

Coinductive

Andreas Lochbihler
with contributions by Johannes Hölzl

October 10, 2017

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	5
1.1	Case operator	6
1.2	Corecursion for <i>enat</i>	7
1.3	Less as greatest fixpoint	10
1.4	Equality as greatest fixpoint	12
1.5	Uniqueness of corecursion	13
1.6	Setup for <code>partial_function</code>	13
1.7	Misc.	19
2	Coinductive lists and their operations	20
2.1	Auxiliary lemmata	21
2.2	Type definition	21
2.3	Properties of predefined functions	23
2.4	The subset of finite lazy lists <i>lfinite</i>	25

2.5	Concatenating two lists: <i>lappend</i>	26
2.6	The prefix ordering on lazy lists: <i>lprefix</i>	29
2.7	Setup for partial_function	32
2.8	Monotonicity and continuity of already defined functions	38
2.9	More function definitions	41
2.10	Converting ordinary lists to lazy lists: <i>llist-of</i>	44
2.11	Converting finite lazy lists to ordinary lists: <i>list-of</i>	45
2.12	The length of a lazy list: <i>llength</i>	47
2.13	Taking and dropping from lazy lists: <i>ltake</i> , <i>ldropn</i> , and <i>ldrop</i>	49
2.14	Taking the <i>n</i> -th element of a lazy list: <i>lnth</i>	60
2.15	<i>iterates</i>	65
2.16	More on the prefix ordering on lazy lists: <i>op</i> \sqsubseteq and <i>lstrict-prefix</i>	67
2.17	Length of the longest common prefix	71
2.18	Ziping two lazy lists to a lazy list of pairs <i>lzip</i>	72
2.19	Taking and dropping from a lazy list: <i>ltakeWhile</i> and <i>ldrop-While</i>	77
2.20	<i>llist-all2</i>	84
2.21	The last element <i>llast</i>	90
2.22	Distinct lazy lists <i>ldistinct</i>	92
2.23	Sortedness <i>lsorted</i>	95
2.24	Lexicographic order on lazy lists: <i>llexord</i>	99
2.25	The filter functional on lazy lists: <i>lfilter</i>	106
2.26	Concatenating all lazy lists in a lazy list: <i>lconcat</i>	113
2.27	Sublist view of a lazy list: <i>lnths</i>	122
2.28	<i>llist-sum</i>	126
2.29	Alternative view on ' <i>a</i> <i>llist</i> as datatype with constructors <i>llist-of</i> and <i>inf-llist</i>	127
2.30	Setup for lifting and transfer	129
	2.30.1 Relator and predicator properties	129
	2.30.2 Transfer rules for the Transfer package	129
3	Instantiation of the order type classes for lazy lists	133
3.1	Instantiation of the order type class	133
3.2	Prefix ordering as a lower semilattice	134
4	Infinite lists as a codatatype	136
4.1	Lemmas about operations from <i>Stream</i>	137
4.2	Link ' <i>a</i> <i>stream</i> to ' <i>a</i> <i>llist</i>	137
4.3	Link ' <i>a</i> <i>stream</i> with <i>nat</i> \Rightarrow ' <i>a</i>	142
4.4	Function iteration <i>siterate</i> and <i>sconst</i>	143
4.5	Counting elements	143
4.6	First index of an element	145
4.7	<i>stakeWhile</i>	147

5	Terminated coinductive lists and their operations	147
5.1	Auxiliary lemmas	148
5.2	Type definition	148
5.3	Code generator setup	149
5.4	Connection with <i>'a llist</i>	150
5.5	Library function definitions	155
5.6	<i>tfinite</i>	155
5.7	The terminal element <i>terminal</i>	155
5.8	<i>tmap</i>	156
5.9	Appending two terminated lazy lists <i>tappend</i>	156
5.10	Appending a terminated lazy list to a lazy list <i>lappendt</i> . . .	157
5.11	Filtering terminated lazy lists <i>tfilter</i>	157
5.12	Concatenating a terminated lazy list of lazy lists <i>tconcat</i> . . .	158
5.13	<i>tllist-all2</i>	159
5.14	From a terminated lazy list to a lazy list <i>llist-of-tllist</i>	162
5.15	The nth element of a terminated lazy list <i>tnth</i>	163
5.16	The length of a terminated lazy list <i>tlength</i>	163
5.17	<i>tdropn</i>	164
5.18	<i>tset</i>	164
5.19	Setup for Lifting/Transfer	165
	5.19.1 Relator and predicator properties	165
	5.19.2 Transfer rules for the Transfer package	165
6	Setup for Isabelle's quotient package for lazy lists	169
6.1	Rules for the Quotient package	169
7	Setup for Isabelle's quotient package for terminated lazy lists	172
7.1	Rules for the Quotient package	172
8	Code generator setup to implement lazy lists lazily	178
8.1	Lazy lists	178
9	Code generator setup to implement terminated lazy lists lazily	184
10	CCPO topologies	189
10.1	The filter <i>at'</i>	190
10.2	The type class <i>ccpo-topology</i>	191
10.3	Instances for <i>ccpo-topologys</i> and continuity theorems	194
11	A CCPO topology on lazy lists with examples	196
11.1	Continuity and closedness of predefined constants	196
11.2	Define <i>lfilter</i> as continuous extension	203
11.3	Define <i>lconcat</i> as continuous extension	204

11.4	Define <i>ldropWhile</i> as continuous extension	205
11.5	Define <i>ldrop</i> as continuous extension	206
11.6	Define more functions on lazy lists as continuous extensions	207
12	Ccpo structure for terminated lazy lists	213
12.1	The ccpo structure	215
12.2	Continuity of predefined constants	220
12.3	Definition of recursive functions	224
13	Example definitions using the CCPO structure on terminated lazy lists	225
14	Example: Koenig's lemma	227
15	Definition of the function lmirror	233
16	The Hamming stream defined as a least fixpoint	236
17	Manual construction of a resumption codatatype	244
17.1	Auxiliary definitions and lemmata similar to <i>Old-Datatype</i>	244
17.2	Definition for the codatatype universe	245
17.3	Definition of the codatatype as a type	247

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  $a \neq \infty \implies a + b - a = b$ 
by(cases b) auto

lemma enat-the-enat:  $n \neq \infty \implies \text{enat} (\text{the-enat } n) = n$ 
by auto

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

lemma enat-le-plus-same:  $x \leq (x :: \text{enat}) + y \iff x \leq y + x$ 
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:  $n \neq \infty \implies \text{the-enat} (\text{eSuc } n) = \text{Suc} (\text{the-enat } n)$ 
by(cases n)(simp-all add: eSuc-enat)

coinductive-set enat-set :: enat set
where  $0 \in \text{enat-set}$ 
  |  $n \in \text{enat-set} \implies (\text{eSuc } n) \in \text{enat-set}$ 

lemma enat-set-eq-UNIV [simp]: enat-set = UNIV
proof
  show  $\text{enat-set} \subseteq \text{UNIV}$  by blast
  show  $\text{UNIV} \subseteq \text{enat-set}$ 
  proof
    fix x :: enat
    assume  $x \in \text{UNIV}$ 
```

```

thus  $x \in \text{enat-set}$ 
proof coinduct
  case (enat-set  $x$ )
  show ?case
  proof(cases  $x = 0$ )
    case True thus ?thesis by simp
  next
  case False
  then obtain  $n$  where  $x = \text{eSuc } n$ 
    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  $n = \text{eSuc } n'$ 
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 neg-zero-conv-eSuc:  $n \neq 0 \iff (\exists n'. n = \text{eSuc } n')$ 
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* $::\ enat \Rightarrow enat$ **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*: $\text{numeral } k = \text{eSuc } n \longleftrightarrow \text{epred-numeral } k = n$
and *Suc-eq-numeral*: $\text{eSuc } n = \text{numeral } k \longleftrightarrow n = \text{epred-numeral } k$
and *less-numeral-Suc*: $\text{numeral } k < \text{eSuc } n \longleftrightarrow \text{epred-numeral } k < n$
and *less-eSuc-numeral*: $\text{eSuc } n < \text{numeral } k \longleftrightarrow n < \text{epred-numeral } k$
and *le-numeral-eSuc*: $\text{numeral } k \leq \text{eSuc } n \longleftrightarrow \text{epred-numeral } k \leq n$
and *le-eSuc-numeral*: $\text{eSuc } n \leq \text{numeral } k \longleftrightarrow n \leq \text{epred-numeral } k$
and *diff-eSuc-numeral*: $\text{eSuc } n - \text{numeral } k = n - \text{epred-numeral } k$
and *diff-numeral-eSuc*: $\text{numeral } k - \text{eSuc } n = \text{epred-numeral } k - n$
and *max-eSuc-numeral*: $\text{max } (\text{eSuc } n) (\text{numeral } k) = \text{eSuc } (\text{max } n (\text{epred-numeral } k))$
and *max-numeral-eSuc*: $\text{max } (\text{numeral } k) (\text{eSuc } n) = \text{eSuc } (\text{max } (\text{epred-numeral } k) n)$
and *min-eSuc-numeral*: $\text{min } (\text{eSuc } n) (\text{numeral } k) = \text{eSuc } (\text{min } n (\text{epred-numeral } k))$
and *min-numeral-eSuc*: $\text{min } (\text{numeral } k) (\text{eSuc } n) = \text{eSuc } (\text{min } (\text{epred-numeral } k) n)$
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*: $\text{epred } 1 = 0$
and *epred-numeral*: $\text{epred } (\text{numeral } i) = \text{epred-numeral } i$
and *epred-Infty*: $\text{epred } \infty = \infty$
and *epred-enat*: $\text{epred } (\text{enat } m) = \text{enat } (m - 1)$
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*: $a \neq 0 \implies \text{epred } (a + b) = \text{epred } a + b$
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]*: $\text{epred } (\text{min } a b) = \text{min } (\text{epred } a) (\text{epred } b)$
by(*cases a b rule: enat-coexhaust[case-product enat-coexhaust]*) *simp-all*

lemma *epred-le-epredI*: $n \leq m \implies \text{epred } n \leq \text{epred } m$
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*]:
 $\text{monotone } (\text{fun-ord } (\lambda y x. x \leq y)) (\lambda y x. x \leq y) (\lambda f. \text{eSuc } (f x))$
by (*auto simp: monotone-def fun-ord-def*)

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

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

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

lemma *epred-enat-unfold* [*simp*]:
 $\text{epred } (\text{enat-unfold stop next } a) = (\text{if stop } a \text{ then } 0 \text{ else } \text{enat-unfold stop next } (\text{next } a))$
by(*simp add: enat-unfold-next*)

lemma *epred-max*: $\text{epred } (\text{max } x y) = \text{max } (\text{epred } x) (\text{epred } y)$
by(*cases x y rule: enat.exhaust*[*case-product enat.exhaust*]) *simp-all*

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

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

lemma *epred-Sup*: $\text{epred } (\text{Sup } A) = \text{Sup } (\text{epred } \text{' } A)$
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* :: $(\text{enat} \times \text{enat}) \text{ set}$

where

Le-enat-zero: $(0, n) \in \text{Le-enat}$
| *Le-enat-add*: $\llbracket (m, n) \in \text{Le-enat}; k \neq 0 \rrbracket \implies (\text{eSuc } m, n + k) \in \text{Le-enat}$

lemma *ile-into-Le-enat*:

$m \leq n \implies (m, n) \in \text{Le-enat}$

proof –

assume $m \leq n$

hence $(m, n) \in \{(m, n) \mid m \leq n\}$ **by** *simp*

thus *?thesis*

proof *coinduct*

case $(\text{Le-enat } m \ n)$

hence $m \leq n$ **by** *simp*

show *?case*

proof(*cases m = 0*)

case *True*

hence *?Le-enat-zero* **by** *simp*

thus *?thesis ..*

next

case *False*

with $\langle m \leq n \rangle$ **obtain** $m' \ n'$ **where** $m = \text{eSuc } m' \ n = n' + 1 \ m' \leq n'$

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*:

$(m, n) \in \text{Le-enat} \implies n < \text{enat } l \implies m < \text{enat } l$

proof(*induct l arbitrary: m n*)

case *0*

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

next

case $(\text{Suc } l)$

from $\langle (m, n) \in \text{Le-enat} \rangle$ **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 ⟹ 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 ⟹ m ≤ n
by(rule enat-less-imp-le)(erule Le-enat-imp-ile-enat-k)

```

```

lemma Le-enat-eq-ile:
  (m, n) ∈ Le-enat ⟷ m ≤ 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:
    ∧m n. (m, n) ∈ X
    ⟹ m = 0 ∨ (∃ m' n' k. m = eSuc m' ∧ n = n' + enat k ∧ k ≠ 0 ∧
      ((m', n') ∈ X ∨ m' ≤ 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:
    ∧m n. P m n
    ⟹ (n = 0 ⟶ m = 0) ∧
      (m ≠ 0 ⟶ n ≠ 0 ⟶ (∃ k n'. P (epred m) n' ∧ epred n = n' + k) ∨
      epred m ≤ 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  $n \neq 0$  and disj:  $(\exists k\ n'. P\ m'\ n' \wedge \text{epred } n = n' + k) \vee m' \leq \text{epred } n$ 
by auto
  from  $\langle n \neq 0 \rangle$  obtain  $n'$  where  $n': n = \text{eSuc } n'$ 
  by(cases n rule: enat-coexhaust) auto
  from disj show ?thesis
  proof
    assume  $m' \leq \text{epred } n$ 
    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  $\exists k\ n'. P\ m'\ n' \wedge \text{epred } n = n' + k$ 
    then obtain  $k\ n''$  where  $n'': \text{epred } n = n'' + k$  and  $k: P\ m'\ n''$  by blast
    show ?thesis using eSuc k n' n''
    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:  $(m, n) \in X$ 
and step:
   $\bigwedge m\ n. (m, n) \in X$ 
   $\implies m = 0 \wedge n = 0 \vee (\exists m'\ n'. m = \text{eSuc } m' \wedge n = \text{eSuc } n' \wedge ((m', n') \in$ 
 $X \vee m' = n'))$ 
shows  $m = n$ 
proof(rule antisym)
  from major show  $m \leq n$ 
  by(coinduct rule: enat-leI)
  (drule step, auto simp add: eSuc-plus-1 enat-I[symmetric])

  from major have  $(n, m) \in \{(n, m). (m, n) \in X\}$  by simp
  thus  $n \leq m$ 
proof(coinduct rule: enat-leI)
  case (Leenat n m)

```

hence $(m, n) \in X$ **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: $\bigwedge m n. P m n \implies (m = 0 \longleftrightarrow n = 0) \wedge (m \neq 0 \longrightarrow n \neq 0 \longrightarrow P (epred m) (epred n) \vee epred m = epred n)$*
shows $m = n$
proof –
from *major* **have** $(m, n) \in \{(m, n). P m n\}$ **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*]:

$\llbracket P m n; \bigwedge m n. P m n \implies m = 0 \longleftrightarrow n = 0; \bigwedge m n. \llbracket P m n; m \neq 0; n \neq 0 \rrbracket \implies P (epred m) (epred n) \vee epred m = epred n \rrbracket$
 $\implies m = n$
by(*erule enat-coinduct*) *blast*

1.5 Uniqueness of corecursion

lemma *enat-unfold-unique*:

assumes $h: !!x. h x = (\text{if stop } x \text{ then } 0 \text{ else } eSuc (h (next x)))$
shows $h x = \text{enat-unfold stop next } x$
by(*coinduction arbitrary: x rule: enat-coinduct*)(*subst (1 3) h, auto*)

1.6 Setup for partial function

lemma *enat-diff-cancel-left*: $\llbracket m \leq x; m \leq y \rrbracket \implies x - m = y - m \longleftrightarrow x = (y$

$:: \text{enat})$

by(*cases x y m rule: enat.exhaust[case-product enat.exhaust[case-product enat.exhaust]]*)(*simp-all, arith*)

lemma *finite-lessThan-enatI*:

assumes $m \neq \infty$
shows *finite* $\{..<m :: \text{enat}\}$

proof –

from *assms* **obtain** m' **where** $m: m = \text{enat } m'$ **by** *auto*

have $\{..<enat\ m'\} \subseteq enat \text{ ‘ } \{..<m'\}$
by(rule subsetI)(case-tac x, auto)
thus ?thesis **unfolding** m **by**(rule finite-subset) simp
qed

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

have range enat $\subseteq \{..<\infty\}$ **by** auto
moreover **assume** finite $\{..<\infty :: enat\}$
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*:

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

lemma *enat-minus-mono1*: $x \leq y \implies x - m \leq y - m \text{ (} m :: enat \text{)}$

by(cases m x y rule: enat.exhaust[case-product enat.exhaust[case-product enat.exhaust]])
simp-all

lemma *max-enat-minus1*: $max\ n\ m - k = max\ (n - k)\ (m - k) \text{ (} m - k :: enat \text{)}$

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 (($\lambda n :: enat. n - m$) ‘ 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 $\bigsqcup A - m = \bigsqcup ((\lambda n :: enat. n - m) \text{ ‘ } A)$

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 (($\lambda y :: enat. x + y$) ‘ Y) = x + Sup Y
proof(cases finite Y)

