

Coinductive

Andreas Lochbihler
with contributions by Johannes Hözl

March 19, 2025

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	2
1.1	Case operator	3
1.2	Corecursion for <i>enat</i>	4
1.3	Less as greatest fixpoint	7
1.4	Equality as greatest fixpoint	9
1.5	Uniqueness of corecursion	10
1.6	Setup for partial_function	10
1.7	Misc.	16
2	Coinductive lists and their operations	17
2.1	Auxiliary lemmata	18
2.2	Type definition	18
2.3	Properties of predefined functions	20
2.4	The subset of finite lazy lists <i>lfinite</i>	22

2.5	Concatenating two lists: <i>lappend</i>	23
2.6	The prefix ordering on lazy lists: <i>lprefix</i>	26
2.7	Setup for <i>partial_function</i>	29
2.8	Monotonicity and continuity of already defined functions	35
2.9	More function definitions	38
2.10	Converting ordinary lists to lazy lists: <i>llist-of</i>	41
2.11	Converting finite lazy lists to ordinary lists: <i>list-of</i>	43
2.12	The length of a lazy list: <i>llength</i>	44
2.13	Taking and dropping from lazy lists: <i>ltake</i> , <i>ldropn</i> , and <i>ldrop</i>	46
2.14	Taking the <i>n</i> -th element of a lazy list: <i>lnth</i>	57
2.15	<i>iterates</i>	62
2.16	More on the prefix ordering on lazy lists: (\sqsubseteq) and <i>lstrict-prefix</i>	64
2.17	Length of the longest common prefix	68
2.18	Zipping two lazy lists to a lazy list of pairs <i>lzip</i>	69
2.19	Taking and dropping from a lazy list: <i>ltakeWhile</i> and <i>ldropWhile</i>	74
2.20	<i>llist-all2</i>	81
2.21	The last element <i>llast</i>	87
2.22	Distinct lazy lists <i>ldistinct</i>	89
2.23	Sortedness <i>lsorted</i>	93
2.24	Lexicographic order on lazy lists: <i>llexord</i>	96
2.25	The filter functional on lazy lists: <i>lfilter</i>	103
2.26	Concatenating all lazy lists in a lazy list: <i>lconcat</i>	110
2.27	Sublist view of a lazy list: <i>lnths</i>	119
2.28	<i>lsum-list</i>	123
2.29	Alternative view on ' <i>a llist</i> ' as datatype with constructors <i>llist-of</i> and <i>inf-llist</i>	124
2.30	Setup for lifting and transfer	126
2.30.1	Relator and predicator properties	126
2.30.2	Transfer rules for the Transfer package	126
3	Instantiation of the order type classes for lazy lists	130
3.1	Instantiation of the order type class	130
3.2	Prefix ordering as a lower semilattice	131
4	Infinite lists as a codatatype	133
4.1	Lemmas about operations from <i>HOL-Library.Stream</i>	134
4.2	Link ' <i>a stream</i> to ' <i>a llist</i>	134
4.3	Link ' <i>a stream</i> with <i>nat</i> \Rightarrow ' <i>a</i>	139
4.4	Function iteration <i>siterate</i> and <i>sconst</i>	140
4.5	Counting elements	140
4.6	First index of an element	142
4.7	<i>stakeWhile</i>	144

5	Terminated coinductive lists and their operations	144
5.1	Auxiliary lemmas	144
5.2	Type definition	145
5.3	Code generator setup	146
5.4	Connection with ' <i>a llist</i> '	147
5.5	Library function definitions	152
5.6	<i>tfinite</i>	152
5.7	The terminal element <i>terminal</i>	152
5.8	<i>tmap</i>	153
5.9	Appending two terminated lazy lists <i>tappend</i>	153
5.10	Appending a terminated lazy list to a lazy list <i>lappendt</i>	154
5.11	Filtering terminated lazy lists <i>tfilter</i>	154
5.12	Concatenating a terminated lazy list of lazy lists <i>tconcat</i>	155
5.13	<i>tllist-all2</i>	156
5.14	From a terminated lazy list to a lazy list <i>llist-of-tllist</i>	159
5.15	The nth element of a terminated lazy list <i>tnth</i>	160
5.16	The length of a terminated lazy list <i>tlength</i>	160
5.17	<i>tdropn</i>	161
5.18	<i>tset</i>	161
5.19	Setup for Lifting/Transfer	162
5.19.1	Relator and predicator properties	162
5.19.2	Transfer rules for the Transfer package	162
6	Setup for Isabelle's quotient package for lazy lists	166
6.1	Rules for the Quotient package	166
7	Setup for Isabelle's quotient package for terminated lazy lists	169
7.1	Rules for the Quotient package	169
8	Code generator setup to implement lazy lists lazily	174
8.1	Lazy lists	174
9	Code generator setup to implement terminated lazy lists lazily	181
10	CCPO topologies	185
10.1	The filter <i>at'</i>	186
10.2	The type class <i>ccpo-topology</i>	187
10.3	Instances for <i>ccpo-topologys</i> and continuity theorems	190
11	A CCPO topology on lazy lists with examples	192
11.1	Continuity and closedness of predefined constants	193
11.2	Define <i>lfilter</i> as continuous extension	199
11.3	Define <i>lconcat</i> as continuous extension	200

11.4 Define <i>ldropWhile</i> as continuous extension	201
11.5 Define <i>ldrop</i> as continuous extension	202
11.6 Define more functions on lazy lists as continuous extensions .	203
12 Cpo structure for terminated lazy lists	210
12.1 The cpo structure	211
12.2 Continuity of predefined constants	217
12.3 Definition of recursive functions	220
13 Example definitions using the CCPo structure on terminated lazy lists	221
14 Example: Koenig's lemma	223
15 Definition of the function lmirror	229
16 The Hamming stream defined as a least fixpoint	233
17 Manual construction of a resumption codatatype	240
17.1 Auxiliary definitions and lemmata similar to <i>HOL-Library.Old-Datatype</i>	241
17.2 Definition for the codatatype universe	241
17.3 Definition of the codatatype as a type	243

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
by(simp add: inj-on-def)

lemma Sup-range-enat [simp]: Sup (range enat) = ∞
by(auto dest: finite-imageD simp add: Sup-enat-def)

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
by(cases b) auto

lemma enat-the-enat: n ≠ ∞ ⟹ enat (the-enat n) = n
by auto

lemma enat-min-eq-0-iff:
  fixes a b :: enat
  shows min a b = 0 ⟷ a = 0 ∨ b = 0
by(auto simp add: min-def)

lemma enat-le-plus-same: x ≤ (x :: enat) + y x ≤ y + x
by(auto simp add: less-eq-enat-def plus-enat-def split: enat.split)

lemma the-enat-0 [simp]: the-enat 0 = 0
by(simp add: zero-enat-def)

lemma the-enat-eSuc: n ≠ ∞ ⟹ the-enat (eSuc n) = Suc (the-enat n)
by(cases n)(simp-all add: eSuc-enat)

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
  show enat-set ⊆ UNIV by blast
  show UNIV ⊆ enat-set
  proof
    fix x :: enat
    assume x ∈ UNIV
```

```

thus  $x \in \text{enat-set}$ 
proof coinduct
  case ( $\text{enat-set } x$ )
  show ?case
  proof(cases  $x = 0$ )
    case True thus ?thesis by simp
  next
    case False
    then obtain  $n$  where  $x = eSuc n$ 
      by(cases  $x$ )(fastforce simp add: eSuc-def zero-enat-def gr0-conv-Suc
                           split: enat.splits) +
      thus ?thesis by auto
    qed
  qed
  qed
qed

```

1.1 Case operator

```

lemma enat-coexhaust:
  obtains (0)  $n = 0$ 
  | ( $eSuc$ )  $n'$  where  $n = eSuc n'$ 
proof -
  have  $n \in \text{enat-set}$  by auto
  thus thesis by cases (erule that) +
qed

locale co begin

free-constructors (plugins del: code) case-enat for
   $0::\text{enat}$ 
  |  $eSuc$  epred
where
  epred  $0 = 0$ 
  apply (erule enat-coexhaust, assumption)
  apply (rule eSuc-inject)
  by (rule zero-ne-eSuc)

end

lemma enat-cocase-0 [simp]: co.case-enat  $z s 0 = z$ 
by (rule co.enat.case(1))

lemma enat-cocase-eSuc [simp]: co.case-enat  $z s (eSuc n) = s n$ 
by (rule co.enat.case(2))

lemma neq-zero-conv-eSuc:  $n \neq 0 \longleftrightarrow (\exists n'. n = eSuc n')$ 
by(cases  $n$  rule: enat-coexhaust) simp-all

```

```

lemma enat-cocase-cert:
  assumes CASE  $\equiv$  co.case-enat c d
  shows (CASE 0  $\equiv$  c) && (CASE (eSuc n)  $\equiv$  d n)
  using assms by simp-all

lemma enat-cosplit-asm:
  P (co.case-enat c d n) = ( $\neg$  (n = 0  $\wedge$   $\neg$  P c  $\vee$  ( $\exists$  m. n = eSuc m  $\wedge$   $\neg$  P (d m))))
  by (rule co.enat.split-asm)

lemma enat-cosplit:
  P (co.case-enat c d n) = ((n = 0  $\longrightarrow$  P c)  $\wedge$  ( $\forall$  m. n = eSuc m  $\longrightarrow$  P (d m)))
  by (rule co.enat.split)

abbreviation epred :: enat => enat where epred  $\equiv$  co.epred

lemma epred-0 [simp]: epred 0 = 0 by(rule co.enat.sel(1))
lemma epred-eSuc [simp]: epred (eSuc n) = n by(rule co.enat.sel(2))
declare co.enat.collapse[simp]
lemma epred-conv-minus: epred n = n - 1
  by(cases n rule: co.enat.exhaust)(simp-all)

```

1.2 Corecursion for enat

```

lemma case-enat-numeral [simp]: case-enat f i (numeral v) = (let n = numeral v
  in f n)
  by(simp add: numeral-eq-enat)

lemma case-enat-0 [simp]: case-enat f i 0 = f 0
  by(simp add: zero-enat-def)

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)
  by(simp-all add: eSuc-def split: enat.split)

definition epred-numeral :: num  $\Rightarrow$  enat
where [code del]: epred-numeral = enat  $\circ$  pred-numeral

lemma numeral-eq-eSuc: numeral k = eSuc (epred-numeral k)
  by(simp add: numeral-eq-Suc eSuc-def epred-numeral-def numeral-eq-enat)

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)
  by(simp-all add: epred-numeral-def enat-numeral zero-enat-def)

lemma [simp]:

```

```

shows eq-numeral-eSuc: numeral k = eSuc n  $\longleftrightarrow$  epred-numeral k = n
and Suc-eq-numeral: eSuc n = numeral k  $\longleftrightarrow$  n = epred-numeral k
and less-numeral-Suc: numeral k < eSuc n  $\longleftrightarrow$  epred-numeral k < n
and less-eSuc-numeral: eSuc n < numeral k  $\longleftrightarrow$  n < epred-numeral k
and le-numeral-eSuc: numeral k  $\leq$  eSuc n  $\longleftrightarrow$  epred-numeral k  $\leq$  n
and le-eSuc-numeral: eSuc n  $\leq$  numeral k  $\longleftrightarrow$  n  $\leq$  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)
by(simp-all add: numeral-eq-eSuc)

```

```

lemma enat-cocase-numeral [simp]:
  co.case-enat a f (numeral v) = (let pv = epred-numeral v in f pv)
by(simp add: numeral-eq-eSuc)

```

```

lemma enat-cocase-add-eq-if [simp]:
  co.case-enat a f ((numeral v) + n) = (let pv = epred-numeral v in f (pv + n))
by(simp add: numeral-eq-eSuc iadd-Suc)

```

```

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)
by(simp-all add: epred-conv-minus one-enat-def zero-enat-def eSuc-def epred-numeral-def
  numeral-eq-enat split: enat.split)

```

```

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

```

```

lemma epred-iadd1: a  $\neq$  0  $\implies$  epred (a + b) = epred a + b
apply(cases a b rule: enat.exhaust[case-product enat.exhaust])
apply(simp-all add: epred-conv-minus eSuc-def one-enat-def zero-enat-def split:
  enat.splits)
done

```

```

lemma epred-min [simp]: epred (min a b) = min (epred a) (epred b)
by(cases a b rule: enat-coexhaust[case-product enat-coexhaust]) simp-all

```

```

lemma epred-le-epredI: n  $\leq$  m  $\implies$  epred n  $\leq$  epred m
by(cases m n rule: enat-coexhaust[case-product enat-coexhaust]) simp-all

```

```

lemma epred-minus-epred [simp]:
   $m \neq 0 \implies \text{epred } n - \text{epred } m = n - m$ 
by(cases n m rule: enat-coexhaust[case-product enat-coexhaust])(simp-all add: epred-conv-minus)

lemma eSuc-epred:  $n \neq 0 \implies \text{eSuc}(\text{epred } n) = n$ 
by(cases n rule: enat-coexhaust) simp-all

lemma epred-inject:  $\llbracket x \neq 0; y \neq 0 \rrbracket \implies \text{epred } x = \text{epred } y \longleftrightarrow x = y$ 
by(cases x y rule: enat.exhaust[case-product enat.exhaust])(auto simp add: zero-enat-def)

lemma monotone-fun-eSuc[partial-function-mono]:
  monotone (fun-ord ( $\lambda y. x. x \leq y$ )) ( $\lambda y. x. x \leq y$ ) ( $\lambda f. \text{eSuc}(f x)$ )
by (auto simp: monotone-def fun-ord-def)

partial-function (gfp) enat-unfold :: ('a  $\Rightarrow$  bool)  $\Rightarrow$  ('a  $\Rightarrow$  'a)  $\Rightarrow$  'a  $\Rightarrow$  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  $\implies$  enat-unfold stop next a = 0
by(simp add: enat-unfold)

lemma enat-unfold-next:  $\neg \text{stop } a \implies \text{enat-unfold stop next } a = \text{eSuc}(\text{enat-unfold stop next } (next a))$ 
by(simp add: enat-unfold)

lemma enat-unfold-eq-0 [simp]:
  enat-unfold stop next a = 0  $\longleftrightarrow$  stop a
by(simp add: enat-unfold)

lemma epred-enat-unfold [simp]:
  epred(enat-unfold stop next a) = (if stop a then 0 else enat-unfold stop next (next a))
by(simp add: enat-unfold-next)

lemma epred-max: epred(max x y) = max(epred x)(epred y)
by(cases x y rule: enat.exhaust[case-product enat.exhaust]) simp-all

lemma epred-Max:
  assumes finite A A  $\neq \{\}$ 
  shows epred(Max A) = Max(epred `A)
  using assms
  proof induction
    case (insert x A)
    thus ?case by(cases A = {})(simp-all add: epred-max)
  qed simp

lemma finite-imageD2:  $\llbracket \text{finite } (f ` A); \text{inj-on } f (A - B); \text{finite } B \rrbracket \implies \text{finite } A$ 
by (metis Diff-subset finite-Diff2 image-mono inj-on-finite)

```

```

lemma epred-Sup: epred (Sup A) = Sup (epred ` A)
by(auto 4 4 simp add: bot-enat-def Sup-enat-def epred-Max inj-on-def neq-zero-conv-eSuc
dest: finite-imageD2[where B={0}])

```

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 –

```

```

assume m ≤ n

```

```

hence (m, n) ∈ {(m, n)|m n. m ≤ n} by simp

```

```

thus ?thesis

```

```

proof coinduct

```

```

case (Le-enat m n)

```

```

hence m ≤ n by simp

```

```

show ?case

```

```

proof(cases m = 0)

```

```

case True

```

```

hence ?Le-enat-zero by simp

```

```

thus ?thesis ..

```

```

next

```

```

case False

```

```

with <m ≤ n> obtain m' n' where m = eSuc m' n = n' + 1 m' ≤ n'

```

```

by(cases m rule: enat-coexhaust, simp)

```

```

(cases n rule: enat-coexhaust, auto simp add: eSuc-plus-1[symmetric])

```

```

hence ?Le-enat-add by fastforce

```

```

thus ?thesis ..

```

```

qed

```

```

qed

```

```

qed

```

```

lemma Le-enat-imp-ile-enat-k:

```

```

(m, n) ∈ Le-enat ⇒ n < enat l ⇒ m < enat l

```

```

proof(induct l arbitrary: m n)

```

```

case 0

```

```

thus ?case by(simp add: zero-enat-def[symmetric])

```

```

next

```

```

case (Suc l)

```

```

from <(m, n) ∈ Le-enat> show ?case

```

```

proof cases

```

```

case Le-enat-zero

```

```

with <n < enat (Suc l)> show ?thesis by auto

```

```

next

```

```

case (Le-enat-add M N K)
  from ⟨n = N + K⟩ ⟨n < enat (Suc l)⟩ ⟨K ≠ 0⟩
  have N < enat l by(cases N)(cases K, auto simp add: zero-enat-def)
    with ⟨(M, N) ∈ Le-enat⟩ have M < enat l by(rule Suc)
    with ⟨m = eSuc M⟩ show ?thesis by(simp add: eSuc-enat[symmetric])
  qed
qed

lemma enat-less-imp-le:
  assumes k: !!k. n < enat k  $\implies m < enat k$ 
  shows m ≤ n
proof(cases n)
  case (enat n')
    with k[of Suc n'] show ?thesis by(cases m) simp-all
  qed simp

lemma Le-enat-imp-ile:
  ⟨(m, n) ∈ Le-enat⟩  $\implies m \leq n$ 
by(rule enat-less-imp-le)(erule Le-enat-imp-ile-enat-k)

lemma Le-enat-eq-ile:
  ⟨(m, n) ∈ Le-enat⟩  $\longleftrightarrow m \leq n$ 
by(blast intro: Le-enat-imp-ile ile-into-Le-enat)

lemma enat-leI [consumes 1, case-names Leenat, case-conclusion Leenat zero eSuc]:
  assumes major: (m, n) ∈ X
  and step:
     $\bigwedge m\ n.\ (m, n) \in X$ 
     $\implies m = 0 \vee (\exists m' n' k.\ m = eSuc m' \wedge n = n' + enat k \wedge k \neq 0 \wedge ((m', n') \in X \vee m' \leq n'))$ 
  shows m ≤ n
  apply(rule Le-enat.coinduct[unfolded Le-enat-eq-ile, where X=λx y. (x, y) ∈ X])
  apply(fastforce simp add: zero-enat-def dest: step intro: major)+
  done

lemma enat-le-coinduct[consumes 1, case-names le, case-conclusion le 0 eSuc]:
  assumes P: P m n
  and step:
     $\bigwedge m\ n.\ P\ m\ n$ 
     $\implies (n = 0 \longrightarrow m = 0) \wedge$ 
     $(m \neq 0 \longrightarrow n \neq 0 \longrightarrow (\exists k\ n'. P\ (epred m)\ n' \wedge epred n = n' + k) \vee epred m \leq epred n)$ 
  shows m ≤ n
proof -
  from P have ⟨(m, n) ∈ {(m, n). P m n}⟩ by simp
  thus ?thesis
proof(coinduct rule: enat-leI)
  case (Leenat m n)
  hence P m n by simp

```

```

show ?case
proof(cases m rule: enat-coexhaust)
  case 0
  thus ?thesis by simp
next
  case (eSuc m')
  with step[OF ‹P m n›]
  have n ≠ 0 and disj: (exists k n'. P m' n' ∧ epred n = n' + k) ∨ m' ≤ epred n
by auto
  from ‹n ≠ 0› obtain n' where n': n = eSuc n'
    by(cases n rule: enat-coexhaust) auto
  from disj show ?thesis
  proof
    assume m' ≤ epred n
    with eSuc n' show ?thesis
      by(auto 4 3 intro: exI[where x=Suc 0] simp add: eSuc-enat[symmetric]
iadd-Suc-right zero-enat-def[symmetric])
  next
    assume ∃ k n'. P m' n' ∧ epred n = n' + k
    then obtain k n'' where n'': epred n = n'' + k and k: P m' n'' by blast
    show ?thesis using eSuc k n' n'' by blast
    by(cases k)(auto 4 3 simp add: iadd-Suc-right[symmetric] eSuc-enat intro:
exI[where x=∞])
  qed
  qed
  qed
qed

```

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) ∈ X
  and step:
    ⋀ m n. (m, n) ∈ X
    ⟹ m = 0 ∧ n = 0 ∨ (∃ m' n'. m = eSuc m' ∧ n = eSuc n' ∧ ((m', n') ∈ X
    ∨ m' = n'))
  shows m = n
  proof(rule antisym)
    from major show m ≤ n
      by(coinduct rule: enat-leI)
        (drule step, auto simp add: eSuc-plus-1 enat-1[symmetric])
  from major have (n, m) ∈ {(n, m). (m, n) ∈ X} by simp
  thus n ≤ m
  proof(coinduct rule: enat-leI)
    case (Leenat n m)
    hence (m, n) ∈ X by simp
    from step[OF this] show ?case

```

```

    by(auto simp add: eSuc-plus-1 enat-1[symmetric])
qed
qed

lemma enat-coinduct [consumes 1, case-names Eq-enat, case-conclusion Eq-enat
zero eSuc]:
assumes major: P m n
and step:  $\bigwedge m n. P m n$ 
 $\implies (m = 0 \leftrightarrow n = 0) \wedge$ 
 $(m \neq 0 \rightarrow n \neq 0 \rightarrow P(epred m) (epred n) \vee epred m = epred n)$ 
shows m = n
proof -
from major have (m, n) ∈ {(m, n). P m n} by simp
thus ?thesis
proof(coinduct rule: enat-equalityI)
case (Eq-enat m n)
hence P m n by simp
from step[OF this] show ?case
by(cases m n rule: enat-coexhaust[case-product enat-coexhaust]) auto
qed
qed

lemma enat-coinduct2 [consumes 1, case-names zero eSuc]:
 $\llbracket P m n; \bigwedge m n. P m n \rrbracket \implies m = 0 \leftrightarrow n = 0;$ 
 $\bigwedge m n. \llbracket P m n; m \neq 0; n \neq 0 \rrbracket \implies P(epred m) (epred n) \vee epred m =$ 
 $epred n \rrbracket$ 
 $\implies m = n$ 
by(erule enat-coinduct) blast

```

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-unfold stop next x
by(coinduction arbitrary: x rule: enat-coinduct)(subst (1 3) h, auto)

```

1.6 Setup for partial_function

```

lemma enat-diff-cancel-left:  $\llbracket m \leq x; m \leq y \rrbracket \implies x - m = y - m \leftrightarrow x = (y :: enat)$ 
by(cases x y m rule: enat.exhaust[case-product enat.exhaust[case-product enat.exhaust]])(simp-all,
arith)

```

```

lemma finite-lessThan-enatI:
assumes m ≠ ∞
shows finite {..

```

```

thus ?thesis unfolding m by(rule finite-subset) simp
qed

lemma infinite-lessThan-infty:  $\neg \text{finite } \{\dots < \infty :: \text{enat}\}$ 
proof
  have range enat  $\subseteq \{\dots < \infty\}$  by auto
  moreover assume finite  $\{\dots < \infty :: \text{enat}\}$ 
  ultimately have finite (range enat) by(rule finite-subset)
  hence finite (UNIV :: nat set)
    by(rule finite-imageD)(simp add: inj-on-def)
    thus False by simp
qed

lemma finite-lessThan-enat-iff:
  finite  $\{\dots < m :: \text{enat}\} \longleftrightarrow m \neq \infty$ 
by(cases m)(auto intro: finite-lessThan-enatI simp add: infinite-lessThan-infty)

lemma enat-minus-mono1:  $x \leq y \implies x - m \leq y - (m :: \text{enat})$ 
by(cases m x y rule: enat.exhaust[case-product enat.exhaust[case-product enat.exhaust]]) simp-all

lemma max-enat-minus1:  $\max n m - k = \max (n - k) (m - k :: \text{enat})$ 
by(cases n m k rule: enat.exhaust[case-product enat.exhaust[case-product enat.exhaust]]) simp-all

lemma Max-enat-minus1:
  assumes finite A A  $\neq \{\}$ 
  shows Max A - m = Max (( $\lambda n :: \text{enat}. n - m$ ) ` A)
  using assms proof induct
    case (insert x A)
    thus ?case by(cases A = {})(simp-all add: max-enat-minus1)
  qed simp

lemma Sup-enat-minus1:
  assumes m  $\neq \infty$ 
  shows  $\bigcup A - m = \bigcup ((\lambda n :: \text{enat}. n - m) ` A)$ 
proof -
  from assms obtain m' where m = enat m' by auto
  thus ?thesis
    by(auto simp add: Sup-enat-def Max-enat-minus1 finite-lessThan-enat-iff enat-diff-cancel-left
      inj-on-def dest!: finite-imageD2[where B={.. < enat m'}])
  qed

lemma Sup-image-eadd1:
  assumes Y  $\neq \{\}$ 
  shows Sup (( $\lambda y :: \text{enat}. y + x$ ) ` Y) = Sup Y + x
proof(cases finite Y)
  case True
  thus ?thesis by(simp add: Sup-enat-def Max-add-commute assms)

```

```

next
  case False
    thus ?thesis
      proof(cases x)
        case (enat x)
          hence  $\neg \text{finite } ((\lambda y. y+x) ` Y)$  using False
            by(auto dest!: finite-imageD intro: inj-onI)
          with False show ?thesis by(simp add: Sup-enat-def assms)
    next
      case infinity
        hence  $(+) x ` Y = \{\infty\}$  using assms by auto
        thus ?thesis using infinity by(simp add: image-constant-conv assms)
      qed
    qed

lemma Sup-image-eadd2:
   $Y \neq \{\} \implies \text{Sup } ((\lambda y :: \text{enat}. x + y) ` Y) = x + \text{Sup } Y$ 
  by(simp add: Sup-image-eadd1 add.commute)

lemma mono2mono-eSuc [THEN lfp.mono2mono, cont-intro, simp]:
  shows monotone-eSuc: monotone  $(\leq)$   $(\leq)$  eSuc
  by(rule monotoneI) simp

lemma mcont2mcont-eSuc [THEN lfp.mcont2mcont, cont-intro, simp]:
  shows mcont-eSuc: mcont Sup  $(\leq)$  Sup  $(\leq)$  eSuc
  by(intro mcontI contI)(simp-all add: monotone-eSuc eSuc-Sup)

lemma mono2mono-epred [THEN lfp.mono2mono, cont-intro, simp]:
  shows monotone-epred: monotone  $(\leq)$   $(\leq)$  epred
  by(rule monotoneI)(simp add: epred-le-epredI)

lemma mcont2mcont-epred [THEN lfp.mcont2mcont, cont-intro, simp]:
  shows mcont-epred: mcont Sup  $(\leq)$  Sup  $(\leq)$  epred
  by(simp add: mcont-def monotone-epred cont-def epred-Sup)

lemma enat-cocase-mono [partial-function-mono, cont-intro]:
  
$$\begin{aligned} &[\text{monotone orda ordb zero}; \bigwedge n. \text{monotone orda ordb } (\lambda f. \text{esuc } f n)] \\ &\implies \text{monotone orda ordb } (\lambda f. \text{co.case-enat } (\text{zero } f) (\text{esuc } f) x) \end{aligned}$$

  by(cases x rule: co.enat.exhaust) simp-all

lemma enat-cocase-mcont [cont-intro, simp]:
  
$$\begin{aligned} &[\text{mcont luba orda lubb ordb zero}; \bigwedge n. \text{mcont luba orda lubb ordb } (\lambda f. \text{esuc } f n)] \\ &\implies \text{mcont luba orda lubb ordb } (\lambda f. \text{co.case-enat } (\text{zero } f) (\text{esuc } f) x) \end{aligned}$$

  by(cases x rule: co.enat.exhaust) simp-all

lemma eSuc-mono [partial-function-mono]:
   $\text{monotone } (\text{fun-ord } (\leq)) \leq f \implies \text{monotone } (\text{fun-ord } (\leq)) \leq (\lambda x. \text{eSuc } (f x))$ 
  by(rule mono2mono-eSuc)

```

```

lemma mono2mono-enat-minus1 [THEN lfp.mono2mono, cont-intro, simp]:
  shows monotone-enat-minus1: monotone ( $\leq$ ) ( $\leq$ ) ( $\lambda n. n - m :: \text{enat}$ )
  by(rule monotoneI)(rule enat-minus-mono1)

lemma mcont2mcont-enat-minus [THEN lfp.mcont2mcont, cont-intro, simp]:
  shows mcont-enat-minus:  $m \neq \infty \implies \text{mcont Sup} (\leq) \text{ Sup} (\leq) (\lambda n. n - m :: \text{enat})$ 
  by(rule mcontI)(simp-all add: monotone-enat-minus1 contI Sup-enat-minus1)

lemma monotone-eadd1: monotone ( $\leq$ ) ( $\leq$ ) ( $\lambda x. x + y :: \text{enat}$ )
  by(auto intro!: monotoneI)

lemma monotone-eadd2: monotone ( $\leq$ ) ( $\leq$ ) ( $\lambda y. x + y :: \text{enat}$ )
  by(auto intro!: monotoneI)

lemma mono2mono-eadd[THEN lfp.mono2mono2, cont-intro, simp]:
  shows monotone-eadd: monotone (rel-prod ( $\leq$ ) ( $\leq$ )) ( $\leq$ ) ( $\lambda(x, y). x + y :: \text{enat}$ )
  by(simp add: monotone-eadd1 monotone-eadd2)

lemma mcont-eadd2: mcont Sup ( $\leq$ ) Sup ( $\leq$ ) ( $\lambda y. x + y :: \text{enat}$ )
  by(auto intro: mcontI monotone-eadd2 contI Sup-image-eadd2[symmetric])

lemma mcont-eadd1: mcont Sup ( $\leq$ ) Sup ( $\leq$ ) ( $\lambda x. x + y :: \text{enat}$ )
  by(auto intro: mcontI monotone-eadd1 contI Sup-image-eadd1[symmetric])

lemma mcont2mcont-eadd [cont-intro, simp]:
   $\llbracket \text{mcont lub ord Sup} (\leq) (\lambda x. f x);$ 
   $\text{mcont lub ord Sup} (\leq) (\lambda x. g x) \rrbracket$ 
   $\implies \text{mcont lub ord Sup} (\leq) (\lambda x. f x + g x :: \text{enat})$ 
  by(best intro: ccpo.mcont2mcont[OF complete-lattice ccpo] mcont-eadd1 mcont-eadd2
  ccpo.mcont-const[OF complete-lattice ccpo])

lemma eadd-partial-function-mono [partial-function-mono]:
   $\llbracket \text{monotone (fun-ord} (\leq)) (\leq) f; \text{monotone (fun-ord} (\leq)) (\leq) g \rrbracket$ 
   $\implies \text{monotone (fun-ord} (\leq)) (\leq) (\lambda x. f x + g x :: \text{enat})$ 
  by(rule mono2mono-eadd)

lemma monotone-max-enat1: monotone ( $\leq$ ) ( $\leq$ ) ( $\lambda x. \max x y :: \text{enat}$ )
  by(auto intro!: monotoneI simp add: max-def)

lemma monotone-max-enat2: monotone ( $\leq$ ) ( $\leq$ ) ( $\lambda y. \max x y :: \text{enat}$ )
  by(auto intro!: monotoneI simp add: max-def)

lemma mono2mono-max-enat[THEN lfp.mono2mono2, cont-intro, simp]:
  shows monotone-max-enat: monotone (rel-prod ( $\leq$ ) ( $\leq$ )) ( $\leq$ ) ( $\lambda(x, y). \max x y :: \text{enat}$ )
  by(simp add: monotone-max-enat1 monotone-max-enat2)

lemma max-Sup-enat2:

```

```

assumes Y ≠ {}
shows max x (Sup Y) = Sup ((λy :: enat. max x y) ` Y)
proof(cases finite Y)
  case True
    hence max x (Max Y) = Max (max x ` Y) using assms
    proof(induct)
      case (insert y Y)
      thus ?case
        by(cases Y = {})(simp-all, metis max.assoc max.left-commute max.left-idem)
    qed simp
    thus ?thesis using True by(simp add: Sup-enat-def assms)
  next
    case False
    show ?thesis
    proof(cases x)
      case infinity
      hence max x ` Y = {∞} using assms by auto
      thus ?thesis using False by(simp add: Sup-enat-def assms)
    next
      case (enat x')
      { assume finite (max x ` Y)
        hence finite (max x ` {y ∈ Y. y > x})
        by (rule rev-finite-subset)(auto simp add: image-iff dest: max.absorb4 sym)
        hence finite {y ∈ Y. y > x}
        by(rule finite-imageD)(auto intro!: inj-onI simp add: max-def split: if-split-asm)
        moreover have finite {y ∈ Y. y ≤ x}
        by(rule finite-enat-bounded)(auto simp add: enat)
        ultimately have finite ({y ∈ Y. y > x} ∪ {y ∈ Y. y ≤ x}) by simp
        also have {y ∈ Y. y > x} ∪ {y ∈ Y. y ≤ x} = Y by auto
        finally have finite Y . }
      thus ?thesis using False by(auto simp add: Sup-enat-def assms)
    qed
  qed

lemma max-Sup-enat1:
  Y ≠ {} ==> max (Sup Y) x = Sup ((λy :: enat. max y x) ` Y)
  by(subst (1 2) max.commute)(rule max-Sup-enat2)

lemma mcont-max-enat1: mcont Sup (≤) Sup (≤) (λx. max x y :: enat)
  by(auto intro!: mcontI contI max-Sup-enat1 simp add: monotone-max-enat1)

lemma mcont-max-enat2: mcont Sup (≤) Sup (≤) (λy. max x y :: enat)
  by(auto intro!: mcontI contI max-Sup-enat2 simp add: monotone-max-enat2)

lemma mcont2mcont-max-enat [cont-intro, simp]:
  [ mcont lub ord Sup (≤) (λx. f x);
    mcont lub ord Sup (≤) (λx. g x) ]
  ==> mcont lub ord Sup (≤) (λx. max (f x) (g x) :: enat)
  by(best intro: ccpo.mcont2mcont'[OF complete-lattice-ccpo] mcont-max-enat1 mcont-max-enat2)

```

```

ccpo.mcont-const[OF complete-lattice-ccpo])

lemma max-enat-partial-function-mono [partial-function-mono]:
   $\llbracket \text{monotone}(\text{fun-ord}(\leq))(\leq f; \text{monotone}(\text{fun-ord}(\leq))(\leq g) \rrbracket$ 
   $\implies \text{monotone}(\text{fun-ord}(\leq))(\leq (\lambda x. \max(f x)(g x)) :: \text{enat})$ 
by(rule mono2mono-max-enat)

lemma chain-epredI:
  Complete-Partial-Order.chain( $\leq$ ) Y
   $\implies \text{Complete-Partial-Order.chain}(\leq)(\text{epred}^*(Y \cap \{x. x \neq 0\}))$ 
by(auto intro: chainI dest: chainD)

lemma monotone-enat-le-case:
  fixes bot
  assumes mono: monotone( $\leq$ ) ord( $\lambda x. f x (eSuc x)$ )
  and ord:  $\bigwedge x. \text{ord}(\text{bot}(f x (eSuc x)))$ 
  and bot: ord bot bot
  shows monotone( $\leq$ ) ord( $\lambda x. \text{case } x \text{ of } 0 \Rightarrow \text{bot} \mid eSuc x' \Rightarrow f x' x$ )
proof -
  have monotone( $\leq$ ) ord( $\lambda x. \text{if } x \leq 0 \text{ then } \text{bot} \text{ else } f(\text{epred } x) x$ )
  proof(rule monotone-if-bot)
    fix x y :: enat
    assume  $x \leq y \dashv x \leq 0$ 
    thus ord(f(epred x) x) (f(epred y) y)
    by(cases x y rule: co.enat.exhaust[case-product co.enat.exhaust])(auto intro:
    monotoneD[OF mono])
  next
    fix x :: enat
    assume  $\neg x \leq 0$ 
    thus ord bot(f(epred x) x)
    by(cases x rule: co.enat.exhaust)(auto intro: ord)
  qed(rule bot)
  also have ( $\lambda x. \text{if } x \leq 0 \text{ then } \text{bot} \text{ else } f(\text{epred } x) x = (\lambda x. \text{case } x \text{ of } 0 \Rightarrow \text{bot} \mid eSuc x' \Rightarrow f x' x)$ 
  by(auto simp add: fun-eq-iff split: co.enat.split)
  finally show ?thesis .
qed

lemma mcont-enat-le-case:
  fixes bot
  assumes ccpo: class.ccpo lub ord(mk-less ord)
  and mcont: mcont Sup( $\leq$ ) lub ord( $\lambda x. f x (eSuc x)$ )
  and ord:  $\bigwedge x. \text{ord}(\text{bot}(f x (eSuc x)))$ 
  shows mcont Sup( $\leq$ ) lub ord( $\lambda x. \text{case } x \text{ of } 0 \Rightarrow \text{bot} \mid eSuc x' \Rightarrow f x' x$ )
proof -
  from ccpo
  have mcont Sup( $\leq$ ) lub ord( $\lambda x. \text{if } x \leq 0 \text{ then } \text{bot} \text{ else } f(\text{epred } x) x$ )
  proof(rule mcont-if-bot)
    fix x y :: enat

```

```

assume  $x \leq y \neg x \leq 0$ 
thus  $\text{ord} (f (\text{epred } x) x) (f (\text{epred } y) y)$ 
      by(cases x y rule: co.enat.exhaust[case-product co.enat.exhaust])(auto intro:
mcont-monoD[OF mcont])
```

next

```

fix  $Y :: \text{enat set}$ 
assume  $\text{chain}: \text{Complete-Partial-Order.chain} (\leq) Y$ 
and  $Y: Y \neq \{\} \wedge \forall x. x \in Y \implies \neg x \leq 0$ 
from  $Y$  have  $Y': Y \cap \{x. x \neq 0\} \neq \{\}$  by auto
from  $Y(2)$  have  $\text{eq}: Y = eSuc ' (epred ' (Y \cap \{x. x \neq 0\}))$ 
      by(fastforce intro: rev-image-eqI)
let  $?Y = \text{epred} ' (Y \cap \{x. x \neq 0\})$ 
from  $\text{chain-epredI [OF chain]} Y'$ 
have  $f (\bigsqcup ?Y) (eSuc (\bigsqcup ?Y)) = \text{lub} ((\lambda x. f x (eSuc x)) ' ?Y)$ 
      using mcont [THEN mcont-contD] by blast
moreover from  $\text{chain-epredI [OF chain]} Y'$ 
have  $\bigsqcup (eSuc ' ?Y) = eSuc (\bigsqcup ?Y)$ 
      using mcont-eSuc [THEN mcont-contD, symmetric] by blast
ultimately show  $f (\text{epred} (\text{Sup } Y)) (\text{Sup } Y) = \text{lub} ((\lambda x. f (\text{epred } x) x) ' Y)$ 
      by (subst (1 2 3) eq (simp add: image-image))
```

next

```

fix  $x :: \text{enat}$ 
assume  $\neg x \leq 0$ 
thus  $\text{ord} \text{bot} (f (\text{epred } x) x)$ 
      by(cases x rule: co.enat.exhaust)(auto intro: ord)
```

qed

```

also have  $(\lambda x. \text{if } x \leq 0 \text{ then } \text{bot} \text{ else } f (\text{epred } x) x) = (\lambda x. \text{case } x \text{ of } 0 \Rightarrow \text{bot} | eSuc x' \Rightarrow f x' x)$ 
      by(auto simp add: fun-eq-iff split: co.enat.split)
finally show  $?thesis$ .
```

qed

1.7 Misc.

```

lemma  $\text{enat-add-mono} [\text{simp}]$ :
 $\text{enat } x + y < \text{enat } x + z \longleftrightarrow y < z$ 
by(cases y(case-tac [|] z, simp-all))
```

```

lemma  $\text{enat-add1-eq} [\text{simp}]$ :  $\text{enat } x + y = \text{enat } x + z \longleftrightarrow y = z$ 
by (metis enat-add-mono add.commute neq-iff)
```

```

lemma  $\text{enat-add2-eq} [\text{simp}]$ :  $y + \text{enat } x = z + \text{enat } x \longleftrightarrow y = z$ 
by (metis enat-add1-eq add.commute)
```

```

lemma  $\text{enat-less-enat-plusI}$ :  $x < y \implies \text{enat } x < \text{enat } y + z$ 
by(cases z simp-all)
```

```

lemma  $\text{enat-less-enat-plusI2}$ :
 $\text{enat } y < z \implies \text{enat } (x + y) < \text{enat } x + z$ 
```

```

by (metis enat-add-mono plus-enat-simps(1))

lemma min-enat1-conv-enat:  $\bigwedge a b. \min(\text{enat } a) b = \text{enat} (\text{case } b \text{ of } \text{enat } b' \Rightarrow$ 
 $\min a b' \mid \infty \Rightarrow a)$ 
  and min-enat2-conv-enat:  $\bigwedge a b. \min a (\text{enat } b) = \text{enat} (\text{case } a \text{ of } \text{enat } a' \Rightarrow \min$ 
 $a' b \mid \infty \Rightarrow b)$ 
by(simp-all split: enat.split)

lemma eSuc-le-iff:  $eSuc x \leq y \longleftrightarrow (\exists y'. y = eSuc y' \wedge x \leq y')$ 
by(cases y rule: co.enat.exhaust) simp-all

lemma eSuc-eq-infinity-iff:  $eSuc n = \infty \longleftrightarrow n = \infty$ 
by(cases n)(simp-all add: zero-enat-def eSuc-enat)

lemma infinity-eq-eSuc-iff:  $\infty = eSuc n \longleftrightarrow n = \infty$ 
by(cases n)(simp-all add: zero-enat-def eSuc-enat)

lemma enat-cocase-inf:  $(\text{case } \infty \text{ of } 0 \Rightarrow a \mid eSuc b \Rightarrow f b) = f \infty$ 
by(auto split: co.enat.split simp add: infinity-eq-eSuc-iff)

lemma eSuc-Inf:  $eSuc (\text{Inf } A) = \text{Inf} (eSuc ` A)$ 
proof -
  { assume A ≠ {}
    then obtain a where a ∈ A by blast
    then have eSuc (LEAST a. a ∈ A) = (LEAST a. a ∈ eSuc ` A)
    proof (rule LeastI2-wellorder)
      fix a assume a ∈ A and b: ∀ b. b ∈ A → a ≤ b
      then have a: eSuc a ∈ eSuc ` A
        by auto
      then show eSuc a = (LEAST a. a ∈ eSuc ` A)
        by (rule LeastI2-wellorder) (metis (full-types) b a antisym eSuc-le-iff imageE)
    qed }
    then show ?thesis
      by (simp add: Inf-enat-def)
qed

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  $\wedge\wedge$  n) m = m + n
by(induct n arbitrary: m) simp-all

lemma wlog-linorder-le [consumes 0, case-names le symmetry]:
  assumes le:  $\bigwedge a b :: 'a :: \text{linorder}. a \leq b \implies P a b$ 
  and sym:  $P b a \implies P a b$ 
  shows P a b
  proof(cases a ≤ b)
    case True thus ?thesis by(rule le)
  next
    case False
    hence b ≤ a by simp
    hence P b a by(rule le)
    thus ?thesis by(rule sym)
  qed

```

2.2 Type definition

```

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

Coiterator setup.

primcorec unfold-llist :: ('a ⇒ bool)  $\Rightarrow$  ('a ⇒ 'b)  $\Rightarrow$  ('a ⇒ 'a)  $\Rightarrow$  'a ⇒ 'b llist
where
  p a  $\implies$  unfold-llist p g21 g22 a = LNil |
  -  $\implies$  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 ( $\lambda\_. \text{False}$ ) MORE LTL x
= unfold-llist IS-LNIL LHD LTL x
by(coinduction arbitrary: x) auto

```

lemmas about generated constants

lemma *eq-LConsD*: $xs = LCons\ y\ ys \implies xs \neq LNil \wedge lhd\ xs = y \wedge ltl\ xs = ys$
by *auto*

lemma

shows *LNil-eq-lmap*: $LNil = lmap\ f\ xs \longleftrightarrow xs = LNil$
and *lmap-eq-LNil*: $lmap\ f\ xs = LNil \longleftrightarrow xs = LNil$
by(*cases xs, simp-all*)+

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

lemma *ltl-lmap[simp]*: $ltl\ (lmap\ f\ xs) = lmap\ f\ (ltl\ xs)$
by(*cases xs, simp-all*)

declare *llist.map-ident[simp]*

lemma *lmap-eq-LCons-conv*:

$lmap\ f\ xs = LCons\ y\ ys \longleftrightarrow$
 $(\exists x\ xs'.\ xs = LCons\ x\ xs' \wedge y = f\ x \wedge ys = lmap\ f\ xs')$
by(*cases xs*)(*auto*)

lemma *lmap-conv-unfold-llist*:

$lmap\ f = unfold\text{-}llist\ (\lambda xs.\ xs = LNil) (f \circ lhd)\ ltl\ (\mathbf{is}\ ?lhs = ?rhs)$

proof

fix *x*

show $?lhs\ x = ?rhs\ x$ **by**(*coinduction arbitrary*: *x*) *auto*

qed

lemma *lmap-unfold-llist*:

$lmap\ f\ (unfold\text{-}llist\ IS\text{-}LNIL\ LHD\ LTL\ b) = unfold\text{-}llist\ IS\text{-}LNIL\ (f \circ LHD)\ LTL\ b$
by(*coinduction arbitrary*: *b*) *auto*

lemma *lmap-corec-llist*:

$lmap\ f\ (corec\text{-}llist\ IS\text{-}LNIL\ LHD\ endORmore\ TTL\text{-}end\ TTL\text{-}more\ b) =$
 $corec\text{-}llist\ IS\text{-}LNIL\ (f \circ LHD)\ endORmore\ (lmap\ f \circ TTL\text{-}end)\ TTL\text{-}more\ b$
by(*coinduction arbitrary*: *b* rule: *llist.coinduct-strong*) *auto*

lemma *unfold-llist-ltl-unroll*:

$unfold\text{-}llist\ IS\text{-}LNIL\ LHD\ LTL\ (LTL\ b) = unfold\text{-}llist\ (IS\text{-}LNIL \circ LTL)\ (LHD \circ LTL)\ LTL\ b$
by(*coinduction arbitrary*: *b*) *auto*

lemma *ltl-unfold-llist*:

$ltl\ (unfold\text{-}llist\ IS\text{-}LNIL\ LHD\ LTL\ a) =$
 $(if\ IS\text{-}LNIL\ a\ then\ LNil\ else\ unfold\text{-}llist\ IS\text{-}LNIL\ LHD\ LTL\ (LTL\ a))$
by(*simp*)

lemma *unfold-llist-eq-LCons* [*simp*]:

```

unfold-llist IS-LNIL LHD LTL b = LCons x xs  $\longleftrightarrow$ 
 $\neg$  IS-LNIL b  $\wedge$  x = LHD b  $\wedge$  xs = unfold-llist IS-LNIL LHD LTL (LTL b)
by(subst unfold-llist.code) auto

```

```

lemma unfold-llist-id [simp]: unfold-llist lnull lhd ltl xs = xs
by(coinduction arbitrary: xs) simp-all

```

```

lemma lset-eq-empty [simp]: lset xs = {}  $\longleftrightarrow$  lnull xs
by(cases xs) simp-all

```

```
declare llist.setsel(1)[simp]
```

```

lemma lset-ltl: lset (ltl xs)  $\subseteq$  lset xs
by(cases xs) auto

```

```

lemma in-lset-ltlD: x  $\in$  lset (ltl xs)  $\implies$  x  $\in$  lset xs
using lset-ltl[of xs] by auto

```

induction rules

```

theorem llist-set-induct[consumes 1, case-names find step]:
assumes x  $\in$  lset xs and  $\bigwedge_{xs. \neg lnull xs} P (lhd xs) xs$ 
and  $\bigwedge_{xs. y. [\neg lnull xs; y \in lset (ltl xs); P y (ltl xs)]} P y xs$ 
shows P x xs
using assms by(induct)(fastforce simp del: llist.disc(2) iff: llist.disc(2), auto)

```

Test quickcheck setup

```

lemma  $\bigwedge_{xs. xs = LNil}$ 
quickcheck[random, expect=counterexample]
quickcheck[exhaustive, expect=counterexample]
oops

```

```

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

```

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  $\implies$  lnull (ltl xs)
unfolding lnull-def by simp

```

```

lemma neq-LNil-conv: xs  $\neq$  LNil  $\longleftrightarrow$  ( $\exists x xs'. xs = LCons x xs'$ )
by(cases xs) auto

```

```
lemma not-lnull-conv:  $\neg$  lnull xs  $\longleftrightarrow$  ( $\exists x xs'. xs = LCons x xs'$ )
```

```

unfolding lnull-def by(simp add: neq-LNil-conv)

lemma lset-LCons:
  lset (LCons x xs) = insert x (lset xs)
by simp

lemma lset-intros:
  x ∈ lset (LCons x xs)
  x ∈ lset xs ⇒ x ∈ lset (LCons x' xs)
by simp-all

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'
using assms by(cases xs) auto

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
using major apply(induct y≡x xs rule: llist-set-induct)
using 1 2 by(auto simp add: not-lnull-conv)

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
using major
apply(induct rule: lset-induct')
apply(rule find)
apply(case-tac x'=x)
apply(simp-all add: find step)
done

lemmas lset-LNil = llist.set(1)

lemma lset-lnull: lnull xs ⇒ lset xs = {}
by(auto dest: llist.collapse)

Alternative definition of lset for nitpick

inductive lsetp :: 'a llist ⇒ 'a ⇒ bool
where
  lsetp (LCons x xs) x
  | lsetp xs x ⇒ lsetp (LCons x' xs) x

lemma lset-into-lsetp:

```

```

 $x \in lset xs \implies lsetp xs x$ 
by(induct rule: lset-induct)(blast intro: lsetp.intros)+

lemma lsetp-into-lset:
 $lsetp xs x \implies x \in lset xs$ 
by(induct rule: lsetp.induct)(blast intro: lset-intros)+

lemma lset-eq-lsetp [nitpick-unfold]:
 $lset xs = \{x. lsetp xs x\}$ 
by(auto intro: lset-into-lsetp dest: lsetp-into-lset)

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  $\Rightarrow$  'a llist  $\Rightarrow$  'a set
where gen-lset A xs = A  $\cup$  lset xs

lemma gen-lset-code [code]:
 $gen-lset A LNil = A$ 
 $gen-lset A (LCons x xs) = gen-lset (insert x A) xs$ 
by(auto simp add: gen-lset-def)

lemma lset-code [code]:
 $lset = gen-lset \{\}$ 
by(simp add: gen-lset-def fun-eq-iff)

definition lmmember :: 'a  $\Rightarrow$  'a llist  $\Rightarrow$  bool
where lmmember x xs  $\longleftrightarrow$  x  $\in$  lset xs

lemma lmmember-code [code]:
 $lmember x LNil \longleftrightarrow False$ 
 $lmember x (LCons y ys) \longleftrightarrow x = y \vee lmmember x ys$ 
by(simp-all add: lmmember-def)

lemma lset-lmmember [code-unfold]:
 $x \in lset xs \longleftrightarrow lmmember x xs$ 
by(simp add: lmmember-def)

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

2.4 The subset of finite lazy lists lfinit
inductive lfinit :: 'a llist  $\Rightarrow$  bool
where
 $| lfinit-LNil: lfinit LNil$ 
 $| lfinit-LConsI: lfinit xs \implies lfinit (LCons x xs)$ 

declare lfinit-LNil [iff]

```

```

lemma lnull-imp-lfinite [simp]: lnull xs  $\implies$  lfinite xs
by(auto dest: llist.collapse)

lemma lfinite-LCons [simp]: lfinite (LCons x xs) = lfinite xs
by(auto elim: lfinite.cases intro: lfinite-LConsi)

lemma lfinite-ltl [simp]: lfinite (ltl xs) = lfinite xs
by(cases xs) simp-all

lemma lfinite-code [code]:
  lfinite LNil = True
  lfinite (LCons x xs) = lfinite xs
by simp-all

lemma lfinite-induct [consumes 1, case-names LNil LCons]:
  assumes lfinite: lfinite xs
  and LNil:  $\bigwedge_{xs} \text{lnull } xs \implies P \text{ } xs$ 
  and LCons:  $\bigwedge_{xs} [\text{lfinite } xs; \neg \text{lnull } xs; P \text{ } (\text{ltl } xs)] \implies P \text{ } xs$ 
  shows P xs
  using lfinite by(induct)(auto intro: LCons LNil)

lemma lfinite-imp-finite-lset:
  assumes lfinite xs
  shows finite (lset xs)
  using assms by induct auto

```

2.5 Concatenating two lists: lappend

```

primcorec lappend :: 'a llist  $\Rightarrow$  'a llist  $\Rightarrow$  'a llist
where
  lappend xs ys = (case xs of LNil  $\Rightarrow$  ys | LCons x xs'  $\Rightarrow$  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)
  by(case-tac [!] xs) simp-all

lemma lnull-lappend [simp]:
  lnull (lappend xs ys)  $\longleftrightarrow$  lnull xs  $\wedge$  lnull ys
  by(auto simp add: lappend-def)

lemma lappend-eq-LNil-iff:
  lappend xs ys = LNil  $\longleftrightarrow$  xs = LNil  $\wedge$  ys = LNil

```

```

using lnull-lappend unfolding lnull-def .

lemma LNil-eq-lappend-iff:
  LNil = lappend xs ys  $\longleftrightarrow$  xs = LNil  $\wedge$  ys = LNil
  by(auto dest: sym simp add: lappend-eq-LNil-iff)

lemma lappend-LNil2 [simp]: lappend xs LNil = xs
  by(coinduction arbitrary: xs) simp-all

lemma shows lappend-lnull1: lnull xs  $\implies$  lappend xs ys = ys
  and lappend-lnull2: lnull ys  $\implies$  lappend xs ys = xs
  unfolding lnull-def by simp-all

lemma lappend-assoc: lappend (lappend xs ys) zs = lappend xs (lappend ys zs)
  by(coinduction arbitrary: xs rule: llist.coinduct-strong) auto

lemma lmap-lappend-distrib:
  lmap f (lappend xs ys) = lappend (lmap f xs) (lmap f ys)
  by(coinduction arbitrary: xs rule: llist.coinduct-strong) auto

lemma lappend-snocL1-conv-LCons2:
  lappend (lappend xs (LCons y LNil)) ys = lappend xs (LCons y ys)
  by(simp add: lappend-assoc)

lemma lappend-ltl:  $\neg$  lnull xs  $\implies$  lappend (ltl xs) ys = ltl (lappend xs ys)
  by simp

lemma lfinite-lappend [simp]:
  lfinite (lappend xs ys)  $\longleftrightarrow$  lfinite xs  $\wedge$  lfinite ys
  (is ?lhs  $\longleftrightarrow$  ?rhs)
proof
  assume ?lhs thus ?rhs
  proof(induct zs=lappend xs ys arbitrary: xs ys)
    case lfinite-LNil
    thus ?case by(simp add: LNil-eq-lappend-iff)
  next
    case (lfinite-LConsI zs z)
    thus ?case by(cases xs)(cases ys, simp-all)
  qed
next
  assume ?rhs (is ?xs  $\wedge$  ?ys)
  then obtain ?xs ?ys ..
  thus ?lhs by(induct simp-all)
qed

lemma lappend-inf:  $\neg$  lfinite xs  $\implies$  lappend xs ys = xs
  by(coinduction arbitrary: xs) auto

lemma lfinite-lmap [simp]:

```

```

lfinite (lmap f xs) = lfinite xs
(is ?lhs ↔ ?rhs)
proof
  assume ?lhs
  thus ?rhs
    by(induct zs≡lmap f xs arbitrary: xs rule: lfinite-induct) fastforce+
next
  assume ?rhs
  thus ?lhs by(induct) simp-all
qed

lemma lset-lappend-lfinite [simp]:
  lfinite xs ⇒ lset (lappend xs ys) = lset xs ∪ lset ys
by(induct rule: lfinite.induct) auto

lemma lset-lappend: lset (lappend xs ys) ⊆ lset xs ∪ lset ys
proof(cases lfinite xs)
  case True
  thus ?thesis by simp
next
  case False
  thus ?thesis by(auto simp add: lappend-inf)
qed

lemma lset-lappendI: lset xs ⊆ lset (lappend xs ys)
by(rule subsetI)(erule lset-induct, simp-all)

lemma lset-lappend-conv: lset (lappend xs ys) = (if lfinite xs then lset xs ∪ lset ys
else lset xs)
by(simp add: lappend-inf)

lemma in-lset-lappend-iff: x ∈ lset (lappend xs ys) ↔ x ∈ lset xs ∨ lfinite xs ∧
x ∈ lset ys
by(simp add: lset-lappend-conv)

lemma split-llist-first:
  assumes x ∈ lset xs
  shows ∃ ys zs. xs = lappend ys (LCons x zs) ∧ lfinite ys ∧ x ∉ lset ys
using assms
proof(induct)
  case find thus ?case by(auto intro: exI[where x=LNil])
next
  case step thus ?case by(fastforce intro: exI[where x=LCons a b for a b])
qed

lemma split-llist: x ∈ lset xs ⇒ ∃ ys zs. xs = lappend ys (LCons x zs) ∧ lfinite
ys
by(blast dest: split-llist-first)

```

2.6 The prefix ordering on lazy lists: *lprefix*

```

coinductive lprefix :: 'a llist ⇒ 'a llist ⇒ bool (infix ⊑ 65)
where
  LNil-lprefix [simp, intro!]: LNil ⊑ xs
  | Le-LCons: xs ⊑ ys ⇒⇒ LCons x xs ⊑ LCons x ys

lemma lprefixI [consumes 1, case-names lprefix,
                 case-conclusion lprefix LeLNil LeLCons]:
  assumes major: (xs, ys) ∈ X
  and step:
    ∧ xs ys. (xs, ys) ∈ X
    ⇒⇒ lnull xs ∨ (∃ x xs' ys'. xs = LCons x xs' ∧ ys = LCons x ys' ∧
                    ((xs', ys') ∈ X ∨ xs' ⊑ ys'))
  shows xs ⊑ ys
  using major by(rule lprefix.coinduct)(auto dest: step)

lemma lprefix-coinduct [consumes 1, case-names lprefix, case-conclusion lprefix
  LNil LCons, coinduct pred: lprefix]:
  assumes major: P xs ys
  and step: ∧ xs ys. P xs ys
  ⇒⇒ (lnull ys → lnull xs) ∧
       (¬ lnull xs → ¬ lnull ys → lhd xs = lhd ys ∧ (P (ltl xs) (ltl ys)) ∨ ltl xs ⊑
        ltl ys))
  shows xs ⊑ ys
  proof –
    from major have (xs, ys) ∈ {(xs, ys). P xs ys} by simp
    thus ?thesis
    proof(coinduct rule: lprefixI)
      case (lprefix xs ys)
      hence P xs ys by simp
      from step[OF this] show ?case
        by(cases xs)(auto simp add: not-lnull-conv)
    qed
  qed

lemma lprefix-refl [intro, simp]: xs ⊑ xs
by(coinduction arbitrary: xs) simp-all

lemma lprefix-LNil [simp]: xs ⊑ LNil ↔ lnull xs
unfolding lnull-def by(subst lprefix.simps) simp

lemma lprefix-lnull: lnull ys ⇒⇒ xs ⊑ ys ↔ lnull xs
unfolding lnull-def by auto

lemma lnull-lprefix: lnull xs ⇒⇒ lprefix xs ys
by(simp add: lnull-def)

lemma lprefix-LCons-conv:
  xs ⊑ LCons y ys ↔

```

$xs = LNil \vee (\exists xs'. xs = LCons y xs' \wedge xs' \sqsubseteq ys)$
by(*subst lprefix.simps*) *simp*

lemma *LCons-lprefix-LCons* [*simp*]:
 $LCons x xs \sqsubseteq LCons y ys \longleftrightarrow x = y \wedge xs \sqsubseteq ys$
by(*simp add: lprefix-LCons-conv*)

lemma *LCons-lprefix-conv*:
 $LCons x xs \sqsubseteq ys \longleftrightarrow (\exists ys'. ys = LCons x ys' \wedge xs \sqsubseteq ys')$
by(*cases ys*)(*auto*)

lemma *lprefix-ltlI*: $xs \sqsubseteq ys \implies ltl xs \sqsubseteq ltl ys$
by(*cases ys*)(*auto simp add: lprefix-LCons-conv*)

lemma *lprefix-code* [*code*]:
 $LNil \sqsubseteq ys \longleftrightarrow True$
 $LCons x xs \sqsubseteq LNil \longleftrightarrow False$
 $LCons x xs \sqsubseteq LCons y ys \longleftrightarrow x = y \wedge xs \sqsubseteq ys$
by *simp-all*

lemma *lprefix-lhdD*: $\llbracket xs \sqsubseteq ys; \neg lnull xs \rrbracket \implies lhd xs = lhd ys$
by(*clar simp simp add: not-lnull-conv LCons-lprefix-conv*)

lemma *lprefix-lnullD*: $\llbracket xs \sqsubseteq ys; lnull ys \rrbracket \implies lnull xs$
by(*auto elim: lprefix.cases*)

lemma *lprefix-not-lnullD*: $\llbracket xs \sqsubseteq ys; \neg lnull xs \rrbracket \implies \neg lnull ys$
by(*auto elim: lprefix.cases*)

lemma *lprefix-expand*:
 $(\neg lnull xs \implies \neg lnull ys \wedge lhd xs = lhd ys \wedge ltl xs \sqsubseteq ltl ys) \implies xs \sqsubseteq ys$
by(*cases xs ys rule: llist.exhaust[case-product llist.exhaust])(*simp-all*)*

lemma *lprefix-antisym*:
 $\llbracket xs \sqsubseteq ys; ys \sqsubseteq xs \rrbracket \implies xs = ys$
by(*coinduction arbitrary: xs ys*)(*auto simp add: not-lnull-conv lprefix-lnull*)

lemma *lprefix-trans* [*trans*]:
 $\llbracket xs \sqsubseteq ys; ys \sqsubseteq zs \rrbracket \implies xs \sqsubseteq zs$
by(*coinduction arbitrary: xs ys zs*)(*auto 4 3 dest: lprefix-lnullD lprefix-lhdD intro: lprefix-ltlI*)

lemma *preorder-lprefix* [*cont-intro*]:
class.preorder (\sqsubseteq) (*mk-less* (\sqsubseteq))
by(*unfold-locales*)(*auto simp add: mk-less-def intro: lprefix-trans*)

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

```

proof
  fix x
  assume x ∈ lset xs
  thus x ∈ lset ys using assms
    by(induct arbitrary: ys)(auto simp add: LCons-lprefix-conv)
qed

lemma lprefix-lappend-sameI:
  assumes xs ⊑ ys
  shows lappend zs xs ⊑ lappend zs ys
proof(cases lfinite zs)
  case True
  thus ?thesis using assms by induct auto
qed(simp add: lappend-inf)

lemma not-lfinite-lprefix-conv-eq:
  assumes nfin: ¬ lfinite xs
  shows xs ⊑ ys ↔ xs = ys
proof
  assume xs ⊑ ys
  with nfin show xs = ys
    by(coinduction arbitrary: xs ys)(auto dest: lprefix-lnullD lprefix-lhdD intro:
      lprefix-ltlI)
qed simp

lemma lprefix-lappend: xs ⊑ lappend xs ys
by(coinduction arbitrary: xs) auto

lemma lprefix-down-linear:
  assumes xs ⊑ zs ys ⊑ zs
  shows xs ⊑ ys ∨ ys ⊑ xs
proof(rule disjCI)
  assume ¬ ys ⊑ xs
  with assms show xs ⊑ ys
    by(coinduction arbitrary: xs ys zs)(auto simp add: not-lnull-conv LCons-lprefix-conv,
      simp add: lnull-def)
qed

lemma lprefix-lappend-same [simp]:
  lappend xs ys ⊑ lappend xs zs ↔ (lfinite xs → ys ⊑ zs)
  (is ?lhs ↔ ?rhs)
proof
  assume ?lhs
  show ?rhs
  proof
    assume lfinite xs
    thus ys ⊑ zs using ‹?lhs› by(induct) simp-all
  qed
next

```

```

assume ?rhs
show ?lhs
proof(cases lfinite xs)
  case True
    hence yszs: ys ⊑ zs by(rule ‹?rhs›[rule-format])
    from True show ?thesis by induct (simp-all add: yszs)
  next
    case False thus ?thesis by(simp add: lappend-inf)
  qed
qed

```

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)
by(simp add: lSup-def)

lemma lhd-lSup [simp]: ∃ x ∈ A. ¬ lnull x ⇒ lhd (lSup A) = (THE x. x ∈ lhd ‘(A ∩ {xs. ¬ lnull xs}))
by(auto simp add: lSup-def)

lemma ltl-lSup [simp]: ltl (lSup A) = lSup (ltl ‘(A ∩ {xs. ¬ lnull xs}))
by(cases ∀ xs ∈ A. lnull xs)(auto 4 3 simp add: lSup-def intro: llist.expand)

lemma lhd-lSup-eq:
  assumes chain: Complete-Partial-Order.chain (⊑) Y
  shows [| xs ∈ Y; ¬ lnull xs |] ⇒ lhd (lSup Y) = lhd xs
by(subst lhd-lSup)(auto 4 3 dest: chainD[OF chain] lprefix-lhdD intro!: the-equality)

lemma lSup-empty [simp]: lSup {} = LNil
by(simp add: lSup-def)

lemma lSup-singleton [simp]: lSup {xs} = xs
by(coinduction arbitrary: xs) simp-all

lemma LCons-image-Int-not-lnull: (LCons x ‘ A ∩ {xs. ¬ lnull xs}) = LCons x ‘ A
by auto

lemma lSup-LCons: A ≠ {} ⇒ lSup (LCons x ‘ A) = LCons x (lSup A)
by(rule llist.expand)(auto simp add: image-image lhd-lSup exI LCons-image-Int-not-lnull)

```

intro!: the-equality)

lemma *lSup-eq-LCons-iff*:

lSup Y = LCons x xs \longleftrightarrow ($\exists x \in Y. \neg lnull x \wedge x = (\text{THE } x. x \in lhd ' (Y \cap \{xs. \neg lnull xs\})) \wedge xs = lSup (ltl ' (Y \cap \{xs. \neg lnull xs\}))$)
by(*auto dest: eq-LConsD simp add: lnull-def[symmetric] intro: llist.expand*)

lemma *lSup-insert-LNil*: *lSup (insert LNil Y) = lSup Y*
by(*rule llist.expand*) *simp-all*

lemma *lSup-minus-LNil*: *lSup (Y - {LNil}) = lSup Y*
using *lSup-insert-LNil[where Y=Y - {LNil}, symmetric]*
by(*cases LNil ∈ Y*)(*simp-all add: insert-absorb*)

lemma *chain-lprefix-ltl*:

assumes *chain: Complete-Partial-Order.chain (≤) A*
shows *Complete-Partial-Order.chain (≤) (ltl ' (A ∩ {xs. ¬ lnull xs}))*
by(*auto intro!: chainI dest: chainD[OF chain] intro: lprefix-ltlI*)

lemma *lSup-finite-prefixes*: *lSup {ys. ys ≤ xs ∧ lfinite ys} = xs* (**is** *lSup (?C xs) = -*)
proof(*coinduction arbitrary: xs*)
case (*Eq-llist xs*)
have *?lnull*
by(*cases xs*)(*auto simp add: lprefix-LCons-conv*)
moreover
have *¬ lnull xs \Longrightarrow ltl ' ({ys. ys ≤ xs ∧ lfinite ys} ∩ {xs. ¬ lnull xs}) = {ys. ys ≤ ltl xs ∧ lfinite ys}*
by(*auto 4 4 intro!: rev-image-eqI[where x=LCons (lhd xs) ys for ys] intro: llist.expand lprefix-ltlI simp add: LCons-lprefix-conv*)
hence *?LCons* **by**(*auto 4 3 intro!: the-equality dest: lprefix-lhdD intro: rev-image-eqI)
ultimately show ?case ..
qed*

lemma *lSup-finite-gen-prefixes*:

assumes *zs ≤ xs lfinite zs*
shows *lSup {ys. ys ≤ xs ∧ zs ≤ ys ∧ lfinite ys} = xs*
using *⟨lfinite zs⟩ ⟨zs ≤ xs⟩*
proof(*induction arbitrary: xs*)
case *lfinite-LNil*
thus *?case* **by**(*simp add: lSup-finite-prefixes*)
next
case (*lfinite-LConsI zs z*)
from *⟨LCons z zs ≤ xs⟩ obtain xs' where xs: xs = LCons z xs'*
and *zs ≤ xs'* **by**(*auto simp add: LCons-lprefix-conv*)
show *?case* (**is** *?lhs = ?rhs*)
proof(*rule llist.expand*)
show *lnull ?lhs = lnull ?rhs* **using** *xs lfinite-LConsI*
by(*auto 4 3 simp add: lprefix-LCons-conv del: disjCI intro: disjI2*)

```

next
assume lnull:  $\neg \text{lnull } ?lhs \neg \text{lnull } ?rhs$ 
have lhd ?lhs = lhd ?rhs using lnull xs
  by(auto intro!: rev-image-eqI simp add: LCons-lprefix-conv)
moreover
have  $\text{ltl} (\{ys. ys \sqsubseteq xs \wedge \text{LCons } z \text{ zs} \sqsubseteq ys \wedge \text{lfinite } ys\} \cap \{xs. \neg \text{lnull } xs\}) =$ 
   $\{ys. ys \sqsubseteq xs' \wedge \text{zs} \sqsubseteq ys \wedge \text{lfinite } ys\}$ 
  using xs  $\hookrightarrow$  lnull ?rhs
  by(auto 4 3 simp add: lprefix-LCons-conv intro: rev-image-eqI del: disjCI intro: disjI2)
  hence  $\text{ltl } ?lhs = \text{ltl } ?rhs$  using lfinite-LConsI.IH[ $\text{OF } \langle \text{zs} \sqsubseteq xs' \rangle$ ] xs by simp
  ultimately show lhd ?lhs = lhd ?rhs  $\wedge$   $\text{ltl } ?lhs = \text{ltl } ?rhs$  ..
qed
qed

lemma lSup-strict-prefixes:
 $\neg \text{lfinite } xs \implies \text{lSup } \{ys. ys \sqsubseteq xs \wedge ys \neq xs\} = xs$ 
(is -  $\implies$  lSup (?C xs) = -)
proof(coinduction arbitrary: xs)
  case (Eq-list xs)
  then obtain x x' xs' where xs:  $xs = \text{LCons } x (\text{LCons } x' xs')$   $\neg \text{lfinite } xs'$ 
    by(cases xs)(simp, rename-tac xs', case-tac xs', auto)
  hence ?lnull by(auto intro: exI[where x=LCons x (LCons x' LNil)])
  moreover hence  $\neg \text{lnull } (\text{lSup } \{ys. ys \sqsubseteq xs \wedge ys \neq xs\})$  using xs by auto
  hence lhd (lSup {ys. ys  $\sqsubseteq$  xs  $\wedge$  ys  $\neq$  xs}) = lhd xs using xs
    by(auto 4 3 intro!: the-equality intro: rev-image-eqI dest: lprefix-lhdD)
  moreover from xs
  have  $\text{ltl} (\{ys. ys \sqsubseteq xs \wedge ys \neq xs\} \cap \{xs. \neg \text{lnull } xs\}) = \{ys. ys \sqsubseteq \text{ltl } xs \wedge ys \neq \text{ltl } xs\}$ 
    by(auto simp add: lprefix-LCons-conv intro: image-eqI[where x=LCons x (LCons x' ys) for ys] image-eqI[where x=LCons x LNil])
    ultimately show ?case using Eq-list by clarsimp
qed

lemma chain-lprefix-lSup:
 $\llbracket \text{Complete-Partial-Order.chain } (\sqsubseteq) A; xs \in A \rrbracket$ 
 $\implies xs \sqsubseteq \text{lSup } A$ 
proof(coinduction arbitrary: xs A)
  case (lprefix xs A)
  note chain = <Complete-Partial-Order.chain ( $\sqsubseteq$ ) A>
  from <xs  $\in$  A> show ?case
    by(auto 4 3 dest: chainD[ $\text{OF chain}$ ] lprefix-lhdD intro: chain-lprefix-ltl[ $\text{OF chain}$ ]
    intro!: the-equality[symmetric])
  qed

lemma chain-lSup-lprefix:
 $\llbracket \text{Complete-Partial-Order.chain } (\sqsubseteq) A; \bigwedge xs. xs \in A \implies xs \sqsubseteq zs \rrbracket$ 
 $\implies \text{lSup } A \sqsubseteq zs$ 
proof(coinduction arbitrary: A zs)

