *lfinite xs*}))))
tSup
⟨*proof*⟩

end

lifting-update *tllist.lifting*
lifting-forget *tllist.lifting*

interpretation *tllist*: *partial-function-definitions tllist-ord b tSup b* **for** *b*
⟨*proof*⟩

lemma *tllist-case-mono* [*partial-function-mono*, *cont-intro*]:
  assumes *tnil*: $\bigwedge b.$ *monotone orda ordb* ($\lambda f.$ *tnil f b*)
  and *tcons*: $\bigwedge x\ xs.$ *monotone orda ordb* ($\lambda f.$ *tcons f x xs*)
  shows *monotone orda ordb* ($\lambda f.$ *case-tllist* (*tnil f*) (*tcons f*) *xs*)
⟨*proof*⟩

12.3 Definition of recursive functions

locale *tllist-pf* = **fixes** *b* :: *'b*
begin

⟨*ML*⟩

abbreviation *mono-tllist* **where** *mono-tllist* ≡ *monotone* (*fun-ord* (*tllist-ord b*)) (*tllist-ord b*)

lemma *LCons-mono* [*partial-function-mono*, *cont-intro*]:
  mono-tllist A ⟹ *mono-tllist* ($\lambda f.$ *TCons x* (*A f*))
⟨*proof*⟩

end

lemma *mono-tllist-lappendt2* [*partial-function-mono*]:
  tllist-pf.mono-tllist b A ⟹ *tllist-pf.mono-tllist b* ($\lambda f.$ *lappendt xs* (*A f*))
⟨*proof*⟩

lemma *mono-tllist-tappend2* [*partial-function-mono*]:
  assumes $\bigwedge y.$ *tllist-pf.mono-tllist b* (*C y*)
  shows *tllist-pf.mono-tllist b* ($\lambda f.$ *tappend xs* ($\lambda y.$ *C y f*))
⟨*proof*⟩
 including *tllist.lifting*
 ⟨*proof*⟩

end

13 Example definitions using the CCPO structure on terminated lazy lists

theory *TLList-CCPO-Examples* **imports**
  ../TLList-CCPO
begin

context fixes $b :: 'b$ **begin**
interpretation *tllist-pf b* ⟨*proof*⟩

context fixes $P :: 'a \Rightarrow bool$
notes [[*function-internals*]]
begin

partial-function (*tllist*) *tfilter* :: $('a, 'b)$ *tllist* ⇒ $('a, 'b)$ *tllist*
where
tfilter xs = (*case xs of TNil b'* ⇒ *TNil b'* | *TCons x xs'* ⇒ *if P x then TCons x* (*tfilter xs'*) *else tfilter xs'*)

end

simps-of-case *tfilter-simps* [*simp*]: *tfilter.simps*

lemma *is-TNil-tfilter*: *is-TNil* (*tfilter P xs*) ⟷ (∀ *x* ∈ *tset xs*. ¬ *P x*) (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

end

lemma *mcont2mcont-tfilter*[*THEN tllist.mcont2mcont*, *simp*, *cont-intro*]:
shows *mcont-tfilter*: *mcont* (*tSup b*) (*tllist-ord b*) (*tSup b*) (*tllist-ord b*) (*tfilter b P*)
⟨*proof*⟩

lemma *tfilter-tfilter*:
tfilter b P (*tfilter b Q xs*) = *tfilter b* (λ*x*. *P x* ∧ *Q x*) *xs* (**is** *?lhs xs* = *?rhs xs*)
⟨*proof*⟩

declare *ccpo.admissible-leI*[*OF complete-lattice-ccpo*, *cont-intro*, *simp*]

lemma *tset-tfilter*: *tset* (*tfilter b P xs*) = {*x*∈*tset xs*. *P x*}
⟨*proof*⟩

context fixes $b :: 'b$ **begin**
interpretation *tllist-pf b* ⟨*proof*⟩

partial-function (*tllist*) *tconcat* :: $('a\ llist, 'b)$ *tllist* ⇒ $('a, 'b)$ *tllist*
where
tconcat xs = (*case xs of TNil b* ⇒ *TNil b* | *TCons x xs'* ⇒ *lappendt x* (*tconcat xs'*))

end

simps-of-case *tconcat2-simps* [*simp*]: *tconcat.simps*

end

14 Example: Koenig's lemma

theory *Koenigslemma* **imports**
../Coinductive-List
begin

type-synonym *'node graph* = *'node* ⇒ *'node* ⇒ *bool*
type-synonym *'node path* = *'node llist*

coinductive-set *paths* :: *'node graph* ⇒ *'node path set*
for *graph* :: *'node graph*
where
Empty: *LNil* ∈ *paths graph*
| *Single*: *LCons x LNil* ∈ *paths graph*
| *LCons*: ⟦ *graph x y*; *LCons y xs* ∈ *paths graph* ⟧ ⟹ *LCons x* (*LCons y xs*) ∈ *paths graph*

definition *connected* :: *'node graph* ⇒ *bool*
where *connected graph* ⟷ (∀ *n n'*. ∃ *xs*. *llist-of* (*n* # *xs* @ [*n'*]) ∈ *paths graph*)

inductive-set *reachable-via* :: *'node graph* ⇒ *'node set* ⇒ *'node* ⇒ *'node set*
for *graph* :: *'node graph* **and** *ns* :: *'node set* **and** *n* :: *'node*
where ⟦ *LCons n xs* ∈ *paths graph*; *n'* ∈ *lset xs*; *lset xs* ⊆ *ns* ⟧ ⟹ *n'* ∈ *reachable-via graph ns n*

lemma *connectedD*: *connected graph* ⟹ ∃ *xs*. *llist-of* (*n* # *xs* @ [*n'*]) ∈ *paths graph*
⟨*proof*⟩