```

case True
have  $op + x \text{ ' } Y = \{x + m \mid m. m \in Y\}$  by auto
thus ?thesis using True by(simp add: Sup-enat-def add-Max-commute assms)
next
case False
thus ?thesis
proof(cases x)
  case (enat x')
    hence  $\neg \text{finite } (op + x \text{ ' } Y)$  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  $op + x \text{ ' } Y = \{\infty\}$  using assms by auto
    thus ?thesis using infinity by(simp add: image-constant-conv assms)
qed
qed

```

lemma *Sup-image-eadd2*:

```

 $Y \neq \{\}$   $\implies \text{Sup } ((\lambda y :: \text{enat. } y + x) \text{ ' } Y) = \text{Sup } Y + x$ 
by(subst (1 2) add.commute)(rule Sup-image-eadd1)

```

lemma *mono2mono-eSuc* [*THEN lfp.mono2mono, cont-intro, simp*]:

```

shows monotone-eSuc: monotone op  $\leq$  op  $\leq$  eSuc
by(rule monotoneI) simp

```

lemma *mcont2mcont-eSuc* [*THEN lfp.mcont2mcont, cont-intro, simp*]:

```

shows mcont-eSuc: mcont Sup op  $\leq$  Sup op  $\leq$  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 op  $\leq$  op  $\leq$  epred
by(rule monotoneI)(simp add: epred-le-epredI)

```

lemma *mcont2mcont-epred* [*THEN lfp.mcont2mcont, cont-intro, simp*]:

```

shows mcont-epred: mcont Sup op  $\leq$  Sup op  $\leq$  epred
by(simp add: mcont-def monotone-epred cont-def epred-Sup)

```

lemma *enat-cocase-mono* [*partial-function-mono, cont-intro*]:

```

 $\llbracket \text{monotone } \text{orda } \text{ordb } \text{zero}; \bigwedge n. \text{monotone } \text{orda } \text{ordb } (\lambda f. \text{esuc } f \ n) \rrbracket$ 
 $\implies \text{monotone } \text{orda } \text{ordb } (\lambda f. \text{co.case-enat } (\text{zero } f) (\text{esuc } f) \ x)$ 
by(cases x rule: co.enat.exhaust) simp-all

```

lemma *enat-cocase-mcont* [*cont-intro, simp*]:

```

 $\llbracket \text{mcont } \text{luba } \text{orda } \text{lubb } \text{ordb } \text{zero}; \bigwedge n. \text{mcont } \text{luba } \text{orda } \text{lubb } \text{ordb } (\lambda f. \text{esuc } f \ n) \rrbracket$ 
 $\implies \text{mcont } \text{luba } \text{orda } \text{lubb } \text{ordb } (\lambda f. \text{co.case-enat } (\text{zero } f) (\text{esuc } f) \ x)$ 
by(cases x rule: co.enat.exhaust) simp-all

```

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

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

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

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

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

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

lemma *mcont-eadd2*: $\text{mcont } Sup \text{ } op \leq \text{ } Sup \text{ } op \leq (\lambda y. x + y :: \text{enat})$
by(*auto intro: mcontI monotone-eadd2 contI Sup-image-eadd1[symmetric]*)

lemma *mcont-eadd1*: $\text{mcont } Sup \text{ } op \leq \text{ } Sup \text{ } op \leq (\lambda x. x + y :: \text{enat})$
by(*auto intro: mcontI monotone-eadd1 contI Sup-image-eadd2[symmetric]*)

lemma *mcont2mcont-eadd* [*cont-intro, simp*]:
 $\llbracket \text{mcont } lub \text{ } ord \text{ } Sup \text{ } op \leq (\lambda x. f x); \text{mcont } lub \text{ } ord \text{ } Sup \text{ } op \leq (\lambda x. g x) \rrbracket$
 $\implies \text{mcont } lub \text{ } ord \text{ } Sup \text{ } op \leq (\lambda x. f x + g x :: \text{enat})$
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*]:
 $\llbracket \text{monotone } (\text{fun-ord } op \leq) \text{ } op \leq f; \text{monotone } (\text{fun-ord } op \leq) \text{ } op \leq g \rrbracket$
 $\implies \text{monotone } (\text{fun-ord } op \leq) \text{ } op \leq (\lambda x. f x + g x :: \text{enat})$
by(*rule mono2mono-eadd*)

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

lemma *monotone-max-enat2*: $\text{monotone } op \leq \text{ } op \leq (\lambda y. \text{max } x \text{ } y :: \text{enat})$
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 op ≤ op ≤) op ≤ (λ(x, y). max
x y :: 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 finite-subset[rotated]) auto
  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 op ≤ Sup op ≤ (λx. max x y :: enat)
by(auto intro!: mcontI contI max-Sup-enat1 simp add: monotone-max-enat1)

lemma mcont-max-enat2: mcont Sup op ≤ Sup op ≤ (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 op ≤ (λx. f x);*
 mcont lub ord Sup op ≤ (λx. g x)]
 ⇒ *mcont lub ord Sup op ≤ (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*]:
 [*monotone (fun-ord op ≤) op ≤ f; monotone (fun-ord op ≤) op ≤ g*]
 ⇒ *monotone (fun-ord op ≤) op ≤ (λx. max (f x) (g x) :: enat)*
by(*rule mono2mono-max-enat*)

lemma *chain-epredI*:
Complete-Partial-Order.chain op ≤ Y
 ⇒ *Complete-Partial-Order.chain op ≤ (epred ‘ (Y ∩ {x. x ≠ 0}))*
by(*auto intro: chainI dest: chainD*)

lemma *monotone-enat-le-case*:
fixes *bot*
assumes *mono: monotone op ≤ ord (λx. f x (eSuc x))*
and *ord: ∧x. ord bot (f x (eSuc x))*
and *bot: ord bot bot*
shows *monotone op ≤ ord (λx. case x of 0 ⇒ bot | eSuc x' ⇒ f x' x)*
proof –
have *monotone op ≤ ord (λx. if x ≤ 0 then bot else f (epred x) x)*
proof(*rule monotone-if-bot*)
 fix *x y :: enat*
 assume *x ≤ y ∧ x ≤ 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 *¬ x ≤ 0*
 thus *ord bot (f (epred x) x)*
 by(*cases x rule: co.enat.exhaust*)(*auto intro: ord*)
qed(*rule bot*)
also have *(λx. if x ≤ 0 then bot else f (epred x) x) = (λx. case x of 0 ⇒ bot |*
eSuc x' ⇒ 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 op ≤ lub ord (λx. f x (eSuc x))*
and *ord: ∧x. ord bot (f x (eSuc x))*

```

shows  $mcont\ Sup\ op \leq lub\ ord\ (\lambda x. case\ x\ of\ 0 \Rightarrow bot \mid eSuc\ x' \Rightarrow f\ x'\ x)$ 
proof -
  from ccpo
  have  $mcont\ Sup\ op \leq lub\ ord\ (\lambda x. if\ x \leq 0\ then\ bot\ else\ f\ (epred\ x)\ x)$ 
  proof(rule mcont-if-bot)
    fix  $x\ y :: enat$ 
    assume  $x \leq y \wedge x \leq 0$ 
    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: mcont-monoD[OF mcont])
  next
    fix  $Y :: enat\ set$ 
    assume chain: Complete-Partial-Order.chain op ≤ Y
    and  $Y: Y \neq \{\} \wedge x. x \in Y \Rightarrow \neg x \leq 0$ 
    from  $Y$  have  $Y': Y \cap \{x. x \neq 0\} \neq \{\}$  by auto
    from  $Y(2)$  have  $eq: Y = eSuc\ ' (epred\ ' (Y \cap \{x. x \neq 0\}))$ 
    by(fastforce intro: rev-image-eqI)
    let  $?Y = epred\ ' (Y \cap \{x. x \neq 0\})$ 
    from chain-epredI [OF chain] Y'
    have  $f\ (\bigsqcup\ ?Y)\ (eSuc\ (\bigsqcup\ ?Y)) = lub\ ((\lambda x. f\ x\ (eSuc\ x))\ ' ?Y)$ 
    using mcont [THEN mcont-contD] by blast
    moreover from chain-epredI [OF chain] Y'
    have  $SUPREMUM\ ?Y\ eSuc = eSuc\ (\bigsqcup\ ?Y)$ 
    using mcont-eSuc [THEN mcont-contD, symmetric] by blast
    ultimately show  $f\ (epred\ (Sup\ Y))\ (Sup\ Y) = lub\ ((\lambda x. f\ (epred\ x)\ x)\ ' Y)$ 
    by (subst (1 2 3) eq (simp add: image-image))
  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
  also have  $(\lambda x. if\ x \leq 0\ then\ bot\ else\ f\ (epred\ x)\ x) = (\lambda x. case\ x\ of\ 0 \Rightarrow bot \mid eSuc\ x' \Rightarrow f\ x'\ x)$ 
  by(auto simp add: fun-eq-iff split: co.enat.split)
  finally show ?thesis .
qed

```

1.7 Misc.

lemma *enat-add-mono [simp]*:

$enat\ x + y < enat\ x + z \iff y < z$

by(*cases y*)(*case-tac [!] z, simp-all*)

lemma *enat-add1-eq [simp]*: $enat\ x + y = enat\ x + z \iff y = z$

by (*metis enat-add-mono add commute neq-iff*)

lemma *enat-add2-eq [simp]*: $y + enat\ x = z + enat\ x \iff y = z$

by (*metis enat-add1-eq add commute*)

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

lemma *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*: $e\text{Suc } x \leq y \iff (\exists y'. y = e\text{Suc } y' \wedge x \leq y')$
by(*cases y rule: co.enat.exhaust*) *simp-all*

lemma *eSuc-eq-infinity-iff*: $e\text{Suc } n = \infty \iff n = \infty$
by(*cases n*)(*simp-all add: zero-enat-def eSuc-enat*)

lemma *infinity-eq-eSuc-iff*: $\infty = e\text{Suc } n \iff n = \infty$
by(*cases n*)(*simp-all add: zero-enat-def eSuc-enat*)

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

lemma *eSuc-Inf*: $e\text{Suc } (\text{Inf } A) = \text{Inf } (e\text{Suc } ` A)$

proof –

{ **assume** $A \neq \{\}$

then obtain a **where** $a \in A$ **by** *blast*

then have $e\text{Suc } (\text{LEAST } a. a \in A) = (\text{LEAST } a. a \in e\text{Suc } ` A)$

proof (*rule LeastI2-wellorder*)

fix a **assume** $a \in A$ **and** $b: \forall b. b \in A \longrightarrow a \leq b$

then have $a: e\text{Suc } a \in e\text{Suc } ` A$

by *auto*

then show $e\text{Suc } a = (\text{LEAST } a. a \in e\text{Suc } ` A)$

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*]: $(\text{Suc } ^n) m = m + n$
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 $b \leq a$ **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*) (*ltl: 'a llist*)

for

map: lmap

rel: llist-all2

where

lhd LNil = undefined

| *ltl LNil = LNil*

Coiterator setup.

primcorec *unfold-llist* :: $('a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'b \text{ llist}$

where

$p a \implies \text{unfold-llist } p \ g21 \ g22 \ a = \text{LNil}$ |

$- \implies \text{unfold-llist } p \ g21 \ g22 \ a = \text{LCons } (g21 \ a) (\text{unfold-llist } p \ g21 \ g22 \ (g22 \ a))$

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* ($\lambda\cdot$. *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*:

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

proof

fix x

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

qed

lemma *lmap-unfold-llist*:

$lmap\ f\ (unfold-llist\ IS-LNIL\ LHD\ LTL\ b) = unfold-llist\ IS-LNIL\ (f \circ LHD)\ LTL\ b$

by(*coinduction arbitrary: b*) *auto*

lemma *lmap-corec-llist*:

$lmap\ f\ (corec-llist\ IS-LNIL\ LHD\ endORMore\ TTL-end\ TTL-more\ b) =$
 $corec-llist\ IS-LNIL\ (f \circ LHD)\ endORMore\ (lmap\ f \circ TTL-end)\ TTL-more\ b$

by(*coinduction arbitrary: b rule: llist.coinduct-strong*) *auto*

lemma *unfold-llist-ltl-unroll*:

$unfold-llist\ IS-LNIL\ LHD\ LTL\ (LTL\ b) = unfold-llist\ (IS-LNIL \circ LTL)\ (LHD \circ$

$LTL)\ LTL\ b$

by(*coinduction arbitrary: b*) *auto*

lemma *ltl-unfold-llist*:

ltl ($unfold_llist$ $IS-LNIL$ LHD LTL a) =
 (if $IS-LNIL$ a then $LNil$ else $unfold_llist$ $IS-LNIL$ LHD LTL (LTL a))
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 $llist.set_sel(1)$ [$simp$]

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** $\bigwedge xs. \neg lnull$ $xs \implies P$ (lhd xs) xs
and $\bigwedge xs y. [\neg lnull$ $xs; y \in lset$ (ltl xs); P y (ltl xs)] $\implies P$ 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*:
 $x \in lset (LCons x xs)$
 $x \in lset xs \implies x \in lset (LCons x' xs)$
by *simp-all*

lemma *lset-cases* [*elim?*]:
assumes $x \in lset xs$
obtains xs' **where** $xs = LCons x xs'$
 $| x' xs'$ **where** $xs = LCons x' xs' \wedge x \in lset xs'$
using *assms* **by** (*cases xs*) *auto*

lemma *lset-induct'* [*consumes 1, case-names find step*]:
assumes *major*: $x \in lset xs$
and *1*: $\bigwedge xs. P (LCons x xs)$
and *2*: $\bigwedge x' xs. \llbracket x \in lset xs; P xs \rrbracket \implies P (LCons x' xs)$
shows $P xs$
using *major* **apply** (*induct y \equiv 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*: $x \in lset xs$
and *find*: $\bigwedge xs. P (LCons x xs)$
and *step*: $\bigwedge x' xs. \llbracket x \in lset xs; x \neq x'; P xs \rrbracket \implies P (LCons x' xs)$
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*: $lnull xs \implies lset xs = \{\}$
by (*auto dest: llist.collapse*)

Alternative definition of *lset* for nitpick

inductive *lsetp* :: $'a \text{ llist} \Rightarrow 'a \Rightarrow \text{bool}$

where

$lsetp (LCons x xs) x$
| $lsetp xs x \implies lsetp (LCons x' xs) x$

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* :: $'a \text{ set} \Rightarrow 'a \text{ llist} \Rightarrow 'a \text{ set}$

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 *lmember* :: $'a \Rightarrow 'a \text{ llist} \Rightarrow \text{bool}$

where $lmember x xs \longleftrightarrow x \in lset xs$

lemma *lmember-code* [*code*]:

$lmember x LNil \longleftrightarrow \text{False}$
 $lmember x (LCons y ys) \longleftrightarrow x = y \vee lmember x ys$
by(*simp-all add: lmember-def*)

lemma *lset-lmember* [*code-unfold*]:

$x \in lset xs \longleftrightarrow lmember x xs$
by(*simp add: lmember-def*)

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

2.4 The subset of finite lazy lists *lfinite*

inductive *lfinite* :: $'a \text{ llist} \Rightarrow \text{bool}$

where

lfinite-LNil: *lfinite LNil*
| *lfinite-LConsI*: *lfinite xs* \implies *lfinite (LCons x xs)*

declare *lfinite-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*: $\bigwedge xs. [\text{lfinite } xs; \neg \text{lnull } xs; P \text{ (ltl } xs)] \implies P \text{ } xs$
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 $\equiv$ 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*: $\neg \text{lfinite } xs \implies \text{lappend } xs \text{ } ys = xs$
by(*coinduction arbitrary: xs*) *auto*

lemma *lfinite-lmap* [*simp*]:
 $\text{lfinite } (\text{lmap } f \text{ } xs) = \text{lfinite } xs$
(**is** *?lhs* \longleftrightarrow *?rhs*)

proof

assume *?lhs*

thus *?rhs*

by(*induct zs* \equiv *lmap f xs arbitrary: xs rule: lfinite-induct*) *fastforce+*

next

assume *?rhs*

thus *?lhs* **by**(*induct*) *simp-all*

qed

lemma *lset-lappend-lfinite* [*simp*]:
 $\text{lfinite } xs \implies \text{lset } (\text{lappend } xs \text{ } ys) = \text{lset } xs \cup \text{lset } ys$
by(*induct rule: lfinite.induct*) *auto*

lemma *lset-lappend*: $\text{lset } (\text{lappend } xs \text{ } ys) \subseteq \text{lset } xs \cup \text{lset } ys$

proof(*cases lfinite xs*)

case *True*

thus *?thesis* **by** *simp*

next

case *False*

thus *?thesis* **by**(*auto simp add: lappend-inf*)

qed

lemma *lset-lappend1*: $\text{lset } xs \subseteq \text{lset } (\text{lappend } xs \text{ } ys)$
by(*rule subsetI*)(*erule lset-induct, simp-all*)

lemma *lset-lappend-conv*: $\text{lset } (\text{lappend } xs \text{ } ys) = (\text{if } \text{lfinite } xs \text{ then } \text{lset } xs \cup \text{lset } ys \text{ else } \text{lset } xs)$

by(*simp add: lappend-inf*)

lemma *in-lset-lappend-iff*: $x \in \text{lset } (\text{lappend } xs \text{ } ys) \longleftrightarrow x \in \text{lset } xs \vee \text{lfinite } xs \wedge x \in \text{lset } ys$

by(*simp add: lset-lappend-conv*)

lemma *split-llist-first*:

assumes $x \in \text{lset } xs$

shows $\exists ys \text{ } zs. xs = \text{lappend } ys \text{ } (\text{LCons } x \text{ } zs) \wedge \text{lfinite } ys \wedge x \notin \text{lset } ys$

using *assms*

proof(*induct*)

case *find* **thus** *?case* **by**(*auto intro: exI*[**where** $x = \text{LNil}$])

next

case *step* **thus** *?case* **by**(*fastforce intro: exI*[**where** $x = \text{LCons } a \text{ } b$ **for** $a \text{ } b$])

qed

lemma *split-llist*: $x \in \text{lset } xs \implies \exists ys \ zs. \ xs = \text{lappend } ys \ (\text{LCons } x \ zs) \wedge \text{lfinite } ys$
by(*blast dest: split-llist-first*)

2.6 The prefix ordering on lazy lists: *lprefix*

coinductive *lprefix* :: 'a llist \Rightarrow 'a llist \Rightarrow bool (**infix** \sqsubseteq 65)

where

$LNil$ -*lprefix* [*simp, intro!*]: $LNil \sqsubseteq xs$
| *Le-LCons*: $xs \sqsubseteq ys \implies LCons \ x \ xs \sqsubseteq LCons \ x \ ys$

lemma *lprefixI* [*consumes 1, case-names lprefix, case-conclusion lprefix LeLNil LeLCons*]:

assumes *major*: $(xs, ys) \in X$

and step:

$\bigwedge xs \ ys. (xs, ys) \in X$
 $\implies \text{lnull } xs \vee (\exists x \ xs' \ ys'. \ xs = LCons \ x \ xs' \wedge \ ys = LCons \ x \ ys' \wedge$
 $((xs', ys') \in X \vee xs' \sqsubseteq ys'))$

shows $xs \sqsubseteq ys$

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: $\bigwedge xs \ ys. P \ xs \ ys$

$\implies (\text{lnull } ys \longrightarrow \text{lnull } xs) \wedge$

$(\neg \text{lnull } xs \longrightarrow \neg \text{lnull } ys \longrightarrow \text{lhd } xs = \text{lhd } ys \wedge (P \ (\text{ltl } xs) \ (\text{ltl } ys) \vee \text{ltl } xs \sqsubseteq \text{ltl } ys))$

shows $xs \sqsubseteq ys$

proof –

from *major have* $(xs, ys) \in \{(xs, ys). P \ xs \ ys\}$ **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*]: $xs \sqsubseteq xs$

by(*coinduction arbitrary: xs simp-all*)

lemma *lprefix-LNil* [*simp*]: $xs \sqsubseteq LNil \iff \text{lnull } xs$

unfolding *lnull-def* **by**(*subst lprefix.simps*)*simp*

lemma *lprefix-lnull*: $\text{lnull } ys \implies xs \sqsubseteq ys \iff \text{lnull } xs$

unfolding *lnull-def* **by** *auto*

lemma *lnull-lprefix*: $lnull\ xs \implies lprefix\ xs\ ys$
by(*simp add: lnull-def*)

lemma *lprefix-LCons-conv*:
 $xs \sqsubseteq LCons\ y\ ys \longleftrightarrow$
 $xs = LNil \vee (\exists xs'. xs = LCons\ y\ xs' \wedge xs' \sqsubseteq ys)$
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(*clarsimp 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 op  $\sqsubseteq$  (mk-less op  $\sqsubseteq$ )
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  $x \in lset\ xs$ 
  thus  $x \in lset\ ys$  using assms
  by(induct arbitrary: ys)(auto simp add: LCons-lprefix-conv)
qed

lemma lprefix-lappend-sameI:
  assumes  $xs \sqsubseteq ys$ 
  shows  $lappend\ zs\ xs \sqsubseteq lappend\ zs\ ys$ 
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:  $\neg lfinite\ xs$ 
  shows  $xs \sqsubseteq ys \iff xs = ys$ 
proof
  assume  $xs \sqsubseteq ys$ 
  with nfin show  $xs = ys$ 
  by(coinduction arbitrary: xs ys)(auto dest: lprefix-lnullD lprefix-lhdD intro:
  lprefix-ltlI)
qed simp

lemma lprefix-lappend:  $xs \sqsubseteq lappend\ xs\ ys$ 
by(coinduction arbitrary: xs) auto

lemma lprefix-down-linear:
  assumes  $xs \sqsubseteq zs\ ys \sqsubseteq zs$ 
  shows  $xs \sqsubseteq ys \vee ys \sqsubseteq xs$ 
proof(rule disjCI)
  assume  $\neg ys \sqsubseteq xs$ 
  with assms show  $xs \sqsubseteq ys$ 
  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]:
   $lappend\ xs\ ys \sqsubseteq lappend\ xs\ zs \iff (lfinite\ xs \implies ys \sqsubseteq zs)$ 
  (is ?lhs  $\iff$  ?rhs)
proof