```

```

case (lprefix A zs)
note chain = `Complete-Partial-Order.chain ( $\sqsubseteq$ ) A`
from  $\forall xs. xs \in A \longrightarrow xs \sqsubseteq zs$ 
show ?case
  by(auto 4 4 dest: lprefix-lnullD lprefix-lhdD intro: chain-lprefix-ltl[OF chain]
lprefix-ltlI rev-image-eqI intro!: the-equality)
qed

lemma llist-ccpo [simp, cont-intro]: class ccpo lSup ( $\sqsubseteq$ ) (mk-less ( $\sqsubseteq$ ))
by(unfold-locales)(auto dest: lprefix-antisym intro: lprefix-trans chain-lprefix-lSup
chain-lSup-lprefix simp add: mk-less-def)

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

lemma llist-partial-function-definitions:
  partial-function-definitions ( $\sqsubseteq$ ) lSup
by(unfold-locales)(auto dest: lprefix-antisym intro: lprefix-trans chain-lprefix-lSup
chain-lSup-lprefix)

interpretation llist: partial-function-definitions ( $\sqsubseteq$ ) lSup
  rewrites lSup {}  $\equiv$  LNil
by(rule llist-partial-function-definitions)(simp)

abbreviation mono-list  $\equiv$  monotone (fun-ord ( $\sqsubseteq$ )) ( $\sqsubseteq$ )

interpretation llist-lift: partial-function-definitions fun-ord lprefix fun-lub lSup
  rewrites fun-lub lSup {}  $\equiv$   $\lambda\_. LNil$ 
by(rule llist-partial-function-definitions[THEN partial-function-lift])(simp)

abbreviation mono-list-lift  $\equiv$  monotone (fun-ord (fun-ord lprefix)) (fun-ord lprefix)

lemma lprefixes-chain:
  Complete-Partial-Order.chain ( $\sqsubseteq$ ) {ys. lprefix ys xs}
by(rule chainI)(auto dest: lprefix-down-linear)

lemma llist-gen-induct:
  assumes adm: ccpo.admissible lSup ( $\sqsubseteq$ ) P
  and step:  $\exists zs. zs \sqsubseteq xs \wedge lfinite zs \wedge (\forall ys. zs \sqsubseteq ys \longrightarrow ys \sqsubseteq xs \longrightarrow lfinite ys \longrightarrow P ys)$ 
  shows P xs
proof -
  from step obtain zs
    where zs: zs  $\sqsubseteq$  xs lfinite zs
    and ys:  $\bigwedge ys. [zs \sqsubseteq ys; ys \sqsubseteq xs; lfinite ys] \implies P ys$  by blast
    let ?C = {ys. ys  $\sqsubseteq$  xs  $\wedge$  zs  $\sqsubseteq$  ys  $\wedge$  lfinite ys}
    from lprefixes-chain[of xs]
    have Complete-Partial-Order.chain ( $\sqsubseteq$ ) ?C
      by(auto dest: chain-compr)

```

```

with adm have P (lSup ?C)
  by(rule ccpo.admissibleD)(auto intro: ys zs)
also have lSup ?C = xs
  using lSup-finite-gen-prefixes[OF zs] by simp
finally show ?thesis .
qed

lemma llist-induct [case-names adm LNil LCons, induct type: llist]:
assumes adm: ccpo.admissible lSup ( $\sqsubseteq$ ) P
and LNil: P LNil
and LCons:  $\bigwedge x \text{ xs}. \llbracket \text{lfinite } \text{xs}; P \text{ xs} \rrbracket \implies P (\text{LCons } x \text{ xs})$ 
shows P xs
proof -
  { fix ys :: 'a llist
    assume lfinite ys
    hence P ys by(induct)(simp-all add: LNil LCons) }
  note [intro] = this
  show ?thesis using adm
    by(rule llist-gen-induct)(auto intro: exI[where x=LNil])
qed

lemma LCons-mono [partial-function-mono, cont-intro]:
mono-llist A  $\implies$  mono-llist ( $\lambda f. \text{LCons } x (A f)$ )
by(rule monotoneI)(auto dest: monotoneD)

lemma mono2mono-LCons [THEN llist.mono2mono, simp, cont-intro]:
shows monotone-LCons: monotone ( $\sqsubseteq$ ) ( $\sqsubseteq$ ) (LCons x)
by(auto intro: monotoneI)

lemma mcont2mcont-LCons [THEN llist.mcont2mcont, simp, cont-intro]:
shows mcont-LCons: mcont lSup ( $\sqsubseteq$ ) lSup ( $\sqsubseteq$ ) (LCons x)
by(simp add: mcont-def monotone-LCons lSup-LCons[symmetric] contI)

lemma mono2mono-ltl[THEN llist.mono2mono, simp, cont-intro]:
shows monotone-ltl: monotone ( $\sqsubseteq$ ) ( $\sqsubseteq$ ) ltl
by(auto intro: monotoneI lprefix-ltlI)

lemma cont-ltl: cont lSup ( $\sqsubseteq$ ) lSup ( $\sqsubseteq$ ) ltl
proof(rule contI)
fix Y :: 'a llist set
assume Y  $\neq \{\}$ 
have ltl (lSup Y) = lSup (insert LNil (ltl ` (Y  $\cap$  {xs.  $\neg$  lnull xs})))
  by(simp add: lSup-insert-LNil)
also have insert LNil (ltl ` (Y  $\cap$  {xs.  $\neg$  lnull xs})) = insert LNil (ltl ` Y) by
auto
also have lSup ... = lSup (ltl ` Y) by(simp add: lSup-insert-LNil)
finally show ltl (lSup Y) = lSup (ltl ` Y) .
qed

```

```

lemma mcont2mcont-ltl [THEN llist.mcont2mcont, simp, cont-intro]:
  shows mcont-ltl: mcont lSup ( $\sqsubseteq$ ) lSup ( $\sqsubseteq$ ) ltl
  by(simp add: mcont-def monotone-ltl cont-ltl)

lemma llist-case-mono [partial-function-mono, cont-intro]:
  assumes lnil: monotone orda ordb lnil
  and lcons:  $\bigwedge x \in xs. \text{monotone orda ordb } (\lambda f. \text{lcons } f \ x \ xs)$ 
  shows monotone orda ordb  $(\lambda f. \text{case}-llist (lnil \ f) (\text{lcons } f) \ x)$ 
  by(rule monotoneI)(auto split: llist.split dest: monotoneD[OF lnil] monotoneD[OF lcons])

lemma mcont-llist-case [cont-intro, simp]:
   $\llbracket \text{mcont luba orda lubb ordb } (\lambda x. f \ x); \bigwedge x \in xs. \text{mcont luba orda lubb ordb } (\lambda y. g \ x \ xs \ y) \rrbracket$ 
   $\implies \text{mcont luba orda lubb ordb } (\lambda y. \text{case } xs \text{ of } LNil \Rightarrow f \ y \mid LCons \ x \ xs' \Rightarrow g \ x \ xs' \ y)$ 
  by(cases xs) simp-all

lemma monotone-lprefix-case [cont-intro, simp]:
  assumes mono:  $\bigwedge x. \text{monotone } (\sqsubseteq) \ (\sqsubseteq) \ (\lambda xs. f \ x \ xs \ (LCons \ x \ xs))$ 
  shows monotone ( $\sqsubseteq$ ) ( $\sqsubseteq$ )  $(\lambda xs. \text{case } xs \text{ of } LNil \Rightarrow LNil \mid LCons \ x \ xs' \Rightarrow f \ x \ xs')$ 
  by(rule llist.monotone-if-bot[where  $f = \lambda xs. f \ (lhd \ xs) \ (ltl \ xs) \ xs$  and bound=LNil])(auto 4 3 split: llist.split simp add: not-lnull-conv LCons-lprefix-conv dest: monotoneD[OF mono])

lemma mcont-lprefix-case-aux:
  fixes f bot
  defines g  $\equiv \lambda xs. f \ (lhd \ xs) \ (ltl \ xs) \ (LCons \ (lhd \ xs) \ (ltl \ xs))$ 
  assumes mcont:  $\bigwedge x. \text{mcont lSup } (\sqsubseteq) \ \text{lub ord } (\lambda xs. f \ x \ xs \ (LCons \ x \ xs))$ 
  and ccpo: class ccpo lub ord (mk-less ord)
  and bot:  $\bigwedge x. \text{ord bot } x$ 
  shows mcont lSup ( $\sqsubseteq$ ) lub ord  $(\lambda xs. \text{case } xs \text{ of } LNil \Rightarrow \text{bot} \mid LCons \ x \ xs' \Rightarrow f \ x \ xs' \ xs)$ 
  proof(rule llist.mcont-if-bot[where  $f = g$  and bound=LNil and bot=bot, OF ccpo bot])
    fix Y :: 'a llist set
    assume chain: Complete-Partial-Order.chain ( $\sqsubseteq$ ) Y
    and Y:  $Y \neq \{\}$   $\bigwedge x. x \in Y \implies \neg x \sqsubseteq LNil$ 
    from Y have Y':  $Y \cap \{xs. \neg \text{lnull } xs\} \neq \{\}$  by auto
    from Y obtain x xs where LCons x xs  $\in Y$  by(fastforce simp add: not-lnull-conv)
    with Y(2) have eq:  $Y = LCons \ x \ ` (ltl \ ` (Y \cap \{xs. \neg \text{lnull } xs\}))$ 
    by(force dest: chainD[OF chain] simp add: LCons-lprefix-conv lprefix-LCons-conv intro: imageI rev-image-eqI)
    show g (lSup Y) = lub (g ` Y)
    by(subst (1 2) eq)(simp add: lSup-LCons Y' g-def mcont-contD[OF mcont] chain chain-lprefix-ltl, simp add: image-image)
    qed(auto 4 3 split: llist.split simp add: not-lnull-conv LCons-lprefix-conv g-def intro: mcont-monoD[OF mcont])

```

```

lemma mcont-lprefix-case [cont-intro, simp]:
  assumes  $\bigwedge x. \text{mcont } lSup (\sqsubseteq) lSup (\sqsubseteq) (\lambda xs. f x xs (LCons x xs))$ 
  shows  $\text{mcont } lSup (\sqsubseteq) lSup (\sqsubseteq) (\lambda xs. \text{case } xs \text{ of } LNil \Rightarrow LNil \mid LCons x xs' \Rightarrow f x xs' xs)$ 
  using assms by(rule mcont-lprefix-case-aux)(simp-all add: llist-ccpo)

lemma monotone-lprefix-case-lfp [cont-intro, simp]:
  fixes  $f :: - \Rightarrow - :: \text{order-bot}$ 
  assumes mono:  $\bigwedge x. \text{monotone } (\sqsubseteq) (\leq) (\lambda xs. f x xs (LCons x xs))$ 
  shows  $\text{monotone } (\sqsubseteq) (\leq) (\lambda xs. \text{case } xs \text{ of } LNil \Rightarrow \perp \mid LCons x xs \Rightarrow f x xs (LCons x xs))$ 
  by(rule llist.monotone-if-bot[where bound=LNil and bot=⊥ and f=λxs. f (lhd xs) (lhd xs) xs])(auto 4 3 simp add: not-lnull-conv LCons-lprefix-conv dest: monotoneD[OF mono] split: llist.split)

lemma mcont-lprefix-case-lfp [cont-intro, simp]:
  fixes  $f :: - \Rightarrow - :: \text{complete-lattice}$ 
  assumes  $\bigwedge x. \text{mcont } lSup (\sqsubseteq) Sup (\leq) (\lambda xs. f x xs (LCons x xs))$ 
  shows  $\text{mcont } lSup (\sqsubseteq) Sup (\leq) (\lambda xs. \text{case } xs \text{ of } LNil \Rightarrow \perp \mid LCons x xs \Rightarrow f x xs (LCons x xs))$ 
  using assms by(rule mcont-lprefix-case-aux)(rule complete-lattice-ccpo', simp)

declaration <Partial-Function.init llist @{term llist.fixp-fun}
@{term llist.mono-body} @{thm llist.fixp-rule-uc} @{thm llist.fixp-strong-induct-uc}
NONE>

```

2.8 Monotonicity and continuity of already defined functions

```

lemma fixes  $f F$ 
  defines  $F \equiv \lambda lmap xs. \text{case } xs \text{ of } LNil \Rightarrow LNil \mid LCons x xs \Rightarrow LCons (f x) (lmap xs)$ 
  shows lmap-conv-fixp:  $\text{lmap } f \equiv \text{ccpo.fixp } (\text{fun-lub } lSup) (\text{fun-ord } (\sqsubseteq)) F$  (is ?lhs ≡ ?rhs)
  and lmap-mono:  $\bigwedge xs. \text{mono-llist } (\lambda lmap. F lmap xs)$  (is PROP ?mono)
  proof(intro eq-reflection ext)
    show mono: PROP ?mono unfolding F-def by(tactic <Partial-Function.mono-tac
@{context} 1)
    fix xs
    show ?lhs xs = ?rhs xs
    proof(coinduction arbitrary: xs)
      case Eq-llist
        show ?case by(subst (1 3 4) llist.mono-body-fixp[OF mono])(auto simp add:
F-def split: llist.split)
      qed
    qed
  qed

lemma mono2mono-lmap[THEN llist.mono2mono, simp, cont-intro]:
  shows monotone-lmap: monotone ( $\sqsubseteq$ ) ( $\sqsubseteq$ ) (lmap f)

```

```

by(rule llist.fixp-preserves-mono1[OF lmap-mono lmap-conv-fixp]) simp

lemma mcont2mcont-lmap[THEN llist.mcont2mcont, simp, cont-intro]:
  shows mcont-lmap: mcont lSup ( $\sqsubseteq$ ) lSup ( $\sqsubseteq$ ) (lmap f)
by(rule llist.fixp-preserves-mcont1[OF lmap-mono lmap-conv-fixp]) simp

lemma [partial-function-mono]: mono-llist F  $\implies$  mono-llist ( $\lambda f. lmap g (F f)$ )
by(rule mono2mono-lmap)

lemma mono-llist-lappend2 [partial-function-mono]:
  mono-llist A  $\implies$  mono-llist ( $\lambda f. lappend xs (A f)$ )
by(auto intro!: monotoneI lprefix-lappend-sameI simp add: fun-ord-def dest: monotoneD)

lemma mono2mono-lappend2 [THEN llist.mono2mono, cont-intro, simp]:
  shows monotone-lappend2: monotone ( $\sqsubseteq$ ) ( $\sqsubseteq$ ) (lappend xs)
by(rule monotoneI)(rule lprefix-lappend-sameI)

lemma mcont2mcont-lappend2 [THEN llist.mcont2mcont, cont-intro, simp]:
  shows mcont-lappend2: mcont lSup ( $\sqsubseteq$ ) lSup ( $\sqsubseteq$ ) (lappend xs)
proof(cases lfinite xs)
  case True
  thus ?thesis by induct(simp-all add: monotone-lappend2)
next
  case False
  hence lappend xs = ( $\lambda -. xs$ ) by(simp add: fun-eq-iff lappend-inf)
  thus ?thesis by(simp add: ccpo.cont-const[OF llist-ccpo])
qed

lemma fixes f F
  defines F  $\equiv$   $\lambda lset xs. \text{case } xs \text{ of } LNil \Rightarrow \{\} \mid LCons x xs \Rightarrow insert x (lset xs)$ 
  shows lset-conv-fixp: lset  $\equiv$  ccpo.fixp (fun-lub Union) (fun-ord ( $\sqsubseteq$ )) F (is -  $\equiv$  ?fixp)
    and lset-mono:  $\bigwedge x. \text{monotone} (\text{fun-ord} (\sqsubseteq)) (\sqsubseteq) (\lambda f. F f x)$  (is PROP ?mono)
proof(rule eq-reflection ext antisym subsetI)+
  show mono: PROP ?mono unfolding F-def by(tactic ‹Partial-Function.mono-tac @{context} 1›)
  fix x xs
  show ?fixp xs  $\subseteq$  lset xs
    by(induct arbitrary: xs rule: lfp.fixp-induct-uc[of  $\lambda x. x F \lambda x. x$ , OF mono reflexive refl])(auto simp add: F-def split: llist.split)

  assume x  $\in$  lset xs
  thus x  $\in$  ?fixp xs by induct(subst lfp.mono-body-fixp[OF mono], simp add: F-def)+
qed

lemma mono2mono-lset [THEN lfp.mono2mono, cont-intro, simp]:
  shows monotone-lset: monotone ( $\sqsubseteq$ ) ( $\sqsubseteq$ ) lset

```

```

by(rule lfp.fixp-preserves-mono1[OF lset-mono lset-conv-fixp]) simp

lemma mcont2mcont-lset[THEN mcont2mcont, cont-intro, simp]:
  shows mcont-lset: mcont lSup ( $\sqsubseteq$ ) Union ( $\sqsubseteq$ ) lset
by(rule lfp.fixp-preserves-mcont1[OF lset-mono lset-conv-fixp]) simp

lemma lset-lSup: Complete-Partial-Order.chain ( $\sqsubseteq$ ) Y  $\implies$  lset (lSup Y) =  $\bigcup$  (lset ` Y)
by(cases Y = {}) (simp-all add: mcont-lset[THEN mcont-contD])

lemma lfinite-lSupD: lfinite (lSup A)  $\implies$   $\forall$  xs  $\in$  A. lfinite xs
by(induct ys $\equiv$ lSup A arbitrary: A rule: lfinite-induct) fastforce+

lemma monotone-enat-le-lprefix-case [cont-intro, simp]:
  monotone ( $\leq$ ) ( $\sqsubseteq$ ) ( $\lambda$ x. f x (eSuc x))  $\implies$  monotone ( $\leq$ ) ( $\sqsubseteq$ ) ( $\lambda$ x. case x of 0  $\Rightarrow$  LNil | eSuc x'  $\Rightarrow$  f x' x)
by(erule monotone-enat-le-case) simp-all

lemma mcont-enat-le-lprefix-case [cont-intro, simp]:
  assumes mcont Sup ( $\leq$ ) lSup ( $\sqsubseteq$ ) ( $\lambda$ x. f x (eSuc x))
  shows mcont Sup ( $\leq$ ) lSup ( $\sqsubseteq$ ) ( $\lambda$ x. case x of 0  $\Rightarrow$  LNil | eSuc x'  $\Rightarrow$  f x' x)
  using llist-ccpo assms by(rule mcont-enat-le-case) simp

lemma compact-LConsI:
  assumes ccpo.compact lSup ( $\sqsubseteq$ ) xs
  shows ccpo.compact lSup ( $\sqsubseteq$ ) (LCons x xs)
  using llist-ccpo
  proof(rule ccpo.compactI)
    from assms have ccpo.admissible lSup ( $\sqsubseteq$ ) ( $\lambda$ ys.  $\neg$  xs  $\sqsubseteq$  ys) by cases
    hence [cont-intro]: ccpo.admissible lSup ( $\sqsubseteq$ ) ( $\lambda$ ys.  $\neg$  xs  $\sqsubseteq$  ltl ys)
      using mcont-ltl by(rule admissible-subst)
    have [cont-intro]: ccpo.admissible lSup ( $\sqsubseteq$ ) ( $\lambda$ ys.  $\neg$  lnull ys  $\wedge$  lhd ys  $\neq$  x)
    proof(rule ccpo.admissibleI)
      fix Y
      assume chain: Complete-Partial-Order.chain ( $\sqsubseteq$ ) Y
      and *: Y  $\neq$  {}  $\forall$  ys $\in$ Y.  $\neg$  lnull ys  $\wedge$  lhd ys  $\neq$  x
      from * show  $\neg$  lnull (lSup Y)  $\wedge$  lhd (lSup Y)  $\neq$  x
        by(subst lhd-lSup)(auto 4 4 dest: chainD[OF chain] intro!: the-equality[symmetric]
        dest: lprefix-lhdD)
      qed
    have eq: ( $\lambda$ ys.  $\neg$  LCons x xs  $\sqsubseteq$  ys) = ( $\lambda$ ys.  $\neg$  xs  $\sqsubseteq$  ltl ys  $\vee$  ys = LNil  $\vee$   $\neg$  lnull ys  $\wedge$  lhd ys  $\neq$  x)
      by(auto simp add: fun-eq-iff LCons-lprefix-conv neq-LNil-conv)
    show ccpo.admissible lSup ( $\sqsubseteq$ ) ( $\lambda$ ys.  $\neg$  LCons x xs  $\sqsubseteq$  ys)
      by(simp add: eq)
    qed

lemma compact-LConsD:

```

```

assumes ccpo.compact lSup ( $\sqsubseteq$ ) (LCons x xs)
shows ccpo.compact lSup ( $\sqsubseteq$ ) xs
using llist-cppo
proof(rule ccpo.compactI)
  from assms have ccpo.admissible lSup ( $\sqsubseteq$ ) ( $\lambda ys. \neg LCons x xs \sqsubseteq ys$ ) by cases
  hence ccpo.admissible lSup ( $\sqsubseteq$ ) ( $\lambda ys. \neg LCons x xs \sqsubseteq LCons x ys$ )
    by(rule admissible-subst)(rule mcont-LCons)
  thus ccpo.admissible lSup ( $\sqsubseteq$ ) ( $\lambda ys. \neg xs \sqsubseteq ys$ ) by simp
qed

lemma compact-LCons-iff [simp]:
  ccpo.compact lSup ( $\sqsubseteq$ ) (LCons x xs)  $\longleftrightarrow$  ccpo.compact lSup ( $\sqsubseteq$ ) xs
by(blast intro: compact-LConsI compact-LConsD)

lemma compact-lfiniteI:
  lfinite xs  $\implies$  ccpo.compact lSup ( $\sqsubseteq$ ) xs
by(induction rule: lfinite.induct) simp-all

lemma compact-lfiniteD:
  assumes ccpo.compact lSup ( $\sqsubseteq$ ) xs
  shows lfinite xs
proof(rule ccontr)
  assume inf:  $\neg$  lfinite xs

  from assms have ccpo.admissible lSup ( $\sqsubseteq$ ) ( $\lambda ys. \neg xs \sqsubseteq ys$ ) by cases
  moreover let ?C = {ys. ys  $\sqsubseteq$  xs  $\wedge$  ys  $\neq$  xs}
  have Complete-Partial-Order.chain ( $\sqsubseteq$ ) ?C
    using lprefixes-chain[of xs] by(auto dest: chain-compr)
  moreover have ?C  $\neq \{\}$  using inf by(cases xs) auto
  ultimately have  $\neg xs \sqsubseteq lSup ?C$ 
    by(rule ccpo.admissibleD)(auto dest: lprefix-antisym)
  also have lSup ?C = xs using inf by(rule lSup-strict-prefixes)
  finally show False by simp
qed

lemma compact-eq-lfinite [simp]: ccpo.compact lSup ( $\sqsubseteq$ ) = lfinite
by(blast intro: compact-lfiniteI compact-lfiniteD)

```

2.9 More function definitions

```

primcorec iterates :: ('a  $\Rightarrow$  'a)  $\Rightarrow$  'a  $\Rightarrow$  'a llist
where iterates f x = LCons x (iterates f (f x))

```

```

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

```

```

definition list-of :: 'a llist  $\Rightarrow$  'a list

```

```

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

definition llength :: 'a llist  $\Rightarrow$  enat
where [code del]:
  llength = enat-unfold lnull ltl

primcorec ltake :: enat  $\Rightarrow$  'a llist  $\Rightarrow$  'a llist
where
  n = 0  $\vee$  lnull xs  $\implies$  lnull (ltake n xs)
  | lhd (ltake n xs) = lhd xs
  | ltl (ltake n xs) = ltake (epred n) (ltl xs)

definition ldropn :: nat  $\Rightarrow$  'a llist  $\Rightarrow$  'a llist
where ldropn n xs = (ltl  $\wedge^n$  n) xs

context notes [[function-internals]]
begin

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

end

primcorec ltakeWhile :: ('a  $\Rightarrow$  bool)  $\Rightarrow$  'a llist  $\Rightarrow$  'a llist
where
  lnull xs  $\vee$   $\neg$  P (lhd xs)  $\implies$  lnull (ltakeWhile P xs)
  | lhd (ltakeWhile P xs) = lhd xs
  | ltl (ltakeWhile P xs) = ltakeWhile P (ltl xs)

context fixes P :: 'a  $\Rightarrow$  bool
notes [[function-internals]]
begin

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

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

end

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

```

```

n)

declare lnth.simps [simp del]

primcorec lzip :: 'a llist  $\Rightarrow$  'b llist  $\Rightarrow$  ('a  $\times$  'b) llist
where
  lnull xs  $\vee$  lnull ys  $\Longrightarrow$  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  $\Rightarrow$  'a
where [nitpick-simp]:
  llast xs = (case llength xs of enat n  $\Rightarrow$  (case n of 0  $\Rightarrow$  undefined  $|$  Suc n'  $\Rightarrow$  lnth xs n'  $|$   $\infty \Rightarrow undefined$ ))

coinductive ldistinct :: 'a llist  $\Rightarrow$  bool
where
  LNil [simp]: ldistinct LNil
  | LCons: [xnotin lset xs; ldistinct xs]  $\Longrightarrow$  ldistinct (LCons x xs)

hide-fact (open) LNil LCons

definition inf_llist :: (nat  $\Rightarrow$  'a)  $\Rightarrow$  'a llist
where [code del]: inf_llist f = lmap f (iterates Suc 0)

abbreviation repeat :: 'a  $\Rightarrow$  'a llist
where repeat  $\equiv$  iterates ( $\lambda x. x$ )

definition lstrict-prefix :: 'a llist  $\Rightarrow$  'a llist  $\Rightarrow$  bool
where [code del]: lstrict-prefix xs ys  $\equiv$  xs ⊑ ys  $\wedge$  xs ≠ ys

longest common prefix

definition llcp :: 'a llist  $\Rightarrow$  'a llist  $\Rightarrow$  enat
where [code del]:
  llcp xs ys =
    enat-unfold ( $\lambda (xs, ys).$  lnull xs  $\vee$  lnull ys  $\vee$  lhd xs ≠ lhd ys) (map-prod ltl ltl)
  (xs, ys)

coinductive llexord :: ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  'a llist  $\Rightarrow$  'a llist  $\Rightarrow$  bool
for r :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool
where
  llexord-LCons-eq: llexord r xs ys  $\Longrightarrow$  llexord r (LCons x xs) (LCons x ys)
  | llexord-LCons-less: r x y  $\Longrightarrow$  llexord r (LCons x xs) (LCons y ys)
  | llexord-LNil [simp, intro!]: llexord r LN nil ys

context notes [[function-internals]]
begin

partial-function (llist) lconcat :: 'a llist llist  $\Rightarrow$  'a llist

```

```

where lconcat xs = (case xs 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'-simp[simp]:
lhd' LNil = None
lhd' (LCons x xs) = Some x
unfolding lhd'-def by auto

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

lemma ltl'-simp[simp]:
ltl' LNil = None
ltl' (LCons x xs) = Some xs
unfolding ltl'-def by auto

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
by(cases xs)(simp-all add: hd-def lhd-def)

lemma ltl-llist-of [simp]: ltl (llist-of xs) = llist-of (tl xs)
by(cases xs) simp-all

lemma lfinite-llist-of [simp]: lfinite (llist-of xs)
by(induct xs) auto

lemma lfinite-eq-range-llist-of: lfinite xs ↔ xs ∈ range llist-of
proof
assume lfinite xs
thus xs ∈ range llist-of
by(induct rule: lfinite.induct)(auto intro: llist-of.simps[symmetric])
next
assume xs ∈ range llist-of
thus lfinite xs by(auto intro: lfinite-llist-of)
qed

lemma lnull-llist-of [simp]: lnull (llist-of xs) ↔ xs = []

```

```

by(cases xs) simp-all

lemma llist-of-eq-LNil-conv:
  llist-of xs = LNil  $\longleftrightarrow$  xs = []
by(cases xs) simp-all

lemma llist-of-eq-LCons-conv:
  llist-of xs = LCons y ys  $\longleftrightarrow$  ( $\exists$  xs'. xs = y # xs'  $\wedge$  ys = llist-of xs')
by(cases xs) auto

lemma lappend-llist-of-llist-of:
  lappend (llist-of xs) (llist-of ys) = llist-of (xs @ ys)
by(induct xs) simp-all

lemma lfinite-rev-induct [consumes 1, case-names Nil snoc]:
  assumes fin: lfinite xs
  and Nil: P LNil
  and snoc:  $\bigwedge$  x xs. [[ lfinite xs; P xs ]]  $\implies$  P (lappend xs (LCons x LNil))
  shows P xs
proof -
  from fin obtain xs' where xs: xs = llist-of xs'
  unfolding lfinite-eq-range-llist-of by blast
  show ?thesis unfolding xs
    by(induct xs' rule: rev-induct)(auto simp add: Nil lappend-llist-of-llist-of[symmetric]
      dest: snoc[rotated])
qed

lemma lappend-llist-of-LCons:
  lappend (llist-of xs) (LCons y ys) = lappend (llist-of (xs @ [y])) ys
by(induct xs) simp-all

lemma lmap-llist-of [simp]:
  lmap f (llist-of xs) = llist-of (map f xs)
by(induct xs) simp-all

lemma lset-llist-of [simp]: lset (llist-of xs) = set xs
by(induct xs) simp-all

lemma llist-of-inject [simp]: llist-of xs = llist-of ys  $\longleftrightarrow$  xs = ys
proof
  assume llist-of xs = llist-of ys
  thus xs = ys
  proof(induct xs arbitrary: ys)
    case Nil thus ?case by(cases ys) auto
  next
    case Cons thus ?case by(cases ys) auto
  qed
qed simp

```

```
lemma inj-llist-of [simp]: inj llist-of
by(rule inj-onI) simp
```

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

```
lemma list-of-llist-of [simp]: list-of (llist-of xs) = xs
by(fastforce simp add: list-of-def intro: inv-f-f inj-onI)
```

```
lemma llist-of-list-of [simp]: lfinite xs ==> llist-of (list-of xs) = xs
unfolding lfinite-eq-range-llist-of by auto
```

```
lemma list-of-LNil [simp, nitpick-simp]: list-of LNil = []
using list-of-llist-of[of []] by simp
```

```
lemma list-of-LCons [simp]: lfinite xs ==> list-of (LCons x xs) = x # list-of xs
proof(induct arbitrary: x rule: lfinite.induct)
  case lfinite-LNil
    show ?case using list-of-llist-of[of [x]] by simp
  next
    case (lfinite-LConsI xs' x')
      from ‹list-of (LCons x' xs') = x' # list-of xs'› show ?case
        using list-of-llist-of[of x # x' # list-of xs']
          llist-of-list-of[OF ‹lfinite xs'›] by simp
  qed
```

```
lemma list-of-LCons-conv [nitpick-simp]:
  list-of (LCons x xs) = (if lfinite xs then x # list-of xs else undefined)
by(auto)(auto simp add: list-of-def)
```

```
lemma list-of-lappend:
  assumes lfinite xs lfinite ys
  shows list-of (lappend xs ys) = list-of xs @ list-of ys
using lfinite_xs by induct(simp-all add: lfinite_ys)
```

```
lemma list-of-lmap [simp]:
  assumes lfinite xs
  shows list-of (lmap f xs) = map f (list-of xs)
using assms by induct simp-all
```

```
lemma set-list-of [simp]:
  assumes lfinite xs
  shows set (list-of xs) = lset xs
using assms by(induct)(simp-all)
```

```
lemma hd-list-of [simp]: lfinite xs ==> hd (list-of xs) = lhd xs
by(clarify simp add: lfinite-eq-range-llist-of)
```

```
lemma tl-list-of: lfinite xs ==> tl (list-of xs) = list-of (ltl xs)
by(auto simp add: lfinite-eq-range-llist-of)
```

Efficient implementation via tail recursion suggested by Brian Huffman

```
definition list-of-aux :: 'a list  $\Rightarrow$  'a llist  $\Rightarrow$  '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 []
by(simp add: fun-eq-iff list-of-def list-of-aux-def)

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
by(simp-all add: list-of-aux-def)
```

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)
by(simp-all add: llength-def enat-unfold)
```

```
lemma llength-eq-0 [simp]: llength xs = 0  $\longleftrightarrow$  lnull xs
by(cases xs) simp-all
```

```
lemma llength-lnull [simp]: lnull xs  $\Longrightarrow$  llength xs = 0
by simp
```

```
lemma epred-llength:
  epred (llength xs) = llength (ltl xs)
by(cases xs) simp-all
```

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

```
lemma llength-lmap [simp]: llength (lmap f xs) = llength xs
by(coinduction arbitrary: xs rule: enat-coinduct)(auto simp add: epred-llength)
```

```
lemma llength-lappend [simp]: llength (lappend xs ys) = llength xs + llength ys
by(coinduction arbitrary: xs ys rule: enat-coinduct)(simp-all add: iadd-is-0 epred-iadd1
split-paired-all epred-llength, auto)
```

```
lemma llength-llist-of [simp]:
  llength (llist-of xs) = enat (length xs)
by(induct xs)(simp-all add: zero-enat-def eSuc-def)
```

```
lemma length-list-of:
  lfinite xs  $\Longrightarrow$  enat (length (list-of xs)) = llength xs
apply(rule sym)
by(induct rule: lfinite.induct)(auto simp add: eSuc-enat zero-enat-def)
```

```
lemma length-list-of-conv-the-enat:
  lfinite xs  $\Longrightarrow$  length (list-of xs) = the-enat (llength xs)
```

```

unfolding lfinite-eq-range-llist-of by auto

lemma llength-eq-enat-lfiniteD: llength xs = enat n ==> lfinite xs
proof(induct n arbitrary: xs)
  case [folded zero-enat-def]: 0
    thus ?case by simp
  next
    case (Suc n)
    note len = <llength xs = enat (Suc n)>
    then obtain x xs' where xs = LCons x xs'
      by(cases xs)(auto simp add: zero-enat-def)
    moreover with len have llength xs' = enat n
      by(simp add: eSuc-def split: enat.split-asm)
    hence lfinite xs' by(rule Suc)
    ultimately show ?case by simp
qed

lemma lfinite-llength-enat:
  assumes lfinite xs
  shows ∃n. llength xs = enat n
  using assms
  by induct(auto simp add: eSuc-def zero-enat-def)

lemma lfinite-conv-llength-enat:
  lfinite xs ↔ (∃n. llength xs = enat n)
  by(blast dest: llength-eq-enat-lfiniteD lfinite-llength-enat)

lemma not-lfinite-llength:
  ¬ lfinite xs ==> llength xs = ∞
  by(simp add: lfinite-conv-llength-enat)

lemma llength-eq-infty-conv-lfinite:
  llength xs = ∞ ↔ ¬ lfinite xs
  by(simp add: lfinite-conv-llength-enat)

lemma lfinite-finite-index: lfinite xs ==> finite {n. enat n < llength xs}
by(auto dest: lfinite-llength-enat)

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
  by(simp-all add: gen-llength-def iadd-Suc eSuc-enat[symmetric] iadd-Suc-right)

lemma llength-code [code]: llength = gen-llength 0
by(simp add: gen-llength-def fun-eq-iff zero-enat-def)

```

```

lemma fixes  $F$ 
  defines  $F \equiv \lambda llength\ xs. \text{case } xs \text{ of } LNil \Rightarrow 0 \mid LCons\ x\ xs \Rightarrow eSuc\ (llength\ xs)$ 
  shows  $llength\text{-conv-fixp}: llength \equiv \text{ccpo.fixp}(\text{fun-lub Sup})(\text{fun-ord } (\leq)) F$  (is -  

     $\equiv ?fixp$ )
    and  $llength\text{-mono}: \bigwedge xs. \text{monotone } (\text{fun-ord } (\leq)) (\leq) (\lambda llength. F\ llength\ xs)$  (is  

       $PROP\ ?mono$ )
  proof(intro eq-reflection ext)
    show  $mono: PROP\ ?mono$  unfolding  $F\text{-def}$  by(tactic ‹Partial-Function.mono-tac  

    @{context} 1›)
    fix  $xs$ 
    show  $llength\ xs = ?fixp\ xs$ 
    by(coinduction arbitrary:  $xs$  rule: enat-coinduct)(subst (1 2 3) lfp.mono-body-fixp[OF  

      mono], fastforce simp add:  $F\text{-def}$  split: llist.split del: iffI)+
  qed

```

```

lemma  $\text{mono2mono}\text{-llength}[THEN lfp.mono2mono, simp, cont-intro]:$ 
  shows  $\text{monotone}\text{-llength}: \text{monotone } (\sqsubseteq) (\leq) llength$ 
  by(rule lfp.fixp-preserves-mono1[OF llength-mono llength-conv-fixp])(fold bot-enat-def,  

    simp)

```

```

lemma  $\text{mcont2mcont}\text{-llength}[THEN lfp.mcont2mcont, simp, cont-intro]:$ 
  shows  $\text{mcont}\text{-llength}: \text{mcont } lSup\ (\sqsubseteq) \text{Sup } (\leq) llength$ 
  by(rule lfp.fixp-preserves-mcont1[OF llength-mono llength-conv-fixp])(fold bot-enat-def,  

    simp)

```

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

```

lemma  $ltake\text{-LNil} [simp, code, nitpick-simp]: ltake\ n\ LNil = LNil$ 
  by simp

```

```

lemma  $ltake\text{-0} [simp]: ltake\ 0\ xs = LNil$ 
  by(simp add: ltake-def)

```

```

lemma  $ltake\text{-eSuc}\text{-LCons} [simp]:$ 
   $ltake\ (eSuc\ n)\ (LCons\ x\ xs) = LCons\ x\ (ltake\ n\ xs)$ 
  by(rule llist.expand)(simp-all)

```

```

lemma  $ltake\text{-eSuc}:$ 
   $ltake\ (eSuc\ n)\ xs =$ 
   $(\text{case } xs \text{ of } LNil \Rightarrow LNil \mid LCons\ x\ xs' \Rightarrow LCons\ x\ (ltake\ n\ xs'))$ 
  by(cases xs) simp-all

```

```

lemma  $lnull\text{-ltake} [simp]: lnull\ (ltake\ n\ xs) \longleftrightarrow lnull\ xs \vee n = 0$ 
  by(cases n xs rule: enat-coexhaust[case-product llist.exhaust]) simp-all

```

```

lemma  $ltake\text{-eq}\text{-LNil}\text{-iff}: ltake\ n\ xs = LNil \longleftrightarrow xs = LNil \vee n = 0$ 
  by(cases n xs rule: enat-coexhaust[case-product llist.exhaust]) simp-all

```

```

lemma LNil-eq-ltake-iff [simp]:  $LNil = ltake n xs \leftrightarrow xs = LNil \vee n = 0$ 
by(cases n xs rule: enat-coexhaust[case-product llist.exhaust]) simp-all

lemma ltake-LCons [code, nitpick-simp]:
 $ltake n (LCons x xs) =$ 
 $(\text{case } n \text{ of } 0 \Rightarrow LNil \mid eSuc n' \Rightarrow LCons x (ltake n' xs))$ 
by(rule llist.expand)(simp-all split: co.enat.split)

lemma lhd-ltake [simp]:  $n \neq 0 \implies lhd (ltake n xs) = lhd xs$ 
by(cases n xs rule: enat-coexhaust[case-product llist.exhaust]) simp-all

lemma ltl-ltake:  $ltl (ltake n xs) = ltake (\text{epred } n) (ltl xs)$ 
by(cases n xs rule: enat-coexhaust[case-product llist.exhaust]) simp-all

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

declare ltake.sel(2) [simp del]

lemma ltake-ltl:  $ltake n (ltl xs) = ltl (ltake (eSuc n) xs)$ 
by(cases xs) simp-all

lemma llength-ltake [simp]:  $llength (ltake n xs) = \min n (llength xs)$ 
by(coinduction arbitrary: n xs rule: enat-coinduct)(auto 4 3 simp add: enat-min-eq-0-iff
epred-llength ltl-ltake)

lemma ltake-lmap [simp]:  $ltake n (lmap f xs) = lmap f (ltake n xs)$ 
by(coinduction arbitrary: n xs)(auto 4 3 simp add: ltl-ltake)

lemma ltake-ltake [simp]:  $ltake n (ltake m xs) = ltake (\min n m) xs$ 
by(coinduction arbitrary: n m xs)(auto 4 4 simp add: enat-min-eq-0-iff ltl-ltake)

lemma lset-ltake:  $lset (ltake n xs) \subseteq lset xs$ 
proof(rule subsetI)
  fix x
  assume  $x \in lset (ltake n xs)$ 
  thus  $x \in lset xs$ 
  proof(induct ltake n xs arbitrary: xs n rule: lset-induct)
    case find thus ?case
      by(cases xs)(simp, cases n rule: enat-coexhaust, simp-all)
    next
      case step thus ?case
        by(cases xs)(simp, cases n rule: enat-coexhaust, simp-all)
    qed
  qed

lemma ltake-all:  $llength xs \leq m \implies ltake m xs = xs$ 
by(coinduction arbitrary: m xs)(auto simp add: epred-llength[symmetric] ltl-ltake
intro: epred-le-epredI)

```

```

lemma ltake-llist-of [simp]:
  ltake (enat n) (llist-of xs) = llist-of (take n xs)
proof(induct n arbitrary: xs)
  case 0
  thus ?case unfolding zero-enat-def[symmetric]
    by(cases xs) simp-all
next
  case (Suc n)
  thus ?case unfolding eSuc-enat[symmetric]
    by(cases xs) simp-all
qed

lemma lfinite-ltake [simp]:
  lfinite (ltake n xs)  $\longleftrightarrow$  lfinite xs  $\vee$  n <  $\infty$ 
  (is ?lhs  $\longleftrightarrow$  ?rhs)
proof
  assume ?lhs
  thus ?rhs
    by(induct ys≡ltake n xs arbitrary: n xs rule: lfinite-induct)(fastforce simp add:
      zero-enat-def ltl-ltake)+
  next
  assume ?rhs (is ?xs  $\vee$  ?n)
  thus ?lhs
    proof
      assume ?xs thus ?thesis
        by(induct xs arbitrary: n)(simp-all add: ltake-LCons split: enat-cosplit)
    next
      assume ?n
      then obtain n' where n = enat n' by auto
      moreover have lfinite (ltake (enat n') xs)
      by(induct n' arbitrary: xs)
        (auto simp add: zero-enat-def[symmetric] eSuc-enat[symmetric] ltake-eSuc
         split: llist.split)
      ultimately show ?thesis by simp
    qed
  qed

lemma ltake-lappend1: n  $\leq$  llength xs  $\implies$  ltake n (lappend xs ys) = ltake n xs
by(coinduction arbitrary: n xs)(auto intro!: exI simp add: llength-ltl epred-le-epredI
  ltl-ltake)

lemma ltake-lappend2:
  llength xs  $\leq$  n  $\implies$  ltake n (lappend xs ys) = lappend xs (ltake (n - llength xs)
  ys)
by(coinduction arbitrary: n xs rule: llist.coinduct-strong)(auto intro!: exI simp add:
  llength-ltl epred-le-epredI ltl-ltake)

lemma ltake-lappend:

```

```

ltake n (lappend xs ys) = lappend (ltake n xs) (ltake (n - llength xs) ys)
by(coinduction arbitrary: n xs ys rule: llist.coinduct-strong)(auto intro!: exI simp
add: llength-ltl ltl-ltake)

lemma take-list-of:
assumes lfinite xs
shows take n (list-of xs) = list-of (ltake (enat n) xs)
using assms
by(induct arbitrary: n)
  (simp-all add: take-Cons zero-enat-def[symmetric] eSuc-enat[symmetric] split:
nat.split)

lemma ltake-eq-ltake-antimono:
[| ltake n xs = ltake n ys; m ≤ n |] ==> ltake m xs = ltake m ys
by (metis ltake-ltake min-def)

lemma ltake-is-lprefix [simp, intro]: ltake n xs ⊑ xs
by(coinduction arbitrary: xs n)(auto simp add: ltl-ltake)

lemma lprefix-ltake-same [simp]:
ltake n xs ⊑ ltake m xs ↔ n ≤ m ∨ llength xs ≤ m
(is ?lhs ↔ ?rhs)
proof(rule iffI disjCI)+
assume ?lhs ¬ llength xs ≤ m
thus n ≤ m
proof(coinduction arbitrary: n m xs rule: enat-le-coinduct)
case (le n m xs)
thus ?case by(cases xs)(auto 4 3 simp add: ltake-LCons split: co.enat.splits)
qed
next
assume ?rhs
thus ?lhs
proof
assume n ≤ m
thus ?thesis
by(coinduction arbitrary: n m xs)(auto 4 4 simp add: ltl-ltake epred-le-epredI)
qed(simp add: ltake-all)
qed

lemma fixes f F
defines F ≡ λltake 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 (⊑)))
(λltake. case-prod (F (curry ltake)))(is ?lhs ≡ ?rhs)
and ltake-mono: ∀nxs. mono-llist (λltake. case nxs of (n, xs) ⇒ F (curry ltake)
n xs) (is PROP ?mono)
proof(intro eq-reflection ext)
show mono: PROP ?mono unfolding F-def by(tactic ‹Partial-Function.mono-tac
@{context} 1›)

```

```

fix n xs
show ?lhs n xs = ?rhs n xs
proof(coinduction arbitrary: n xs)
  case Eq-llist
    show ?case by(subst (1 3 4) llist.mono-body-fixp[OF mono])(auto simp add:
F-def split: llist.split prod.split co.enat.split)
  qed
qed

lemma monotone-ltake: monotone (rel-prod ( $\leq$ ) ( $\sqsubseteq$ )) ( $\sqsubseteq$ ) (case-prod ltake)
by(rule llist.fixp-preserves-mono2[OF ltake-mono ltake-conv-fixp]) simp

lemma mono2mono-ltake1 [THEN llist.mono2mono, cont-intro, simp]:
  shows monotone-ltake1: monotone ( $\leq$ ) ( $\sqsubseteq$ ) ( $\lambda n.$  ltake n xs)
  using monotone-ltake by auto

lemma mono2mono-ltake2 [THEN llist.mono2mono, cont-intro, simp]:
  shows monotone-ltake2: monotone ( $\sqsubseteq$ ) ( $\sqsubseteq$ ) (ltake n)
  using monotone-ltake by auto

lemma mcont-ltake: mcont (prod-lub Sup lSup) (rel-prod ( $\leq$ ) ( $\sqsubseteq$ )) lSup ( $\sqsubseteq$ ) (case-prod
ltake)
by(rule llist.fixp-preserves-mcont2[OF ltake-mono ltake-conv-fixp]) simp

lemma mcont2mcont-ltake1 [THEN llist.mcont2mcont, cont-intro, simp]:
  shows mcont-ltake1: mcont Sup ( $\leq$ ) lSup ( $\sqsubseteq$ ) ( $\lambda n.$  ltake n xs)
  using mcont-ltake by auto

lemma mcont2mcont-ltake2 [THEN llist.mcont2mcont, cont-intro, simp]:
  shows mcont-ltake2: mcont lSup ( $\sqsubseteq$ ) lSup ( $\sqsubseteq$ ) (ltake n)
  using mcont-ltake by auto

lemma [partial-function-mono]: mono-llist F  $\implies$  mono-llist ( $\lambda f.$  ltake n (F f))
by(rule mono2mono-ltake2)

lemma llist-induct2:
  assumes adm: ccpo.admissible (prod-lub lSup lSup) (rel-prod ( $\sqsubseteq$ ) ( $\sqsubseteq$ )) ( $\lambda x.$  P (fst
x)) ( $\lambda x.$  P (snd x))
  and LNil: P LNil LNil
  and LCons1:  $\bigwedge x \in xs. [\![ lfinite xs; P xs LNil ]\!] \implies P (LCons x xs) LNil$ 
  and LCons2:  $\bigwedge y \in ys. [\![ lfinite ys; P LNil ys ]\!] \implies P LNil (LCons y ys)$ 
  and LCons:  $\bigwedge x \in xs \ y \in ys. [\![ lfinite xs; lfinite ys; P xs ys ]\!] \implies P (LCons x xs)$ 
   $(LCons y ys)$ 
  shows P xs ys
proof –
  let ?C = ( $\lambda n.$  (ltake n xs, ltake n ys)) ` range enat
  have Complete-Partial-Order.chain (rel-prod ( $\sqsubseteq$ ) ( $\sqsubseteq$ )) ?C
    by(rule chainI) auto
  with adm have P (fst (prod-lub lSup lSup ?C)) (snd (prod-lub lSup lSup ?C))

```

```

proof(rule ccpo.admissibleD)
  fix xsys
  assume xsys ∈ ?C
  then obtain n where xsys = (ltake (enat n) xs, ltake (enat n) ys) by auto
  moreover {
    fix xs :: 'a llist
    assume lfinite xs
    hence P xs LNil by induct(auto intro: LNil LCons1) }
  note 1 = this
  { fix ys :: 'b llist
    assume lfinite ys
    hence P LNil ys by induct(auto intro: LNil LCons2) }
  note 2 = this
  have P (ltake (enat n) xs) (ltake (enat n) ys)
    by(induct n arbitrary: xs ys)(auto simp add: zero-enat-def[symmetric] LNil
    eSuc-enat[symmetric] ltake-eSuc split: llist.split intro: LNil LCons 1 2)
    ultimately show P (fst xsys) (snd xsys) by simp
  qed simp
  also have fst (prod-lub lSup lSup ?C) = xs
    unfolding prod-lub-def fst-conv
    by(subst image-image)(simp add: mcont-contD[OF mcont-ltake1, symmetric]
    ltake-all)
  also have snd (prod-lub lSup lSup ?C) = ys
    unfolding prod-lub-def snd-conv
    by(subst image-image)(simp add: mcont-contD[OF mcont-ltake1, symmetric]
    ltake-all)
  finally show ?thesis .
qed

lemma ldropn-0 [simp]: ldropn 0 xs = xs
by(simp add: ldropn-def)

lemma ldropn-LNil [code, simp]: ldropn n LNil = LNil
by(induct n)(simp-all add: ldropn-def)

lemma ldropn-lnull: lnull xs ==> ldropn n xs = LNil
by(simp add: lnull-def)

lemma ldropn-LCons [code]:
  ldropn n (LCons x xs) = (case n of 0 => LCons x xs | Suc n' => ldropn n' xs)
by(cases n)(simp-all add: ldropn-def funpow-swap1)

lemma ldropn-Suc: ldropn (Suc n) xs = (case xs of LNil => LNil | LCons x xs' =>
ldropn n xs')
by(simp split: llist.split)(simp add: ldropn-def funpow-swap1)

lemma ldropn-Suc-LCons [simp]: ldropn (Suc n) (LCons x xs) = ldropn n xs
by(simp add: ldropn-LCons)

```

```

lemma ltl-ldropn: ltl (ldropn n xs) = ldropn n (ltl xs)
proof(induct n arbitrary: xs)
  case 0 thus ?case by simp
next
  case (Suc n)
  thus ?case
    by(cases xs)(simp-all add: ldropn-Suc split: llist.split)
qed

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
  by(simp-all add: ldrop.simps split: co.enat.split)

lemma ldrop-lnull: lnull xs  $\implies$  ldrop n xs = LNil
  by(simp add: lnull-def)

lemma fixes f F
  defines F  $\equiv$   $\lambda$ ldropn xs. case xs of LNil  $\Rightarrow$   $\lambda$ . LNil | 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-lub (fun-lub lSup)) (fun-ord (fun-ord lprefix)) ( $\lambda$ ldrop. F ldrop) (is ?lhs  $\equiv$  ?rhs)
  and ldrop-mono:  $\bigwedge$ xs. mono-llist-lift ( $\lambda$ ldrop. F ldrop xs) (is PROP ?mono)
  proof(intro eq-reflection ext)
    show mono: PROP ?mono unfolding F-def by(tactic <Partial-Function.mono-tac @{context} 1)
    fix n xs
    show ?lhs xs n = ?rhs xs n
      by(induction n arbitrary: xs)
        (subst llist-lift.mono-body-fixp[OF mono], simp add: F-def split: llist.split)+
  qed

lemma ldropn-fixp-case-conv:
   $(\lambda$ xs. case xs of LNil  $\Rightarrow$   $\lambda$ . LNil | 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 | LCons x xs  $\Rightarrow$  if n = 0 then LCons x xs else f xs (n - 1))
  by(auto simp add: fun-eq-iff split: llist.split)

lemma monotone-ldropn-aux: monotone lprefix (fun-ord lprefix) ( $\lambda$ xs n. ldropn n xs)
  by(rule llist-lift.fixp-preserves-mono1[OF ldrop-mono ldrop-conv-fixp])
    (simp add: ldropn-fixp-case-conv monotone-fun-ord-apply)

lemma mono2mono-ldropn[THEN llist.mono2mono, cont-intro, simp]:
  shows monotone-ldropn': monotone lprefix lprefix ( $\lambda$ xs. ldropn n xs)
  using monotone-ldropn-aux by(auto simp add: monotone-def fun-ord-def)

```

```

lemma mcont-ldropn-aux: mcont lSup lprefix (fun-lub lSup) (fun-ord lprefix) ( $\lambda xs$   

 $n$ . ldropn n xs)  

by(rule llist-lift.fixp-preserves-mcont1[OF ldrop-mono ldrop-conv-fixp])  

(simp add: ldropn-fixp-case-conv mcont-fun-lub-apply)  

  

lemma mcont2mcont-ldropn [THEN llist.mcont2mcont, cont-intro, simp]:  

shows mcont-ldropn: mcont lSup lprefix lSup lprefix (ldropn n)  

using mcont-ldropn-aux by(auto simp add: mcont-fun-lub-apply)  

  

lemma monotone-enat-cocase [cont-intro, simp]:  

 $\llbracket \bigwedge n. \text{monotone } (\leq) \text{ ord } (\lambda n. f n (eSuc n));$   

 $\bigwedge n. \text{ord } a (f n (eSuc n)); \text{ord } a a \rrbracket$   

 $\implies \text{monotone } (\leq) \text{ ord } (\lambda n. \text{case } n \text{ of } 0 \Rightarrow a \mid eSuc n' \Rightarrow f n' n)$   

by(rule monotone-enat-le-case)  

  

lemma monotone-ldrop: monotone (rel-prod (=) ( $\sqsubseteq$ )) ( $\sqsubseteq$ ) (case-prod ldrop)  

by(rule llist.fixp-preserves-mono2[OF ldrop.mono ldrop-def]) simp  

  

lemma mono2mono-ldrop2 [THEN llist.mono2mono, cont-intro, simp]:  

shows monotone-ldrop2: monotone ( $\sqsubseteq$ ) ( $\sqsubseteq$ ) (ldrop n)  

by(simp add: monotone-ldrop[simplified])  

  

lemma mcont-ldrop: mcont (prod-lub the-Sup lSup) (rel-prod (=) ( $\sqsubseteq$ )) lSup ( $\sqsubseteq$ )  

(case-prod ldrop)  

by(rule llist.fixp-preserves-mcont2[OF ldrop.mono ldrop-def]) simp  

  

lemma mcont2monct-ldrop2 [THEN llist.mcont2mcont, cont-intro, simp]:  

shows mcont-ldrop2: mcont lSup ( $\sqsubseteq$ ) lSup ( $\sqsubseteq$ ) (ldrop n)  

by(simp add: mcont-ldrop[simplified])  

  

lemma ldrop-eSuc-conv-ltl: ldrop (eSuc n) xs = ltl (ldrop n xs)  

proof(induct xs arbitrary: n)  

case LCons thus ?case by(cases n rule: co.enat.exhaust) simp-all  

qed simp-all  

  

lemma ldrop-ltl: ldrop n (ltl xs) = ldrop (eSuc n) xs  

proof(induct xs arbitrary: n)  

case LCons thus ?case by(cases n rule: co.enat.exhaust) simp-all  

qed simp-all  

  

lemma lnull-ldropn [simp]: lnull (ldropn n xs)  $\longleftrightarrow$  llength xs  $\leq$  enat n  

proof(induction n arbitrary: xs)  

case (Suc n)  

from Suc.IH[of ltl xs] show ?case  

by(cases xs)(simp-all add: eSuc-enat[symmetric])  

qed(simp add: zero-enat-def[symmetric])  

  

lemma ldrop-eq-LNil [simp]: ldrop n xs = LNil  $\longleftrightarrow$  llength xs  $\leq$  n  

proof(induction xs arbitrary: n)

```

```

case (LCons x xs n)
  thus ?case by(cases n rule: co.enat.exhaust) simp-all
qed simp-all

lemma lnull-ldrop [simp]: lnull (ldrop n xs)  $\longleftrightarrow$  llength xs  $\leq$  n
unfolding lnull-def by(fact ldrop-eq-LNil)

lemma ldropn-eq-LNil: (ldropn n xs = LNil) = (llength xs  $\leq$  enat n)
using lnull-ldropn unfolding lnull-def .

lemma ldropn-all: llength xs  $\leq$  enat m  $\implies$  ldropn m xs = LNil
by(simp add: ldropn-eq-LNil)

lemma ldrop-all: llength xs  $\leq$  m  $\implies$  ldrop m xs = LNil
by(simp)

lemma ltl-ldrop: ltl (ldrop n xs) = ldrop n (ltl xs)
by(simp add: ldrop-eSuc-conv-ltl ldrop-ltl)

lemma ldrop-eSuc:
ldrop (eSuc n) xs = (case xs of LNil  $\Rightarrow$  LNil  $|$  LCons x xs'  $\Rightarrow$  ldrop n xs')
by(cases xs) simp-all

lemma ldrop-LCons:
ldrop n (LCons x xs) = (case n of 0  $\Rightarrow$  LCons x xs  $|$  eSuc n'  $\Rightarrow$  ldrop n' xs')
by(cases n rule: enat-coexhaust) simp-all

lemma ldrop-inf [code, simp]: ldrop  $\infty$  xs = LNil
by(induction xs)(simp-all add: ldrop-LCons enat-cocase-inf)

lemma ldrop-enat [code]: ldrop (enat n) xs = ldropn n xs
proof(induct n arbitrary: xs)
  case Suc thus ?case
    by(cases xs)(simp-all add: eSuc-enat[symmetric])
  qed(simp add: zero-enat-def[symmetric])

lemma lfinite-ldropn [simp]: lfinite (ldropn n xs) = lfinite xs
by(induct n arbitrary: xs)(simp-all add: ldropn-Suc split: llist.split)

lemma lfinite-ldrop [simp]:
lfinite (ldrop n xs)  $\longleftrightarrow$  lfinite xs  $\vee$  n =  $\infty$ 
by(cases n)(simp-all add: ldrop-enat)

lemma ldropn-ltl: ldropn n (ltl xs) = ldropn (Suc n) xs
by(simp add: ldropn-def funpow-swap1)

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

lemma lset-ldropn-subset: lset (ldropn n xs)  $\subseteq$  lset xs

```

```

by(induct n arbitrary: xs)(fastforce simp add: ldropn-Suc split: llist.split-asm) +
lemma in-lset-ldropnD: x ∈ lset (ldropn n xs) ==> x ∈ lset xs
using lset-ldropn-subset[of n xs] by auto

lemma lset-ldrop-subset: lset (ldrop n xs) ⊆ lset xs
proof(induct xs arbitrary: n)
  case LCons thus ?case
    by(cases n rule: co.enat.exhaust) auto
qed simp-all

lemma in-lset-ldropD: x ∈ lset (ldrop n xs) ==> x ∈ lset xs
using lset-ldrop-subset[of n xs] by(auto)

lemma lappend-ltake-ldrop: lappend (ltake n xs) (ldrop n xs) = xs
by(coinduction arbitrary: n xs rule: llist.coinduct-strong)
  (auto simp add: ldrop-ltl ltl-ltake intro!: arg-cong2[where f=lappend])

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(induct n arbitrary: xs)
  case 0
  thus ?case by(simp add: zero-enat-def[symmetric] lappend-lnull1)
next
  case (Suc n)
  { fix zs
    assume llength zs ≤ enat n
    hence the-enat (eSuc (llength zs)) = Suc (the-enat (llength zs))
      by(auto intro!: the-enat-eSuc iff del: not-infinity-eq)
    note eq = this
    from Suc show ?case
      by(cases xs)(auto simp add: not-less not-le eSuc-enat[symmetric] eq)
  qed

lemma ldropn-lappend2:
  llength xs ≤ enat n ==> ldropn n (lappend xs ys) = ldropn (n - the-enat (llength xs)) ys
  by(auto simp add: ldropn-lappend)

lemma lappend-ltake-enat-ldropn [simp]: lappend (ltake (enat n) xs) (ldropn n xs)
= xs
by(fold ldrop-enat)(rule lappend-ltake-ldrop)

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

by(cases n)(cases $\text{llength } xs$, simp-all add: $\text{ldropn-lappend not-less ldrop-enat}$)

lemma $\text{ltake-plus-conv-lappend}$:

$\text{ltake } (n + m) \text{ } xs = \text{lappend } (\text{ltake } n \text{ } xs) \text{ } (\text{ltake } m \text{ } (\text{ldrop } n \text{ } xs))$

by(coinduction arbitrary: $n \text{ } m \text{ } xs$ rule: $\text{llist.coinduct-strong}$)(auto intro!: exI simp add: iadd-is-0 ttl-ltake epred-iadd1 ldrop-ttl)

lemma ldropn-eq-LConsD :

$\text{ldropn } n \text{ } xs = \text{LCons } y \text{ } ys \implies \text{enat } n < \text{llength } xs$

proof(induct n arbitrary: xs)

case 0 thus ?case **by**(simp add: zero-enat-def[symmetric])

next

case ($\text{Suc } n$) thus ?case **by**(cases xs)(simp-all add: Suc-ile-eq)

qed

lemma ldrop-eq-LConsD :

$\text{ldrop } n \text{ } xs = \text{LCons } y \text{ } ys \implies n < \text{llength } xs$

by(rule ccontr)(simp add: not-less ldrop-all)

lemma ldropn-lmap [simp]: $\text{ldropn } n \text{ } (\text{lmap } f \text{ } xs) = \text{lmap } f \text{ } (\text{ldropn } n \text{ } xs)$

by(induct n arbitrary: xs)(simp-all add: ldropn-Suc split: llist.split)

lemma ldrop-lmap [simp]: $\text{ldrop } n \text{ } (\text{lmap } f \text{ } xs) = \text{lmap } f \text{ } (\text{ldrop } n \text{ } xs)$

proof(induct xs arbitrary: n)

case LCons thus ?case **by**(cases n rule: co.enat.exhaust) simp-all

qed simp-all

lemma ldropn-ldropn [simp]:

$\text{ldropn } n \text{ } (\text{ldropn } m \text{ } xs) = \text{ldropn } (n + m) \text{ } xs$

by(induct m arbitrary: xs)(auto simp add: ldropn-Suc split: llist.split)

lemma ldrop-ldrop [simp]:

$\text{ldrop } n \text{ } (\text{ldrop } m \text{ } xs) = \text{ldrop } (n + m) \text{ } xs$

proof(induct xs arbitrary: m)

case LCons thus ?case **by**(cases m rule: co.enat.exhaust)(simp-all add: iadd-Suc-right)

qed simp-all

lemma llength-ldropn [simp]: $\text{llength } (\text{ldropn } n \text{ } xs) = \text{llength } xs - \text{enat } n$

proof(induct n arbitrary: xs)

case 0 thus ?case **by**(simp add: zero-enat-def[symmetric])

next

case ($\text{Suc } n$) thus ?case **by**(cases xs)(simp-all add: eSuc-enat[symmetric])

qed

lemma $\text{enat-llength-ldropn}$:

$\text{enat } n \leq \text{llength } xs \implies \text{enat } (n - m) \leq \text{llength } (\text{ldropn } m \text{ } xs)$

by(cases $\text{llength } xs$) simp-all

```

lemma ldropn-llist-of [simp]: ldropn n (llist-of xs) = llist-of (drop n xs)
proof(induct n arbitrary: xs)
  case Suc thus ?case by(cases xs) simp-all
qed simp

lemma ldrop-llist-of: ldrop (enat n) (llist-of xs) = llist-of (drop n xs)
proof(induct xs arbitrary: n)
  case Cons thus ?case by(cases n)(simp-all add: zero-enat-def[symmetric] eSuc-enat[symmetric])
qed simp

lemma drop-list-of:
  lfinite xs  $\implies$  drop n (list-of xs) = list-of (ldropn n xs)
  by (metis ldropn-llist-of list-of-llist-of llist-of-list-of)

lemma llength-ldrop: llength (ldrop n xs) = (if n =  $\infty$  then 0 else llength xs - n)
proof(induct xs arbitrary: n)
  case (LCons x xs)
    thus ?case by simp(cases n rule: co.enat.exhaust, simp-all)
qed simp-all

lemma ltake-ldropn: ltake n (ldropn m xs) = ldropn m (ltake (n + enat m) xs)
by(induct m arbitrary: n xs)(auto simp add: zero-enat-def[symmetric] ldropn-Suc eSuc-enat[symmetric] iadd-Suc-right split: llist.split)

lemma ldropn-ltake: ldropn n (ltake m xs) = ltake (m - enat n) (ldropn n xs)
by(induct n arbitrary: m xs)(auto simp add: zero-enat-def[symmetric] ltake-LCons ldropn-Suc eSuc-enat[symmetric] iadd-Suc-right split: llist.split co.enat.split-asm)

lemma ltake-ldrop: ltake n (ldrop m xs) = ldrop m (ltake (n + m) xs)
by(induct xs arbitrary: n m)(simp-all add: ldrop-LCons iadd-Suc-right split: co.enat.split)

lemma ldrop-ltake: ldrop n (ltake m xs) = ltake (m - n) (ldrop n xs)
by(induct xs arbitrary: n m)(simp-all add: ltake-LCons ldrop-LCons split: co.enat.split)

```

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

```

lemma lnth-LNil:
  lnth LNil n = undefined n
  by(cases n)(simp-all add: lnth.simps)

lemma lnth-0 [simp]:
  lnth (LCons x xs) 0 = x
  by(simp add: lnth.simps)

lemma lnth-Suc-LCons [simp]:
  lnth (LCons x xs) (Suc n) = lnth xs n
  by(simp add: lnth.simps)

```

```

lemma lnth-LCons:
  lnth (LCons x xs) n = (case n of 0 => x | Suc n' => lnth xs n')
  by(cases n) simp-all

lemma lnth-LCons': lnth (LCons x xs) n = (if n = 0 then x else lnth xs (n - 1))
  by(simp add: lnth-LCons split: nat.split)

lemma lhd-conv-lnth:
  ¬ lnull xs ==> lhd xs = lnth xs 0
  by(auto simp add: lhd-def not-lnull-conv)

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

lemma lnth-ltl: ¬ lnull xs ==> lnth (ltl xs) n = lnth xs (Suc n)
  by(auto simp add: not-lnull-conv)

lemma lhd-ldropn:
  enat n < llength xs ==> lhd (ldropn n xs) = lnth xs n
  proof(induct n arbitrary: xs)
    case 0 thus ?case by(cases xs) auto
    next
      case (Suc n)
      from ⟨enat (Suc n) < llength xs⟩ obtain x xs'
        where [simp]: xs = LCons x xs' by(cases xs) auto
      from ⟨enat (Suc n) < llength xs⟩
      have enat n < llength xs' by(simp add: Suc-ile-eq)
      hence lhd (ldropn n xs') = lnth xs' n by(rule Suc)
      thus ?case by simp
    qed

lemma lhd-ldrop:
  assumes n < llength xs
  shows lhd (ldrop n xs) = lnth xs (the-enat n)
  proof –
    from assms obtain n' where n: n = enat n' by(cases n) auto
    from assms show ?thesis unfolding n
    proof(induction n' arbitrary: xs)
      case 0 thus ?case
        by(simp add: zero-enat-def[symmetric] lhd-conv-lnth)
      next
        case (Suc n')
        thus ?case
          by(cases xs)(simp-all add: eSuc-enat[symmetric], simp add: eSuc-enat)
        qed
    qed

lemma lnth-beyond:
  llength xs ≤ enat n ==> lnth xs n = undefined (n - (case llength xs of enat m => m))