lemma *paths-LConsD*:
assumes *LCons x xs* ∈ *paths graph*
shows *xs* ∈ *paths graph*
⟨*proof*⟩

lemma *paths-lappendD1*:
assumes *lappend xs ys* ∈ *paths graph*
shows *xs* ∈ *paths graph*
⟨*proof*⟩

lemma *paths-lappendD2*:
assumes *lfinite xs*
and *lappend xs ys* ∈ *paths graph*
shows *ys* ∈ *paths graph*
⟨*proof*⟩

lemma *path-avoid-node*:

assumes *path*: *LCons n xs* ∈ *paths graph*
and *set*: *x* ∈ *lset xs*
and *n-neq-x*: *n* ≠ *x*
shows ∃ *xs'*. *LCons n xs'* ∈ *paths graph* ∧ *lset xs'* ⊆ *lset xs* ∧ *x* ∈ *lset xs'* ∧ *n* ∉ *lset xs'*
⟨*proof*⟩

lemma *reachable-via-subset-unfold*:
reachable-via graph ns n ⊆ (⋃ *n'* ∈ {*n'*. *graph n n'*} ∩ *ns*. *insert n'* (*reachable-via graph* (*ns* − {*n'*}) *n'*))
(**is** *?lhs* ⊆ *?rhs*)
⟨*proof*⟩

theorem *koenigslemma*:
fixes *graph* :: *'node graph*
and *n* :: *'node*
assumes *connected*: *connected graph*
and *infinite*: *infinite* (*UNIV* :: *'node set*)
and *finite-branching*: ⋀*n*. *finite* {*n'*. *graph n n'*}
shows ∃ *xs* ∈ *paths graph*. *n* ∈ *lset xs* ∧ ¬ *lfinite xs* ∧ *ldistinct xs*
⟨*proof*⟩

end

15 Definition of the function lmirror

theory *LMirror* **imports** *../Coinductive-List* **begin**

This theory defines a function *lmirror*.

primcorec *lmirror-aux* :: *'a llist* ⇒ *'a llist* ⇒ *'a llist*
where
lmirror-aux acc xs = (*case xs of LNil* ⇒ *acc* | *LCons x xs'* ⇒ *LCons x* (*lmirror-aux* (*LCons x acc*) *xs'*))

definition *lmirror* :: *'a llist* ⇒ *'a llist*
where *lmirror* = *lmirror-aux LNil*

simps-of-case *lmirror-aux-simps* [*simp*]: *lmirror-aux.code*

lemma *lnull-lmirror-aux* [*simp*]:
lnull (*lmirror-aux acc xs*) = (*lnull xs* ∧ *lnull acc*)
⟨*proof*⟩

lemma *ltl-lmirror-aux*:
ltl (*lmirror-aux acc xs*) = (*if lnull xs then ltl acc else lmirror-aux* (*LCons* (*lhd xs*) *acc*) (*ltl xs*))
⟨*proof*⟩

lemma *lhd-lmirror-aux*:
lhd (*lmirror-aux acc xs*) = (*if lnull xs then lhd acc else lhd xs*)
⟨*proof*⟩

declare *lmirror-aux.sel*[*simp del*]

lemma *lfinite-lmirror-aux* [*simp*]:
lfinite (*lmirror-aux acc xs*) ⟷ *lfinite xs* ∧ *lfinite acc*
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *lmirror-aux-inf*:
¬ *lfinite xs* ⟹ *lmirror-aux acc xs* = *xs*
⟨*proof*⟩

lemma *lmirror-aux-acc*:
lmirror-aux (*lappend ys zs*) *xs* = *lappend* (*lmirror-aux ys xs*) *zs*
⟨*proof*⟩

lemma *lmirror-aux-LCons*:
lmirror-aux acc (*LCons x xs*) = *LCons x* (*lappend* (*lmirror-aux LNil xs*) (*LCons x acc*))
⟨*proof*⟩

lemma *llength-lmirror-aux*: *llength* (*lmirror-aux acc xs*) = *2* ∗ *llength xs* + *llength acc*
⟨*proof*⟩

lemma *lnull-lmirror* [*simp*]: *lnull* (*lmirror xs*) = *lnull xs*
⟨*proof*⟩

lemma *lmirror-LNil* [*simp*]: *lmirror LNil* = *LNil*
⟨*proof*⟩

lemma *lmirror-LCons* [*simp*]: *lmirror* (*LCons x xs*) = *LCons x* (*lappend* (*lmirror xs*) (*LCons x LNil*))
⟨*proof*⟩

lemma *ltl-lmirror* [*simp*]:
¬ *lnull xs* ⟹ *ltl* (*lmirror xs*) = *lappend* (*lmirror* (*ltl xs*)) (*LCons* (*lhd xs*) *LNil*)
⟨*proof*⟩

lemma *lmap-lmirror-aux*: *lmap f* (*lmirror-aux acc xs*) = *lmirror-aux* (*lmap f acc*) (*lmap f xs*)
⟨*proof*⟩

lemma *lmap-lmirror*: *lmap f* (*lmirror xs*) = *lmirror* (*lmap f xs*)
⟨*proof*⟩

lemma *lset-lmirror-aux*: *lset* (*lmirror-aux acc xs*) = *lset* (*lappend xs acc*)
⟨*proof*⟩

lemma *lset-lmirror* [*simp*]: *lset* (*lmirror xs*) = *lset xs*
⟨*proof*⟩

lemma *llength-lmirror* [*simp*]: *llength* (*lmirror xs*) = *2* ∗ *llength xs*
⟨*proof*⟩