```

```

assume ?lhs
show ?rhs
proof
  assume lfinite xs
  thus ys  $\sqsubseteq$  zs using ⟨?lhs⟩ by(induct) simp-all
qed
next
  assume ?rhs
  show ?lhs
  proof(cases lfinite xs)
    case True
      hence yszs: ys  $\sqsubseteq$  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 \Rightarrow 'a llist

where

```

lSup A =
  (if  $\forall x \in A. \text{lnull } x$  then LNil
   else LCons (THE x. x  $\in$  lhd ' (A  $\cap$  {xs.  $\neg$  lnull xs})) (lSup (ltl ' (A  $\cap$  {xs.  $\neg$ 
lnull xs}))))

```

declare lSup.simps[*simp del*]

lemma lnull-lSup [simp]: lnull (lSup A) \longleftrightarrow ($\forall x \in A. \text{lnull } x$)
by(*simp* *add*: lSup-def)

lemma lhd-lSup [simp]: $\exists x \in A. \neg \text{lnull } x \implies \text{lhd } (lSup A) = (\text{THE } x. x \in \text{lhd ' (A} \cap \{xs. \neg \text{lnull } xs\}))$
by(*auto* *simp* *add*: lSup-def)

lemma ltl-lSup [simp]: ltl (lSup A) = lSup (ltl ' (A \cap {xs. \neg lnull xs}))
by(*cases* $\forall xs \in A. \text{lnull } xs$)(*auto* 4 3 *simp* *add*: lSup-def *intro*: llist.expand)

lemma lhd-lSup-eq:

```

assumes chain: Complete-Partial-Order.chain op  $\sqsubseteq$  Y
shows  $\llbracket xs \in Y; \neg \text{lnull } xs \rrbracket \implies \text{lhd } (lSup Y) = \text{lhd } xs$ 
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*: $(LCons\ x\ 'A \cap \{xs.\ \neg\ lnull\ xs\}) = LCons\ x\ 'A$

by *auto*

lemma *lSup-LCons*: $A \neq \{\} \implies lSup\ (LCons\ x\ 'A) = LCons\ x\ (lSup\ A)$

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*:

$lSup\ Y = LCons\ x\ xs \iff (\exists x \in Y.\ \neg\ lnull\ x) \wedge x = (THE\ x.\ x \in lhd\ ' (Y \cap \{xs.\ \neg\ lnull\ xs\})) \wedge xs = lSup\ (ltl\ ' (Y \cap \{xs.\ \neg\ lnull\ xs\}))$

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 \in Y*)(*simp-all add: insert-absorb*)

lemma *chain-lprefix-ltl*:

assumes *chain*: *Complete-Partial-Order.chain op* $\sqsubseteq A$

shows *Complete-Partial-Order.chain op* $\sqsubseteq (ltl\ ' (A \cap \{xs.\ \neg\ lnull\ xs\}))$

by(*auto intro!: chainI dest: chainD[OF chain] intro: lprefix-ltlI*)

lemma *lSup-finite-prefixes*: $lSup\ \{ys.\ ys \sqsubseteq xs \wedge lfinite\ ys\} = xs$ (**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 $\neg\ lnull\ xs \implies ltl\ ' (\{ys.\ ys \sqsubseteq xs \wedge lfinite\ ys\} \cap \{xs.\ \neg\ lnull\ xs\}) = \{ys.\ ys \sqsubseteq ltl\ xs \wedge lfinite\ ys\}$

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 $zs \sqsubseteq xs$ *lfinite zs*

shows $lSup\ \{ys.\ ys \sqsubseteq xs \wedge zs \sqsubseteq ys \wedge lfinite\ ys\} = xs$

using (*lfinite zs*) ($zs \sqsubseteq xs$)

proof(*induction arbitrary: xs*)

case *lfinite-LNil*

thus *?case* **by**(*simp add: lSup-finite-prefixes*)

next

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

lemma *lSup-strict-prefixes*:

$\neg lfinite\ xs \implies lSup\ \{ys. ys \sqsubseteq xs \wedge ys \neq xs\} = xs$
(is $- \implies lSup\ (?C\ xs) = -$)

proof (*coinduction arbitrary: xs*)

case (*Eq-llist xs*)

then obtain $x\ x'\ xs'$ **where** $xs: xs = LCons\ x\ (LCons\ x'\ xs') \neg lfinite\ xs'$

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 $\neg lnull\ (lSup\ \{ys. ys \sqsubseteq xs \wedge ys \neq xs\})$ **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 $ltl\ ' (\{ys. ys \sqsubseteq xs \wedge ys \neq xs\} \cap \{xs. \neg lnull\ xs\}) = \{ys. ys \sqsubseteq ltl\ xs \wedge ys \neq ltl\ xs\}$

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-llist* **by** *clarsimp*

qed

lemma *chain-lprefix-lSup*:

$[Complete-Partial-Order.chain\ op \sqsubseteq A; xs \in A]$

$\implies xs \sqsubseteq lSup\ A$

proof (*coinduction arbitrary: xs A*)

case (*lprefix xs A*)

note $chain = \langle Complete-Partial-Order.chain\ op \sqsubseteq A \rangle$

from $\langle xs \in A \rangle$ **show** $?case$

by (*auto 4 3 dest: chainD[OF chain] lprefix-lhdD intro: chain-lprefix-ltl[OF*

chain] *intro!*: *the-equality*[*symmetric*])
qed

lemma *chain-lSup-lprefix*:

$\llbracket \text{Complete-Partial-Order.chain } op \sqsubseteq A; \bigwedge xs. xs \in A \implies xs \sqsubseteq zs \rrbracket$
 $\implies lSup A \sqsubseteq zs$

proof(*coinduction arbitrary*: *A zs*)

case (*lprefix A zs*)

note *chain* = $\langle \text{Complete-Partial-Order.chain } op \sqsubseteq A \rangle$

from $\langle \forall xs. xs \in A \longrightarrow xs \sqsubseteq zs \rangle$

show *?case*

by(*auto* 4 4 *dest*: *lprefix-lnullD lprefix-lhdD intro*: *chain-lprefix-lll*[*OF chain*]
lprefix-lllI rev-image-eqI intro!: *the-equality*)

qed

lemma *lList-ccpo* [*simp, cont-intro*]: *class.ccpo lSup op* \sqsubseteq (*mk-less op* \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 op \sqsubseteq *lSup*

by(*unfold-locales*)(*auto dest*: *lprefix-antisym intro*: *lprefix-trans chain-lprefix-lSup*
chain-lSup-lprefix)

interpretation *lList*: *partial-function-definitions op* \sqsubseteq *lSup*

rewrites *lSup* $\{\}$ \equiv *LNil*

by(*rule lList-partial-function-definitions*)(*simp*)

abbreviation *mono-lList* \equiv *monotone (fun-ord op* \sqsubseteq) *op* \sqsubseteq

interpretation *lList-lift*: *partial-function-definitions fun-ord lprefix fun-lub lSup*

rewrites *fun-lub lSup* $\{\}$ \equiv $\lambda\cdot$. *LNil*

by(*rule lList-partial-function-definitions*[*THEN partial-function-lift*])(*simp*)

abbreviation *mono-lList-lift* \equiv *monotone (fun-ord (fun-ord lprefix)) (fun-ord lprefix)*

lemma *lprefixes-chain*:

Complete-Partial-Order.chain op \sqsubseteq $\{ys. lprefix ys xs\}$

by(*rule chainI*)(*auto dest*: *lprefix-down-linear*)

lemma *lList-gen-induct*:

assumes *adm*: *ccpo.admissible lSup op* \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 $ys: \bigwedge ys. [zs \sqsubseteq ys; ys \sqsubseteq xs; \textit{lfinite} \textit{ys}] \implies P \textit{ys}$ **by** *blast*
let $?C = \{ys. ys \sqsubseteq xs \wedge zs \sqsubseteq ys \wedge \textit{lfinite} \textit{ys}\}$
from *lprefixes-chain*[*of xs*]
have *Complete-Partial-Order.chain* $op \sqsubseteq ?C$
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 op \sqsubseteq P*
and *LNil: P LNil*
and *LCons: \bigwedge x xs. [lfinite xs; P xs] \implies P (LCons x xs)*
shows $P \textit{xs}$
proof –
{ **fix** $ys :: 'a \textit{llist}$
assume *lfinite ys*
hence $P \textit{ys}$ **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. LCons x (A f))
by(*rule monotoneI*)(*auto dest: monotoneD*)

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

lemma *mcont2mcont-LCons* [*THEN llist.mcont2mcont, simp, cont-intro*]:
shows *mcont-LCons: mcont lSup op \sqsubseteq lSup op \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 op \sqsubseteq op \sqsubseteq ltl*
by(*auto intro: monotoneI lprefix-ltlI*)

lemma *cont-ltl: cont lSup op \sqsubseteq lSup op \sqsubseteq ltl*
proof(*rule contI*)
fix $Y :: 'a \textit{llist set}$
assume $Y \neq \{\}$
have $ltl (lSup Y) = lSup (\textit{insert} \textit{LNil} (ltl ' (Y \cap \{xs. \neg \textit{lnull} \textit{xs}\})))$

by(*simp add: lSup-insert-LNil*)
also have $\text{insert LNil } (ltl \text{ ' } (Y \cap \{xs. \neg \text{lnull } xs\})) = \text{insert LNil } (ltl \text{ ' } Y)$ **by**
auto
also have $\text{lSup } \dots = \text{lSup } (ltl \text{ ' } Y)$ **by**(*simp add: lSup-insert-LNil*)
finally show $\text{ltl } (\text{lSup } Y) = \text{lSup } (ltl \text{ ' } Y)$.
qed

lemma *mcont2mcont-ltl* [*THEN llist.mcont2mcont, simp, cont-intro*]:
shows $\text{mcont-ltl: mcont lSup op } \sqsubseteq \text{lSup op } \sqsubseteq \text{ltl}$
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 $\text{lcons: } \bigwedge x \text{ xs. monotone orda ordb } (\lambda f. \text{lcons } f \text{ x xs})$
shows $\text{monotone orda ordb } (\lambda f. \text{case-lList } (\text{lnil } f) (\text{lcons } f) \text{ x})$
by(*rule monotoneI*)(*auto split: lList.split dest: monotoneD[OF lnil] monotoneD[OF lcons]*)

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

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

lemma *mcont-lprefix-case-aux*:
fixes *f bot*
defines $g \equiv \lambda xs. f (\text{lhd } xs) (\text{ltl } xs) (\text{LCons } (\text{lhd } xs) (\text{ltl } xs))$
assumes $\text{mcont: } \bigwedge x. \text{mcont lSup op } \sqsubseteq \text{lub ord } (\lambda xs. f \text{ x xs } (\text{LCons } x \text{ xs}))$
and *ccpo: class.ccpo lub ord (mk-less ord)*
and *bot: } \bigwedge x. \text{ord bot } x
shows $\text{mcont lSup op } \sqsubseteq \text{lub ord } (\lambda xs. \text{case xs of LNil } \Rightarrow \text{bot } \mid \text{LCons } x \text{ xs}' \Rightarrow f \text{ x xs}' \text{ xs})$
proof(*rule lList.mcont-if-bot*[**where** $f = g$ **and** $\text{bound} = \text{LNil}$ **and** $\text{bot} = \text{bot}$, *OF ccpo bot*])
fix $Y :: 'a \text{ lList set}$
assume *chain: Complete-Partial-Order.chain op } \sqsubseteq Y*
and $Y: Y \neq \{\} \bigwedge x. x \in Y \implies \neg x \sqsubseteq \text{LNil}$
from Y **have** $Y': Y \cap \{xs. \neg \text{lnull } xs\} \neq \{\}$ **by** *auto*
from Y **obtain** $x \text{ xs}$ **where** $\text{LCons } x \text{ xs} \in Y$ **by**(*fastforce simp add: not-lnull-conv*)
with $Y(2)$ **have** $\text{eq: } Y = \text{LCons } x \text{ ' } (ltl \text{ ' } (Y \cap \{xs. \neg \text{lnull } xs\}))$*

by(force dest: chainD[OF chain] simp add: LCons-lprefix-conv lprefix-LCons-conv
 intro: imageI rev-image-eqI)
show $g (lSup Y) = lub (g \text{ ` } Y)$
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 in-
 tro: mcont-monoD[OF mcont])

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

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

declaration $\ll \text{ Partial-Function.init llist } @\{\text{term llist.fixp-fun}\}$
 $@\{\text{term llist.mono-body}\} @\{\text{thm llist.fixp-rule-uc}\} @\{\text{thm llist.fixp-strong-induct-uc}\}$
 NONE \gg