```

```

proof(induct n arbitrary: xs)
  case 0 thus ?case by(simp add: zero-enat-def[symmetric] lnth-def lnull-def)
next
  case Suc thus ?case
    by(cases xs)(simp-all add: zero-enat-def lnth-def eSuc-enat[symmetric] split:
enat.split, auto simp add: eSuc-enat)
qed

lemma lnth-lmap [simp]:
  enat n < llength xs  $\implies$  lnth (lmap f xs) n = f (lnth xs n)
proof(induct n arbitrary: xs)
  case 0 thus ?case by(cases xs) simp-all
next
  case (Suc n)
  from <enat (Suc n) < llength xs obtain x xs'
  where xs: xs = LCons x xs' and len: enat n < llength xs'
  by(cases xs)(auto simp add: Suc-ile-eq)
  from len have lnth (lmap f xs') n = f (lnth xs' n) by(rule Suc)
  with xs show ?case by simp
qed

lemma lnth-ldropn [simp]:
  enat (n + m) < llength xs  $\implies$  lnth (ldropn n xs) m = lnth xs (m + n)
proof(induct n arbitrary: xs)
  case (Suc n)
  from <enat (Suc n + m) < llength xs
  obtain x xs' where xs = LCons x xs' by(cases xs) auto
  moreover with <enat (Suc n + m) < llength xs
  have enat (n + m) < llength xs' by(simp add: Suc-ile-eq)
  hence lnth (ldropn n xs') m = lnth xs' (m + n) by(rule Suc)
  ultimately show ?case by simp
qed simp

lemma lnth-ldrop [simp]:
  n + enat m < llength xs  $\implies$  lnth (ldrop n xs) m = lnth xs (m + the-enat n)
by(cases n)(simp-all add: ldrop-enat)

lemma in-lset-conv-lnth:
  x ∈ lset xs  $\longleftrightarrow$  ( $\exists$  n. enat n < llength xs  $\wedge$  lnth xs n = x)
  (is ?lhs  $\longleftrightarrow$  ?rhs)
proof
  assume ?rhs
  then obtain n where enat n < llength xs lnth xs n = x by blast
  thus ?lhs
  proof(induct n arbitrary: xs)
    case 0
    thus ?case
      by(auto simp add: zero-enat-def[symmetric] not-lnull-conv)
next

```

```

case (Suc n)
thus ?case
    by(cases xs)(auto simp add: eSuc-enat[symmetric])
qed
next
    assume ?lhs
    thus ?rhs
    proof(induct)
        case (find xs)
        show ?case by(auto intro: exI[where x=0] simp add: zero-enat-def[symmetric])
    next
        case (step x' xs)
        thus ?case
            by(auto 4 4 intro: exI[where x=Suc n for n] ileI1 simp add: eSuc-enat[symmetric])
        qed
    qed

lemma lset-conv-lnth: lset xs = {lnth xs n | n. enat n < llength xs}
by(auto simp add: in-lset-conv-lnth)

lemma lnth-llist-of [simp]: lnth (llist-of xs) = nth xs
proof(rule ext)
    fix n
    show lnth (llist-of xs) n = xs ! n
    proof(induct xs arbitrary: n)
        case Nil thus ?case by(cases n)(simp-all add: nth-def lnth-def)
    next
        case Cons thus ?case by(simp add: lnth-LCons split: nat.split)
    qed
qed

lemma nth-list-of [simp]:
assumes lfinite xs
shows nth (list-of xs) = lnth xs
using assms
by induct(auto intro: simp add: nth-def lnth-LNil nth-Cons split: nat.split)

lemma lnth-lappend1:
    enat n < llength xs  $\implies$  lnth (lappend xs ys) n = lnth xs n
proof(induct n arbitrary: xs)
    case 0 thus ?case by(auto simp add: zero-enat-def[symmetric] not-lnull-conv)
next
    case (Suc n)
    from ⟨enat (Suc n) < llength xs⟩ obtain x xs'
        where [simp]: xs = LCons x xs' and len: enat n < llength xs'
        by(cases xs)(auto simp add: Suc-ile-eq)
    from len have lnth (lappend xs' ys) n = lnth xs' n by(rule Suc)
    thus ?case by simp
qed

```

```

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(induct xs arbitrary: n)
  case (Cons x xs) thus ?case by(cases n) auto
qed simp

lemma lnth-lappend2:
  [| llength xs = enat k; k ≤ n |] ⇒ lnth (lappend xs ys) n = lnth ys (n - k)
proof(induct n arbitrary: xs k)
  case 0 thus ?case by(simp add: zero-enat-def[symmetric] lappend-lnull1)
next
  case (Suc n) thus ?case
    by(cases xs)(auto simp add: eSuc-def zero-enat-def split: enat.split-asm)
qed

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)))
by(cases llength xs)(auto simp add: lnth-lappend1 lnth-lappend2)

lemma lnth-ltake:
  enat m < n ⇒ lnth (ltake n xs) m = lnth xs m
proof(induct m arbitrary: xs n)
  case 0 thus ?case
    by(cases n rule: enat-coexhaust)(auto, cases xs, auto)
next
  case (Suc m)
  from ⟨enat (Suc m) < n⟩ obtain n' where n = eSuc n'
    by(cases n rule: enat-coexhaust) auto
  with ⟨enat (Suc m) < n⟩ have enat m < n' by(simp add: eSuc-enat[symmetric])
  with Suc ⟨n = eSuc n'⟩ show ?case by(cases xs) auto
qed

lemma ldropn-Suc-conv-ldropn:
  enat n < llength xs ⇒ LCons (lnth xs n) (ldropn (Suc n) xs) = ldropn n xs
proof(induct n arbitrary: xs)
  case 0 thus ?case by(cases xs) auto
next
  case (Suc n)
  from ⟨enat (Suc n) < llength xs⟩ obtain x xs'
    where [simp]: xs = LCons x xs' by(cases xs) auto
  from ⟨enat (Suc n) < llength xs⟩
  have enat n < llength xs' by(simp add: Suc-ile-eq)
  hence LCons (lnth xs' n) (ldropn (Suc n) xs') = ldropn n xs' by(rule Suc)
  thus ?case by simp
qed

```

```

lemma ltake-Suc-conv-snoc-lnth:
  enat m < llength xs  $\implies$  ltake (enat (Suc m)) xs = lappend (ltake (enat m) xs)
  (LCons (lnth xs m) LNil)
by(metis eSuc-enat eSuc-plus-1 ltake-plus-conv-lappend ldrop-enat ldropn-Suc-conv-ldropn
ltake-0 ltake-eSuc-LCons one-eSuc)

lemma lappend-eq-lappend-conv:
assumes len: llength xs = llength us
shows lappend xs ys = lappend us vs  $\longleftrightarrow$ 
  xs = us  $\wedge$  (lfinite xs  $\longrightarrow$  ys = vs) (is ?lhs  $\longleftrightarrow$  ?rhs)
proof
  assume ?rhs
  thus ?lhs by(auto simp add: lappend-inf)
next
  assume eq: ?lhs
  show ?rhs
  proof(intro conjI impI)
    show xs = us using len eq
    proof(coinduction arbitrary: xs us)
      case (Eq-llist xs us)
      thus ?case
        by(cases xs us rule: llist.exhaust[case-product llist.exhaust]) auto
    qed
    assume lfinite xs
    then obtain xs' where xs = llist-of xs'
    by(auto simp add: lfinite-eq-range-llist-of)
    hence lappend (llist-of xs') ys = lappend (llist-of xs') vs
    using eq `xs = us` by blast
    thus ys = vs
    by (induct xs') simp-all
  qed
qed

```

2.15 iterates

```

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

lemma lfinite-iterates [iff]:  $\neg$  lfinite (iterates f x)
proof
  assume lfinite (iterates f x)
  thus False
  by(induct zs $\equiv$ iterates f x arbitrary: x rule: lfinite-induct) auto
qed

lemma lmap-iterates: lmap f (iterates f x) = iterates f (f x)
by(coinduction arbitrary: x) auto

```

```

lemma iterates-lmap: iterates f x = LCons x (lmap f (iterates f x))
by(simp add: lmap-iterates)(rule iterates)

lemma lappend-iterates: lappend (iterates f x) xs = iterates f x
by(coinduction arbitrary: x) auto

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)
by(induct n) simp-all

lemma iterates-equality:
  assumes h: ∀x. h x = LCons x (lmap f (h x))
  shows h = iterates f
proof –
  { fix x
    have ¬ lnull (h x) lhd (h x) = x ltl (h x) = lmap f (h x)
    by(subst h, simp)+ }
  note [simp] = this

  { fix x
    define n :: nat where n = 0
    have (lmap f ∘ n) (h x) = (lmap f ∘ n) (iterates f x)
    by(coinduction arbitrary: n)(auto simp add: funpow-swap1 lmap-iterates intro:
      ext[where x=Suc n for n])
    thus ?thesis by auto
  qed

lemma llength-iterates [simp]: llength (iterates f x) = ∞
by(coinduction arbitrary: x rule: enat-coinduct)(auto simp add: epred-llength)

lemma ldropn-iterates: ldropn n (iterates f x) = iterates f ((f ∘ n) x)
proof(induct n arbitrary: x)
  case 0 thus ?case by simp
  next
    case (Suc n)
    have ldropn (Suc n) (iterates f x) = ldropn n (iterates f (f x))
    by(subst iterates)simp
    also have ... = iterates f ((f ∘ n) (f x)) by(rule Suc)
    finally show ?case by(simp add: funpow-swap1)
  qed

lemma ldrop-iterates: ldrop (enat n) (iterates f x) = iterates f ((f ∘ n) x)
proof(induct n arbitrary: x)
  case Suc thus ?case

```

```

by(subst iterates)(simp add: eSuc-enat[symmetric] funpow-swap1)
qed(simp add: zero-enat-def[symmetric])

lemma lnth-iterates [simp]: lnth (iterates f x) n = (f ^~ n) x
proof(induct n arbitrary: x)
  case 0 thus ?case by(subst iterates) simp
next
  case (Suc n)
  hence lnth (iterates f (f x)) n = (f ^~ n) (f x) .
  thus ?case by(subst iterates)(simp add: funpow-swap1)
qed

lemma lset-iterates:
  lset (iterates f x) = {(f ^~ n) x | n. True}
by(auto simp add: lset-conv-lnth)

lemma lset-repeat [simp]: lset (repeat x) = {x}
by(simp add: lset-iterates id-def[symmetric])

```

2.16 More on the prefix ordering on lazy lists: (\sqsubseteq) and lstrict-prefix

```

lemma lstrict-prefix-code [code, simp]:
  lstrict-prefix LNil LNil  $\longleftrightarrow$  False
  lstrict-prefix LNil (LCons y ys)  $\longleftrightarrow$  True
  lstrict-prefix (LCons x xs) LNil  $\longleftrightarrow$  False
  lstrict-prefix (LCons x xs) (LCons y ys)  $\longleftrightarrow$  x = y  $\wedge$  lstrict-prefix xs ys
by(auto simp add: lstrict-prefix-def)

lemma lmap-lprefix: xs  $\sqsubseteq$  ys  $\Longrightarrow$  lmap f xs  $\sqsubseteq$  lmap f ys
by(rule monotoneD[OF monotone-lmap])

lemma lprefix-llength-eq-imp-eq:
   $\llbracket xs \sqsubseteq ys; \text{llength } xs = \text{llength } ys \rrbracket \Longrightarrow xs = ys$ 
by(coinduction arbitrary: xs ys)(auto simp add: not-lnull-conv)

lemma lprefix-llength-le: xs  $\sqsubseteq$  ys  $\Longrightarrow$  llength xs  $\leq$  llength ys
using monotone-llength by(rule monotoneD)

lemma lstrict-prefix-llength-less:
  assumes lstrict-prefix xs ys
  shows llength xs < llength ys
proof(rule ccontr)
  assume  $\neg$  llength xs < llength ys
  moreover from ‹lstrict-prefix xs ys› have xs  $\sqsubseteq$  ys xs  $\neq$  ys
    unfolding lstrict-prefix-def by simp-all
  from ‹xs  $\sqsubseteq$  ys› have llength xs  $\leq$  llength ys
    by(rule lprefix-llength-le)
  ultimately have llength xs = llength ys by auto
  with ‹xs  $\sqsubseteq$  ys› have xs = ys

```

```

  by(rule lprefix-llength-eq-imp-eq)
  with `xs ≠ ys` show False by contradiction
qed

lemma lstrict-prefix-lfinite1: lstrict-prefix xs ys ==> lfinite xs
by (metis lstrict-prefix-def not-lfinite-lprefix-conv-eq)

lemma wfP-lstrict-prefix: wfP lstrict-prefix
proof(unfold wfP-def)
  have wf {x :: enat. x < y}
  unfolding wf-def by(blast intro: less-induct)
  hence wf (inv-image {(x, y). x < y} llength) by(rule wf-inv-image)
  moreover have {(xs, ys). lstrict-prefix xs ys} ⊆ inv-image {(x, y). x < y} llength
    by(auto intro: lstrict-prefix-llength-less)
  ultimately show wf {(xs, ys). lstrict-prefix xs ys} by(rule wf-subset)
qed

lemma llist-less-induct[case-names less]:
  (Λxs. (Λys. lstrict-prefix ys xs ==> P ys) ==> P xs) ==> P xs
by(rule wfP-induct[OF wfP-lstrict-prefix]) blast

lemma ltake-enat-eq-imp-eq: (Λn. ltake (enat n) xs = ltake (enat n) ys) ==> xs =
ys
by(coinduction arbitrary: xs ys)(auto simp add: zero-enat-def lnull-def neq-LNil-conv
ltake-eq-LNil-iff eSuc-enat[symmetric] elim: allE[where x=Suc n for n])

lemma ltake-enat-lprefix-imp-lprefix:
  assumes Λn. lprefix (ltake (enat n) xs) (ltake (enat n) ys)
  shows lprefix xs ys
proof -
  have ccpo.admissible Sup (≤) (λn. ltake n xs ⊑ ltake n ys) by simp
  hence ltake (Sup (range enat)) xs ⊑ ltake (Sup (range enat)) ys
    by(rule ccpo.admissibleD)(auto intro: assms)
  thus ?thesis by(simp add: ltake-all)
qed

lemma lprefix-conv-lappend: xs ⊑ ys ↔ (∃zs. ys = lappend xs zs) (is ?lhs ↔
?rhs)
proof(rule iffI)
  assume ?lhs
  hence ys = lappend xs (ldrop (llength xs) ys)
    by(coinduction arbitrary: xs ys)(auto dest: lprefix-lnullD lprefix-lhdD intro:
lprefix-ltlI simp add: not-lnull-conv lprefix-LCons-conv intro: exI[where x=LNil])
  thus ?rhs ..
next
  assume ?rhs
  thus ?lhs by(coinduct rule: lprefix-coinduct) auto
qed

```

```

lemma lappend-lprefixE:
  assumes lappend xs ys ⊑ zs
  obtains zs' where zs = lappend xs zs'
  using assms unfolding lprefix-conv-lappend by(auto simp add: lappend-assoc)

lemma lprefix-lfiniteD:
   $\llbracket xs \sqsubseteq ys; lfinite ys \rrbracket \implies lfinite xs$ 
  unfolding lprefix-conv-lappend by auto

lemma lprefix-lappendD:
  assumes xs ⊑ lappend ys zs
  shows xs ⊑ ys ∨ ys ⊑ xs
  proof(rule ccontr)
    assume  $\neg (xs \sqsubseteq ys \vee ys \sqsubseteq xs)$ 
    hence  $\neg xs \sqsubseteq ys \neg ys \sqsubseteq xs$  by simp-all
    from  $\langle xs \sqsubseteq lappend ys zs \rangle$  obtain xs'
      where lappend xs xs' = lappend ys zs
      unfolding lprefix-conv-lappend by auto
    hence eq: lappend (ltake (llength ys) xs) (lappend (ldrop (llength ys) xs) xs') =
      lappend ys zs
      unfolding lappend-assoc[symmetric] by(simp only: lappend-ltake-ldrop)
    moreover have llength xs ≥ llength ys
    proof(rule ccontr)
      assume  $\neg ?thesis$ 
      hence llength xs < llength ys by simp
      hence ltake (llength ys) xs = xs by(simp add: ltake-all)
      hence lappend xs (lappend (ldrop (llength ys) xs) xs') =
        lappend (ltake (llength xs) ys) (lappend (ldrop (llength xs) ys) zs)
        unfolding lappend-assoc[symmetric] lappend-ltake-ldrop
        using eq by(simp add: lappend-assoc)
      hence xs: xs = ltake (llength xs) ys using  $\langle llength xs < llength ys \rangle$ 
        by(subst (asm) lappend-eq-lappend-conv)(auto simp add: min-def)
      have xs ⊑ ys by(subst xs) auto
      with  $\neg ys \sqsubseteq xs$  show False by contradiction
    qed
    ultimately have ys: ys = ltake (llength ys) xs
    by(subst (asm) lappend-eq-lappend-conv)(simp-all add: min-def)
    have ys ⊑ xs by(subst ys) auto
    with  $\neg ys \sqsubseteq xs$  show False by contradiction
  qed

lemma lstrict-prefix-lappend-conv:
  lstrict-prefix xs (lappend xs ys)  $\longleftrightarrow$  lfinite xs ∧  $\neg lnull ys$ 
  proof -
    { assume lfinite xs xs = lappend xs ys
      hence lnull ys by induct auto }
    thus ?thesis
      by(auto simp add: lstrict-prefix-def lprefix-lappend lappend-inf lappend-lnull2
        elim: contrapos-np)

```

qed

lemma *lprefix-llist-ofI*:

$\exists zs. ys = xs @ zs \implies llist-of xs \sqsubseteq llist-of ys$
by(*clar simp simp add: lappend-llist-of-llist-of[symmetric] lprefix-lappend*)

lemma *lprefix-llist-of [simp]*: $llist-of xs \sqsubseteq llist-of ys \longleftrightarrow prefix xs ys$

by(*auto simp add: prefix-def lprefix-conv-lappend*)*(metis lfinite-lappend lfinite-llist-of list-of-lappend list-of-llist-of lappend-llist-of-llist-of)*+

lemma *llimit-induct [case-names LNil LCons limit]*:

— The limit case is just an instance of admissibility

assumes *LNil: P LNil*

and *LCons: $\bigwedge x xs. [\lfloor lfinite xs; P xs \rfloor \implies P (LCons x xs)]$*

and *limit: ($\bigwedge ys. lstrict-prefix ys xs \implies P ys$) $\implies P xs$*

shows *P xs*

proof(rule *limit*)

fix *ys*

assume *lstrict-prefix ys xs*

hence *lfinite ys* by(*rule lstrict-prefix-lfinite1*)

thus *P ys* by(*induct*)*(blast intro: LNil LCons)*+

qed

lemma *lmap-lstrict-prefix*:

lstrict-prefix xs ys $\implies lstrict-prefix (lmap f xs) (lmap f ys)$

by (*metis llength-lmap lmap-lprefix lprefix-llength-eq-imp-eq lstrict-prefix-def*)

lemma *lprefix-lnthD*:

assumes *xs $\sqsubseteq ys$ and enat n < llength xs*

shows *lnth xs n = lnth ys n*

using *assms* by (*metis lnth-lappend1 lprefix-conv-lappend*)

lemma *lfinite-lSup-chain*:

assumes *chain: Complete-Partial-Order.chain (\sqsubseteq) A*

shows *lfinite (lSup A) \longleftrightarrow finite A $\wedge (\forall xs \in A. lfinite xs)$ (is ?lhs \longleftrightarrow ?rhs)*

proof(intro iffI conjI)

assume ?lhs

then obtain n where *n: llength (lSup A) = enat n* unfolding *lfinite-conv-llength-enat*

..

have *llength 'A $\subseteq \{.. < enat (Suc n)\}$*

by(*auto dest!: chain-lprefix-lSup[OF chain] lprefix-llength-le simp add: n intro: le-less-trans*)

hence *finite (llength 'A)* by(*rule finite-subset*)*(simp add: finite-lessThan-enat-iff)*

moreover have *inj-on llength A* by(*rule inj-onI*)*(auto 4 3 dest: chainD[OF chain] lprefix-llength-eq-imp-eq)*

ultimately show *finite A* by(*rule finite-imageD*)

next

assume ?rhs

hence *finite A $\forall xs \in A. lfinite xs$* by *simp-all*

```

show ?lhs
proof(cases A = {})
  case False
  with chain <finite A>
  have lSup A ∈ A by(rule ccpo.in-chain-finite[OF llist ccpo])
  with <∀ xs∈A. lfinite xs> show ?thesis ..
qed simp
qed(rule lfinite-lSupD)

Setup for ( $\sqsubseteq$ ) for Nitpick

definition finite-lprefix :: 'a llist ⇒ 'a llist ⇒ bool
where finite-lprefix = ( $\sqsubseteq$ )

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)
by(simp-all add: lprefix-LCons-conv finite-lprefix-def lnull-def)

lemma lprefix-nitpick-simps [nitpick-simp]:
xs ⊑ ys = (if lfinite xs then finite-lprefix xs ys else xs = ys)
by(simp add: finite-lprefix-def not-lfinite-lprefix-conv-eq)

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)
by(simp-all add: llcp-def enat-unfold split: llist.split)

lemma llcp-eq-0-iff:
llcp xs ys = 0 ↔ lnull xs ∨ lnull ys ∨ lhd xs ≠ lhd ys
by(simp add: llcp-def)

lemma epred-llcp:
[| ¬ lnull xs; ¬ lnull ys; lhd xs = lhd ys |]
  ==> epred (llcp xs ys) = llcp (lhd xs) (lhd ys)
by(simp add: llcp-def)

lemma llcp-commute: llcp xs ys = llcp ys xs
by(coinduction arbitrary: xs ys rule: enat-coinduct)(auto simp add: llcp-eq-0-iff
epred-llcp)

```

```

lemma llcp-same-conv-length [simp]: llcp xs xs = llength xs
by(coinduction arbitrary: xs rule: enat-coinduct)(auto simp add: llcp-eq-0-iff epred-llcp
epred-llength)

lemma llcp-lappend-same [simp]:
  llcp (lappend xs ys) (lappend xs zs) = llength xs + llcp ys zs
by(coinduction arbitrary: xs rule: enat-coinduct)(auto simp add: iadd-is-0 llcp-eq-0-iff
epred-iadd1 epred-llcp epred-llength)

lemma llcp-lprefix1 [simp]: xs ⊑ ys ==> llcp xs ys = llength xs
by (metis add-0-right lappend-LNil2 llcp-LNil1 llcp-lappend-same lprefix-conv-lappend)

lemma llcp-lprefix2 [simp]: ys ⊑ xs ==> llcp xs ys = llength ys
by (metis llcp-commute llcp-lprefix1)

lemma llcp-le-length: llcp xs ys ≤ min (llength xs) (llength ys)
proof –
  define m n where m = llcp xs ys and n = min (llength xs) (llength ys)
  hence (m, n) ∈ {(llcp xs ys, min (llength xs) (llength ys)) | xs ys :: 'a llist. True}
  by blast
  thus m ≤ n
  proof(coinduct rule: enat-leI)
    case (Leenat m n)
    then obtain xs ys :: 'a llist where m = llcp xs ys n = min (llength xs) (llength
ys) by blast
    thus ?case
      by(cases xs ys rule: llist.exhaust[case-product llist.exhaust])(auto 4 3 intro!:
exI[where x=Suc 0] simp add: eSuc-enat[symmetric] iadd-Suc-right zero-enat-def[symmetric])
      qed
  qed

lemma llcp-ltake1: llcp (ltake n xs) ys = min n (llcp xs ys)
by(coinduction arbitrary: n xs ys rule: enat-coinduct)(auto simp add: llcp-eq-0-iff
enat-min-eq-0-iff epred-llcp ltl-ltake)

lemma llcp-ltake2: llcp xs (ltake n ys) = min n (llcp xs ys)
by(metis llcp-commute llcp-ltake1)

lemma llcp-ltake [simp]: llcp (ltake n xs) (ltake m ys) = min (min n m) (llcp xs
ys)
by(metis llcp-ltake1 llcp-ltake2 min.assoc)

```

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)
by(auto intro: llist.expand)

```

```

lemma lnull-lzip [simp]: lnull (lzip xs ys)  $\longleftrightarrow$  lnull xs  $\vee$  lnull ys
by(simp add: lzip-def)

lemma lzip-eq-LNil-conv: lzip xs ys = LNil  $\longleftrightarrow$  xs = LNil  $\vee$  ys = LNil
using lnull-lzip unfolding lnull-def .

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

lemma lzip-eq-LCons-conv:
lzip xs ys = LCons z zs  $\longleftrightarrow$ 
( $\exists x\, xs'\, y\, ys'.\, xs = LCons\, x\, xs' \wedge ys = LCons\, y\, ys' \wedge z = (x, y) \wedge zs = lzip\, xs'$ 
 $ys')$ 
by(cases xs ys rule: llist.exhaust[case-product llist.exhaust]) auto

lemma lzip-lappend:
llength xs = llength us
 $\implies$  lzip (lappend xs ys) (lappend us vs) = lappend (lzip xs us) (lzip ys vs)
by(coinduction arbitrary: xs ys us vs rule: llist.coinduct-strong)(auto 4 6 simp add:
llength-ltl)

lemma llength-lzip [simp]:
llength (lzip xs ys) = min (llength xs) (llength ys)
by(coinduction arbitrary: xs ys rule: enat-coinduct)(auto simp add: enat-min-eq-0-iff
epred.llength)

lemma ltake-lzip: ltake n (lzip xs ys) = lzip (ltake n xs) (ltake n ys)
by(coinduction arbitrary: xs ys n)(auto simp add: ltl-ltake)

lemma ldropn-lzip [simp]:
ldropn n (lzip xs ys) = lzip (ldropn n xs) (ldropn n ys)
by(induct n arbitrary: xs ys)(simp-all add: ldropn-Suc split: llist.split)

lemma
fixes F
defines F  $\equiv$   $\lambda lzip\, (xs,\ ys).\ case\ xs\ of\ LNil \Rightarrow LNil \mid LCons\, x\, xs' \Rightarrow case\ ys\ of\ LNil \Rightarrow LNil \mid LCons\, y\, ys' \Rightarrow LCons\, (x,\ y)$  (curry lzip xs' ys')
shows lzip-conv-fixp: lzip  $\equiv$  curry (ccpo.fixp (fun-lub lSup) (fun-ord ( $\sqsubseteq$ )) F) (is ?lhs  $\equiv$  ?rhs)
and lzip-mono: mono-llist ( $\lambda lzip.\ F\ lzip\ xs$ ) (is ?mono xs)
proof(intro eq-reflection ext)
show mono:  $\bigwedge xs.\ ?mono\ xs$  unfolding F-def by(tactic ‹Partial-Function.mono-tac
@{context} 1›)
fix xs ys
show lzip xs ys = ?rhs xs ys
proof(coinduction arbitrary: xs ys)
case Eq-llist show ?case
by(subst (1 3 4) llist.mono-body-fixp[OF mono])(auto simp add: F-def split):

```

```

llist.split)
qed
qed

lemma monotone-lzip: monotone (rel-prod (≤) (≤)) (≤) (case-prod lzip)
by(rule llist.fixp-preserves-mono2[OF lzip-mono lzip-conv-fixp]) simp

lemma mono2mono-lzip1 [THEN llist.mono2mono, cont-intro, simp]:
  shows monotone-lzip1: monotone (≤) (≤) (λxs. lzip xs ys)
by(simp add: monotone-lzip[simplified])

lemma mono2mono-lzip2 [THEN llist.mono2mono, cont-intro, simp]:
  shows monotone-lzip2: monotone (≤) (≤) (λys. lzip xs ys)
by(simp add: monotone-lzip[simplified])

lemma mcont-lzip: mcont (prod-lub lSup lSup) (rel-prod (≤) (≤)) lSup (≤) (case-prod
lzip)
by(rule llist.fixp-preserves-mcont2[OF lzip-mono lzip-conv-fixp]) simp

lemma mcont2mcont-lzip1 [THEN llist.mcont2mcont, cont-intro, simp]:
  shows mcont-lzip1: mcont lSup (≤) lSup (≤) (λxs. lzip xs ys)
by(simp add: mcont-lzip[simplified])

lemma mcont2mcont-lzip2 [THEN llist.mcont2mcont, cont-intro, simp]:
  shows mcont-lzip2: mcont lSup (≤) lSup (≤) (λys. lzip xs ys)
by(simp add: mcont-lzip[simplified])

lemma ldrop-lzip [simp]: ldrop n (lzip xs ys) = lzip (ldrop n xs) (ldrop n ys)
proof(induct xs arbitrary: ys n)
  case LCons
  thus ?case by(cases ys n rule: llist.exhaust[case-product co.enat.exhaust]) simp-all
qed simp-all

lemma lzip-iterates:
  lzip (iterates f x) (iterates g y) = iterates (λ(x, y). (f x, g y)) (x, y)
by(coinduction arbitrary: x y) auto

lemma lzip-llist-of [simp]:
  lzip (llist-of xs) (llist-of ys) = llist-of (zip xs ys)
proof(induct xs arbitrary: ys)
  case (Cons x xs')
  thus ?case by(cases ys) simp-all
qed simp

lemma lnth-lzip:
  [ enat n < llength xs; enat n < llength ys ]
  ==> lnth (lzip xs ys) n = (lnth xs n, lnth ys n)
proof(induct n arbitrary: xs ys)
  case 0 thus ?case

```

```

  by(simp add: zero-enat-def[symmetric] lnth-0-conv-lhd)
next
  case (Suc n)
  thus ?case
    by(cases xs ys rule: llist.exhaust[case-product llist.exhaust])(auto simp add:
eSuc-enat[symmetric])
qed

lemma lset-lzip:
lset (lzip xs ys) =
  {(lnth xs n, lnth ys n)|n. enat n < min (llength xs) (llength ys)}
by(auto simp add: lset-conv-lnth lnth-lzip)(auto intro!: exI simp add: lnth-lzip)

lemma lset-lzipD1: (x, y) ∈ lset (lzip xs ys) ==> x ∈ lset xs
proof(induct lzip xs ys arbitrary: xs ys rule: lset-induct)
  case [symmetric]: find
  thus ?case by(auto simp add: lzip-eq-LCons-conv)
next
  case (step z zs)
  thus ?case by -(drule sym, auto simp add: lzip-eq-LCons-conv)
qed

lemma lset-lzipD2: (x, y) ∈ lset (lzip xs ys) ==> y ∈ lset ys
proof(induct lzip xs ys arbitrary: xs ys rule: lset-induct)
  case [symmetric]: find
  thus ?case by(auto simp add: lzip-eq-LCons-conv)
next
  case (step z zs)
  thus ?case by -(drule sym, auto simp add: lzip-eq-LCons-conv)
qed

lemma lset-lzip-same [simp]: lset (lzip xs xs) = (λx. (x, x)) ` lset xs
by(auto 4 3 simp add: lset-lzip in-lset-conv-lnth)

lemma lfinitelzip [simp]:
lfinitelzip(xs, ys) ↔ lfinitexs ∨ lfinitelys (is ?lhs ↔ ?rhs)
proof
  assume ?lhs
  thus ?rhs
    by(induct zs≡lzip xs ys arbitrary: xs ys rule: lfinitelzip-induct) fastforce+
next
  assume ?rhs (is ?xs ∨ ?ys)
  thus ?lhs
  proof
    assume ?xs
    thus ?thesis
    proof(induct arbitrary: ys)
      case (lfinitelzipI xs x)
      thus ?case by(cases ys) simp-all
    qed
  qed
qed

```

```

qed simp
next
assume ?ys
thus ?thesis
proof(induct arbitrary: xs)
  case (lfinite-LConsI ys y)
  thus ?case by(cases xs) simp-all
qed simp
qed
qed

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 -
let ?xs' = ltake (llength us) xs
let ?xs'' = ldrop (llength us) xs
let ?ys' = ltake (llength us) ys
let ?ys'' = ldrop (llength us) ys

from eq have llength (lzip xs ys) = llength (lappend us vs) by simp
hence min (llength xs) (llength ys) ≥ llength us
  by(auto simp add: enat-le-plus-same)
hence len: llength xs ≥ llength us llength ys ≥ llength us by(auto)

hence leneq: llength ?xs' = llength ?ys' by(simp add: min-def)
have xs: xs = lappend ?xs' ?xs'' and ys: ys = lappend ?ys' ?ys''
  by(simp-all add: lappend-ltake-ldrop)
hence lappend us vs = lzip (lappend ?xs' ?xs'') (lappend ?ys' ?ys'')
  using eq by simp
with len have lappend us vs = lappend (lzip ?xs' ?ys') (lzip ?xs'' ?ys'')
  by(simp add: lzip-lappend min-def)
hence us: us = lzip ?xs' ?ys'
  and vs: lfinite us → vs = lzip ?xs'' ?ys'' using len
  by(simp-all add: min-def lappend-eq-lappend-conv)
show ?thesis
proof(cases lfinite us)
  case True
  with leneq xs ys us vs len show ?thesis by fastforce
next
case False
let ?xs'' = lmap fst vs
let ?ys'' = lmap snd vs
from False have lappend us vs = us by(simp add: lappend-inf)
moreover from False have llength us = ∞
  by(rule not-lfinite-llength)
moreover with len

```

```

have llength xs = ∞ llength ys = ∞ by auto
moreover with ⟨llength us = ∞⟩
have xs = ?xs' ys = ?ys' by(simp-all add: ltake-all)
from ⟨llength us = ∞⟩ len
have ¬ lfinite ?xs' ¬ lfinite ?ys'
by(auto simp del: llength-ltake lfinite-ltake
    simp add: ltake-all dest: lfinite-length-enat)
with ⟨xs = ?xs'⟩ ⟨ys = ?ys'⟩
have xs = lappend ?xs' ?xs'' ys = lappend ?ys' ?ys''
by(simp-all add: lappend-inf)
moreover have vs = lzip ?xs'' ?ys''
by(coinduction arbitrary: vs) auto
ultimately show ?thesis using eq by(fastforce simp add: ltake-all)
qed
qed

lemma lzip-lmap [simp]:
lzip (lmap f xs) (lmap g ys) = lmap (λ(x, y). (f x, g y)) (lzip xs ys)
by(coinduction arbitrary: xs ys) auto

lemma lzip-lmap1:
lzip (lmap f xs) ys = lmap (λ(x, y). (f x, y)) (lzip xs ys)
by(subst (4) llist.map-ident[symmetric])(simp only: lzip-lmap)

lemma lzip-lmap2:
lzip xs (lmap f ys) = lmap (λ(x, y). (x, f y)) (lzip xs ys)
by(subst (1) llist.map-ident[symmetric])(simp only: lzip-lmap)

lemma lmap-fst-lzip-conv-ltake:
lmap fst (lzip xs ys) = ltake (min (llength xs) (llength ys)) xs
by(coinduction arbitrary: xs ys)(auto simp add: enat-min-eq-0-iff ltl-ltake epred-llength)

lemma lmap-snd-lzip-conv-ltake:
lmap snd (lzip xs ys) = ltake (min (llength xs) (llength ys)) ys
by(coinduction arbitrary: xs ys)(auto simp add: enat-min-eq-0-iff ltl-ltake epred-llength)

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)
by(coinduction arbitrary: xs ys)(auto simp add: enat-min-eq-0-iff ltl-ltake epred-llength)

```

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)

```

```

by(auto simp add: ltakeWhile-def intro: llist.expand)

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)
by(simp-all add: ldropWhile.simps)

lemma fixes f F P
  defines F ≡ λltakeWhile 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 (λltakeWhile. F ltakeWhile xs) (is PROP
  ?mono)
  proof(intro eq-reflection ext)
    show mono: PROP ?mono unfolding F-def by(tactic ‹Partial-Function.mono-tac
    @{context} 1›)
    fix xs
    show ?lhs xs = ?rhs xs
    proof(coinduction arbitrary: xs)
      case Eq-llist
      show ?case by(subst (1 3 4) llist.mono-body-fixp[OF mono])(auto simp add:
      F-def split: llist.split prod.split co.enat.split)
      qed
    qed
  qed

lemma mono2mono-ltakeWhile[THEN llist.mono2mono, cont-intro, simp]:
  shows monotone-ltakeWhile: monotone lprefix lprefix (ltakeWhile P)
by(rule llist.fixp-preserves-mono1[OF ltakeWhile-mono ltakeWhile-conv-fixp]) simp

lemma mcont2mcont-ltakeWhile [THEN llist.mcont2mcont, cont-intro, simp]:
  shows mcont-ltakeWhile: mcont lSup lprefix lSup lprefix (ltakeWhile P)
by(rule llist.fixp-preserves-mcont1[OF ltakeWhile-mono ltakeWhile-conv-fixp]) simp

lemma mono-llist-ltakeWhile [partial-function-mono]:
  mono-llist F ⟹ mono-llist (λf. ltakeWhile P (F f))
by(rule mono2mono-ltakeWhile)

lemma mono2mono-ldropWhile [THEN llist.mono2mono, cont-intro, simp]:
  shows monotone-ldropWhile: monotone (⊓) (⊓) (ldropWhile P)
by(rule llist.fixp-preserves-mono1[OF ldropWhile.mono ldropWhile-def]) simp

lemma mcont2mcont-ldropWhile [THEN llist.mcont2mcont, cont-intro, simp]:
  shows mcont-ldropWhile: mcont lSup (⊓) lSup (⊓) (ldropWhile P)
by(rule llist.fixp-preserves-mcont1[OF ldropWhile.mono ldropWhile-def]) simp

lemma lnull-ltakeWhile [simp]: lnull (ltakeWhile P xs) ⟷ (¬ lnull xs → ¬ P
(lhd xs))

```

```

by(cases xs) simp-all

lemma ltakeWhile-eq-LNil-iff: ltakeWhile P xs = LNil  $\longleftrightarrow$  (xs  $\neq$  LNil  $\longrightarrow$   $\neg$  P (lhd xs))
using lnull-ltakeWhile unfolding lnull-def .

lemmas lhd-ltakeWhile = ltakeWhile.sel(1)

lemma ltl-ltakeWhile:
 $ltl(ltakeWhile P xs) = (if P (lhd xs) then ltakeWhile P (ltl xs) else LNil)$ 
by(cases xs) simp-all

lemma lprefix-ltakeWhile: ltakeWhile P xs  $\sqsubseteq$  xs
by(coinduction arbitrary: xs)(auto simp add: ltl-ltakeWhile)

lemma llength-ltakeWhile-le: llength (ltakeWhile P xs)  $\leq$  llength xs
by(rule lprefix-llength-le)(rule lprefix-ltakeWhile)

lemma ltakeWhile-nth: enat i  $<$  llength (ltakeWhile P xs)  $\Longrightarrow$  lnth (ltakeWhile P xs) i = lnth xs i
by(rule lprefix-lnthD[OF lprefix-ltakeWhile])

lemma ltakeWhile-all:  $\forall x \in lset xs. P x \Longrightarrow ltakeWhile P xs = xs$ 
by(coinduction arbitrary: xs)(auto 4 3 simp add: ltl-ltakeWhile simp del: ltakeWhile.disc-iff dest: in-lset-ltlD)

lemma lset-ltakeWhileD:
assumes x  $\in$  lset (ltakeWhile P xs)
shows x  $\in$  lset xs  $\wedge$  P x
using assms
by(induct ys $\equiv$ ltakeWhile P xs arbitrary: xs rule: llist-set-induct)(auto simp add: ltl-ltakeWhile dest: in-lset-ltlD)

lemma lset-ltakeWhile-subset:
lset (ltakeWhile P xs)  $\subseteq$  lset xs  $\cap$  {x. P x}
by(auto dest: lset-ltakeWhileD)

lemma ltakeWhile-all-conv: ltakeWhile P xs = xs  $\longleftrightarrow$  lset xs  $\subseteq$  {x. P x}
by (metis Int-Collect Int-absorb2 le-infE lset-ltakeWhile-subset ltakeWhile-all)

lemma llength-ltakeWhile-all: llength (ltakeWhile P xs) = llength xs  $\longleftrightarrow$  ltakeWhile P xs = xs
by(auto intro: lprefix-llength-eq-imp-eq lprefix-ltakeWhile)

lemma ldropWhile-eq-LNil-iff: ldropWhile P xs = LNil  $\longleftrightarrow$  ( $\forall x \in lset xs. P x$ )
by(induct xs) simp-all

lemma lnull-ldropWhile [simp]:
lnull (ldropWhile P xs)  $\longleftrightarrow$  ( $\forall x \in lset xs. P x$ ) (is ?lhs  $\longleftrightarrow$  ?rhs)

```

```

unfolding lnull-def by(simp add: ldropWhile-eq-LNil-iff)

lemma lset-ldropWhile-subset:
  lset (ldropWhile P xs) ⊆ lset xs
by(induct xs) auto

lemma in-lset-ldropWhileD:  $x \in \text{lset}(\text{ldropWhile } P \text{ } xs) \implies x \in \text{lset } xs$ 
using lset-ldropWhile-subset[of P xs] by auto

lemma ltakeWhile-lmap: ltakeWhile P (lmap f xs) = lmap f (ltakeWhile (P ∘ f) xs)
by(coinduction arbitrary: xs)(auto simp add: ltl-ltakeWhile)

lemma ldropWhile-lmap: ldropWhile P (lmap f xs) = lmap f (ldropWhile (P ∘ f) xs)
by(induct xs) simp-all

lemma llength-ltakeWhile-lt-iff: llength (ltakeWhile P xs) < llength xs  $\longleftrightarrow$  ( $\exists x \in \text{lset } xs. \neg P x$ )
  (is ?lhs  $\longleftrightarrow$  ?rhs)
proof
  assume ?lhs
  hence ltakeWhile P xs ≠ xs by auto
  thus ?rhs by(auto simp add: ltakeWhile-all-conv)
next
  assume ?rhs
  hence ltakeWhile P xs ≠ xs by(auto simp add: ltakeWhile-all-conv)
  thus ?lhs unfolding llength-ltakeWhile-all[symmetric]
    using llength-ltakeWhile-le[of P xs] by(auto)
qed

lemma ltakeWhile-K-False [simp]: ltakeWhile (λ-. False) xs = LNil
by(simp add: ltakeWhile-def)

lemma ltakeWhile-K-True [simp]: ltakeWhile (λ-. True) xs = xs
by(coinduction arbitrary: xs) simp

lemma ldropWhile-K-False [simp]: ldropWhile (λ-. False) = id
proof
  fix xs
  show ldropWhile (λ-. False) xs = id xs
    by(induct xs) simp-all
qed

lemma ldropWhile-K-True [simp]: ldropWhile (λ-. True) xs = LNil
by(induct xs)(simp-all)

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

```

```

by(coinduction arbitrary: xs rule: llist.coinduct-strong)(auto 4 4 simp add: not-lnull-conv
lset-lnull intro: ccontr)

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(coinduction arbitrary: xs rule: llist.coinduct-strong)
case (Eq-llist xs)
have ?lnull by(auto simp add: lset-lnull)
moreover have ?LCons
by(clarsimp split: if-split-asm split del: if-split simp add: ltl-ltakeWhile)(auto 4
3 simp add: not-lnull-conv)
ultimately show ?case ..
qed

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(cases ∃x∈lset xs. ¬ P x)
case True
then obtain x where x ∈ lset xs ¬ P x by blast
thus ?thesis by induct simp-all
next
case False
note xs = this
show ?thesis
proof(cases lfinite xs)
case False
thus ?thesis using xs by(simp add: lappend-inf)
next
case True
thus ?thesis using xs by induct simp-all
qed
qed

lemma lfinite-ltakeWhile:
lfinite (ltakeWhile P xs) ↔ lfinite xs ∨ (∃x ∈ lset xs. ¬ P x) (is ?lhs ↔ ?rhs)
proof
assume ?lhs
thus ?rhs by(auto simp add: ltakeWhile-all)
next
assume ?rhs
thus ?lhs
proof
assume lfinite xs
with lprefix-ltakeWhile show ?thesis by(rule lprefix-lfiniteD)
next

```

```

assume  $\exists x \in lset xs. \neg P x$ 
then obtain x where  $x \in lset xs \neg P x$  by blast
thus ?thesis by(induct rule: lset-induct) simp-all
qed
qed

lemma llength-ltakeWhileEqinfinity:
 $llength(ltakeWhile P xs) = \infty \longleftrightarrow \neg lfinite xs \wedge ltakeWhile P xs = xs$ 
unfolding llength-ltakeWhile-all[symmetric] llength-eq-infty-conv-lfinite[symmetric]
by(auto)(auto simp add: llength-eq-infty-conv-lfinite lfinite-ltakeWhile intro: sym)

lemma llength-ltakeWhileEqinfinity':
 $llength(ltakeWhile P xs) = \infty \longleftrightarrow \neg lfinite xs \wedge (\forall x \in lset xs. P x)$ 
by (metis lfinite-ltakeWhile llength-eq-infty-conv-lfinite)

lemma lzip-ltakeWhile-fst:  $lzip(ltakeWhile P xs) ys = ltakeWhile(P \circ fst)(lzip xs ys)$ 
by(coinduction arbitrary: xs ys)(auto simp add: ltl-ltakeWhile simp del: simp del: ltakeWhile.disc-iff)

lemma lzip-ltakeWhile-snd:  $lzip xs (ltakeWhile P ys) = ltakeWhile(P \circ snd)(lzip xs ys)$ 
by(coinduction arbitrary: xs ys)(auto simp add: ltl-ltakeWhile)

lemma ltakeWhile-lappend1:
 $\llbracket x \in lset xs; \neg P x \rrbracket \implies ltakeWhile P (lappend xs ys) = ltakeWhile P xs$ 
by(induct rule: lset-induct) simp-all

lemma ltakeWhile-lappend2:
 $lset xs \subseteq \{x. P x\} \implies ltakeWhile P (lappend xs ys) = lappend xs (ltakeWhile P ys)$ 
by(coinduction arbitrary: xs ys rule: llist.coinduct-strong)(auto 4 4 simp add: not-lnull-conv lappend-lnull1)

lemma ltakeWhile-cong [cong, fundef-cong]:
assumes xs:  $xs = ys$ 
and PQ:  $\bigwedge x. x \in lset ys \implies P x = Q x$ 
shows  $ltakeWhile P xs = ltakeWhile Q ys$ 
using PQ unfolding xs
by(coinduction arbitrary: ys)(auto simp add: ltl-ltakeWhile not-lnull-conv)

lemma lnth-llength-ltakeWhile:
assumes len:  $llength(ltakeWhile P xs) < llength xs$ 
shows  $\neg P(lnth xs (\text{the-enat}(llength(ltakeWhile P xs))))$ 
proof
assume P:  $P(lnth xs (\text{the-enat}(llength(ltakeWhile P xs))))$ 
from len obtain len where  $llength(ltakeWhile P xs) = \text{enat } len$ 
by(cases llength(ltakeWhile P xs)) auto
with P len show False apply simp

```

```

proof(induct len arbitrary: xs)
  case 0 thus ?case by(simp add: zero-enat-def[symmetric] lnth-0-conv-lhd)
next
  case (Suc len) thus ?case by(cases xs)(auto split: if-split-asm simp add:
eSuc-enat[symmetric])
  qed
qed

lemma assumes  $\exists x \in lset xs. \neg P x$ 
shows lhd-ldropWhile:  $\neg P (lhd (ldropWhile P xs))$  (is ?thesis1)
and lhd-ldropWhile-in-lset:  $lhd (ldropWhile P xs) \in lset xs$  (is ?thesis2)
proof -
  from assms obtain x where  $x \in lset xs \neg P x$  by blast
  thus ?thesis1 ?thesis2 by induct simp-all
  qed

lemma ldropWhile-eq-ldrop:
  ldropWhile P xs = ldrop (llength (ltakeWhile P xs)) xs
  (is ?lhs = ?rhs)
proof(rule lprefix-antisym)
  show ?lhs  $\sqsubseteq$  ?rhs
    by(induct arbitrary: xs rule: ldropWhile.fixp-induct)(auto split: llist.split)
  show ?rhs  $\sqsubseteq$  ?lhs
    proof(induct arbitrary: xs rule: ldrop.fixp-induct)
      case (3 ldrop xs)
        thus ?case by(cases xs) auto
      qed simp-all
  qed

lemma ldropWhile-cong [cong]:
   $\llbracket xs = ys; \bigwedge x. x \in lset ys \implies P x = Q x \rrbracket \implies ldropWhile P xs = ldropWhile Q ys$ 
by(simp add: ldropWhile-eq-ldrop)

lemma ltakeWhile-repeat:
  ltakeWhile P (repeat x) = (if P x then repeat x else LNil)
by(coinduction arbitrary: x)(auto simp add: ltl-ltakeWhile)

lemma ldropWhile-repeat: ldropWhile P (repeat x) = (if P x then LNil else repeat x)
by(simp add: ldropWhile-eq-ldrop ltakeWhile-repeat)

lemma lfinite-ldropWhile: lfinite (ldropWhile P xs)  $\longleftrightarrow (\exists x \in lset xs. \neg P x) \longrightarrow$ 
lfinite xs
by(auto simp add: ldropWhile-eq-ldrop llength-eq-infnty-conv-lfinite lfinite-ltakeWhile)

lemma llength-ldropWhile:
  llength (ldropWhile P xs) =
  (if  $\exists x \in lset xs. \neg P x$  then llength xs - llength (ltakeWhile P xs) else 0)

```

```

by(auto simp add: ldropWhile-eq-ldrop llength-ldrop llength-ltakeWhile-all ltakeWhile-all-conv llength-ltakeWhile-eq-infinity zero-enat-def dest!: lfinite-llength-enat)

lemma lhd-ldropWhile-conv-lnth:
   $\exists x \in lset xs. \neg P x \implies lhd(ldropWhile P xs) = lnth xs (\text{the-enat}(llength(ltakeWhile P xs)))$ 
by(simp add: ldropWhile-eq-ldrop lhd-ldrop llength-ltakeWhile-lt-iff)

```

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
by(cases xs) simp-all

lemma llist-all2-LNil2 [simp]: llist-all2 P xs LNil  $\longleftrightarrow$  xs = LNil
by(cases xs) simp-all

lemma llist-all2-LCons1:
  llist-all2 P (LCons x xs) ys  $\longleftrightarrow$  ( $\exists y ys'. ys = LCons y ys' \wedge P x y \wedge llist-all2 P xs ys'$ )
by(cases ys) simp-all

lemma llist-all2-LCons2:
  llist-all2 P xs (LCons y ys)  $\longleftrightarrow$  ( $\exists x xs'. xs = LCons x xs' \wedge P x y \wedge llist-all2 P xs' ys$ )
by(cases xs) simp-all

lemma llist-all2-list-of [simp]:
  llist-all2 P (llist-of xs) (llist-of ys) = list-all2 P xs ys
by(induct xs ys rule: list-induct2')(simp-all)

lemma llist-all2-conv-lzip:
  llist-all2 P xs ys  $\longleftrightarrow$  llength xs = llength ys  $\wedge$  ( $\forall (x, y) \in lset(lzip xs ys). P x y$ )
by(auto 4 4 elim!: GrpE simp add:
  llist-all2-def lmap-fst-lzip-conv-ltake lmap-snd-lzip-conv-ltake ltake-all
  intro!: GrpI relcomppI[of - xs - - ys])

lemma llist-all2-llengthD:
  llist-all2 P xs ys  $\implies$  llength xs = llength ys
by(simp add: llist-all2-conv-lzip)

lemma llist-all2-lnullD: llist-all2 P xs ys  $\implies$  lnull xs  $\longleftrightarrow$  lnull ys
unfolding lnull-def by auto

lemma llist-all2-all-lnthI:

```

```

 $\llist{length} \ xs = \llist{length} \ ys;$ 
 $\wedge n. \ enat{n} < \llist{length} \ xs \implies P(\lnth{xs}{n})(\lnth{ys}{n})$ 
 $\implies \llist{all2} \ P \ xs \ ys$ 
by(auto simp add: llist-all2-conv-lzip lset-lzip)

lemma llist-all2-lnthD:
 $\llist{all2} \ P \ xs \ ys; \ enat{n} < \llist{length} \ xs \implies P(\lnth{xs}{n})(\lnth{ys}{n})$ 
by(fastforce simp add: llist-all2-conv-lzip lset-lzip)

lemma llist-all2-lnthD2:
 $\llist{all2} \ P \ xs \ ys; \ enat{n} < \llist{length} \ ys \implies P(\lnth{xs}{n})(\lnth{ys}{n})$ 
by(fastforce simp add: llist-all2-conv-lzip lset-lzip)

lemma llist-all2-conv-all-lnth:
 $\llist{all2} \ P \ xs \ ys \longleftrightarrow$ 
 $\llist{length} \ xs = \llist{length} \ ys \wedge$ 
 $(\forall n. \ enat{n} < \llist{length} \ ys \longrightarrow P(\lnth{xs}{n})(\lnth{ys}{n}))$ 
by(auto dest: llist-all2-llengthD llist-all2-lnthD2 intro: llist-all2-all-lnthI)

lemma llist-all2-True [simp]: llist-all2 ( $\lambda x. \ True$ ) xs ys  $\longleftrightarrow \llist{length} \ xs = \llist{length} \ ys$ 
by(simp add: llist-all2-conv-all-lnth)

lemma llist-all2-reflI:
 $(\bigwedge x. \ x \in \llist{xs} \implies P(x)) \implies \llist{all2} \ P \ xs \ xs$ 
by(auto simp add: llist-all2-conv-all-lnth lset-conv-lnth)

lemma llist-all2-lmap1:
 $\llist{all2} \ P \ (\llist{lmap} \ f \ xs) \ ys \longleftrightarrow \llist{all2} \ (\lambda x. \ P(fx)) \ xs \ ys$ 
by(auto simp add: llist-all2-conv-all-lnth)

lemma llist-all2-lmap2:
 $\llist{all2} \ P \ xs \ (\llist{lmap} \ g \ ys) \longleftrightarrow \llist{all2} \ (\lambda x y. \ P(x)(g y)) \ xs \ ys$ 
by(auto simp add: llist-all2-conv-all-lnth)

lemma llist-all2-lfiniteD:
 $\llist{all2} \ P \ xs \ ys \implies \llist{lfinite} \ xs \longleftrightarrow \llist{lfinite} \ ys$ 
by(drule llist-all2-llengthD)(simp add: lfinite-conv-llength-enat)

lemma llist-all2-coinduct[consumes 1, case-names LNil LCons, case-conclusion LCons lhd ltl, coinduct pred]:
assumes major:  $X \ xs \ ys$ 
and step:
 $\bigwedge \{xs, ys. \ X \ xs \ ys \implies \llist{lnull} \ xs \longleftrightarrow \llist{lnull} \ ys$ 
 $\bigwedge \{xs, ys. \ [\ X \ xs \ ys; \neg \llist{lnull} \ xs; \neg \llist{lnull} \ ys \ ] \implies P(\llist{lhd} \ xs)(\llist{lhd} \ ys) \wedge (X(\llist{ltl} \ xs)(\llist{ltl} \ ys) \vee \llist{all2} \ P(\llist{ltl} \ xs)(\llist{ltl} \ ys))$ 
shows llist-all2 P xs ys
proof(rule llist-all2-all-lnthI)
from major show llength xs = llength ys

```

```

by(coinduction arbitrary: xs ys rule: enat-coinduct)(auto 4 3 dest: step llist-all2-llengthD
simp add: epred-llength)

fix n
assume enat n < llength xs
thus P (lnth xs n) (lnth ys n)
  using major ‹llength xs = llength ys›
proof(induct n arbitrary: xs ys)
  case 0 thus ?case
    by(cases lnull xs)(auto dest: step simp add: zero-enat-def[symmetric] lnth-0-conv-lhd)
  next
    case (Suc n)
      from step[OF ‹X xs ys›] ‹enat (Suc n) < llength xs› Suc show ?case
        by(auto 4 3 simp add: not-lnull-conv Suc-ile-eq intro: Suc.hyps llist-all2-lnthD
dest: llist-all2-llengthD)
      qed
    qed

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'
  using assms
  by(cases xs)(auto simp add: llist-all2-LCons1)

lemma llist-all2-mono:
  ‹llist-all2 P xs ys; ⋀x y. P x y ⟹ P' x y› ⟹ llist-all2 P' xs ys
  by(auto simp add: llist-all2-conv-all-lnth)

lemma llist-all2-left: llist-all2 (λx -. P x) xs ys ↔ llength xs = llength ys ∧
(∀x∈lset xs. P x)
  by(fastforce simp add: llist-all2-conv-all-lnth lset-conv-lnth)

lemma llist-all2-right: llist-all2 (λ-. P) xs ys ↔ llength xs = llength ys ∧
(∀x∈lset ys. P x)
  by(fastforce simp add: llist-all2-conv-all-lnth lset-conv-lnth)

lemma llist-all2-lsetD1: ‹llist-all2 P xs ys; x ∈ lset xs› ⟹ ∃y∈lset ys. P x y
  by(auto 4 4 simp add: llist-all2-conv-lzip lset-lzip lset-conv-lnth split-beta lnth-lzip
simp del: split-paired-All)

lemma llist-all2-lsetD2: ‹llist-all2 P xs ys; y ∈ lset ys› ⟹ ∃x∈lset xs. P x y
  by(auto 4 4 simp add: llist-all2-conv-lzip lset-lzip lset-conv-lnth split-beta lnth-lzip
simp del: split-paired-All)

lemma llist-all2-conj:
  llist-all2 (λx y. P x y ∧ Q x y) xs ys ↔ llist-all2 P xs ys ∧ llist-all2 Q xs ys

```

```

by(auto simp add: llist-all2-conv-all-lnth)

lemma llist-all2-lhdD:
  [l] llist-all2 P xs ys; ¬ lnull xs [/] ==> P (lhd xs) (lhd ys)
by(auto simp add: not-lnull-conv llist-all2-LCons1)

lemma llist-all2-lhdD2:
  [l] llist-all2 P xs ys; ¬ lnull ys [/] ==> P (lhd xs) (lhd ys)
by(auto simp add: not-lnull-conv llist-all2-LCons2)

lemma llist-all2-ltlI:
  llist-all2 P xs ys ==> llist-all2 P (ltl xs) (ltl ys)
by(cases xs)(auto simp add: llist-all2-LCons1)

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(cases lfinite xs)
  case True
  with 1 have lfinite ys by(auto dest: llist-all2-lfiniteD)
  from 1 2[OF True this] show ?thesis
    by(coinduction arbitrary: xs ys)(auto dest: llist-all2-lnullD llist-all2-lhdD intro:
      llist-all2-ltlI simp add: lappend-eq-LNil-iff)
  next
  case False
  with 1 have ¬ lfinite ys by(auto dest: llist-all2-lfiniteD)
  with False 1 show ?thesis by(simp add: lappend-inf)
qed

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 -
  from assms have len: llengt(xs + llengt(xs') = llengt(ys) by(auto dest: llist-all2-llengthD)
  hence len-xs: llengt(xs) ≤ llengt(ys) and len-xs': llengt(xs') ≤ llengt(ys)
    by (metis enat-le-plus-same llengt-lappend)+

  show llist-all2 P xs (ltake (llength xs) ys)
  proof(rule llist-all2-all-lnthI)
    show llengt(xs) = llengt(ltake (llength xs) ys)
      using len-xs by(simp add: min-def)
  next
    fix n
    assume n: enat n < llengt(xs)
    also have ... ≤ llengt(lappend(xs xs') n) by(simp add: enat-le-plus-same)
    finally have P (lnth(lappend(xs xs') n) (lnth ys n))
      using assms by -(rule llist-all2-lnthD)

```

```

also from n have lnth ys n = lnth (ltake (llength xs) ys) n by(rule lnth-ltake[symmetric])
also from n have lnth (lappend xs xs') n = lnth xs n by(simp add: lnth-lappend1)
  finally show P (lnth xs n) (lnth (ltake (llength xs) ys) n) .
qed

assume lfinite xs
thus llist-all2 P xs' (ldrop (llength xs) ys) using assms
  by(induct arbitrary: ys)(auto simp add: llist-all2-LCons1)
qed

lemma lmap-eq-lmap-conv-llist-all2:
  lmap f xs = lmap g ys  $\longleftrightarrow$  llist-all2 ( $\lambda x y. f x = g y$ ) xs ys (is ?lhs  $\longleftrightarrow$  ?rhs)
proof
  assume ?lhs
  thus ?rhs
    by(coinduction arbitrary: xs ys)(auto simp add: neq-LNil-conv lnull-def LNil-eq-lmap
lmap-eq-LNil)
  next
    assume ?rhs
    thus ?lhs
      by(coinduction arbitrary: xs ys)(auto dest: llist-all2-lnullD llist-all2-lhdD llist-all2-ltlI)
  qed

lemma llist-all2-expand:
   $\llbracket \text{lnull } xs \longleftrightarrow \text{lnull } ys; \neg \text{lnull } xs; \neg \text{lnull } ys \rrbracket \implies P (\text{lhd } xs) (\text{lhd } ys) \wedge \text{llist-all2 } P (\text{ltl } xs) (\text{ltl } ys) \rrbracket$ 
 $\implies \text{llist-all2 } P \text{ xs ys}$ 
by(cases xs)(auto simp add: not-lnull-conv)

lemma llist-all2-llength-ltakeWhileD:
  assumes major: llist-all2 P xs ys
  and Q:  $\bigwedge x y. P x y \implies Q1 x \longleftrightarrow Q2 y$ 
  shows llength (ltakeWhile Q1 xs) = llength (ltakeWhile Q2 ys)
using major
by(coinduction arbitrary: xs ys rule: enat-coinduct)(auto 4 3 simp add: not-lnull-conv
llist-all2-LCons1 llist-all2-LCons2 dest!: Q)

lemma llist-all2-lzipI:
   $\llbracket \text{llist-all2 } P \text{ xs ys}; \text{llist-all2 } P' \text{ xs' ys'} \rrbracket$ 
 $\implies \text{llist-all2 } (\text{rel-prod } P P') (\text{lzip } xs \text{ xs'}) (\text{lzip } ys \text{ ys'})$ 
by(coinduction arbitrary: xs xs' ys ys')(auto 6 6 dest: llist-all2-lhdD llist-all2-lnullD
intro: llist-all2-ltlI)

lemma llist-all2-ltakeI:
  llist-all2 P xs ys  $\implies$  llist-all2 P (ltake n xs) (ltake n ys)
by(auto simp add: llist-all2-conv-all-lnth lnth-ltake)

lemma llist-all2-ldropnI:
  llist-all2 P xs ys  $\implies$  llist-all2 P (ldropn n xs) (ldropn n ys)

```

```

by(cases llength ys)(auto simp add: llist-all2-conv-all-lnth)

lemma llist-all2-ldropI:
  llist-all2 P xs ys ==> llist-all2 P (ldrop n xs) (ldrop n ys)
by(cases llength ys)(auto simp add: llist-all2-conv-all-lnth llength-ldrop)

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))
  using assms
  proof(coinduction arbitrary: Y)
    case LNil
    thus ?case
      by(auto dest: llist-all2-lnullD simp add: split-beta)
  next
    case (LCons Y)
    note chain = `Complete-Partial-Order.chain - Y`
    from LCons have Y: ⋀xs ys. (xs, ys) ∈ Y ==> llist-all2 P xs ys by blast
    from LCons obtain xs ys where xs sys Y: (xs, ys) ∈ Y
      and [simp]: ¬ lnull xs ¬ lnull ys
      by(auto 4 3 dest: llist-all2-lnullD simp add: split-beta)
      from xs sys Y have lhd xs ∈ lhd ‘ (fst ‘ Y ∩ {xs. ¬ lnull xs})
        by(auto intro: rev-image-eqI)
        hence (THE x. x ∈ lhd ‘ (fst ‘ Y ∩ {xs. ¬ lnull xs})) = lhd xs
          by(rule the-equality)(auto dest!: lprefix-lhdD chainD[OF chain xs sys Y])
        moreover from xs sys Y have lhd ys ∈ lhd ‘ (snd ‘ Y ∩ {xs. ¬ lnull xs})
          by(auto intro: rev-image-eqI)
        hence (THE x. x ∈ lhd ‘ (snd ‘ Y ∩ {xs. ¬ lnull xs})) = lhd ys
          by(rule the-equality)(auto dest!: lprefix-lhdD chainD[OF chain xs sys Y])
        moreover from xs sys Y have llist-all2 P xs ys by(rule Y)
        hence P (lhd xs) (lhd ys) by(rule llist-all2-lhdD) simp
        ultimately have ?lhd using LCons by simp
      moreover {
        let ?Y = map-prod ltl ltl ‘ (Y ∩ {(xs, ys). ¬ lnull xs ∧ ¬ lnull ys})
        have Complete-Partial-Order.chain (rel-prod (⊓) (⊓)) ?Y
          by(rule chainI)(auto 4 3 dest: Y chainD[OF chain] intro: lprefix-ltlI)
        moreover
        have ltl ‘ (fst ‘ Y ∩ {xs. ¬ lnull xs}) = fst ‘ ?Y
          and ltl ‘ (snd ‘ Y ∩ {xs. ¬ lnull xs}) = snd ‘ ?Y
          by(fastforce simp add: image-image dest: Y llist-all2-lnullD intro: rev-image-eqI)+
        ultimately have ?ltl by(auto 4 3 intro: llist-all2-ltlI dest: Y) }
        ultimately show ?case ..
      qed
    lemma admissible-llist-all2 [cont-intro, simp]:
      assumes f: mcont lub ord lSup (⊓) (λx. f x)
      and g: mcont lub ord lSup (⊓) (λx. g x)
      shows ccpo.admissible lub ord (λx. llist-all2 P (f x) (g x))

```

```

proof(rule ccpo.admissibleI)
  fix Y
  assume chain: Complete-Partial-Order.chain ord Y
  and Y:  $\forall x \in Y. llist-all2 P (f x) (g x)$ 
  and Y ≠ {}
  from chain have Complete-Partial-Order.chain (rel-prod (⊑) (⊑)) (( $\lambda x. (f x, g x)$ ) ` Y)
    by(rule chain-imageI)(auto intro: mcont-monoD[OF f] mcont-monoD[OF g])
  from llist-all2-lSupI[OF this, of P] chain Y
  show llist-all2 P (f (lub Y)) (g (lub Y)) using `Y ≠ {}`
    by(simp add: mcont-contD[OF f chain] mcont-contD[OF g chain] image-image)
qed

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:  $\bigwedge x y. P x y \implies Q1 x \longleftrightarrow Q2 y$ 
  shows llist-all2 P (ldropWhile Q1 xs) (ldropWhile Q2 ys)
— cannot prove this with parallel induction over xs and ys because  $\lambda x. \neg llist-all2 P (f x) (g x)$  is not admissible.
  using * by(induction arbitrary: xs ys rule: ldropWhile-fixp-parallel-induct)(auto
  split: llist.split dest: Q)

lemma llist-all2-same [simp]: llist-all2 P xs xs  $\longleftrightarrow (\forall x \in lset xs. P x x)$ 
by(auto simp add: llist-all2-conv-all-lnth in-lset-conv-lnth Ball-def)

lemma llist-all2-trans:
   $\llist-all2 P xs ys; llist-all2 P ys zs; transp P \implies llist-all2 P xs zs$ 
  apply(rule llist-all2-all-lnthI)
  apply(simp add: llist-all2-llengthD)
  apply(frule llist-all2-llengthD)
  apply(drule (1) llist-all2-lnthD)
  apply(drule llist-all2-lnthD)
  apply simp
  apply(erule (2) transpD)
  done

```

2.21 The last element llast

lemma llast-LNil: llast LNil = undefined

```

by(simp add: llast-def zero-enat-def)

lemma llast-LCons: llast (LCons x xs) = (if lnull xs then x else llast xs)
by(cases llength xs)(auto simp add: llast-def eSuc-def zero-enat-def not-lnull-conv
split: enat.splits)

lemma llast-lfinite: ¬ lfinite xs ==> llast xs = undefined
by(simp add: llast-def lfinite-conv-llength-enat)

lemma [simp, code]:
  shows llast-singleton: llast (LCons x LNil) = x
  and llast-LCons2: llast (LCons x (LCons y xs)) = llast (LCons y xs)
by(simp-all add: llast-LCons)

lemma llast-lappend:
  llast (lappend xs ys) = (if lnull ys then llast xs else if lfinite xs then llast ys else
undefined)
proof(cases lfinite xs)
  case True
  hence ¬ lnull ys ==> llast (lappend xs ys) = llast ys
  by(induct rule: lfinite.induct)(simp-all add: llast-LCons)
  with True show ?thesis by(simp add: lappend-lnull2)
next
  case False thus ?thesis by(simp add: llast-lfinite)
qed

lemma llast-lappend-LCons [simp]:
  lfinite xs ==> llast (lappend xs (LCons y ys)) = llast (LCons y ys)
by(simp add: llast-lappend)

lemma llast-ldropn: enat n < llength xs ==> llast (ldropn n xs) = llast xs
proof(induct n arbitrary: xs)
  case 0 thus ?case by simp
next
  case (Suc n) thus ?case by(cases xs)(auto simp add: Suc-ile-eq llast-LCons)
qed

lemma llast-ldrop:
  assumes n < llength xs
  shows llast (ldrop n xs) = llast xs
proof -
  from assms obtain n' where n: n = enat n' by(cases n) auto
  show ?thesis using assms unfolding n
  proof(induct n' arbitrary: xs)
    case 0 thus ?case by(simp add: zero-enat-def[symmetric])
  next
    case Suc thus ?case by(cases xs)(auto simp add: eSuc-enat[symmetric] llast-LCons)
  qed
qed

```