lemma *lmirror-llist-of* [*simp*]: *lmirror* (*llist-of xs*) = *llist-of* (*xs* @ *rev xs*)
⟨*proof*⟩

lemma *list-of-lmirror* [*simp*]: *lfinite xs* ⟹ *list-of* (*lmirror xs*) = *list-of xs* @ *rev* (*list-of xs*)
⟨*proof*⟩

lemma *llist-all2-lmirror-aux*:
⟦ *llist-all2 P acc acc′*; *llist-all2 P xs xs′* ⟧
⟹ *llist-all2 P* (*lmirror-aux acc xs*) (*lmirror-aux acc′ xs′*)
⟨*proof*⟩

lemma *enat-mult-cancel1* [*simp*]:
k ∗ *m* = *k* ∗ *n* ⟷ *m* = *n* ∨ *k* = *0* ∨ *k* = (∞ :: *enat*) ∧ *n* ≠ *0* ∧ *m* ≠ *0*
⟨*proof*⟩

lemma *llist-all2-lmirror-auxD*:
⟦ *llist-all2 P* (*lmirror-aux acc xs*) (*lmirror-aux acc′ xs′*); *llist-all2 P acc acc′*; *lfinite acc* ⟧
⟹ *llist-all2 P xs xs′*
⟨*proof*⟩

lemma *llist-all2-lmirrorI*:
llist-all2 P xs ys ⟹ *llist-all2 P* (*lmirror xs*) (*lmirror ys*)
⟨*proof*⟩

lemma *llist-all2-lmirrorD*:
llist-all2 P (*lmirror xs*) (*lmirror ys*) ⟹ *llist-all2 P xs ys*
⟨*proof*⟩

lemma *llist-all2-lmirror* [*simp*]:
llist-all2 P (*lmirror xs*) (*lmirror ys*) ⟷ *llist-all2 P xs ys*
⟨*proof*⟩

lemma *lmirror-parametric* [*transfer-rule*]:
includes *lifting-syntax*
shows (*llist-all2 A* ===> *llist-all2 A*) *lmirror lmirror*
⟨*proof*⟩

end

16 The Hamming stream defined as a least fixpoint

theory *Hamming-Stream* **imports**
../Coinductive-List
HOL−Computational-Algebra.Primes
begin

lemma *infinity-inf-enat* [*simp*]:
fixes $n :: enat$
shows $\infty \sqcap n = n\ n \sqcap \infty = n$
⟨*proof*⟩

lemma *eSuc-inf-eSuc* [*simp*]: $eSuc\ n \sqcap eSuc\ m = eSuc\ (n \sqcap m)$
⟨*proof*⟩

lemma *if-pull2*: $(if\ b\ then\ f\ x\ x'\ else\ f\ y\ y') = f\ (if\ b\ then\ x\ else\ y)\ (if\ b\ then\ x'\ else\ y')$
⟨*proof*⟩

context *ord* **begin**

primcorec *lmerge* :: $'a\ llist \Rightarrow 'a\ llist \Rightarrow 'a\ llist$
where
$lmerge\ xs\ ys =$
$(case\ xs\ of\ LNil \Rightarrow LNil \mid LCons\ x\ xs' \Rightarrow$
$case\ ys\ of\ LNil \Rightarrow LNil \mid LCons\ y\ ys' \Rightarrow$
$if\ lhd\ xs < lhd\ ys\ then\ LCons\ x\ (lmerge\ xs'\ ys)$
$else\ LCons\ y\ (if\ lhd\ ys < lhd\ xs\ then\ lmerge\ xs\ ys'\ else\ lmerge\ xs'\ ys'))$

lemma *lnull-lmerge* [*simp*]: $lnull\ (lmerge\ xs\ ys) \longleftrightarrow (lnull\ xs \vee lnull\ ys)$
⟨*proof*⟩

lemma *lmerge-eq-LNil-iff*: $lmerge\ xs\ ys = LNil \longleftrightarrow (xs = LNil \vee ys = LNil)$
⟨*proof*⟩

lemma *lhd-lmerge*: $[\![\neg\ lnull\ xs; \neg\ lnull\ ys]\!] \Longrightarrow lhd\ (lmerge\ xs\ ys) = (if\ lhd\ xs < lhd\ ys\ then\ lhd\ xs\ else\ lhd\ ys)$
⟨*proof*⟩

lemma *ltl-lmerge*:
$[\![\neg\ lnull\ xs; \neg\ lnull\ ys]\!] \Longrightarrow$
$ltl\ (lmerge\ xs\ ys) =$
$(if\ lhd\ xs < lhd\ ys\ then\ lmerge\ (ltl\ xs)\ ys$
$else\ if\ lhd\ ys < lhd\ xs\ then\ lmerge\ xs\ (ltl\ ys)$
$else\ lmerge\ (ltl\ xs)\ (ltl\ ys))$
⟨*proof*⟩

declare *lmerge.sel* [*simp del*]

lemma *lmerge-simps*:
lmerge (*LCons x xs*) (*LCons y ys*) =
(*if x* < *y then LCons x* (*lmerge xs* (*LCons y ys*))
else if y < *x then LCons y* (*lmerge* (*LCons x xs*) *ys*)
else LCons y (*lmerge xs ys*))
⟨*proof*⟩

lemma *lmerge-LNil* [*simp*]:
lmerge LNil ys = *LNil*
lmerge xs LNil = *LNil*
⟨*proof*⟩

lemma *lprefix-lmergeI*:
⟦ *lprefix xs xs*′; *lprefix ys ys*′ ⟧
⟹ *lprefix* (*lmerge xs ys*) (*lmerge xs*′ *ys*′)
⟨*proof*⟩