2.8 Monotonicity and continuity of already defined functions

lemma fixes $f F$
defines $F \equiv \lambda \text{ lmap } xs. \text{ case } xs \text{ of } LNil \Rightarrow LNil \mid LCons x xs \Rightarrow LCons (f x) (\text{lmap } xs)$
shows $\text{lmap-conv-fixp: lmap } f \equiv \text{ccpo.fixp (fun-lub lSup) (fun-ord } op \sqsubseteq) F$ (**is** ?lhs \equiv ?rhs)
and $\text{lmap-mono: } \bigwedge xs. \text{ mono-llist } (\lambda \text{ lmap. } F \text{ lmap } xs)$ (**is** PROP ?mono)
proof(intro eq-reflection ext)
show mono: PROP ?mono **unfolding** F-def **by**(tactic $\ll \text{ Partial-Function.mono-tac } @\{\text{context}\} 1 \gg$)
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

lemma mono2mono-lmap[THEN llist.mono2mono, simp, cont-intro]:
shows monotone-lmap: monotone op \sqsubseteq op \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 op \sqsubseteq lSup op \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 ($\lambda f. \text{lmap } g (F f)$)
by(rule mono2mono-lmap)

lemma mono-llist-lappend2 [partial-function-mono]:
mono-llist A \implies mono-llist ($\lambda f. \text{lappend } xs (A f)$)
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 op \sqsubseteq op \sqsubseteq (lappend xs)
by(rule monotoneI)(rule lprefix-lappend-sameI)

lemma mcont2mcont-lappend2 [THEN llist.mcont2mcont, cont-intro, simp]:
shows mcont-lappend2: mcont lSup op \sqsubseteq lSup op \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 $\equiv \lambda \text{lset } xs. \text{case } xs \text{ of } LNil \Rightarrow \{\} \mid LCons \ x \ xs \Rightarrow \text{insert } x \ (\text{lset } xs)$
shows lset-conv-fixp: lset \equiv ccpo.fixp (fun-lub Union) (fun-ord op \sqsubseteq) F (**is** - \equiv ?fixp)

and lset-mono: $\bigwedge x. \text{monotone } (\text{fun-ord } op \sqsubseteq) \ op \sqsubseteq (\lambda f. F f x)$ (**is** PROP ?mono)
proof(rule eq-reflection ext antisym subsetI)+

show mono: PROP ?mono **unfolding** F-def **by**(tactic $\langle\langle$ Partial-Function.mono-tac @ {context} 1 $\rangle\rangle$)

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

lemma mono2mono-lset [$\text{THEN } \text{lfp.mono2mono}, \text{cont-intro}, \text{simp}$]:
shows $\text{monotone-lset: } \text{monotone } op \sqsubseteq op \subseteq \text{lset}$
by($\text{rule } \text{lfp.fixp-preserves-mono1}[\text{OF } \text{lset-mono } \text{lset-conv-fixp}]$) simp

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

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

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

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

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

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

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

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

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

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

from *assms* **have** $ccpo.admissible\ lSup\ op \sqsubseteq (\lambda ys. \neg xs \sqsubseteq ys)$ **by** *cases*
moreover **let** $?C = \{ys. ys \sqsubseteq xs \wedge ys \neq xs\}$
have $Complete-Partial-Order.chain\ op \sqsubseteq ?C$
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*]: $ccpo.compact\ lSup\ op \sqsubseteq = lfinite$
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 $l\text{-}list\text{-}of :: 'a\ list \Rightarrow 'a\ llist$

where

$l\text{-}list\text{-}of\ [] = LNil$
 $| l\text{-}list\text{-}of\ (x\#\!xs) = LCons\ x\ (l\text{-}list\text{-}of\ xs)$

definition $list\text{-}of :: 'a\ llist \Rightarrow 'a\ list$

where $[code\ del]: list\text{-}of\ xs = (if\ lfinite\ xs\ then\ inv\ l\text{-}list\text{-}of\ xs\ else\ undefined)$

definition $l\text{-}length :: 'a\ llist \Rightarrow enat$

where $[code\ del]:$

$l\text{-}length = enat\text{-}unfold\ lnull\ ltl$

primcorec $l\text{-}take :: enat \Rightarrow 'a\ llist \Rightarrow 'a\ llist$

where

$n = 0 \vee lnull\ xs \Longrightarrow lnull\ (l\text{-}take\ n\ xs)$
 $| lhd\ (l\text{-}take\ n\ xs) = lhd\ xs$
 $| ltl\ (l\text{-}take\ n\ xs) = l\text{-}take\ (epred\ n)\ (ltl\ xs)$

definition $l\text{-}dropn :: nat \Rightarrow 'a\ llist \Rightarrow 'a\ llist$

where $l\text{-}dropn\ n\ xs = (ltl\ \hat{\ }^n)\ xs$

context notes $[[function\text{-}internals]]$

begin

partial-function $(l\text{-}list)\ l\text{-}drop :: enat \Rightarrow 'a\ llist \Rightarrow 'a\ llist$

where

$l\text{-}drop\ n\ xs = (case\ n\ of\ 0 \Rightarrow xs\ | eSuc\ n' \Rightarrow case\ xs\ of\ LNil \Rightarrow LNil\ | LCons\ x\ xs' \Rightarrow l\text{-}drop\ n'\ xs')$

end

primcorec $l\text{-}take\ While :: ('a \Rightarrow bool) \Rightarrow 'a\ llist \Rightarrow 'a\ llist$

where

$lnull\ xs \vee \neg P\ (lhd\ xs) \Longrightarrow lnull\ (l\text{-}take\ While\ P\ xs)$
 $| lhd\ (l\text{-}take\ While\ P\ xs) = lhd\ xs$
 $| ltl\ (l\text{-}take\ While\ P\ xs) = l\text{-}take\ While\ P\ (ltl\ xs)$

context fixes $P :: 'a \Rightarrow bool$

notes $[[function\text{-}internals]]$

begin

partial-function $(l\text{-}list)\ l\text{-}drop\ While :: 'a\ llist \Rightarrow 'a\ llist$

where $l\text{-}drop\ While\ xs = (case\ xs\ of\ LNil \Rightarrow LNil\ | LCons\ x\ xs' \Rightarrow if\ P\ x\ then\ l\text{-}drop\ While\ xs'\ else\ xs)$

partial-function $(l\text{-}list)\ l\text{-}filter :: 'a\ llist \Rightarrow 'a\ llist$

where $l\text{-}filter\ xs = (case\ xs\ of\ LNil \Rightarrow LNil\ | LCons\ x\ xs' \Rightarrow if\ P\ x\ then\ LCons\ x$

$(\text{lfilter } xs') \text{ else } \text{lfilter } xs'$

end

primrec $\text{lnth} :: 'a \text{ llist} \Rightarrow \text{nat} \Rightarrow 'a$

where

$\text{lnth } xs \ 0 = (\text{case } xs \text{ of } LNil \Rightarrow \text{undefined } (0 :: \text{nat}) \mid LCons \ x \ xs' \Rightarrow x)$
 $\mid \text{lnth } xs \ (\text{Suc } n) = (\text{case } xs \text{ of } LNil \Rightarrow \text{undefined } (\text{Suc } n) \mid LCons \ x \ xs' \Rightarrow \text{lnth } xs' \ n)$

declare $\text{lnth.simps} \ [\text{simp } \text{del}]$

primcorec $\text{lzip} :: 'a \text{ llist} \Rightarrow 'b \text{ llist} \Rightarrow ('a \times 'b) \text{ llist}$

where

$\text{lnull } xs \vee \text{lnull } ys \Longrightarrow \text{lnull } (\text{lzip } xs \ ys)$
 $\mid \text{lhd } (\text{lzip } xs \ ys) = (\text{lhd } xs, \text{lhd } ys)$
 $\mid \text{ltl } (\text{lzip } xs \ ys) = \text{lzip } (\text{ltl } xs) \ (\text{ltl } ys)$

definition $\text{llast} :: 'a \text{ llist} \Rightarrow 'a$

where $[\text{nitpick-simp}]$:

$\text{llast } xs = (\text{case } \text{llength } xs \text{ of } \text{enat } n \Rightarrow (\text{case } n \text{ of } 0 \Rightarrow \text{undefined} \mid \text{Suc } n' \Rightarrow \text{lnth } xs \ n') \mid \infty \Rightarrow \text{undefined})$

coinductive $\text{ldistinct} :: 'a \text{ llist} \Rightarrow \text{bool}$

where

$LNil \ [\text{simp}]: \text{ldistinct } LNil$
 $\mid LCons: \llbracket x \notin \text{lset } xs; \text{ldistinct } xs \rrbracket \Longrightarrow \text{ldistinct } (LCons \ x \ xs)$

hide-fact (open) $LNil \ LCons$

definition $\text{inf-llist} :: (\text{nat} \Rightarrow 'a) \Rightarrow 'a \text{ llist}$

where $[\text{code del}]: \text{inf-llist } f = \text{lmap } f \ (\text{iterates } \text{Suc } 0)$

abbreviation $\text{repeat} :: 'a \Rightarrow 'a \text{ llist}$

where $\text{repeat} \equiv \text{iterates } (\lambda x. \ x)$

definition $\text{lstrict-prefix} :: 'a \text{ llist} \Rightarrow 'a \text{ llist} \Rightarrow \text{bool}$

where $[\text{code del}]: \text{lstrict-prefix } xs \ ys \equiv xs \sqsubseteq ys \wedge xs \neq ys$

longest common prefix

definition $\text{llcp} :: 'a \text{ llist} \Rightarrow 'a \text{ llist} \Rightarrow \text{enat}$

where $[\text{code del}]$:

$\text{llcp } xs \ ys =$
 $\text{enat-unfold } (\lambda(xs, ys). \ \text{lnull } xs \vee \text{lnull } ys \vee \text{lhd } xs \neq \text{lhd } ys) \ (\text{map-prod } \text{ltl } \text{ltl})$
 (xs, ys)

coinductive $\text{llexord} :: ('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \text{ llist} \Rightarrow 'a \text{ llist} \Rightarrow \text{bool}$

for $r :: 'a \Rightarrow 'a \Rightarrow \text{bool}$

where

$llexord-LCons-eq: llexord\ r\ xs\ ys \implies llexord\ r\ (LCons\ x\ xs)\ (LCons\ x\ ys)$
 $| llexord-LCons-less: r\ x\ y \implies llexord\ r\ (LCons\ x\ xs)\ (LCons\ y\ ys)$
 $| llexord-LNil\ [simp, intro!]: llexord\ r\ LNil\ ys$

context notes $[[function-internals]]$

begin

partial-function $(l\text{list})\ lconcat :: 'a\ l\text{list}\ l\text{list} \Rightarrow 'a\ l\text{list}$

where $lconcat\ xss = (case\ xss\ of\ LNil \Rightarrow LNil\ | LCons\ xs\ xss' \Rightarrow lappend\ xs\ (lconcat\ xss'))$

end

definition $l\text{nth}s :: 'a\ l\text{list} \Rightarrow nat\ set \Rightarrow 'a\ l\text{list}$

where $l\text{nth}s\ xs\ A = lmap\ fst\ (lfilter\ (\lambda(x, y). y \in A)\ (lzip\ xs\ (iterates\ Suc\ 0)))$

definition (in *monoid-add*) $l\text{sum-list} :: 'a\ l\text{list} \Rightarrow 'a$

where $l\text{sum-list}\ xs = (if\ lfinite\ xs\ then\ sum-list\ (list-of\ xs)\ else\ 0)$

2.10 Converting ordinary lists to lazy lists: *l\text{list-of}*

lemma $lhd-l\text{list-of}\ [simp]: lhd\ (l\text{list-of}\ xs) = hd\ xs$

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

lemma $l\text{tl-l\text{list-of}}\ [simp]: l\text{tl}\ (l\text{list-of}\ xs) = l\text{list-of}\ (tl\ xs)$

by(cases xs) simp-all

lemma $l\text{finite-l\text{list-of}}\ [simp]: l\text{finite}\ (l\text{list-of}\ xs)$

by(induct xs) auto

lemma $l\text{finite-eq-range-l\text{list-of}}: l\text{finite}\ xs \longleftrightarrow xs \in \text{range}\ l\text{list-of}$

proof

assume $l\text{finite}\ xs$

thus $xs \in \text{range}\ l\text{list-of}$

by(induct rule: *lfinite.induct*)(auto intro: *l\text{list-of}.simps[symmetric]*)

next

assume $xs \in \text{range}\ l\text{list-of}$

thus $l\text{finite}\ xs$ **by**(auto intro: *l\text{finite-l\text{list-of}}*)

qed

lemma $l\text{null-l\text{list-of}}\ [simp]: l\text{null}\ (l\text{list-of}\ xs) \longleftrightarrow xs = []$

by(cases xs) simp-all

lemma $l\text{list-of-eq-LNil-conv}$:

$l\text{list-of}\ xs = LNil \longleftrightarrow xs = []$

by(cases xs) simp-all

lemma $l\text{list-of-eq-LCons-conv}$:

$l\text{list-of}\ xs = LCons\ y\ ys \longleftrightarrow (\exists\ xs'. xs = y \# xs' \wedge ys = l\text{list-of}\ xs')$

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*: $P \text{ LNil}$

and *snoc*: $\bigwedge x \text{ xs. } \llbracket \text{lfinite } xs; P \text{ xs} \rrbracket \implies P (\text{lappend } xs (\text{LCons } x \text{ LNil}))$

shows $P \text{ xs}$

proof –

from *fin* **obtain** *xs'* **where** *xs*: $xs = \text{llist-of } xs'$

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*]: $\text{lset } (\text{llist-of } xs) = \text{set } xs$

by(induct *xs*) simp-all

lemma *llist-of-inject* [*simp*]: $\text{llist-of } xs = \text{llist-of } ys \iff xs = ys$

proof

assume $\text{llist-of } xs = \text{llist-of } ys$

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*]: $\text{list-of } (\text{llist-of } xs) = xs$

by(*fastforce simp add: list-of-def intro: inv-f-f inj-onI*)

lemma *list-of-list-of* [*simp*]: $lfinite\ xs \implies list-of\ (list-of\ xs) = xs$
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*]: $lfinite\ xs \implies list-of\ (LCons\ x\ xs) = x \# list-of\ xs$
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 $\langle list-of\ (LCons\ x'\ xs') = x' \# list-of\ xs' \rangle$ **show** *?case*
using *list-of-llist-of*[*of* $x \# x' \# list-of\ xs'$]
llist-of-list-of[*OF* $\langle lfinite\ xs' \rangle$] **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 $\langle lfinite\ xs \rangle$ **by** *induct(simp-all add: \langle lfinite ys \rangle)*

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*]: $lfinite\ xs \implies hd\ (list-of\ xs) = lhd\ xs$
by(*clarsimp simp add: lfinite-eq-range-llist-of*)

lemma *tl-list-of*: $lfinite\ xs \implies tl\ (list-of\ xs) = list-of\ (ttl\ xs)$
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 \implies *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 \implies *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 \implies *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 \implies *lfinite* xs
proof(*induct* n arbitrary: xs)
case [*folded zero-enat-def*]: 0
thus ?*case* **by** *simp*
next

case (*Suc n*)
note $len = \langle llength\ xs = enat\ (Suc\ n) \rangle$
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 $\exists n. llength\ xs = enat\ n$
using *assms*
by *induct(auto simp add: eSuc-def zero-enat-def)*

lemma *lfinite-conv-llength-enat*:
 $lfinite\ xs \longleftrightarrow (\exists n. llength\ xs = enat\ n)$
by (*blast dest: llength-eq-enat-lfiniteD lfinite-llength-enat*)

lemma *not-lfinite-llength*:
 $\neg lfinite\ xs \implies llength\ xs = \infty$
by (*simp add: lfinite-conv-llength-enat*)

lemma *llength-eq-infty-conv-lfinite*:
 $llength\ xs = \infty \longleftrightarrow \neg lfinite\ xs$
by (*simp add: lfinite-conv-llength-enat*)

lemma *lfinite-finite-index*: $lfinite\ xs \implies finite\ \{n. enat\ n < llength\ xs\}$
by (*auto dest: lfinite-llength-enat*)

tail-recursive implementation for *llength*

definition *gen-llength* :: $nat \Rightarrow 'a\ list \Rightarrow enat$
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 $F \equiv \lambda length\ xs. case\ xs\ of\ LNil \Rightarrow 0 \mid LCons\ x\ xs \Rightarrow eSuc\ (llength\ xs)$
shows $llength-conv-fixp: llength \equiv cpo.fixp\ (fun-lub\ Sup)\ (fun-ord\ op \le)\ F$ (**is**
 $- \equiv ?fixp$)
and $llength-mono: \bigwedge xs. monotone\ (fun-ord\ op \le)\ op \le (\lambda length. F\ length\ xs)$
(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 *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-def split: llist.split del: iffI)
qed

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

lemma *mcont2mcont-llength[THEN lfp.mcont2mcont, simp, cont-intro]:*
shows *mcont-llength: mcont lSup op \sqsubseteq Sup op \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-LNil [simp, code, nitpick-simp]: ltake n LNil = LNil*
by *simp*

lemma *ltake-0 [simp]: ltake 0 xs = LNil*
by(*simp add: ltake-def*)

lemma *ltake-eSuc-LCons [simp]:*
ltake (eSuc n) (LCons x xs) = LCons x (ltake n xs)
by(*rule llist.expand*)(*simp-all*)