```

lemma llast-llist-of [simp]: llast (llist-of xs) = last xs
by(induct xs)(auto simp add: last-def zero-enat-def llast-LCons llast-LNil)

lemma llast-conv-lnth: llength xs = eSuc (enat n)  $\Rightarrow$  llast xs = lnth xs n
by(clar simp simp add: llast-def zero-enat-def[symmetric] eSuc-enat split: nat.split)

lemma llast-lmap:
  assumes lfinite xs  $\neg$  lnull xs
  shows llast (lmap f xs) = f (llast xs)
  using assms
  proof induct
    case (lfinite-LConsI xs)
    thus ?case by(cases xs) simp-all
  qed simp

```

2.22 Distinct lazy lists ldistinct

```

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

```

```

lemma ldistinct-LNil-code [code]:
  ldistinct LNil = True
by simp

```

```

lemma ldistinct-llist-of [simp]:
  ldistinct (llist-of xs)  $\longleftrightarrow$  distinct xs
by(induct xs) auto

```

```

lemma ldistinct-coinduct [consumes 1, case-names ldistinct, case-conclusion ldistinct lhd ltl, coinduct pred: ldistinct]:
  assumes X xs
  and step:  $\bigwedge_{xs} \llbracket X \text{ xs}; \neg \text{lnull } xs \rrbracket$ 
   $\implies \text{lhd } xs \notin \text{lset } (\text{ltl } xs) \wedge (X \text{ (ltl } xs) \vee \text{ldistinct } (\text{ltl } xs))$ 
  shows ldistinct xs
  using <X xs>
  proof(coinduct)
    case (ldistinct xs)
    thus ?case by(cases xs)(auto dest: step)
  qed

```

```

lemma ldistinct-lhdD:
   $\llbracket \text{ldistinct } xs; \neg \text{lnull } xs \rrbracket \implies \text{lhd } xs \notin \text{lset } (\text{ltl } xs)$ 
by(clar simp simp add: not-lnull-conv)

```

```

lemma ldistinct-ltlI:
  ldistinct xs  $\implies$  ldistinct (ltl xs)
by(cases xs) simp-all

```

```

lemma ldistinct-lSup:
   $\llbracket \text{Complete-Partial-Order}.chain (\sqsubseteq) Y; \forall xs \in Y. \text{ldistinct } xs \rrbracket \implies \text{ldistinct } (\text{lSup } Y)$ 
proof(coinduction arbitrary: Y)
  case (ldistinct Y)
  hence chain: Complete-Partial-Order.chain ( $\sqsubseteq$ ) Y
  and distinct:  $\bigwedge xs. xs \in Y \implies \text{ldistinct } xs$  by blast+
  have ?lhd using chain by(auto 4 4 simp add: lset-lSup chain-lprefix-ltl dest: distinct lhd-lSup-eq ldistinct-lhdD)
  moreover have ?ltl by(auto 4 3 simp add: chain-lprefix-ltl chain intro: ldistinct-ltlI distinct)
  ultimately show ?case ..
qed

lemma admissible-ldistinct [cont-intro, simp]:
  assumes mcont: mcont lub ord lSup ( $\sqsubseteq$ ) ( $\lambda x. f x$ )
  shows ccpo.admissible lub ord ( $\lambda x. \text{ldistinct } (f x)$ )
proof(rule ccpo.admissibleI)
  fix Y
  assume chain: Complete-Partial-Order.chain ord Y
  and distinct:  $\forall x \in Y. \text{ldistinct } (f x)$ 
  and Y  $\neq \{\}$ 
  thus ldistinct (f (lub Y))
  by(simp add: mcont-contD[OF mcont] ldistinct-lSup chain-imageI mcont-monoD[OF mcont])
qed

lemma ldistinct-lappend:
  ldistinct (lappend xs ys)  $\longleftrightarrow$  ldistinct xs  $\wedge$  (lfinite xs  $\longrightarrow$  ldistinct ys  $\wedge$  lset xs  $\cap$  lset ys =  $\{\}$ )
  (is ?lhs = ?rhs)
proof(intro iffI conjI strip)
  assume ?lhs
  thus ldistinct xs
  by(coinduct)(auto simp add: not-lnull-conv in-lset-lappend-iff)

  assume lfinite xs
  thus ldistinct ys lset xs  $\cap$  lset ys =  $\{\}$ 
  using ‹?lhs› by induct simp-all
next
  assume ?rhs
  thus ?lhs
  by(coinduction arbitrary: xs)(auto simp add: not-lnull-conv in-lset-lappend-iff)
qed

lemma ldistinct-lprefix:
   $\llbracket \text{ldistinct } xs; ys \sqsubseteq xs \rrbracket \implies \text{ldistinct } ys$ 
by(clarify simp add: lprefix-conv-lappend ldistinct-lappend)

```

```

lemma admissible-not-ldistinct[THEN admissible-subst, cont-intro, simp]:
  ccpo.admissible lSup ( $\sqsubseteq$ ) ( $\lambda x. \neg ldistinct x$ )
by(rule ccpo.admissibleI)(auto dest: ldistinct-lprefix intro: chain-lprefix-lSup)

lemma ldistinct-ltake: ldistinct xs  $\implies$  ldistinct (ltake n xs)
by (metis ldistinct-lprefix ltake-is-lprefix)

lemma ldistinct-ldropn:
  ldistinct xs  $\implies$  ldistinct (ldropn n xs)
by(induct n arbitrary: xs)(simp, case-tac xs, simp-all)

lemma ldistinct-ldrop: ldistinct xs  $\implies$  ldistinct (ldrop n xs)
proof(induct xs arbitrary: n)
  case (LCons x xs) thus ?case
    by(cases n rule: co.enat.exhaust) simp-all
  qed simp-all

lemma ldistinct-conv-lnth:
  ldistinct xs  $\longleftrightarrow$  ( $\forall i j. enat i < llength xs \longrightarrow enat j < llength xs \longrightarrow i \neq j \longrightarrow lnth xs i \neq lnth xs j$ )
  (is ?lhs  $\longleftrightarrow$  ?rhs)
proof(intro iffI strip)
  assume ?rhs
  thus ?lhs
proof(coinduct xs)
  case (ldistinct xs)
  from  $\neg lnull xs$ 
  obtain x xs' where LCons: xs = LCons x xs'
  by(auto simp add: not-lnull-conv)
  have x  $\notin$  lset xs'
  proof
    assume x  $\in$  lset xs'
    then obtain j where enat j < llength xs' lnth xs' j = x
    unfolding lset-conv-lnth by auto
    hence enat 0 < llength xs enat (Suc j) < llength xs lnth xs (Suc j) = x lnth
    xs 0 = x
    by(simp-all add: LCons Suc-ile-eq zero-enat-def[symmetric])
    thus False by(auto dest: ldistinct(1)[rule-format])
  qed
  moreover {
    fix i j
    assume enat i < llength xs' enat j < llength xs' i  $\neq$  j
    hence enat (Suc i) < llength xs enat (Suc j) < llength xs
    by(simp-all add: LCons Suc-ile-eq)
    with  $\langle i \neq j \rangle$  have lnth xs (Suc i)  $\neq$  lnth xs (Suc j)
    by(auto dest: ldistinct(1)[rule-format])
    hence lnth xs' i  $\neq$  lnth xs' j unfolding LCons by simp }
  ultimately show ?case using LCons by simp
  qed

```

```

next
  assume ?lhs
  fix i j
  assume enat i < llength xs enat j < llength xs i ≠ j
  thus lnth xs i ≠ lnth xs j
  proof(induct i j rule: wlog-linorder-le)
    case symmetry thus ?case by simp
  next
    case (le i j)
    from ‹?lhs› have ldistinct (ldropn i xs) by(rule ldistinct-ldropn)
    also note ldropn-Suc-conv-ldropn[symmetric]
    also from le have i < j by simp
    hence lnth xs j ∈ lset (ldropn (Suc i) xs) using le unfolding in-lset-conv-lnth
      by(cases llength xs)(auto intro!: ext[where x=j - Suc i])
    ultimately show ?case using ‹enat i < llength xs› by auto
  qed
qed

lemma ldistinct-lmap [simp]:
ldistinct (lmap f xs) ↔ ldistinct xs ∧ inj-on f (lset xs)
(is ?lhs ↔ ?rhs)
proof(intro iffI conjI)
  assume dist: ?lhs
  thus ldistinct xs
    by(coinduct)(auto simp add: not-lnull-conv)

  show inj-on f (lset xs)
  proof(rule inj-onI)
    fix x y
    assume x ∈ lset xs and y ∈ lset xs and f x = f y
    then obtain i j
      where enat i < llength xs x = lnth xs i enat j < llength xs y = lnth xs j
      unfolding lset-conv-lnth by blast
    with dist ‹f x = f y› show x = y
      unfolding ldistinct-conv-lnth by auto
  qed
next
  assume ?rhs
  thus ?lhs
    by(coinduction arbitrary: xs)(auto simp add: not-lnull-conv)
qed

lemma ldistinct-lzipI1: ldistinct xs ==> ldistinct (lzip xs ys)
by(coinduction arbitrary: xs ys)(auto simp add: not-lnull-conv dest: lset-lzipD1)

lemma ldistinct-lzipI2: ldistinct ys ==> ldistinct (lzip xs ys)
by(coinduction arbitrary: xs ys)(auto 4 3 simp add: not-lnull-conv dest: lset-lzipD2)

```

2.23 Sortedness *lsorted*

context *ord* **begin**

coinductive *lsorted* :: '*a* llist \Rightarrow bool

where

LNil [*simp*]: *lsorted LNil*

 | *Singleton* [*simp*]: *lsorted (LCons x LNil)*

 | *LCons-LCons*: $\llbracket x \leq y; \text{lsorted}(\text{LCons } y \text{ xs}) \rrbracket \implies \text{lsorted}(\text{LCons } x (\text{LCons } y \text{ xs}))$

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 ttl, coinduct pred: lsorted*]:

assumes *major: X xs*

and *step: $\bigwedge_{xs. \llbracket X \text{ xs}; \neg \text{lnull } xs; \neg \text{lnull } (\text{ttl } xs) \rrbracket \implies \text{lhd } xs \leq \text{lhd } (\text{ttl } xs) \wedge (X \text{ } (\text{ttl } xs) \vee \text{lsorted } (\text{ttl } xs))$*

shows *lsorted xs*

using *major* **by** *coinduct(subst disj-commute, auto 4 4 simp add: neq-LNil-conv dest: step)*

lemma *lsorted-ltlI*: *lsorted xs \implies lsorted (ttl xs)*

by(*erule lsorted.cases*) *simp-all*

lemma *lsorted-lhdD*:

$\llbracket \text{lsorted } xs; \neg \text{lnull } xs; \neg \text{lnull } (\text{ttl } xs) \rrbracket \implies \text{lhd } xs \leq \text{lhd } (\text{ttl } xs)$

by(*auto elim: lsorted.cases*)

lemma *lsorted-LCons'*:

lsorted (LCons x xs) \longleftrightarrow ($\neg \text{lnull } xs \longrightarrow x \leq \text{lhd } xs \wedge \text{lsorted } xs$)

by(*cases xs*) *auto*

lemma *lsorted-lSup*:

$\llbracket \text{Complete-Partial-Order.chain } (\sqsubseteq) \text{ } Y; \forall xs \in Y. \text{lsorted } xs \rrbracket$

$\implies \text{lsorted } (\text{lSup } Y)$

proof(*coinduction arbitrary: Y*)

case (*lsorted Y*)

hence *sorted: $\bigwedge_{xs. xs \in Y} \text{lsorted } xs$* **by** *blast*

note *chain = <Complete-Partial-Order.chain (\sqsubseteq) Y>*

from $\langle \neg \text{lnull } (\text{lSup } Y) \rangle \langle \neg \text{lnull } (\text{ttl } (\text{lSup } Y)) \rangle$

obtain *xs where xs $\in Y \neg \text{lnull } xs \neg \text{lnull } (\text{ttl } xs)$ **by** *auto**

hence *lhd (lSup Y) = lhd xs lhd (ttl (lSup Y)) = lhd (ttl xs) lhd xs \leq lhd (ttl xs)*

using *chain sorted* **by**(*auto intro: lhd-lSup-eq chain-lprefix-ttl lsorted-lhdD*)

```

hence ?lhd by simp
moreover have ?ltl using chain sorted by(auto intro: chain-lprefix-ltl lsorted-ltlI)
ultimately show ?case ..
qed

lemma lsorted-lprefixD:
  [] xs ⊑ ys; lsorted ys ] ==> lsorted xs
proof(coinduction arbitrary: xs ys)
  case (lsorted xs ys)
  hence lhd xs = lhd ys lhd (ltl xs) = lhd (ltl ys)
    by(auto dest: lprefix-lhdD lprefix-ltlI)
  moreover have lhd ys ≤ lhd (ltl ys) using lsorted
    by(auto intro: lsorted-lhdD dest: lprefix-lnullD lprefix-ltlI)
  ultimately have ?lhd by simp
  moreover have ?ltl using lsorted by(blast intro: lsorted-ltlI lprefix-ltlI)
  ultimately show ?case ..
qed

lemma admissible-lsorted [cont-intro, simp]:
  assumes mcont: mcont lub ord lSup (⊑) (λx. f x)
  and ccpo: class ccpo lub ord (mk-less ord)
  shows ccpo.admissible lub ord (λx. lsorted (f x))
proof(rule ccpo.admissibleI)
  fix Y
  assume chain: Complete-Partial-Order.chain ord Y
  and sorted: ∀ x∈Y. lsorted (f x)
  and Y ≠ {}
  thus lsorted (f (lub Y))
    by(simp add: mcont-contD[OF mcont] lsorted-lSup chain-imageI mcont-monoD[OF
      mcont])
  qed

lemma admissible-not-lsorted[THEN admissible-subst, cont-intro, simp]:
  ccpo.admissible lSup (⊑) (λxs. ¬ lsorted xs)
by(rule ccpo.admissibleI)(auto dest: lsorted-lprefixD[rotated] intro: chain-lprefix-lSup)

lemma lsorted-ltake [simp]: lsorted xs ==> lsorted (ltake n xs)
by(rule lsorted-lprefixD)(rule ltake-is-lprefix)

lemma lsorted-ldropn [simp]: lsorted xs ==> lsorted (ldropn n xs)
by(induct n arbitrary: xs)(fastforce simp add: ldropn-Suc lsorted-LCons' ldropn-lnull
split: llist.split)+

lemma lsorted-ldrop [simp]: lsorted xs ==> lsorted (ldrop n xs)
by(induct xs arbitrary: n)(auto simp add: ldrop-LCons lsorted-LCons' ldrop-lnull
split: co.enat.split)

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
  assume ?lhs
  { fix y
    assume y  $\in$  lset xs
    hence x  $\leq$  y using <?lhs>
      by(induct arbitrary: x)(auto intro: order-trans) }
  with <?lhs> show ?rhs by cases auto
next
  assume ?rhs
  thus ?lhs by(cases xs) simp-all
qed

lemma lsorted-coinduct [consumes 1, case-names lsorted, case-conclusion lsorted
lhd ltl, coinduct pred: lsorted]:
  assumes major: X xs
  and step:  $\bigwedge xs. \llbracket X xs; \neg lnull xs \rrbracket \implies (\forall x \in lset (ltl xs). lhd xs \leq x) \wedge (X (ltl xs) \vee lsorted (ltl xs))$ 
  shows lsorted xs
  using major by(coinduct rule: lsorted-coinduct')(auto dest: step)

lemma lsortedD:  $\llbracket lsorted xs; \neg lnull xs; y \in lset (ltl xs) \rrbracket \implies lhd xs \leq y$ 
by(clarify simp add: not-lnull-conv lsorted-LCons)

end

lemma lsorted-lmap':
  assumes ord.lsorted orda xs monotone orda ordb f
  shows ord.lsorted ordb (lmap f xs)
  using <ord.lsorted orda xs>
  by(coinduction arbitrary: xs rule: ord.lsorted-coinduct')(auto intro: monotoneD[OF
<monotone orda ordb f>] ord.lsorted-lhdD ord.lsorted-ltlI)

lemma lsorted-lmap:
  assumes lsorted xs monotone ( $\leq$ ) ( $\leq$ ) f
  shows lsorted (lmap f xs)
  using <lsorted xs>
  by(coinduction arbitrary: xs rule: lsorted-coinduct')(auto intro: monotoneD[OF
<monotone ( $\leq$ ) ( $\leq$ ) f>] lsorted-lhdD lsorted-ltlI)

context linorder begin

```

```

lemma lsorted-ldistinct-lset-unique:
   $\llbracket \text{lsorted } xs; \text{ldistinct } xs; \text{lsorted } ys; \text{ldistinct } ys; \text{lset } xs = \text{lset } ys \rrbracket$ 
   $\implies xs = ys$ 
proof(coinduction arbitrary: xs ys)
  case (Eq-llist xs ys)
  hence ?lnull by(cases ys)(auto simp add: lset-lnull)
  moreover from Eq-llist have ?LCons
    by(auto 4 3 intro: lsorted-ltlI ldistinct-ltlI simp add: not-lnull-conv insert-eq-iff
  lsorted-LCons split: if-split-asm)
    ultimately show ?case ..
qed

end

lemma lsorted-llist-of[simp]: lsorted (llist-of xs)  $\longleftrightarrow$  sorted xs
by(induct xs)(auto simp: lsorted-LCons)

```

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:  $\bigwedge_{xs\ ys} \llbracket X\ xs\ ys; \neg\ lnull\ xs \rrbracket$ 
   $\implies \neg\ lnull\ ys \wedge$ 
   $(\neg\ lnull\ ys \longrightarrow r\ (lhd\ xs)\ (lhd\ ys) \vee$ 
   $lhd\ xs = lhd\ ys \wedge (X\ (ltl\ xs)\ (ltl\ ys)) \vee llexord\ r\ (ltl\ xs)\ (ltl\ ys))$ 
shows llexord r xs ys
using X
proof(coinduct)
  case (llexord xs ys)
  thus ?case
    by(cases xs ys rule: llist.exhaust[case-product llist.exhaust])(auto dest: step)
qed

lemma llexord-refl [simp, intro!]:
  llexord r xs xs
proof -
  define ys where ys = xs
  hence xs = ys by simp
  thus llexord r xs ys
    by(coinduct xs ys) auto
qed

lemma llexord-LCons-LCons [simp]:
  llexord r (LCons x xs) (LCons y ys)  $\longleftrightarrow$  (x = y  $\wedge$  llexord r xs ys  $\vee$  r x y)
by(auto intro: llexord.intros(1,2) elim: llexord.cases)

lemma lnull-llexord [simp]: lnull xs  $\implies$  llexord r xs ys
unfolding lnull-def by simp

```

```

lemma llexord-LNil-right [simp]:
  lnull ys  $\implies$  llexord r xs ys  $\longleftrightarrow$  lnull xs
by(auto elim: llexord.cases)

lemma llexord-LCons-left:
  llexord r (LCons x xs) ys  $\longleftrightarrow$ 
  ( $\exists y\; ys = LCons\; y\; ys' \wedge (x = y \wedge llexord\; r\; xs\; ys' \vee r\; x\; y)$ )
by(cases ys)(auto elim: llexord.cases)

lemma lprefix-imp-llexord:
  assumes xs  $\sqsubseteq$  ys
  shows llexord r xs ys
  using assms
by(coinduct)(auto simp add: not-lnull-conv LCons-lprefix-conv)

lemma llexord-empty:
  llexord ( $\lambda x\; y.\; False$ ) xs ys = xs  $\sqsubseteq$  ys
proof
  assume llexord ( $\lambda x\; y.\; False$ ) xs ys
  thus xs  $\sqsubseteq$  ys
    by(coinduct)(auto elim: llexord.cases)
qed(rule lprefix-imp-llexord)

lemma llexord-append-right:
  llexord r xs (lappend xs ys)
by(rule lprefix-imp-llexord)(auto simp add: lprefix-conv-lappend)

lemma llexord-lappend-leftI:
  assumes llexord r ys zs
  shows llexord r (lappend xs ys) (lappend xs zs)
proof(cases lfinite xs)
  case True thus ?thesis by induct (simp-all add: assms)
next
  case False thus ?thesis by(simp add: lappend-inf)
qed

lemma llexord-lappend-leftD:
  assumes lex: llexord r (lappend xs ys) (lappend xs zs)
  and fin: lfinite xs
  and irrefl:  $\forall x.\; x \in lset\; xs \implies \neg r\; x\; x$ 
  shows llexord r ys zs
  using fin lex irrefl by(induct) simp-all

lemma llexord-lappend-left:
   $\llbracket lfinite\; xs;\; \forall x.\; x \in lset\; xs \implies \neg r\; x\; x \rrbracket$ 
   $\implies$  llexord r (lappend xs ys) (lappend xs zs)  $\longleftrightarrow$  llexord r ys zs
by(blast intro: llexord-lappend-leftI llexord-lappend-leftD)

```

```

lemma antisym-llexord:
  assumes r: antisymp r
  and irrefl:  $\bigwedge x. \neg r x x$ 
  shows antisymp (llexord r)
proof(rule antisympI)
  fix xs ys
  assume llexord r xs ys
  and llexord r ys xs
  hence llexord r xs ys  $\wedge$  llexord r ys xs by auto
  thus xs = ys
  by (coinduct rule: llist.coinduct)
    (auto 4 3 simp add: not-lnull-conv irrefl dest: antisympD[OF r, simplified])
qed

lemma llexord-antisym:
   $\llbracket \text{llexord } r \text{ } xs \text{ } ys; \text{llexord } r \text{ } ys \text{ } xs; \\ !!a \text{ } b. \llbracket r \text{ } a \text{ } b; \text{ } r \text{ } b \text{ } a \rrbracket \implies \text{False} \rrbracket \\ \implies xs = ys$ 
using antisympD[OF antisym-llexord, of r xs ys]
by(auto intro: antisympI)

lemma llexord-trans:
  assumes 1: llexord r xs ys
  and 2: llexord r ys zs
  and trans:  $\text{!!}a \text{ } b \text{ } c. \llbracket r \text{ } a \text{ } b; \text{ } r \text{ } b \text{ } c \rrbracket \implies r \text{ } a \text{ } c$ 
  shows llexord r xs zs
proof –
  from 1 2 have  $\exists ys. \text{llexord } r \text{ } xs \text{ } ys \wedge \text{llexord } r \text{ } ys \text{ } zs$  by blast
  thus ?thesis
    by(coinduct)(auto 4 3 simp add: not-lnull-conv llexord-LCons-left dest: trans)
qed

lemma trans-llexord:
  transp r  $\implies$  transp (llexord r)
  by(auto intro!: transpI elim: llexord-trans dest: transpD)

lemma llexord-linear:
  assumes linear:  $\text{!!}x \text{ } y. \text{ } r \text{ } x \text{ } y \vee x = y \vee r \text{ } y \text{ } x$ 
  shows llexord r xs ys  $\vee$  llexord r ys xs
proof(rule disjCI)
  assume  $\neg \text{llexord } r \text{ } ys \text{ } xs$ 
  thus llexord r xs ys
proof(coinduct rule: llexord-coinduct)
  case (llexord xs ys)
  show ?case
  proof(cases xs)
    case LNil thus ?thesis using llexord by simp
  next
    case (LCons x xs')

```

```

with ⟨¬ llexord r ys xs⟩ obtain y ys'
  where ys: ys = LCons y ys' ∧ r y x y ≠ x ∨ ¬ llexord r ys' xs'
    by(cases ys) auto
  with ⟨¬ r y x⟩ linear[of x y] ys LCons show ?thesis by auto
qed
qed
qed

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)
by auto

lemma llexord-conv:
llexord r xs ys ↔
xs = ys ∨
(∃ zs xs' y ys'. lfinite zs ∧ xs = lappend zs xs' ∧ ys = lappend zs (LCons y ys') ∧
(xs' = LNil ∨ r (lhd xs') y))
(is ?lhs ↔ ?rhs)
proof
assume ?lhs
show ?rhs (is - ∨ ?prefix)
proof(rule disjCI)
assume ¬ ?prefix
with ⟨?lhs⟩ show xs = ys
proof(coinduction arbitrary: xs ys)
case (Eq-llist xs ys)
hence llexord r xs ys
and prefix: ∀ zs xs' y ys'. [| lfinite zs; xs = lappend zs xs';
ys = lappend zs (LCons y ys') |]
implies xs' ≠ LNil ∧ ¬ r (lhd xs') y
by auto
from ⟨llexord r xs ys⟩ show ?case
proof(cases)
case (llexord-LCons-eq xs' ys' x)
{ fix zs xs'' y ys''
assume lfinite zs xs' = lappend zs xs''
and ys' = lappend zs (LCons y ys'')
hence lfinite (LCons x zs) xs = lappend (LCons x zs) xs''
and ys = lappend (LCons x zs) (LCons y ys'')
using llexord-LCons-eq by simp-all
hence xs'' ≠ LNil ∧ ¬ r (lhd xs'') y by(rule prefix) }
with llexord-LCons-eq show ?thesis by auto
next
case (llexord-LCons-less x y xs' ys')
with prefix[of LNil xs y ys'] show ?thesis by simp
next
case llexord-LNil

```

```

thus ?thesis using prefix[of LNil xs lhd ys ltl ys]
  by(cases ys) simp-all
qed
qed
qed
next
assume ?rhs
thus ?lhs
proof(coinduct xs ys)
  case (llexord xs ys)
  show ?case
  proof(cases xs)
    case LNil thus ?thesis using llexord by simp
  next
  case (LCons x xs')
  with llexord obtain y ys' where ys = LCons y ys'
    by(cases ys)(auto dest: sym simp add: LNil-eq-lappend-iff)
  show ?thesis
  proof(cases x = y)
    case True
    from llexord(1) show ?thesis
  proof
    assume xs = ys
    with True LCons `ys = LCons y ys'` show ?thesis by simp
  next
  assume ∃zs xs' y ys'. lfinite zs ∧ xs = lappend zs xs' ∧
    ys = lappend zs (LCons y ys') ∧
    (xs' = LNil ∨ r (lhd xs') y)
  then obtain zs xs' y' ys'''
    where lfinite zs xs = lappend zs xs'
      and ys = lappend zs (LCons y' ys'')
      and xs' = LNil ∨ r (lhd xs') y' by blast
    with True LCons `ys = LCons y ys'` show ?thesis by(cases zs) auto
  qed
  next
  case False
  with LCons llexord `ys = LCons y ys'`
  have r x y by(fastforce elim: lfinite.cases)
  with LCons `ys = LCons y ys'` show ?thesis by auto
qed
qed
qed
qed

```

lemma llexord-conv-ltake-index:

$$\text{llexord } r \text{ } xs \text{ } ys \longleftrightarrow (llength xs \leq llength ys \wedge ltake (llength xs) ys = xs) \vee (\exists n. \text{enat } n < \min (llength xs) (llength ys) \wedge$$

```

ltake (enat n) xs = ltake (enat n) ys ∧ r (lnth xs n) (lnth ys n))
(is ?lhs ↔ ?rhs)
proof(rule iffI)
assume ?lhs
thus ?rhs (is ?A ∨ ?B) unfolding llexord-conv
proof
assume xs = ys thus ?thesis by(simp add: ltake-all)
next
assume ∃ zs xs' y ys'. lfinite zs ∧ xs = lappend zs xs' ∧
ys = lappend zs (LCons y ys') ∧
(xs' = LNil ∨ r (lhd xs') y)
then obtain zs xs' y ys'
where lfinite zs xs' = LNil ∨ r (lhd xs') y
and [simp]: xs = lappend zs xs' ys = lappend zs (LCons y ys')
by blast
show ?thesis
proof(cases xs')
case LNil
hence ?A by(auto intro: enat-le-plus-same simp add: ltake-lappend1 ltake-all)
thus ?thesis ..
next
case LCons
with ⟨xs' = LNil ∨ r (lhd xs') y⟩ have r (lhd xs') y by simp
from ⟨lfinite zs⟩ obtain zs' where [simp]: zs = llist-of zs'
unfolding lfinite-eq-range-llist-of by blast
with LCons have enat (length zs') < min (llength xs) (llength ys)
by(auto simp add: less-enat-def eSuc-def split: enat.split)
moreover have ltake (enat (length zs')) xs = ltake (enat (length zs')) ys
by(simp add: ltake-lappend1)
moreover have r (lnth xs (length zs')) (lnth ys (length zs'))
using LCons ⟨r (lhd xs') y⟩
by(simp add: lappend-llist-of-LCons lnth-lappend1)
ultimately have ?B by blast
thus ?thesis ..
qed
qed
next
assume ?rhs (is ?A ∨ ?B)
thus ?lhs
proof
assume ?A thus ?thesis
proof(coinduct)
case (llexord xs ys)
thus ?case by(cases xs, simp)(cases ys, auto)
qed
next
assume ?B
then obtain n where len: enat n < min (llength xs) (llength ys)
and takexs: ltake (enat n) xs = ltake (enat n) ys

```

```

and r:  $r (\lnth xs n) (\lnth ys n)$  by blast
have  $xs = lappend (ltake (enat n) xs) (ldrop (enat n) xs)$ 
  by(simp only: lappend-ltake-ldrop)
moreover from takexs len
have  $ys = lappend (ltake (enat n) xs) (LCons (\lnth ys n) (ldrop (enat (Suc n))$ 
 $ys)))$ 
  by(simp add: ldropn-Suc-conv-ldropn ldrop-enat)
moreover from r len
have  $r (lhd (ldrop (enat n) xs)) (\lnth ys n)$ 
  by(simp add: lhd-ldrop)
moreover have lfinite (ltake (enat n) xs) by simp
ultimately show ?thesis unfolding llexord-conv by blast
qed
qed

```

lemma llexord-llist-of:

$$\text{llexord } r \text{ (llist-of } xs) \text{ (llist-of } ys) \longleftrightarrow$$

$$xs = ys \vee (xs, ys) \in \text{lexord } \{(x, y). r x y\}$$

$$(\text{is } ?lhs \longleftrightarrow ?rhs)$$

proof

$$\begin{aligned} &\text{assume } ?lhs \\ &\{ \text{fix } zs \text{ } xs' \text{ } y \text{ } ys' \\ &\quad \text{assume lfinite } zs \text{ llist-of } xs = lappend \text{ } zs \text{ } xs' \\ &\quad \text{and llist-of } ys = lappend \text{ } zs \text{ (LCons } y \text{ } ys') \\ &\quad \text{and } xs' = LNil \vee r (lhd \text{ } xs') \text{ } y \\ &\quad \text{from } \langle \text{lfinite } zs \rangle \text{ obtain } zs' \text{ where [simp]: } zs = \text{llist-of } zs' \\ &\quad \text{unfolding lfinite-eq-range.llist-of by blast} \\ &\quad \text{have lfinite (llist-of } xs) \text{ by simp} \\ &\quad \text{hence lfinite } xs' \text{ unfolding } \langle \text{llist-of } xs = lappend \text{ } zs \text{ } xs' \rangle \text{ by simp} \\ &\quad \text{then obtain } XS' \text{ where [simp]: } xs' = \text{llist-of } XS' \\ &\quad \text{unfolding lfinite-eq-range.llist-of by blast} \\ &\quad \text{from } \langle \text{llist-of } xs = lappend \text{ } zs \text{ } xs' \rangle \text{ have [simp]: } xs = zs' @ XS' \\ &\quad \text{by(simp add: lappend-llist-of-llist-of)} \\ &\quad \text{have lfinite (llist-of } ys) \text{ by simp} \\ &\quad \text{hence lfinite } ys' \text{ unfolding } \langle \text{llist-of } ys = lappend \text{ } zs \text{ (LCons } y \text{ } ys') \rangle \text{ by simp} \\ &\quad \text{then obtain } YS' \text{ where [simp]: } ys' = \text{llist-of } YS' \\ &\quad \text{unfolding lfinite-eq-range.llist-of by blast} \\ &\quad \text{from } \langle \text{llist-of } ys = lappend \text{ } zs \text{ (LCons } y \text{ } ys') \rangle \text{ have [simp]: } ys = zs' @ y \# YS' \\ &\quad \text{by(auto simp add: llist-of.simps(2)[symmetric] lappend-llist-of-llist-of simp del: llist-of.simps(2))} \\ &\quad \text{have } (\exists a \text{ } ys'. ys = xs @ a \# ys') \vee \\ &\quad (\exists zs \text{ } a \text{ } b. r \text{ } a \text{ } b \wedge (\exists xs'. xs = zs @ a \# xs') \wedge (\exists ys'. ys = zs @ b \# ys')) \\ &\quad (\text{is } ?A \vee ?B) \\ &\quad \text{proof(cases } xs') \\ &\quad \quad \text{case } LNil \text{ thus ?thesis by(auto simp add: llist-of-eq-LNil-conv)} \\ &\quad \quad \text{next} \\ &\quad \quad \text{case } (LCons } x \text{ } xs'') \\ &\quad \quad \text{with } \langle xs' = LNil \vee r (lhd \text{ } xs') \text{ } y \rangle \\ &\quad \quad \text{have } r (lhd \text{ } xs') \text{ } y \text{ by(auto simp add: llist-of-eq-LCons-conv)}$$

```

with LCons have ?B by(auto simp add: llist-of-eq-LCons-conv) fastforce
thus ?thesis ..

qed
hence (xs, ys) ∈ {(x, y). ∃ a v. y = x @ a # v ∨
                     (∃ u a b v w. (a, b) ∈ {(x, y). r x y} ∧
                                   x = u @ a # v ∧ y = u @ b # w)}
by auto }
with ‹?lhs› show ?rhs
  unfolding lexord-def llexord-conv llist-of-inject by blast
next
assume ?rhs
thus ?lhs
proof
  assume xs = ys thus ?thesis by simp
next
  assume (xs, ys) ∈ lexord {(x, y). r x y}
  thus ?thesis
    by(coinduction arbitrary: xs ys)(auto, auto simp add: neq-Nil-conv)
qed
qed

```

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)
  by(simp-all add: lfilter.simps)

declare lfilter.mono[cont-intro]

lemma mono2mono-lfilter[THEN llist.mono2mono, simp, cont-intro]:
  shows monotone-lfilter: monotone (≤) (≤) (lfilter P)
  by(rule llist.fixp-preserves-mono1[OF lfilter.mono lfilter-def]) simp

lemma mcont2mcont-lfilter[THEN llist.mcont2mcont, simp, cont-intro]:
  shows mcont-lfilter: mcont lSup (≤) lSup (≤) (lfilter P)
  by(rule llist.fixp-preserves-mcont1[OF lfilter.mono lfilter-def]) simp

lemma lfilter-mono [partial-function-mono]:
  mono-list A ==> mono-list (λf. lfilter P (A f))
  by(rule mono2mono-lfilter)

lemma lfilter-LCons-seek: ~ (p x) ==> lfilter p (LCons x l) = lfilter p l
by simp

lemma lfilter-LCons-found:
  P x ==> lfilter P (LCons x xs) = LCons x (lfilter P xs)
by simp

```

```

lemma lfilter-eq-LNil: lfilter P xs = LNil  $\longleftrightarrow$  ( $\forall x \in lset xs. \neg P x$ )
by(induction xs) simp-all

notepad begin
fix P xs
have (lfilter P xs = LNil)  $\longleftrightarrow$  ( $\forall x \in lset xs. \neg P x$ )
proof(intro iffI strip)
assume  $\forall x \in lset xs. \neg P x$ 
hence lfilter P xs  $\subseteq$  LNil
by(induction arbitrary: xs rule: lfilter.fixp-induct)(simp-all split: llist.split del: lprefix-LNil)
thus lfilter P xs = LNil by simp
next
fix x
assume x  $\in lset xs$  lfilter P xs = LNil
thus  $\neg P x$  by induction(simp-all split: if-split-asm)
qed
end

lemma diverge-lfilter-LNil [simp]:  $\forall x \in lset xs. \neg P x \implies lfilter P xs = LNil$ 
by(simp add: lfilter-eq-LNil)

lemmas lfilter-False = diverge-lfilter-LNil

lemma lnull-lfilter [simp]: lnull (lfilter P xs)  $\longleftrightarrow$  ( $\forall x \in lset xs. \neg P x$ )
unfolding lnull-def by(simp add: lfilter-eq-LNil)

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

lemma lhd-lfilter [simp]: lhd (lfilter P xs) = lhd (ldropWhile (Not o P) xs)
proof(cases  $\exists x \in lset xs. P x$ )
case True
then obtain x where x  $\in lset xs$  and P x by blast
from  $\langle x \in lset xs \rangle$  show ?thesis by induct(simp-all add:  $\langle P x \rangle$ )
qed(simp add: o-def)

lemma ltl-lfilter: ltl (lfilter P xs) = lfilter P (ltl (ldropWhile (Not o P) xs))
by(induct xs) simp-all

lemma lfilter-eq-LCons:
lfilter P xs = LCons x xs'  $\implies$ 
 $\exists xs''. xs' = lfilter P xs'' \wedge ldropWhile (Not o P) xs = LCons x xs''$ 
by(drule eq-LConsD)(auto intro!: exI simp add: ltl-lfilter o-def ldropWhile-eq-LNil-iff
intro: llist.expand)

lemma lfilter-K-True [simp]: lfilter (%-. True) xs = xs
by(induct xs) simp-all

```

```

lemma lfilter-K-False [simp]: lfilter (λ-. False) xs = LNil
by simp

lemma lfilter-lappend-lfinite [simp]:
  lfinite xs  $\implies$  lfilter P (lappend xs ys) = lappend (lfilter P xs) (lfilter P ys)
by(induct rule: lfinite.induct) auto

lemma lfinite-lfilterI [simp]: lfinite xs  $\implies$  lfinite (lfilter P xs)
by(induct rule: lfinite.induct) simp-all

lemma lset-lfilter [simp]: lset (lfilter P xs) = {x ∈ lset xs. P x}
by induct(auto simp add: Collect-conj-eq)

notepad begin — show lset-lfilter by fixpoint induction
note [simp del] = lset-lfilter
fix P xs
have lset (lfilter P xs) = lset xs ∩ {x. P x} (is ?lhs = ?rhs)
proof
  show ?lhs ⊆ ?rhs
  by(induct arbitrary: xs rule: lfilter.fixp-induct)(auto split: llist.split)
  show ?rhs ⊆ ?lhs
proof
  fix x
  assume x ∈ ?rhs
  hence x ∈ lset xs P x by simp-all
  thus x ∈ ?lhs by induction simp-all
qed
qed
end

lemma lfilter-lfilter: lfilter P (lfilter Q xs) = lfilter (λx. P x ∧ Q x) xs
by(induction xs) simp-all

notepad begin — show lfilter-lfilter by fixpoint induction
fix P Q xs
have ∀ xs. lfilter P (lfilter Q xs) ⊑ lfilter (λx. P x ∧ Q x) xs
  by(rule lfilter.fixp-induct)(auto split: llist.split)
moreover have ∀ xs. lfilter (λx. P x ∧ Q x) xs ⊑ lfilter P (lfilter Q xs)
  by(rule lfilter.fixp-induct)(auto split: llist.split)
ultimately have lfilter P (lfilter Q xs) = lfilter (λx. P x ∧ Q x) xs
  by(blast intro: lprefix-antisym)
end

lemma lfilter-idem [simp]: lfilter P (lfilter P xs) = lfilter P xs
by(simp add: lfilter-lfilter)

lemma lfilter-lmap: lfilter P (lmap f xs) = lmap f (lfilter (P o f) xs)
by(induct xs) simp-all

```

```

lemma lfilter-llist-of [simp]:
  lfilter P (llist-of xs) = llist-of (filter P xs)
by(induct xs) simp-all

lemma lfilter-cong [cong]:
assumes xsys: xs = ys
and set:  $\bigwedge x. x \in lset ys \implies P x = Q x$ 
shows lfilter P xs = lfilter Q ys
using set unfolding xsys
by(induction ys)(simp-all add: Bex-def[symmetric])

lemma llength-lfilter-ile:
  llength (lfilter P xs)  $\leq$  llength xs
by(induct xs)(auto intro: order-trans)

lemma lfinite-lfilter:
  lfinite (lfilter P xs)  $\longleftrightarrow$ 
  lfinite xs  $\vee$  finite {n. enat n < llength xs  $\wedge$  P (lnth xs n)}
proof
  assume lfinite (lfilter P xs)
  { assume  $\neg$  lfinite xs
    with lfinite (lfilter P xs)
    have finite {n. enat n < llength xs  $\wedge$  P (lnth xs n)}
    proof(induct ys=lfilter P xs arbitrary: xs rule: lfinite-induct)
      case LNil
      hence  $\forall x \in lset xs. \neg P x$  by(auto)
      hence eq: {n. enat n < llength xs  $\wedge$  P (lnth xs n)} = {}
      by(auto simp add: lset-conv-lnth)
      show ?case unfolding eq ..
    next
      case (LCons xs)
      from  $\neg lnull (lfilter P xs)$ 
      have exP:  $\exists x \in lset xs. P x$  by simp
      let ?xs = ltl (ldropWhile ( $\lambda x. \neg P x$ ) xs)
      let ?xs' = ltakeWhile ( $\lambda x. \neg P x$ ) xs
      from exP obtain n where n: llength ?xs' = enat n
      using lfinite-conv-llength-enat[of ?xs']
      by(auto simp add: lfinite-ltakeWhile)
      have finite ({n. enat n < llength ?xs}  $\cap$  {n. P (lnth ?xs n)}) (is finite ?A)
        using LCons [[simproc add: finite-Collect]] by(auto simp add: ltl-lfilter
        lfinite-ldropWhile)
        hence finite (( $\lambda m. n + 1 + m$ ) ` ({n. enat n < llength ?xs}  $\cap$  {n. P (lnth
        ?xs n)}))
        (is finite (?f ` -)
        by(rule finite-imageI)
        moreover have xs: xs = lappend ?xs' (LCons (lhd (ldropWhile ( $\lambda x. \neg P x$ )
        xs)) ?xs)
        using exP by simp
        { fix m

```

```

assume m < n
hence lnth ?xs' m ∈ lset ?xs' using n
    unfolding in-lset-conv-lnth by auto
    hence ¬ P (lnth ?xs' m) by(auto dest: lset-ltakeWhileD) }
    hence {n. enat n < llength xs ∧ P (lnth xs n)} ⊆ insert (the-enat (llength
?xs')) (?f ' ?A)
        using n by(subst (1 2) xs)(cases llength ?xs, auto simp add: lnth-lappend
not-less not-le lnth-LCons' eSuc-enat split: if-split-asm intro!: rev-image-eqI)
        ultimately show ?case by(auto intro: finite-subset)
    qed }
thus lfinite xs ∨ finite {n. enat n < llength xs ∧ P (lnth xs n)} by blast
next
assume lfinite xs ∨ finite {n. enat n < llength xs ∧ P (lnth xs n)}
moreover {
    assume lfinite xs
    with llength-lfilter-ile[of P xs] have lfinite (lfilter P xs)
        by(auto simp add: lfinite-eq-range-llist-of)
} moreover {
    assume nfin: ¬ lfinite xs
    hence len: llength xs = ∞ by(rule not-lfinite-llength)
    assume fin: finite {n. enat n < llength xs ∧ P (lnth xs n)}
    then obtain m where {n. enat n < llength xs ∧ P (lnth xs n)} ⊆ {..<m}
        unfolding finite-nat-iff-bounded by auto
    hence ⋀n. [m ≤ n; enat n < llength xs] ==> ¬ P (lnth xs n) by auto
    hence lfinite (lfilter P xs)
    proof(induct m arbitrary: xs)
        case 0
        hence lnull (lfilter P xs) by(auto simp add: in-lset-conv-lnth)
        thus ?case by simp
    next
        case (Suc m)
        { fix n
            assume m ≤ n enat n < llength (ltl xs)
            hence Suc m ≤ Suc n enat (Suc n) < llength xs
                by(case-tac [] xs)(auto simp add: Suc-ile-eq)
            hence ¬ P (lnth xs (Suc n)) by(rule Suc)
            hence ¬ P (lnth (ltl xs) n)
                using <enat n < llength (ltl xs)> by(cases xs) (simp-all) }
            hence lfinite (lfilter P (ltl xs)) by(rule Suc)
            thus ?case by(cases xs) simp-all
        qed }
        ultimately show lfinite (lfilter P xs) by blast
    qed

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 –

```

```

let ?us = ltakeWhile (Not o P) ys
  and ?vs = ltl (ldropWhile (Not o P) ys)
from assms have ¬ lnull (lfilter P ys) by auto
hence exP: ∃ x ∈ lset ys. P x by simp
from eq-LConsD[OF assms]
have x: x = lhd (ldropWhile (Not o P) ys)
  and xs: xs = ltl (lfilter P ys) by auto
from exP
have ys = lappend ?us (LCons x ?vs) unfolding x by simp
moreover have lfinite ?us using exP by(simp add: lfinite-ltakeWhile)
moreover have ∀ u ∈ lset ?us. ¬ P u by(auto dest: lset-ltakeWhileD)
moreover have P x unfolding x o-def
  using exP by(rule lhd-ldropWhile[where P=Not o P, simplified])
moreover have xs = lfilter P ?vs unfolding xs by(simp add: ltl-lfilter)
ultimately show ?thesis by blast
qed

```

```

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
using ⟨lfinite ys⟩ ⟨lfilter P xs = lappend ys zs⟩
proof(induct arbitrary: xs zs)
case lfinite-LNil
hence xs = lappend LNil xs LNil = lfilter P LNil zs = lfilter P xs
  by simp-all
thus ?case by blast
next
case (lfinite-LConsI ys y)
note IH = ⟨∀ xs zs. lfilter P xs = lappend ys zs ⟹
  ∃ us vs. xs = lappend us vs ∧ lfinite us ∧
  ys = lfilter P us ∧ zs = lfilter P vs⟩
from ⟨lfilter P xs = lappend (LCons y ys) zs⟩
have lfilter P xs = LCons y (lappend ys zs) by simp
from lfilter-eq-LConsD[OF this] obtain us vs
  where xs: xs = lappend us (LCons y vs) lfinite us
    P y ∀ u ∈ lset us. ¬ P u
  and vs: lfilter P vs = lappend ys zs by auto
from IH[OF vs] obtain us' vs' where vs = lappend us' vs' lfinite us'
  and ys = lfilter P us' zs = lfilter P vs' by blast
with xs show ?case
by(fastforce simp add: lappend-snocL1-conv-LCons2[symmetric, where ys=lappend
  us' vs'])
  lappend-assoc[symmetric])
qed

```

```

lemma ldistinct-lfilterI: ldistinct xs ⟹ ldistinct (lfilter P xs)
by(induction xs) simp-all

```

```

notepad begin
fix P xs
assume *: ldistinct xs
from * have ldistinct (lfilter P xs) ∧ lset (lfilter P xs) ⊆ lset xs
  by(induction arbitrary: xs rule: lfilter.fixp-induct)(simp-all split: llist.split, fast-force)
from * have ldistinct (lfilter P xs)
  — only works because we use strong fixpoint induction
  by(induction arbitrary: xs rule: lfilter.fixp-induct)(simp-all split: llist.split, auto
4 4 dest: monotone-lset[THEN monotoneD] simp add: fun-ord-def)
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(induct n m rule: wlog-linorder-le)
  case symmetry thus ?case by simp
next
  case (le n m)
  thus ?case
    proof(induct n arbitrary: xs m)
      case 0 thus ?case
      proof(cases m)
        case 0 thus ?thesis .
      next
        case (Suc m')
        with 0 show ?thesis
          by(cases xs)(simp-all add: Suc-ile-eq, auto simp add: lset-conv-lnth)
      qed
    next
    case (Suc n)
    from <Suc n ≤ m> obtain m' where m [simp]: m = Suc m' by(cases m) simp
    with <Suc n ≤ m> have n ≤ m' by simp
    moreover from <enat (Suc n) < llength xs>
    obtain x xs' where xs [simp]: xs = LCons x xs' by(cases xs) simp
    from <ldistinct (lfilter P xs)> have ldistinct (lfilter P xs') by(simp split: if-split-asm)
    moreover from <enat (Suc n) < llength xs> <enat m < llength xs>
    have enat n < llength xs' enat m' < llength xs' by(simp-all add: Suc-ile-eq)
    moreover note <P a>
    moreover have lnth xs' n = a lnth xs' m' = a
      using <lnth xs (Suc n) = a> <lnth xs m = a> by simp-all
    ultimately have m' = n by(rule Suc)
    thus ?case by simp
  qed
qed

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:  $\bigwedge x y. P x y \implies Q1 x \longleftrightarrow Q2 y$ 
  shows llist-all2 P (lfilter Q1 xs) (lfilter Q2 ys)
  using * by(induction arbitrary: xs ys rule: lfilter-fixp-parallel-induct)(auto split:
  llist.split dest: Q)

lemma distinct-filterD:
   $\llbracket \text{distinct } (\text{filter } P \text{ xs}); n < \text{length } xs; m < \text{length } xs; P x; xs ! n = x; xs ! m = x \rrbracket \implies m = n$ 
  using ldistinct-lfilterD[of P llist-of xs n m x] by simp

lemma lprefix-lfilterI:
  xs ⊑ ys  $\implies$  lfilter P xs ⊑ lfilter P ys
  by(rule monotoneD[OF monotone-lfilter])

context preorder begin

lemma lsorted-lfilterI:
  lsorted xs  $\implies$  lsorted (lfilter P xs)
  by(induct xs)(simp-all add: lsorted-LCons)

lemma lsorted-lfilter-same:
  lsorted (lfilter ( $\lambda x. x = c$ ) xs)
  by(induct xs)(auto simp add: lsorted-LCons)

end

lemma lfilter-id-conv: lfilter P xs = xs  $\longleftrightarrow$  ( $\forall x \in \text{lset } xs. P x$ ) (is ?lhs  $\longleftrightarrow$  ?rhs)
proof
  assume ?rhs thus ?lhs by(induct xs) auto
next
  assume ?lhs
  with lset-lfilter[of P xs] show ?rhs by auto
qed

lemma lfilter-repeat [simp]: lfilter P (repeat x) = (if P x then repeat x else LNil)
by(simp add: lfilter-id-conv)

```

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)
  by(simp-all add: lconcat.simps)

declare lconcat.mono[cont-intro]

```

```

lemma mono2mono-lconcat[THEN llist.mono2mono, cont-intro, simp]:
  shows monotone-lconcat: monotone ( $\sqsubseteq$ ) ( $\sqsubseteq$ ) lconcat
  by(rule llist.fixp-preserves-mono1[OF lconcat.mono lconcat-def]) simp

lemma mcont2mcont-lconcat[THEN llist.mcont2mcont, cont-intro, simp]:
  shows mcont-lconcat: mcont lSup ( $\sqsubseteq$ ) lSup ( $\sqsubseteq$ ) lconcat
  by(rule llist.fixp-preserves-mcont1[OF lconcat.mono lconcat-def]) simp

lemma lconcat-eq-LNil: lconcat xss = LNil  $\longleftrightarrow$  lset xss  $\subseteq \{LNil\}$  (is ?lhs  $\longleftrightarrow$  ?rhs)
  by(induction xss)(auto simp add: lappend-eq-LNil-iff)

lemma lnull-lconcat [simp]: lnull (lconcat xss)  $\longleftrightarrow$  lset xss  $\subseteq \{xs. lnull xs\}$ 
  unfolding lnull-def by(simp add: lconcat-eq-LNil)

lemma lconcat-llist-of:
  lconcat (llist-of (map llist-of xs)) = llist-of (concat xs)
  by(induct xs)(simp-all add: lappend-llist-of-llist-of)

lemma lhd-lconcat [simp]:
   $\llbracket \neg lnull xss; \neg lnull (lhd xss) \rrbracket \implies lhd (lconcat xss) = lhd (lhd xss)$ 
  by(clar simp simp add: not-lnull-conv)

lemma ltl-lconcat [simp]:
   $\llbracket \neg lnull xss; \neg lnull (lhd xss) \rrbracket \implies ltl (lconcat xss) = lappend (ltl (lhd xss))$ 
  (lconcat (ltl xss))
  by(clar simp simp add: not-lnull-conv)

lemma lmap-lconcat:
  lmap f (lconcat xss) = lconcat (lmap (lmap f) xss)
  by(induct xss)(simp-all add: lmap-lappend-distrib)

lemma lconcat-lappend [simp]:
  assumes lfinite xss
  shows lconcat (lappend xss yss) = lappend (lconcat xss) (lconcat yss)
  using assms
  by induct (simp-all add: lappend-assoc)

lemma lconcat-eq-LCons-conv:
  lconcat xss = LCons x xs  $\longleftrightarrow$ 
   $(\exists xs' xss' xss''. xss = lappend (llist-of xss') (LCons (LCons x xs') xss'') \wedge$ 
   $xs = lappend xs' (lconcat xss'') \wedge set xss' \subseteq \{xs. lnull xs\})$ 
  (is ?lhs  $\longleftrightarrow$  ?rhs)
  proof
    assume ?rhs
    then obtain xs' xss' xss''
      where xss = lappend (llist-of xss') (LCons (LCons x xs') xss'')
      and xs = lappend xs' (lconcat xss'')

```

```

and set  $xss' \subseteq \{xs. lnull xs\}$  by blast
moreover from ⟨set  $xss' \subseteq \{xs. lnull xs\}lnull(lconcat(llist-of xss'))$  by simp
ultimately show ?lhs by(simp add: lappend-lnull1)
next
assume ?lhs
hence  $\neg lnull(lconcat xss)$  by simp
hence  $\neg lset xss \subseteq \{xs. lnull xs\}$  by simp
hence  $\neg lnull(lfilter(\lambda xs. \neg lnull xs) xss)$  by(auto)
then obtain  $y ys$  where  $yys: lfilter(\lambda xs. \neg lnull xs) xss = LCons y ys$ 
  unfolding not-lnull-conv by auto
from lfilter-eq-LConsD[OF this]
obtain  $us vs$  where  $xss: xss = lappend us (LCons y vs)$ 
  and  $lfinite us$ 
  and  $lset us \subseteq \{xs. lnull xs\} \cap lnull y$ 
  and  $ys: ys = lfilter(\lambda xs. \neg lnull xs) vs$  by blast
from ⟨lfinite us⟩ obtain  $us'$  where [simp]:  $us = llist-of us'$ 
  unfolding lfinite-eq-range-llist-of by blast
from ⟨lset us ⊆ {xs. lnull xs}⟩ have  $us: lnull(lconcat us)$ 
  unfolding lnull-lconcat .
from ⟨¬ lnull y⟩ obtain  $y' ys'$  where  $y: y = LCons y' ys'$ 
  unfolding not-lnull-conv by blast
from ⟨?lhs⟩ us have [simp]:  $y' = x xs = lappend ys' (lconcat vs)$ 
  unfolding xss y by(simp-all add: lappend-lnull1)
from ⟨lset us ⊆ {xs. lnull xs}⟩ ys show ?rhs unfolding xss y by simp blast
qed

```

```

lemma llength-lconcat-lfinite-conv-sum:
assumes lfinite xss
shows llength (lconcat xss) = ( $\sum i \mid enat i < llength xss. llength(lnth xss i)$ )
using assms
proof(induct)
  case lfinite-LNil thus ?case by simp
next
  case (lfinite-LConsI xss xs)
  have { $i. enat i \leq llength xss$ } = insert 0 {Suc i |  $i. enat i < llength xss$ }
    by(auto simp add: zero-enat-def[symmetric] Suc-ileq gr0-conv-Suc)
  also have ... = insert 0 (Suc ` { $i. enat i < llength xss$ }) by auto
  also have  $0 \notin Suc` \{i. enat i < llength xss\}$  by auto
  moreover from ⟨lfinite xss⟩ have finite { $i. enat i < llength xss$ }
    by(rule lfinite-finite-index)
  ultimately show ?case using lfinite-LConsI
    by(simp add: sum.reindex)
qed

```

```

lemma lconcat-lfilter-neq-LNil:
lconcat (lfilter(\lambda xs. \neg lnull xs) xss) = lconcat xss
by(induct xss)(simp-all add: lappend-lnull1)

```

```

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
   $\implies$  llist-all2 A (lconcat xss) (lconcat yss)
by(induct arbitrary: xss yss rule: lconcat-fixp-parallel-induct)(auto split: llist.split
intro: llist-all2-lappendI)

lemma llength-lconcat-eqI:
  fixes xss :: 'a llist llist and yss :: 'b llist llist
  assumes llist-all2 ( $\lambda$ xs ys. llength xs = llength ys) xss yss
  shows llength (lconcat xss) = llength (lconcat yss)
  apply(rule llist-all2-llengthD[where P= $\lambda$ - -. True])
  apply(rule llist-all2-lconcatI)
  apply(simp add: llist-all2-True[abs-def] assms)
  done

lemma lset-lconcat-lfinite:
   $\forall$  xs  $\in$  lset xss. lfinite xs  $\implies$  lset (lconcat xss) = ( $\bigcup$  xs  $\in$  lset xss. lset xs)
by(induction xss) auto

lemma lconcat-ltake:
  lconcat (ltake (enat n) xss) = ltake ( $\sum$  i < n. llength (lnth xss i)) (lconcat xss)
proof(induct n arbitrary: xss)
  case 0 thus ?case by(simp add: zero-enat-def[symmetric])
  next
  case (Suc n)
  show ?case
  proof(cases xss)
    case LNil thus ?thesis by simp
  next
  case (LCons xs xss')
  hence lconcat (ltake (enat (Suc n)) xss) = lappend xs (lconcat (ltake (enat n)
  xss')) by(simp add: eSuc-enat[symmetric])
  also have lconcat (ltake (enat n) xss') = ltake ( $\sum$  i < n. llength (lnth xss' i))
  (lconcat xss') by(rule Suc)
  also have lappend xs ... = ltake (llength xs + ( $\sum$  i < n. llength (lnth xss' i)))
  (lappend xs (lconcat xss')) by(cases llength xs)(simp-all add: ltake-plus-conv-lappend ltake-lappend1
  ltake-all ldropn-lappend2 lappend-inf lfinite-conv-llength-enat ldrop-enat)
  also have ( $\sum$  i < n. llength (lnth xss' i)) = ( $\sum$  i = 1..<Suc n. llength (lnth xss
  i)) by (rule sum.reindex-cong [symmetric, of Suc])
  (auto simp add: LCons image-iff less-Suc-eq-0-disj)
  also have llength xs + ... = ( $\sum$  i < Suc n. llength (lnth xss i))
  unfolding atLeast0LessThan[symmetric] LCons

```

```

    by(subst (2) sum.atLeast-Suc-lessThan) simp-all
    finally show ?thesis using LCons by simp
qed
qed

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'
using assms
proof(induct n arbitrary: xss)
case 0
hence ¬ lnull (lconcat xss) by auto
then obtain x xs where concat-xss: lconcat xss = LCons x xs
unfolding not-lnull-conv by blast
then obtain xs' xss' xss'' where xss: xss = lappend (llist-of xss') (LCons (LCons x xs') xss'')
and xs: xs = lappend xs' (lconcat xss'')
and LNil: set xss' ⊆ {xs. lnull xs}
unfolding lconcat-eq-LCons-conv by blast
from LNil have lnull (lconcat (llist-of xss')) by simp
moreover have [simp]: lnth xss (length xss') = LCons x xs'
unfolding xss by(simp add: lnth-lappend2)
ultimately have lnth (lconcat xss) 0 = lnth (lnth xss (length xss')) 0
using concat-xss xss by(simp)
moreover have enat 0 < llength (lnth xss (length xss'))
by(simp add: zero-enat-def[symmetric])
moreover have enat (length xss') < llength xss unfolding xss
by simp
moreover have (∑ i < length xss'. llength (lnth xss i)) = (∑ i < length xss'.
0)
proof(rule sum.cong)
show {..< length xss'} = {..< length xss'} by simp
next
fix i
assume i ∈ {..< length xss'}
hence xss' ! i ∈ set xss' unfolding in-set-conv-nth by blast
with LNil have xss' ! i = LNil by auto
moreover from ⟨i ∈ {..< length xss'}⟩ have lnth xss i = xss' ! i
unfolding xss by(simp add: lnth-lappend1)
ultimately show llength (lnth xss i) = 0 by simp
qed
hence enat 0 = (∑ i < length xss'. llength (lnth xss i)) + enat 0
by(simp add: zero-enat-def[symmetric])
ultimately show ?case by blast
next
case (Suc n)

```

```

from <enat (Suc n) < llength (lconcat xss)>
have ¬ lnull (lconcat xss) by auto
then obtain x xs where concat-xss: lconcat xss = LCons x xs
  unfolding not-lnull-conv by blast
then obtain xs' xss' xss'' where xss: xss = lappend (llist-of xss') (LCons (LCons
x xs') xss'')
  and xs: xs = lappend xs' (lconcat xss'')
  and LNil: set xss' ⊆ {xs. lnull xs}
  unfolding lconcat-eq-LCons-conv by blast
from LNil have concat-xss': lnull (lconcat (llist-of xss')) by simp
from xs have xs = lconcat (LCons xs' xss'') by simp
with concat-xss <enat (Suc n) < llength (lconcat xss)>
have enat n < llength (lconcat (LCons xs' xss''))
  by(simp add: Suc-ile-eq)
from Suc.hyps[OF this] obtain m n'
  where nth-n: lnth (lconcat (LCons xs' xss'')) n = lnth (lnth (LCons xs' xss''))
m) n'
  and n': enat n' < llength (lnth (LCons xs' xss'') m)
  and m': enat m < llength (LCons xs' xss'')
  and n-eq: enat n = (∑ i < m. llength (lnth (LCons xs' xss'') i)) + enat n'
  by blast
from n-eq obtain N where N: (∑ i < m. llength (lnth (LCons xs' xss'') i)) =
enat N
  and n: n = N + n'
  by(cases ∑ i < m. llength (lnth (LCons xs' xss'') i)) simp-all

{ fix i
assume i: i < length xss'
hence xss' ! i = LNil using LNil unfolding set-conv-nth by auto
hence lnth xss i = LNil using i unfolding xss
  by(simp add: lnth-lappend1) }
note lnth-prefix = this

show ?case
proof(cases m > 0)
  case True
  then obtain m' where [simp]: m = Suc m' by(cases m) auto
  have lnth (lconcat xss) (Suc n) = lnth (lnth xss (m + length xss')) n'
    using concat-xss' nth-n unfolding xss by(simp add: lnth-lappend2 lappend-lnull1)
  moreover have enat n' < llength (lnth xss (m + length xss'))
    using concat-xss' n' unfolding xss by(simp add: lnth-lappend2)
  moreover have enat (m + length xss') < llength xss
    using concat-xss' m' unfolding xss
    apply (simp add: Suc-ile-eq)
    apply (simp add: eSuc-enat[symmetric] eSuc-plus-1
      plus-enat-simps(1)[symmetric] del: plus-enat-simps(1))
  done

```

```

moreover have enat (m + length xss') < llength xss
  using m' unfolding xss
  apply(simp add: Suc-ile-eq)
  apply (simp add: eSuc-enat[symmetric] eSuc-plus-1
    plus-enat-simps(1)[symmetric] del: plus-enat-simps(1))
  done
moreover
{ have (∑ i < m + length xss'. llenth (lnth xss i)) =
  (∑ i < length xss'. llenth (lnth xss i)) +
  (∑ i = length xss'..+length xss'. llenth (lnth xss i))
  by(subst (1 2) atLeast0LessThan[symmetric])(subst sum.atLeastLessThan-concat,
  simp-all)
  also from lnth-prefix have (∑ i < length xss'. llenth (lnth xss i)) = 0 by
  simp
  also have {length xss'..+length xss'} = {0+length xss'..+length xss'}
  by auto
  also have (∑ i = 0 ... llenth (lnth xss (i + length xss'))) =
  (∑ i = 0... llenth (lnth xss (i + length xss'))))
  by(rule sum.shift-bounds-nat-ivl)
  also have ... = (∑ i = 0... llenth (lnth (LCons (LCons x xs') xss'') i))
  unfolding xss by(subst lnth-lappend2) simp+
  also have ... = eSuc (llenth xs') + (∑ i = Suc 0... llenth (lnth (LCons
  (LCons x xs') xss'') i))
  by(subst sum.atLeast-Suc-lessThan) simp-all
  also {
    fix i
    assume i ∈ {Suc 0..}
    then obtain i' where i = Suc i' by(cases i) auto
    hence llenth (lnth (LCons (LCons x xs') xss'') i) = llenth (lnth (LCons
    xs' xss'') i)
    by simp }
    hence (∑ i = Suc 0... llenth (lnth (LCons (LCons x xs') xss'') i)) =
    (∑ i = Suc 0... llenth (lnth (LCons xs' xss'') i)) by(simp)
    also have eSuc (llenth xs') + ... = 1 + (llenth (lnth (LCons xs' xss'') 0)
    + ...)
    by(simp add: eSuc-plus-1 ac-simps)
    also note sum.atLeast-Suc-lessThan[symmetric, OF ‹0 < m›]
    finally have enat (Suc n) = (∑ i < m + length xss'. llenth (lnth xss i)) +
    enat n'
    unfolding eSuc-enat[symmetric] n-eq by(simp add: eSuc-plus-1 ac-simps
    atLeast0LessThan) }
    ultimately show ?thesis by blast
next
case False
hence [simp]: m = 0 by auto
have lnth (lconcat xss) (Suc n) = lnth (lnth xss (length xss')) (Suc n')
  using concat-xss n-eq xs n'
  unfolding xss by(simp add: lnth-lappend1 lnth-lappend2)
moreover have enat (Suc n') < llenth (lnth xss (length xss'))

```

```

using concat-xss n' unfolding XSS by(simp add: lnth-lappend2 Suc-ile-eq)
moreover have enat (length XSS) < llength XSS unfolding XSS
  by simp
  moreover from lnth-prefix have (∑ i<length XSS'. llength (lnth XSS i)) = 0
by simp
hence enat (Suc n) = (∑ i<length XSS'. llength (lnth XSS i)) + enat (Suc n')
  using n-eq by simp
ultimately show ?thesis by blast
qed
qed

lemma lprefix-lconcatI:
XSS ⊑ YSS ==> lconcat XSS ⊑ lconcat YSS
by(rule monotoneD[OF monotone-lconcat])

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
using assms by(auto intro: lprefix-lnthD lprefix-lconcatI)

lemma lfinite-lconcat [simp]:
lfinite (lconcat XSS) ↔ lfinite (lfilter (λxs. ¬ lnull xs) XSS) ∧ (∀ xs ∈ lset XSS.
lfinite xs)
(is ?lhs ↔ ?rhs)
proof
assume ?lhs
thus ?rhs (is ?concl XSS)
proof(induct lconcat XSS arbitrary: XSS)
case [symmetric]: lfinite-LNil
moreover hence lnull (lfilter (λxs. ¬ lnull xs) XSS)
  by(auto simp add: lconcat-eq-LNil)
ultimately show ?case by(auto)
next
case (lfinite-LConsI xs x)
from ⟨LCons x xs = lconcat XSS⟩[symmetric]
obtain XSS' XSS'' XSS''' where XSS [simp]: XSS = lappend (llist-of XSS') (LCons (LCons x xs') XSS'')
  and xs [simp]: xs = lappend xs' (lconcat XSS'')
  and XSS'': set XSS'' ⊆ {xs. lnull xs} unfolding lconcat-eq-LCons-conv by blast
have xs = lconcat (LCons xs' XSS'') by simp
hence ?concl (LCons xs' XSS'') by(rule lfinite-LConsI)
thus ?case using ⟨lfinite xs' XSS'⟩ by(auto split: if-split-asm)
qed
next
assume ?rhs
then obtain lfinite (lfilter (λxs. ¬ lnull xs) XSS)
  and ∀ xs ∈ lset XSS. lfinite xs ..
thus ?lhs
proof(induct lfilter (λxs. ¬ lnull xs) XSS arbitrary: XSS)

```

```

case lfinite-LNil
from ⟨LNil = lfilter (λxs. ¬ lnull xs) xss⟩[symmetric]
have lset xss ⊆ {xs. lnull xs} unfolding lfilter-empty-conv by blast
hence lnull (lconcat xss) by(simp)
thus ?case by(simp)
next
case (lfinite-LConsI xs x)
from lfilter-eq-LConsD[OF ⟨LCons x xs = lfilter (λxs. ¬ lnull xs) xss⟩[symmetric]]
obtain xss' xss'' where xss [simp]: xss = lappend xss' (LCons x xss'')
and xss': lfinite xss' lset xss' ⊆ {xs. lnull xs}
and x: ¬ lnull x
and xs [simp]: xs = lfilter (λxs. ¬ lnull xs) xss'' by blast
moreover
from ⟨∀ xs ∈ lset xss. lfinite xs⟩ xss' have ∀ xs ∈ lset xss''. lfinite xs by auto
with xs have lfinite (lconcat xss'') by(rule lfinite-LConsI)
ultimately show ?case using ⟨∀ xs ∈ lset xss. lfinite xs⟩ by(simp)
qed
qed

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))
using assms by induct(simp-all add: list-of-lappend)

lemma lfilter-lconcat-lfinite:
∀ xs ∈ lset xss. lfinite xs
⇒ lfilter P (lconcat xss) = lconcat (lmap (lfilter P) xss)
by(induct xss) simp-all

lemma lconcat-repeat-LNil [simp]: lconcat (repeat LNil) = LNil
by(simp add: lconcat-eq-LNil)

lemma lconcat-lmap-singleton [simp]: lconcat (lmap (λx. LCons (f x) LNil) xs) =
lmap f xs
by(induct xs) simp-all

lemma lset-lconcat-subset: lset (lconcat xss) ⊆ (⋃ xs ∈ lset xss. lset xs)
by(induct xss)(auto dest: subsetD[OF lset-lappend])

lemma ldistinct-lconcat:
[ $\llbracket \text{ldistinct } \textit{xss}; \bigwedge \textit{ys}. \textit{ys} \in \textit{lset } \textit{xss} \implies \text{ldistinct } \textit{ys};$ 
 $\bigwedge \textit{ys} \textit{zs}. [\llbracket \textit{ys} \in \textit{lset } \textit{xss}; \textit{zs} \in \textit{lset } \textit{xss}; \textit{ys} \neq \textit{zs} \rrbracket \implies \textit{lset } \textit{ys} \cap \textit{lset } \textit{zs} = \{\}] \implies \text{ldistinct } (\text{lconcat } \textit{xss})$ 
apply(induction arbitrary: xss rule: lconcat.fixp-induct)
apply(auto simp add: ldistinct-lappend fun-ord-def split: llist.split)
apply(blast dest!: subsetD[OF lprefix-lsetD] subsetD[OF lset-lconcat-subset])
done