lemma [*partial-function-mono*]:
assumes *F*: *mono-llist F* **and** *G*: *mono-llist G*
shows *mono-llist* (λ*f*. *lmerge* (*F f*) (*G f*))
⟨*proof*⟩

lemma *in-lset-lmergeD*: *x* ∈ *lset* (*lmerge xs ys*) ⟹ *x* ∈ *lset xs* ∨ *x* ∈ *lset ys*
⟨*proof*⟩

lemma *lset-lmerge*: *lset* (*lmerge xs ys*) ⊆ *lset xs* ∪ *lset ys*
⟨*proof*⟩

lemma *lfinite-lmergeD*: *lfinite* (*lmerge xs ys*) ⟹ *lfinite xs* ∨ *lfinite ys*
⟨*proof*⟩

lemma fixes *F*
defines *F* ≡ λ*lmerge* (*xs*, *ys*). *case xs of LNil* ⇒ *LNil* | *LCons x xs*′ ⇒ *case ys of LNil* ⇒ *LNil* | *LCons y ys*′ ⇒ (*if x* < *y then LCons x* (*curry lmerge xs*′ *ys*) *else if y* < *x then LCons y* (*curry lmerge xs ys*′) *else LCons y* (*curry lmerge xs*′ *ys*′))
shows *lmerge-conv-fixp*: *lmerge* ≡ *curry* (*ccpo.fixp* (*fun-lub lSup*) (*fun-ord lprefix*) *F*) (**is** *?lhs* ≡ *?rhs*)
and *lmerge-mono*: *mono-llist* (λ*lmerge*. *F lmerge xs*) (**is** *?mono xs*)
⟨*proof*⟩

lemma *monotone-lmerge*: *monotone* (*rel-prod lprefix lprefix*) *lprefix* (*case-prod lmerge*)
⟨*proof*⟩

lemma *mono2mono-lmerge1* [*THEN llist.mono2mono*, *cont-intro*, *simp*]:
shows *monotone-lmerge1*: *monotone lprefix lprefix* (λ*xs*. *lmerge xs ys*)
⟨*proof*⟩

lemma *mono2mono-lmerge2* [*THEN llist.mono2mono*, *cont-intro*, *simp*]:
  shows *monotone-lmerge2*: *monotone lprefix lprefix* (λ*ys*. *lmerge xs ys*)
⟨*proof*⟩

lemma *mcont-lmerge*: *mcont* (*prod-lub lSup lSup*) (*rel-prod lprefix lprefix*) *lSup lprefix* (*case-prod lmerge*)
⟨*proof*⟩

lemma *mcont2mcont-lmerge1* [*THEN llist.mcont2mcont*, *cont-intro*, *simp*]:
  shows *mcont-lmerge1*: *mcont lSup lprefix lSup lprefix* (λ*xs*. *lmerge xs ys*)
⟨*proof*⟩

lemma *mcont2mcont-lmerge2* [*THEN llist.mcont2mcont*, *cont-intro*, *simp*]:
  shows *mcont-lmerge2*: *mcont lSup lprefix lSup lprefix* (λ*ys*. *lmerge xs ys*)
⟨*proof*⟩

lemma *lfinite-lmergeI* [*simp*]: ⟦ *lfinite xs*; *lfinite ys* ⟧ ⟹ *lfinite* (*lmerge xs ys*)
⟨*proof*⟩

lemma *linfinite-lmerge* [*simp*]: ⟦ ¬ *lfinite xs*; ¬ *lfinite ys* ⟧ ⟹ ¬ *lfinite* (*lmerge xs ys*)
⟨*proof*⟩

lemma *llength-lmerge-above*: *llength xs* ⊓ *llength ys* ≤ *llength* (*lmerge xs ys*)
⟨*proof*⟩

end

context *linorder* **begin**

lemma *in-lset-lmergeI1*:
  ⟦ *x* ∈ *lset xs*; *lsorted xs*; ¬ *lfinite ys*; ∃ *y*∈*lset ys*. *x* ≤ *y* ⟧
  ⟹ *x* ∈ *lset* (*lmerge xs ys*)
⟨*proof*⟩

lemma *in-lset-lmergeI2*:
  ⟦ *x* ∈ *lset ys*; *lsorted ys*; ¬ *lfinite xs*; ∃ *y*∈*lset xs*. *x* ≤ *y* ⟧
  ⟹ *x* ∈ *lset* (*lmerge xs ys*)
⟨*proof*⟩

lemma *lsorted-lmerge*: ⟦ *lsorted xs*; *lsorted ys* ⟧ ⟹ *lsorted* (*lmerge xs ys*)
⟨*proof*⟩

lemma *ldistinct-lmerge*:
  ⟦ *lsorted xs*; *lsorted ys*; *ldistinct xs*; *ldistinct ys* ⟧
  ⟹ *ldistinct* (*lmerge xs ys*)
⟨*proof*⟩

end

partial-function (*llist*) *hamming′* :: *unit* ⇒ *nat llist*
where
hamming′ - =
LCons 1 (*lmerge* (*lmap* ((∗) *2*) (*hamming′* ())) (*lmerge* (*lmap* ((∗) *3*) (*hamming′* ())) (*lmap* ((∗) *5*) (*hamming′* ()))))

definition *hamming* :: *nat llist*
where *hamming* = *hamming′* ()

lemma *lnull-hamming* [*simp*]: ¬ *lnull hamming*
⟨*proof*⟩

lemma *hamming-eq-LNil-iff* [*simp*]: *hamming* = *LNil* ⟷ *False*
⟨*proof*⟩

lemma *lhd-hamming* [*simp*]: *lhd hamming* = *1*
⟨*proof*⟩

lemma *ltl-hamming* [*simp*]:
ltl hamming = *lmerge* (*lmap* ((∗) *2*) *hamming*) (*lmerge* (*lmap* ((∗) *3*) *hamming*) (*lmap* ((∗) *5*) *hamming*))
⟨*proof*⟩