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

lemma *lnull-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-eq-LNil-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 \longleftrightarrow 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) =
(case n of 0 \Rightarrow LNil | eSuc n' \Rightarrow LCons x (ltake n' xs))

```

by(rule llist.expand)(simp-all split: co.enat.split)

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

lemma ltl-ltake:  $\text{ltl } (\text{ltake } n \text{ } xs) = \text{ltake } (\text{epred } n) (\text{ltl } xs)$ 
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:  $\text{ltake } n (\text{ltl } xs) = \text{ltl } (\text{ltake } (\text{eSuc } n) \text{ } xs)$ 
by(cases xs) simp-all

lemma llength-ltake [simp]:  $\text{llength } (\text{ltake } n \text{ } xs) = \min n (\text{llength } xs)$ 
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]:  $\text{ltake } n (\text{lmap } f \text{ } xs) = \text{lmap } f (\text{ltake } n \text{ } xs)$ 
by(coinduction arbitrary: n xs)(auto 4 3 simp add: ltl-ltake)

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

lemma lset-ltake:  $\text{lset } (\text{ltake } n \text{ } xs) \subseteq \text{lset } xs$ 
proof(rule subsetI)
  fix x
  assume  $x \in \text{lset } (\text{ltake } n \text{ } xs)$ 
  thus  $x \in \text{lset } xs$ 
  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:  $\text{llength } xs \leq m \implies \text{ltake } m \text{ } xs = xs$ 
by(coinduction arbitrary: m xs)(auto simp add: epred-llength[symmetric] ltl-ltake
intro: epred-le-epredI)

lemma ltake-llist-of [simp]:
   $\text{ltake } (\text{enat } n) (\text{llist-of } xs) = \text{llist-of } (\text{take } n \text{ } xs)$ 
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  $\equiv$  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 ]  $\implies$  ltake m xs = ltake m ys
by (metis ltake-ltake min-def)

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

lemma lprefix-ltake-same [simp]:
  ltake n xs  $\sqsubseteq$  ltake m xs  $\longleftrightarrow$   $n \leq m \vee \text{length } xs \leq m$ 
  (is ?lhs  $\longleftrightarrow$  ?rhs)
proof(rule iffI disjCI)+
  assume ?lhs  $\neg \text{length } xs \leq m$ 
  thus  $n \leq m$ 
  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  $n \leq m$ 
    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  $F \equiv \lambda \text{take } n \text{ xs. case } xs \text{ of } LNil \Rightarrow LNil \mid LCons \ x \ \text{xs} \Rightarrow \text{case } n \text{ of } 0 \Rightarrow$ 
 $LNil \mid eSuc \ n \Rightarrow LCons \ x \ (ltake \ n \ \text{xs})$ 
  shows ltake-conv-fix:  $ltake \equiv \text{curry } (ccpo.\text{fixp } (fun\text{-lub } lSup) (fun\text{-ord } op \sqsubseteq))$ 
 $(\lambda \text{take. case-prod } (F \ (\text{curry } ltake)))$  (is ?lhs  $\equiv$  ?rhs)
  and ltake-mono:  $\bigwedge nxs. \text{mono-llist } (\lambda \text{take. case } nxs \text{ of } (n, \text{xs}) \Rightarrow F \ (\text{curry } ltake)$ 
 $n \ \text{xs})$  (is PROP ?mono)
proof(intro eq-reflection ext)
  show mono: PROP ?mono unfolding F-def by(tactic  $\ll$  Partial-Function.mono-tac
 $@\{\text{context}\} \ 1 \ \gg$ )
  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:
 $F\text{-def split: llist.split prod.split co.enat.split}$ )
  qed

```

qed

lemma *monotone-ltake*: *monotone (rel-prod op ≤ op ⊆) op ⊆ (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 op ≤ op ⊆ (λn. ltake n xs)*
using *monotone-ltake* **by** *auto*

lemma *mono2mono-ltake2* [*THEN llist.mono2mono, cont-intro, simp*]:
 shows *monotone-ltake2: monotone op ⊆ op ⊆ (ltake n)*
using *monotone-ltake* **by** *auto*

lemma *mcont-ltake*: *mcont (prod-lub Sup lSup) (rel-prod op ≤ op ⊆) lSup op ⊆*
(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 op ≤ lSup op ⊆ (λn. ltake n xs)*
using *mcont-ltake* **by** *auto*

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

lemma [*partial-function-mono*]: *mono-llist F ⇒ mono-llist (λf. ltake n (F f))*
by(*rule mono2mono-ltake2*)

lemma *llist-induct2*:

assumes *adm: ccpo.admissible (prod-lub lSup lSup) (rel-prod op ⊆ op ⊆) (λx. P*
(fst x) (snd x))

and *LNil: P LNil LNil*

and *LCons1: ∧x xs. [lfinite xs; P xs LNil] ⇒ P (LCons x xs) LNil*

and *LCons2: ∧y ys. [lfinite ys; P LNil ys] ⇒ P LNil (LCons y ys)*

and *LCons: ∧x xs y ys. [lfinite xs; lfinite ys; P xs ys] ⇒ P (LCons x xs)*
(LCons y ys)

shows *P xs ys*

proof –

let *?C = (λn. (ltake n xs, ltake n ys)) ‘ range enat*

have *Complete-Partial-Order.chain (rel-prod op ⊆ op ⊆) ?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:  $lnull\ xs \implies ldropn\ n\ xs = LNil$ 
by(simp add: lnull-def)

```

```

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

```

```

lemma ldropn-Suc:  $ldropn\ (Suc\ n)\ xs = (case\ xs\ of\ LNil \implies LNil \mid LCons\ x\ xs' \implies ldropn\ n\ xs')$ 
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 $F \equiv \lambda dropn\ xs. case\ xs\ of\ LNil \Rightarrow \lambda -. LNil \mid LCons\ x\ xs \Rightarrow \lambda n. if\ n = 0\ then\ LCons\ x\ xs\ else\ ldropn\ xs\ (n - 1)$
 shows *ldrop-conv-fixp*: $(\lambda xs\ n. ldropn\ n\ xs) \equiv cppo.fixp\ (fun-lub\ (fun-lub\ lSup))\ (fun-ord\ (fun-ord\ lprefix))\ (\lambda ldrop. F\ ldrop)$ (**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* $\ll Partial-Function.mono-tac$ $@\{context\}\ 1 \gg$)
 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*:
 $(\lambda xs. case\ xs\ of\ LNil \Rightarrow \lambda -. LNil \mid LCons\ x\ xs \Rightarrow \lambda n. if\ n = 0\ then\ LCons\ x\ xs\ else\ f\ xs\ (n - 1)) =$
 $(\lambda xs\ n. case\ xs\ of\ LNil \Rightarrow LNil \mid LCons\ x\ xs \Rightarrow if\ n = 0\ then\ LCons\ x\ xs\ else\ f\ xs\ (n - 1))$
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*]:

[[$\bigwedge n. \text{monotone } op \leq \text{ord } (\lambda n. f n (eSuc n))$;
 $\bigwedge n. \text{ord } a (f n (eSuc n)); \text{ord } a a$]]

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

by(*rule monotone-enat-le-case*)

lemma *monotone-ldrop*: *monotone (rel-prod op = op \sqsubseteq) op \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 op \sqsubseteq op \sqsubseteq (ldrop n)*

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

lemma *mcont-ldrop*: *mcont (prod-lub the-Sup lSup) (rel-prod op = op \sqsubseteq) lSup op \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 op \sqsubseteq lSup op \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*: $(\text{ldropn } n \text{ } xs = \text{LNil}) = (\text{llength } xs \leq \text{enat } n)$
using *lnull-ldropn unfolding lnull-def* .

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

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

lemma *ltl-ldrop*: $\text{ltl } (\text{ldrop } n \text{ } xs) = \text{ldrop } n \text{ } (\text{ltl } xs)$
by(*simp add: ldrop-eSuc-conv-ltl ldrop-ltl*)

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

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

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

lemma *ldrop-enat* [*code*]: $\text{ldrop } (\text{enat } n) \text{ } xs = \text{ldropn } n \text{ } xs$
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*]: $\text{lfinite } (\text{ldropn } n \text{ } xs) = \text{lfinite } xs$
by(*induct n arbitrary: xs*)(*simp-all add: ldropn-Suc split: llist.split*)

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

lemma *ldropn-ltl*: $\text{ldropn } n \text{ } (\text{ltl } xs) = \text{ldropn } (\text{Suc } n) \text{ } xs$
by(*simp add: ldropn-def funpow-swap1*)

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

lemma *lset-ldropn-subset*: $\text{lset } (\text{ldropn } n \text{ } xs) \subseteq \text{lset } xs$
by(*induct n arbitrary: xs*)(*fastforce simp add: ldropn-Suc split: llist.split-asm*)+

lemma *in-lset-ldropnD*: $x \in \text{lset } (\text{ldropn } n \text{ } xs) \implies x \in \text{lset } xs$
using *lset-ldropn-subset[of n xs]* **by** *auto*

lemma *lset-ldrop-subset*: $\text{lset } (\text{ldrop } n \text{ } xs) \subseteq \text{lset } xs$
proof(*induct xs arbitrary: n*)

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

lemma *in-lset-ldropD*: $x \in \text{lset } (\text{ldrop } n \text{ } xs) \implies x \in \text{lset } xs$
using *lset-ldrop-subset*[*of n xs*] **by**(*auto*)

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

lemma *ldropn-lappend*:
 $\text{ldropn } n (\text{lappend } xs \text{ } ys) =$
(*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 $\text{llength } zs \leq \text{enat } n$
hence $\text{the-enat } (\text{eSuc } (\text{llength } zs)) = \text{Suc } (\text{the-enat } (\text{llength } zs))$
by(*auto intro!: the-enat-eSuc iff del: not-infinity-eq*) }
note $eq = \text{this}$
from *Suc* **show** *?case*
by(*cases xs*)(*auto simp add: not-less not-le eSuc-enat*[*symmetric*] *eq*)
qed

lemma *ldropn-lappend2*:
 $\text{llength } xs \leq \text{enat } n \implies \text{ldropn } n (\text{lappend } xs \text{ } ys) = \text{ldropn } (n - \text{the-enat } (\text{llength } xs)) \text{ } ys$
by(*auto simp add: ldropn-lappend*)

lemma *lappend-ltake-enat-ldropn* [*simp*]: $\text{lappend } (\text{ltake } (\text{enat } n) \text{ } xs) (\text{ldropn } n \text{ } xs) = xs$
by(*fold ldrop-enat*)(*rule lappend-ltake-ldrop*)

lemma *ldrop-lappend*:
 $\text{ldrop } n (\text{lappend } xs \text{ } ys) =$
(*if n < llength xs then lappend (ldrop n xs) ys*
else ldrop (n - llength xs) ys)
— cannot prove this directly using fixpoint induction, because $op -$ is not a least fixpoint
by(*cases n*)(*cases llength xs, simp-all add: ldropn-lappend not-less ldrop-enat*)

lemma *ltake-plus-conv-lappend*:
 $\text{ltake } (n + m) \text{ } xs = \text{lappend } (\text{ltake } n \text{ } xs) (\text{ltake } m (\text{ldrop } n \text{ } xs))$
by(*coinduction arbitrary: n m xs rule: llist.coinduct-strong*)(*auto intro!: exI simp*)

add: iadd-is-0 ltl-ltake epred-iadd1 ldrops-ldrop

lemma *ldrops-eq-LConsD*:

ldrops n xs = LCons y ys \implies enat n < llength xs

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: Suc-ile-eq*)

qed

lemma *ldrop-eq-LConsD*:

ldrop n xs = LCons y ys \implies n < llength xs

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

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

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

lemma *ldrop-lmap [simp]*: *ldrop n (lmap f xs) = lmap f (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 *ldrops-ldrops [simp]*:

ldrops n (ldrops m xs) = ldrops (n + m) xs

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

lemma *ldrop-ldrop [simp]*:

ldrop n (ldrop m xs) = ldrop (n + m) 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-ldrops [simp]*: *llength (ldrops n xs) = llength xs - enat n*

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]*)

qed

lemma *enat-llength-ldrops*:

enat n \leq llength xs \implies enat (n - m) \leq llength (ldrops m xs)

by(*cases llength xs*) *simp-all*

lemma *ldrops-llist-of [simp]*: *ldrops 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 = ∞ 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 \Rightarrow x | Suc n' \Rightarrow 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:
   $\neg \text{lnull } xs \implies \text{lhd } xs = \text{lnth } xs \ 0$ 
by(auto simp add: lhd-def not-lnull-conv)

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

lemma lnth-ltl:  $\neg \text{lnull } xs \implies \text{lnth } (\text{ltl } xs) \ n = \text{lnth } xs \ (\text{Suc } n)$ 
by(auto simp add: not-lnull-conv)

lemma lhd-ldropn:
   $\text{enat } n < \text{llength } xs \implies \text{lhd } (\text{ldropn } n \ xs) = \text{lnth } xs \ n$ 
proof(induct n arbitrary: xs)
  case 0 thus ?case by(cases xs) auto
next
  case (Suc n)
  from  $\langle \text{enat } (\text{Suc } n) < \text{llength } xs \rangle$  obtain x xs'
    where [simp]:  $xs = \text{LCons } x \ xs'$  by(cases xs) auto
  from  $\langle \text{enat } (\text{Suc } n) < \text{llength } xs \rangle$ 
  have  $\text{enat } n < \text{llength } xs'$  by(simp add: Suc-ile-eq)
  hence  $\text{lhd } (\text{ldropn } n \ xs') = \text{lnth } xs' \ n$  by(rule Suc)
  thus ?case by simp
qed

lemma lhd-ldrop:
  assumes  $n < \text{llength } xs$ 
  shows  $\text{lhd } (\text{ldrop } n \ xs) = \text{lnth } xs \ (\text{the-enat } n)$ 
proof –
  from assms obtain n' where  $n: n = \text{enat } n'$  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:
   $\text{llength } xs \leq \text{enat } n \implies \text{lnth } xs \ n = \text{undefined } (n - (\text{case } \text{llength } xs \ \text{of } \text{enat } m \Rightarrow m))$ 
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]:
   $enat\ n < \text{length}\ xs \implies \text{lnth}\ (lmap\ f\ xs)\ n = f\ (\text{lnth}\ xs\ n)$ 
proof(induct n arbitrary: xs)
  case 0 thus ?case by(cases xs) simp-all
next
  case (Suc n)
  from  $\langle enat\ (Suc\ n) < \text{length}\ xs \rangle$  obtain x xs'
    where xs: xs = LCons x xs' and len: enat n < length xs'
    by(cases xs)(auto simp add: Suc-ile-eq)
  from len have  $\text{lnth}\ (lmap\ f\ xs')\ n = f\ (\text{lnth}\ xs'\ n)$  by(rule Suc)
  with xs show ?case by simp
qed

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

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

lemma in-lset-conv-lnth:
   $x \in \text{lset}\ xs \iff (\exists n. enat\ n < \text{length}\ xs \wedge \text{lnth}\ xs\ n = x)$ 
  (is ?lhs  $\iff$  ?rhs)
proof
  assume ?rhs
  then obtain n where  $enat\ n < \text{length}\ xs$   $\text{lnth}\ xs\ n = x$  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:  $lset\ xs = \{lnth\ xs\ n \mid n.\ enat\ n < llength\ xs\}$ 
by(auto simp add: in-lset-conv-lnth)

```