```

2.27 Sublist view of a lazy list: lnths

```

lemma lnths-empty [simp]:  $\text{lnths } xs \{ \} = LNil$ 
by(auto simp add: lnths-def lfilter-empty-conv)

lemma lnths-LNil [simp]:  $\text{lnths } LNil A = LNil$ 
by(simp add: lnths-def)

lemma lnths-LCons:
 $\text{lnths } (LCons x xs) A =$ 
 $(\text{if } 0 \in A \text{ then } LCons x (\text{lnths } xs \{ n. Suc n \in A \}) \text{ else } \text{lnths } xs \{ n. Suc n \in A \})$ 
proof -
  let ?it = iterates Suc
  let ?f =  $\lambda(x, y). (x, Suc y)$ 
  { fix n
    have lzip xs (?it (Suc n)) = lmap ?f (lzip xs (?it n))
    by(coinduction arbitrary: xs n)(auto)
    hence lmap fst (lfilter ( $\lambda(x, y). y \in A$ ) (lzip xs (?it (Suc n)))) =
      lmap fst (lfilter ( $\lambda(x, y). Suc y \in A$ ) (lzip xs (?it n)))
    by(simp add: lfilter-lmap o-def split-def llist.map-comp) }
  thus ?thesis
    by(auto simp add: lnths-def)(subst iterates, simp) +
  qed

lemma lset-lnths:
 $\text{lset } (\text{lnths } xs I) = \{ \text{lnth } xs i | i. \text{enat } i < \text{llength } xs \wedge i \in I \}$ 
apply(auto simp add: lnths-def lset-lzip)
apply(rule-tac x=(lnth xs i, i) in image-eqI)
apply auto
done

lemma lset-lnths-subset:  $\text{lset } (\text{lnths } xs I) \subseteq \text{lset } xs$ 
by(auto simp add: lset-lnths in-lset-conv-lnth)

lemma lnths-singleton [simp]:
 $\text{lnths } (LCons x LNil) A = (\text{if } 0 : A \text{ then } LCons x LNil \text{ else } LNil)$ 
by (simp add: lnths-LCons)

lemma lnths-upd-eq-ltake [simp]:
 $\text{lnths } xs \{ .. < n \} = \text{ltake } (\text{enat } n) xs$ 
apply(rule sym)
proof(induct n arbitrary: xs)
  case 0 thus ?case by(simp add: zero-enat-def[symmetric])
next
  case (Suc n)
  thus ?case
    by(cases xs)(simp-all add: eSuc-enat[symmetric] lnths-LCons lessThan-def)
qed

lemma lnths-llist-of [simp]:

```

```

lnths (llist-of xs) A = llist-of (nths xs A)
by(induct xs arbitrary: A)(simp-all add: lnths-LCons nths-Cons)

lemma llength-lnths-ile: llength (lnths xs A) ≤ llength xs
proof -
  have llength (lfilter (λ(x, y). y ∈ A) (lzip xs (iterates Suc 0))) ≤
    llength (lzip xs (iterates Suc 0))
  by(rule llength-lfilter-ile)
  thus ?thesis by(simp add: lnths-def)
qed

lemma lnths-lmap [simp]:
  lnths (lmap f xs) A = lmap f (lnths xs A)
by(simp add: lnths-def lzip-lmap1 llist.map-comp lfilter-lmap o-def split-def)

lemma lfilter-conv-lnths:
  lfilter P xs = lnths xs {n. enat n < llength xs ∧ P (lnth xs n)}
proof -
  have lnths xs {n. enat n < llength xs ∧ P (lnth xs n)} =
    lmap fst (lfilter (λ(x, y). enat y < llength xs ∧ P (lnth xs y)))
    (lzip xs (iterates Suc 0)))
  by(simp add: lnths-def)
  also have ∀(x, y)∈lset (lzip xs (iterates Suc 0)). enat y < llength xs ∧ x = lnth
  xs y
  by(auto simp add: lset-lzip)
  hence lfilter (λ(x, y). enat y < llength xs ∧ P (lnth xs y)) (lzip xs (iterates Suc
  0)) =
    lfilter (P ∘ fst) (lzip xs (iterates Suc 0))
  by -(rule lfilter-cong[OF refl], auto)
  also have lmap fst (lfilter (P ∘ fst) (lzip xs (iterates Suc 0))) =
    lfilter P (lmap fst (lzip xs (iterates Suc 0)))
  unfolding lfilter-lmap ..
  also have lmap fst (lzip xs (iterates Suc 0)) = xs
  by(simp add: lmap-fst-lzip-conv-ltake ltake-all)
  finally show ?thesis ..
qed

lemma ltake-iterates-Suc:
  ltake (enat n) (iterates Suc m) = llist-of [m..<n + m]
proof(induct n arbitrary: m)
  case 0 thus ?case by(simp add: zero-enat-def[symmetric])
next
  case (Suc n)
  have ltake (enat (Suc n)) (iterates Suc m) =
    LCons m (ltake (enat n) (iterates Suc (Suc m)))
  by(subst iterates)(simp add: eSuc-enat[symmetric])
  also note Suc
  also have LCons m (llist-of [Suc m..<n + Suc m]) = llist-of [m..<Suc n+m]
  unfolding llist-of.simps[symmetric]

```

```

  by(auto simp del: llist-of.simps simp add: upt-conv-Cons)
  finally show ?case .
qed

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 -
  let ?it = iterates Suc
  from assms have fin: lfinite xs by(rule llength-eq-enat-lfiniteD)
  have lnths (lappend xs ys) A =
    lmap fst (lfilter (λ(x, y). y ∈ A) (lzip (lappend xs ys) (?it 0)))
    by(simp add: lnths-def)
  also have ?it 0 = lappend (ltake (enat k) (?it 0)) (ldrop (enat k) (?it 0))
    by(simp only: lappend-ltake-ldrop)
  also note lzip-lappend
  also note lfilter-lappend-lfinite
  also note lmap-lappend-distrib
  also have lzip xs (ltake (enat k) (?it 0)) = lzip xs (?it 0)
    using len by(subst (1 2) lzip-conv-lzip-ltake-min-llength) simp
  also note lnths-def[symmetric]
  also have ldrop (enat k) (?it 0) = ?it k
    by(simp add: ldrop-iterates)
  also { fix n m
    have ?it (n + m) = lmap (λn. n + m) (?it n)
      by(coinduction arbitrary: n)(force)+ }
  from this[of 0 k] have ?it k = lmap (λn. n + k) (?it 0) by simp
  also note lzip-lmap2
  also note lfilter-lmap
  also note llist.map-comp
  also have fst ∘ (λ(x, y). (x, y + k)) = fst
    by(simp add: o-def split-def)
  also have (λ(x, y). y ∈ A) ∘ (λ(x, y). (x, y + k)) = (λ(x, y). y ∈ {n. n + k ∈ A})
    by(simp add: fun-eq-iff)
  also note lnths-def[symmetric]
  finally show ?thesis using len by simp
qed

lemma lnths-split:
  lnths xs A =
    lappend (lnths (ltake (enat n) xs) A) (lnths (ldropn n xs) {m. n + m ∈ A})
proof(cases enat n ≤ llength xs)
  case False thus ?thesis by(auto simp add: ltake-all ldropn-all)
next
  case True
  have xs = lappend (ltake (enat n) xs) (ldrop (enat n) xs)
    by(simp only: lappend-ltake-ldrop)

```

```

hence  $xs = lappend(ltake(enat n) xs)(ldropn n xs)$  by simp
hence  $lnths xs A = lnths(lappend(ltake(enat n) xs)(ldropn n xs)) A$ 
      by(simp)
also note  $lnths-lappend-lfinite[where k=n]$ 
finally show ?thesis using True by(simp add: min-def ac-simps)
qed

```

```

lemma  $lnths-cong$ :
assumes  $xs = ys$  and  $A: \bigwedge n. enat n < llengtys \implies n \in A \longleftrightarrow n \in B$ 
shows  $lnths xs A = lnths ys B$ 
proof -
have  $lfilter(\lambda(x, y). y \in A)(lzip ys (\text{iterates } Suc 0)) =$ 
       $lfilter(\lambda(x, y). y \in B)(lzip ys (\text{iterates } Suc 0))$ 
      by(rule lfilter-cong[OF refl])(auto simp add: lset-lzip A)
thus ?thesis unfolding  $\langle xs = ys \rangle$   $lnths-def$  by simp
qed

```

```

lemma  $lnths-insert$ :
assumes  $n: enat n < llengt xs$ 
shows  $lnths xs (insert n A) =$ 
       $lappend(lnths(ltake(enat n) xs) A) (LCCons(lnth xs n)$ 
       $(lnths(ldropn(Suc n) xs) \{m. Suc(n + m) \in A\}))$ 
proof -
have  $lnths xs (insert n A) =$ 
       $lappend(lnths(ltake(enat n) xs) (insert n A))$ 
       $(lnths(ldropn n xs) \{m. n + m \in (insert n A)\})$ 
      by(rule lnths-split)
also have  $lnths(ltake(enat n) xs) (insert n A) =$ 
       $lnths(ltake(enat n) xs) A$ 
      by(rule lnths-cong[OF refl]) simp
also { from n obtain X XS where  $ldropn n xs = LCCons X XS$ 
      by(cases ldropn n xs)(auto simp add: ldropn-eq-LNil)
moreover have  $lnth(ldropn n xs) 0 = lnth xs n$ 
      using n by(simp)
moreover have  $ltl(ldropn n xs) = ldropn(Suc n) xs$ 
      by(cases xs)(simp-all add: ltl-ldropn)
ultimately have  $ldropn n xs = LCCons(lnth xs n) (ldropn(Suc n) xs)$  by simp
hence  $lnths(ldropn n xs) \{m. n + m \in insert n A\} =$ 
       $LCCons(lnth xs n) (lnths(ldropn(Suc n) xs) \{m. Suc(n + m) \in A\})$ 
      by(simp add: lnths-LCons)
finally show ?thesis .
qed

```

```

lemma  $lfinite-lnths [simp]$ :
lfinite  $(lnths xs A) \longleftrightarrow lfinite xs \vee finite A$ 
proof
assume  $lfinite(lnths xs A)$ 
hence  $lfinite xs \vee$ 
       $finite \{n. enat n < llengt xs \wedge (\lambda(x, y). y \in A)(lnth(lzip xs (\text{iterates } Suc$ 

```

```

0)) n)}
  by(simp add: lnths-def lfinite-lfilter)
  also have {n. enat n < llength xs ∧ (λ(x, y). y ∈ A) (lnth (lzip xs (iterates Suc 0)) n)} =
    {n. enat n < llength xs ∧ n ∈ A} by(auto simp add: lnth-lzip)
  finally show lfinite xs ∨ finite A
    by(auto simp add: not-lfinite-llength elim: contrapos-np)
next
  assume lfinite xs ∨ finite A
  moreover
    have {n. enat n < llength xs ∧ (λ(x, y). y ∈ A) (lnth (lzip xs (iterates Suc 0)) n)} =
      {n. enat n < llength xs ∧ n ∈ A} by(auto simp add: lnth-lzip)
    ultimately show lfinite (lnths xs A)
      by(auto simp add: lnths-def lfinite-lfilter)
qed

```

2.28 lsum-list

context monoid-add **begin**

lemma lsum-list-0 [simp]: lsum-list (lmap (λ_. 0) xs) = 0
by(simp add: lsum-list-def)

lemma lsum-list-llist-of [simp]: lsum-list (llist-of xs) = sum-list xs
by(simp add: lsum-list-def)

lemma lsum-list-lappend: [lfinite xs; lfinite ys] ⇒ lsum-list (lappend xs ys) = lsum-list xs + lsum-list ys
by(simp add: lsum-list-def list-of-lappend)

lemma lsum-list-LNil [simp]: lsum-list LNil = 0
by(simp add: lsum-list-def)

lemma lsum-list-LCons [simp]: lfinite xs ⇒ lsum-list (LCons x xs) = x + lsum-list xs
by(simp add: lsum-list-def)

lemma lsum-list-inf [simp]: ¬ lfinite xs ⇒ lsum-list xs = 0
by(simp add: lsum-list-def)

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)
using assms
by(auto simp add: lsum-list-def intro: sum-list-mono)

2.29 Alternative view on 'a llist as datatype with constructors llist-of and inf-list

```

lemma lnull-inf-list [simp]:  $\neg \text{lnull}(\text{inf-list } f)$ 
by(simp add: inf-list-def)

lemma inf-list-neq-LNil:  $\text{inf-list } f \neq \text{LNil}$ 
using lnull-inf-list unfolding lnull-def .

lemmas LNil-neq-inf-list = inf-list-neq-LNil[symmetric]

lemma lhd-inf-list [simp]:  $\text{lhd}(\text{inf-list } f) = f \ 0$ 
by(simp add: inf-list-def)

lemma ltl-inf-list [simp]:  $\text{ltl}(\text{inf-list } f) = \text{inf-list}(\lambda n. f(\text{Suc } n))$ 
by(simp add: inf-list-def lmap-iterates[symmetric] llist.map-comp)

lemma inf-list-rec [code, nitpick-simp]:
 $\text{inf-list } f = \text{LCons}(f \ 0)(\text{inf-list}(\lambda n. f(\text{Suc } n)))$ 
by(rule llist.expand) simp-all

lemma lfinite-inf-list [iff]:  $\neg \text{lfinite}(\text{inf-list } f)$ 
by(simp add: inf-list-def)

lemma iterates-conv-inf-list:
 $\text{iterates } f a = \text{inf-list}(\lambda n. (f \wedge n) \ a)$ 
by(coinduction arbitrary: a)(auto simp add: funpow-swap1)

lemma inf-list-neq-list-of [simp]:
 $\text{llist-of } xs \neq \text{inf-list } f$ 
 $\text{inf-list } f \neq \text{llist-of } xs$ 
using lfinite-list-of[of xs] lfinite-inf-list[of f] by fastforce+

```

```

lemma lnth-inf-list [simp]:  $\text{lnth}(\text{inf-list } f) \ n = f \ n$ 
proof(induct n arbitrary: f)
  case 0 thus ?case by(subst inf-list-rec) simp
next
  case (Suc n)
  from Suc[of  $\lambda n. f(\text{Suc } n)$ ] show ?case
    by(subst inf-list-rec) simp
qed

```

```

lemma inf-list-lprefix [simp]:  $\text{inf-list } f \sqsubseteq xs \longleftrightarrow xs = \text{inf-list } f$ 
by(auto simp add: not-lfinite-lprefix-conv-eq)

lemma llength-inf-list [simp]:  $\text{llength}(\text{inf-list } f) = \infty$ 
by(simp add: inf-list-def)

lemma lset-inf-list [simp]:  $\text{lset}(\text{inf-list } f) = \text{range } f$ 

```

```

by(auto simp add: lset-conv-lnth)

lemma inf_llist-inj [simp]:
  inf_llist f = inf_llist g  $\longleftrightarrow$  f = g
unfolding inf_llist-def lmap-eq-lmap-conv_llist-all2
by(simp add: iterates-conv-inf_llist fun-eq-iff)

lemma inf_llist-lnth [simp]:  $\neg$  lfinite xs  $\implies$  inf_llist (lnth xs) = xs
by(coinduction arbitrary: xs)(auto simp add: lnth-0-conv-lhd fun-eq-iff lnth-ltl)

lemma llist-exhaust:
  obtains (llist-of) ys where xs = llist-of ys
  | (inf_llist) f where xs = inf_llist f
proof(cases lfinite xs)
  case True
  then obtain ys where xs = llist-of ys
  unfolding lfinite-eq-range_llist-of by auto
  thus ?thesis by(rule that)
next
  case False
  hence xs = inf_llist (lnth xs) by simp
  thus thesis by(rule that)
qed

lemma lappend-inf_llist [simp]: lappend (inf_llist f) xs = inf_llist f
by(simp add: lappend-inf)

lemma lmap-inf_llist [simp]:
  lmap f (inf_llist g) = inf_llist (f o g)
by(simp add: inf_llist-def llist.map-comp)

lemma ltake-enat-inf_llist [simp]:
  ltake (enat n) (inf_llist f) = llist-of (map f [0..<n])
by(simp add: inf_llist-def ltake-iterates-Suc)

lemma ldropn-inf_llist [simp]:
  ldropn n (inf_llist f) = inf_llist ( $\lambda m. f (m + n)$ )
unfolding inf_llist-def ldropn-lmap ldropn-iterates
by(simp add: iterates-conv-inf_llist o-def)

lemma ldrop-enat-inf_llist:
  ldrop (enat n) (inf_llist f) = inf_llist ( $\lambda m. f (m + n)$ )
proof(induct n arbitrary: f)
  case Suc thus ?case by(subst inf_llist-rec)(simp add: eSuc-enat[symmetric])
qed(simp add: zero-enat-def[symmetric])

lemma lzzip-inf_llist-inf_llist [simp]:
  lzzip (inf_llist f) (inf_llist g) = inf_llist ( $\lambda n. (f n, g n)$ )
by(coinduction arbitrary: f g) auto

```

```

lemma lzip-llist-of-inf-llist [simp]:
  lzip (llist-of xs) (inf-llist f) = llist-of (zip xs (map f [0..<length xs]))
proof(induct xs arbitrary: f)
  case Nil thus ?case by simp
next
  case (Cons x xs)
  have map f [0..<length (x # xs)] = f 0 # map (λn. f (Suc n)) [0..<length xs]
    by(simp add: upt-conv-Cons map-Suc-upt[symmetric] del: upt-Suc)
  with Cons[of λn. f (Suc n)]
  show ?case by(subst inf-llist-rec)(simp)
qed

lemma lzip-inf-llist-llist-of [simp]:
  lzip (inf-llist f) (llist-of xs) = llist-of (zip (map f [0..<length xs]) xs)
proof(induct xs arbitrary: f)
  case Nil thus ?case by simp
next
  case (Cons x xs)
  have map f [0..<length (x # xs)] = f 0 # map (λn. f (Suc n)) [0..<length xs]
    by(simp add: upt-conv-Cons map-Suc-upt[symmetric] del: upt-Suc)
  with Cons[of λn. f (Suc n)]
  show ?case by(subst inf-llist-rec)(simp)
qed

lemma llist-all2-inf-llist [simp]:
  llist-all2 P (inf-llist f) (inf-llist g)  $\longleftrightarrow$  ( $\forall n$ . P (f n) (g n))
by(simp add: llist-all2-conv-lzip)

lemma llist-all2-llist-of-inf-llist [simp]:
   $\neg$  llist-all2 P (llist-of xs) (inf-llist f)
by(simp add: llist-all2-conv-lzip)

lemma llist-all2-inf-llist-llist-of [simp]:
   $\neg$  llist-all2 P (inf-llist f) (llist-of xs)
by(simp add: llist-all2-conv-lzip)

lemma (in monoid-add) lsum-list-infllist: lsum-list (inf-llist f) = 0
by simp

```

2.30 Setup for lifting and transfer

2.30.1 Relator and predicate properties

abbreviation llist-all == pred-llist

2.30.2 Transfer rules for the Transfer package

context includes lifting-syntax
begin

```

lemma set1-pre-lolist-transfer [transfer-rule]:
  (rel-pre-lolist A B ==> rel-set A) set1-pre-lolist set1-pre-lolist
  by(auto simp add: rel-pre-lolist-def vimage2p-def rel-fun-def set1-pre-lolist-def rel-set-def
  collect-def sum-set-defs prod-set-defs elim: rel-sum.cases split: sum.split-asm)
lemma set2-pre-lolist-transfer [transfer-rule]:
  (rel-pre-lolist A B ==> rel-set B) set2-pre-lolist set2-pre-lolist
  by(auto simp add: rel-pre-lolist-def vimage2p-def rel-fun-def set2-pre-lolist-def rel-set-def
  collect-def sum-set-defs prod-set-defs elim: rel-sum.cases split: sum.split-asm)
lemma LNil-transfer [transfer-rule]: llist-all2 P LNil LNil
by simp
lemma LCons-transfer [transfer-rule]:
  (A ==> llist-all2 A ==> llist-all2 A) LCons LCons
  unfolding rel-fun-def by simp
lemma case-lolist-transfer [transfer-rule]:
  (B ==> (A ==> llist-all2 A ==> B) ==> llist-all2 A ==> B)
  case-lolist case-lolist
  unfolding rel-fun-def by (simp split: llist.split)
lemma unfold-lolist-transfer [transfer-rule]:
  ((A ==> (=) ==> (A ==> B) ==> (A ==> A) ==> A ==> llist-all2 B) unfold-lolist unfold-lolist
  proof(rule rel-funI)+
    fix IS-LNIL1 :: 'a  $\Rightarrow$  bool and IS-LNIL2 LHD1 LHD2 LTL1 LTL2 x y
    assume rel: (A ==> (=) IS-LNIL1 IS-LNIL2 (A ==> B) LHD1 LHD2
    (A ==> A) LTL1 LTL2)
    and A x y
    from <A x y>
    show llist-all2 B (unfold-lolist IS-LNIL1 LHD1 LTL1 x) (unfold-lolist IS-LNIL2
    LHD2 LTL2 y)
    apply(coinduction arbitrary: x y)
    using rel by(auto 4 4 elim: rel-funE)
  qed
lemma llist-corec-transfer [transfer-rule]:
  ((A ==> (=) ==> (A ==> B) ==> (A ==> (=)) ==> (A ==> llist-all2 B) ==> (A ==> A) ==> A ==> llist-all2 B) corec-lolist
  corec-lolist
  proof(rule rel-funI)+
    fix IS-LNIL1 :: 'a  $\Rightarrow$  bool and IS-LNIL2 LHD1 LHD2
    and STOP1 :: 'a  $\Rightarrow$  bool and STOP2 MORE1 MORE2 LTL1 LTL2 x y
    assume [transfer-rule]: (A ==> (=) IS-LNIL1 IS-LNIL2 (A ==> B) LHD1
    LHD2)
    (A ==> (=) STOP1 STOP2 (A ==> llist-all2 B) MORE1 MORE2 (A ==> A) LTL1 LTL2)

```

```

assume A x y
thus llist-all2 B (corec-llist IS-LNIL1 LHD1 STOP1 MORE1 LTL1 x) (corec-llist
IS-LNIL2 LHD2 STOP2 MORE2 LTL2 y)
proof(coinduction arbitrary: x y)
  case [transfer-rule]: (LNil x y)
    show ?case by simp transfer-prover
  next
    case (LCons x y)
      note [transfer-rule] = ‹A x y›
      have ?lhd by(simp add: LCons[simplified]) transfer-prover
      moreover
        have STOP1 x = STOP2 y llist-all2 B (MORE1 x) (MORE2 y) A (LTL1 x)
        (LTL2 y) by transfer-prover+
        hence ?ltl by(auto simp add: LCons[simplified])
        ultimately show ?case ..
    qed
qed

lemma ltl-transfer [transfer-rule]:
  (llist-all2 A ==> llist-all2 A) ltl ltl
  unfolding ltl-def[abs-def] by transfer-prover

lemma lset-transfer [transfer-rule]:
  (llist-all2 A ==> rel-set A) lset lset
  by (intro rel-funI rel-setI) (auto simp only: in-lset-conv-lnth llist-all2-conv-all-lnth
  Bex-def)

lemma lmap-transfer [transfer-rule]:
  ((A ==> B) ==> llist-all2 A ==> llist-all2 B) lmap lmap
  by(auto simp add: rel-fun-def llist-all2-lmap1 llist-all2-lmap2 elim: llist-all2-mono)

lemma lappend-transfer [transfer-rule]:
  (llist-all2 A ==> llist-all2 A ==> llist-all2 A) lappend lappend
  by(auto simp add: rel-fun-def intro: llist-all2-lappendI)

lemma iterates-transfer [transfer-rule]:
  ((A ==> A) ==> A ==> llist-all2 A) iterates iterates
  unfolding iterates-def split-def corec-llist-never-stop by transfer-prover

lemma lfinite-transfer [transfer-rule]:
  (llist-all2 A ==> (=)) lfinite lfinite
  by(auto dest: llist-all2-lfiniteD)

lemma llist-of-transfer [transfer-rule]:
  (list-all2 A ==> llist-all2 A) llist-of llist-of
  unfolding llist-of-def by transfer-prover

lemma llength-transfer [transfer-rule]:
  (llist-all2 A ==> (=)) llength llength

```

```

by(auto dest: llist-all2-llengthD)

lemma ltake-transfer [transfer-rule]:
  ((=) ==> llist-all2 A ==> llist-all2 A) ltake ltake
by(auto intro: llist-all2-ltakeI)

lemma ldropn-transfer [transfer-rule]:
  ((=) ==> llist-all2 A ==> llist-all2 A) ldropn ldropn
unfolding ldropn-def[abs-def] by transfer-prover

lemma ldrop-transfer [transfer-rule]:
  ((=) ==> llist-all2 A ==> llist-all2 A) ldrop ldrop
by(auto intro: llist-all2-ldropI)

lemma ltakeWhile-transfer [transfer-rule]:
  ((A ==> (=)) ==> llist-all2 A ==> llist-all2 A) ltakeWhile ltakeWhile
proof(rule rel-funI)+
  fix P :: 'a ⇒ bool and Q :: 'b ⇒ bool and xs :: 'a llist and ys :: 'b llist
  assume PQ: (A ==> (=)) P Q
  assume llist-all2 A xs ys
  thus llist-all2 A (ltakeWhile P xs) (ltakeWhile Q ys)
    apply(coinduction arbitrary: xs ys)
    using PQ by(auto 4 4 elim: rel-funE simp add: not-lnull-conv llist-all2-LCons2
      llist-all2-LCons1)
qed

lemma ldropWhile-transfer [transfer-rule]:
  ((A ==> (=)) ==> llist-all2 A ==> llist-all2 A) ldropWhile ldropWhile
unfolding ldropWhile-eq-ldrop[abs-def] by transfer-prover

lemma lzip-ltransfer [transfer-rule]:
  (llist-all2 A ==> llist-all2 B ==> llist-all2 (rel-prod A B)) lzip lzip
by(auto intro: llist-all2-lzipI)

lemma inf_llist-transfer [transfer-rule]:
  (((=) ==> A) ==> llist-all2 A) inf_llist inf_llist
unfolding inf_llist-def[abs-def] by transfer-prover

lemma lfilter-transfer [transfer-rule]:
  ((A ==> (=)) ==> llist-all2 A ==> llist-all2 A) lfilter lfilter
by(auto simp add: rel-fun-def intro: llist-all2-lfilterI)

lemma lconcat-transfer [transfer-rule]:
  (llist-all2 (llist-all2 A) ==> llist-all2 A) lconcat lconcat
by(auto intro: llist-all2-lconcatI)

lemma lnths-transfer [transfer-rule]:
  (llist-all2 A ==> (=) ==> llist-all2 A) lnths lnths
unfolding lnths-def[abs-def] by transfer-prover

```

```

lemma llist-all-transfer [transfer-rule]:
  ((A ==> (=)) ==> llist-all2 A ==> (=)) llist-all llist-all
  unfolding pred-llist-def[abs-def] by transfer-prover

lemma llist-all2-rsp:
  assumes r:  $\forall x y. R x y \rightarrow (\forall a b. R a b \rightarrow S x a = T y b)$ 
  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(cases llength x = llength a)
    case True
    thus ?thesis using l1 l2
      by(simp add: llist-all2-conv-all-lnth)(blast dest: r[rule-format])
  next
    case False
    with llist-all2-llengthD[OF l1] llist-all2-llengthD[OF l2]
    show ?thesis by(simp add: llist-all2-conv-all-lnth)
  qed

lemma llist-all2-transfer [transfer-rule]:
  ((R ==> R ==> (=)) ==> llist-all2 R ==> llist-all2 R ==> (=))
  llist-all2 llist-all2
  by (simp add: llist-all2-rsp rel-fun-def)

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]:  $xs < ys = lstrict-prefix\ xs\ ys$ 

instance

```

```

proof(intro-classes)
  fix xs ys zs :: 'a llist
  show (xs < ys) = (xs ≤ ys ∧ ¬ ys ≤ xs)
    unfolding less-llist-def less-eq-llist-def lstrict-prefix-def
    by(auto simp: lprefix-antisym)
  show xs ≤ xs unfolding less-eq-llist-def by(rule lprefix-refl)
  show [xs ≤ ys; ys ≤ zs] ⇒ xs ≤ zs
    unfolding less-eq-llist-def by(rule lprefix-trans)
  show [xs ≤ ys; ys ≤ xs] ⇒ xs = ys
    unfolding less-eq-llist-def by(rule lprefix-antisym)
qed

end

lemma le-llist-conv-lprefix [iff]: ( $\leq$ ) = lprefix
by(simp add: less-eq-llist-def fun-eq-iff)

lemma less-llist-conv-lstrict-prefix [iff]: (<) = lstrict-prefix
by(simp add: less-llist-def fun-eq-iff)

instantiation llist :: (type) order-bot begin

definition bot = LNil

instance
by(intro-classes)(simp add: bot-llist-def)

end

lemma llist-of-lprefix-llist-of [simp]:
  lprefix (llist-of xs) (llist-of ys) ⇔ xs ≤ ys
proof(induct xs arbitrary: ys)
  case (Cons x xs) thus ?case
    by(cases ys)(auto simp add: LCons-lprefix-conv)
qed simp

```

3.2 Prefix ordering as a lower semilattice

```

instantiation llist :: (type) semilattice-inf begin

definition [code del]:
  inf xs ys =
  unfold-llist (λ(xs, ys). xs ≠ LNil → ys ≠ LNil → lhd xs ≠ lhd ys)
  (lhd ∘ snd) (map-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)

```

```

unfolding inf-llist-def by simp-all

lemma llist-inf-eq-LNil [simp]:
  lnull (inf xs ys)  $\longleftrightarrow$  (xs  $\neq$  LNil  $\longrightarrow$  ys  $\neq$  LNil  $\longrightarrow$  lhd xs  $\neq$  lhd ys)
by(simp add: inf-llist-def)

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)
using assms by(simp-all add: inf-llist-def)

instance
proof
  fix xs ys zs :: 'a llist
  show inf xs ys  $\leq$  xs unfolding le-llist-conv-lprefix
    by(coinduction arbitrary: xs ys) auto

  show inf xs ys  $\leq$  ys unfolding le-llist-conv-lprefix
    by(coinduction arbitrary: xs ys) auto

  assume xs  $\leq$  ys xs  $\leq$  zs
  thus xs  $\leq$  inf ys zs unfolding le-llist-conv-lprefix
    proof(coinduction arbitrary: xs ys zs)
      case (lprefix xs ys zs)
      thus ?case by(cases xs)(auto 4 4 simp add: LCons-lprefix-conv)
    qed
  qed

end

lemma llength-inf [simp]: llength (inf xs ys) = llcp xs ys
by(coinduction arbitrary: xs ys rule: enat-coinduct)(auto simp add: llcp-eq-0-iff
epred-llength epred-llcp)

instantiation llist :: (type) ccpo
begin

definition Sup A = lSup A

instance
  by intro-classes
  (auto simp: Sup-llist-def less-eq-llist-def[abs-def] intro!: llist.lub-upper llist.lub-least)

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:  $P x \implies \text{eq-onp } P x x$ 
by(simp add: eq-onp-def)

primcorec unfold-stream ::  $('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'b \text{ stream where}$ 
   $\text{unfold-stream } g1\ g2\ a = g1\ a \# \# \text{unfold-stream } g1\ g2\ (g2\ a)$ 

The following setup should be done by the BNF package.

congruence rule
declare stream.map-cong [cong]

lemmas about generated constants

lemma eq-SConsD:  $xs = SCons\ y\ ys \implies shd\ xs = y \wedge stl\ xs = ys$ 
by auto

declare stream.map-ident[simp]

lemma smap-eq-SCons-conv:
   $smap\ f\ xs = y \# \# \text{ys} \longleftrightarrow$ 
   $(\exists x\ xs'.\ xs = x \# \# \text{xs}' \wedge y = f\ x \wedge \text{ys} = \text{smap}\ f\ \text{xs}')$ 
by(cases xs)(auto)

lemma smap-unfold-stream:
   $\text{smap}\ f\ (\text{unfold-stream SHD STL } b) = \text{unfold-stream } (f \circ \text{SHD})\ \text{STL } b$ 
by(coinduction arbitrary: b) auto

lemma smap-corec-stream:
   $\text{smap}\ f\ (\text{corec-stream SHD endORmore STL-end STL-more } b) =$ 
   $\text{corec-stream } (f \circ \text{SHD})\ \text{endORmore } (\text{smap}\ f \circ \text{STL-end})\ \text{STL-more } b$ 
by(coinduction arbitrary: b rule: stream.coinduct-strong) auto

lemma unfold-stream-ltl-unroll:
   $\text{unfold-stream SHD STL } (STL\ b) = \text{unfold-stream } (\text{SHD} \circ \text{STL})\ \text{STL } b$ 
by(coinduction arbitrary: b) auto

lemma unfold-stream-eq-SCons [simp]:
   $\text{unfold-stream SHD STL } b = x \# \# \text{xs} \longleftrightarrow$ 
   $x = \text{SHD } b \wedge \text{xs} = \text{unfold-stream SHD STL } (STL\ b)$ 
by(subst unfold-stream.ctr) auto

```

lemma *unfold-stream-id* [*simp*]: *unfold-stream shd stl xs = xs*
by(*coinduction arbitrary*: *xs*) *simp-all*

lemma *sset-neq-empty* [*simp*]: *sset xs ≠ {}*
by(*cases xs*) *simp-all*

declare *stream.setsel(1)*[*simp*]

lemma *sset-stl*: *sset (stl xs) ⊆ sset xs*
by(*cases xs*) *auto*

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)
by(*coinduction arbitrary*: *a b*) *auto*

lemma *szip-smap1*: *szip (smap f xs) ys = smap (apfst f) (szip xs ys)*
by(*coinduction arbitrary*: *xs ys*) *auto*

lemma *szip-smap2*: *szip xs (smap g ys) = smap (apsnd g) (szip xs ys)*
by(*coinduction arbitrary*: *xs ys*) *auto*

lemma *szip-smap* [*simp*]: *szip (smap f xs) (smap g ys) = smap (map-prod f g) (szip xs ys)*
by(*coinduction arbitrary*: *xs ys*) *auto*

lemma *smap-fst-szip* [*simp*]: *smap fst (szip xs ys) = xs*
by(*coinduction arbitrary*: *xs ys*) *auto*

lemma *smap-snd-szip* [*simp*]: *smap snd (szip xs ys) = ys*
by(*coinduction arbitrary*: *xs ys*) *auto*

lemma *snth-shift*: *snth (shift xs ys) n = (if n < length xs then xs ! n else snth ys (n - length xs))*
by *simp*

declare *szip-unfold* [*simp, nitpick-simp*]

lemma *szip-shift*:

length xs = length us
 $\implies \text{szip} (\text{xs} @\text{--} \text{ys}) (\text{us} @\text{--} \text{zs}) = \text{zip} \text{ xs us} @\text{--} \text{szip} \text{ ys zs}$
by(*induct xs arbitrary*: *us*)(*auto simp add: Suc-length-conv*)

4.2 Link 'a stream to 'a llist

definition *llist-of-stream* :: *'a stream ⇒ 'a llist*

```

where llist-of-stream = unfold-llist ( $\lambda\_. \text{False}$ ) shd stl

definition stream-of-llist :: 'a llist  $\Rightarrow$  'a stream
where stream-of-llist = unfold-stream lhd ltl

lemma lnull-llist-of-stream [simp]:  $\neg \text{lnull} (\text{llist-of-stream } xs)$ 
by(simp add: llist-of-stream-def)

lemma ltl-llist-of-stream [simp]: ltl (llist-of-stream xs) = llist-of-stream (stl xs)
by(simp add: llist-of-stream-def)

lemma stl-stream-of-llist [simp]: stl (stream-of-llist xs) = stream-of-llist (ltl xs)
by(simp add: stream-of-llist-def)

lemma shd-stream-of-llist [simp]: shd (stream-of-llist xs) = lhd xs
by(simp add: stream-of-llist-def)

lemma lhd-llist-of-stream [simp]: lhd (llist-of-stream xs) = shd xs
by(simp add: llist-of-stream-def)

lemma stream-of-llist-llist-of-stream [simp]:
  stream-of-llist (llist-of-stream xs) = xs
by(coinduction arbitrary: xs) simp-all

lemma llist-of-stream-stream-of-llist [simp]:
   $\neg \text{lfinite } xs \implies \text{llist-of-stream} (\text{stream-of-llist } xs) = xs$ 
by(coinduction arbitrary: xs) auto

lemma lfinite-llist-of-stream [simp]:  $\neg \text{lfinite} (\text{llist-of-stream } xs)$ 
proof
  assume lfinite (llist-of-stream xs)
  thus False
    by(induct llist-of-stream xs arbitrary: xs rule: lfinite-induct) auto
qed

lemma stream-from-llist: type-definition llist-of-stream stream-of-llist {xs.  $\neg \text{lfinite } xs$ }
by(unfold-locales) simp-all

interpretation stream: type-definition llist-of-stream stream-of-llist {xs.  $\neg \text{lfinite } xs$ }
by(fact stream-from-llist)

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 .

lemma cr-streamI:  $\neg \text{lfinite } xs \implies \text{cr-stream } xs (\text{stream-of-llist } xs)$ 
by(simp add: cr-stream-def Abs-stream-inverse)

lemma llist-of-stream-unfold-stream [simp]:
  llist-of-stream (unfold-stream SHD STL x) = unfold-llist ( $\lambda \cdot. \text{False}$ ) SHD STL x
by(coinduction arbitrary: x) auto

lemma llist-of-stream-corec-stream [simp]:
  llist-of-stream (corec-stream SHD endORmore STL-more STL-end x) =
    corec-llist ( $\lambda \cdot. \text{False}$ ) SHD endORmore (llist-of-stream  $\circ$  STL-more) STL-end x
by(coinduction arbitrary: x rule: llist.coinduct-strong) auto

lemma LCons-llist-of-stream [simp]: LCons x (llist-of-stream xs) = llist-of-stream
(x ## xs)
by(rule sym)(simp add: llist-of-stream-def)

lemma lmap-llist-of-stream [simp]:
  lmap f (llist-of-stream xs) = llist-of-stream (smap f xs)
by(coinduction arbitrary: xs) auto

lemma lset-llist-of-stream [simp]: lset (llist-of-stream xs) = sset xs (is ?lhs = ?rhs)
proof(intro set-eqI iffI)
  fix x
  assume x ∈ ?lhs
  thus x ∈ ?rhs
    by(induct llist-of-stream xs arbitrary: xs rule: llist-set-induct)
      (auto dest: stream.setsel(2))
next
  fix x
  assume x ∈ ?rhs
  thus x ∈ ?lhs
    proof(induct)
      case (shd xs)
        thus ?case using llist.setsel(1)[of llist-of-stream xs] by simp
next
  case stl
  thus ?case
    by(auto simp add: ltl-llist-of-stream[symmetric] simp del: ltl-llist-of-stream
dest: in-lset-ltlD)
  qed
qed

lemma lnth-list-of-stream [simp]:

```

```

lnth (llist-of-stream xs) = snth xs
proof
  fix n
  show lnth (llist-of-stream xs) n = snth xs n
    by(induction n arbitrary: xs)(simp-all add: lnth-0-conv-lhd lnth-ltl[symmetric])
qed

lemma llist-of-stream-siterates [simp]: llist-of-stream (siterate f x) = iterates f x
by(coinduction arbitrary: x) auto

lemma lappend-llist-of-stream-conv-shift [simp]:
  lappend (llist-of xs) (llist-of-stream ys) = llist-of-stream (xs @- ys)
by(induct xs) simp-all

lemma lzip-llist-of-stream [simp]:
  lzip (llist-of-stream xs) (llist-of-stream ys) = llist-of-stream (szip xs ys)
by(coinduction arbitrary: xs ys) auto

context includes lifting-syntax
begin

lemma lmap-infinite-transfer [transfer-rule]:
  ((=) ==> eq-onp (λxs. ¬ lfinite xs) ==> eq-onp (λxs. ¬ lfinite xs)) lmap
  lmap
by(simp add: rel-fun-def eq-onp-def)

lemma lset-infinite-transfer [transfer-rule]:
  (eq-onp (λxs. ¬ lfinite xs) ==> (=)) lset lset
by(simp add: rel-fun-def eq-onp-def)

lemma unfold-stream-transfer [transfer-rule]:
  ((=) ==> (=) ==> (=) ==> pcr-stream (=)) (unfold-llist (λ-. False))
  unfold-stream
by(auto simp add: stream.pcr-cr-eq cr-stream-def intro!: rel-funI)

lemma corec-stream-transfer [transfer-rule]:
  ((=) ==> (=) ==> ((=) ==> pcr-stream (=)) ==> (=) ==> (=)
  ==> pcr-stream (=))
  (corec-llist (λ-. False)) corec-stream
apply(auto intro!: rel-funI simp add: cr-stream-def stream.pcr-cr-eq)
apply(rule fun-cong) back
apply(rule-tac x=yc in fun-cong)
apply(rule-tac x=xb in arg-cong)
apply(auto elim: rel-funE)
done

lemma shd-transfer [transfer-rule]: (pcr-stream A ==> A) lhd shd
by(auto simp add: pcr-stream-def cr-stream-def intro!: rel-funI relcomppI)(frule
  llist-all2-lhdD, auto dest: llist-all2-lnullD)

```

```

lemma stl-transfer [transfer-rule]: (pcr-stream A ==> pcr-stream A) ltl stl
by(auto simp add: pcr-stream-def cr-stream-def intro!: rel-funI relcomppI dest: llist-all2-ltlI)

lemma llist-of-stream-transfer [transfer-rule]: (pcr-stream (=) ==> (=)) id llist-of-stream
by(simp add: rel-fun-def stream.pcr-cr-eq cr-stream-def)

lemma stream-of-list-transfer [transfer-rule]:
  (eq-onp ( $\lambda xs. \neg lfinite\ xs$ ) ==> pcr-stream (=)) ( $\lambda xs. xs$ ) stream-of-list
by(simp add: eq-onp-def rel-fun-def stream.pcr-cr-eq cr-stream-def)

lemma SCons-transfer [transfer-rule]:
  (A ==> pcr-stream A ==> pcr-stream A) LCons (##)
by(auto simp add: cr-stream-def pcr-stream-def intro!: rel-funI relcomppI intro: llist-all2-expand)

lemma sset-transfer [transfer-rule]: (pcr-stream A ==> rel-set A) lset sset
by(auto 4 3 simp add: pcr-stream-def cr-stream-def intro!: rel-funI relcomppI rel-setI dest: llist-all2-lsetD1 llist-all2-lsetD2)

lemma smap-transfer [transfer-rule]:
  ((A ==> B) ==> pcr-stream A ==> pcr-stream B) lmap smap
by(auto simp add: cr-stream-def pcr-stream-def intro!: rel-funI relcomppI dest: lmap-transfer[THEN rel-funD] elim: rel-funD)

lemma snth-transfer [transfer-rule]: (pcr-stream (=) ==> (=)) lnth snth
by(rule rel-funI)(clarsimp simp add: stream.pcr-cr-eq cr-stream-def fun-eq-iff)

lemma siterate-transfer [transfer-rule]:
  ((=) ==> (=) ==> pcr-stream (=)) iterates siterate
by(rule rel-funI)+(clarsimp simp add: stream.pcr-cr-eq cr-stream-def)

context
  fixes xs
  assumes inf:  $\neg lfinite\ xs$ 
  notes [transfer-rule] = eq-onpI[where P=λxs. ¬ lfinite xs, OF inf]
  begin

    lemma smap-stream-of-list [simp]:
      shows smap f (stream-of-list xs) = stream-of-list (lmap f xs)
      by transfer simp

    lemma sset-stream-of-list [simp]:
      assumes  $\neg lfinite\ xs$ 
      shows sset (stream-of-list xs) = lset xs
      by transfer simp

  end

```

```

lemma llist-all2-llist-of-stream [simp]:
  llist-all2 P (llist-of-stream xs) (llist-of-stream ys) = stream-all2 P xs ys
apply(cases xs ys rule: stream.Abs-cases[case-product stream.Abs-cases])
apply(simp add: llist-all2-def stream-all2-def)
apply(safe elim!: GrpE)
apply(rule-tac b=stream-of-lolist b in relcomppI; auto intro!: GrpI)
apply(rule-tac b=llist-of-stream b in relcomppI; auto intro!: GrpI)
done

lemma stream-all2-transfer [transfer-rule]:
  ((=) ==> pcr-stream (=) ==> pcr-stream (=) ==> (=)) llist-all2 stream-all2
by(simp add: rel-fun-def stream.pcr-cr-eq cr-stream-def)

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
using assms
apply transfer
apply(rule-tac X=λxs ys. ¬ lfinite xs ∧ ¬ lfinite ys ∧ X xs ys in llist-all2-coinduct)
apply auto
done

lemma shift-transfer [transfer-rule]:
  ((=) ==> pcr-stream (=) ==> pcr-stream (=)) (lappend ∘ llist-of) shift
by(clarify simp add: rel-fun-def stream.pcr-cr-eq cr-stream-def)

lemma szip-transfer [transfer-rule]:
  (pcr-stream (=) ==> pcr-stream (=) ==> pcr-stream (=)) lzip szip
by(simp add: stream.pcr-cr-eq cr-stream-def rel-fun-def)

```

4.3 Link 'a stream with nat ⇒ 'a

lift-definition of-seq :: (nat ⇒ 'a) ⇒ 'a stream is inf-lolist by simp

```

lemma of-seq-rec [code]: of-seq f = f 0 # of-seq (f ∘ Suc)
by transfer (subst inf-lolist-rec, simp add: o-def)

```

```

lemma snth-of-seq [simp]: snth (of-seq f) = f
by transfer (simp add: fun-eq-iff)

```

```

lemma snth-SCons: snth (x # xs) n = (case n of 0 ⇒ x | Suc n' ⇒ snth xs n')
by(simp split: nat.split)

```

```

lemma snth-SCons-simps [simp]:
shows snth-SCons-0: (x # xs) !! 0 = x
and snth-SCons-Suc: (x # xs) !! Suc n = xs !! n
by(simp-all add: snth-SCons)

```

```

lemma of-seq-snth [simp]: of-seq (snth xs) = xs
by transfer simp

lemma shd-of-seq [simp]: shd (of-seq f) = f 0
by transfer simp

lemma stl-of-seq [simp]: stl (of-seq f) = of-seq ( $\lambda n. f (Suc n)$ )
by transfer simp

lemma sset-of-seq [simp]: sset (of-seq f) = range f
by transfer simp

lemma smap-of-seq [simp]: smap f (of-seq g) = of-seq (f  $\circ$  g)
by transfer simp
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)
by transfer (rule lmap-iterates)

lemma siterate-smap: siterate f x = x ## (smap f (siterate f x))
by transfer (rule iterates-lmap)

lemma siterate-conv-of-seq: siterate f a = of-seq ( $\lambda n. (f \wedge\!\! n) a$ )
by transfer (rule iterates-conv-inf-llist)

lemma sconst-conv-of-seq: sconst a = of-seq ( $\lambda -. a$ )
by(simp add: siterate-conv-of-seq)

lemma szip-sconst1 [simp]: szip (sconst a) xs = smap (Pair a) xs
by(coinduction arbitrary: xs) auto

lemma szip-sconst2 [simp]: szip xs (sconst b) = smap ( $\lambda x. (x, b)$ ) xs
by(coinduction arbitrary: xs) auto

end

```

4.5 Counting elements

```

partial-function (lfp) scount :: ('s stream  $\Rightarrow$  bool)  $\Rightarrow$  's stream  $\Rightarrow$  enat where
  scount P  $\omega$  = (if P  $\omega$  then eSuc (scount P (stl  $\omega$ )) else scount P (stl  $\omega$ ))

lemma scount-simps:
  P  $\omega \implies$  scount P  $\omega$  = eSuc (scount P (stl  $\omega$ ))
   $\neg P \omega \implies$  scount P  $\omega$  = scount P (stl  $\omega$ )
  using scount.simps[of P  $\omega$ ] by auto

```

```

lemma scount-eq-0I: alw (not P) ω  $\implies$  scount P ω = 0
by (induct arbitrary: ω rule: scount.fixp-induct)
  (auto simp: bot-enat-def intro!: admissible-all admissible-imp admissible-eq-mcontI
    mcont-const)

lemma scount-eq-0D: scount P ω = 0  $\implies$  alw (not P) ω
proof (induction rule: alw.coinduct)
  case (alw ω) with scount.simps[of P ω] show ?case
    by (simp split: if-split-asm)
qed

lemma scount-eq-0-iff: scount P ω = 0  $\longleftrightarrow$  alw (not P) ω
by (metis scount-eq-0D scount-eq-0I)

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)}
  using assms
proof (induction ω)
  case (step ω)
    have eq: {i. P (sdrop i ω)} = (if P ω then {0} else {})  $\cup$  (Suc ` {i. P (sdrop i (stl ω))})
    apply (intro set-eqI)
    apply (case-tac x)
    apply (auto simp: image-iff)
    done
  { case 1 show ?case
    using step unfolding eq by (auto simp: scount-simps card-image zero-notin-Suc-image
      eSuc-enat) }
  { case 2 show ?case
    using step unfolding eq by auto }
next
  case (base ω)
  then have [simp]: {i. P (sdrop i ω)} = {}
  by (simp add: not-HLD alw-iff-sdrop)
  { case 1 show ?case using base by (simp add: scount-eq-0I enat-0) }
  { case 2 show ?case by simp }
qed

lemma scount-finite: ev (alw (not P)) ω  $\implies$  scount P ω < ∞
using scount-eq-card[of P ω] by auto

lemma scount-infinite:
  alw (ev P) ω  $\implies$  scount P ω = ∞
proof (coinduction arbitrary: ω rule: enat-coinduct)
  case (Eq-enat ω)
  then have ev P ω alw (ev P) ω

```

```

    by auto
  then show ?case
    by (induction rule: ev-induct-strong) (auto simp add: scount-simps)
qed

lemma scount-infinite-iff: scount P ω = ∞ ↔ alw (ev P) ω
  by (metis enat-ord-simps(4) not-alw-not scount-finite scount-infinite)

lemma scount-eq:
  scount P ω = (if alw (ev P) ω then ∞ else enat (card {i. P (sdrop i ω)}))
  by (auto simp: scount-infinite-iff scount-eq-card not-alw-iff not-ev-iff)

```

4.6 First index of an element

```

partial-function (gfp) sfirrst :: ('s stream ⇒ bool) ⇒ 's stream ⇒ enat where
  sfirrst P ω = (if P ω then 0 else eSuc (sfirrst P (stl ω)))

```

```

lemma sfirrst-eq-0: sfirrst P ω = 0 ↔ P ω
  by (subst sfirrst.simps) auto

lemma sfirrst-0[simp]: P ω ⇒ sfirrst P ω = 0
  by (subst sfirrst.simps) auto

lemma sfirrst-eSuc[simp]: ¬ P ω ⇒ sfirrst P ω = eSuc (sfirrst P (stl ω))
  by (subst sfirrst.simps) auto

lemma less-sfirrstD:
  fixes n :: nat
  assumes enat n < sfirrst P ω shows ¬ P (sdrop n ω)
  using assms
proof (induction n arbitrary: ω)
  case (Suc n) then show ?case
    by (auto simp: sfirrst.simps[of - ω] eSuc-enat[symmetric] split: if-split-asm)
qed (simp add: enat-0 sfirrst-eq-0)

lemma sfirrst-finite: sfirrst P ω < ∞ ↔ ev P ω
proof
  assume sfirrst P ω < ∞
  then obtain n where sfirrst P ω = enat n by auto
  then show ev P ω
  proof (induction n arbitrary: ω)
    case (Suc n) then show ?case
      by (auto simp add: eSuc-enat[symmetric] sfirrst.simps[of P ω] split: if-split-asm)
  qed (auto simp add: enat-0 sfirrst-eq-0)
next
  assume ev P ω then show sfirrst P ω < ∞
    by (induction rule: ev-induct-strong) (auto simp: eSuc-enat)
qed

```

```

lemma sfirst-Stream: sfirst P (s ## x) = (if P (s ## x) then 0 else eSuc (sfirst P x))
  by (subst sfirst.simps) (simp add: HLD-iff)

lemma less-sfirst-iff: (not P until (alw P)) ω ⇒ enat n < sfirst P ω ↔ ¬ P (sdrop n ω)
  proof (induction n arbitrary: ω)
    case 0 then show ?case
      by (simp add: enat-0 sfirst-eq-0 HLD-iff)
  next
    case (Suc n)
    from Suc.preds have **: P ω ⇒ P (stl ω)
      by (auto elim: UNTIL.cases)
    have *: ¬ P (sdrop n (stl ω)) ↔ enat n < sfirst P (stl ω)
      using Suc.preds by (intro Suc.IH[symmetric]) (auto intro: UNTIL.intros elim: UNTIL.cases)
    show ?case
      unfolding sdrop.simps * by (cases P ω) (simp-all add: ** eSuc-enat[symmetric])
  qed

lemma sfirst-eq-Inf: sfirst P ω = Inf {enat i | i. P (sdrop i ω)}
  proof (rule antisym)
    show sfirst P ω ≤ ⋀ {enat i | i. P (sdrop i ω)}
    proof (safe intro!: Inf-greatest)
      fix ω i assume P (sdrop i ω) then show sfirst P ω ≤ enat i
      proof (induction i arbitrary: ω)
        case (Suc i) then show ?case
          by (auto simp add: sfirst.simps[of P ω] eSuc-enat[symmetric])
      qed auto
    qed
    show ⋀ {enat i | i. P (sdrop i ω)} ≤ sfirst P ω
    proof (induction arbitrary: ω rule: sfirst.fixp-induct)
      case (3 f)
      have {enat i | i. P (sdrop i ω)} = (if P ω then {0} else {}) ∪ eSuc ‘ {enat i | i. P (sdrop i (stl ω))}
      apply (intro set-eqI)
      apply (case-tac x rule: enat-coexhaust)
      apply (auto simp add: enat-0-iff image-iff eSuc-enat-iff)
      done
      with 3[of stl ω] show ?case
        by (auto simp: inf.absorb1 eSuc-Inf[symmetric])
    qed simp-all
  qed

```

lemma sfirst-eq-enat-iff: sfirst P ω = enat n ↔ ev-at P n ω
 by (induction n arbitrary: ω)
 (simp-all add: eSuc-enat[symmetric] sfirst.simps enat-0)

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)
by(simp add: stakeWhile-def LCons-llist-of-stream[symmetric] del: LCons-llist-of-stream)

lemma lnull-stakeWhile [simp]: lnull (stakeWhile P xs) ↔ ¬ P (shd xs)
by(simp add: stakeWhile-def)

lemma lhd-stakeWhile [simp]: P (shd xs) ⇒ lhd (stakeWhile P xs) = shd xs
by(simp add: stakeWhile-def)

lemma ltl-stakeWhile [simp]:
  ltl (stakeWhile P xs) = (if P (shd xs) then stakeWhile P (stl xs) else LNil)
by(simp add: stakeWhile-def)

lemma stakeWhile-K-False [simp]: stakeWhile (λ_. False) xs = LNil
by(simp add: stakeWhile-def)

lemma stakeWhile-K-True [simp]: stakeWhile (λ_. True) xs = llist-of-stream xs
by(simp add: stakeWhile-def)

lemma stakeWhile-smap: stakeWhile P (smap f xs) = lmap f (stakeWhile (P ∘ f) xs)
by(simp add: stakeWhile-def ltakeWhile-lmap[symmetric] del: o-apply)

lemma lfinite-stakeWhile [simp]: lfinite (stakeWhile P xs) ↔ (∃ x ∈ sset xs. ¬ P x)
by(simp add: stakeWhile-def lfinite-ltakeWhile)

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 constructors *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) = (forall x y. p = (x, y) → R x)

```

```

by(cases p) simp

lemma split-fst-asm: R (fst p)  $\longleftrightarrow$  ( $\neg (\exists x y. p = (x, y) \wedge \neg R x)$ )
by(cases p) simp

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  $\Rightarrow$  '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
apply(rule ext)
apply(subst terminal0.simps)
apply(case-tac x)
apply(simp-all add: terminal-def)
done

lemmas terminal-TNil [code, nitpick-simp] = tllist.sel(1)

lemma terminal-TCons [simp, code, nitpick-simp]: terminal (TCons x xs) = terminal xs
by simp

declare tllist.sel(2) [simp del]

primcorec unfold-tllist :: ('a  $\Rightarrow$  bool)  $\Rightarrow$  ('a  $\Rightarrow$  'b)  $\Rightarrow$  ('a  $\Rightarrow$  'c)  $\Rightarrow$  ('a  $\Rightarrow$  'a)  $\Rightarrow$  ('a  $\Rightarrow$  ('c, 'b) tllist where
  p a  $\Longrightarrow$  unfold-tllist p g1 g21 g22 a = TNil (g1 a) |
  -  $\Longrightarrow$  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]
oops
```

```
lemma TCons x xs = TCons x xs
quickcheck[narrowing, expect=no-counterexample]
oops
```

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))
  by(simp)
```

```
lemma is-TNil-ttl [simp]: is-TNil xs ==> is-TNil (ttl xs)
by(cases xs) simp-all
```

```
lemma terminal-ttl [simp]: terminal (ttl xs) = terminal xs
by(cases xs) simp-all
```

```
lemma unfold-tllist-eq-TNil [simp]:
  unfold-tllist IS-TNIL TNIL THD TTL a = TNil b <=> IS-TNIL a ∧ b = TNIL
  a
  by(auto simp add: unfold-tllist.code)
```

```
lemma TNil-eq-unfold-tllist [simp]:
  TNil b = unfold-tllist IS-TNIL TNIL THD TTL a <=> IS-TNIL a ∧ b = TNIL
  a
  by(auto simp add: unfold-tllist.code)
```

```
lemma tmap-is-TNil: is-TNil xs ==> tmap f g xs = TNil (g (terminal xs))
by(clarsimp simp add: is-TNil-def)
```

```
declare tllist.map-sel(2)[simp]
```

```
lemma ttl-tmap [simp]: ttl (tmap f g xs) = tmap f g (ttl xs)
by(cases xs) simp-all
```

```

lemma tmap-eq-TNil-conv:
  tmap f g xs = TNil y  $\longleftrightarrow$  ( $\exists y'. xs = TNil y' \wedge g y' = y$ )
by(cases xs) simp-all

lemma TNil-eq-tmap-conv:
  TNil y = tmap f g xs  $\longleftrightarrow$  ( $\exists y'. xs = TNil y' \wedge g y' = y$ )
by(cases xs) auto

declare tllist.setsel(1)[simp]

lemma tset-ttl: tset (ttl xs)  $\subseteq$  tset xs
by(cases xs) auto

lemma in-tset-ttlD:  $x \in tset (\text{ttl } xs) \implies x \in tset xs$ 
using tset-ttl[of xs] by auto

theorem tllist-set-induct[consumes 1, case-names find step]:
  assumes  $x \in tset xs$  and  $\bigwedge_{xs} \neg is\text{-TNil } xs \implies P (\text{thd } xs) xs$ 
  and  $\bigwedge_{xs} y. [\neg is\text{-TNil } xs; y \in tset (\text{ttl } xs); P y (\text{ttl } xs)] \implies P y xs$ 
  shows  $P x xs$ 
using assms by(induct)(fastforce simp del: tllist.disc(2) iff: tllist.disc(2), auto)

theorem set2-tllist-induct[consumes 1, case-names find step]:
  assumes  $x \in set2\text{-tllist } xs$  and  $\bigwedge_{xs} is\text{-TNil } xs \implies P (\text{terminal } xs) xs$ 
  and  $\bigwedge_{xs} y. [\neg is\text{-TNil } xs; y \in set2\text{-tllist } (ttl xs); P y (ttl xs)] \implies P y xs$ 
  shows  $P x xs$ 
using assms by(induct)(fastforce simp del: tllist.disc(1) iff: tllist.disc(1), auto)

```

5.4 Connection with 'a llist

```

context fixes b :: 'b begin
primcorec tllist-of-llist :: 'a llist  $\Rightarrow$  ('a, 'b) tllist where
  tllist-of-llist xs = (case xs of LNil  $\Rightarrow$  TNil b | LCons x xs'  $\Rightarrow$  TCons x (tllist-of-llist xs'))
end

primcorec llist-of-tllist :: ('a, 'b) tllist  $\Rightarrow$  'a llist
where llist-of-tllist xs = (case xs of TNil -  $\Rightarrow$  LNil | TCons x xs'  $\Rightarrow$  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  $\implies$  terminal (tllist-of-llist b xs) = b
unfolding lnull-def by simp

```

```

declare tllist-of-llist.sel(1)[simp del]

lemma lhd-LNil: lhd LNil = undefined
by(simp add: lhd-def)

lemma thd-TNil: thd (TNil b) = undefined
by(simp add: thd-def)

lemma thd-tllist-of-llist [simp]: thd (tllist-of-llist b xs) = lhd xs
by(cases xs)(simp-all add: thd-TNil lhd-LNil)

lemma ttl-tllist-of-llist [simp]: ttl (tllist-of-llist b xs) = tllist-of-llist b (ltl xs)
by(cases xs) simp-all

lemma llist-of-tllist-eq-LNil:
  llist-of-tllist xs = LNil  $\longleftrightarrow$  is-TNil xs
using llist-of-tllist.disc-if(1) unfolding lnull-def .

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]:  $\neg$  is-TNil xs  $\implies$  lhd (llist-of-tllist xs) = thd xs
by(cases xs) simp-all

lemma ltl-llist-of-tllist [simp]:
  ltl (llist-of-tllist xs) = llist-of-tllist (ttl xs)
by(cases xs) simp-all

lemma tllist-of-llist-cong [cong]:
  assumes xs = xs' lfinite xs'  $\implies$  b = b'
  shows tllist-of-llist b xs = tllist-of-llist b' xs'
proof(unfold `xs = xs'`)
  from assms have lfinite xs'  $\longrightarrow$  b = b' by simp
  thus tllist-of-llist b xs' = tllist-of-llist b' xs'
    by(coinduction arbitrary: xs') auto
qed

lemma llist-of-tllist-inverse [simp]:
  tllist-of-llist (terminal b) (llist-of-tllist b) = b
by(coinduction arbitrary: b) simp-all

lemma tllist-of-llist-eq [simp]: tllist-of-llist b' xs = TNil b  $\longleftrightarrow$  b = b'  $\wedge$  xs = LNil
by(cases xs) auto

lemma TNil-eq-tllist-of-llist [simp]: TNil b = tllist-of-llist b' xs  $\longleftrightarrow$  b = b'  $\wedge$  xs

```

```

= LNil
by(cases xs) auto

lemma tllist-of-llist-inject [simp]:
tllist-of-llist b xs = tllist-of-llist c ys  $\longleftrightarrow$  xs = ys  $\wedge$  (lfinite ys  $\longrightarrow$  b = c)
(is ?lhs  $\longleftrightarrow$  ?rhs)
proof(intro iffI conjI impI)
assume ?rhs
thus ?lhs by(auto intro: tllist-of-llist-cong)
next
assume ?lhs
thus xs = ys
by(coinduction arbitrary: xs ys)(auto simp add: lnull-def neq-LNil-conv)
assume lfinite ys
thus b = c using <?lhs>
unfolding <xs = ys> by(induct) simp-all
qed

lemma tllist-of-llist-inverse [simp]:
llist-of-tllist (tllist-of-llist b xs) = xs
by(coinduction arbitrary: xs) auto

definition cr-tllist :: ('a llist  $\times$  'b)  $\Rightarrow$  ('a, 'b) tllist  $\Rightarrow$  bool
where cr-tllist  $\equiv$  ( $\lambda$ (xs, b) ys. tllist-of-llist b xs = ys)

lemma Quotient-tllist:
Quotient ( $\lambda$ (xs, a) (ys, b). xs = ys  $\wedge$  (lfinite ys  $\longrightarrow$  a = b))
( $\lambda$ (xs, a). tllist-of-llist a xs) ( $\lambda$ ys. (llist-of-tllist ys, terminal ys)) cr-tllist
unfolding Quotient-alt-def cr-tllist-def by(auto intro: tllist-of-llist-cong)

lemma reflp-tllist: reflp ( $\lambda$ (xs, a) (ys, b). xs = ys  $\wedge$  (lfinite ys  $\longrightarrow$  a = b))
by(simp add: reflp-def)

setup-lifting Quotient-tllist reflp-tllist

context includes lifting-syntax
begin

lemma TNil-transfer [transfer-rule]:
(B ==> pcr-tllist A B) (Pair LNil) TNil
by(force simp add: pcr-tllist-def cr-tllist-def)

lemma TCons-transfer [transfer-rule]:
(A ==> pcr-tllist A B ==> pcr-tllist A B) (apfst  $\circ$  LCons) TCons
by(force simp add: pcr-tllist-def llist-all2-LCons1 cr-tllist-def)

lemma tmap-tllist-of-llist:
tmap f g (tllist-of-llist b xs) = tllist-of-llist (g b) (lmap f xs)
by(coinduction arbitrary: xs)(auto simp add: tmap-is-TNil)

```

```

lemma tmap-transfer [transfer-rule]:
  ((=) ==> (=) ==> pcr-tllist (=) (=) ==> pcr-tllist (=) (=)) (map-prod
  o lmap) tmap
by(force simp add: cr-tllist-def tllist.pcr-cr-eq tmap-tllist-of-llist)

lemma lset-llist-of-tllist [simp]:
  lset (llist-of-tllist xs) = tset xs (is ?lhs = ?rhs)
proof(intro set-eqI iffI)
  fix x
  assume x ∈ ?lhs
  thus x ∈ ?rhs
    by(induct llist-of-tllist xs arbitrary: xs rule: llist-set-induct)(auto simp: tl-
list.set-sel(2))
  next
  fix x
  assume x ∈ ?rhs
  thus x ∈ ?lhs
    proof(induct rule: tllist-set-induct)
      case (find xs)
      thus ?case by(cases xs) auto
  next
  case step
  thus ?case
    by(auto simp add: ltl-llist-of-tllist[symmetric] simp del: ltl-llist-of-tllist dest:
in-lset-ltlD)
  qed
qed

lemma tset-tllist-of-llist [simp]:
  tset (tllist-of-llist b xs) = lset xs
by(simp add: lset-llist-of-tllist[symmetric] del: lset-llist-of-tllist)

lemma tset-transfer [transfer-rule]:
  (pcr-tllist (=) (=) ==> (=)) (lset o fst) tset
by(auto simp add: cr-tllist-def tllist.pcr-cr-eq)

lemma is-TNil-transfer [transfer-rule]:
  (pcr-tllist (=) (=) ==> (=)) (λ(xs, b). lnull xs) is-TNil
by(auto simp add: tllist.pcr-cr-eq cr-tllist-def)

lemma thd-transfer [transfer-rule]:
  (pcr-tllist (=) (=) ==> (=)) (lhd o fst) thd
by(auto simp add: cr-tllist-def tllist.pcr-cr-eq)

lemma ttl-transfer [transfer-rule]:
  (pcr-tllist A B ==> pcr-tllist A B) (apfst ltl) ttl
by(force simp add: pcr-tllist-def cr-tllist-def intro: llist-all2-ltlI)

```

```

lemma llist-of-tllist-transfer [transfer-rule]:
  (pcr-tllist (=) B ==> (=)) fst llist-of-tllist
by(auto simp add: pcr-tllist-def cr-tllist-def llist.rel-eq)

lemma tllist-of-llist-transfer [transfer-rule]:
  ((=) ==> (=) ==> pcr-tllist (=) (=)) (λb xs. (xs, b)) tllist-of-llist
by(auto simp add: tllist.pcr-cr-eq cr-tllist-def)

lemma terminal-tllist-of-llist-lfinite [simp]:
  lfinite xs ==> terminal (tllist-of-llist b xs) = b
by(induct rule: lfinite.induct) simp-all

lemma set2-tllist-tllist-of-llist [simp]:
  set2-tllist (tllist-of-llist b xs) = (if lfinite xs then {b} else {})
proof(cases lfinite xs)
  case True
  thus ?thesis by(induct) auto
next
  case False
  { fix x
    assume x ∈ set2-tllist (tllist-of-llist b xs)
    hence False using False
    by(induct tllist-of-llist b xs arbitrary: xs rule: set2-tllist-induct) fastforce+
  }
  thus ?thesis using False by auto
qed

lemma set2-tllist-transfer [transfer-rule]:
  (pcr-tllist A B ==> rel-set B) (λ(xs, b). if lfinite xs then {b} else {}) set2-tllist
by(auto 4 4 simp add: pcr-tllist-def cr-tllist-def dest: llist-all2-lfiniteD intro: rel-setI)

lemma tllist-all2-transfer [transfer-rule]:
  ((=) ==> (=) ==> pcr-tllist (=) (=) ==> pcr-tllist (=) (=) ==> (=))
  (λP Q (xs, b) (ys, b'). llist-all2 P xs ys ∧ (lfinite xs → Q b b')) tllist-all2
unfolding tllist.pcr-cr-eq
apply(rule rel-funI)+
apply(clarify simp add: cr-tllist-def llist-all2-def tllist-all2-def)
apply(safe elim!: GrpE)
  apply simp-all
  apply(rule-tac b=tllist-of-llist (b, ba) bb in relcomppI)
  apply(auto intro!: GrpI simp add: tmap-tllist-of-llist)[2]
  apply(rule-tac b=tllist-of-llist (b, ba) bb in relcomppI)
  apply(auto simp add: tmap-tllist-of-llist intro!: GrpI split: if-split-asm)[2]
  apply(rule-tac b=llist-of-tllist bb in relcomppI)
  apply(auto intro!: GrpI)
  apply(transfer, auto intro: GrpI split: if-split-asm)+
done