lemma *hamming-unfold*:
hamming = *LCons 1* (*lmerge* (*lmap* ((∗) *2*) *hamming*) (*lmerge* (*lmap* ((∗) *3*) *hamming*) (*lmap* ((∗) *5*) *hamming*)))
⟨*proof*⟩

definition *smooth* :: *nat* ⇒ *bool*
where *smooth n* ⟷ (∀ *p*. *prime p* ⟶ *p dvd n* ⟶ *p* ≤ *5*)

lemma *smooth-0* [*simp*]: ¬ *smooth 0*
⟨*proof*⟩

lemma *smooth-Suc0* [*simp*]: *smooth* (*Suc 0*)
⟨*proof*⟩

lemma *smooth-gt0*: *smooth n* ⟹ *n* > *0*
⟨*proof*⟩

lemma *smooth-ge-Suc0*: *smooth n* ⟹ *n* ≥ *Suc 0*
⟨*proof*⟩

lemma *prime-nat-dvdD*: *prime p* ⟹ (*n* :: *nat*) *dvd p* ⟹ *n* = *1* ∨ *n* = *p*
⟨*proof*⟩

lemma *smooth-times* [*simp*]: *smooth* (*x* ∗ *y*) ⟷ *smooth x* ∧ *smooth y*

⟨*proof*⟩

lemma *smooth2* [*simp*]: *smooth 2*
⟨*proof*⟩

lemma *smooth3* [*simp*]: *smooth 3*
⟨*proof*⟩

lemma *smooth5* [*simp*]: *smooth 5*
⟨*proof*⟩

lemma *hamming-in-smooth*: *lset hamming* ⊆ {*n*. *smooth n*}
⟨*proof*⟩

lemma *lfinite-hamming* [*simp*]: ¬ *lfinite hamming*
⟨*proof*⟩

lemma *lsorted-hamming* [*simp*]: *lsorted hamming*
and *ldistinct-hamming* [*simp*]: *ldistinct hamming*
⟨*proof*⟩

lemma *smooth-hamming*:
assumes *smooth n*
shows *n* ∈ *lset hamming*
⟨*proof*⟩

corollary *hamming-smooth*: *lset hamming* = {*n*. *smooth n*}
⟨*proof*⟩

lemma *hamming-THE*:
(*THE xs*. *lsorted xs* ∧ *ldistinct xs* ∧ *lset xs* = {*n*. *smooth n*}) = *hamming*
⟨*proof*⟩

end

17 Manual construction of a resumption codatatype

theory *Resumption* **imports**
HOL−Library.Old-Datatype
begin

This theory defines the following codatatype:

```
codatatype ('a,'b,'c,'d) resumption =
    Terminal 'a
  | Linear 'b "('a,'b,'c,'d) resumption"
  | Branch 'c "'d => ('a,'b,'c,'d) resumption"
```

17.1 Auxiliary definitions and lemmata similar to *HOL−Library.Old-Datatype*

lemma *Lim-mono*: $(\bigwedge d.\ rs\ d \subseteq rs'\ d) \Longrightarrow Old\text{-}Datatype.Lim\ rs \subseteq Old\text{-}Datatype.Lim\ rs'$
⟨*proof*⟩

lemma *Lim-UN1*: $Old\text{-}Datatype.Lim\ (\lambda x.\ \bigcup y.\ f\ x\ y) = (\bigcup y.\ Old\text{-}Datatype.Lim\ (\lambda x.\ f\ x\ y))$
⟨*proof*⟩

Inverse for *Old-Datatype.Lim* like *Old-Datatype.Split* and *Old-Datatype.Case* for *Scons* and *In0/In1*

definition $DTBranch :: (('b \Rightarrow ('a,\ 'b)\ Old\text{-}Datatype.dtree) \Rightarrow 'c) \Rightarrow ('a,\ 'b)\ Old\text{-}Datatype.dtree \Rightarrow 'c$
where $DTBranch\ f\ M = (THE\ u.\ \exists x.\ M = Old\text{-}Datatype.Lim\ x \wedge u = f\ x)$

lemma *DTBranch-Lim* [*simp*]: $DTBranch\ f\ (Old\text{-}Datatype.Lim\ M) = f\ M$
⟨*proof*⟩

Lemmas for *ntrunc* and *Old-Datatype.Lim*

lemma *ndepth-Push-Node-Inl-aux*:
$case\text{-}nat\ (Inl\ n)\ f\ k = Inr\ 0 \Longrightarrow Suc\ (LEAST\ x.\ f\ x = Inr\ 0) <= k$
⟨*proof*⟩

lemma *ndepth-Push-Node-Inl*:
$ndepth\ (Push\text{-}Node\ (Inl\ a)\ n) = Suc\ (ndepth\ n)$
⟨*proof*⟩

lemma *ntrunc-Lim* [*simp*]: $ntrunc\ (Suc\ k)\ (Old\text{-}Datatype.Lim\ rs) = Old\text{-}Datatype.Lim\ (\lambda x.\ ntrunc\ k\ (rs\ x))$
⟨*proof*⟩