```

lemma lnth-llist-of [simp]:  $lnth\ (l\text{list-of}\ xs) = nth\ xs$ 
proof(rule ext)
  fix n
  show  $lnth\ (l\text{list-of}\ xs)\ n = xs\ !\ n$ 
  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  $\langle enat\ (Suc\ n) < llength\ xs \rangle$  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:
   $lnth\ (lappend\ (l\text{list-of}\ xs)\ ys)\ n =$ 
  (if  $n < llength\ xs$  then  $xs\ !\ n$  else  $lnth\ ys\ (n - llength\ xs)$ )
proof(induct xs arbitrary: n)
  case (Cons x xs) thus ?case by(cases n) auto
qed simp

```

lemma *lnth-lappend2*:
 $\llbracket \text{llength } xs = \text{enat } k; k \leq n \rrbracket \implies \text{lnth } (\text{lappend } xs \text{ } ys) \ n = \text{lnth } ys \ (n - k)$
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*:
 $\text{lnth } (\text{lappend } xs \text{ } ys) \ n = (\text{if } \text{enat } n < \text{llength } xs \text{ then } \text{lnth } xs \ n \text{ else } \text{lnth } ys \ (n - \text{the-enat } (\text{llength } xs)))$
by(*cases llength xs*)(*auto simp add: lnth-lappend1 lnth-lappend2*)

lemma *lnth-ltake*:
 $\text{enat } m < n \implies \text{lnth } (\text{ltake } n \text{ } xs) \ m = \text{lnth } xs \ m$
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 $\langle \text{enat } (\text{Suc } m) < n \rangle$ **obtain** *n'* **where** $n = \text{eSuc } n'$
by(*cases n rule: enat-coexhaust*) *auto*
with $\langle \text{enat } (\text{Suc } m) < n \rangle$ **have** $\text{enat } m < n'$ **by**(*simp add: eSuc-enat[symmetric]*)
with *Suc* $\langle n = \text{eSuc } n' \rangle$ **show** ?*case* **by**(*cases xs*) *auto*
qed

lemma *ldropn-Suc-conv-ldropn*:
 $\text{enat } n < \text{llength } xs \implies \text{LCons } (\text{lnth } xs \ n) \ (\text{ldropn } (\text{Suc } n) \ xs) = \text{ldropn } n \ xs$
proof(*induct n arbitrary: xs*)
case 0 **thus** ?*case* **by**(*cases xs*) *auto*
next
case (*Suc n*)
from $\langle \text{enat } (\text{Suc } n) < \text{llength } xs \rangle$ **obtain** *x xs'*
where [*simp*]: $xs = \text{LCons } x \ xs'$ **by**(*cases xs*) *auto*
from $\langle \text{enat } (\text{Suc } n) < \text{llength } xs \rangle$
have $\text{enat } n < \text{llength } xs'$ **by**(*simp add: Suc-ile-eq*)
hence $\text{LCons } (\text{lnth } xs' \ n) \ (\text{ldropn } (\text{Suc } n) \ xs') = \text{ldropn } n \ xs'$ **by**(*rule Suc*)
thus ?*case* **by** *simp*
qed

lemma *ltake-Suc-conv-snoc-lnth*:
 $\text{enat } m < \text{llength } xs \implies \text{ltake } (\text{enat } (\text{Suc } m)) \ xs = \text{lappend } (\text{ltake } (\text{enat } m) \ xs) \ (\text{LCons } (\text{lnth } xs \ m) \ \text{LNil})$
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 ltl-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 $f :: 'a \Rightarrow 'a$
shows $lnull\text{-funpow-lmap}$: $lnull ((lmap f \hat{\hat{}} n) xs) \longleftrightarrow lnull xs$
and $lhd\text{-funpow-lmap}$: $\neg lnull xs \implies lhd ((lmap f \hat{\hat{}} n) xs) = (f \hat{\hat{}} n) (lhd xs)$
and $ltl\text{-funpow-lmap}$: $\neg lnull xs \implies ltl ((lmap f \hat{\hat{}} n) xs) = (lmap f \hat{\hat{}} n) (ltl xs)$
by(*induct n simp-all*)

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

{ **fix** x
def $n \equiv 0 :: nat$
have $(lmap f \hat{\hat{}} n) (h x) = (lmap f \hat{\hat{}} n) (iterates f x)$
by(*coinduction arbitrary: n*)(*auto simp add: funpow-swap1 lmap-iterates intro*):
exI[**where** $x = Suc\ n$ **for** n] }
thus *?thesis* **by** *auto*
qed

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

lemma *ldropn-iterates*: $ldropn\ n (iterates f x) = iterates f ((f \hat{\hat{}} n) x)$
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** $\dots = iterates f ((f \hat{\hat{}} n) (f x))$ **by**(*rule Suc*)
finally **show** *?case* **by**(*simp add: funpow-swap1*)
qed

lemma *ldrop-iterates*: $ldrop (enat\ n) (iterates f x) = iterates f ((f \hat{\hat{}} n) x)$
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]: $lnth (iterates f x) n = (f \hat{\hat{}} n) x$
proof(*induct n arbitrary: x*)
case 0 **thus** *?case* **by**(*subst iterates*) *simp*
next

case (*Suc n*)
hence $l\text{nth } (\text{iterates } f (f x)) n = (f \text{ } ^n) (f x)$.
thus *?case* **by**(*subst iterates*)(*simp add: funpow-swap1*)
qed

lemma *lset-iterates*:
 $l\text{set } (\text{iterates } f x) = \{(f \text{ } ^n) x \mid n. \text{True}\}$
by(*auto simp add: lset-conv-lnth*)

lemma *lset-repeat* [*simp*]: $l\text{set } (\text{repeat } x) = \{x\}$
by(*simp add: lset-iterates id-def[symmetric]*)

2.16 More on the prefix ordering on lazy lists: $op \sqsubseteq$ and *lstrict-prefix*

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

lemma *lmap-lprefix*: $xs \sqsubseteq ys \implies l\text{map } f xs \sqsubseteq l\text{map } f ys$
by(*rule monotoneD[OF monotone-lmap]*)

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

lemma *lprefix-llength-le*: $xs \sqsubseteq ys \implies \text{length } xs \leq \text{length } ys$
using *monotone-llength* **by**(*rule monotoneD*)

lemma *lstrict-prefix-llength-less*:
assumes *lstrict-prefix xs ys*
shows $\text{length } xs < \text{length } ys$
proof(*rule ccontr*)
assume $\neg \text{length } xs < \text{length } ys$
moreover from (*lstrict-prefix xs ys*) **have** $xs \sqsubseteq ys \wedge xs \neq ys$
unfolding *lstrict-prefix-def* **by** *simp-all*
from ($xs \sqsubseteq ys$) **have** $\text{length } xs \leq \text{length } ys$
by(*rule lprefix-llength-le*)
ultimately have $\text{length } xs = \text{length } ys$ **by** *auto*
with ($xs \sqsubseteq ys$) **have** $xs = ys$
by(*rule lprefix-llength-eq-imp-eq*)
with ($xs \neq ys$) **show** *False* **by** *contradiction*
qed

lemma *lstrict-prefix-lfinite1*: $l\text{strict-prefix } xs ys \implies l\text{finite } xs$
by (*metis lstrict-prefix-def not-lfinite-lprefix-conv-eq*)

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

lemma *llist-less-induct*[*case-names less*]:
 $(\bigwedge xs. (\bigwedge ys. \text{lstrict-prefix } ys \text{ } xs \implies P \text{ } ys) \implies P \text{ } xs) \implies P \text{ } xs$
by(*rule wfP-induct*[*OF wfP-lstrict-prefix*]) *blast*

lemma *ltake-enat-eq-imp-eq*: $(\bigwedge n. \text{ltake } (\text{enat } n) \text{ } xs = \text{ltake } (\text{enat } n) \text{ } ys) \implies xs = ys$
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** $x = \text{eSuc } n$ **for** n]

lemma *ltake-enat-lprefix-imp-lprefix*:
 assumes $\bigwedge n. \text{lprefix } (\text{ltake } (\text{enat } n) \text{ } xs) (\text{ltake } (\text{enat } n) \text{ } ys)$
 shows *lprefix* $xs \text{ } ys$
proof –
 have *ccpo.admissible* *Sup op* $\leq (\lambda n. \text{ltake } n \text{ } xs \sqsubseteq \text{ltake } n \text{ } ys)$ **by** *simp*
 hence *ltake* (*Sup* (*range* *enat*)) $xs \sqsubseteq \text{ltake } (\text{Sup } (\text{range } \text{enat})) \text{ } ys$
 by(*rule ccpo.admissibleD*)(*auto intro: assms*)
 thus *?thesis* **by**(*simp add: ltake-all*)
qed

lemma *lprefix-conv-lappend*: $xs \sqsubseteq ys \iff (\exists zs. ys = \text{lappend } xs \text{ } zs)$ (**is** *?lhs* \iff *?rhs*)
proof(*rule iffI*)
 assume *?lhs*
 hence $ys = \text{lappend } xs (\text{ldrop } (\text{length } xs) \text{ } ys)$
 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** $x = \text{LNil}$])
 thus *?rhs* **..**
next
 assume *?rhs*
 thus *?lhs* **by**(*coinduct rule: lprefix-coinduct*) *auto*
qed

lemma *lappend-lprefixE*:
 assumes *lappend* $xs \text{ } ys \sqsubseteq zs$
 obtains zs' **where** $zs = \text{lappend } xs \text{ } zs'$
using *assms* **unfolding** *lprefix-conv-lappend* **by**(*auto simp add: lappend-assoc*)

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

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

lemma *lstrict-prefix-lappend-conv*:
 $\text{lstrict-prefix } xs (\text{lappend } xs \ ys) \longleftrightarrow \text{lfinite } xs \wedge \neg \text{lnull } ys$
proof –
{ **assume** $\text{lfinite } xs \ xs = \text{lappend } xs \ ys$
hence $\text{lnull } ys$ **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*:
 $\exists zs. ys = xs @ zs \implies \text{llist-of } xs \sqsubseteq \text{llist-of } ys$
by(*clarsimp simp add: lappend-llist-of-llist-of[symmetric] lprefix-lappend*)

lemma *lprefix-llist-of* [*simp*]: $l\text{list-of } xs \sqsubseteq l\text{list-of } ys \longleftrightarrow \text{prefix } xs \ ys$
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*: $\bigwedge x \ xs. \llbracket l\text{finite } xs; P \ xs \rrbracket \implies P \ (LCons \ x \ xs)$
and *limit*: $(\bigwedge ys. l\text{strict-prefix } ys \ xs \implies P \ ys) \implies P \ xs$
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 $\implies l\text{strict-prefix } (lmap \ f \ xs) \ (lmap \ f \ ys)$
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 op* $\sqsubseteq A$
shows $l\text{finite } (lSup \ A) \longleftrightarrow \text{finite } A \wedge (\forall xs \in A. l\text{finite } xs)$ (**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 $l\text{length } 'A \subseteq \{.. < enat \ (Suc \ n)\}$
by (*auto dest!: chain-lprefix-lSup[OF chain] lprefix-llength-le simp add: n intro: le-less-trans*)
hence *finite* $(l\text{length } 'A)$ **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* $\forall xs \in A. l\text{finite } xs$ **by** *simp-all*
show *?lhs*
proof (*cases A = {}*)
case *False*
with *chain* $\langle \text{finite } A \rangle$
have $lSup \ A \in A$ **by** (*rule ccpo.in-chain-finite[OF llist-ccpo]*)

with $\langle \forall xs \in A. \text{ lfinite } xs \rangle$ **show** *?thesis ..*
qed *simp*
qed(*rule lfinite-lSupD*)

Setup for $op \sqsubseteq$ for Nitpick

definition *finite-lprefix* :: 'a llist \Rightarrow 'a llist \Rightarrow bool
where *finite-lprefix* = $op \sqsubseteq$

lemma *finite-lprefix-nitpick-simps* [*nitpick-simp*]:
finite-lprefix $xs \text{ LNil}$ $\longleftrightarrow xs = \text{LNil}$
finite-lprefix $\text{LNil } xs$ $\longleftrightarrow \text{True}$
finite-lprefix $xs (\text{LCons } y \text{ } ys)$ \longleftrightarrow
 $xs = \text{LNil} \vee (\exists xs'. xs = \text{LCons } y \text{ } xs' \wedge \text{ finite-lprefix } xs' \text{ } ys)$
by(*simp-all add: lprefix-LCons-conv finite-lprefix-def lnull-def*)

lemma *lprefix-nitpick-simps* [*nitpick-simp*]:
 $xs \sqsubseteq ys = (\text{if } \text{lfinite } xs \text{ then } \text{finite-lprefix } xs \text{ } ys \text{ else } xs = ys)$
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*: $\text{llcp } \text{LNil } ys = 0$
and *llcp-LNil2*: $\text{llcp } xs \text{ LNil} = 0$
and *llcp-LCons*: $\text{llcp } (\text{LCons } x \text{ } xs) (\text{LCons } y \text{ } ys) = (\text{if } x = y \text{ then } \text{eSuc } (\text{llcp } xs \text{ } ys) \text{ else } 0)$
by(*simp-all add: llcp-def enat-unfold split: llist.split*)

lemma *llcp-eq-0-iff*:
 $\text{llcp } xs \text{ } ys = 0 \longleftrightarrow \text{lnull } xs \vee \text{lnull } ys \vee \text{lhd } xs \neq \text{lhd } ys$
by(*simp add: llcp-def*)

lemma *epred-llcp*:
 $\llbracket \neg \text{lnull } xs; \neg \text{lnull } ys; \text{lhd } xs = \text{lhd } ys \rrbracket$
 $\implies \text{epred } (\text{llcp } xs \text{ } ys) = \text{llcp } (\text{ltl } xs) (\text{ltl } ys)$
by(*simp add: llcp-def*)

lemma *llcp-commute*: $\text{llcp } xs \text{ } ys = \text{llcp } ys \text{ } xs$
by(*coinduction arbitrary: xs ys rule: enat-coinduct*)(*auto simp add: llcp-eq-0-iff epred-llcp*)

lemma *llcp-same-conv-length* [*simp*]: $\text{llcp } xs \text{ } xs = \text{llength } xs$
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]: $xs \sqsubseteq ys \implies llcp\ xs\ ys = llength\ xs$
by (metis add-0-right lappend-LNil2 llcp-LNil1 llcp-lappend-same lprefix-conv-lappend)

lemma *llcp-lprefix2* [simp]: $ys \sqsubseteq xs \implies llcp\ xs\ ys = llength\ ys$
by (metis llcp-commute llcp-lprefix1)

lemma *llcp-le-length*: $llcp\ xs\ ys \leq \min (llength\ xs)\ (llength\ ys)$
proof –

def $m \equiv llcp\ xs\ ys$ **and** $n \equiv \min (llength\ xs)\ (llength\ ys)$
hence $(m, n) \in \{(llcp\ xs\ ys, \min (llength\ xs)\ (llength\ ys)) \mid xs\ ys :: 'a\ llist.\ True\}$
by blast
thus $m \leq n$
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 | LCons x xs' \Rightarrow case ys of*
LNil \Rightarrow LNil | 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 op \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 \ll Partial-Function.mono-tac*
@{context} 1 \gg)

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 op \sqsubseteq op \sqsubseteq) op \sqsubseteq (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 op* \sqsubseteq *op* \sqsubseteq ($\lambda xs. lzip\ xs\ ys$)
by(*simp add: monotone-lzip[simplified]*)

lemma *mono2mono-lzip2* [*THEN llist.mono2mono, cont-intro, simp*]:
shows *monotone-lzip2*: *monotone op* \sqsubseteq *op* \sqsubseteq ($\lambda ys. lzip\ xs\ ys$)
by(*simp add: monotone-lzip[simplified]*)

lemma *mcont-lzip*: *mcont (prod-lub lSup lSup) (rel-prod op* \sqsubseteq *op* \sqsubseteq) *lSup op* \sqsubseteq
(*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 op* \sqsubseteq *lSup op* \sqsubseteq ($\lambda xs. lzip\ xs\ ys$)
by(*simp add: mcont-lzip[simplified]*)

lemma *mcont2mcont-lzip2* [*THEN llist.mcont2mcont, cont-intro, simp*]:
shows *mcont-lzip2*: *mcont lSup op* \sqsubseteq *lSup op* \sqsubseteq ($\lambda ys. lzip\ xs\ ys$)
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 ($\lambda(x, y). (f\ x, g\ y)$) (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*:
 $\llbracket\ enat\ n < llength\ xs; enat\ n < llength\ ys\ \rrbracket$
 $\implies lnth\ (lzip\ xs\ ys)\ n = (lnth\ xs\ n, lnth\ ys\ n)$
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*:
 $lset (lzip\ xs\ ys) =$
 $\{(lnth\ xs\ n,\ lnth\ ys\ n) \mid n.\ enat\ n < \min (llength\ xs)\ (llength\ ys)\}$
by(*auto simp add: lset-conv-lnth lnth-lzip*)(*auto intro!: exI simp add: lnth-lzip*)

lemma *lset-lzipD1*: $(x, y) \in lset (lzip\ xs\ ys) \implies x \in lset\ xs$
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** $\text{--}(drule\ sym,\ auto\ simp\ add:\ lzip\ eq\ LCons\ conv)$
qed

lemma *lset-lzipD2*: $(x, y) \in lset (lzip\ xs\ ys) \implies y \in lset\ ys$
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** $\text{--}(drule\ sym,\ auto\ simp\ add:\ lzip\ eq\ LCons\ conv)$
qed

lemma *lset-lzip-same* [*simp*]: $lset (lzip\ xs\ xs) = (\lambda x.\ (x, x)) \text{ ' } lset\ xs$
by(*auto 4 3 simp add: lset-lzip in-lset-conv-lnth*)

lemma *lfinite-lzip* [*simp*]:
 $lfinite (lzip\ xs\ ys) \longleftrightarrow lfinite\ xs \vee lfinite\ ys$ (**is** ?*lhs* \longleftrightarrow ?*rhs*)
proof
assume ?*lhs*
thus ?*rhs*
by(*induct zs \equiv lzip xs ys arbitrary: xs ys rule: lfinite-induct*) *fastforce*+
next
assume ?*rhs* (**is** ?*xs* \vee ?*ys*)
thus ?*lhs*
proof
assume ?*xs*
thus ?*thesis*
proof(*induct arbitrary: ys*)
case (*lfinite-LConsI xs x*)
thus ?*case* **by**(*cases ys*) *simp-all*
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 $\exists xs' xs'' ys' ys''$. $xs = lappend xs' xs'' \wedge ys = lappend ys' ys'' \wedge$
 $llength xs' = llength ys' \wedge us = lzip xs' ys' \wedge$
 $vs = lzip xs'' ys''$

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*) \geq *llength us*
by(*auto simp add: enat-le-plus-same*)
hence *len*: *llength xs* \geq *llength us* *llength ys* \geq *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* \longrightarrow $vs = lzip ?xs'' ?ys''$ **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 \neg *lfinite* ?*xs'* \neg *lfinite* ?*ys'*

```

  by(auto simp del: llength-ltake lfinite-ltake
      simp add: ltake-all dest: lfinite-llength-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*]:
 $lzip (lmap f xs) (lmap g ys) = lmap (\lambda(x, y). (f x, g y)) (lzip xs ys)$
by(coinduction arbitrary: xs ys) auto