```

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). apfst (lappend xs) (f b)
by(auto simp add: split-def lappend-inf)
```

```
lift-definition lappendt :: 'a llist ⇒ ('a, 'b) tllist ⇒ ('a, 'b) tllist
is apfst ∘ lappend
by(simp add: split-def)
```

```
lift-definition tfilter :: 'b ⇒ ('a ⇒ bool) ⇒ ('a, 'b) tllist ⇒ ('a, 'b) tllist
is λb P (xs, b'). (lfilter P xs, if lfinite xs then b' else b)
by(simp add: split-beta)
```

```
lift-definition tconcat :: 'b ⇒ ('a llist, 'b) tllist ⇒ ('a, 'b) tllist
is λb (xss, b'). (lconcat xss, if lfinite xss then b' else b)
by(simp add: split-beta)
```

```
lift-definition tnth :: ('a, 'b) tllist ⇒ nat ⇒ 'a
is lnth ∘ fst by auto
```

```
lift-definition tlength :: ('a, 'b) tllist ⇒ enat
is llength ∘ fst by auto
```

```
lift-definition tdropn :: nat ⇒ ('a, 'b) tllist ⇒ ('a, 'b) tllist
is apfst ∘ ldropn by auto
```

```
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
using assms
by transfer (clarsimp, erule lfinite.induct)
```

```
lemma is-TNil-tfinite [simp]: is-TNil xs ==> tfinite xs
by transfer clarsimp
```

5.7 The terminal element $terminal$

```
lemma terminal-tinfinite:
assumes ¬ tfinite xs
```

```

shows terminal xs = undefined
unfolding terminal0-terminal[symmetric]
using assms
apply(rule contrapos-np)
by(induct xs rule: terminal0.raw-induct[rotated 1, OF refl, consumes 1])(auto split:
tllist.split-asm)

lemma terminal-tllist-of-llist:
terminal (tllist-of-llist y xs) = (if lfinite xs then y else undefined)
by(simp add: terminal-tinfinite)

lemma terminal-transfer [transfer-rule]:
(pcr-tllist A (=) ==> (=)) (λ(xs, b). if lfinite xs then b else undefined) terminal
by(force simp add: cr-tllist-def pcr-tllist-def terminal-tllist-of-llist dest: llist-all2-lfiniteD)

lemma terminal-tmap [simp]: tfinite xs ==> terminal (tmap f g xs) = g (terminal
xs)
by(induct rule: tfinite-induct) simp-all

```

5.8 tmap

```

lemma tmap-eq-TCons-conv:
tmap f g xs = TCons y ys ↔
(∃ z zs. xs = TCons z zs ∧ f z = y ∧ tmap f g zs = ys)
by(cases xs) simp-all

lemma TCons-eq-tmap-conv:
TCons y ys = tmap f g xs ↔
(∃ z zs. xs = TCons z zs ∧ f z = y ∧ tmap f g zs = ys)
by(cases xs) auto

```

5.9 Appending two terminated lazy lists tappend

```

lemma tappend-TNil [simp, code, nitpick-simp]:
tappend (TNil b) f = f b
by transfer auto

lemma tappend-TCons [simp, code, nitpick-simp]:
tappend (TCons a tr) f = TCons a (tappend tr f)
by transfer(auto simp add: apfst-def map-prod-def split: prod.splits)

lemma tappend-TNil2 [simp]:
tappend xs TNil = xs
by transfer auto

lemma tappend-assoc: tappend (tappend xs f) g = tappend xs (λb. tappend (f b)
g)
by transfer(auto simp add: split-beta lappend-assoc)

lemma terminal-tappend:

```

```
terminal (tappend xs f) = (if tfinite xs then terminal (f (terminal xs)) else terminal xs)
```

```
by transfer(auto simp add: split-beta)
```

```
lemma tfinite-tappend: tfinite (tappend xs f)  $\longleftrightarrow$  tfinite xs  $\wedge$  tfinite (f (terminal xs))
```

```
by transfer auto
```

```
lift-definition tcast :: ('a, 'b) tllist  $\Rightarrow$  ('a, 'c) tllist  
is  $\lambda(xs, a). (xs, \text{undefined})$  by clar simp
```

```
lemma tappend-inf:  $\neg$  tfinite xs  $\implies$  tappend xs f = tcast xs  
by(transfer)(auto simp add: apfst-def map-prod-def split-beta lappend-inf)
```

tappend is the monadic bind on ('a, 'b) tllist

```
lemmas tlist-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  
by transfer auto
```

```
lemma lappendt-LCons [simp, code, nitpick-simp]:  
lappendt (LCons x xs) tr = TCons x (lappendt xs tr)  
by transfer auto
```

```
lemma terminal-lappendt-lfinite [simp]:  
lfinite xs  $\implies$  terminal (lappendt xs ys) = terminal ys  
by transfer auto
```

```
lemma tlist-of-llist-eq-lappendt-conv:  
tlist-of-llist a xs = lappendt ys zs  $\longleftrightarrow$   
 $(\exists xs' a'. xs = lappendt ys xs' \wedge zs = tlist-of-llist a' xs' \wedge (lfinite ys \longrightarrow a = a'))$   
by transfer auto
```

```
lemma tset-lappendt-lfinite [simp]:  
lfinite xs  $\implies$  tset (lappendt xs ys) = lset xs  $\cup$  tset ys  
by transfer auto
```

5.11 Filtering terminated lazy lists tfilter

```
lemma tfilter-TNil [simp]:  
tfilter b' P (TNil b) = TNil b  
by transfer auto
```

```
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)  
by transfer auto
```

```

lemma is-TNil-tfilter[simp]:
  is-TNil (tfilter y P xs)  $\longleftrightarrow$  ( $\forall x \in tset xs. \neg P x$ )
by transfer auto

lemma tfilter-empty-conv:
  tfilter y P xs = TNil y'  $\longleftrightarrow$  ( $\forall x \in tset xs. \neg P x$ )  $\wedge$  (if tfinite xs then terminal
  xs = y' else y = y')
by transfer(clarsimp simp add: lfilter-eq-LNil)

lemma tfilter-eq-TConsD:
  tfilter a P ys = TCons x xs  $\Longrightarrow$ 
   $\exists us vs. ys = lappendt us (TCons x vs) \wedge lfinite us \wedge (\forall u \in lset us. \neg P u) \wedge P$ 
  x  $\wedge$  xs = tfilter a P vs
by transfer(fastforce dest: lfilter-eq-LConsD[OF sym])

Use a version of tfilter for code generation that does not evaluate the first
argument

definition tfilter' :: (unit  $\Rightarrow$  'b)  $\Rightarrow$  ('a  $\Rightarrow$  bool)  $\Rightarrow$  ('a, 'b) tllist  $\Rightarrow$  ('a, 'b) tllist
where [simp, code del]: tfilter' b = tfilter (b ())

lemma tfilter-code [code, code-unfold]:
  tfilter = ( $\lambda b.$  tfilter' ( $\lambda \_. b$ ))
by simp

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)
by simp-all

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'
by transfer auto

```

```

lemma tconcat-TCons [simp]: tconcat b (TCons a tr) = lappendt a (tconcat b tr)
by transfer auto

```

Use a version of tconcat for code generation that does not evaluate the first argument

```

definition tconcat' :: (unit  $\Rightarrow$  'b)  $\Rightarrow$  ('a llist, 'b) tllist  $\Rightarrow$  ('a, 'b) tllist
where [simp, code del]: tconcat' b = tconcat (b ())

```

```

lemma tconcat-code [code, code-unfold]: tconcat = ( $\lambda b.$  tconcat' ( $\lambda \_. b$ )))
by simp

```

```

lemma tconcat'-code [code]:
  tconcat' b (TNil b') = TNil b'
  tconcat' b (TCons a tr) = lappendt a (tconcat' b tr)
by simp-all

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  $\longleftrightarrow$  ( $\exists$  b'. ts = TNil b'  $\wedge$  Q b')
by transfer auto

lemma tllist-all2-TNil2: tllist-all2 P Q ts (TNil b')  $\longleftrightarrow$  ( $\exists$  b. ts = TNil b  $\wedge$  Q b')
by transfer auto

lemma tllist-all2-TCons1:
  tllist-all2 P Q (TCons x ts) ts'  $\longleftrightarrow$  ( $\exists$  x' ts''. ts' = TCons x' ts''  $\wedge$  P x x'  $\wedge$ 
  tllist-all2 P Q ts ts'')
by transfer(fastforce simp add: llist-all2-LCons1 dest: llist-all2-lfiniteD)

lemma tllist-all2-TCons2:
  tllist-all2 P Q ts' (TCons x ts)  $\longleftrightarrow$  ( $\exists$  x' ts''. ts' = TCons x' ts''  $\wedge$  P x' x  $\wedge$ 
  tllist-all2 P Q ts'' ts)
by transfer(fastforce simp add: llist-all2-LCons2 dest: llist-all2-lfiniteD)

lemma tllist-all2-coinduct [consumes 1, case-names tllist-all2, case-conclusion tl-
list-all2 is-TNil TNil TCons, coinduct pred: tllist-all2]:
  assumes X xs ys
  and  $\bigwedge$  xs ys. X xs ys  $\Longrightarrow$ 
  (is-TNil xs  $\longleftrightarrow$  is-TNil ys)  $\wedge$ 
  (is-TNil xs  $\longrightarrow$  is-TNil ys  $\longrightarrow$  R (terminal xs) (terminal ys))  $\wedge$ 
  ( $\neg$  is-TNil xs  $\longrightarrow$   $\neg$  is-TNil ys  $\longrightarrow$  P (thd xs) (thd ys)  $\wedge$  (X (ttl xs) (ttl ys)  $\vee$ 
  tllist-all2 P R (ttl xs) (ttl ys)))
  shows tllist-all2 P R xs ys
  using assms
  apply(transfer fixing: P R)
  apply clar simp
  apply(rule conjI)
  apply(erule llist-all2-coinduct, blast, blast)
  apply (rule impI)
  subgoal premises prems for X xs b ys c
  proof -
    from ⟨lfinite xs⟩ ⟨X (xs, b) (ys, c)⟩

```

```

show R b c
  by(induct arbitrary: ys rule: lfinite-induct)(auto dest: prems(2))
qed
done

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'
  using assms
  by(cases xs)(fastforce simp add: tllist-all2-TCons1 tllist-all2-TNil1)+

lemma tllist-all2-tmap1:
  tllist-all2 P Q (tmap f g xs) ys  $\longleftrightarrow$  tllist-all2 ( $\lambda x. P(f x)$ ) ( $\lambda x. Q(g x)$ ) xs ys
  by(transfer)(auto simp add: llist-all2-lmap1)

lemma tllist-all2-tmap2:
  tllist-all2 P Q xs (tmap f g ys)  $\longleftrightarrow$  tllist-all2 ( $\lambda x y. P x (f y)$ ) ( $\lambda x y. Q x (g y)$ )
  xs ys
  by(transfer)(auto simp add: llist-all2-lmap2)

lemma tllist-all2-mono:
   $\llbracket tllist-all2 P Q xs ys; \bigwedge x y. P x y \implies P' x y; \bigwedge x y. Q x y \implies Q' x y \rrbracket$ 
   $\implies tllist-all2 P' Q' xs ys$ 
  by transfer(auto elim!: llist-all2-mono)

lemma tllist-all2-tlengthD: tllist-all2 P Q xs ys  $\implies$  tlength xs = tlength ys
  by(transfer)(auto dest: llist-all2-tlengthD)

lemma tllist-all2-tfiniteD: tllist-all2 P Q xs ys  $\implies$  tfinite xs = tfinite ys
  by(transfer)(auto dest: llist-all2-tfiniteD)

lemma tllist-all2-tfinite1-terminalD:
   $\llbracket tllist-all2 P Q xs ys; tfinite xs \rrbracket \implies Q(\text{terminal } xs) (\text{terminal } ys)$ 
  by(frule tllist-all2-tfiniteD)(transfer, auto)

lemma tllist-all2-tfinite2-terminalD:
   $\llbracket tllist-all2 P Q xs ys; tfinite ys \rrbracket \implies Q(\text{terminal } xs) (\text{terminal } ys)$ 
  by(metis tllist-all2-tfinite1-terminalD tllist-all2-tfiniteD)

lemma tllist-all2D-list-all2-list-of-tllist:
  tllist-all2 P Q xs ys  $\implies$  llist-all2 P (list-of-tllist xs) (list-of-tllist ys)
  by(transfer) auto

lemma tllist-all2-is-TNilD:
  tllist-all2 P Q xs ys  $\implies$  is-TNil xs  $\longleftrightarrow$  is-TNil ys
  by(cases xs)(auto simp add: tllist-all2-TNil1 tllist-all2-TCons1)

```

```

lemma tllist-all2-thdD:
   $\llbracket \text{tllist-all2 } P \ Q \ xs \ ys; \neg \text{is-TNil } xs \vee \neg \text{is-TNil } ys \rrbracket \implies P (\text{thd } xs) (\text{thd } ys)$ 
by(cases xs)(auto simp add: tllist-all2-TNil1 tllist-all2-TCons1)

lemma tllist-all2-ttlI:
   $\llbracket \text{tllist-all2 } P \ Q \ xs \ ys; \neg \text{is-TNil } xs \vee \neg \text{is-TNil } ys \rrbracket \implies \text{tllist-all2 } P \ Q (\text{ttl } xs)$ 
  ( $\text{ttl } ys$ )
by(cases xs)(auto simp add: tllist-all2-TNil1 tllist-all2-TCons1)

lemma tllist-all2-refl:
   $\text{tllist-all2 } P \ Q \ xs \ xs \longleftrightarrow (\forall x \in \text{tset } xs. \ P \ x \ x) \wedge (\text{tfinite } xs \longrightarrow Q (\text{terminal } xs))$ 
  ( $\text{terminal } xs$ )
by transfer(auto)

lemma tllist-all2-reflI:
   $\llbracket \bigwedge x. \ x \in \text{tset } xs \implies P \ x \ x; \text{tfinite } xs \implies Q (\text{terminal } xs) (\text{terminal } xs) \rrbracket$ 
   $\implies \text{tllist-all2 } P \ Q \ xs \ xs$ 
by(simp add: tllist-all2-refl)

lemma tllist-all2-conv-all-tnth:
   $\text{tllist-all2 } P \ Q \ xs \ ys \longleftrightarrow$ 
   $\text{tlength } xs = \text{tlength } ys \wedge$ 
   $(\forall n. \ \text{enat } n < \text{tlength } xs \longrightarrow P (\text{tnth } xs \ n) (\text{tnth } ys \ n)) \wedge$ 
   $(\text{tfinite } xs \longrightarrow Q (\text{terminal } xs) (\text{terminal } ys))$ 
by transfer(auto 4 4 simp add: llist-all2-conv-all-tnth split: if-split-asm dest: lfinite-tlength-enat not-lfinite-tlength)

lemma tllist-all2-tnthD:
   $\llbracket \text{tllist-all2 } P \ Q \ xs \ ys; \text{enat } n < \text{tlength } xs \rrbracket$ 
   $\implies P (\text{tnth } xs \ n) (\text{tnth } ys \ n)$ 
by(simp add: tllist-all2-conv-all-tnth)

lemma tllist-all2-tnthD2:
   $\llbracket \text{tllist-all2 } P \ Q \ xs \ ys; \text{enat } n < \text{tlength } ys \rrbracket$ 
   $\implies P (\text{tnth } xs \ n) (\text{tnth } ys \ n)$ 
by(simp add: tllist-all2-conv-all-tnth)

lemmas tllist-all2-eq = tllist.rel-eq

lemma tmap-eq-tmap-conv-tllist-all2:
   $\text{tmap } f \ g \ xs = \text{tmap } f' \ g' \ ys \longleftrightarrow$ 
   $\text{tllist-all2 } (\lambda x \ y. \ f \ x = f' \ y) (\lambda x \ y. \ g \ x = g' \ y) \ xs \ ys$ 
apply transfer
apply(clarify simp add: lmap-eq-lmap-conv-llist-all2)
apply(auto dest: llist-all2-lfiniteD)
done

lemma tllist-all2-trans:

```

```

 $\llist-all2 P Q xs ys; \llist-all2 P Q ys zs; \text{transp } P; \text{transp } Q \llbracket$ 
 $\implies \llist-all2 P Q xs zs$ 
by transfer(auto elim: llist-all2-trans dest: llist-all2-lfiniteD transpD)

lemma tllist-all2-tappendI:
 $\llist-all2 P Q xs ys;$ 
 $\llbracket \text{tfinite } xs; \text{tfinite } ys; Q (\text{terminal } xs) (\text{terminal } ys) \rrbracket$ 
 $\implies \llist-all2 P R (xs' (\text{terminal } xs)) (ys' (\text{terminal } ys)) \llbracket$ 
 $\implies \llist-all2 P R (\text{tappend } xs xs') (\text{tappend } ys ys')$ 
apply transfer
apply(auto 4 3 simp add: apfst-def map-prod-def lappend-inf split: prod.split-asm
dest: llist-all2-lfiniteD intro: llist-all2-lappendI)
apply(frule llist-all2-lfiniteD, simp add: lappend-inf)
done

lemma llist-all2-tllist-of-llistI:
 $\llist-all2 A B xs ys \implies \llist-all2 A (\text{llist-of-tllist } xs) (\text{llist-of-tllist } ys)$ 
by(coinduction arbitrary: xs ys)(auto dest: tllist-all2-is-TNilD tllist-all2-thdD intro: tllist-all2-ttlI)

lemma tllist-all2-tllist-of-llist [simp]:
 $\llist-all2 A B (\text{tllist-of-llist } b xs) (\text{tllist-of-llist } c ys) \longleftrightarrow$ 
 $\llist-all2 A xs ys \wedge (\text{lfinite } xs \longrightarrow B b c)$ 
by transfer auto

```

5.14 From a terminated lazy list to a lazy list llist-of-tllist

```

lemma llist-of-tllist-tmap [simp]:
 $\text{llist-of-tllist} (\text{tmap } f g xs) = \text{lmap } f (\text{llist-of-tllist } xs)$ 
by transfer auto

lemma llist-of-tllist-tappend:
 $\text{llist-of-tllist} (\text{tappend } xs f) = \text{lappend} (\text{llist-of-tllist } xs) (\text{llist-of-tllist} (f (\text{terminal } xs)))$ 
by(transfer)(auto simp add: lappend-inf)

lemma llist-of-tllist-lappendt [simp]:
 $\text{llist-of-tllist} (\text{lappendt } xs tr) = \text{lappend } xs (\text{llist-of-tllist } tr)$ 
by transfer auto

lemma llist-of-tllist-tfilter [simp]:
 $\text{llist-of-tllist} (\text{tfilter } b P tr) = \text{lfilter } P (\text{llist-of-tllist } tr)$ 
by transfer auto

lemma llist-of-tllist-tconcat:
 $\text{llist-of-tllist} (\text{tconcat } b trs) = \text{lconcat} (\text{llist-of-tllist } trs)$ 
by transfer auto

lemma llist-of-tllist-eq-lappend-conv:

```

$llist\text{-}of\text{-}tllist\ xs = lappend\ us\ vs \longleftrightarrow$
 $(\exists ys. xs = lappendt\ us\ ys \wedge vs = llist\text{-}of\text{-}tllist\ ys \wedge \text{terminal}\ xs = \text{terminal}\ ys)$
by transfer auto

5.15 The nth element of a terminated lazy list $tnth$

lemma $tnth\text{-}TNil$ [*nitpick-simp*]:

$tnth\ (TNil\ b)\ n = \text{undefined}\ n$
by(transfer)(simp add: lnth-LNil)

lemma $tnth\text{-}TCons$:

$tnth\ (TCons\ x\ xs)\ n = (\text{case}\ n\ \text{of}\ 0 \Rightarrow x \mid \text{Suc}\ n' \Rightarrow \text{tnth}\ xs\ n')$
by(transfer)(auto simp add: lnth-LCons split: nat.split)

lemma $tnth\text{-}code$ [*simp, nitpick-simp, code*]:

shows $tnth\text{-}0$: $tnth\ (TCons\ x\ xs)\ 0 = x$
and $tnth\text{-Suc-TCons}$: $tnth\ (TCons\ x\ xs)\ (\text{Suc}\ n) = \text{tnth}\ xs\ n$
by(simp-all add: tnth-TCons)

lemma $lnth\text{-}llist\text{-}of\text{-}tllist$ [*simp*]:

$lnth\ (llist\text{-}of\text{-}tllist\ xs) = \text{tnth}\ xs$
by(transfer)(auto)

lemma $tnth\text{-}tmap$ [*simp*]: $\text{enat}\ n < \text{tlength}\ xs \implies \text{tnth}\ (\text{tmap}\ f\ g\ xs)\ n = f\ (\text{tnth}\ xs\ n)$

by transfer simp

5.16 The length of a terminated lazy list $tlength$

lemma [*simp, nitpick-simp*]:

shows $tlength\text{-}TNil$: $tlength\ (TNil\ b) = 0$
and $tlength\text{-}TCons$: $tlength\ (TCons\ x\ xs) = \text{eSuc}\ (\text{tlength}\ xs)$
apply(transfer, simp)
apply(transfer, auto)
done

lemma $llength\text{-}llist\text{-}of\text{-}tllist$ [*simp*]: $llength\ (llist\text{-}of\text{-}tllist\ xs) = \text{tlength}\ xs$
by transfer auto

lemma $tlength\text{-}tmap$ [*simp*]: $\text{tlength}\ (\text{tmap}\ f\ g\ xs) = \text{tlength}\ xs$
by transfer simp

definition $\text{gen-tlength} :: \text{nat} \Rightarrow ('a, 'b) \text{tllist} \Rightarrow \text{enat}$
where $\text{gen-tlength}\ n\ xs = \text{enat}\ n + \text{tlength}\ xs$

lemma gen-tlength-code [*code*]:

$\text{gen-tlength}\ n\ (TNil\ b) = \text{enat}\ n$
 $\text{gen-tlength}\ n\ (TCons\ x\ xs) = \text{gen-tlength}\ (n + 1)\ xs$
by(simp-all add: gen-tlength-def iadd-Suc eSuc-enat[symmetric] iadd-Suc-right)

lemma *tlength-code* [code]: *tlength* = *gen-tlength* 0
by(*simp add*: *gen-tlength-def fun-eq-iff zero-enat-def*)

5.17 *tdropn*

lemma *tdropn-0* [simp, code, nitpick-simp]: *tdropn* 0 *xs* = *xs*
by transfer auto

lemma *tdropn-TNil* [simp, code]: *tdropn* *n* (*TNil b*) = (*TNil b*)
by transfer(auto)

lemma *tdropn-Suc-TCons* [simp, code]: *tdropn* (*Suc n*) (*TCons x xs*) = *tdropn* *n*
xs
by transfer(auto)

lemma *tdropn-Suc* [nitpick-simp]: *tdropn* (*Suc n*) *xs* = (case *xs* of *TNil b* \Rightarrow *TNil b* | *TCons x xs'* \Rightarrow *tdropn n xs'*)
by(cases *xs*) simp-all

lemma *lappendt-ltake-tdropn*:
lappendt (*ltake* (*enat n*) (*llist-of-tllist xs*)) (*tdropn n xs*) = *xs*
by transfer (auto)

lemma *llist-of-tllist-tdropn* [simp]:
llist-of-tllist (*tdropn n xs*) = *ldropn n* (*llist-of-tllist xs*)
by transfer auto

lemma *tdropn-Suc-conv-tdropn*:
enat n < tlength xs \Longrightarrow *TCons* (*tnth xs n*) (*tdropn (Suc n) xs*) = *tdropn n xs*
by transfer(auto simp add: *ldropn-Suc-conv-ldropn*)

lemma *tlength-tdropn* [simp]: *tlength* (*tdropn n xs*) = *tlength xs* - *enat n*
by transfer auto

lemma *tnth-tdropn* [simp]: *enat (n + m) < tlength xs* \Longrightarrow *tnth* (*tdropn n xs*) *m* = *tnth xs (m + n)*
by transfer auto

5.18 *tset*

lemma *tset-induct* [consumes 1, case-names find step]:
assumes *x* \in *tset xs*
and \bigwedge *xs. P (TCons x xs)*
and \bigwedge *x' xs. [x* \in *tset xs; x* \neq *x'*; *P xs*] \Longrightarrow *P (TCons x' xs)*
shows *P xs*
using assms
by transfer(clarsimp, erule lset-induct)

lemma *tset-conv-tnth*: *tset xs* = {*tnth xs n | n . enat n < tlength xs*}
by transfer(simp add: lset-conv-tnth)

```

lemma in-tset-conv-tnth:  $x \in tset xs \longleftrightarrow (\exists n. enat n < tlength xs \wedge nth xs n = x)$ 
using tset-conv-tnth[of xs] by auto

```

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]:
  ( $\text{rel-pre-tllist } A B C \implies \text{rel-set } A$ ) set1-pre-tllist set1-pre-tllist
  by(auto simp add: rel-pre-tllist-def vimage2p-def rel-fun-def set1-pre-tllist-def rel-set-def
  collect-def sum-set-defs prod-set-defs elim: rel-sum.cases split: sum.split-asm)

lemma set2-pre-tllist-transfer [transfer-rule]:
  ( $\text{rel-pre-tllist } A B C \implies \text{rel-set } B$ ) set2-pre-tllist set2-pre-tllist
  by(auto simp add: rel-pre-tllist-def vimage2p-def rel-fun-def set2-pre-tllist-def rel-set-def
  collect-def sum-set-defs prod-set-defs elim: rel-sum.cases split: sum.split-asm)

lemma set3-pre-tllist-transfer [transfer-rule]:
  ( $\text{rel-pre-tllist } A B C \implies \text{rel-set } C$ ) set3-pre-tllist set3-pre-tllist
  by(auto simp add: rel-pre-tllist-def vimage2p-def rel-fun-def set3-pre-tllist-def rel-set-def
  collect-def sum-set-defs prod-set-defs elim: rel-sum.cases split: sum.split-asm)

lemma TNil-transfer2 [transfer-rule]: ( $B \implies \text{tllist-all2 } A B$ ) TNil TNil
by auto
declare TNil-transfer [transfer-rule]

lemma TCons-transfer2 [transfer-rule]:
  ( $A \implies \text{tllist-all2 } A B \implies \text{tllist-all2 } A B$ ) TCons TCons
  unfolding rel-fun-def by simp
  declare TCons-transfer [transfer-rule]

lemma case-tllist-transfer [transfer-rule]:
  ( $(B \implies C) \implies (A \implies \text{tllist-all2 } A B \implies C) \implies \text{tllist-all2 } A B \implies C$ )
  case-tllist case-tllist
  unfolding rel-fun-def
  by (simp add: tllist-all2-TNil1 tllist-all2-TNil2 split: tllist.split)

lemma unfold-tllist-transfer [transfer-rule]:
  ( $(A \implies (=)) \implies (A \implies B) \implies (A \implies C) \implies (A \implies A) \implies A \implies \text{tllist-all2 } C B$ )
  unfold-tllist unfold-tllist

```

```

proof(rule rel-funI)+
fix IS-TNIL1 :: 'a ⇒ bool and IS-TNIL2
TERMINAL1 TERMINAL2 THD1 THD2 TTL1 TTL2 x y
assume rel: (A ==> (=)) IS-TNIL1 IS-TNIL2 (A ==> B) TERMINAL1
TERMINAL2
(A ==> C) THD1 THD2 (A ==> A) TTL1 TTL2
and A x y
show tllist-all2 C B (unfold-tllist IS-TNIL1 TERMINAL1 THD1 TTL1 x) (unfold-tllist
IS-TNIL2 TERMINAL2 THD2 TTL2 y)
using ⟨A x y⟩
apply(coinduction arbitrary: x y)
using rel by(auto 4 4 elim: rel-funE)
qed

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(rule rel-funI)+
fix IS-TNIL1 MORE1 :: 'a ⇒ bool and IS-TNIL2
TERMINAL1 TERMINAL2 THD1 THD2 MORE2 STOP1 STOP2 TTL1 TTL2
x y
assume rel: (A ==> (=)) IS-TNIL1 IS-TNIL2 (A ==> B) TERMINAL1
TERMINAL2
(A ==> C) THD1 THD2 (A ==> (=)) MORE1 MORE2
(A ==> tllist-all2 C B) STOP1 STOP2 (A ==> A) TTL1 TTL2
and A x y
show tllist-all2 C B (corec-tllist IS-TNIL1 TERMINAL1 THD1 MORE1 STOP1
TTL1 x) (corec-tllist IS-TNIL2 TERMINAL2 THD2 MORE2 STOP2 TTL2 y)
using ⟨A x y⟩
apply(coinduction arbitrary: x y)
using rel by(auto 4 4 elim: rel-funE)
qed

lemma ttl-transfer2 [transfer-rule]:
(tllist-all2 A B ==> tllist-all2 A B) ttl ttl
unfolding ttl-def[abs-def] by transfer-prover
declare ttl-transfer [transfer-rule]

lemma tset-transfer2 [transfer-rule]:
(tllist-all2 A B ==> rel-set A) tset tset
by (intro rel-funI rel-setI) (auto simp only: in-tset-conv-tnth tllist-all2-conv-all-tnth
Bex-def)

lemma tmap-transfer2 [transfer-rule]:
((A ==> B) ==> (C ==> D) ==> tllist-all2 A C ==> tllist-all2 B
D) tmap tmap
by(auto simp add: rel-fun-def tllist-all2-tmap1 tllist-all2-tmap2 elim: tllist-all2-mono)
declare tmap-transfer [transfer-rule]

```

```

lemma is-TNil-transfer2 [transfer-rule]:
  ( $tllist\text{-}all2 A B \implies (=)$ ) is-TNil is-TNil
by(auto dest: tllist-all2-is-TNilD)
declare is-TNil-transfer [transfer-rule]

lemma tappend-transfer [transfer-rule]:
  ( $tllist\text{-}all2 A B \implies (B \implies tllist\text{-}all2 A C) \implies tllist\text{-}all2 A C$ ) tappend
  tappend
by(auto intro: tllist-all2-tappendI elim: rel-funE)
declare tappend.transfer [transfer-rule]

lemma lappendt-transfer [transfer-rule]:
  ( $llist\text{-}all2 A \implies tllist\text{-}all2 A B \implies tllist\text{-}all2 A B$ ) lappendt lappendt
  unfolding rel-fun-def
by transfer(auto intro: llist-all2-lappendI)
declare lappendt.transfer [transfer-rule]

lemma llist-of-tllist-transfer2 [transfer-rule]:
  ( $tllist\text{-}all2 A B \implies llist\text{-}all2 A$ ) llist-of-tllist llist-of-tllist
by(auto intro: llist-all2-tllist-of-tllistI)
declare llist-of-tllist-transfer [transfer-rule]

lemma tllist-of-llist-transfer2 [transfer-rule]:
  ( $B \implies llist\text{-}all2 A \implies tllist\text{-}all2 A B$ ) tllist-of-llist tllist-of-llist
by(auto intro!: rel-funI)
declare tllist-of-llist-transfer [transfer-rule]

lemma tlength-transfer [transfer-rule]:
  ( $tllist\text{-}all2 A B \implies (=)$ ) tlength tlength
by(auto dest: tllist-all2-tlengthD)
declare tlength.transfer [transfer-rule]

lemma tdropn-transfer [transfer-rule]:
  ( $(=) \implies tllist\text{-}all2 A B \implies tllist\text{-}all2 A B$ ) tdropn tdropn
  unfolding rel-fun-def
by transfer(auto intro: llist-all2-ldropnI)
declare tdropn.transfer [transfer-rule]

lemma tfilter-transfer [transfer-rule]:
  ( $B \implies (A \implies (=)) \implies tllist\text{-}all2 A B \implies tllist\text{-}all2 A B$ ) tfilter
  tfilter
  unfolding rel-fun-def
by transfer(auto intro: llist-all2-lfilterI dest: llist-all2-lfiniteD)
declare tfilter.transfer [transfer-rule]

lemma tconcat-transfer [transfer-rule]:
  ( $B \implies tllist\text{-}all2 (llist\text{-}all2 A) B \implies tllist\text{-}all2 A B$ ) tconcat tconcat
  unfolding rel-fun-def

```

```

by transfer(auto intro: llist-all2-lconcatI dest: llist-all2-lfiniteD)
declare tconcat.transfer [transfer-rule]

lemma tllist-all2-rsp:
assumes R1:  $\forall x y. R1 x y \rightarrow (\forall a b. R1 a b \rightarrow S x a = T y b)$ 
and R2:  $\forall x y. R2 x y \rightarrow (\forall a b. R2 a b \rightarrow S' x a = T' y b)$ 
and xsyss: 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
assume tllist-all2 S S' xs xs'
with xsyss xs'ys' show tllist-all2 T T' ys ys'
proof(coinduction arbitrary: ys ys' xs xs')
case (tllist-all2 ys ys' xs xs')
thus ?case
by cases (auto 4 4 simp add: tllist-all2-TCons1 tllist-all2-TCons2 tllist-all2-TNil1
tllist-all2-TNil2 dest: R1[rule-format] R2[rule-format])
qed
next
assume tllist-all2 T T' ys ys'
with xsyss xs'ys' show tllist-all2 S S' xs xs'
proof(coinduction arbitrary: xs xs' ys ys')
case (tllist-all2 xs xs' ys ys')
thus ?case
by cases(auto 4 4 simp add: tllist-all2-TCons1 tllist-all2-TCons2 tllist-all2-TNil1
tllist-all2-TNil2 dest: R1[rule-format] R2[rule-format])
qed
qed

lemma tllist-all2-transfer2 [transfer-rule]:
((R1 ==> R1 ==> (=)) ==> (R2 ==> R2 ==> (=)) ==>
tllist-all2 R1 R2 ==> tllist-all2 R1 R2 ==> (=)) tllist-all2 tllist-all2
by (simp add: tllist-all2-rsp rel-fun-def)
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
    by (erule transpE) blast

  lemma id-respect [quot-respect]:
    (R ==> R) id id
    by (fact id-rsp)

  lemma id-preserve [quot-preserve]:
    assumes Quotient3 R Abs Rep
    shows (Rep --> Abs) id = id
    using Quotient3-abs-rep [OF assms] by (simp add: fun_eq_iff)

  functor lmap: lmap
    by (simp-all add: fun_eq_iff id-def llist.map-comp)

  declare llist.map_id0 [id-simps]

  lemma reflp_llist_all2: reflp R ==> reflp (llist_all2 R)
    by (rule reflpI) (auto simp add: llist_all2_conv_all_lnth elim: reflpE)

  lemma symp_llist_all2: symp R ==> symp (llist_all2 R)
    by (rule sympI) (auto simp add: llist_all2_conv_all_lnth elim: sympE)

  lemma transp_llist_all2: transp R ==> transp (llist_all2 R)
    by (rule transpI) (rule llist_all2_trans)

  lemma llist_equivp [quot-equiv]:
    equivp R ==> equivp (llist_all2 R)
    by (simp add: equivp_refl symp_transp_refl_llist_all2 symp_llist_all2 transp_llist_all2)

  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(intro ext)
fix IS-LNIL LHD LTL a
show ?lhs IS-LNIL LHD LTL a = ?rhs IS-LNIL LHD LTL a
  by(coinduction arbitrary: a)(auto simp add: Quotient3-abs-rep[OF q1] Quo-
tient3-abs-rep[OF q2])
qed

lemma Quotient-lmap-Abs-Rep:
Quotient3 R Abs Rep  $\implies$  lmap Abs (lmap Rep a) = a
  by (drule abs-o-rep) (simp add: llist.map-id0 llist.map-comp)

lemma llist-all2-rel:
assumes Quotient3 R Abs Rep
shows llist-all2 R r s  $\longleftrightarrow$  llist-all2 R r r  $\wedge$  llist-all2 R s s  $\wedge$  (lmap Abs r =
lmap Abs s)
  (is ?lhs  $\longleftrightarrow$  ?rhs)
proof
assume ?lhs
hence llist-all2 R r r
  apply –
  apply(rule llist-all2-refI)
  apply(clar simp simp add: lset-conv-lnth)
  apply(metis Quotient3-rel[OF assms] llist-all2-lnthD)
  done
moreover from ‹?lhs› have llist-all2 R s s
  apply –
  apply(rule llist-all2-refI)
  apply(clar simp simp add: lset-conv-lnth)
  apply(metis Quotient3-rel[OF assms] llist-all2-lnthD2)
  done
moreover from ‹?lhs› have llength r = llength s by(rule llist-all2-llengthD)
hence lmap Abs r = lmap Abs s using ‹?lhs›
  unfolding lmap-eq-lmap-conv-llist-all2
  apply –
  apply(erule llist-all2-all-lnthI)
  apply(drule (1) llist-all2-lnthD)
  apply(metis Quotient3-rel[OF assms])
  done
ultimately show ?rhs by blast
next
assume ?rhs thus ?lhs
  unfolding lmap-eq-lmap-conv-llist-all2
  by(clar simp simp add: llist-all2-conv-all-lnth simp del: llist-all2-same)(metis
Quotient3-rel[OF assms])
qed

lemma Quotient-llist-all2-lmap-Rep:
Quotient3 R Abs Rep  $\implies$  llist-all2 R (lmap Rep a) (lmap Rep a)
  by(auto intro!: llist-all2-all-lnthI intro: Quotient3-rep-reflp)

```

```

lemma llist-quotient [quot-thm]:
  Quotient3 R Abs Rep ==> Quotient3 (llist-all2 R) (lmap Abs) (lmap Rep)
by(blast intro: Quotient3I dest: Quotient-lmap-Abs-Rep Quotient-llist-all2-lmap-Rep
llist-all2-rel)

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
using Quotient3-abs-rep[OF assms]
by(simp add: fun-eq-iff llist.map-comp o-def)

lemmas LCons-respect [quot-respect] = LCons-transfer

lemma LNil-preserve [quot-preserve]:
  lmap Abs LNil = LNil
by simp

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
using Quotient3-abs-rep[OF a] Quotient3-abs-rep[OF b]
by(simp-all add: fun-eq-iff llist.map-comp o-def)

lemma lmap-respect [quot-respect]:
shows ((R1 ===> R2) ===> (llist-all2 R1) ===> llist-all2 R2) lmap lmap
and ((R1 ===> ()) ===> (llist-all2 R1) ===> ()) lmap lmap
by(fact lmap-transfer)(simp add: llist-all2-conv-all-lnth lmap-eq-lmap-conv-llist-all2
rel-fun-def)

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
using Quotient3-abs-rep[OF assms]
by(simp add: fun-eq-iff llist-all2-lmap1 llist-all2-lmap2)

lemma llist-all2-preserve2 [quot-preserve]:
assumes Quotient3 R Abs Rep
shows (llist-all2 ((Rep ---> Rep ---> id) R) l m) = (l = m)

```

```

by (simp add: map-fun-def [abs-def] Quotient3-rel-rep [OF assms] llist.rel-eq
comp-def)

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(intro ext)
fix IS-LNIL LHD endORmore LTL-end LTL-more b
show ?lhs IS-LNIL LHD endORmore LTL-end LTL-more b = ?rhs IS-LNIL LHD
endORmore LTL-end LTL-more b
by(coinduction arbitrary: b rule: llist.coinduct-strong)
(auto simp add: Quotient3-abs-rep[OF q1] Quotient3-abs-rep[OF q2] Quo-
tient-lmap-Abs-Rep[OF q2])
qed

end

```

7 Setup for Isabelle's quotient package for terminated lazy lists

```

theory Quotient-TLLList imports
  TLLList
  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
by(simp add: fun-eq-iff tllist.map-id)

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
using Quotient3-abs-rep[OF q1] Quotient3-abs-rep[OF q2] Quotient3-abs-rep[OF
q3]

```

```

by(simp add: fun-eq-iff split: sum.split)

lemma case-sum-preserve2 [quot-preserve]:
  assumes q: Quotient3 R Abs Rep
  shows ((id ---> Rep) ---> (id ---> Rep) ---> id ---> Abs) case-sum
  = case-sum
  using Quotient3-abs-rep[OF q]
  by(auto intro!: ext split: sum.split)

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
  using Quotient3-abs-rep[OF q1] Quotient3-abs-rep[OF q2] Quotient3-abs-rep[OF q3]
  by(simp add: fun-eq-iff split: prod.split)

lemma case-prod-preserve2 [quot-preserve]:
  assumes q: Quotient3 R Abs Rep
  shows ((id ---> id ---> Rep) ---> id ---> Abs) case-prod = case-prod
  using Quotient3-abs-rep[OF q]
  by(auto intro!: ext)

lemma id-preserve [quot-preserve]:
  assumes Quotient3 R Abs Rep
  shows (Rep ---> Abs) id = id
  using Quotient3-abs-rep[OF assms]
  by(auto intro: ext)

functor tmap: tmap
  by(simp-all add: fun-eq-iff tmap-id-id tllist.map-comp)

lemma reflp-tllist-all2:
  assumes R: reflp R and Q: reflp Q
  shows reflp (tllist-all2 R Q)
proof(rule reflpI)
  fix xs
  show tllist-all2 R Q xs xs
    apply(coinduction arbitrary: xs)
    using assms by(auto elim: reflpE)
qed

lemma symp-tllist-all2: [[ symp R; symp S ]] ==> symp (tllist-all2 R S)
by (rule sympI)(auto 4 3 simp add: tllist-all2-conv-all-tnth elim: sympE dest: lfinite-llength-enat not-lfinite-llength)

lemma transp-tllist-all2: [[ transp R; transp S ]] ==> transp (tllist-all2 R S)

```

```

by (rule transpI) (rule tllist-all2-trans)

lemma tllist-equivp [quot-equiv]:
  [] equivp R; equivp S [] ==> equivp (tllist-all2 R S)
  by (simp add: equivp-reflp-symp-transp reflp-tllist-all2 symp-tllist-all2 transp-tllist-all2)

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
  using Quotient3-abs-rep[OF q1] Quotient3-abs-rep[OF q2] Quotient3-abs-rep[OF
  q3] Quotient3-abs-rep[OF q4]
  by(simp-all add: fun-eq-iff tllist.map-comp o-def)

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
  by(drule abs-o-rep)+(simp add: tllist.map-comp tmap-id-id)

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)
  by(coinduction arbitrary: ts)(auto simp add: Quotient3-rep-reflp[OF q1] Quotient3-rep-reflp[OF
  q2])

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(intro iffI conjI)
    assume ?lhs
    thus tllist-all2 R1 R2 r r
      apply -
      apply(rule tllist-all2-reflI)
      apply(clarsimp simp add: in-tset-conv-tnth)
      apply(metis tllist-all2-tnthD Quotient3-rel [OF q1])
      apply(metis tllist-all2-tfinite1-terminalD Quotient3-rel [OF q2])
  qed

```

```

done

from ‹?lhs› show tllist-all2 R1 R2 s s
  apply –
  apply(rule tllist-all2-refI)
  apply(clar simp simp add: in-tset-conv-tnth)
  apply(metis tllist-all2-tnthD2 Quotient3-rel [OF q1])
  apply(metis tllist-all2-tfinite2-terminalD Quotient3-rel [OF q2])
  done

from ‹?lhs› show tmap Abs1 Abs2 r = tmap Abs1 Abs2 s
  unfolding tmap-eq-tmap-conv-tllist-all2
  by(rule tllist-all2-mono)(metis Quotient3-rel[OF q1] Quotient3-rel[OF q2])+
next
  assume ?rhs
  thus ?lhs
    unfolding tmap-eq-tmap-conv-tllist-all2
    apply(clar simp simp add: tllist-all2-conv-all-tnth)
    apply(subst Quotient3-rel[OF q1, symmetric])
    apply(subst Quotient3-rel[OF q2, symmetric])
    apply(auto 4 3 dest: lfinite-llength-enat not-lfinite-llength)
    done
  qed

lemma tllist-quotient [quot-thm]:
  ‹[ Quotient3 R1 Abs1 Rep1; Quotient3 R2 Abs2 Rep2 ]
  ⟹ Quotient3 (tllist-all2 R1 R2) (tmap Abs1 Abs2) (tmap Rep1 Rep2)›
by(blast intro: Quotient3I dest: Quotient3-tmap-Abs-Rep Quotient3-tllist-all2-tmap-tmapI
tllist-all2-rel)

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
using Quotient3-abs-rep[OF q1] Quotient3-abs-rep[OF q2]
by(simp add: fun-eq-iff tllist.map-comp o-def tmap-id-id[unfolded id-def])

lemmas TCons-respect [quot-respect] = TCons-transfer2

lemma TNil-preserve [quot-preserve]:
  assumes Quotient3 R2 Abs2 Rep2
  shows (Rep2 --> tmap Abs1 Abs2) TNil = TNil
using Quotient3-abs-rep[OF assms]
by(simp add: fun-eq-iff)

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
  assume ?lhs
  thus ?rhs
  proof(coinduct)
    case (tllist-all2 ts ts')
    thus ?case using Quotient3-abs-rep[OF q1] Quotient3-abs-rep[OF q2]
      by(cases ts)(case-tac [|] ts', auto simp add: tllist-all2-TNil1 tllist-all2-TCons1)
  qed
next
  assume ?rhs
  thus ?lhs
    apply(coinduction arbitrary: ts ts')
    using Quotient3-abs-rep[OF q1] Quotient3-abs-rep[OF q2]
    by(auto dest: tllist-all2-is-TNilD intro: tllist-all2-tfinite1-terminalD tllist-all2-thdD
tllist-all2-ttlI)
  qed

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
by(simp add: fun-eq-iff tllist-all2-prs[OF assms])

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)) = (=)
  by (simp add: fun-eq-iff map-fun-def comp-def Quotient3-rel-rep[OF q1] Quo-
tient3-rel-rep[OF q2]
tllist-all2-eq)

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(intro ext)
fix IS-TNIL TNIL THD endORmore TTL-end TTL-more b
show ?lhs IS-TNIL TNIL THD endORmore TTL-end TTL-more b = ?rhs
IS-TNIL TNIL THD endORmore TTL-end TTL-more b
by(coinduction arbitrary: b rule: tllist.coinduct-strong)(auto simp add: Quo-
tient3-abs-rep[OF q1] Quotient3-abs-rep[OF q2] Quotient3-abs-rep[OF q3] Quo-
tient3-tmap-Abs-Rep[OF q3 q2])
qed

end

theory Coinductive imports
  Coinductive-List-Prefix
  Coinductive-Stream
  TLLList
  Quotient-Coinductive-List
  Quotient-TLLList
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*'
 $[[\text{code drop: } \text{partial-term-of} :: - \text{ llist itself} \Rightarrow -]]$

lemma *partial-term-of-llist-code* [*code*]:
fixes *tytok* :: '*a* :: *partial-term-of llist itself* **shows**
 $\text{partial-term-of } \text{tytok} (\text{Quickcheck-Narrowing.Narrowing-variable } p \text{ tt}) \equiv$
 $\text{Code-Evaluation.Free } (\text{STR } "-") (\text{Typerep.typerep } \text{TYPE}'a \text{ llist})$
 $\text{partial-term-of } \text{tytok} (\text{Quickcheck-Narrowing.Narrowing-constructor } 0 \text{ []}) \equiv$
 $\text{Code-Evaluation.Const } (\text{STR } "\text{Coinductive-List.llist.LNil}") (\text{Typerep.typerep } \text{TYPE}'a \text{ llist})$
 $\text{partial-term-of } \text{tytok} (\text{Quickcheck-Narrowing.Narrowing-constructor } 1 \text{ [head, tail]})$
 \equiv
 $\text{Code-Evaluation.App}$
 $(\text{Code-Evaluation.App}$
 $(\text{Code-Evaluation.Const}$
 $(\text{STR } "\text{Coinductive-List.llist.LCons}")$
 $(\text{Typerep.typerep } \text{TYPE}'a \Rightarrow 'a \text{ llist} \Rightarrow 'a \text{ llist}))$
 $(\text{partial-term-of } \text{TYPE}'a \text{ head})$
 $(\text{partial-term-of } \text{TYPE}'a \text{ llist} \text{ tail})$
by—(*rule partial-term-of-anything*)+

declare *option.splits* [*split*]

lemma *Lazy-llist-inject* [*simp*]:
 $\text{Lazy-llist } xs = \text{Lazy-llist } ys \longleftrightarrow xs = ys$
by(*auto simp add: fun-eq-iff*)

lemma *Lazy-llist-inverse* [*code, simp*]:
 $\text{force } (\text{Lazy-llist } xs) = xs ()$
by(*auto*)

lemma *force-inverse* [*simp*]:
 $\text{Lazy-llist } (\lambda _. \text{force } xs) = xs$
by(*auto split: llist.split*)

lemma *LNil-Lazy-llist* [*code*]: *LNil* = *Lazy-llist* ($\lambda _. \text{None}$)
by(*simp*)

lemma *LCons-Lazy-llist* [*code, code-unfold*]: *LCons* *x* *xs* = *Lazy-llist* ($\lambda _. \text{Some } (x, xs)$)
by *simp*

lemma *lnull-lazy* [*code*]: *lnull* = *Option.is-none* \circ *force*
unfolding *lnull-def*
by (*rule ext*) (*simp add: Option.is-none-def split: llist.split*)

```

declare [[code drop: equal-class.equal :: 'a :: equal llist  $\Rightarrow$  -]]

lemma equal-llist-Lazy-list [code]:
  equal-class.equal (Lazy-list xs) (Lazy-list ys)  $\longleftrightarrow$ 
  (case xs () of None  $\Rightarrow$  (case ys () of None  $\Rightarrow$  True | -  $\Rightarrow$  False)
  | Some (x, xs')  $\Rightarrow$ 
    (case ys () of None  $\Rightarrow$  False
    | Some (y, ys')  $\Rightarrow$  if x = y then equal-class.equal xs' ys' else False))
by(auto simp add: equal-llist-def)

declare [[code drop: corec-llist]]

lemma corec-llist-Lazy-list [code]:
  corec-llist IS-LNIL LHD endORmore LTL-end LTL-more b =
  Lazy-list ( $\lambda$ . 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)))
by(subst llist.corec-code) simp

declare [[code drop: unfold-llist]]

lemma unfold-llist-Lazy-list [code]:
  unfold-llist IS-LNIL LHD LTL b =
  Lazy-list ( $\lambda$ . if IS-LNIL b then None else Some (LHD b, unfold-llist IS-LNIL
  LHD LTL (LTL b)))
by(subst unfold-llist.code) simp

declare [[code drop: case-llist]]

lemma case-llist-Lazy-list [code]:
  case-llist n c (Lazy-list xs) = (case xs () of None  $\Rightarrow$  n | Some (x, ys)  $\Rightarrow$  c x ys)
by simp

declare [[code drop: lappend]]

lemma lappend-Lazy-list [code]:
  lappend (Lazy-list xs) ys =
  Lazy-list ( $\lambda$ . case xs () of None  $\Rightarrow$  force ys | Some (x, xs')  $\Rightarrow$  Some (x, lappend
  xs' ys))
by(auto split: llist.splits)

declare [[code drop: lmap]]

lemma lmap-Lazy-list [code]:
  lmap f (Lazy-list xs) = Lazy-list ( $\lambda$ . map-option (map-prod f (lmap f)) (xs ()))
by simp

declare [[code drop: lfinite]]

```

```

lemma lfinite-Lazy-llist [code]:
  lfinite (Lazy-llist xs) = (case xs () of None ⇒ True | Some (x, ys) ⇒ lfinite ys)
by simp

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)
by(simp add: list-of-aux-code)

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)
by(simp add: gen-llength-code)

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))
by(cases n rule: enat-coexhaust) auto

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))
by(cases n)(auto simp add: eSuc-enat[symmetric] split: llist.split)

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)
by auto

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))
by(auto split: llist.split)

declare [[code drop: lzip]]

lemma lzip-Lazy-list [code]:
lzip (Lazy-list xs) (Lazy-list ys) =
Lazy-list (λ-. Option.bind (xs ()) (λ(x, xs'). map-option (λ(y, ys'). ((x, y), lzip
xs' ys')) (ys ()))))
by auto

declare [[code drop: gen-lset]]

lemma lset-Lazy-list [code]:
gen-lset A (Lazy-list xs) =
(case xs () of None ⇒ A | Some (y, ys) ⇒ gen-lset (insert y A) ys)
by(auto simp add: gen-lset-code)

declare [[code drop: lmember]]

lemma lmember-Lazy-list [code]:
lmember x (Lazy-list xs) =
(case xs () of None ⇒ False | Some (y, ys) ⇒ x = y ∨ lmember x ys)
by(simp add: lmember-def)

declare [[code drop: llist-all2]]

lemma llist-all2-Lazy-list [code]:
llist-all2 P (Lazy-list xs) (Lazy-list 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'))
by auto

declare [[code drop: lhd]]

lemma lhd-Lazy-list [code]:
lhd (Lazy-list xs) = (case xs () of None ⇒ undefined | Some (x, xs') ⇒ x)
by(simp add: lhd-def)

declare [[code drop: ltl]]

lemma ltl-Lazy-list [code]:
ltl (Lazy-list xs) = Lazy-list (λ-. case xs () of None ⇒ None | Some (x, ys) ⇒
force ys)
by(auto split: llist.split)

declare [[code drop: llast]]

```

```

lemma llast-Lazy-llist [code]:
  llast (Lazy-llist xs) =
    (case xs () of
      None  $\Rightarrow$  undefined
      | Some (x, xs')  $\Rightarrow$ 
        (case force xs' of None  $\Rightarrow$  x | Some (x', xs'')  $\Rightarrow$  llast (LCons x' xs''))
  by(auto simp add: llast-def zero-enat-def eSuc-def split: enat.split llist.splits)

declare [[code drop: ldistinct]]

lemma ld़istinct-Lazy-llist [code]:
  ld़istinct (Lazy-llist xs) =
    (case xs () of None  $\Rightarrow$  True | Some (x, ys)  $\Rightarrow$  x  $\notin$  lset ys  $\wedge$  ld़istinct ys)
  by(auto)

declare [[code drop: lprefix]]

lemma lprefix-Lazy-llist [code]:
  lprefix (Lazy-llist xs) (Lazy-llist ys) =
    (case xs () of
      None  $\Rightarrow$  True
      | Some (x, xs')  $\Rightarrow$ 
        (case ys () of None  $\Rightarrow$  False | Some (y, ys')  $\Rightarrow$  x = y  $\wedge$  lprefix xs' ys')
  by auto

declare [[code drop: lstrict-prefix]]

lemma lstrict-prefix-Lazy-llist [code]:
  lstrict-prefix (Lazy-llist xs) (Lazy-llist ys)  $\longleftrightarrow$ 
    (case ys () of
      None  $\Rightarrow$  False
      | Some (y, ys')  $\Rightarrow$ 
        (case xs () of None  $\Rightarrow$  True | Some (x, xs')  $\Rightarrow$  x = y  $\wedge$  lstrict-prefix xs' ys')
  by auto

declare [[code drop: llcp]]

lemma llcp-Lazy-llist [code]:
  llcp (Lazy-llist xs) (Lazy-llist ys) =
    (case xs () of None  $\Rightarrow$  0
     | Some (x, xs')  $\Rightarrow$  (case ys () of None  $\Rightarrow$  0
       | Some (y, ys')  $\Rightarrow$  if x = y then eSuc (llcp xs' ys') else 0))
  by auto

declare [[code drop: llexord]]

lemma llexord-Lazy-llist [code]:
  llexord r (Lazy-llist xs) (Lazy-llist ys)  $\longleftrightarrow$ 
    (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'())
by auto

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))
by(auto split: llist.split)

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')))
by(auto split: llist.split)

declare option.splits [split del]
declare Lazy-llist-def [simp del]

Simple ML test for laziness

ML-val ‹
val zeros = @{code iterates} (fn x => x + 1) 0;
val lhd = @{code lhd} zeros;
val ltl = @{code ltl} zeros;
val ltl' = @{code force} ltl;

val ltake = @{code ltake} (@{code eSuc} (@{code eSuc} @{code 0::enat})) zeros;
val ldrop = @{code ldrop} (@{code eSuc} @{code 0::enat}) zeros;
val list-of = @{code list-of} ltake;

val ltakeWhile = @{code ltakeWhile} (fn _ => true) zeros;
val ldropWhile = @{code ldropWhile} (fn _ => false) zeros;
val hd = @{code lhd} ldropWhile;

val lfilter = @{code lfilter} (fn _ => false) zeros;
›

hide-const (open) force

end

```

9 Code generator setup to implement terminated lazy lists lazily

```

theory Lazy-TLLList imports
  TTLList
  Lazy-LList
begin

code-identifier code-module Lazy-TLLList  $\rightarrow$ 
  (SML) TTLList and
  (OCaml) TTLList and
  (Haskell) TTLList and
  (Scala) TTLList

definition Lazy-tllist :: (unit  $\Rightarrow$  'a  $\times$  ('a, 'b) tllist + 'b)  $\Rightarrow$  ('a, 'b) tllist
where [code del]:
  Lazy-tllist xs = (case xs () of Inl (x, ys)  $\Rightarrow$  TCons x ys | Inr b  $\Rightarrow$  TNil b)

definition force :: ('a, 'b) tllist  $\Rightarrow$  'a  $\times$  ('a, 'b) tllist + 'b
where [simp, code del]: force xs = (case xs of TNil b  $\Rightarrow$  Inr b | TCons x ys  $\Rightarrow$  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  $\Rightarrow$  -]]

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)  $\equiv$ 
      Code-Evaluation.Free (STR "-") (Typerep.typerep TYPE('a, 'b) tllist))
    partial-term-of tytok (Quickcheck-Narrowing.Narrowing-constructor 0 [b])  $\equiv$ 
      Code-Evaluation.App
      (Code-Evaluation.Const (STR "TLLList.tllist.TNil") (Typerep.typerep TYPE('b
       $\Rightarrow$  ('a, 'b) tllist)))
      (partial-term-of TYPE('b) b)
    partial-term-of tytok (Quickcheck-Narrowing.Narrowing-constructor 1 [head, tail])
     $\equiv$ 
      Code-Evaluation.App
      (Code-Evaluation.App
        (Code-Evaluation.Const
          (STR "TLLList.tllist.TCons")
          (Typerep.typerep TYPE('a  $\Rightarrow$  ('a, 'b) tllist  $\Rightarrow$  ('a, 'b) tllist)))
        (partial-term-of TYPE('a) head))
      (partial-term-of TYPE('a, 'b) tllist) tail)
  by-(rule partial-term-of-anything)+

declare Lazy-tllist-def [simp]

```

```

declare sum.splits [split]

lemma TNil-Lazy-tllist [code]:
  TNil b = Lazy-tllist ( $\lambda\_. \text{Inr } b$ )
by simp

lemma TCons-Lazy-tllist [code, code-unfold]:
  TCons x xs = Lazy-tllist ( $\lambda\_. \text{Inl } (x, xs)$ )
by simp

lemma Lazy-tllist-inverse [simp, code]:
  force (Lazy-tllist xs) = xs ()
by(simp)

declare [[code drop: equal-class.equal :: (-, -) tllist  $\Rightarrow$  -]]

lemma equal-tllist-Lazy-tllist [code]:
  equal-class.equal (Lazy-tllist xs) (Lazy-tllist ys) =
  (case xs () of
    Inr b  $\Rightarrow$  (case ys () of Inr b'  $\Rightarrow$  b = b' | -  $\Rightarrow$  False)
    | Inl (x, xs')  $\Rightarrow$ 
      (case ys () of Inr b'  $\Rightarrow$  False | Inl (y, ys')  $\Rightarrow$  if x = y then equal-class.equal xs' ys' else False))
by(auto simp add: equal-tllist-def)

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)  $\longleftrightarrow$ 
  (case xs () of Inl -  $\Rightarrow$  False | Inr -  $\Rightarrow$  True)
by simp

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
  ( $\lambda\_. \text{if IS-TNIL } b \text{ then Inr (TNIL } b\text{)} \text{ else Inl (THD } b, \text{ if endORmore } b \text{ then TTL-end } b \text{ else corec-tllist IS-TNIL }$ 
  TNIL THD endORmore TTL-end TTL-more (TTL-more b)))  

by(rule tllist.expand) simp-all

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)))
by(rule tllist.expand) auto

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)
  by simp

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))
  by(simp add: Lazy-llist-def split: option.splits)

declare [[code drop: terminal]]

lemma terminal-Lazy-tllist [code]:
  terminal (Lazy-tllist xs) =
    (case xs () of Inl (-, ys) ⇒ terminal ys | Inr b ⇒ b)
  by simp

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))
  by simp

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))
  by(auto split: tllist.split)

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))

```

```

by(auto simp add: Lazy-tllist-def split: option.split tllist.split)

declare [[code drop: TLLList.tfilter']]

lemma tfilter'-Lazy-tllist [code]:
  TLLList.tfilter' b P (Lazy-tllist xs) =
    Lazy-tllist (λ-. case xs () of Inl (x, xs') ⇒ if P x then Inl (x, TLLList.tfilter' b P
    xs') else force (TLLList.tfilter' b P xs') | Inr b' ⇒ Inr b')
  by(simp split: tllist.split)

declare [[code drop: TLLList.tconcat']]

lemma tconcat-Lazy-tllist [code]:
  TLLList.tconcat' b (Lazy-tllist xss) =
    Lazy-tllist (λ-. case xss () of Inr b' ⇒ Inr b' | Inl (xs, xss') ⇒ force (lappendt xs
    (TLLList.tconcat' b xss'))))
  by(simp split: tllist.split)

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'))
  by(simp add: tllist-all2-TNil1 tllist-all2-TNil2)

declare [[code drop: llist-of-tllist]]

lemma llist-of-tllist-Lazy-tllist [code]:
  llist-of-tllist (Lazy-tllist xs) =
    Lazy-tllist (λ-. case xs () of Inl (x, ys) ⇒ Some (x, llist-of-tllist ys) | Inr b ⇒
    None)
  by(simp add: Lazy-tllist-def)

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))
  by(cases n)(auto simp add: tnth-TNil)

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')

```

```

by(simp add: gen-tlength-code)

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'))
  by(cases n)(auto split: tllist.split)

declare Lazy-tllist-def [simp del]
declare sum.splits [split del]

Simple ML test for laziness

ML-val <
  val zeros = @{code unfold-tllist} (K false) (K 0) (K 0) I ();
  val thd = @{code thd} zeros;
  val ttl = @{code ttl} zeros;
  val ttl' = @{code force} ttl;

  val tdropn = @{code tdropn} (@{code Suc} @{code 0::nat}) zeros;
  val tfilter = @{code tfilter} 1 (K false) zeros;
>

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)
  by auto

lemma dropWhile-False: (∀x. x ∈ set xs ⇒ P x) ⇒ dropWhile P xs = []
  by simp

abbreviation (in order) chain ≡ Complete-Partial-Order.chain (≤)

lemma (in linorder) chain-linorder: chain C
  by (simp add: chain-def linear)

```

```

lemma continuous-add-ereal:
  assumes  $0 \leq t$ 
  shows continuous-on  $\{-\infty::\text{ereal} <..\}$   $(\lambda x. t + x)$ 
proof (subst continuous-on-open-vimage, (intro open-greaterThan allI impI)+)
  fix  $B :: \text{ereal set}$  assume open  $B$ 
  show open  $((\lambda x. t + x) -` B \cap \{-\infty <..\})$ 
  proof (cases  $t$ )
    case (real  $t'$ )
      then have  $*: (\lambda x. t + x) -` B \cap \{-\infty <..\} = (\lambda x. 1 * x + (-t)) ` (B \cap \{-\infty <..\})$ 
      apply (simp add: set-eq-iff image-iff Bex-def)
      apply (intro allI iffI)
      apply (rule-tac  $x = x + \text{ereal } t'$  in exI)
      apply (case-tac  $x$ )
      apply (auto simp: ac-simps)
      done
    show ?thesis
    unfolding *
    apply (rule ereal-open-affinity-pos)
    using ‹open B›
    apply (auto simp: real)
    done
  qed (insert ‹ $0 \leq tqed$ 
```

```

lemma tendsto-add-ereal:
   $0 \leq x \implies 0 \leq y \implies (f \longrightarrow y) F \implies ((\lambda z. x + f z :: \text{ereal}) \longrightarrow x + y) F$ 
  apply (rule tendsto-compose[where  $f=f$ ])
  using continuous-add-ereal[where  $t=x$ ]
  unfolding continuous-on-def
  apply (auto simp add: at-within-open[where  $S = \{-\infty <..\}]$ )
  done

```

```

lemma tendsto-LimI:  $(f \longrightarrow y) F \implies (f \longrightarrow \text{Lim } F f) F$ 
  by (metis tendsto-Lim tendsto-bot)

```

10.1 The filter at'

abbreviation (in ccpo) compact-element \equiv ccpo.compact Sup (\leq)

```

lemma tendsto-unique-eventually:
  fixes  $x x' :: 'a :: \text{t2-space}$ 
  shows  $F \neq \text{bot} \implies \text{eventually } (\lambda x. f x = g x) F \implies (f \longrightarrow x) F \implies (g \longrightarrow x') F \implies x = x'$ 
  by (metis tendsto-unique filterlim-cong)

```

```

lemma (in ccpo) ccpo-Sup-upper2: chain  $C \implies x \in C \implies y \leq x \implies y \leq \text{Sup } C$ 
  by (blast intro: ccpo-Sup-upper order-trans)

```

```

lemma tendsto-open-vimage: ( $\bigwedge B. \text{open } B \implies \text{open } (f -` B)) \implies f -l\rightarrow f l$ 
  using continuous-on-open-vimage[of UNIV f] continuous-on-def[of UNIV f] by
  simp

lemma open-vimageI: ( $\bigwedge x. f -x\rightarrow f x) \implies \text{open } A \implies \text{open } (f -` A)$ 
  using continuous-on-open-vimage[of UNIV f] continuous-on-def[of UNIV f] by
  simp

lemma principal-bot: principal x = bot  $\longleftrightarrow$  x = {}
  by (auto simp: filter-eq-iff eventually-principal)

definition at' x = (if open {x} then principal {x} else at x)

lemma at'-bot: at' x  $\neq$  bot
  by (simp add: at'-def at-eq-bot-iff principal-bot)

lemma tendsto-id-at'[simp, intro]: (( $\lambda x. x) \longrightarrow x) (at' x)
  by (simp add: at'-def topological-tendstoI eventually-principal tendsto-ident-at)

lemma cont-at': (f  $\longrightarrow$  f x) (at' x)  $\longleftrightarrow$  f -x $\rightarrow$  f x
  using at-eq-bot-iff[of x] by (auto split: if-split-asm intro!: topological-tendstoI
  simp: eventually-principal at'-def)$ 
```

10.2 The type class *ccpo-topology*

Temporarily relax type constraints for *open*.