17.2 Definition for the codatatype universe

Constructors

definition $TERMINAL :: 'a \Rightarrow ('c + 'b + 'a,\ 'd)\ Old\text{-}Datatype.dtree$
where $TERMINAL\ a = In0\ (Old\text{-}Datatype.Leaf\ (Inr\ (Inr\ a)))$

definition $LINEAR :: 'b \Rightarrow ('c + 'b + 'a,\ 'd)\ Old\text{-}Datatype.dtree \Rightarrow ('c + 'b + 'a,\ 'd)\ Old\text{-}Datatype.dtree$
where $LINEAR\ b\ r = In1\ (In0\ (Scons\ (Old\text{-}Datatype.Leaf\ (Inr\ (Inl\ b)))\ r))$

definition $BRANCH :: 'c \Rightarrow ('d \Rightarrow ('c + 'b + 'a,\ 'd)\ Old\text{-}Datatype.dtree) \Rightarrow ('c + 'b + 'a,\ 'd)\ Old\text{-}Datatype.dtree$
where $BRANCH\ c\ rs = In1\ (In1\ (Scons\ (Old\text{-}Datatype.Leaf\ (Inl\ c))\ (Old\text{-}Datatype.Lim\ rs)))$

case operator

definition *case-RESUMPTION* :: $('a \Rightarrow 'e) \Rightarrow ('b \Rightarrow (('c + 'b + 'a, 'd)\ Old\text{-}Datatype.dtree) \Rightarrow 'e) \Rightarrow ('c \Rightarrow ('d \Rightarrow ('c + 'b + 'a, 'd)\ Old\text{-}Datatype.dtree) \Rightarrow 'e) \Rightarrow ('c + 'b + 'a, 'd)\ Old\text{-}Datatype.dtree \Rightarrow 'e$
where
case-RESUMPTION t l br =
Old-Datatype.Case (*t o inv* (*Old-Datatype.Leaf o Inr o Inr*))
(*Old-Datatype.Case* (*Old-Datatype.Split* (λ*x*. *l* (*inv* (*Old-Datatype.Leaf o Inr o Inl*) *x*)))
(*Old-Datatype.Split* (λ*x*. *DTBranch* (*br* (*inv* (*Old-Datatype.Leaf o Inl*) *x*)))))

lemma [*iff*]:
shows *TERMINAL-not-LINEAR*: $TERMINAL\ a \neq LINEAR\ b\ r$
and *LINEAR-not-TERMINAL*: $LINEAR\ b\ R \neq TERMINAL\ a$
and *TERMINAL-not-BRANCH*: $TERMINAL\ a \neq BRANCH\ c\ rs$
and *BRANCH-not-TERMINAL*: $BRANCH\ c\ rs \neq TERMINAL\ a$
and *LINEAR-not-BRANCH*: $LINEAR\ b\ r \neq BRANCH\ c\ rs$
and *BRANCH-not-LINEAR*: $BRANCH\ c\ rs \neq LINEAR\ b\ r$
and *TERMINAL-inject*: $TERMINAL\ a = TERMINAL\ a' \longleftrightarrow a = a'$
and *LINEAR-inject*: $LINEAR\ b\ r = LINEAR\ b'\ r' \longleftrightarrow b = b' \wedge r = r'$
and *BRANCH-inject*: $BRANCH\ c\ rs = BRANCH\ c'\ rs' \longleftrightarrow c = c' \wedge rs = rs'$
⟨*proof*⟩

lemma *case-RESUMPTION-simps* [*simp*]:
shows *case-RESUMPTION-TERMINAL*: *case-RESUMPTION t l br* (*TERMINAL a*) = *t a*
and *case-RESUMPTION-LINEAR*: *case-RESUMPTION t l br* (*LINEAR b r*) = *l b r*
and *case-RESUMPTION-BRANCH*: *case-RESUMPTION t l br* (*BRANCH c rs*) = *br c rs*
⟨*proof*⟩

lemma *LINEAR-mono*: $r \subseteq r' \Longrightarrow LINEAR\ b\ r \subseteq LINEAR\ b\ r'$
⟨*proof*⟩

lemma *BRANCH-mono*: $(\bigwedge d.\ rs\ d \subseteq rs'\ d) \Longrightarrow BRANCH\ c\ rs \subseteq BRANCH\ c\ rs'$
⟨*proof*⟩

lemma *LINEAR-UN*: $LINEAR\ b\ (\bigcup x.\ f\ x) = (\bigcup x.\ LINEAR\ b\ (f\ x))$
⟨*proof*⟩

lemma *BRANCH-UN*: $BRANCH\ b\ (\lambda d.\ \bigcup x.\ f\ d\ x) = (\bigcup x.\ BRANCH\ b\ (\lambda d.\ f\ d\ x))$
⟨*proof*⟩

The codatatype universe

coinductive-set *resumption* :: $('c + 'b + 'a, 'd)\ Old\text{-}Datatype.dtree\ set$
where
resumption-TERMINAL:

TERMINAL a ∈ resumption
| *resumption-LINEAR*:
r ∈ resumption ⟹ LINEAR b r ∈ resumption
| *resumption-BRANCH*:
(⋀d. rs d ∈ resumption) ⟹ BRANCH c rs ∈ resumption

17.3 Definition of the codatatype as a type

typedef *('a,'b,'c,'d) resumption = resumption :: ('c + 'b + 'a, 'd) Old-Datatype.dtree set*
⟨*proof*⟩