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

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

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

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

lemma *lzip-conv-lzip-ltake-min-llength*:
 $lzip xs ys =$
 $lzip (ltake (\min (\text{llength } xs) (\text{llength } ys)) xs)$
 $(ltake (\min (\text{llength } xs) (\text{llength } ys)) ys)$
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 $F \equiv \lambda \text{takeWhile } xs. \text{ case } xs \text{ of } LNil \Rightarrow LNil \mid LCons \ x \ xs \Rightarrow \text{if } P \ x \ \text{then } LCons \ x \ (\text{takeWhile } xs) \ \text{else } LNil$

shows *takeWhile-conv-fixp*: $\text{takeWhile } P \equiv \text{ccpo.fixp } (\text{fun-lub } lSup) (\text{fun-ord } lprefix) \ F$ (**is** *?lhs* \equiv *?rhs*)

and *takeWhile-mono*: $\bigwedge xs. \text{mono-llist } (\lambda \text{takeWhile}. F \ \text{takeWhile } xs)$ (**is** *PROP ?mono*)

proof(*intro eq-reflection ext*)

show *mono: PROP ?mono unfolding F-def* **by**(*tactic* \ll *Partial-Function.mono-tac* $\@ \{ \text{context} \} \ 1 \ \gg$)

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

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

shows *monotone-takeWhile*: $\text{monotone } lprefix \ lprefix \ (\text{takeWhile } P)$

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

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

shows *mcont-takeWhile*: $\text{mcont } lSup \ lprefix \ lSup \ lprefix \ (\text{takeWhile } P)$

by(*rule llist.fixp-preserves-mcont1* [*OF takeWhile-mono takeWhile-conv-fixp*]) *simp*

lemma *mono-llist-takeWhile* [*partial-function-mono*]:

$\text{mono-llist } F \Longrightarrow \text{mono-llist } (\lambda f. \text{takeWhile } P \ (F \ f))$

by(*rule mono2mono-takeWhile*)

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

shows *monotone-ldropWhile*: $\text{monotone } op \sqsubseteq op \sqsubseteq (\text{ldropWhile } P)$

by(*rule llist.fixp-preserves-mono1* [*OF ldropWhile.mono ldropWhile-def*]) *simp*

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

shows *mcont-ldropWhile*: $\text{mcont } lSup \ op \sqsubseteq lSup \ op \sqsubseteq (\text{ldropWhile } P)$

by(*rule llist.fixp-preserves-mcont1* [*OF ldropWhile.mono ldropWhile-def*]) *simp*

lemma *lnull-takeWhile* [*simp*]: $\text{lnull } (\text{takeWhile } P \ xs) \longleftrightarrow (\neg \text{lnull } xs \longrightarrow \neg P \ (\text{lhd } xs))$

by(*cases xs*) *simp-all*

lemma *takeWhile-eq-LNil-iff*: $\text{takeWhile } P \ xs = LNil \longleftrightarrow (xs \neq LNil \longrightarrow \neg P \ (\text{lhd } xs))$

using *lnull-takeWhile unfolding lnull-def* .

lemmas $lhd\text{-}ltakeWhile = ltakeWhile.sel(1)$

lemma $ltl\text{-}ltakeWhile$:

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

lemma $lprefix\text{-}ltakeWhile$: $ltakeWhile P xs \sqsubseteq xs$

by(coinduction arbitrary: xs)(auto simp add: ltl-ltakeWhile)

lemma $llength\text{-}ltakeWhile\text{-}le$: $llength (ltakeWhile P xs) \leq llength xs$

by(rule lprefix-llength-le)(rule lprefix-ltakeWhile)

lemma $ltakeWhile\text{-}nth$: $enat i < llength (ltakeWhile P xs) \implies lnth (ltakeWhile P xs) i = lnth xs i$

by(rule lprefix-lnthD[OF lprefix-ltakeWhile])

lemma $ltakeWhile\text{-}all$: $\forall x \in lset xs. P x \implies ltakeWhile P xs = xs$

by(coinduction arbitrary: xs)(auto 4 3 simp add: ltl-ltakeWhile simp del: ltakeWhile.disc-iff dest: in-lset-ltlD)

lemma $lset\text{-}ltakeWhileD$:

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 $lset\text{-}ltakeWhile\text{-}subset$:

$lset (ltakeWhile P xs) \subseteq lset xs \cap \{x. P x\}$

by(auto dest: lset-ltakeWhileD)

lemma $ltakeWhile\text{-}all\text{-}conv$: $ltakeWhile P xs = xs \iff lset xs \subseteq \{x. P x\}$

by (metis Int-Collect Int-absorb2 le-infE lset-ltakeWhile-subset ltakeWhile-all)

lemma $llength\text{-}ltakeWhile\text{-}all$: $llength (ltakeWhile P xs) = llength xs \iff ltakeWhile P xs = xs$

by(auto intro: lprefix-llength-eq-imp-eq lprefix-ltakeWhile)

lemma $ldropWhile\text{-}eq\text{-}LNil\text{-}iff$: $ldropWhile P xs = LNil \iff (\forall x \in lset xs. P x)$

by(induct xs) simp-all

lemma $lnull\text{-}ldropWhile$ [*simp*]:

$lnull (ldropWhile P xs) \iff (\forall x \in lset xs. P x) \text{ (is ?lhs } \iff \text{ ?rhs)}$

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

lemma $lset\text{-}ldropWhile\text{-}subset$:

$lset (ldropWhile P xs) \subseteq lset xs$

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 $P \text{ } xs$] **by** *auto*

lemma *ltakeWhile-lmap*: $\text{ltakeWhile } P \text{ } (\text{lmap } f \text{ } xs) = \text{lmap } f \text{ } (\text{ltakeWhile } (P \circ f) \text{ } xs)$
by(*coinduction arbitrary: xs*)(*auto simp add: ltl-ltakeWhile*)

lemma *ldropWhile-lmap*: $\text{ldropWhile } P \text{ } (\text{lmap } f \text{ } xs) = \text{lmap } f \text{ } (\text{ldropWhile } (P \circ f) \text{ } xs)$
by(*induct xs*) *simp-all*

lemma *llength-ltakeWhile-lt-iff*: $\text{llength } (\text{ltakeWhile } P \text{ } xs) < \text{llength } xs \iff (\exists x \in \text{lset } xs. \neg P \text{ } x)$
(is *?lhs* \iff *?rhs)*

proof

assume *?lhs*

hence $\text{ltakeWhile } P \text{ } xs \neq xs$ **by** *auto*

thus *?rhs* **by**(*auto simp add: ltakeWhile-all-conv*)

next

assume *?rhs*

hence $\text{ltakeWhile } P \text{ } xs \neq xs$ **by**(*auto simp add: ltakeWhile-all-conv*)

thus *?lhs* **unfolding** *llength-ltakeWhile-all*[*symmetric*]

using *llength-ltakeWhile-le*[of $P \text{ } xs$] **by**(*auto*)

qed

lemma *ltakeWhile-K-False* [*simp*]: $\text{ltakeWhile } (\lambda-. \text{False}) \text{ } xs = \text{LNil}$
by(*simp add: ltakeWhile-def*)

lemma *ltakeWhile-K-True* [*simp*]: $\text{ltakeWhile } (\lambda-. \text{True}) \text{ } xs = xs$
by(*coinduction arbitrary: xs*) *simp*

lemma *ldropWhile-K-False* [*simp*]: $\text{ldropWhile } (\lambda-. \text{False}) = \text{id}$

proof

fix *xs*

show $\text{ldropWhile } (\lambda-. \text{False}) \text{ } xs = \text{id } xs$

by(*induct xs*) *simp-all*

qed

lemma *ldropWhile-K-True* [*simp*]: $\text{ldropWhile } (\lambda-. \text{True}) \text{ } xs = \text{LNil}$
by(*induct xs*)(*simp-all*)

lemma *lappend-ltakeWhile-ldropWhile* [*simp*]:

$\text{lappend } (\text{ltakeWhile } P \text{ } xs) \text{ } (\text{ldropWhile } P \text{ } xs) = xs$

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

lemma *ltakeWhile-lappend*:

$\text{ltakeWhile } P \text{ } (\text{lappend } xs \text{ } ys) =$


```

    (if  $\exists x \in lset\ xs. \neg P\ x$  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
 $\exists\ simp\ add: not-lnull-conv$ )
  ultimately show  $?case\ ..$ 
qed

```

```

lemma ldropWhile-lappend:
   $ldropWhile\ P\ (lappend\ xs\ ys) =$ 
  (if  $\exists x \in lset\ xs. \neg P\ x$  then  $lappend\ (ldropWhile\ P\ xs)\ ys$ 
   else if  $lfinite\ xs$  then  $ldropWhile\ P\ ys$  else  $LNil$ )
proof(cases  $\exists x \in lset\ xs. \neg P\ x$ )
  case True
  then obtain  $x$  where  $x \in lset\ xs \neg P\ x$  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:
   $lfinite\ (ltakeWhile\ P\ xs) \longleftrightarrow lfinite\ xs \vee (\exists x \in lset\ xs. \neg P\ x)$  (is  $?lhs \longleftrightarrow ?rhs$ )
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-ltakeWhile-eq-infinity*:

llength (ltakeWhile P xs) = ∞ \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-ltakeWhile-eq-infinity'*:

llength (ltakeWhile P xs) = ∞ \longleftrightarrow \neg *lfinite xs* \wedge ($\forall x \in \text{lset } xs. P x$)

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*:

$\llbracket x \in \text{lset } xs; \neg P x \rrbracket \implies \text{ltakeWhile } P (\text{lappend } xs \text{ } ys) = \text{ltakeWhile } P xs$

by(*induct rule: lset-induct*) *simp-all*

lemma *ltakeWhile-lappend2*:

lset xs \subseteq {x. P x}

$\implies \text{ltakeWhile } P (\text{lappend } xs \text{ } ys) = \text{lappend } xs (\text{ltakeWhile } P ys)$

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: $\bigwedge x. x \in \text{lset } ys \implies P x = Q x$*

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 $\neg P (\text{lnth } xs (\text{the-enat } (\text{llength } (\text{ltakeWhile } P xs))))$

proof

assume *P: P (lnth xs (the-enat (llength (ltakeWhile P xs))))*

from *len* **obtain** *len* **where** *llength (ltakeWhile P xs) = 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* $\exists x \in \text{lset } xs. \neg P x$
shows *lhd-ldropWhile*: $\neg P (\text{lhd } (\text{ldropWhile } P xs))$ (**is** *?thesis1*)
and *lhd-ldropWhile-in-lset*: $\text{lhd } (\text{ldropWhile } P xs) \in \text{lset } xs$ (**is** *?thesis2*)
proof –
from *assms* **obtain** *x* **where** $x \in \text{lset } xs \neg P x$ **by** *blast*
thus *?thesis1 ?thesis2* **by** *induct simp-all*
qed

lemma *ldropWhile-eq-ldrop*:
 $\text{ldropWhile } P xs = \text{ldrop } (\text{llength } (\text{ltakeWhile } P xs)) xs$
(**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 ($\exists \text{ldrop } xs$)
thus *?case* **by**(*cases xs*) *auto*
qed *simp-all*
qed

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

lemma *ltakeWhile-repeat*:
 $\text{ltakeWhile } P (\text{repeat } x) = (\text{if } P x \text{ then } \text{repeat } x \text{ else } \text{LNil})$
by(*coinduction arbitrary: x*)(*auto simp add: ltl-ltakeWhile*)

lemma *ldropWhile-repeat*: $\text{ldropWhile } P (\text{repeat } x) = (\text{if } P x \text{ then } \text{LNil} \text{ else } \text{repeat } x)$
by(*simp add: ldropWhile-eq-ldrop ltakeWhile-repeat*)

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

lemma *llength-ldropWhile*:
 $\text{llength } (\text{ldropWhile } P xs) =$
(*if* $\exists x \in \text{lset } xs. \neg P x$ *then* $\text{llength } xs - \text{llength } (\text{ltakeWhile } P xs)$ *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*:
 $\exists x \in \text{lset } xs. \neg P x \implies \text{lhd } (\text{ldropWhile } P xs) = \text{lnth } xs$ (*the-enat* (llength

(*ltakeWhile P xs*))
by(*simp add: ldropWhile-eq-ldrop lhd-ldrop llength-ltakeWhile-lt-iff*)

2.20 *l*list-all2

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

lemma *l*list-all2-LNil1 [*simp*]: *l*list-all2 P LNil xs \longleftrightarrow xs = LNil
by(*cases xs simp-all*)

lemma *l*list-all2-LNil2 [*simp*]: *l*list-all2 P xs LNil \longleftrightarrow xs = LNil
by(*cases xs simp-all*)

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

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

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

lemma *l*list-all2-conv-lzip:
*l*list-all2 P xs ys \longleftrightarrow *l*length xs = *l*length ys \wedge ($\forall (x, y) \in$ *l*set (lzip xs ys). P x y)
by(*auto 4 4 elim!: GrpE simp add:*
*l*list-all2-def *l*map-fst-lzip-conv-ltake *l*map-snd-lzip-conv-ltake *l*take-all
intro!: GrpI relcomppI[of - xs - - ys])

lemma *l*list-all2-*l*lengthD:
*l*list-all2 P xs ys \implies *l*length xs = *l*length ys
by(*simp add: llist-all2-conv-lzip*)

lemma *l*list-all2-*l*nullD: *l*list-all2 P xs ys \implies *l*null xs \longleftrightarrow *l*null ys
unfolding *l*null-def **by** *auto*

lemma *l*list-all2-all-*l*nthI:
 \llbracket *l*length xs = *l*length ys;
 $\bigwedge n.$ *enat* n < *l*length xs \implies P (*l*nth xs n) (*l*nth ys n) \rrbracket
 \implies *l*list-all2 P xs ys
by(*auto simp add: llist-all2-conv-lzip lset-lzip*)

lemma *l*list-all2-lnthD:

$\llbracket \text{l$ list-all2 } P \text{ } xs \text{ } ys; \text{ enat } n < \text{llength } xs \rrbracket \implies P (\text{lnth } xs \text{ } n) (\text{lnth } ys \text{ } n)

by(*fastforce simp add: llist-all2-conv-lzip lset-lzip*)

lemma *l*list-all2-lnthD2:

$\llbracket \text{l$ list-all2 } P \text{ } xs \text{ } ys; \text{ enat } n < \text{llength } ys \rrbracket \implies P (\text{lnth } xs \text{ } n) (\text{lnth } ys \text{ } n)

by(*fastforce simp add: llist-all2-conv-lzip lset-lzip*)

lemma *l*list-all2-conv-all-lnth:

$\text{l$ list-all2 } P \text{ } xs \text{ } ys \longleftrightarrow
 $\text{l$ length } xs = \text{llength } ys \wedge

$(\forall n. \text{ enat } n < \text{l$ length } ys \longrightarrow P (\text{lnth } xs \text{ } n) (\text{lnth } ys \text{ } n))

by(*auto dest: llist-all2-llengthD llist-all2-lnthD2 intro: llist-all2-all-lnthI*)

lemma *l*list-all2-True [*simp*]: *l*list-all2 ($\lambda _ . \text{ True}$) *xs ys* \longleftrightarrow *l*length *xs* = *l*length *ys*

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

lemma *l*list-all2-reflI:

$(\bigwedge x. x \in \text{l$ set } xs \implies P \text{ } x \text{ } x) \implies \text{llist-all2 } P \text{ } xs \text{ } xs

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

lemma *l*list-all2-lmap1:

$\text{l$ list-all2 } P (\text{lmap } f \text{ } xs) \text{ } ys \longleftrightarrow \text{llist-all2 } (\lambda x. P (f \text{ } x)) \text{ } xs \text{ } ys

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

lemma *l*list-all2-lmap2:

$\text{l$ list-all2 } P \text{ } xs (\text{lmap } g \text{ } ys) \longleftrightarrow \text{llist-all2 } (\lambda x \text{ } y. P \text{ } x (g \text{ } y)) \text{ } xs \text{ } ys

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

lemma *l*list-all2-lfiniteD:

$\text{l$ list-all2 } P \text{ } xs \text{ } ys \implies \text{lfinite } xs \longleftrightarrow \text{lfinite } ys

by(*drule llist-all2-llengthD*)(*simp add: lfinite-conv-llength-enat*)

lemma *l*list-all2-coinduct[*consumes 1, case-names LNil LCons, case-conclusion LCons lhd ltl, coinduct pred*]:

assumes *major*: *X xs ys*

and *step*:

$\bigwedge xs \text{ } ys. X \text{ } xs \text{ } ys \implies \text{l$ null } xs \longleftrightarrow \text{lnull } ys

$\bigwedge xs \text{ } ys. \llbracket X \text{ } xs \text{ } ys; \neg \text{l$ null } xs; \neg \text{lnull } ys \rrbracket \implies P (\text{lhd } xs) (\text{lhd } ys) \wedge (X (\text{ltl } xs) (\text{ltl } ys) \vee \text{llist-all2 } P (\text{ltl } xs) (\text{ltl } ys))

shows *l*list-all2 *P xs ys*

proof(*rule llist-all2-all-lnthI*)

from *major* **show** *l*length *xs* = *l*length *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;  $\bigwedge x y. P x y \implies P' x y$  ]]  $\implies$  llist-all2 P' xs ys
by(auto simp add: llist-all2-conv-all-lnth)