```

setup <Sign.add-const-constraint
  (@{const-name open}, SOME @{typ 'a::open set  $\Rightarrow$  bool})>

class ccpo-topology = open + ccpo +
  assumes open-ccpo: open A  $\longleftrightarrow$  ( $\forall C. \text{chain } C \longrightarrow C \neq \{\} \longrightarrow \text{Sup } C \in A \longrightarrow C \cap A \neq \{\}$ )
  begin

lemma open-ccpoD:
  assumes open A chain C C  $\neq \{\} \text{ Sup } C \in A$ 
  shows  $\exists c \in C. \forall c' \in C. c \leq c' \longrightarrow c' \in A$ 
  proof (rule ccontr)
    assume  $\neg ?thesis$ 
    then have *:  $\bigwedge c. c \in C \implies \exists c' \in C. c \leq c' \wedge c' \notin A$ 
    by auto
    with <chain C> <C  $\neq \{\}$ > have chain (C - A) C - A  $\neq \{\}$ 
    by (auto intro: chain-Diff)
    moreover have Sup C = Sup (C - A)
    proof (safe intro!: order.antisym ccpo-Sup-least <chain C> chain-Diff)
      fix c assume c  $\in C$ 
      with * obtain c' where c'  $\in C. c \leq c' \wedge c' \notin A$ 
      by auto

```

```

with `c ∈ C` show c ≤ ⋃(C - A)
  by (intro ccpo-Sup-upper2 `chain (C - A)` auto)
qed (auto intro: `chain C` ccpo-Sup-upper)
ultimately show False
  using `open A` `Sup C ∈ A` by (auto simp: open-ccpo)
qed

lemma open-ccpo-Iic: open {.. b}
  by (auto simp: open-ccpo) (metis Int-iff atMost-iff ccpo-Sup-upper empty-iff or-
der-trans)

subclass topological-space
proof
  show open (UNIV::'a set)
    unfolding open-ccpo by auto
next
  fix S T :: 'a set assume open S open T
  show open (S ∩ T)
    unfolding open-ccpo
  proof (intro allI impI)
    fix C assume chain C C ≠ {} and ⋃ C ∈ S ∩ T
    with open-ccpoD[OF `open S` C] open-ccpoD[OF `open T` C]
    show C ∩ (S ∩ T) ≠ {}
      unfolding chain-def by blast
  qed
next
  fix K :: 'a set set assume *: ∀ D ∈ K. open D
  show open (⋃ K)
    unfolding open-ccpo
  proof (intro allI impI)
    fix C assume chain C C ≠ {} ⋃ C ∈ (⋃ K)
    with * obtain D where D ∈ K ⋃ C ∈ D C ∩ D ≠ {}
      by (auto simp: open-ccpo)
    then show C ∩ (⋃ K) ≠ {}
      by auto
  qed
qed

lemma closed-ccpo: closed A ↔ (∀ C. chain C → C ≠ {} → C ⊆ A → Sup
C ∈ A)
  unfolding closed-def open-ccpo by auto

lemma closed-admissible: closed {x. P x} ↔ ccpo.admissible Sup (≤) P
  unfolding closed-ccpo ccpo.admissible-def by auto

lemma open-singletonI-compact: compact-element x ==> open {x}
  using admissible-compact-neq[of Sup (≤) x]
  by (simp add: closed-admissible[symmetric] open-closed Collect-neg-eq)

```

```

lemma closed-Ici: closed {.. b}
  by (auto simp: closed-ccpo intro: ccpo-Sup-least)

lemma closed-Iic: closed {b ..}
  by (auto simp: closed-ccpo intro: ccpo-Sup-upper2)

ccpo-topologys are also t2-spaces. This is necessary to have a unique continuous extension.

subclass t2-space
proof
  fix x y :: 'a assume x ≠ y
  show ∃ U V. open U ∧ open V ∧ x ∈ U ∧ y ∈ V ∧ U ∩ V = {}
  proof cases
    { fix x y assume x ≠ y x ≤ y
      then have open {..x} ∧ open (− {..x}) ∧ x ∈ {..x} ∧ y ∈ − {..x} ∧ {..x} ∩ − {..x} = {}
      by (auto intro: open-ccpo-Iic closed-Ici) }
    moreover assume x ≤ y ∨ y ≤ x
    ultimately show ?thesis
      using ⟨x ≠ y⟩ by (metis Int-commute)
  next
    assume ¬ (x ≤ y ∨ y ≤ x)
    then have open ( {..x} ∩ − {..y}) ∧ open ( {..y} ∩ − {..x}) ∧
      x ∈ {..x} ∩ − {..y} ∧ y ∈ {..y} ∩ − {..x} ∧ ( {..x} ∩ − {..y}) ∩ ( {..y} ∩ − {..x}) = {}
    by (auto intro: open-ccpo-Iic closed-Ici)
    then show ?thesis by auto
  qed
qed

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 (λx. g x ≤ f x) F
  shows y ≤ x
  proof (rule ccontr)
    assume ¬ y ≤ x

    show False
    proof cases
      assume x ≤ y
      with ⟨¬ y ≤ x⟩
      have open {..x} open (− {..x}) x ∈ {..x} y ∈ − {..x} {..x} ∩ − {..x} = {}
      by (auto intro: open-ccpo-Iic closed-Ici)
      with topological-tendstoD[OF x, of {..x}] topological-tendstoD[OF y, of − {..x}]
      have eventually (λz. f z ≤ x) F eventually (λz. ¬ g z ≤ x) F

```

```

    by auto
  with ev have eventually ( $\lambda x. \text{False}$ ) F by eventually-elim (auto intro: order-trans)
  with F show False by (auto simp: eventually-False)
next
  assume  $\neg x \leq y$ 
  with  $\neg y \leq x$  have open ( $\{\dots x\} \cap -\{\dots y\}$ ) open ( $\{\dots y\} \cap -\{\dots x\}$ )
     $x \in \{\dots x\} \cap -\{\dots y\}$   $y \in \{\dots y\} \cap -\{\dots x\}$  ( $\{\dots x\} \cap -\{\dots y\}$ )  $\cap (\{\dots y\} \cap -\{\dots x\}) = \{\}$ 
    by (auto intro: open-ccpo-Iic closed-Ici)
  with topological-tendstoD[ $OF x$ , of  $\{\dots x\} \cap -\{\dots y\}$ ]
    topological-tendstoD[ $OF y$ , of  $\{\dots y\} \cap -\{\dots x\}$ ]
  have eventually ( $\lambda z. f z \leq x \wedge \neg f z \leq y$ ) F eventually ( $\lambda z. g z \leq y \wedge \neg g z \leq x$ ) F
    by auto
  with ev have eventually ( $\lambda x. \text{False}$ ) F by eventually-elim (auto intro: order-trans)
  with F show False by (auto simp: eventually-False)
qed
qed

```

```

lemma tendsto-ccpoI:
  fixes f :: 'a::ccpo-topology  $\Rightarrow$  'b::ccpo-topology
  shows ( $\bigwedge C. \text{chain } C \implies C \neq \{\} \implies \text{chain } (f' C) \wedge f(Sup C) = Sup(f'C)$ )
     $\implies f -x \rightarrow f x$ 
  by (intro tendsto-open-vimage) (auto simp: open-ccpo)

```

```

lemma tendsto-mcont:
  assumes mcont: mcont Sup ( $\leq$ ) Sup ( $\leq$ ) (f :: 'a :: ccpo-topology  $\Rightarrow$  'b :: ccpo-topology)
  shows f  $-l \rightarrow f l$ 
proof (intro tendsto-ccpoI conjI)
  fix C :: 'a set assume C: chain C  $C \neq \{\}$ 
  show chain (f'C)
    using mcont
    by (intro chain-imageI[where le-a= $(\leq)$ ] C) (simp add: mcont-def monotone-def)
    show f  $(\bigsqcup C) = \bigsqcup(f' C)$ 
    using mcont C by (simp add: mcont-def cont-def)
qed

```

10.3 Instances for ccpo-topologys and continuity theorems

```

instantiation set :: (type) ccpo-topology
begin

```

```

definition open-set :: 'a set set  $\Rightarrow$  bool where
  open-set A  $\longleftrightarrow$  ( $\forall C. \text{chain } C \longrightarrow C \neq \{\} \longrightarrow Sup C \in A \longrightarrow C \cap A \neq \{\}$ )

```

```

instance

```

```

by intro-classes (simp add: open-set-def)

end

instantiation enat :: ccpo-topology
begin

instance
proof
fix A :: enat set
show open A = ( $\forall C$ . chain  $C \rightarrow C \neq \{\} \rightarrow \bigsqcup C \in A \rightarrow C \cap A \neq \{\}$ )
proof (intro iffI allI impI)
fix C x assume open A chain C  $C \neq \{\} \bigsqcup C \in A$ 

show  $C \cap A \neq \{\}$ 
proof cases
assume  $\bigsqcup C = \infty$ 
with  $\langle \bigsqcup C \in A \rangle \langle \text{open } A \rangle$  obtain n where {enat n <..}  $\subseteq A$ 
unfolding open-enat-iff by auto
with  $\langle \bigsqcup C = \infty \rangle \text{ Sup-eq-top-iff}[of C]$  show ?thesis
by (auto simp: top-enat-def)
next
assume  $\bigsqcup C \neq \infty$ 
then obtain n where  $C \subseteq \{\dots \text{enat } n\}$ 
unfolding Sup-eq-top-iff top-enat-def[symmetric] by (auto simp: not-less
top-enat-def)
moreover have finite {\dots enat n}
by (auto intro: finite-enat-bounded)
ultimately have finite C
by (auto intro: finite-subset)
from in-chain-finite[OF chain C finite C C neq {}]  $\langle \bigsqcup C \in A \rangle$  show
?thesis
by auto
qed
next
assume C:  $\forall C$ . chain  $C \rightarrow C \neq \{\} \rightarrow \bigsqcup C \in A \rightarrow C \cap A \neq \{\}$ 
show open A
unfolding open-enat-iff
proof safe
assume  $\infty \in A$ 
{ fix C :: enat set assume infinite C
then have  $\bigsqcup C = \infty$ 
by (auto simp: Sup-enat-def)
with infinite C C[THEN spec, of C]  $\langle \infty \in A \rangle$  have  $C \cap A \neq \{\}$ 
by auto }
note inf-C = this

show  $\exists x. \{\text{enat } x <..\} \subseteq A$ 
proof (rule ccontr)

```

```

assume  $\neg (\exists x. \{enat\} x <.. \subseteq A)$ 
with  $\langle \infty \in A \rangle$  have  $\bigwedge x. \exists y > x. enat y \notin A$ 
    by (simp add: subset-eq Bex-def) (metis enat.exhaust enat-ord-simps(2))
then have infinite {n. enat n  $\notin A$ }
    unfolding infinite-nat-iff-unbounded by auto
then have infinite (enat ` {n. enat n  $\notin A$ })
    by (auto dest!: finite-imageD)
from inf-C[OF this] show False
    by auto
qed
qed
qed
qed

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-topology, complete-distrib-lattice})
by(simp add: isCont-def tendsto-inf2 tendsto-ident-at)

lemmas tendsto-sup1[THEN tendsto-compose, tendsto-intros] =
tendsto-mcont[OF mcont-sup1]

lemma isCont-If: isCont f x  $\implies$  isCont g x  $\implies$  isCont ( $\lambda x. if Q then f x else g x$ )
by (cases Q) auto

lemma isCont-enat-case: isCont (f (epred n)) x  $\implies$  isCont g x  $\implies$  isCont ( $\lambda x. co.case-enat (g x) (\lambda n. f n x) n$ ) x
by (cases n rule: enat-coexhaust) auto

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  $\Rightarrow$  'b::t2-space
  assumes f:  $\bigwedge x. isCont f x$  and g:  $\bigwedge x. isCont g x$ 
  shows closed {x. f x = g x}
  by (intro closed-Collect-eq continuous-at-imp-continuous-on ballI assms)

```

```

instantiation llist :: (type) ccpo-topology
begin

definition open-llist :: 'a llist set  $\Rightarrow$  bool where
  open-llist A  $\longleftrightarrow$  ( $\forall C.$  chain C  $\longrightarrow$  C  $\neq \{\}$   $\longrightarrow$  Sup C  $\in$  A  $\longrightarrow$  C  $\cap$  A  $\neq \{\}$ )

instance
  by intro-classes (simp add: open-llist-def)

end

```

11.1 Continuity and closedness of predefined constants

```

lemma tendsto-mcont-llist: mcont lSup lprefix lSup lprefix f  $\Longrightarrow$  f  $-l\rightarrow$  f l
  by (auto simp add: Sup-llist-def[abs-def] intro!: tendsto-mcont)

lemma tendsto-ltl[THEN tendsto-compose, tendsto-intros]: ltl  $-l\rightarrow$  ltl l
  by (intro tendsto-mcont-llist mcont-ltl)

lemma tendsto-lappend2[THEN tendsto-compose, tendsto-intros]: lappend l  $-l'\rightarrow$  lappend l l'
  by (intro tendsto-mcont-llist mcont-lappend2)

lemma tendsto-LCons[THEN tendsto-compose, tendsto-intros]: LCons x  $-l\rightarrow$  LCons x l
  by (intro tendsto-mcont-llist mcont-LCons)

lemma tendsto-lmap[THEN tendsto-compose, tendsto-intros]: lmap f  $-l\rightarrow$  lmap f l
  by (intro tendsto-mcont-llist mcont-lmap)

lemma tendsto-llength[THEN tendsto-compose, tendsto-intros]: llength  $-l\rightarrow$  llength l
  by (intro tendsto-mcont) (simp add: Sup-llist-def[abs-def])

lemma tendsto-lset[THEN tendsto-compose, tendsto-intros]: lset  $-l\rightarrow$  lset l
  by (rule tendsto-mcont)(simp add: Sup-llist-def[abs-def])

lemma open-lhd: open {l.  $\neg$  lnull l  $\wedge$  lhd l = x}
  unfolding open-ccpo set-eq-iff
proof (simp add: imp-conjL Sup-llist-def del: lhd-lSup, intro allI impI)
  fix C assume Complete-Partial-Order.chain lprefix C lhd (lSup C) = x
  moreover assume  $\exists c \in C.$   $\neg$  lnull c
  then obtain c where c  $\in$  C  $\neg$  lnull c
    by auto
  ultimately show  $\exists c.$  c  $\in$  C  $\wedge$   $\neg$  lnull c  $\wedge$  lhd c = x
    by (force simp: lhd-lSup-eq)
qed

```

```

lemma open-LCons': assumes A: open A shows open (LCons x ` A)
proof -
  have open (ltl -` A ∩ {l. ¬ lnull l ∧ lhd l = x})
    by (intro open-Int open-vimageI open-lhd A tends-to-intros)
  also have (ltl -` A ∩ {l. ¬ lnull l ∧ lhd l = x}) = LCons x ` A
    by force
  finally show ?thesis .
qed

lemma open-Ici: lfinite xs ==> open {xs ..}
proof (induct xs rule: lfinite.induct)
  case lfinite-LNil then show ?case
    by (simp add: atLeast-def)
next
  case (lfinite-LConsI xs x)
  moreover have {LCons x xs ..} = LCons x ` {xs ..}
    by (auto simp: LCons-lprefix-conv)
  ultimately show ?case
    by (auto intro: open-LCons')
qed

lemma open-lfinite[simp]: lfinite x ==> open {x}
proof (induct rule: lfinite.induct)
  show open {LNil}
    using open-ccpo-Iic[of LNil] by (simp add: atMost-def lnull-def)
qed (auto dest: open-LCons')

lemma open-singleton-iff-lfinite: open {x} <→ lfinite x
proof
  assume lfinite x then show open {x}
  unfolding compact-eq-lfinite[symmetric] Sup-llist-def[abs-def, symmetric] less-eq-llist-def[abs-def, symmetric]
    by (rule open-singletonI-compact)
next
  assume open {x}
  show lfinite x
  proof (rule ccontr)
    let ?C = {ys. lprefix ys x ∧ ys ≠ x}
    assume inf: ¬ lfinite x
    note lSup-strict-prefixes[OF this] ‹open {x}›
    moreover have chain ?C
      using lprefixes-chain[of x] by (auto dest: chain-compr)
    moreover have ?C ≠ {}
      using inf by (cases x) auto
    ultimately show False
      by (auto simp: open-ccpo Sup-llist-def)
  qed
qed

```

```

lemma closure-eq-lfinite:
  assumes closed-Q: closed {xs. Q xs}
  assumes downwards-Q:  $\bigwedge_{xs \in \text{closed}} Q xs \implies \text{lprefix } ys \in \text{closed} \implies Q ys$ 
  shows {xs. Q xs} = closure {xs. lfinite xs  $\wedge$  Q xs}
  proof (rule closure-unique[symmetric])
    fix T assume T: {xs. lfinite xs  $\wedge$  Q xs}  $\subseteq$  T and closed T

    show {xs. Q xs}  $\subseteq$  T
    proof clarify
      fix xs :: 'a llist
      let ?F = {ys. lprefix ys xs  $\wedge$  lfinite ys}
      assume Q xs
      with T downwards-Q have ?F  $\subseteq$  T
        by auto
      moreover have chain ?F ?F  $\neq \{\}$ 
        by (auto intro: lprefixes-chain chain-subset)
      moreover have lSup ?F = xs
        by (rule lSup-finite-prefixes)
      ultimately show xs  $\in$  T
        using <closed T> by (auto simp: closed-ccpo Sup-llist-def)
    qed
  qed (auto simp: closed-Q)

  lemma closure-lfinite: closure {xs. lfinite xs} = UNIV
  using closure-eq-lfinite[of λ_. True] by auto

  lemma closed-ldistinct: closed {xs. ldistinct xs}
  unfolding closed-ccpo by (auto simp: ldistinct-lSup Sup-llist-def subset-eq)

  lemma ldistinct-closure: {xs. ldistinct xs} = closure {xs. lfinite xs  $\wedge$  ldistinct xs}
  by (rule closure-eq-lfinite[OF closed-ldistinct ldistinct-lprefix])

  lemma closed-ldistinct': ( $\bigwedge x. \text{isCont } f x$ )  $\implies$  closed {xs. ldistinct (f xs)}
  using continuous-closed-vimage[of - f, OF closed-ldistinct] by auto

  lemma closed-lsorted: closed {xs. lsorted xs}
  unfolding closed-ccpo by (auto simp: lsorted-lSup Sup-llist-def subset-eq)

  lemma lsorted-closure: {xs. lsorted xs} = closure {xs. lfinite xs  $\wedge$  lsorted xs}
  by (rule closure-eq-lfinite[OF closed-lsorted lsorted-lprefixD])

  lemma closed-lsorted': ( $\bigwedge x. \text{isCont } f x$ )  $\implies$  closed {xs. lsorted (f xs)}
  using continuous-closed-vimage[of - f, OF closed-lsorted] by auto

  lemma closed-in-lset: closed {l. x  $\in$  lset l}
  unfolding closed-ccpo by (auto simp add: subset-eq lset-lSup Sup-llist-def)

  lemma closed-llist-all2:
  closed {(x, y). llist-all2 R x y}

```

```

proof -
{ fix a b assume *:  $\bigwedge A B. \text{open } A \implies \text{open } B \implies a \in A \implies b \in B \implies (\exists x \in A. \exists y \in B. \text{llist-all2 } R x y)$ 
then have llist-all2 R a b
proof (coinduction arbitrary: a b)
case LNil
from LNil[rule-format, of {LNil} - {LNil}] LNil[rule-format, of - {LNil}]
{LNil}]
show ?case
by (auto simp: closed-def[symmetric] lnull-def)
next
case LCons
show ?case
proof
show ?lhd
using LCons(1)[rule-format, OF open-lhd open-lhd, of lhd a lhd b]
LCons(2,3)
by (auto dest: llist-all2-lhdD)
show ?ltl
proof (rule, simp, safe)
fix A B assume open A open B ltl a ∈ A ltl b ∈ B
with LCons(1)[rule-format, OF open-LCons' open-LCons', of A B lhd a
lhd b] LCons(2,3)
show  $\exists a' \in A. \exists b' \in B. \text{llist-all2 } R a' b'$ 
by (auto simp: not-lnull-conv)
qed
qed
qed
then show ?thesis
unfolding closed-def open-prod-def
by (auto simp: subset-eq)
qed

lemma closed-list-all2:
fixes f g :: 'b::t2-space ⇒ 'a llist
assumes f:  $\bigwedge x. \text{isCont } f x$  and g:  $\bigwedge x. \text{isCont } g x$ 
shows closed {x. llist-all2 R (f x) (g x)}
using continuous-closed-vimage[OF closed-llist-all2 isCont-Pair[OF f g]] by
simp

lemma at-botI-lfinite[simp]: lfinite l ⇒ at l = bot
by (simp add: at-eq-bot-iff)

lemma at-eq-lfinite: at l = (if lfinite l then bot else at' l)
by (auto simp: at'-def open-singleton-iff-lfinite)

lemma eventually-lfinite: eventually lfinite (at' x)
apply (simp add: at'-def open-singleton-iff-lfinite eventually-principal eventually-at-topological)

```

```

apply (intro exI[of - {.. x}] impI conjI open-ccpo-Iic)
apply (auto simp: lstrict-prefix-def intro!: lstrict-prefix-lfinite1)
done

lemma eventually-nhds-llist:
eventually P (nhds l)  $\longleftrightarrow$  ( $\exists xs \leq l. l\text{finite } xs \wedge (\forall ys \geq xs. ys \leq l \longrightarrow P ys)$ )
unfolding eventually-nhds
proof safe
let ?F = {l'. lprefix l' l  $\wedge$  lfinite l'}
fix A assume open A l  $\in$  A  $\forall l \in A. P l$ 
moreover have chain ?F ?F  $\neq \{\}$ 
by (auto simp: chain-def dest: lprefix-down-linear)
moreover have Sup ?F = l
unfolding Sup-llist-def by (rule lSup-finite-prefixes)
ultimately have  $\exists xs. xs \in ?F \wedge (\forall ys \geq xs. ys \in ?F \longrightarrow P ys)$ 
using open-ccpoD[of A ?F] by auto
then show  $\exists xs \leq l. l\text{finite } xs \wedge (\forall ys \geq xs. ys \leq l \longrightarrow P ys)$ 
by (metis (lifting) `l  $\in A`  $\forall l \in A. P l` le-llist-conv-lprefix mem-Collect-eq
not-lfinite-lprefix-conv-eq)
next
fix xs assume xs  $\leq l$  lfinite xs  $\forall ys \geq xs. ys \leq l \longrightarrow P ys$ 
then show  $\exists S. open S \wedge l \in S \wedge Ball S P$ 
by (intro exI[of - {xs ..}  $\cap$  {.. l}] conjI open-Int open-Ici open-ccpo-Iic) auto
qed

lemma nhds-lfinite: lfinite l  $\Longrightarrow$  nhds l = principal {l}
unfolding filter-eq-iff eventually-principal eventually-nhds-llist
by (auto simp del: le-llist-conv-lprefix)

lemma eventually-at'-llist:
eventually P (at' l)  $\longleftrightarrow$  ( $\exists xs \leq l. l\text{finite } xs \wedge (\forall ys \geq xs. l\text{finite } ys \longrightarrow ys \leq l \longrightarrow P ys)$ )
proof cases
assume lfinite l
then show ?thesis
by (auto simp add: eventually-filtermap at'-def open-singleton-iff-lfinite
eventually-principal lfinite-eq-range-llist-of)
next
assume  $\neg l\text{finite } l$ 
then show ?thesis
by (auto simp: eventually-filtermap at'-def open-singleton-iff-lfinite
eventually-at-filter eventually-nhds-llist)
(metis not-lfinite-lprefix-conv-eq)
qed

lemma eventually-at'-llistI: ( $\bigwedge xs. l\text{finite } xs \Longrightarrow xs \leq l \Longrightarrow P xs$ )  $\Longrightarrow$  eventually P (at' l)
by (auto simp: eventually-at'-llist)$$ 
```

```

lemma Lim-at'-lfinite: lfinite xs  $\implies$  Lim (at' xs) f = f xs
  by (rule tendsto-Lim[OF at'-bot]) (auto simp add: at'-def topological-tendstoI
eventually-principal)

lemma filterlim-at'-list:
  ( $f \longrightarrow y$ ) (at' (x::'a llist))  $\implies$  f -x→ y
  unfolding at'-def by (auto split: if-split-asm simp: open-singleton-iff-lfinite)

lemma tendsto-mcont-llist': mcont lSup lprefix lSup lprefix f  $\implies$  (f  $\longrightarrow$  f x) (at'
(x :: 'a llist))
  by(auto simp add: at'-def nhds-lfinite[symmetric] open-singleton-iff-lfinite tend-
sto-at-iff-tendsto-nhds[symmetric] intro: tendsto-mcont-llist)

lemma tendsto-closed:
  assumes eq: closed {x. P x}
  assumes ev:  $\bigwedge ys$ . lfinite ys  $\implies$  ys  $\leq$  x  $\implies$  P ys
  shows P x
proof -
  have x  $\in$  {x. P x}
  proof (rule Lim-in-closed-set)
    show eventually ( $\lambda x$ . x  $\in$  {x. P x}) (at' x)
    unfolding eq using ev
    by (force intro!: eventually-at'-llistI)
  qed (rule assms tendsto-id-at' at'-bot)+
  then show ?thesis
    by simp
qed

lemma tendsto-Sup-at':
  fixes f :: 'a llist  $\Rightarrow$  'b::ccpo-topology
  assumes f:  $\bigwedge x y$ . x  $\leq$  y  $\implies$  lfinite x  $\implies$  lfinite y  $\implies$  f x  $\leq$  f y
  shows (f  $\longrightarrow$  (Sup (f'{xs. lfinite xs  $\wedge$  xs  $\leq$  l}))) (at' l)
  proof (rule topological-tendstoI)
    let ?F = {xs. lfinite xs  $\wedge$  xs  $\leq$  l}

    have ch-F: chain (f'{?F}) f'{?F}  $\neq$  {}
    by (rule chain-imageI[OF chain-subset, OF lprefixes-chain]) (auto simp: f)

    fix A assume A: open A Sup (f'{?F})  $\in$  A then show eventually ( $\lambda x$ . f x  $\in$  A)
(at' l)
    using open-ccpoD[OF - ch-F, OF A] by (auto simp: eventually-at'-llist f simp
del: le-llist-conv-lprefix)
  qed

lemma tendsto-Lim-at':
  fixes f :: 'a llist  $\Rightarrow$  'b::ccpo-topology
  assumes f:  $\bigwedge l$ . f l = Lim (at' l) f'
  assumes mono:  $\bigwedge x y$ . x  $\leq$  y  $\implies$  lfinite x  $\implies$  lfinite y  $\implies$  f' x  $\leq$  f' y

```

```

shows (f —> f l) (at' l)
unfolding f[abs-def]
apply (subst filterlim-cong[OF refl refl eventually-mono[OF eventually-lfinite
Lim-at'-lfinite]])
apply assumption
apply (rule tendsto-LimI[OF tendsto-Sup-at'[OF mono]])
apply assumption+
done

```

lemma *isCont-LCons*[THEN *isCont-o2[rotated]*]: *isCont* (*LCons* *x*) *l*
by (*simp add: isCont-def tendsto-LCons tendsto-ident-at*)

lemma *isCont-lmap*[THEN *isCont-o2[rotated]*]: *isCont* (*lmap* *f*) *l*
by (*simp add: isCont-def tendsto-lmap tendsto-ident-at*)

lemma *isCont-lappend*[THEN *isCont-o2[rotated]*]: *isCont* (*lappend* *xs*) *ys*
by (*simp add: isCont-def tendsto-lappend2 tendsto-ident-at*)

lemma *isCont-lset*[THEN *isCont-o2[rotated]*]: *isCont* *lset* *xs*
by (*simp add: isCont-def tendsto-lset tendsto-ident-at*)

11.2 Define *lfilter* as continuous extension

definition *lfilter'* *P* *l* = *Lim* (at' *l*) ($\lambda \text{xs}.$ *llist-of* (*filter* *P* (*list-of* *xs*)))

lemma *tendsto-lfilter*: (*lfilter'* *P* —> *lfilter'* *P* *xs*) (at' *xs*)
by (*rule tendsto-Lim-at'[OF lfilter'-def]*) (*auto simp add: lfinite-eq-range-llist-of*
less-eq-list-def prefix-def)

lemma *isCont-lfilter*[THEN *isCont-o2[rotated]*]: *isCont* (*lfilter'* *P*) *l*
by (*simp add: isCont-def filterlim-at'-list tendsto-lfilter*)

lemma *lfilter'-lfinite*[*simp*]: *lfinite* *xs* \implies *lfilter'* *P* *xs* = *llist-of* (*filter* *P* (*list-of* *xs*))
by (*simp add: lfilter'-def Lim-at'-lfinite*)

lemma *lfilter'-LNil*: *lfilter'* *P* *LNil* = *LNil*
by *simp*

lemma *lfilter'-LCons* [*simp*]: *lfilter'* *P* (*LCons* *a* *xs*) = (*if P a then LCons a (lfilter'* *P* *xs*) *else lfilter'* *P* *xs*)
by (*rule tendsto-closed[where x=xs, OF closed-Collect-eq-isCont]*)
(*auto intro!: isCont-lfilter isCont-LCons isCont-If*)

lemma *ldistinct-lfilter'*: *ldistinct* *l* \implies *ldistinct* (*lfilter'* *P* *l*)
by (*rule tendsto-closed[OF closed-ldistinct', OF isCont-lfilter]*)
(*auto intro!: distinct-filter dest: ldistinct-lprefix simp: lfinite-eq-range-llist-of*)

```

lemma lfilter'-lmap: lfilter' P (lmap f xs) = lmap f (lfilter' (P o f) xs)
  by (rule tendsto-closed[where x=xs, OF closed-Collect-eq-isCont])
    (auto simp add: filter-map comp-def intro!: isCont-lmap isCont-lfilter)

lemma lfilter'-lfilter': lfilter' P (lfilter' Q xs) = lfilter' (λx. Q x ∧ P x) xs
  by (rule tendsto-closed[where x=xs, OF closed-Collect-eq-isCont]) (auto intro!:
    isCont-lfilter)

lemma lfilter'-LNil-I[simp]: (∀x ∈ lset xs. ¬ P x) ⇒ lfilter' P xs = LNil
  by (rule tendsto-closed[where x=xs, OF closed-Collect-eq-isCont])
    (auto simp add: lfinite-eq-range-llist-of llist-of-eq-LNil-conv filter-empty-conv
      intro: isCont-lfilter dest!: lprefix-lsetD)

lemma lset-lfilter': lset (lfilter' P xs) = lset xs ∩ {x. P x}
  by (rule tendsto-closed[OF closed-Collect-eq-isCont])
    (auto 4 3 intro: isCont-lset isCont-lfilter isCont-inf2)

lemma lfilter'-eq-LNil-iff: lfilter' P xs = LNil ↔ (∀x ∈ lset xs. ¬ P x)
  using lset-lfilter'[of P xs] by auto

lemma lfilter'-eq-lfilter: lfilter' P xs = lfilter P xs
  using isCont-lfilter
  proof(rule tendsto-closed[where x=xs, OF closed-Collect-eq-isCont])
    fix ys :: 'a llist
    assume lfinite ys
    thus lfilter' P ys = lfilter P ys by induction simp-all
  qed(simp-all add: isCont-def tendsto-mcont-llist mcont-lfilter)

```

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)
  apply (rule tendsto-Lim-at'[OF lconcat'-def])
  apply (auto simp add: lfinite-eq-range-llist-of less-eq-list-def prefix-def)
  apply (induct-tac xa)
  apply simp-all
  done

```

```

lemma isCont-lconcat'[THEN isCont-o2[rotated]]: isCont lconcat' l
  by (simp add: isCont-def filterlim-at'-list tendsto-lconcat')

```

```

lemma lconcat'-lfinite[simp]: lfinite xs ⇒ lconcat' xs = foldr lappend (list-of xs)
  LNil
  by (simp add: lconcat'-def Lim-at'-lfinite)

```

```

lemma lconcat'-LNil: lconcat' LNil = LNil
  by simp

```

```

lemma lconcat'-LCons [simp]: lconcat' (LCons l xs) = lappend l (lconcat' xs)
  by (rule tendsto-closed[where x=xs, OF closed-Collect-eq-isCont])
    (auto intro!: isCont-lconcat' isCont-lappend isCont-LCons)

lemma lmap-lconcat: lmap f (lconcat' xss) = lconcat' (lmap (lmap f) (xss::'a llist))
  proof (rule tendsto-closed[where x=xss, OF closed-Collect-eq-isCont])
    fix xs :: 'a llist llist
    assume lfinite xs
    then show lmap f (lconcat' xs) = lconcat' (lmap (lmap f) xs)
      by (induct xs rule: lfinite.induct) (auto simp: lmap-lappend-distrib)
  qed (intro isCont-lconcat' isCont-lappend isCont-LCons continuous-ident isCont-lmap)+

lemmas tendsto-Sup[THEN tendsto-compose, tendsto-intros] =
  mcont-SUP[OF mcont-id' mcont-const, THEN tendsto-mcont]

lemma
  assumes fin:  $\forall xs \in lset xss. lfinite xs$ 
  shows lset (lconcat' xss) =  $(\bigcup_{xs \in lset xss} lset xs)$  (is ?lhs = ?rhs)
  proof(rule tendsto-unique-eventually[OF at'-bot])
    show eventually  $(\lambda x. lset (lconcat' x)) = (\bigcup_{y \in lset x. lset y})$  (at' xss)
  proof(rule eventually-at'-llistI)
    fix xss'
    assume lfinite xss' xss'  $\leq$  xss
    hence  $\forall xs \in lset xss'. lfinite xs$  using fin by (auto dest!: lprefix-lsetD)
    with  $\langle lfinite xss' \rangle$  show lset (lconcat' xss') =  $(\bigcup_{xs \in lset xss'. lset xs})$ 
      by (induct xss') auto
  qed
  qed (rule tendsto-intros tendsto-lconcat' tendsto-id-at')+
```

11.4 Define ldropWhile as continuous extension

definition ldropWhile' P xs = Lim (at' xs) ($\lambda xs. llist-of (dropWhile P (list-of xs))$)

lemma tendsto-ldropWhile':
 $(ldropWhile' P \longrightarrow ldropWhile' P xs)$ (at' xs)
 by (rule tendsto-Lim-at'[OF ldropWhile'-def])
 (auto simp add: lfinite-eq-range-llist-of less-eq-list-def prefix-def dropWhile-append dropWhile-False)

lemma isCont-ldropWhile'[THEN isCont-o2[rotated]]: isCont (ldropWhile' P) l
 by (simp add: isCont-def filterlim-at'-list tendsto-ldropWhile')

lemma ldropWhile'-lfinite[simp]: lfinite xs \implies ldropWhile' P xs = llist-of (dropWhile P (list-of xs))
 by (simp add: ldropWhile'-def Lim-at'-lfinite)

lemma ldropWhile'-LNil: ldropWhile' P LNil = LNil

by simp

```
lemma ldropWhile'-LCons [simp]: ldropWhile' P (LCons l xs) = (if P l then ldrop-
  While' P xs else LCons l xs)
  by (rule tendsto-closed[where x=xs, OF closed-Collect-eq-isCont])
    (auto intro!: isCont-ldropWhile' isCont-If isCont-LCons)
```

```
lemma ldropWhile' P (lmap f xs) = lmap f (ldropWhile' (P o f) xs)
  by (rule tendsto-closed[where x=xs, OF closed-Collect-eq-isCont])
    (auto simp add: dropWhile-map comp-def intro!: isCont-lmap isCont-ldropWhile')
```

```
lemma ldropWhile'-LNil-I[simp]:  $\forall x \in lset xs. P x \implies ldropWhile' P xs = LNil$ 
  by (rule tendsto-closed[where x=xs, OF closed-Collect-eq-isCont])
    (auto simp add: llist-of-eq-LNil-conv intro!: isCont-lmap isCont-ldropWhile'
      dest!: lprefix-lsetD)
```

```
lemma lnull-ldropWhile': lnull (ldropWhile' P xs)  $\longleftrightarrow (\forall x \in lset xs. P x)$  (is ?lhs
   $\longleftrightarrow -$ )
  proof (intro iffI ballI)
    fix x assume x ∈ lset xs ?lhs then show P x by induct (simp-all split:
      if-split-asm)
  qed simp
```

```
lemma lhd-lfilter': lhd (lfilter' P xs) = lhd (ldropWhile' (Not o P) xs)
  proof cases
    assume  $\exists x \in lset xs. P x$ 
    then obtain x where x ∈ lset xs and P x by blast
    from ⟨x ∈ lset xs⟩ show ?thesis by induct (simp-all add: ⟨P x⟩)
  qed simp
```

11.5 Define ldrop as continuous extension

```
primrec edrop where
  edrop n [] = []
  | edrop n (x # xs) = (case n of eSuc n  $\Rightarrow$  edrop n xs | 0  $\Rightarrow$  x # xs)
```

```
lemma mono-edrop: edrop n xs  $\leq$  edrop n (xs @ ys)
  by (induct xs arbitrary: n) (auto split: enat-cosplit)
```

```
lemma edrop-mono: xs  $\leq$  ys  $\implies$  edrop n xs  $\leq$  edrop n ys
  using mono-edrop[of n xs] by (auto simp add: less-eq-list-def prefix-def)
```

```
definition ldrop' n xs = Lim (at' xs) (llist-of o edrop n o list-of)
```

```
lemma ldrop'-lfinite[simp]: lfinite xs  $\implies$  ldrop' n xs = llist-of (edrop n (list-of xs))
  by (simp add: ldrop'-def Lim-at'-lfinite)
```

```
lemma tendsto-ldrop': (ldrop' n  $\longrightarrow$  ldrop' n l) (at' l)
  by (rule tendsto-Lim-at'[OF ldrop'-def]) (auto simp add: lfinite-eq-range.llist-of)
```

```

intro!: edrop-mono)

lemma isCont-ldrop'[THEN isCont-o2[rotated]]: isCont (ldrop' n) l
  by (simp add: isCont-def filterlim-at'-list tendsto-ldrop')

lemma ldrop' n LNil = LNil
  by simp

lemma ldrop' n (LCons x xs) = (case n of 0 ⇒ LCons x xs | eSuc n ⇒ ldrop' n
  xs)
  by (rule tendsto-closed[where x=xs, OF closed-Collect-eq-isCont])
    (auto intro!: isCont-ldrop' isCont-enat-case isCont-LCons split: enat-cosplit)

primrec up :: 'a :: order ⇒ 'a list ⇒ 'a list where
  up a [] = []
  | up a (x # xs) = (if a < x then x # up a xs else up a xs)

lemma set-upD: x ∈ set (up y xs) ⇒ x ∈ set xs ∧ y < x
  by (induct xs arbitrary: y) (auto split: if-split-asm intro: less-trans)

lemma prefix-up: prefix (up a xs) (up a (xs @ ys))
  by (induction xs arbitrary: a) auto

lemma mono-up: xs ≤ ys ⇒ up a xs ≤ up a ys
  unfolding less-eq-list-def by (subst (asm) prefix-def) (auto intro!: prefix-up)

lemma sorted-up: sorted (up a xs)
  by (induction xs arbitrary: a) (auto dest: set-upD intro: less-imp-le)

```

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)
  by (rule tendsto-Lim-at'[OF lup-def]) (auto simp: lfinite-eq-range-llist-of mono-up)

lemma isCont-lup[THEN isCont-o2[rotated]]: isCont (lup a) l
  by (simp add: isCont-def filterlim-at'-list tendsto-lup)

lemma lup-lfinite[simp]: lfinite xs ⇒ lup a xs = llist-of (up a (list-of xs))
  by (simp add: lup-def Lim-at'-lfinite)

lemma lup-LNil: lup a LNil = LNil
  by simp

lemma lup-LCons [simp]: lup a (LCons x xs) = (if a < x then LCons x (lup x xs)
  else lup a xs)
  by (rule tendsto-closed[where x=xs, OF closed-Collect-eq-isCont])

```

```

(auto intro!: isCont-lup isCont-If isCont-LCons)

lemma lset-lup: lset (lup x xs) ⊆ lset xs ∩ {y. x < y}
  by (rule tendsto-le-ccpo[where g=λxs. lset (lup x xs) and f=λxs. lset xs ∩ {y. x < y} and F=at' xs])
    (auto dest!: lprefix-lsetD set-upD intro!: tendsto-intros at'-bot tendsto-lup eventually-at'-llistI)

lemma lsorted-lup: lsorted (lup (a::'a::linorder) l)
  by (rule tendsto-closed[OF closed-lsorted', OF isCont-lup])
    (auto intro!: sorted-up simp: lprefix-conv-lappend)

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)
  by(rule llist.fixp-preserves-mono2[OF lup'.mono lup'-def]) simp

lemma mono2mono-lup'2[THEN llist.mono2mono, simp, cont-intro]:
  shows monotone-lup'2: monotone lprefix lprefix (lup' a)
  using monotone-lup' by auto

lemma mcont-lup': mcont (prod-lub the-Sup lSup) (rel-prod (=) lprefix) lSup lprefix
  (λ(a, xs). lup' a xs)
  by(rule llist.fixp-preserves-mcont2[OF lup'.mono lup'-def]) simp

lemma mcont2mcont-lup'2[THEN llist.mcont2mcont, simp, cont-intro]:
  shows mcont-lup'2: mcont lSup lprefix lSup lprefix (lup' a)
  using mcont-lup' by auto

simp-of-case lup'-simps [simp]: lup'.simps

lemma lset-lup'-subset:
  fixes x :: - :: preorder
  shows lset (lup' x xs) ⊆ lset xs ∩ {y. x < y}
  by(induction xs arbitrary: x)(auto intro: less-trans)

lemma in-lset-lup'D:
  fixes x :: - :: preorder
  assumes y ∈ lset (lup' x xs)
  shows y ∈ lset xs ∧ x < y

```

```

using lset-lup'-subset[of x xs] assms by auto

lemma lsorted-lup':
  fixes x :: - :: preorder
  shows lsorted (lup' x xs)
by(induction xs arbitrary: x)(auto simp add: lsorted-LCons dest: in-lset-lup'D intro: less-imp-le)

lemma ldistinct-lup':
  fixes x :: - :: preorder
  shows ldistinct (lup' x xs)
by(induction xs arbitrary: x)(auto dest: in-lset-lup'D)

context fixes f :: 'a ⇒ 'a begin

partial-function (llist) iterate :: 'a ⇒ 'a llist
where iterate x = LCons x (iterate (f x))

lemma lmap-iterate: lmap f (iterate x) = iterate (f x)
by(induction arbitrary: x rule: iterate.fixp-induct) simp-all

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))
  by (induction xs arbitrary: a) auto

lemma mono-ext: assumes xs ≤ ys shows extup a xs ≤ extup a ys extdown a xs
≤ extdown a ys
  using assms[unfolded less-eq-list-def prefix-def] by (auto simp: less-eq-list-def prefix-ext)

lemma set-ext: set (extup a xs) ⊆ {a} ∪ set xs set (extdown a xs) ⊆ {a} ∪ set xs
  by (induction xs arbitrary: a) auto

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)
  by (rule tendsto-Lim-at'[OF lextup-def]) (auto simp: lfinite-eq-range-llist-of mono-ext)

lemma tendsto-lextdown[tendsto-intros]: (lextdown i —→ lextdown i xs) (at' xs)

```

```

by (rule tendsto-Lim-at'[OF lextdown-def]) (auto simp: lfinite-eq-range-llist-of
mono-ext)

lemma isCont-lextup[THEN isCont-o2[rotated]]: isCont (lextup a) l
by (simp add: isCont-def filterlim-at'-list tendsto-lextup)

lemma isCont-lextdown[THEN isCont-o2[rotated]]: isCont (lextdown a) l
by (simp add: isCont-def filterlim-at'-list tendsto-lextdown)

lemma lextup-lfinite[simp]: lfinite xs ==> lextup i xs = llist-of (extup i (list-of xs))
by (simp add: lextup-def Lim-at'-lfinite)

lemma lextdown-lfinite[simp]: lfinite xs ==> lextdown i xs = llist-of (extdown i
(list-of xs))
by (simp add: lextdown-def Lim-at'-lfinite)

lemma lextup i LNil = LNil lextdown i LNil = LNil
by simp-all

lemma lextup i (LCons x xs) = (if i ≤ x then lextup x xs else LCons i (lextdown
x xs))
by (rule tendsto-closed[where x=xs, OF closed-Collect-eq-isCont])
(auto intro!: isCont-lextdown isCont-lextup isCont-If isCont-LCons)

lemma lextdown i (LCons x xs) = (if x ≤ i then lextdown x xs else LCons i (lextup
x xs))
by (rule tendsto-closed[where x=xs, OF closed-Collect-eq-isCont])
(auto intro!: isCont-lextdown isCont-lextup isCont-If isCont-LCons)

lemma lset (lextup a xs) ⊆ {a} ∪ lset xs
apply (rule tendsto-le-ccpo[where g=λxs. lset (lextup a xs) and f=λxs. {a} ∪
lset xs and F=at' xs])
apply (rule tendsto-intros at'-bot tendsto-lup eventually-at'-llistI tendsto-id-at')+
apply (auto dest!: lprefix-lsetD set-ext[THEN subsetD])
done

lemma lset (lextdown a xs) ⊆ {a} ∪ lset xs
apply (rule tendsto-le-ccpo[where g=λxs. lset (lextdown a xs) and f=λxs. {a}
∪ lset xs and F=at' xs])
apply (rule tendsto-intros at'-bot tendsto-lup eventually-at'-llistI tendsto-id-at')+
apply (auto dest!: lprefix-lsetD set-ext[THEN subsetD])
done

lemma distinct-ext:
assumes distinct xs a ∉ set xs
shows distinct (lextup a xs) distinct (lextdown a xs)
using assms set-ext
apply (induction xs arbitrary: a)
apply auto

```

```

apply (metis eq-iff insert-iff subset-iff)+
done

lemma ldistinct xs ==> anotin lset xs ==> ldistinct (lexup a xs)
  by (rule-tac tendsto-closed[OF closed-ldistinct', OF isCont-lexup])
    (force simp: lfinite-eq-range-llist-of intro!: distinct-ext dest: ldistinct-lprefix dest:
lprefix-lsetD)+

definition esum-list :: ereal llist > ereal where
  esum-list xs = Lim (at' xs) (sum-list o list-of)

lemma esum-list-lfinite[simp]: lfinite xs ==> esum-list xs = sum-list (list-of xs)
  by (simp add: esum-list-def Lim-at'-lfinite)

lemma esum-list-LNil: esum-list LNil = 0
  by simp

context
  fixes xs :: ereal llist
  assumes xs: lset xs ==> 0 ≤ x
begin

lemma esum-list-tendsto-SUP:
  ((sum-list o list-of) —> (SUP ys ∈ {ys. lfinite ys ∧ ys ≤ xs}. esum-list ys)) (at' xs)
    (is (- —> ?y) -)
proof (rule order-tendstoI)
  fix a assume a < ?y
  then obtain ys where llist-of ys ≤ xs a < sum-list ys
    by (auto simp: less-SUP-iff lfinite-eq-range-llist-of)
  moreover
  { fix zs assume ys ≤ zs llist-of zs ≤ xs
    then obtain ys' where zs = ys @ ys'
      by (auto simp: prefix-def less-eq-list-def)
    with ‹llist-of zs ≤ xs› have nonneg: 0 ≤ sum-list ys'
      using xs by (auto simp: lprefix-conv-lappend sum-list-sum-nth intro: sum-nonneg)
    note ‹a < sum-list ys›
    also have sum-list ys ≤ sum-list zs
      using zs add-mono[OF order-refl nonneg] by auto
    finally have a < sum-list zs . }
  ultimately show eventually (λx. a < (sum-list o list-of) x) (at' xs)
    unfolding eventually-at'-llist by (auto simp: lfinite-eq-range-llist-of)
next
  fix a assume ?y < a then show eventually (λx. (sum-list o list-of) x < a) (at' xs)
    by (auto intro!: eventually-at'-llistI dest: SUP-lessD)
qed

lemma tendsto-esum-list: (esum-list —> esum-list xs) (at' xs)

```

```

apply (rule filterlim-cong[where g=sum-list o list-of, THEN iffD2, OF refl refl])
apply (rule eventually-mono[OF eventually-lfinite])
apply simp
unfolding esum-list-def
apply (rule tendsto-LimI)
apply (rule esum-list-tendsto-SUP)
done

lemma isCont-esum-list: isCont esum-list xs
by (simp add: isCont-def filterlim-at'-list tendsto-esum-list)

end

lemma esum-list-nonneg:
( $\bigwedge x. x \in lset xs \implies 0 \leq x$ )  $\implies 0 \leq$  esum-list xs
by (rule tendsto-le[OF at'-bot tendsto-esum-list tendsto-const])
(auto intro!: eventually-at'-llistI sum-nonneg
simp: lprefix-conv-lappend sum-list-sum-nth lfinite-eq-range-llist-of)

lemma esum-list-LCons:
assumes x:  $0 \leq x \bigwedge x. x \in lset xs \implies 0 \leq x$  shows esum-list (LCons x xs) =
x + esum-list xs
proof (rule tendsto-unique-eventually[OF at'-bot])
from x show (( $\lambda xs.$  esum-list (LCons x xs))  $\longrightarrow$  esum-list (LCons x xs)) (at' xs)
by (intro tendsto-compose[OF filterlim-at'-list[OF tendsto-esum-list] tendsto-LCons])
auto
show eventually ( $\lambda xa.$  esum-list (LCons x xa) = x + esum-list xa) (at' xs)
using eventually-lfinite by eventually-elim simp
from x show (( $\lambda xa.$  x + esum-list xa)  $\longrightarrow$  x + esum-list xs) (at' xs)
by (intro esum-list-nonneg tendsto-esum-list tendsto-add-ereal) auto
qed

lemma esum-list-lfilter':
assumes nn:  $\bigwedge x. x \in lset xs \implies 0 \leq x$  shows esum-list (lfilter' ( $\lambda x.$  x  $\neq 0$ ) xs) = esum-list xs
proof (rule tendsto-unique-eventually[OF at'-bot])
show (esum-list  $\longrightarrow$  esum-list xs) (at' xs)
using nn by (rule tendsto-esum-list)
from nn show (( $\lambda xs.$  esum-list (lfilter' ( $\lambda x.$  x  $\neq 0$ ) xs))  $\longrightarrow$  esum-list (lfilter' ( $\lambda x.$  x  $\neq 0$ ) xs)) (at' xs)
by (intro tendsto-compose[OF filterlim-at'-list[OF tendsto-esum-list] tendsto-lfilter])
(auto simp: lset-lfilter')
show eventually ( $\lambda xs.$  esum-list (lfilter' ( $\lambda x.$  x  $\neq 0$ ) xs) = esum-list xs) (at' xs)
using eventually-lfinite
by eventually-elim (simp add: sum-list-map-filter[where f =  $\lambda x.$  x and P= $\lambda x.$  x  $\neq 0$ , simplified])
qed

```

```

function f:: nat list  $\Rightarrow$  nat list where
  f [] = []
  | f (x#xs) = (x * 2) # f (f xs)
  by auto pat-completeness

termination f
proof (relation inv-image natLess length)
  fix x xs assume f-dom xs then show (f xs, x # xs)  $\in$  inv-image natLess length
    by induct (auto simp: f.psimps natLess-def)
  qed (auto intro: wf-less simp add: natLess-def)

lemma length-f[simp]: length (f xs) = length xs
  by (induct rule: f.induct) simp-all

lemma f-mono':  $\exists ys'. f (xs @ ys) = f xs @ ys'$ 
proof (induct length xs arbitrary: xs ys rule: less-induct)
  case less then show ?case
    apply (cases xs)
    apply auto
    apply (metis length-f lessI)
    done
  qed

lemma f-mono: xs  $\leq$  ys  $\implies$  f xs  $\leq$  f ys
  by (auto simp: less-eq-list-def prefix-def f-mono')

definition f' l = Lim (at' l) ( $\lambda l. llist-of (f (list-of l))$ )

lemma f'-lfinite[simp]: lfinite xs  $\implies$  f' xs = llist-of (f (list-of xs))
  by (simp add: f'-def Lim-at'-lfinite)

lemma tendsto-f': (f'  $\longrightarrow$  f' l) (at' l)
  by (rule tendsto-Lim-at'[OF f'-def]) (auto simp add: lfinite-eq-range-llist-of intro!: f-mono)

lemma isCont-f'[THEN isCont-o2[rotated]]: isCont f' l
  by (simp add: isCont-def filterlim-at'-list tendsto-f')

lemma f' LNil = LNil
  by simp

lemma f' (LCons x xs) = LCons (x * 2) (f' (f' xs))
  by (rule tendsto-closed[where x=x, OF closed-Collect-eq-isCont])
    (auto intro!: isCont-f' isCont-LCons)

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
  by(auto simp add: rel-fun-def Set.is-empty-def dest: rel-setD1 rel-setD2)

lemma monotone-comp: [[ monotone orda ordb g; monotone ordb ordc f ]] ==>
  monotone orda ordc (f o g)
  by(rule monotoneI)(simp add: monotoneD)

lemma cont-comp: [[ mcont luba orda lubb ordb g; cont lubb ordb lubc ordc f ]] ==>
  cont luba orda lubc ordc (f o g)
  apply(rule contI)
  apply(frule (2) mcont-contD)
  apply(simp)
  apply(drule (1) contD[OF - chain-imageI])
  apply(erule (1) mcont-monoD)
  apply(simp-all add: image-image o-def)
  done

lemma mcont-comp: [[ mcont luba orda lubb ordb g; mcont lubb ordb lubc ordc f ]] ==>
  mcont luba orda lubc ordc (f o g)
  by(auto simp add: mcont-def intro: cont-comp monotone-comp)

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
  unfolding monotone-def[abs-def] by transfer-prover

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
  unfolding cont-def[abs-def] Set.is-empty-def[symmetric] by transfer-prover

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
  unfolding mcont-def[abs-def] by transfer-prover

```

```
end
```

```
lemma (in ccpo) Sup-Un-less:
assumes chain: Complete-Partial-Order.chain ( $\leq$ ) (A  $\cup$  B)
and AB:  $\forall x \in A. \exists y \in B. x \leq y$ 
shows Sup (A  $\cup$  B) = Sup B
proof(rule order.antisym)
from chain have chain': Complete-Partial-Order.chain ( $\leq$ ) B
by(blast intro: chain-subset)
show Sup (A  $\cup$  B)  $\leq$  Sup B using chain
proof(rule ccpo-Sup-least)
fix x
assume x  $\in$  A  $\cup$  B
thus x  $\leq$  Sup B
proof
assume x  $\in$  A
then obtain y where x  $\leq$  y y  $\in$  B using AB by blast
note {x  $\leq$  y}
also from chain' {y  $\in$  B} have y  $\leq$  Sup B by(rule ccpo-Sup-upper)
finally show ?thesis .
qed(rule ccpo-Sup-upper[OF chain'])
qed
show Sup B  $\leq$  Sup (A  $\cup$  B)
using chain chain' by(blast intro: ccpo-Sup-least ccpo-Sup-upper)
qed
```

12.1 The ccpo structure

```
context includes tllist.lifting fixes b :: 'b begin
```

```
lift-definition tllist-ord :: ('a, 'b) tllist  $\Rightarrow$  ('a, 'b) tllist  $\Rightarrow$  bool
is  $\lambda(xs1, b1) (xs2, b2). \text{if } lfinite\ xs1 \text{ then } b1 = b \wedge lprefix\ xs1\ xs2 \vee xs1 = xs2 \wedge$ 
flat-ord b b1 b2 else xs1 = xs2
by auto
```

```
lift-definition tSup :: ('a, 'b) tllist set  $\Rightarrow$  ('a, 'b) tllist
is  $\lambda A. (lSup\ (fst\ 'A), flat-lub\ b\ (snd\ ' (A \cap \{(xs, -). lfinite\ xs\})))$ 
proof goal-cases
case (1 A1 A2)
hence fst ' A1 = fst ' A2 snd ' (A1 \cap \{(xs, -). lfinite xs\}) = snd ' (A2 \cap \{(xs,
-). lfinite xs\})
by(auto 4 3 simp add: rel-set-def intro: rev-image-eqI)
thus ?case by simp
qed
```

```
lemma tllist-ord-simps [simp, code]:
shows tllist-ord-TNil-TNil: tllist-ord (TNil b1) (TNil b2)  $\longleftrightarrow$  flat-ord b b1 b2
and tllist-ord-TNil-TCons: tllist-ord (TNil b1) (TCons y ys)  $\longleftrightarrow$  b1 = b
```

```

and tllist-ord-TCons-TNil: tllist-ord (TCons x xs) (TNil b2)  $\longleftrightarrow$  False
and tllist-ord-TCons-TCons: tllist-ord (TCons x xs) (TCons y ys)  $\longleftrightarrow$  x = y  $\wedge$ 
tllist-ord xs ys
by(auto simp add: tllist-ord.rep-eq flat-ord-def)

lemma tllist-ord-refl [simp]: tllist-ord xs xs
by transfer(auto simp add: flat-ord-def)

lemma tllist-ord-antisym:  $\llbracket \text{tllist-ord } xs \; ys; \text{tllist-ord } ys \; xs \rrbracket \implies xs = ys$ 
by transfer(auto simp add: flat-ord-def split: if-split-asm intro: lprefix-antisym)

lemma tllist-ord-trans [trans]:  $\llbracket \text{tllist-ord } xs \; ys; \text{tllist-ord } ys \; zs \rrbracket \implies \text{tllist-ord } xs \; zs$ 
by transfer(auto simp add: flat-ord-def split: if-split-asm intro: lprefix-trans)

lemma chain-tllist-llist-of-tllist:
assumes Complete-Partial-Order.chain tllist-ord A
shows Complete-Partial-Order.chain lprefix (llist-of-tllist ` A)
by(rule chainI)(auto 4 3 simp add: tllist-ord.rep-eq split: if-split-asm dest: chainD[OF assms])

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  $\wedge$  tfinite xs}
by(rule chainI)(auto simp add: tllist-ord.rep-eq flat-ord-def dest: chainD[OF assms])

lemma flat-ord-chain-finite:
assumes Complete-Partial-Order.chain (flat-ord b) A
shows finite A
proof –
from assms have  $\exists z. \forall x \in A. x = b \vee x = z$ 
by(clarify simp add: chain-def flat-ord-def) metis
then obtain z where  $\bigwedge x. x \in A \implies x = b \vee x = z$  by blast
hence A ⊆ {b, z} by auto
thus ?thesis by(rule finite-subset) simp
qed

lemma tSup-empty [simp]: tSup {} = TNil b
by(transfer)(simp add: flat-lub-def)

lemma is-TNil-tSup [simp]: is-TNil (tSup A)  $\longleftrightarrow$  ( $\forall x \in A. \text{is-TNil } x$ )
by transfer(simp add: split-beta)

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(cases tfinite xs)

```

```

case True
show ?thesis
proof(cases llist-of-tllist xs = llist-of-tllist (tSup A))
case True
with ⟨tfinite xs⟩ have lfinite (tSup (llist-of-tllist ‘A))
by(simp add: tSup-def image-image)
hence terminal (tSup A) = flat-lub b {terminal xs|xs. xs ∈ A ∧ tfinite xs} (is - = flat-lub - ?A)
by(simp add: tSup-def terminal-tllist-of-llist image-image)(auto intro: rev-image-eqI intro!: arg-cong[where f=flat-lub b])
moreover have flat-ord b (terminal xs) (flat-lub b ?A)
by(rule ccpo ccpo-Sup-upper[OF Partial-Function ccpo[OF flat-interpretation]])(blast intro: chain-tllist-terminal[OF chain] A <tfinite xs>)+
ultimately show ?thesis using True by(simp add: tllist-ord.rep-eq)
next
case False
hence  $\exists ys \in A. \neg tllist\text{-}ord ys xs$ 
by(rule contrapos-np)(auto intro!: lprefix-antisym chain-lSup-lprefix chain-lprefix-lSup simp add: tSup-def image-image A chain-tllist-llist-of-tllist[OF chain] tllist-ord.rep-eq split: if-split-asm)
then obtain ys where  $ys \in A \neg tllist\text{-}ord ys xs$  by blast
with A have tllist-ord xs ys xs ≠ ys by(auto dest: chainD[OF chain])
with True have terminal xs = b by transfer(auto simp add: flat-ord-def)
with True False show ?thesis
by(simp add: tllist-ord.rep-eq tSup-def image-image chain-lprefix-lSup chain-tllist-llist-of-tllist chain A)
qed
next
case False
thus ?thesis using assms by(simp add: tllist-ord.rep-eq tSup-def image-image chain-lprefix-lSup chain-tllist-llist-of-tllist not-lfinite-lprefix-conv-eq[THEN iffD1] terminal-tllist-of-llist split: if-split-asm)
qed

lemma chain-tSup-tllist-ord:
assumes chain: Complete-Partial-Order.chain tllist-ord A
and lub:  $\bigwedge xs'. xs' \in A \implies tllist\text{-}ord xs' xs$ 
shows tllist-ord (tSup A) xs
proof –
have  $\bigwedge xs'. xs' \in llist\text{-}of\text{-}tllist ' A \implies lprefix xs' (llist\text{-}of\text{-}tllist xs)$ 
by(auto dest!: lub simp add: tllist-ord.rep-eq split: if-split-asm)
with chain-tllist-llist-of-tllist[OF chain]
have prefix: lprefix (lSup (llist-of-tllist ‘ A)) (llist-of-tllist xs)
by(rule chain-lSup-lprefix)

show ?thesis
proof(cases tfinite (tSup A))
case False thus ?thesis using prefix
by(simp add: tllist-ord.rep-eq tSup-def image-image not-lfinite-lprefix-conv-eq[THEN
```

```

iffD1])
next
  case True
    from True have fin: lfinite (lSup (llist-of-tllist ` A)) by(simp add: tSup-def
image-image)
    have eq: terminal (tSup A) = flat-lub b {terminal xs|xs. xs ∈ A ∧ tfinite xs}
(is - = flat-lub - ?A)
    by(simp add: tSup-def terminal-tllist-of-llist image-image fin)(auto intro:
rev-image-eqI intro!: arg-cong[where f=flat-lub b])

    show ?thesis
  proof(cases lprefix (llist-of-tllist xs) (lSup (llist-of-tllist ` A)))
    case True
      with prefix have lSup (llist-of-tllist ` A) = llist-of-tllist xs by(rule lpre-
fix-antisym)
      moreover have flat-ord b (flat-lub b ?A) (terminal xs)
      by(rule ccpo ccpo-Sup-least[OF Partial-Function ccpo[OF flat-interpretation]])(auto
intro: chain-tllist-terminal[OF chain] dest: lub simp add: tllist-ord.rep-eq flat-ord-def)
      ultimately show ?thesis using eq by(simp add: tllist-ord.rep-eq tSup-def
image-image)
    next
    case False
    { fix xs'
      assume xs' ∈ A
      with False have ¬ lprefix (llist-of-tllist xs) (llist-of-tllist xs')
      by-(erule contrapos-nn, auto 4 4 intro: lprefix-trans chain-lprefix-lSup
chain-tllist-tllist-of-tllist chain)
      with lub[OF `xs' ∈ A] have terminal xs' = b
      by(auto simp add: tllist-ord.rep-eq split: if-split-asm)
      with chain-tllist-terminal[OF chain] have flat-ord b (flat-lub b ?A) b
      by-(rule ccpo ccpo-Sup-least[OF Partial-Function ccpo[OF flat-interpretation]],

auto simp add: flat-ord-def)
      hence flat-lub b ?A = b by(simp add: flat-ord-def)
      thus ?thesis using True eq prefix
      by(simp add: tSup-def terminal-tllist-of-llist tllist-ord.rep-eq image-image)
    qed
    qed
  qed

lemma tllist-ord ccpo [simp, cont-intro]:
  class ccpo tSup tllist-ord (mk-less tllist-ord)
by unfold-locales(auto simp add: mk-less-def intro: tllist-ord-antisym tllist-ord-trans
chain-tllist-ord-tSup chain-tSup-tllist-ord)

lemma tllist-ord-partial-function-definitions: partial-function-definitions tllist-ord
tSup
by unfold-locales(auto simp add: mk-less-def intro: tllist-ord-antisym tllist-ord-trans
chain-tllist-ord-tSup chain-tSup-tllist-ord)

```

```

interpretation tllist: partial-function-definitions tllist-ord tSup
by(rule tllist-ord-partial-function-definitions)

lemma admissible-mcont-is-TNil [THEN admissible-subst, cont-intro, simp]:
shows admissible-is-TNil: ccpo.admissible tSup tllist-ord is-TNil
by(rule ccpo.admissibleI)(simp)

lemma terminal-tSup:
 $\forall xs \in Y. is\text{-}TNil xs \implies \text{terminal } (tSup Y) = \text{flat-lub } b \text{ (terminal } 'Y)$ 
including tllist.lifting apply transfer
apply (auto simp add: split-def)
apply (rule arg-cong [where  $f = \text{flat-lub } b$ ])
apply force
done

lemma thd-tSup:
 $\exists xs \in Y. \neg is\text{-}TNil xs \implies \text{thd } (tSup Y) = (\text{THE } x. x \in \text{thd } ' (Y \cap \{xs. \neg is\text{-}TNil xs\}))$ 
apply(simp add: tSup-def image-image)
apply(rule arg-cong[where  $f = \text{The}$ ])
apply(auto intro: rev-image-eqI intro!: ext)
done

lemma ex-TCons-raw-parametric:
includes lifting-syntax
shows (rel-set (rel-prod (llist-all2 A) B) ==> (=)) ( $\lambda Y. \exists (xs, b) \in Y. \neg lnull xs$ ) ( $\lambda Y. \exists (xs, b) \in Y. \neg lnull xs$ )
by(auto 4 4 simp add: rel-fun-def dest: rel-setD1 rel-setD2 llist-all2-lnullD intro: rev-bexI)

lift-definition ex-TCons :: ('a, 'b) tllist set  $\Rightarrow$  bool
is  $\lambda Y. \exists (xs, b) \in Y. \neg lnull xs$  parametric ex-TCons-raw-parametric
by (blast dest: rel-setD1 rel-setD2)+

lemma ex-TCons-iff: ex-TCons Y  $\longleftrightarrow$  ( $\exists xs \in Y. \neg is\text{-}TNil xs$ )
by transfer auto

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))
 $(\lambda A. A \cap \{(xs, b). \neg lnull xs\}) (\lambda A. A \cap \{(xs, b). \neg lnull xs\})$ 
by(rule rel-funI rel-setI)+(auto 4 4 dest: llist-all2-lnullD rel-setD2 rel-setD1 intro: rev-bexI)

lift-definition retain-TCons :: ('a, 'b) tllist set  $\Rightarrow$  ('a, 'b) tllist set
is  $\lambda A. A \cap \{(xs, b). \neg lnull xs\}$  parametric retain-TCons-raw-parametric
by(rule rel-setI)(fastforce dest: rel-setD1 rel-setD2)+
```

```

lemma retain-TCons-conv: retain-TCons A = A ∩ {xs. ¬ is-TNil xs}
by(auto simp add: retain-TCons-def intro: rev-image-eqI)

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}))
unfolding ex-TCons-iff[symmetric] retain-TCons-conv[symmetric]
proof (transfer, goal-cases)
  case prems: (1 Y)
  then obtain xs b' where xsb: (xs, b') ∈ Y ∩ lnull xs by blast
  note chain = prems(1)

  have flat-lub b (snd ‘(Y ∩ {(xs, -). lfinite xs})) = flat-lub b (insert b (snd ‘(Y
  ∩ {(xs, -). lfinite xs})))
    by(auto simp add: flat-lub-def)
  also have insert b (snd ‘(Y ∩ {(xs, -). lfinite xs})) = insert b (snd ‘(apfst ttl ‘
  (Y ∩ {(xs, b). ¬ lnull xs}) ∩ {(xs, -). lfinite xs}))
    apply(auto intro: rev-image-eqI)
    apply(erule contrapos-np)
    apply(frule chainD[OF chain ‘(xs, b') ∈ Y])
    using ↪ lnull xs xsb
    by(fastforce split: if-split-asm simp add: lprefix-lnull intro!: rev-image-eqI)
  also have flat-lub b ... = flat-lub b (snd ‘(apfst ttl ‘(Y ∩ {(xs, b). ¬ lnull xs})
  ∩ {(xs, -). lfinite xs}))
    by(auto simp add: flat-lub-def)
  also
  have ttl ‘(fst ‘Y ∩ {xs. ¬ lnull xs}) = fst ‘apfst ttl ‘(Y ∩ {(xs, b). ¬ lnull xs})
    by(auto simp add: image-image)
  ultimately show ?case using prems by clarsimp
qed

lemma tSup-TCons: A ≠ {} ⟹ tSup (TCons x ‘A) = TCons x (tSup A)
unfolding Set.is-empty-def[symmetric]
apply transfer
unfolding Set.is-empty-def
apply(clarsimp simp add: image-image lSup-LCons[symmetric])
apply(rule arg-cong[where f=flat-lub b])
apply(auto 4 3 intro: rev-image-eqI)
done

lemma tllist-ord-terminalD:
  [ tllist-ord xs ys; is-TNil ys ] ⟹ flat-ord b (terminal xs) (terminal ys)
by(cases xs)(auto simp add: is-TNil-def)

lemma tllist-ord-bot [simp]: tllist-ord (TNil b) xs
by(cases xs)(simp-all add: flat-ord-def)

lemma tllist-ord-ttlI:
  tllist-ord xs ys ⟹ tllist-ord (ttl xs) (ttl ys)

```

```
by(cases xs ys rule: tllist.exhaust[case-product tllist.exhaust])(simp-all)
```

```
lemma not-is-TNil-conv:  $\neg \text{is-TNil } xs \longleftrightarrow (\exists x \, xs'. \, xs = TCons \, x \, xs')$   
by(cases xs) simp-all
```

12.2 Continuity of predefined constants

```
lemma mono-tllist-ord-case:
```

```
fixes bot
```

```
assumes mono:  $\bigwedge x. \text{monotone tllist-ord ord } (\lambda xs. f \, x \, xs \, (TCons \, x \, xs))$ 
```

```
and ord: class.preorder ord (mk-less ord)
```

```
and bot:  $\bigwedge x. \text{ord } (\text{bot } b) \, x$ 
```

```
shows monotone tllist-ord ord  $(\lambda xs. \text{case } xs \text{ of } TNil \, b \Rightarrow \text{bot } b \mid TCons \, x \, xs' \Rightarrow f \, x \, xs' \, xs)$ 
```

```
proof -
```

```
interpret preorder ord mk-less ord by (fact ord)
```

```
show ?thesis by(rule monotoneI)(auto split: tllist.split dest: monotoneD[OF mono] simp add: flat-ord-def bot)
```

```
qed
```

```
lemma mcont-lprefix-case-aux:
```

```
fixes f bot ord
```

```
defines g  $\equiv \lambda xs. f \, (\text{thd } xs) \, (\text{ttl } xs) \, (TCons \, (\text{thd } xs) \, (\text{ttl } xs))$ 
```

```
assumes mcont:  $\bigwedge x. \text{mcont } tSup \, \text{tllist-ord } \text{lub } \text{ord } (\lambda xs. f \, x \, xs \, (TCons \, x \, xs))$ 
```

```
and ccpo: class.ccpo lub ord (mk-less ord)
```

```
and bot:  $\bigwedge x. \text{ord } (\text{bot } b) \, x$ 
```

```
and cont-bot: cont (flat-lub b) (flat-ord b) lub ord bot
```

```
shows mcont tSup tllist-ord lub ord  $(\lambda xs. \text{case } xs \text{ of } TNil \, b \Rightarrow \text{bot } b \mid TCons \, x \, xs' \Rightarrow f \, x \, xs' \, xs)$ 
```