Constructors

definition *Terminal :: 'a ⇒ ('a,'b,'c,'d) resumption*
where *Terminal a = Abs-resumption (TERMINAL a)*

definition *Linear :: 'b ⇒ ('a,'b,'c,'d) resumption ⇒ ('a,'b,'c,'d) resumption*
where *Linear b r = Abs-resumption (LINEAR b (Rep-resumption r))*

definition *Branch :: 'c ⇒ ('d ⇒ ('a,'b,'c,'d) resumption) ⇒ ('a,'b,'c,'d) resumption*
where *Branch c rs = Abs-resumption (BRANCH c (λd. Rep-resumption (rs d)))*

lemma [*iff*]:
shows *Terminal-not-Linear*: *Terminal a ≠ Linear b r*
and *Linear-not-Terminal*: *Linear b R ≠ Terminal a*
and *Termina-not-Branch*: *Terminal a ≠ Branch c rs*
and *Branch-not-Terminal*: *Branch c rs ≠ Terminal a*
and *Linear-not-Branch*: *Linear b r ≠ Branch c rs*
and *Branch-not-Linear*: *Branch c rs ≠ Linear b r*
and *Terminal-inject*: *Terminal a = Terminal a' ⟷ a = a'*
and *Linear-inject*: *Linear b r = Linear b' r' ⟷ b = b' ∧ r = r'*
and *Branch-inject*: *Branch c rs = Branch c' rs' ⟷ c = c' ∧ rs = rs'*
⟨*proof*⟩

lemma *Rep-resumption-constructors*:
shows *Rep-resumption-Terminal*: *Rep-resumption (Terminal a) = TERMINAL a*
and *Rep-resumption-Linear*: *Rep-resumption (Linear b r) = LINEAR b (Rep-resumption r)*
and *Rep-resumption-Branch*: *Rep-resumption (Branch c rs) = BRANCH c (λd. Rep-resumption (rs d))*
⟨*proof*⟩

Case operator

definition *case-resumption :: ('a ⇒ 'e) ⇒ ('b ⇒ ('a,'b,'c,'d) resumption ⇒ 'e) ⇒ ('c ⇒ ('d ⇒ ('a,'b,'c,'d) resumption) ⇒ 'e) ⇒ ('a,'b,'c,'d) resumption ⇒ 'e*
where [*code del*]:
case-resumption t l br r =

case-RESUMPTION t (λ*b r*. *l b* (*Abs-resumption r*)) (λ*c rs*. *br c* (λ*d*. *Abs-resumption* (*rs d*))) (*Rep-resumption r*)

lemma *case-resumption-simps* [*simp*, *code*]:
shows *case-resumption-Terminal*: *case-resumption t l br* (*Terminal a*) = *t a*
and *case-resumption-Linear*: *case-resumption t l br* (*Linear b r*) = *l b r*
and *case-resumption-Branch*: *case-resumption t l br* (*Branch c rs*) = *br c rs*
⟨*proof*⟩

declare [[*case-translation case-resumption Terminal Linear Branch*]]

lemma *case-resumption-cert*:
assumes *CASE* ≡ *case-resumption t l br*
shows (*CASE* (*Terminal a*) ≡ *t a*) &&& (*CASE* (*Linear b r*) ≡ *l b r*) &&& (*CASE* (*Branch c rs*) ≡ *br c rs*)
⟨*proof*⟩

code-datatype *Terminal Linear Branch*

⟨*ML*⟩

lemma *resumption-exhaust* [*cases type*: *resumption*]:
obtains (*Terminal*) *a* **where** *x* = *Terminal a*
| (*Linear*) *b r* **where** *x* = *Linear b r*
| (*Branch*) *c rs* **where** *x* = *Branch c rs*
⟨*proof*⟩

lemma *resumption-split*:
P (*case-resumption t l br r*) ⟷
(∀ *a*. *r* = *Terminal a* ⟶ *P* (*t a*)) ∧
(∀ *b r'*. *r* = *Linear b r'* ⟶ *P* (*l b r'*)) ∧
(∀ *c rs*. *r* = *Branch c rs* ⟶ *P* (*br c rs*))
⟨*proof*⟩

lemma *resumption-split-asm*:
P (*case-resumption t l br r*) ⟷
¬ ((∃ *a*. *r* = *Terminal a* ∧ ¬ *P* (*t a*)) ∨
(∃ *b r'*. *r* = *Linear b r'* ∧ ¬ *P* (*l b r'*)) ∨
(∃ *c rs*. *r* = *Branch c rs* ∧ ¬ *P* (*br c rs*)))
⟨*proof*⟩

lemmas *resumption-splits* = *resumption-split resumption-split-asm*

corecursion operator

datatype (*dead 'a*, *dead 'b*, *dead 'c*, *dead 'd*, *dead 'e*) *resumption-corec* =
Terminal-corec 'a
| *Linear-corec 'b 'e*
| *Branch-corec 'c 'd* ⇒ *'e*
| *Resumption-corec* (*'a*, *'b*, *'c*, *'d*) *resumption*

primrec *RESUMPTION-corec-aux* :: *nat* ⇒ (*'e* ⇒ (*'a*,*'b*,*'c*,*'d*,*'e*) *resumption-corec*) ⇒ *'e* ⇒ (*'c* + *'b* + *'a*,*'d*) *Old-Datatype.dtree*
where
RESUMPTION-corec-aux 0 f e = {}
| *RESUMPTION-corec-aux* (*Suc n*) *f e* =
(*case f e of Terminal-corec a* ⇒ *TERMINAL a*
| *Linear-corec b e'* ⇒ *LINEAR b* (*RESUMPTION-corec-aux n f e'*)
| *Branch-corec c es* ⇒ *BRANCH c* (λ*d*. *RESUMPTION-corec-aux n f* (*es d*))
| *Resumption-corec r* ⇒ *Rep-resumption r*)

definition *RESUMPTION-corec* :: (*'e* ⇒ (*'a*,*'b*,*'c*,*'d*,*'e*) *resumption-corec*) ⇒ *'e* ⇒ (*'c* + *'b* + *'a*,*'d*) *Old-Datatype.dtree*
where
RESUMPTION-corec f e = (⋃ *n*. *RESUMPTION-corec-aux n f e*)

lemma *RESUMPTION-corec* [*nitpick-simp*]:
RESUMPTION-corec f e =
(*case f e of Terminal-corec a* ⇒ *TERMINAL a*
| *Linear-corec b e'* ⇒ *LINEAR b* (*RESUMPTION-corec f e'*)
| *Branch-corec c es* ⇒ *BRANCH c* (λ*d*. *RESUMPTION-corec f* (*es d*))
| *Resumption-corec r* ⇒ *Rep-resumption r*)
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *RESUMPTION-corec-type*: *RESUMPTION-corec f e* ∈ *resumption*
⟨*proof*⟩

corecursion operator for the resumption type

definition *resumption-corec* :: (*'e* ⇒ (*'a*,*'b*,*'c*,*'d*,*'e*) *resumption-corec*) ⇒ *'e* ⇒ (*'a*,*'b*,*'c*,*'d*) *resumption*
where
resumption-corec f e = *Abs-resumption* (*RESUMPTION-corec f e*)

lemma *resumption-corec*:
resumption-corec f e =
(*case f e of Terminal-corec a* ⇒ *Terminal a*
| *Linear-corec b e'* ⇒ *Linear b* (*resumption-corec f e'*)
| *Branch-corec c es* ⇒ *Branch c* (λ*d*. *resumption-corec f* (*es d*))
| *Resumption-corec r* ⇒ *r*)
⟨*proof*⟩

Equality as greatest fixpoint

coinductive *Eq-RESUMPTION* :: (*'c*+*'b*+*'a*, *'d*) *Old-Datatype.dtree* ⇒ (*'c*+*'b*+*'a*, *'d*) *Old-Datatype.dtree* ⇒ *bool*
where
EqTERMINAL: *Eq-RESUMPTION* (*TERMINAL a*) (*TERMINAL a*)
| *EqLINEAR*: *Eq-RESUMPTION r r'* ⟹ *Eq-RESUMPTION* (*LINEAR b r*)

(*LINEAR b r'*)
| *EqBRANCH*: (⋀*d*. *Eq-RESUMPTION* (*rs d*) (*rs' d*)) ⟹ *Eq-RESUMPTION* (*BRANCH c rs*) (*BRANCH c rs'*)

lemma *Eq-RESUMPTION-implies-ntrunc-equality*:
Eq-RESUMPTION r r' ⟹ *ntrunc k r* = *ntrunc k r'*
⟨*proof*⟩

lemma *Eq-RESUMPTION-refl*:
assumes *r* ∈ *resumption*
shows *Eq-RESUMPTION r r*
⟨*proof*⟩

lemma *Eq-RESUMPTION-into-resumption*:
assumes *Eq-RESUMPTION r r*
shows *r* ∈ *resumption*
⟨*proof*⟩

lemma *Eq-RESUMPTION-eq*:
Eq-RESUMPTION r r' ⟷ *r* = *r'* ∧ *r* ∈ *resumption*
⟨*proof*⟩

lemma *Eq-RESUMPTION-I* [*consumes 1*, *case-names Eq-RESUMPTION*, *case-conclusion Eq-RESUMPTION EqTerminal EqLinear EqBranch*]:
assumes *X r r'*
and *step*: ⋀*r r'*. *X r r'* ⟹
(∃ *a*. *r* = *TERMINAL a* ∧ *r'* = *TERMINAL a*) ∨
(∃ *R R' b*. *r* = *LINEAR b R* ∧ *r'* = *LINEAR b R'* ∧ (*X R R'* ∨ *Eq-RESUMPTION R R'*)) ∨
(∃ *rs rs' c*. *r* = *BRANCH c rs* ∧ *r'* = *BRANCH c rs'* ∧ (∀ *d*. *X* (*rs d*) (*rs' d*) ∨ *Eq-RESUMPTION* (*rs d*) (*rs' d*)))
shows *r* = *r'*
⟨*proof*⟩

lemma *resumption-equalityI* [*consumes 1*, *case-names Eq-resumption*, *case-conclusion Eq-resumption EqTerminal EqLinear EqBranch*]:
assumes *X r r'*
and *step*: ⋀*r r'*. *X r r'* ⟹
(∃ *a*. *r* = *Terminal a* ∧ *r'* = *Terminal a*) ∨
(∃ *R R' b*. *r* = *Linear b R* ∧ *r'* = *Linear b R'* ∧ (*X R R'* ∨ *R* = *R'*)) ∨
(∃ *rs rs' c*. *r* = *Branch c rs* ∧ *r'* = *Branch c rs'* ∧ (∀ *d*. *X* (*rs d*) (*rs' d*) ∨ *rs d* = *rs' d*))
shows *r* = *r'*
⟨*proof*⟩

Finality of *resumption*: Uniqueness of functions defined by corecursion.

lemma *equals-RESUMPTION-corec*:
assumes *h*: ⋀*x*. *h x* = (*case f x of Terminal-corec a* ⇒ *TERMINAL a*
| *Linear-corec b x'* ⇒ *LINEAR b* (*h x'*)

| *Branch-corec c xs* ⇒ *BRANCH c* (λ*d*. *h* (*xs d*))
| *Resumption-corec r* ⇒ *Rep-resumption r*)
shows *h* = *RESUMPTION-corec f*
⟨*proof*⟩

lemma *equals-resumption-corec*:
assumes *h*: ⋀*x*. *h x* = (*case f x of Terminal-corec a* ⇒ *Terminal a*
| *Linear-corec b x'* ⇒ *Linear b* (*h x'*)
| *Branch-corec c xs* ⇒ *Branch c* (λ*d*. *h* (*xs d*))
| *Resumption-corec r* ⇒ *r*)
shows *h* = *resumption-corec f*
⟨*proof*⟩

end

theory *Coinductive-Examples* **imports**
LList-CCPO-Topology
TLList-CCPO-Examples
Koenigslemma
LMirror
Hamming-Stream
Resumption
begin

end