```
(is mcont - - - ?f)
```

```
proof(rule mcontI)
```

```
interpret b: ccpo lub ord mk-less ord by(rule ccpo)
```

```
show cont tSup tllist-ord lub ord ?f
```

```
proof(rule contI)
```

```
fix Y :: ('a, 'b) tllist set
```

```
assume chain: Complete-Partial-Order.chain tllist-ord Y
```

```
and Y:  $Y \neq \{\}$ 
```

```
from chain have chain': Complete-Partial-Order.chain ord (?f ` Y)
```

```
by(rule chain-imageI)(auto split: tllist.split simp add: flat-ord-def intro!: bot mcont-monoD[OF mcont])
```

```
show ?f (tSup Y) = lub (?f ` Y)
```

```
proof(cases is-TNil (tSup Y))
```

```
case True
```

```
hence ?f ` Y = bot ` terminal ` Y
```

```
by(auto 4 3 split: tllist.split intro: rev-image-eqI intro!: imageI)
```

```
moreover from True have tSup Y = TNil (flat-lub b (terminal ` Y))
```

```
by -(rule tllist.expand, simp-all add: terminal-tSup)
```

```

moreover have Complete-Partial-Order.chain (flat-ord b) (terminal ` Y)
  using chain True by(auto intro: chain-imageI tllist-ord-terminalD)
ultimately show ?thesis using Y
by (simp add: contD [OF cont-bot] cong del: b.SUP-cong-simp)
next
  case False
  hence eq: tSup Y = TCons (thd (tSup Y)) (ttl (tSup Y)) by simp
  have eq'':
    ?f ` Y =
    ( $\lambda x. \text{bot} (\text{terminal } x))` (Y \cap \{xs. \text{is-TNil } xs\}) \cup$ 
    ( $\lambda xs. f (\text{thd } xs) (\text{ttl } xs) xs)` (Y \cap \{xs. \neg \text{is-TNil } xs\})$ 
  by(auto 4 3 split: tllist.splits intro: rev-image-eqI)

from False obtain xs where xs: xs \in Y \neg \text{is-TNil } xs by auto
{ fix ys
  assume ys \in Y \text{is-TNil } ys
  hence terminal ys = b using xs
  by(cases xs ys rule: tllist.exhaust[case-product tllist.exhaust])(auto dest:
chainD[OF chain])
  then have lub: lub (?f ` Y) = lub (( $\lambda xs. f (\text{thd } xs) (\text{ttl } xs) xs)` (Y \cap \{xs. \neg \text{is-TNil } xs\})) using xs chain' unfolding eq'
  by -(erule ccpo.Sup-Un-less[OF ccpo], auto simp add: intro!: bot)
{ fix xs
  assume xs: xs \in Y \cap \{xs. \neg \text{is-TNil } xs\}
  hence (THE x. x \in thd ` (Y \cap \{xs. \neg \text{is-TNil } xs\})) = thd xs
  by(auto dest: chainD[OF chain] simp add: not-is-TNil-conv intro!:
the-equality)
  hence f (THE x. x \in thd ` (Y \cap \{xs. \neg \text{is-TNil } xs\})) (ttl xs) (TCons (THE
x. x \in thd ` (Y \cap \{xs. \neg \text{is-TNil } xs\})) (ttl xs)) = f (thd xs) (ttl xs) xs
  using xs by simp }
  moreover have Complete-Partial-Order.chain tllist-ord (ttl ` (Y \cap \{xs. \neg \text{is-TNil } xs\}))
  using chain by(rule chain-imageI[OF chain-subset])(auto simp add: tl-
list-ord-ttlI)
  moreover from False have Y \cap \{xs. \neg \text{is-TNil } xs\} \neq {} by auto
  ultimately show ?thesis
  apply(subst (1 2) eq)
  using False
  apply(simp add: thd-tSup ttl-tSup[OF chain] mcont-contD[OF mcont] im-
age-image lub)
  done
qed
qed
from mcont-mono[OF mcont] - bot
show monotone tllist-ord ord ?f
by(rule mono-tllist-ord-case)(unfold-locales)
qed$ 
```

```

lemma cont-TNil [simp, cont-intro]: cont (flat-lub b) (flat-ord b) tSup tllist-ord
TNil
apply(rule contI)
apply transfer
apply(simp add: image-image image-constant-conv)
apply(rule arg-cong[where f=flat-lub b])
apply(auto intro: rev-image-eqI)
done

lemma monotone-TCons: monotone tllist-ord tllist-ord (TCons x)
by(rule monotoneI) simp

lemmas mono2mono-TCons[cont-intro] = monotone-TCons[THEN tllist.mono2mono]

lemma mcont-TCons: mcont tSup tllist-ord tSup tllist-ord (TCons x)
apply(rule mcontI)
apply(rule monotone-TCons)
apply(rule contI)
apply(simp add: tSup-TCons)
done

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 ( $\subseteq$ ) tset
including tllist.lifting
by transfer(rule monotone-comp[OF - monotone-lset], auto intro: monotoneI)

lemma mcont2mcont-tset [THEN lfp.mcont2mcont, cont-intro]:
shows mcont-tset: mcont tSup tllist-ord Union ( $\subseteq$ ) tset
including tllist.lifting
apply transfer
apply(rule mcont-comp[OF - mcont-lset])
unfolding mcont-def by(auto intro: monotoneI contI )

end

context includes lifting-syntax
begin

lemma rel-fun-lift:

$$(\bigwedge x. A (f x) (g x)) \implies ((=) \implies A) f g$$


```

```

by(simp add: rel-fun-def)

lemma tllist-ord-transfer [transfer-rule]:
  ((=) ==> pcr-tllist (=) (=) ==> pcr-tllist (=) (=) ==> (=))
    ( $\lambda b (xs_1, b_1) (xs_2, b_2).$  if lfinite  $xs_1$  then  $b_1 = b \wedge lprefix xs_1 xs_2 \vee xs_1 = xs_2 \wedge flat-ord b b_1 b_2$  else  $xs_1 = xs_2$ )
      tllist-ord
  by(rule rel-fun-lift)(rule tllist-ord.transfer)

lemma tSup-transfer [transfer-rule]:
  ((=) ==> rel-set (pcr-tllist (=) (=)) ==> pcr-tllist (=) (=))
    ( $\lambda b A.$  (lSup (fst `A), flat-lub b (snd `( $A \cap \{(xs, -). lfinite xs\}$ ))))
      tSup
  by(rule rel-fun-lift)(rule tSup.transfer)

end

lifting-update tllist.lifting
lifting-forget tllist.lifting

interpretation tllist: partial-function-definitions tllist-ord b tSup b for b
  by(rule tllist-ord-partial-function-definitions)

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)
  by(rule monotoneI)(auto split: tllist.split dest: monotoneD[OF tnil] monotoneD[OF tcons])

```

12.3 Definition of recursive functions

```

locale tllist-pf = fixes b :: 'b
begin

declaration <Partial-Function.init tllist @{term tllist.fixp-fun b}
@{term tllist.mono-body b} @{thm tllist.fixp-rule-uc[where b=b]} @{thm tllist.fixp-induct-uc[where b=b]} NONE>

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))
  by(rule monotoneI)(auto dest: monotoneD)

end

lemma mono-tllist-lappendt2 [partial-function-mono]:

```

```

tllist-pf.mono-tllist b A ==> tllist-pf.mono-tllist b (λf. lappendt xs (A f))
apply(rule monotoneI)
apply(drule (1) monotoneD)
apply(simp add: tllist-ord.rep-eq split: if-split-asm)
apply(auto simp add: lappend-inf)
done

lemma mono-tllist-tappend2 [partial-function-mono]:
assumes ∀y. tllist-pf.mono-tllist b (C y)
shows tllist-pf.mono-tllist b (λf. tappend xs (λy. C y f))
apply(cases tfinite xs)
apply(rule monotoneI)
apply(drule monotoneD[OF assms[where y=terminal xs]])
including tlist.lifting
apply transfer
apply(fastforce split: if-split-asm)
apply(simp add: tappend-inf)
done

end

```

13 Example definitions using the CCPO structure on terminated lazy lists

```

theory TLLList-CCPO-Examples imports
  ..../TLLList-CCPO
begin

context fixes b :: 'b begin
interpretation tllist-pf b .

context fixes P :: 'a ⇒ 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(rule iffI ballI)+
fix x

```

```

assume  $x \in tset xs$  ?lhs
thus  $\neg P x$ 
proof(induct rule: tllist-set-induct)
  case (find xs)
    thus ?case by(cases xs)(simp-all split: if-split-asm)
next
  case (step xs x)
    thus ?case by(cases xs)(simp-all split: if-split-asm)
qed
next
  assume ?rhs
  thus ?lhs
    by(induct arbitrary: xs rule: tfilter.fixp-induct)(simp-all split: tlist.split)
qed
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)
apply(rule tllist.fixp-preserves-mcont1[OF tfilter.mono tfilter-def])
apply(rule mcont-lprefix-case-aux)
apply simp-all
done

lemma tfilter-tfilter:
  tfilter b P (tfilter b Q xs) = tfilter b ( $\lambda x. P x \wedge Q x$ ) xs (is ?lhs xs = ?rhs xs)
proof(rule tllist.leq-antisym)
  have  $\forall xs. tllist-ord b$  (?lhs xs) (?rhs xs)
    by(rule tfilter.fixp-induct[where Pa= $\lambda f. \forall xs. tllist-ord b$  (tfilter b P (f xs)) (?rhs xs)])(auto split: tllist.split)
    thus tllist-ord b (?lhs xs) (?rhs xs) ..
  have  $\forall xs. tllist-ord b$  (?rhs xs) (?lhs xs)
    by(rule tfilter.fixp-induct[where Pa= $\lambda f. \forall xs. tllist-ord b$  (f xs) (?lhs xs)])(auto split: tllist.split)
    thus tllist-ord b (?rhs xs) (?lhs xs) ..
qed

declare ccpo.admissible-leI[OF complete-lattice-ccpo, cont-intro, simp]

lemma tset-tfilter: tset (tfilter b P xs) = { $x \in tset xs. P x$ }
proof(rule equalityI[OF - subsetI])
  show tset (tfilter b P xs)  $\subseteq \{x \in tset xs. P x\}$ 
    by(induct arbitrary: xs rule: tfilter.fixp-induct)(auto split: tllist.split)
  fix x
  assume  $x \in \{x \in tset xs. P x\}$ 
  hence  $x \in tset xs P x$  by simp-all
  thus  $x \in tset (tfilter b P xs)$ 

```

```

  by(induct rule: tset-induct) simp-all
qed

context fixes b :: 'b begin
interpretation tllist-pf b .

partial-function (tllist) tconcat :: ('a llist, 'b) tllist  $\Rightarrow$  ('a, 'b) tllist
where
  tconcat xs = (case xs of TNil b  $\Rightarrow$  TNil b | TCons x xs'  $\Rightarrow$  lappendt x (tconcat xs'))
end

simp-of-case tconcat2-simps [simp]: tconcat.simps
end

```

14 Example: Koenig's lemma

```

theory Koenigslemma imports
  .../Coinductive-List
begin

type-synonym 'node graph = 'node  $\Rightarrow$  'node  $\Rightarrow$  bool
type-synonym 'node path = 'node llist

coinductive-set paths :: 'node graph  $\Rightarrow$  'node path set
  for graph :: 'node graph
  where
    Empty: LNil  $\in$  paths graph
    | Single: LCons x LNil  $\in$  paths graph
    | LCons: [ graph x y; LCons y xs  $\in$  paths graph ]  $\Longrightarrow$  LCons x (LCons y xs)  $\in$  paths graph

definition connected :: 'node graph  $\Rightarrow$  bool
  where connected graph  $\longleftrightarrow$  ( $\forall n n'. \exists xs. llist\text{-}of (n \# xs @ [n']) \in paths graph$ )

inductive-set reachable-via :: 'node graph  $\Rightarrow$  'node set  $\Rightarrow$  'node  $\Rightarrow$  'node set
  for graph :: 'node graph and ns :: 'node set and n :: 'node
  where [ LCons n xs  $\in$  paths graph; n'  $\in$  lset xs; lset xs  $\subseteq$  ns ]  $\Longrightarrow$  n'  $\in$  reachable-via graph ns n

lemma connectedD: connected graph  $\Longrightarrow$   $\exists xs. llist\text{-}of (n \# xs @ [n']) \in paths graph$ 
  unfolding connected-def by blast

lemma paths-LConsD:
  assumes LCons x xs  $\in$  paths graph

```

```

shows xs ∈ paths graph
using assms
by(coinduct)(fastforce elim: paths.cases del: disjCI)

lemma paths-lappendD1:
assumes lappend xs ys ∈ paths graph
shows xs ∈ paths graph
using assms
proof coinduct
case (paths zs)
then show ?case
proof(rule paths.cases)
assume lappend zs ys = LNil then show ?thesis
by(simp add: lappend-eq-LNil-iff)
next
fix x assume lappend zs ys = LCons x LNil
then show ?thesis
by (metis LNil-eq-lappend-iff lappend-LNil2 lappend-ltl lnull-def ltl-simps(2))
next
fix x y ws
assume lappend zs ys = LCons x (LCons y ws) graph x y LCons y ws ∈ paths
graph
then show ?thesis
by (metis lappend-ltl llist.disc(2) lprefix-LCons-conv lprefix-lappend ltl-simps(2))
qed
qed

lemma paths-lappendD2:
assumes lfinite xs
and lappend xs ys ∈ paths graph
shows ys ∈ paths graph
using assms
by induct (fastforce elim: paths.cases intro: paths.intros)+

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 -
from set obtain xs' xs'' where lfinite xs'
and xs: xs = lappend xs' (LCons x xs'')
and x ∉ lset xs'
by(blast dest: split-llist-first)
show ?thesis
proof(cases n ∈ lset xs')
case False
let ?xs' = lappend xs' (LCons x LNil)

```

```

from xs path have lappend (LCons n (lappend xs' (LCons x LNil))) xs'' ∈ paths
graph
  by(simp add: lappend-assoc)
  hence LCons n ?xs' ∈ paths graph by(rule paths-lappendD1)
  moreover have x ∈ lset ?xs' lset ?xs' ⊆ lset xs n ∉ lset ?xs'
    using xs False ‹lfinite xs› n-neq-x by auto
  ultimately show ?thesis by blast
next
  case True
  from ‹lfinite xs› obtain XS' where xs': xs' = llist-of XS'
    unfolding lfinite-eq-range-llist-of by blast
  with True have n ∈ set XS' by simp
  from split-list-last[OF this]
  obtain ys zs where XS': XS' = ys @ n # zs
    and n ∉ set zs by blast
  let ?xs' = lappend (llist-of zs) (LCons x LNil)
  have lfinite (LCons n (llist-of ys)) by simp
  moreover from xs xs' XS' path
    have lappend (LCons n (llist-of ys)) (lappend (LCons n ?xs') xs'') ∈ paths
graph
  by(simp add: lappend-assoc)(metis lappend-assoc lappend-llist-of-LCons lappend-llist-of-llist-of_llist-of.simps(2))
  ultimately have lappend (LCons n ?xs') xs'' ∈ paths graph
  by(rule paths-lappendD2)
  hence LCons n ?xs' ∈ paths graph by(rule paths-lappendD1)
  moreover have x ∈ lset ?xs' lset ?xs' ⊆ lset xs n ∉ lset ?xs'
    using xs xs' XS' ‹lfinite xs› n-neq-x ‹n ∉ set zs› by auto
  ultimately show ?thesis by blast
qed
qed

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
  fix x
  assume x ∈ ?lhs
  then obtain xs where path: LCons n xs ∈ paths graph
  and x ∈ lset xs lset xs ⊆ ns by cases

  from ‹x ∈ lset xs› obtain n' xs' where xs: xs = LCons n' xs' by(cases xs) auto
  with path have graph n n' by cases simp-all
  moreover from ‹lset xs ⊆ ns› xs have n' ∈ ns by simp
  ultimately have n' ∈ {n'. graph n n'} ∩ ns by simp
  thus x ∈ ?rhs
proof(rule UN-I)
  show x ∈ insert n' (reachable-via graph (ns - {n'}) n')
  proof(cases x = n')

```

```

case True thus ?thesis by simp
next
  case False
    with xs <x ∈ lset xs> have x ∈ lset xs' by simp
    with path xs have path': LCons n' xs' ∈ paths graph by cases simp-all
      from <lset xs ⊆ ns> xs have lset xs' ⊆ ns by simp

      from path-avoid-node[OF path' <x ∈ lset xs'>] False
      obtain xs'' where path'': LCons n' xs'' ∈ paths graph
        and lset xs'' ⊆ lset xs' x ∈ lset xs'' n' ∉ lset xs'' by blast
      with False <lset xs ⊆ ns> xs show ?thesis by(auto intro: reachable-via.intros)
      qed
    qed
  qed

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(intro bexI conjI)
  let ?P = λ(n, ns). n'. graph n n' ∧ infinite (reachable-via graph (¬ insert n (insert n' ns)) n') ∧ n' ∉ insert n ns
  define LTL where LTL = (λ(n, ns). let n' = SOME n'. ?P (n, ns) n' in (n', insert n ns))
  define f where f = unfold-llist (λ-. False) fst LTL
  define ns :: 'node set' where ns = {}

  { fix n ns
    assume infinite (reachable-via graph (¬ insert n ns)) n
    hence ∃n'. ?P (n, ns) n'
    proof(rule contrapos-np)
      assume ¬ ?thesis
      hence fin: ∀n'. [graph n n'; n' ∉ insert n ns] ⇒ finite (reachable-via graph (¬ insert n (insert n' ns)) n')
        by blast

      from reachable-via-subset-unfold[of graph - insert n ns n]
      show finite (reachable-via graph (¬ insert n ns)) n
      proof(rule finite-subset[OF - finite-UN-I])
        from finite-branching[of n]
        show finite ({n'. graph n n'} ∩ - insert n ns) by blast
      next
        fix n'
        assume n' ∈ {n'. graph n n'} ∩ - insert n ns
        hence graph n n' n' ∉ insert n ns by simp-all
        from fin[OF this] have finite (reachable-via graph (¬ insert n (insert n' ns))) n' by blast
      qed
    qed
  }

```

```

ns)) n') .
moreover have - insert n (insert n' ns) = - insert n ns - {n'} by auto
ultimately show finite (insert n' (reachable-via graph (- insert n ns -
{n'})) n')) by simp
qed
qed }

note ex-P-I = this

{ fix n ns
have ¬ lnull (f (n, ns))
lhd (f (n, ns)) = n
ltl (f (n, ns)) = (let n' = SOME n'. ?P (n, ns) n' in f (n', insert n ns))
by(simp-all add: f-def LTL-def) }
note f-simps [simp] = this

{ fix n :: 'node and ns :: 'node set and x :: 'node
assume x ∈ lset (f (n, ns)) n ∉ ns
and finite ns infinite (reachable-via graph (- insert n ns) n)
hence x ∉ ns
proof(induct f (n, ns) arbitrary: n ns rule: lset-induct)
case find
thus ?case by(auto 4 4 dest: sym eq-LConsD)
next
case (step x' xs)
let ?n' = Eps (?P (n, ns))
and ?ns' = insert n ns
from eq-LConsD[OF ⟨LCons x' xs = f (n, ns)⟩[symmetric]]
have [simp]: x' = n and xs: xs = f (?n', ?ns') by auto
from ⟨infinite (reachable-via graph (- insert n ns) n)⟩
have ∃ n'. ?P (n, ns) n' by(rule ex-P-I)
hence P: ?P (n, ns) ?n' by(rule someI-ex)
moreover have insert ?n' ?ns' = insert n (insert ?n' ns) by auto
ultimately have ?n' ∉ ?ns' finite ?ns'
and infinite (reachable-via graph (- insert ?n' ?ns') ?n')
using ⟨finite ns⟩ by auto
with xs have x ∉ ?ns' by(rule step)
thus ?case by simp
qed }
note lset = this

show n ∈ lset (f (n, ns))
using llist.setsel(1)[OF f-simps(1), of n ns] by(simp del: llist.setsel(1))

show ¬ lfinit (f (n, ns))
proof
assume lfinit (f (n, ns))
thus False by(induct f (n, ns) arbitrary: n ns rule: lfinit-induct) auto
qed

```

```

let ?X = λxs. ∃n ns. xs = f (n, ns) ∧ finite ns ∧ infinite (reachable-via graph
(– insert n ns) n)

have reachable-via graph (– {n}) n ⊇ – {n}
proof(rule subsetI)
fix n' :: 'node
assume n' ∈ – {n}
hence n ≠ n' by auto
from connected obtain xs
  where llist-of (n # xs @ [n']) ∈ paths graph
    by(blast dest: connectedD)
hence LCons n (llist-of (xs @ [n'])) ∈ paths graph
  and n' ∈ lset (llist-of (xs @ [n'])) by simp-all
from path-avoid-node[OF this <n ≠ n'>] show n' ∈ reachable-via graph (– {n})
n
  by(auto intro: reachable-via.intros)
qed
hence infinite (reachable-via graph (– {n}) n)
  using infinite by(auto dest: finite-subset)
hence X: ?X (f (n, {})) by(auto)

thus f (n, {}) ∈ paths graph
proof(coinduct)
case (paths xs)
then obtain n ns where xs-def: xs = f (n, ns)
  and finite ns and infinite (reachable-via graph (– insert n ns) n) by blast
from <infinite (reachable-via graph (– insert n ns) n)>
have ∃n'. ?P (n, ns) n' by(rule ex-P-I)
hence P: ?P (n, ns) (Eps (?P (n, ns))) by(rule someI-ex)
let ?n' = Eps (?P (n, ns))
let ?ns' = insert n ns
let ?n'' = Eps (?P (?n', ?ns'))
let ?ns'' = insert ?n' ?ns'
have xs = LCons n (LCons ?n' (f (?n'', ?ns'')))
  unfolding xs-def by(auto 4 3 intro: llist.expand)
moreover from P have graph n ?n' by simp
moreover {
  have LCons ?n' (f (?n'', ?ns'')) = f (?n', ?ns')
    by(rule llist.expand) simp-all
  moreover have finite ?ns' using <finite ns> by simp
  moreover have insert ?n' ?ns' = insert n (insert ?n' ns) by auto
  hence infinite (reachable-via graph (– insert ?n' ?ns') ?n') using P by simp
    ultimately have ?X (LCons ?n' (f (?n'', ?ns''))) by blast }
  ultimately have ?LCons by simp
  thus ?case by simp
qed

from <infinite (reachable-via graph (– {n}) n)>
have infinite (reachable-via graph (– insert n ns) n) ∧ finite ns

```

```

    by(simp add: ns-def)
  thus ldistinct (f (n, ns))
  proof(coinduction arbitrary: n ns)
    case (ldistinct n ns)
    then obtain finite ns
      and infinite (reachable-via graph (– insert n ns) n) by simp
      from ⟨infinite (reachable-via graph (– insert n ns) n)⟩
      have ∃ n'. ?P (n, ns) n' by(rule ex-P-I)
      hence P: ?P (n, ns) (Eps (?P (n, ns))) by(rule someI-ex)
      let ?n' = Eps (?P (n, ns))
      let ?ns' = insert n ns
      have eq: insert ?n' ?ns' = insert n (insert ?n' ns) by auto
      hence n ∈ lset (f (?n', ?ns'))
        using P ⟨finite ns⟩ by(auto dest: lset)
      moreover from ⟨finite ns⟩ P eq
      have infinite (reachable-via graph (– insert ?n' ?ns') ?n') by simp
      ultimately show ?case using ⟨finite ns⟩ by auto
    qed
  qed
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
```

```
simp-of-case lmirror-aux-simps [simp]: lmirror-aux.code
```

```
lemma lnull-lmirror-aux [simp]:
  lnull (lmirror-aux acc xs) = (lnull xs ∧ lnull acc)
  by(fact lmirror-aux.simps)
```

```
lemma ltl-lmirror-aux:
  ltl (lmirror-aux acc xs) = (if lnull xs then ltl acc else lmirror-aux (LCons (lhd xs)
  acc) (ltl xs))
  by(cases acc)(simp-all split: llist.split)
```

```
lemma lhd-lmirror-aux:
  lhd (lmirror-aux acc xs) = (if lnull xs then lhd acc else lhd xs)
```

```

by(cases acc)(simp-all split: llist.split)

declare lmirror-aux.sel[simp del]

lemma lfinite-lmirror-aux [simp]:
lfinite (lmirror-aux acc xs)  $\longleftrightarrow$  lfinite xs  $\wedge$  lfinite acc
(is ?lhs  $\longleftrightarrow$  ?rhs)
proof
assume ?lhs
thus ?rhs
proof(induct zs $\equiv$ lmirror-aux acc xs arbitrary: acc xs rule: lfinite-induct)
case LCons
thus ?case
proof(cases lnull xs)
case True
with LCons.hyps(3)[of ltl acc LNil]
show ?thesis by(simp add: ltl-lmirror-aux)
qed(fastforce simp add: ltl-lmirror-aux)
qed simp
next
assume ?rhs
hence lfinite xs lfinite acc by simp-all
thus ?rhs by(induction arbitrary: acc) simp-all
qed

lemma lmirror-aux-inf:
 $\neg$  lfinite xs  $\implies$  lmirror-aux acc xs = xs
by(coinduction arbitrary: acc xs)(auto simp add: lhd-lmirror-aux ltl-lmirror-aux)

lemma lmirror-aux-acc:
lmirror-aux (lappend ys zs) xs = lappend (lmirror-aux ys xs) zs
proof(cases lfinite xs)
case False
thus ?thesis by(simp add: lmirror-aux-inf lappend-inf)
next
case True
thus ?thesis
by(induction arbitrary: ys)(simp-all add: lappend-code(2)[symmetric] del: lappend-code(2))
qed

lemma lmirror-aux-LCons:
lmirror-aux acc (LCons x xs) = LCons x (lappend (lmirror-aux LNil xs) (LCons x acc))
by (metis lappend-code(1) lmirror-aux-acc lmirror-aux-simps(2))

lemma llength-lmirror-aux: llength (lmirror-aux acc xs) = 2 * llength xs + llength
acc
apply(coinduction arbitrary: acc xs rule: enat-coinduct)

```

```

apply(auto simp add: iadd-is-0 epred-iadd1 mult-2 epred-llength ltl-lmirror-aux
iadd-Suc-right)
apply(rule exI conjI refl) +
apply(simp add: iadd-Suc-right llength-ltl)
by (metis (opaque-lifting, no-types) add.commute epred-llength iadd-Suc-right lhd-LCons-ltl
llength-LCons)

lemma lnull-lmirror [simp]: lnull (lmirror xs) = lnull xs
by(simp add: lmirror-def)

lemma lmirror-LNil [simp]: lmirror LNil = LNil
by(simp add: lmirror-def)

lemma lmirror-LCons [simp]: lmirror (LCons x xs) = LCons x (lappend (lmirror
xs) (LCons x LNil))
by(simp only: lmirror-aux-LCons lmirror-def)

lemma ltl-lmirror [simp]:
   $\neg \text{lnull } xs \implies \text{ltl } (\text{lmirror } xs) = \text{lappend } (\text{lmirror } (\text{ltl } xs)) (\text{LCons } (\text{lhd } xs) \text{ LNil})$ 
by(clarify simp add: not-lnull-conv)

lemma lmap-lmirror-aux: lmap f (lmirror-aux acc xs) = lmirror-aux (lmap f acc)
(lmap f xs)
by(coinduction arbitrary: acc xs rule: llist.coinduct-strong)(auto 4 3 simp add:
lhd-lmirror-aux ltl-lmirror-aux)

lemma lmap-lmirror: lmap f (lmirror xs) = lmirror (lmap f xs)
by(simp add: lmirror-def lmap-lmirror-aux)

lemma lset-lmirror-aux: lset (lmirror-aux acc xs) = lset (lappend xs acc)
proof(cases lfinite xs)
  case True thus ?thesis
    by(induction arbitrary: acc simp-all)
  qed(simp add: lmirror-aux-inf lappend-inf)

lemma lset-lmirror [simp]: lset (lmirror xs) = lset xs
by(simp add: lmirror-def lset-lmirror-aux)

lemma llength-lmirror [simp]: llength (lmirror xs) = 2 * llength xs
by(simp add: lmirror-def llength-lmirror-aux)

lemma lmirror-llist-of [simp]: lmirror (llist-of xs) = llist-of (xs @ rev xs)
by(induct xs)(simp-all add: lappend-llist-of-LCons)

lemma list-of-lmirror [simp]: lfinite xs  $\implies$  list-of (lmirror xs) = list-of xs @ rev
(list-of xs)
by (metis list-of-llist-of llist-of-list-of lmirror-llist-of)

lemma llist-all2-lmirror-aux:

```

```

 $\llist-all2 P acc acc'; \llist-all2 P xs xs' ]]$ 
 $\implies \llist-all2 P (\text{lmirror-aux } acc \ xs) (\text{lmirror-aux } acc' \ xs')$ 
by(coinduction arbitrary:  $acc \ acc' \ xs \ xs'$ )(auto simp add: lhd-lmirror-aux ltl-lmirror-aux elim: llist-all2-lhdD intro!: llist-all2-ltlI exI dest: llist-all2-lnullD)

lemma enat-mult-cancel1 [simp]:
 $k * m = k * n \longleftrightarrow m = n \vee k = 0 \vee k = (\infty :: \text{enat}) \wedge n \neq 0 \wedge m \neq 0$ 
by(cases  $k \ m \ n$  rule: enat.exhaust[case-product enat.exhaust[case-product enat.exhaust]])(simp-all add: zero-enat-def)

lemma llist-all2-lmirror-auxD:
 $\llist-all2 P (\text{lmirror-aux } acc \ xs) (\text{lmirror-aux } acc' \ xs'); \llist-all2 P acc acc';$ 
lfinite acc ]
 $\implies \llist-all2 P xs xs'$ 
proof(coinduction arbitrary:  $xs \ xs' \ acc \ acc'$ )
case (LNil  $xs \ xs' \ acc \ acc'$ )
from < $\llist-all2 P (\text{lmirror-aux } acc \ xs) (\text{lmirror-aux } acc' \ xs')$ >
have llength (lmirror-aux acc xs) = llength (lmirror-aux acc' xs')
by(rule llist-all2-llengthD)
moreover from < $\llist-all2 P acc acc'$ >
have llength acc = llength acc' by(rule llist-all2-llengthD)
ultimately have llength xs = llength xs'
using <lfinite acc> by(auto simp add: llength-lmirror-aux dest: lfinite-llength-enat)
thus ?case by auto
next
case (LCons  $xs \ xs' \ acc \ acc'$ )
from < $\llist-all2 P (\text{lmirror-aux } acc \ xs) (\text{lmirror-aux } acc' \ xs')$ >  $\setminus\!\!-\! lnull \ xs \ \setminus\!\!-\! lnull \ xs'$ 
have ?lhd by(auto dest: llist-all2-lhdD simp add: lhd-lmirror-aux)
thus ?case using LCons < $\llist-all2 P (\text{lmirror-aux } acc \ xs) (\text{lmirror-aux } acc' \ xs')$ >[THEN llist-all2-ltlI]
by(auto 4 3 simp add: ltl-lmirror-aux)
qed

lemma llist-all2-lmirrorI:
 $\llist-all2 P xs ys \implies \llist-all2 P (\text{lmirror } xs) (\text{lmirror } ys)$ 
by(simp add: lmirror-def llist-all2-lmirror-aux)

lemma llist-all2-lmirrorD:
 $\llist-all2 P (\text{lmirror } xs) (\text{lmirror } ys) \implies \llist-all2 P xs ys$ 
unfolding lmirror-def by(erule llist-all2-lmirror-auxD) simp-all

lemma llist-all2-lmirror [simp]:
 $\llist-all2 P (\text{lmirror } xs) (\text{lmirror } ys) \longleftrightarrow \llist-all2 P xs ys$ 
by(blast intro: llist-all2-lmirrorI llist-all2-lmirrorD)

lemma lmirror-parametric [transfer-rule]:
includes lifting-syntax
shows (llist-all2 A ===> llist-all2 A) lmirror lmirror

```

```
by(rule rel-funI)(rule llist-all2-lmirrorI)
```

```
end
```

16 The Hamming stream defined as a least fix-point

```
theory Hamming-Stream imports
  ..../Coinductive-List
  HOL-Computational-Algebra.Primes
begin
```

```
lemma infinity-inf-enat [simp]:
  fixes n :: enat
  shows  $\infty \sqcap n = n$   $n \sqcap \infty = n$ 
by(simp-all add: inf-enat-def)
```

```
lemma eSuc-inf-eSuc [simp]: eSuc n  $\sqcap$  eSuc m = eSuc (n  $\sqcap$  m)
by(simp add: inf-enat-def)
```

```
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')
by simp
```

```
context ord begin
```

```
primcorec lmerge :: 'a llist  $\Rightarrow$  'a llist  $\Rightarrow$  'a llist
where
```

```
lmerge xs ys =
(case xs of LNil  $\Rightarrow$  LNil | LCons x xs'  $\Rightarrow$ 
  case ys of LNil  $\Rightarrow$  LNil | 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)
by(simp add: lmerge-def)
```

```
lemma lmerge-eq-LNil-iff: lmerge xs ys = LNil  $\longleftrightarrow$  (xs = LNil  $\vee$  ys = LNil)
using lnull-lmerge unfolding lnull-def .
```

```
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)
by(auto split: llist.split)
```

```
lemma ttl-lmerge:
  [ $\neg$  lnull xs;  $\neg$  lnull ys]  $\Longrightarrow$ 
  ttl (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))
by(auto split: llist.split)

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))
by(rule llist.expand)(simp-all add: lhd-lmerge ltl-lmerge)

lemma lmerge-LNil [simp]:
lmerge LNil ys = LNil
lmerge xs LNil = LNil
by simp-all

lemma lprefix-lmergeI:
[| lprefix xs xs'; lprefix ys ys' |]
implies lprefix (lmerge xs ys) (lmerge xs' ys')
by(coinduction arbitrary: xs xs' ys ys')(fastforce simp add: lhd-lmerge ltl-lmerge
dest: lprefix-lhdD lprefix-lnullD simp add: not-lnull-conv split: if-split-asm)

lemma [partial-function-mono]:
assumes F: mono-list F and G: mono-list G
shows mono-list (λf. lmerge (F f) (G f))
by(blast intro: monotoneI lprefix-lmergeI monotoneD[OF F] monotoneD[OF G])

lemma in-lset-lmergeD: x ∈ lset (lmerge xs ys) implies x ∈ lset xs ∨ x ∈ lset ys
by(induct zs≡lmerge xs ys arbitrary: xs ys rule: llist-set-induct)(auto simp add:
lhd-lmerge ltl-lmerge split: if-split-asm dest: in-lset-ltd)

lemma lset-lmerge: lset (lmerge xs ys) ⊆ lset xs ∪ lset ys
by(auto dest: in-lset-lmergeD)

lemma lfinitelmergeD: lfinitelmerge xs ys implies lfinitelmerge xs ∨ lfinitelmerge ys
by(induct zs≡lmerge xs ys arbitrary: xs ys rule: lfinitel-induct)(fastforce simp add:
ltl-lmerge if-pull2 split: if-split-asm)+

lemma fixes F
defines F ≡ λlmerge (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-list (λlmerge. F lmerge xs) (is ?mono xs)
proof(intro eq-reflection ext)

```

```

show mono:  $\bigwedge xs. ?\text{mono } xs$  unfolding F-def by(tactic ‹Partial-Function.mono-tac
@{context} 1›)
  fix xs ys
  show lmerge xs ys = ?rhs xs ys
  proof(coinduction arbitrary: xs ys)
    case Eq-llist thus ?case
      by(subst (1 3 4) llist.mono-body-fixp[OF mono])(auto simp add: F-def lmerge-simps
split: llist.split)
    qed
  qed

lemma monotone-lmerge: monotone (rel-prod lprefix lprefix) lprefix (case-prod lmerge)
apply(rule llist.fixp-preserves-mono2[OF lmerge-mono lmerge-conv-fixp])
apply(erule conjE|rule allI conjI cont-intro|simp|erule allE, erule llist.mono2mono)+
done

lemma mono2mono-lmerge1 [THEN llist.mono2mono, cont-intro, simp]:
  shows monotone-lmerge1: monotone lprefix lprefix ( $\lambda xs. \text{lmerge } xs \ ys$ )
  by(simp add: monotone-lmerge[simplified])

lemma mono2mono-lmerge2 [THEN llist.mono2mono, cont-intro, simp]:
  shows monotone-lmerge2: monotone lprefix lprefix ( $\lambda ys. \text{lmerge } xs \ ys$ )
  by(simp add: monotone-lmerge[simplified])

lemma mcont-lmerge: mcont (prod-lub lSup lSup) (rel-prod lprefix lprefix) lSup
lprefix (case-prod lmerge)
apply(rule llist.fixp-preserves-mcont2[OF lmerge-mono lmerge-conv-fixp])
apply(erule conjE|rule allI conjI cont-intro|simp|erule allE, erule llist.mcont2mcont)+
done

lemma mcont2mcont-lmerge1 [THEN llist.mcont2mcont, cont-intro, simp]:
  shows mcont-lmerge1: mcont lSup lprefix lSup lprefix ( $\lambda xs. \text{lmerge } xs \ ys$ )
  by(simp add: mcont-lmerge[simplified])

lemma mcont2mcont-lmerge2 [THEN llist.mcont2mcont, cont-intro, simp]:
  shows mcont-lmerge2: mcont lSup lprefix lSup lprefix ( $\lambda ys. \text{lmerge } xs \ ys$ )
  by(simp add: mcont-lmerge[simplified])

lemma lfinitelmergeI [simp]:  $\llbracket \text{lfinitel } xs; \text{lfinitel } ys \rrbracket \implies \text{lfinitel } (\text{lmerge } xs \ ys)$ 
proof(induction arbitrary: ys rule: lfinitel-induct)
  case (LCons xs)
  from LCons.preds LCons.hyps LCons.IH
  show ?case by induct(clarsimp simp add: not-lnull-conv lmerge-simps)+
  qed simp

lemma linfinite-lmerge [simp]:  $\llbracket \neg \text{lfinitel } xs; \neg \text{lfinitel } ys \rrbracket \implies \neg \text{lfinitel } (\text{lmerge } xs \ ys)$ 
by(auto dest: lfinitelmergeD)

```

```

lemma llength-lmerge-above: llength xs □ llength ys ≤ llength (lmerge xs ys)
proof(induction xs arbitrary: ys)
  case (LCons x xs)
  show ?case
  proof(induct ys)
    case LCons thus ?case using LCons.IH
      by(fastforce simp add: bot-enat-def[symmetric] lmerge-simps le-infi1 le-infi2
          intro: order-trans[rotated])
    qed(simp-all add: bot-enat-def[symmetric])
    qed(simp-all add: bot-enat-def[symmetric])
  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(induction arbitrary: ys rule: lset-induct)
  case (find xs)
  then obtain y where y ∈ lset ys x ≤ y by blast
  thus ?case by induct(auto simp add: lmerge-simps not-less)
next
  case (step x' xs)
  then obtain y where y ∈ lset ys x ≤ y by blast
  moreover from <lsorted (LCons x' xs)> <x ∈ lset xs> <x ≠ x'>
  have x' < x lsorted xs by(auto simp add: lsorted-LCons)
  ultimately show ?case using <¬ lfinite ys>
    by induct(auto 4 3 simp add: lmerge-simps <x ≠ x'> not-less intro: step.IH dest:
    in-lset-lmergeD)
qed

lemma in-lset-lmergeI2:
  [ x ∈ lset ys; lsorted ys; ¬ lfinite xs; ∃ y ∈ lset xs. x ≤ y ]
  ==> x ∈ lset (lmerge xs ys)
proof(induction arbitrary: xs rule: lset-induct)
  case (find ys)
  then obtain y where y ∈ lset xs x ≤ y by blast
  thus ?case by induct(auto simp add: lmerge-simps not-less)
next
  case (step x' ys)
  then obtain y where y ∈ lset xs x ≤ y by blast
  moreover from <lsorted (LCons x' ys)> <x ∈ lset ys> <x ≠ x'>
  have x' < x lsorted ys by(auto simp add: lsorted-LCons)
  ultimately show ?case using <¬ lfinite xs>
    by induct(auto 4 3 simp add: lmerge-simps <x ≠ x'> not-less intro: step.IH dest:
    in-lset-lmergeD)
qed

```

```

lemma lsorted-lmerge:  $\llbracket \text{lsorted } xs; \text{lsorted } ys \rrbracket \implies \text{lsorted } (\text{lmerge } xs \text{ } ys)$ 
proof(coinduction arbitrary:  $xs \text{ } ys$ )
  case ( $\text{lsorted } xs \text{ } ys$ )
    have ?lhd using lsorted
      by(auto 4 3 simp add: not-lnull-conv lsorted-LCons lhd-lmerge ltl-lmerge dest!: in-lset-lmergeD)
    moreover have ?ltl using lsorted by(auto simp add: ltl-lmerge intro: lsorted-ltlI)
    ultimately show ?case ..
  qed

lemma ldistinct-lmerge:
   $\llbracket \text{lsorted } xs; \text{lsorted } ys; \text{ldistinct } xs; \text{ldistinct } ys \rrbracket$ 
   $\implies \text{ldistinct } (\text{lmerge } xs \text{ } ys)$ 
by(coinduction arbitrary:  $xs \text{ } ys$ )(auto 4 3 simp add: lhd-lmerge ltl-lmerge not-lnull-conv lsorted-LCons not-less dest!: in-lset-lmergeD dest: order.antisym)

end

partial-function (llist) hamming' :: unit  $\Rightarrow$  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]:  $\neg \text{lnull } \text{hamming}$ 
unfolding hamming-def by(subst hamming'.simp) simp

lemma hamming-eq-LNil-iff [simp]:  $\text{hamming} = \text{LNil} \longleftrightarrow \text{False}$ 
using lnull-hamming unfolding lnull-def by simp

lemma lhd-hamming [simp]:  $\text{lhd } \text{hamming} = 1$ 
unfolding hamming-def by(subst hamming'.simp) simp

lemma ltl-hamming [simp]:
   $\text{ltd } \text{hamming} = \text{lmerge } (\text{lmap } ((*) 2) \text{ } \text{hamming}) \text{ } (\text{lmerge } (\text{lmap } ((*) 3) \text{ } \text{hamming}) \text{ } (\text{lmap } ((*) 5) \text{ } \text{hamming}))$ 
unfolding hamming-def by(subst hamming'.simp) simp

lemma hamming-unfold:
   $\text{hamming} = \text{LCons } 1 \text{ } (\text{lmerge } (\text{lmap } ((*) 2) \text{ } \text{hamming}) \text{ } (\text{lmerge } (\text{lmap } ((*) 3) \text{ } \text{hamming}) \text{ } (\text{lmap } ((*) 5) \text{ } \text{hamming})))$ 
by(rule llist.expand) simp-all

definition smooth :: nat  $\Rightarrow$  bool
where smooth n  $\longleftrightarrow$  ( $\forall p. \text{prime } p \longrightarrow p \text{ dvd } n \longrightarrow p \leq 5$ )

```

```

lemma smooth-0 [simp]:  $\neg \text{smooth } 0$ 
by(auto simp add: smooth-def intro: exI[where x=7])

lemma smooth-Suc0 [simp]:  $\text{smooth} (\text{Suc } 0)$ 
by(auto simp add: smooth-def)

lemma smooth-gt0:  $\text{smooth } n \implies n > 0$ 
by(cases n) simp-all

lemma smooth-ge-Suc0:  $\text{smooth } n \implies n \geq \text{Suc } 0$ 
by(cases n) simp-all

lemma prime-nat-dvdD:  $\text{prime } p \implies (n :: \text{nat}) \text{ dvd } p \implies n = 1 \vee n = p$ 
unfolding prime-nat-iff by simp

lemma smooth-times [simp]:  $\text{smooth} (x * y) \longleftrightarrow \text{smooth } x \wedge \text{smooth } y$ 
by(auto simp add: smooth-def prime-dvd-mult-iff)

lemma smooth2 [simp]:  $\text{smooth } 2$ 
by(auto simp add: smooth-def dest: prime-nat-dvdD[of 2, simplified])

lemma smooth3 [simp]:  $\text{smooth } 3$ 
by(auto simp add: smooth-def dest: prime-nat-dvdD[of 3, simplified])

lemma smooth5 [simp]:  $\text{smooth } 5$ 
by(auto simp add: smooth-def dest: prime-nat-dvdD[of 5, simplified])

lemma hamming-in-smooth:  $\text{lset hamming} \subseteq \{n. \text{smooth } n\}$ 
unfolding hamming-def by(induct rule: hamming'.fixp-induct)(auto 6 6 dest: in-lset-lmergeD)

lemma lfinite-hamming [simp]:  $\neg \text{lfinite hamming}$ 
proof
  assume lfinite hamming
  then obtain n where n:  $\text{llength hamming} = \text{enat } n$  unfolding lfinite-conv-llength-enat
  by fastforce
  have llength (lmap ((*) 3) hamming)  $\sqcap$  llength (lmap ((*) 5) hamming)  $\leq$  llength
  (lmerge (lmap ((*) 3) hamming) (lmap ((*) 5) hamming))
  by(rule llength-lmerge-above)
  hence llength hamming  $\leq \dots$  by simp
  moreover have llength (lmap ((*) 2) hamming)  $\sqcap \dots \leq$ 
    llength (lmerge (lmap ((*) 2) hamming) (lmerge (lmap ((*) 3) hamming) (lmap
    ((*) 5) hamming)))
  by(rule llength-lmerge-above)
  ultimately have llength hamming  $\leq \dots$  by(simp add: inf.absorb1)
  also from n have ...  $< \text{enat } n$ 
  by(subst (asm) hamming-unfold)(cases n, auto simp add: zero-enat-def[symmetric]
  eSuc-enat[symmetric])
  finally show False unfolding n by simp

```

qed

```
lemma lsorted-hamming [simp]: lsorted hamming
  and ldistinct-hamming [simp]: ldistinct hamming
proof -
  have lsorted hamming ∧ ldistinct hamming ∧ lset hamming ⊆ {1..}
  unfolding hamming-def
  by(induct rule: hamming'.fixp-induct)(auto 6 6 simp add: lsorted-LCons dest:
  in-lset-lmergeD intro: smooth-ge-Suc0 lsorted-lmerge lsorted-lmap monotoneI ldistrib-lmerge inj-onI)
  thus lsorted hamming ldistinct hamming by simp-all
qed

lemma smooth-hamming:
assumes smooth n
shows n ∈ lset hamming
using assms
proof(induction n rule: less-induct)
have [simp]:
  monotone (≤) (≤) ((*) 2 :: nat ⇒ nat)
  monotone (≤) (≤) ((*) 3 :: nat ⇒ nat)
  monotone (≤) (≤) ((*) 5 :: nat ⇒ nat)
by(simp-all add: monotone-def)

case (less n)
show ?case
proof(cases n > 1)
  case False
  moreover from ⟨smooth n⟩ have n ≠ 0 by(rule contrapos-pn) simp
  ultimately have n = 1 by(simp)
  thus ?thesis by(subst hamming-unfold) simp
next
  case True
  hence ∃ p. prime p ∧ p dvd n by(metis less-numeral-extra(4) prime-factor-nat)
  with ⟨smooth n⟩ obtain p where prime p p dvd n p ≤ 5 by(auto simp add:
  smooth-def)
  from ⟨p dvd n⟩ obtain n' where n: n = p * n' by(auto simp add: dvd-def)
  with ⟨smooth n⟩ ⟨prime p⟩ have smooth n' by(simp add: smooth-def)
  with ⟨prime p⟩ n have n' < n by(simp add: smooth-gt0 prime-gt-Suc-0-nat)
  hence n': n' ∈ lset hamming using ⟨smooth n'⟩ by(rule less.IH)

  from ⟨p ≤ 5⟩ have p = 0 ∨ p = 1 ∨ p = 2 ∨ p = 3 ∨ p = 4 ∨ p = 5
  by presburger
  with ⟨prime p⟩ have p = 2 ∨ p = 3 ∨ p = 5 by auto
  thus ?thesis
  proof(elim disjE)
    assume p = 2
    with n n' show ?thesis
    by(subst hamming-unfold)(simp-all add: in-lset-lmergeI1 not-less lsorted-lmap)
```

```

bexI[where x=n] bexI[where x=3*n]
next
  assume p = 3
  moreover with n <smooth n'> have 2 * n' < n by(simp add: smooth-gt0)
  hence 2 * n' ∈ lset hamming by(rule less.IH)(simp add: <smooth n'>)
  ultimately show ?thesis using n n'
  by(subst hamming-unfold)(auto 4 4 simp add: not-less lsorted-lmap lsorted-lmerge
intro: in-lset-lmergeI1 in-lset-lmergeI2 bexI[where x=4*n] bexI[where x=5*n])
next
  assume p = 5
  moreover with n <smooth n'> have 2 * (2 * n') < n by(simp add: smooth-gt0)
  hence 2 * (2 * n') ∈ lset hamming by(rule less.IH)(simp only: smooth-times
smooth2 <smooth n'> simp-thms)
  moreover from <p = 5> n <smooth n'> have 3 * n' < n by(simp add:
smooth-gt0)
  hence 3 * n' ∈ lset hamming by(rule less.IH)(simp add: <smooth n'>)
  ultimately show ?thesis using n n'
  by(subst hamming-unfold)(auto 4 4 simp add: lsorted-lmap lsorted-lmerge
intro: in-lset-lmergeI2 bexI[where x=9*n] bexI[where x=8 * n])
qed
qed
qed

```

corollary *hamming-smooth*: *lset hamming* = {*n. smooth n*}

using *hamming-in-smooth* **by**(blast intro: *smooth-hamming*)

lemma *hamming-THE*:

(*THE xs. lsorted xs ∧ ldistinct xs ∧ lset xs = {n. smooth n}*) = *hamming*
by(rule the-equality)(simp-all add: *hamming-smooth lsorted-ldistinct-lset-unique*)

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: ( $\bigwedge d. rs d \subseteq rs' d$ )  $\implies$  Old-Datatype.Lim rs  $\subseteq$  Old-Datatype.Lim rs'  

by(simp add: Lim-def) blast
```

```
lemma Lim-UN1: Old-Datatype.Lim ( $\lambda x. \bigcup y. f x y$ ) = ( $\bigcup y. Old\text{-}Datatype.Lim (\lambda x. f x y)$ )  

by(simp add: Old-Datatype.Lim-def) blast
```

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-Datatype.dtreet)  $\Rightarrow 'c$ )  $\Rightarrow ('a, 'b)$   

Old-Datatype.dtreet  $\Rightarrow 'c$   

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-Datatype.Lim M) = f M  

by(auto simp add: DTBranch-def dest: Lim-inject)
```

Lemmas for ntrunc and Old-Datatype.Lim

```
lemma ndepth-Push-Node-Inl-aux:  

case-nat (Inl n) f k = Inr 0  $\implies$  Suc (LEAST x. f x = Inr 0)  $\leq k$   

apply (induct k, auto)  

apply (erule Least-le)  

done
```

```
lemma ndepth-Push-Node-Inl:  

ndepth (Push-Node (Inl a) n) = Suc (ndepth n)  

using Rep-Node[of n, unfolded Node-def]  

apply(simp add: ndepth-def Push-Node-def Abs-Node-inverse[OF Node-Push-I[OF Rep-Node]])  

apply(simp add: Push-def split-beta)  

apply(rule Least-equality)  

apply(auto elim: LeastI intro: ndepth-Push-Node-Inl-aux)  

done
```

```
lemma ntrunc-Lim [simp]: ntrunc (Suc k) (Old-Datatype.Lim rs) = Old-Datatype.Lim  

( $\lambda x. ntrunc k (rs x)$ )  

unfolding Lim-def ntrunc-def  

apply(rule equalityI)  

apply clarify  

apply(auto simp add: ndepth-Push-Node-Inl)  

done
```

17.2 Definition for the codatatype universe

Constructors

```
definition TERMINAL :: ' $a \Rightarrow ('c + 'b + 'a, 'd)$  Old-Datatype.dtreet  

where TERMINAL a = In0 (Old-Datatype.Leaf (Inr (Inr a)))
```

```

definition LINEAR :: 'b  $\Rightarrow$  ('c + 'b + 'a, 'd) Old-Datatype.dtree  $\Rightarrow$  ('c + 'b +
'a, 'd') Old-Datatype.dtree
where LINEAR b r = In1 (In0 (Scons (Old-Datatype.Leaf (Inr (Inl b))) r))

definition BRANCH :: 'c  $\Rightarrow$  ('d  $\Rightarrow$  ('c + 'b + 'a, 'd') Old-Datatype.dtree)  $\Rightarrow$  ('c
+ 'b + 'a, 'd') Old-Datatype.dtree
where BRANCH c rs = In1 (In1 (Scons (Old-Datatype.Leaf (Inl c)) (Old-Datatype.Lim
rs)))

case operator

definition case-RESUMPTION :: ('a  $\Rightarrow$  'e)  $\Rightarrow$  ('b  $\Rightarrow$  (('c + 'b + 'a, 'd') Old-Datatype.dtree)
 $\Rightarrow$  'e)  $\Rightarrow$  ('c  $\Rightarrow$  ('d  $\Rightarrow$  ('c + 'b + 'a, 'd') Old-Datatype.dtree)  $\Rightarrow$  'e)  $\Rightarrow$  ('c + 'b +
'a, 'd') Old-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 ( $\lambda x.$  l (inv (Old-Datatype.Leaf
      o Inr o Inl x)))))
      (Old-Datatype.Split ( $\lambda x.$  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'
  by(auto simp add: TERMINAL-def LINEAR-def BRANCH-def dest: Lim-inject)

```

```

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
  apply(simp-all add: case-RESUMPTION-def TERMINAL-def LINEAR-def BRANCH-def
o-def)
  apply(rule arg-cong) back
  apply(blast intro: injI inv-f-f)
  apply(rule arg-cong) back
  apply(blast intro: injI inv-f-f)
  apply(rule arg-cong) back
  apply(blast intro: injI inv-f-f)

```

done

lemma *LINEAR-mono*: $r \subseteq r' \implies \text{LINEAR } b \ r \subseteq \text{LINEAR } b \ r'$
by(simp add: LINEAR-def In1-mono In0-mono Scons-mono)

lemma *BRANCH-mono*: $(\bigwedge d. rs \ d \subseteq rs' \ d) \implies \text{BRANCH } c \ rs \subseteq \text{BRANCH } c \ rs'$
by(simp add: BRANCH-def In1-mono Scons-mono Lim-mono)

lemma *LINEAR-UN*: $\text{LINEAR } b \ (\bigcup x. f \ x) = (\bigcup x. \text{LINEAR } b \ (f \ x))$
by (simp add: LINEAR-def In1-UN1 In0-UN1 Scons-UN1-y)

lemma *BRANCH-UN*: $\text{BRANCH } b \ (\lambda d. \bigcup x. f \ d \ x) = (\bigcup x. \text{BRANCH } b \ (\lambda d. f \ d \ x))$
by (simp add: BRANCH-def Lim-UN1 In1-UN1 In0-UN1 Scons-UN1-y)

The codatatype universe

coinductive-set *resumption* :: ('c + 'b + 'a, 'd) Old-Datatype.dtreet set
where
resumption-Terminal:
 Terminal a ∈ *resumption*
| *resumption-LINEAR*:
 r ∈ *resumption* ⇒ *LINEAR* b r ∈ *resumption*
| *resumption-BRANCH*:
 ($\bigwedge d. rs \ d \in \text{resumption}$) ⇒ *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.dtreet set
proof
 show *Terminal* undefined ∈ ?*resumption* by(blast intro: *resumption.intros*)
qed

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'
apply(auto simp add: Terminal-def Linear-def Branch-def simp add: Rep-resumption
resumption.intros Abs-resumption-inject Rep-resumption-inject)
apply(subst (asm) fun-eq-if, auto simp add: Rep-resumption-inject)
done

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))
by(simp-all add: Terminal-def Linear-def Branch-def Abs-resumption-inverse re-
sumption.intros Rep-resumption)

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 (λb r. l b (Abs-resumption r)) (λc rs. br c (λd. 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
by(simp-all add: Terminal-def Linear-def Branch-def case-resumption-def Abs-resumption-inverse
resumption.intros Rep-resumption Rep-resumption-inverse)

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)
  using assms by simp-all

code-datatype Terminal Linear Branch

setup ‹Code.declare-case-global @{thm case-resumption-cert}›
setup ‹

```

```

Nitpick.register-codatatype @{typ ('a,'b,'c,'d) resumption} @{const-name case-resumption}
  (map dest-Const [@{term Terminal}, @{term Linear},
  @{term Branch}])
  ›

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(cases x)
    case (Abs-resumption y)
    note [simp] = ‹x = Abs-resumption y›
    from ‹y ∈ resumption› show thesis
    proof(cases rule: resumption.cases)
      case resumption-TERMINAL thus ?thesis
        by -(rule Terminal, simp add: Terminal-def)
      next
        case (resumption-LINEAR r b)
        from ‹r ∈ resumption› have Rep-resumption (Abs-resumption r) = r
          by(simp add: Abs-resumption-inverse)
        hence y = LINEAR b (Rep-resumption (Abs-resumption r))
          using ‹y = LINEAR b r› by simp
        thus ?thesis by -(rule Linear, simp add: Linear-def)
      next
        case (resumption-BRANCH rs c)
        from ‹∀d. rs d ∈ resumption›
        have eq: rs = (λd. Rep-resumption (Abs-resumption (rs d)))
          by(subst Abs-resumption-inverse) blast+
        show ?thesis using ‹y = BRANCH c rs›
          by -(rule Branch, simp add: Branch-def, subst eq, simp)
      qed
    qed

lemma resumption-split:
  P (case-resumption t l br r) ↔
    (forall a. r = Terminal a → P (t a)) ∧
    (forall b r'. r = Linear b r' → P (l b r')) ∧
    (forall c rs. r = Branch c rs → P (br c rs))
  by(cases r) simp-all

lemma resumption-split-asm:
  P (case-resumption t l br r) ↔
    ¬ ((exists a. r = Terminal a ∧ ¬ P (t a)) ∨
      (exists b r'. r = Linear b r' ∧ ¬ P (l b r')) ∨
      (exists c rs. r = Branch c rs ∧ ¬ P (br c rs)))
  by(cases r) simp-all

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.dtreetree
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 (λd. 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.dtreetree
where
  RESUMPTION-corec f e = (UNION 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 (λd. RESUMPTION-corec f (es d))
  | Resumption-corec r => Rep-resumption r)
  (is ?lhs = ?rhs)
proof
  show ?lhs ⊆ ?rhs unfolding RESUMPTION-corec-def
  proof(rule UN-least)
    fix n
    show RESUMPTION-corec-aux n f e
      ⊆ (case f e of Terminal-corec a => TERMINAL a
      | Linear-corec b e' => LINEAR b (UNION n. RESUMPTION-corec-aux n f e')
      | Branch-corec c es => BRANCH c (λd. UNION n. RESUMPTION-corec-aux n
        f (es d))
      | Resumption-corec r => Rep-resumption r)
    apply(cases n, simp-all split: resumption-corec.split)
    by(rule conjI strip LINEAR-mono[OF UN-upper] BRANCH-mono[OF UN-upper]
      UNIV-I)+
    qed
  next
    show ?rhs ⊆ ?lhs unfolding RESUMPTION-corec-def
    apply(simp split: resumption-corec.split add: LINEAR-UN BRANCH-UN)
    by safe(rule-tac a=Suc n for n in UN-I, rule UNIV-I, simp)+
  qed

```

```

lemma RESUMPTION-corec-type: RESUMPTION-corec f e ∈ resumption
proof –
  have ∃ x. RESUMPTION-corec f e = RESUMPTION-corec f x by blast
  thus ?thesis
  proof coinduct
    case (resumption x)
    then obtain e where x: x = RESUMPTION-corec f e by blast
    thus ?case
    proof(cases f e)
      case (Resumption-corec r)
      thus ?thesis using x
        by(cases r)(simp-all add: RESUMPTION-corec Rep-resumption-constructors
        Rep-resumption)
      qed(auto simp add: RESUMPTION-corec)
    qed
  qed

```

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 (λd. resumption-corec f (es d))
   | Resumption-corec r ⇒ r)
unfolding resumption-corec-def
apply(subst RESUMPTION-corec)
apply(auto split: resumption-corec.splits simp add: Terminal-def Linear-def Branch-def
RESUMPTION-corec-type Abs-resumption-inverse Rep-resumption-inverse)
done

```

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: (Λd. 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(induct k arbitrary: r r' rule: less-induct)
  case (less k)

```

```

note  $IH = \langle \bigwedge k' r r'. [k' < k; Eq\text{-}RESUMPTION } r r' ] \implies ntrunc k' r = ntrunc k' r' \rangle$ 
from  $\langle Eq\text{-}RESUMPTION } r r' \rangle$  show ?case
proof cases
  case EqTERMINAL
  thus ?thesis by simp
next
  case (EqLINEAR R R' b)
  thus ?thesis unfolding LINEAR-def
    apply(cases k, simp)
    apply(rename-tac k', case-tac k', simp)
    apply(rename-tac k'', case-tac k'', simp-all add: IH)
    done
next
  case (EqBRANCH rs rs' c)
  thus ?thesis unfolding BRANCH-def
    apply(cases k, simp)
    apply(rename-tac k', case-tac k', simp)
    apply(rename-tac k'', case-tac k'', simp)
    apply(rename-tac k''', case-tac k''', simp-all)
    apply(rule arg-cong[where f=Old-Datatype.Lim])
    apply(rule ext)
    apply(simp add: IH)
    done
qed
qed

lemma Eq-RESUMPTION-refl:
assumes  $r \in resumption$ 
shows Eq-RESUMPTION  $r r$ 
proof -
  define  $r'$  where  $r' = r$ 
  with assms have  $r = r' \wedge r \in resumption$  by auto
  thus Eq-RESUMPTION  $r r'$ 
  proof(coinduct)
    case (Eq-RESUMPTION  $r r'$ )
    hence [simp]:  $r = r'$  and  $r \in resumption$  by auto
    from  $\langle r \in resumption \rangle$  show ?case
      by(cases rule: resumption.cases) auto
  qed
qed

lemma Eq-RESUMPTION-into-resumption:
assumes Eq-RESUMPTION  $r r$ 
shows  $r \in resumption$ 
using assms
proof(coinduct)
  case resumption thus ?case by cases auto
qed

```

```

lemma Eq-RESUMPTION-eq:
  Eq-RESUMPTION r r'  $\longleftrightarrow$  r = r'  $\wedge$  r  $\in$  resumption
proof(rule iff)
  assume Eq-RESUMPTION r r'
  hence  $\bigwedge k. ntrunc k r = ntrunc k r'$  by(rule Eq-RESUMPTION-implies-ntrunc-equality)
  hence r = r' by(rule ntrunc-equality)
  moreover with ⟨Eq-RESUMPTION r r'⟩ have r  $\in$  resumption
    by(auto intro: Eq-RESUMPTION-into-resumption)
  ultimately show r = r'  $\wedge$  r  $\in$  resumption ..
next
  assume r = r'  $\wedge$  r  $\in$  resumption
  thus Eq-RESUMPTION r r' by(blast intro: Eq-RESUMPTION-refl)
qed

lemma Eq-RESUMPTION-I [consumes 1, case-names Eq-RESUMPTION, case-conclusion
Eq-RESUMPTION EqTerminal EqLinear EqBranch]:
assumes X r r'
and step:  $\bigwedge r r'. X r r' \implies$ 
  ( $\exists a. r = TERMINAL a \wedge r' = TERMINAL a$ )  $\vee$ 
  ( $\exists R R' b. r = LINEAR b R \wedge r' = LINEAR b R' \wedge (X R R' \vee$ 
  Eq-RESUMPTION R R'))  $\vee$ 
  ( $\exists rs rs' c. r = BRANCH c rs \wedge r' = BRANCH c rs' \wedge (\forall d. X (rs d)$ 
  (rs' d)  $\vee$  Eq-RESUMPTION (rs d) (rs' d)))
shows r = r'
proof –
  from ⟨X r r'⟩ have Eq-RESUMPTION r r'
  by(coinduct)(rule step)
  thus ?thesis by(simp add: Eq-RESUMPTION-eq)
qed

lemma resumption-equalityI [consumes 1, case-names Eq-resumption, case-conclusion
Eq-resumption EqTerminal EqLinear EqBranch]:
assumes X r r'
and step:  $\bigwedge r r'. X r r' \implies$ 
  ( $\exists a. r = Terminal a \wedge r' = Terminal a$ )  $\vee$ 
  ( $\exists R R' b. r = Linear b R \wedge r' = Linear b R' \wedge (X R R' \vee R = R')$ )  $\vee$ 
  ( $\exists rs rs' c. r = Branch c rs \wedge r' = Branch c rs' \wedge (\forall d. X (rs d) (rs'$ 
  d)  $\vee$  rs d = rs' d))
shows r = r'
proof –
  define M N where M = Rep-resumption r and N = Rep-resumption r'
  with ⟨X r r'⟩ have  $\exists r r'. M = Rep-resumption r \wedge N = Rep-resumption r' \wedge$ 
  X r r' by blast
  hence M = N
  proof(coinduct rule: Eq-RESUMPTION-I)
    case (Eq-RESUMPTION M N)
    then obtain r r' where [simp]: M = Rep-resumption r N = Rep-resumption
    r'

```

```

and  $X r r'$  by blast
{ assume  $\exists a. r = \text{Terminal } a \wedge r' = \text{Terminal } a$ 
  hence ?EqTerminal by(auto simp add: Rep-resumption-constructors)
  hence ?case .. }
moreover
{ assume  $\exists R R' b. r = \text{Linear } b R \wedge r' = \text{Linear } b R' \wedge (X R R' \vee R = R')$ 
  hence ?EqLinear
    by(clarsimp simp add: Rep-resumption-constructors Eq-RESUMPTION-eq
      Rep-resumption-inject Rep-resumption)
  hence ?case by blast }
moreover
{ assume  $\exists rs rs' c. r = \text{Branch } c rs \wedge r' = \text{Branch } c rs' \wedge (\forall d. X (rs d) (rs' d) \vee rs d = rs' d)$ 
  hence ?EqBranch
    by(clarsimp simp add: Rep-resumption-constructors Eq-RESUMPTION-eq
      Rep-resumption-inject Rep-resumption)
  hence ?case by blast }
ultimately show ?case using step[OF ‹X r r›] by blast
qed
thus ?thesis unfolding M-def N-def by(simp add: Rep-resumption-inject)
qed

```

Finality of *resumption*: Uniqueness of functions defined by corecursion.

```

lemma equals-RESUMPTION-corec:
assumes h:  $\bigwedge x. h x = (\text{case } f x \text{ of Terminal-corec } a \Rightarrow \text{TERMINAL } a$ 
           | Linear-corec b x'  $\Rightarrow \text{LINEAR } b (h x')$ 
           | Branch-corec c xs  $\Rightarrow \text{BRANCH } c (\lambda d. h (xs d))$ 
           | Resumption-corec r  $\Rightarrow \text{Rep-resumption } r)$ 
shows h = RESUMPTION-corec f
proof
  fix x
  define h' where h' = RESUMPTION-corec f
  have h':  $\bigwedge x. h' x = (\text{case } f x \text{ of Terminal-corec } a \Rightarrow \text{TERMINAL } a$ 
           | Linear-corec b x'  $\Rightarrow \text{LINEAR } b (h' x')$ 
           | Branch-corec c xs  $\Rightarrow \text{BRANCH } c (\lambda d. h' (xs d))$ 
           | Resumption-corec r  $\Rightarrow \text{Rep-resumption } r)$ 
  unfolding h'-def by(simp add: RESUMPTION-corec)
  define M N where M = h x and N = h' x
  hence  $\exists x. M = h x \wedge N = h' x$  by blast
  thus M = N
  proof(coinduct rule: Eq-RESUMPTION-I)
    case (Eq-RESUMPTION M N)
    then obtain x where [simp]: M = h x N = h' x by blast
    show ?case
    proof(cases f x)
      case (Terminal-corec a)
      with h h' have ?EqTerminal by simp
      thus ?thesis ..
    next

```

```

case (Linear-corec b x')
with h h' have ?EqLinear by auto
thus ?thesis by blast
next
  case (Branch-corec c xs)
  with h h' have ?EqBranch by auto
  thus ?thesis by blast
next
  case (Resumption-corec r)
  thus ?thesis
  by(cases r)(simp-all add: h h' Rep-resumption-constructors Eq-RESUMPTION-refl
Rep-resumption)
qed
qed
qed

lemma equals-resumption-corec:
assumes h:  $\lambda x. h x = (\text{case } f x \text{ of Terminal-corec } a \Rightarrow \text{Terminal } a$ 
 $| \text{Linear-corec } b x' \Rightarrow \text{Linear } b (h x')$ 
 $| \text{Branch-corec } c xs \Rightarrow \text{Branch } c (\lambda d. h (xs d))$ 
 $| \text{Resumption-corec } r \Rightarrow r)$ 
shows h = resumption-corec f
proof(rule ext)
fix x
{ fix x
from h[of x]
have Rep-resumption (h x) =
(case f x of Terminal-corec a => TERMINAL a
| Linear-corec b x' => LINEAR b (Rep-resumption (h x'))
| Branch-corec c xs => BRANCH c (\lambda d. Rep-resumption (h (xs d)))
| Resumption-corec r => Rep-resumption r)
by(auto split: resumption-corec.split simp add: Rep-resumption-constructors)
}
hence eq: ( $\lambda x. \text{Rep-resumption } (h x)$ ) = RESUMPTION-corec f by(rule equals-RESUMPTION-corec)
hence Abs-resumption (Rep-resumption (h x)) = Abs-resumption (RESUMPTION-corec
f x)
by(subst (asm) fun-eq-iff)(auto)
from this[symmetric] show h x = resumption-corec f x
  unfolding resumption-corec-def by(simp add: Rep-resumption-inverse)
qed

end

```

```

theory Coinductive-Examples imports
LList-CCPO-Topology
TLLList-CCPO-Examples
Koenigslemma
LMirror

```

*Hamming-Stream
Resumption*

begin

end