```

lemma *llist-all2-left*: $llist-all2 (\lambda x -. P x) xs ys \longleftrightarrow llength xs = llength ys \wedge (\forall x \in lset xs. P x)$

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

lemma *llist-all2-right*: $llist-all2 (\lambda -. P) xs ys \longleftrightarrow llength xs = llength ys \wedge (\forall x \in lset ys. P x)$

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

lemma *llist-all2-lsetD1*: $[[llist-all2 P xs ys; x \in lset xs]] \implies \exists y \in lset ys. P x y$

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*: $[[llist-all2 P xs ys; y \in lset ys]] \implies \exists x \in lset xs. P x y$

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 ( $\lambda x y. P x y \wedge Q x y$ ) xs ys  $\longleftrightarrow$  llist-all2 P xs ys  $\wedge$  llist-all2 Q xs ys
by(auto simp add: llist-all2-conv-all-lnth)

```

lemma *llist-all2-lhdD*:

```

[[ llist-all2 P xs ys;  $\neg$  lnull xs ]]  $\implies$  P (lhd xs) (lhd ys)

```

by(*auto simp add: not-lnull-conv llist-all2-LCons1*)

lemma *lhist-all2-lhdD2*:

$\llbracket \text{llist-all2 } P \text{ } xs \text{ } ys; \neg \text{lnull } ys \rrbracket \implies P (\text{lhd } xs) (\text{lhd } ys)$
by(*auto simp add: not-lnull-conv llist-all2-LCons2*)

lemma *lhist-all2-ltII*:

$\text{llist-all2 } P \text{ } xs \text{ } ys \implies \text{llist-all2 } P (\text{ltl } xs) (\text{ltl } ys)$
by(*cases xs*)(*auto simp add: llist-all2-LCons1*)

lemma *lhist-all2-lappendI*:

assumes 1: *llist-all2 P xs ys*
and 2: $\llbracket \text{lfinite } xs; \text{lfinite } ys \rrbracket \implies \text{llist-all2 } P \text{ } xs' \text{ } ys'$
shows $\text{llist-all2 } P (\text{lappend } xs \text{ } xs') (\text{lappend } ys \text{ } ys')$
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-ltII simp add: lappend-eq-LNil-iff*)
 next
 case *False*
 with 1 **have** $\neg \text{lfinite } ys$ **by**(*auto dest: llist-all2-lfiniteD*)
 with *False 1* **show** ?*thesis* **by**(*simp add: lappend-inf*)
qed

lemma *lhist-all2-lappend1D*:

assumes $\text{llist-all2 } P (\text{lappend } xs \text{ } xs') \text{ } ys$
shows $\text{llist-all2 } P \text{ } xs (\text{ltake } (\text{llength } xs) \text{ } ys)$
and $\text{lfinite } xs \implies \text{llist-all2 } P \text{ } xs' (\text{ldrop } (\text{llength } xs) \text{ } ys)$
proof –
 from *assms* **have** $\text{len: llength } xs + \text{llength } xs' = \text{llength } ys$ **by**(*auto dest: llist-all2-llengthD*)
 hence $\text{len-xS: llength } xs \leq \text{llength } ys$ **and** $\text{len-xS': llength } xs' \leq \text{llength } ys$
 by (*metis enat-le-plus-same llength-lappend*)+

 show $\text{llist-all2 } P \text{ } xs (\text{ltake } (\text{llength } xs) \text{ } ys)$
 proof(*rule llist-all2-all-lnthI*)
 show $\text{llength } xs = \text{llength } (\text{ltake } (\text{llength } xs) \text{ } ys)$
 using *len-xS* **by**(*simp add: min-def*)
 next
 fix *n*
 assume *n: enat n < llength xs*
 also **have** $\dots \leq \text{llength } (\text{lappend } xs \text{ } xs')$ **by**(*simp add: enat-le-plus-same*)
 finally **have** $P (\text{lnth } (\text{lappend } xs \text{ } xs') \text{ } n) (\text{lnth } ys \text{ } n)$
 using *assms* **by** –(*rule llist-all2-lnthD*)
 also **from** *n* **have** $\text{lnth } ys \text{ } n = \text{lnth } (\text{ltake } (\text{llength } xs) \text{ } ys) \text{ } n$ **by**(*rule lnth-ltake[symmetric]*)
 also **from** *n* **have** $\text{lnth } (\text{lappend } xs \text{ } xs') \text{ } n = \text{lnth } xs \text{ } n$ **by**(*simp add: lnth-lappend1*)
 finally **show** $P (\text{lnth } xs \text{ } n) (\text{lnth } (\text{ltake } (\text{llength } xs) \text{ } ys) \text{ } n)$.

qed

assume $lfinite\ xs$
thus $l\text{-list-all2}\ P\ xs'\ (l\text{-drop}\ (l\text{-length}\ xs)\ ys)$ **using** $assms$
by($induct\ arbitrary:\ ys$)($auto\ simp\ add:\ l\text{-list-all2-LCons1}$)

qed

lemma $l\text{-map-eq-lmap-conv-l\text{-list-all2}$:

$l\text{-map}\ f\ xs = l\text{-map}\ g\ ys \longleftrightarrow l\text{-list-all2}\ (\lambda x\ y.\ f\ x = g\ y)\ xs\ ys$ (**is** $?lhs \longleftrightarrow ?rhs$)

proof

assume $?lhs$

thus $?rhs$

by($coinduction\ arbitrary:\ xs\ ys$)($auto\ simp\ add:\ neq-LNil-conv\ l\text{-null-def}\ LNil-eq-l\text{-map}\ l\text{-map-eq-LNil}$)

next

assume $?rhs$

thus $?lhs$

by($coinduction\ arbitrary:\ xs\ ys$)($auto\ dest:\ l\text{-list-all2-lnullD}\ l\text{-list-all2-lhdD}\ l\text{-list-all2-ltlI}$)

qed

lemma $l\text{-list-all2-expand}$:

$l\text{-null}\ xs \longleftrightarrow l\text{-null}\ ys;$

$l\text{-list-all2}\ P\ xs\ ys \implies P\ (l\text{-hd}\ xs)\ (l\text{-hd}\ ys) \wedge l\text{-list-all2}\ P\ (l\text{-tl}\ xs)\ (l\text{-tl}\ ys)$

$\implies l\text{-list-all2}\ P\ xs\ ys$

by($cases\ xs$)($auto\ simp\ add:\ not-l\text{-null-conv}$)

lemma $l\text{-list-all2-l\text{-length-ltake}\ WhileD$:

assumes $major:\ l\text{-list-all2}\ P\ xs\ ys$

and $Q:\ \bigwedge x\ y.\ P\ x\ y \implies Q1\ x \longleftrightarrow Q2\ y$

shows $l\text{-length}\ (l\text{-take}\ While\ Q1\ xs) = l\text{-length}\ (l\text{-take}\ While\ Q2\ ys)$

using $major$

by($coinduction\ arbitrary:\ xs\ ys\ rule:\ enat-coinduct$)($auto\ 4\ 3\ simp\ add:\ not-l\text{-null-conv}\ l\text{-list-all2-LCons1}\ l\text{-list-all2-LCons2}\ dest!:\ Q$)

lemma $l\text{-list-all2-lzipI}$:

$l\text{-list-all2}\ P\ xs\ ys;\ l\text{-list-all2}\ P'\ xs'\ ys'$

$\implies l\text{-list-all2}\ (rel\text{-prod}\ P\ P')\ (l\text{-zip}\ xs\ xs')\ (l\text{-zip}\ ys\ ys')$

by($coinduction\ arbitrary:\ xs\ xs'\ ys\ ys'$)($auto\ 6\ 6\ dest:\ l\text{-list-all2-lhdD}\ l\text{-list-all2-lnullD}\ intro:\ l\text{-list-all2-ltlI}$)

lemma $l\text{-list-all2-ltakeI}$:

$l\text{-list-all2}\ P\ xs\ ys \implies l\text{-list-all2}\ P\ (l\text{-take}\ n\ xs)\ (l\text{-take}\ n\ ys)$

by($auto\ simp\ add:\ l\text{-list-all2-conv-all-lnth}\ l\text{-nth-ltake}$)

lemma $l\text{-list-all2-l\text{-dropn}I$:

$l\text{-list-all2}\ P\ xs\ ys \implies l\text{-list-all2}\ P\ (l\text{-dropn}\ n\ xs)\ (l\text{-dropn}\ n\ ys)$

by($cases\ l\text{-length}\ ys$)($auto\ simp\ add:\ l\text{-list-all2-conv-all-lnth}$)

lemma $l\text{-list-all2-l\text{-drop}I$:

$l\text{list-all2 } P \text{ } xs \text{ } ys \implies l\text{list-all2 } P \text{ } (l\text{drop } n \text{ } xs) \text{ } (l\text{drop } n \text{ } ys)$
by(cases llength ys)(auto simp add: llist-all2-conv-all-lnth llength-ldrop)

lemma *l\text{list-all2-lSupI}*:

assumes *Complete-Partial-Order.chain* (rel-prod op \sqsubseteq op \sqsubseteq) $Y \forall (xs, ys) \in Y$.

l\text{list-all2 } P \text{ } xs \text{ } ys

shows *l\text{list-all2 } P \text{ } (l\text{Sup } (fst \text{ ' } Y)) \text{ } (l\text{Sup } (snd \text{ ' } 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** $Y: \bigwedge xs \text{ } ys. (xs, ys) \in Y \implies l\text{list-all2 } P \text{ } xs \text{ } ys$ **by** *blast*

from *LCons* **obtain** $xs \text{ } ys$ **where** $xs \text{ } ys \in Y$

and [*simp*]: $\neg \text{lnull } xs \neg \text{lnull } ys$

by(auto 4 3 dest: llist-all2-lnullD simp add: split-beta)

from $xs \text{ } ys \in Y$ **have** $\text{lhd } xs \in \text{lhd } \text{' } (fst \text{ ' } Y \cap \{xs. \neg \text{lnull } xs\})$

by(auto intro: rev-image-eqI)

hence (*THE* $x. x \in \text{lhd } \text{' } (fst \text{ ' } Y \cap \{xs. \neg \text{lnull } xs\})$) = $\text{lhd } xs$

by(rule the-equality)(auto dest!: lprefix-lhdD chainD[*OF chain* $xs \text{ } ys$])

moreover from $xs \text{ } ys \in Y$ **have** $\text{lhd } ys \in \text{lhd } \text{' } (snd \text{ ' } Y \cap \{xs. \neg \text{lnull } xs\})$

by(auto intro: rev-image-eqI)

hence (*THE* $x. x \in \text{lhd } \text{' } (snd \text{ ' } Y \cap \{xs. \neg \text{lnull } xs\})$) = $\text{lhd } ys$

by(rule the-equality)(auto dest!: lprefix-lhdD chainD[*OF chain* $xs \text{ } ys$])

moreover from $xs \text{ } ys \in Y$ **have** *l\text{list-all2 } P \text{ } xs \text{ } ys* **by**(rule Y)

hence $P \text{ } (\text{lhd } xs) \text{ } (\text{lhd } ys)$ **by**(rule *l\text{list-all2-lhdD}*) *simp*

ultimately have *?lhd* **using** *LCons* **by** *simp*

moreover {

let $?Y = \text{map-prod } \text{ltl } \text{ltl } \text{' } (Y \cap \{(xs, ys). \neg \text{lnull } xs \wedge \neg \text{lnull } ys\})$

have *Complete-Partial-Order.chain* (rel-prod op \sqsubseteq op \sqsubseteq) $?Y$

by(rule *chainI*)(auto 4 3 dest: Y chainD[*OF chain*] intro: lprefix-ltII)

moreover

have $\text{ltl } \text{' } (fst \text{ ' } Y \cap \{xs. \neg \text{lnull } xs\}) = \text{fst } \text{' } ?Y$

and $\text{ltl } \text{' } (snd \text{ ' } Y \cap \{xs. \neg \text{lnull } xs\}) = \text{snd } \text{' } ?Y$

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-ltII dest: Y) }

ultimately show *?case ..*

qed

lemma *admissible-llist-all2* [*cont-intro*, *simp*]:

assumes $f: \text{mcont } \text{lub } \text{ord } \text{lSup } \text{op } \sqsubseteq (\lambda x. f \text{ } x)$

and $g: \text{mcont } \text{lub } \text{ord } \text{lSup } \text{op } \sqsubseteq (\lambda x. g \text{ } x)$

shows *ccpo.admissible lub ord* ($\lambda x. l\text{list-all2 } P \text{ } (f \text{ } x) \text{ } (g \text{ } x)$)

proof(rule *ccpo.admissibleI*)

fix Y

assume *chain*: *Complete-Partial-Order.chain* $\text{ord } Y$

and $Y: \forall x \in Y. \text{llist-all2 } P (f x) (g x)$
and $Y \neq \{\}$
from *chain* **have** *Complete-Partial-Order.chain* (*rel-prod* $op \sqsubseteq op \sqsubseteq$) ($(\lambda x. (f x, g x)) \text{ ' } Y$)
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 $\text{llist-all2 } P (f (\text{lub } Y)) (g (\text{lub } Y))$ **using** $\langle Y \neq \{\} \rangle$
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 $*$: $\text{llist-all2 } P \text{ } xs \text{ } ys$
and $Q: \bigwedge x y. P x y \implies Q1 x \longleftrightarrow Q2 y$
shows $\text{llist-all2 } P (\text{ldropWhile } Q1 \text{ } xs) (\text{ldropWhile } Q2 \text{ } ys)$
— cannot prove this with parallel induction over xs and ys because $\lambda x. \neg \text{llist-all2 } P (f x) (g x)$ 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*]: $\text{llist-all2 } P \text{ } xs \text{ } xs \longleftrightarrow (\forall x \in \text{lset } xs. P x x)$
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*]]
 $\implies \text{llist-all2 } P \text{ } xs \text{ } zs$
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*: $\text{llast } LNil = \text{undefined}$
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-linfinite*: $\neg\ lfinite\ xs \implies\ llast\ xs =\ undefined$
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 $\neg\ lnull\ ys \implies\ llast (lappend\ xs\ ys) = llast\ ys$
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-linfinite)
qed

lemma *llast-lappend-LCons* [simp]:
 $lfinite\ xs \implies\ llast (lappend\ xs\ (LCons\ y\ ys)) = llast (LCons\ y\ ys)$
by(simp add: llast-lappend)

lemma *llast-ldropn*: $enat\ n < llength\ xs \implies\ llast (ldropn\ n\ xs) = llast\ xs$
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]: $llast (l\list-of\ xs) = last\ xs$

by(*induct xs*)(*auto simp add: last-def zero-enat-def llast-LCons llast-LNil*)

lemma *llast-conv-lnth*: $l\text{length } xs = eSuc \ (enat \ n) \implies llast \ xs = lnth \ xs \ n$
by(*clarsimp simp add: last-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 ttl, coinduct pred: ldistinct*]:
assumes $X \ xs$
and *step*: $\bigwedge xs. \llbracket X \ xs; \neg \ lnull \ xs \rrbracket$
 $\implies lhd \ xs \notin lset \ (ttl \ xs) \wedge (X \ (ttl \ xs) \vee ldistinct \ (ttl \ xs))$
shows *ldistinct xs*
using $\langle X \ xs \rangle$
proof(*coinduct*)
case (*ldistinct xs*)
thus *?case* **by**(*cases xs*)(*auto dest: step*)
qed

lemma *ldistinct-lhdD*:
 $\llbracket ldistinct \ xs; \neg \ lnull \ xs \rrbracket \implies lhd \ xs \notin lset \ (ttl \ xs)$
by(*clarsimp simp add: not-lnull-conv*)

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

lemma *ldistinct-lSup*:
 $\llbracket Complete-Partial-Order.chain \ op \sqsubseteq \ Y; \forall xs \in Y. ldistinct \ xs \rrbracket$

$\implies \text{ldistinct } (\text{lSup } Y)$
proof(*coinduction arbitrary: Y*)
case (*ldistinct Y*)
hence *chain: Complete-Partial-Order.chain op \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 op \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:*
 $\text{ldistinct } (\text{lappend } xs \ ys) \iff \text{ldistinct } xs \wedge (\text{lfinite } xs \longrightarrow \text{ldistinct } ys \wedge \text{lset } xs \cap \text{lset } ys = \{\})$
(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(*clarsimp simp add: lprefix-conv-lappend ldistinct-lappend*)

lemma *admissible-not-ldistinct[THEN admissible-subst, cont-intro, simp]:*
ccpo.admissible lSup op \sqsubseteq ($\lambda x. \neg \text{ldistinct } x$)

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 ($\forall i j. \text{enat } i < \text{llength } xs \longrightarrow \text{enat } j < \text{llength } xs \longrightarrow i \neq j \longrightarrow$
lnth xs } i \neq \text{lnth xs } j)
(is ?lhs \longleftrightarrow ?rhs)
proof(*intro iffI strip*)
 assume *?rhs*
 thus *?lhs*
 proof(*coinduct xs*)
 case (*ldistinct xs*)
 from ($\neg \text{lnull } xs$)
 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 (*i \neq j*) **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!: exI[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*: $\llbracket x \leq y; \textit{lsorted} (\textit{LCons } y \textit{ } xs) \rrbracket \Longrightarrow \textit{lsorted} (\textit{LCons } x (\textit{LCons } y \textit{ } xs))$

inductive-simps *lsorted-LCons-LCons* [*simp*]:

lsorted (*LCons* *x* (*LCons* *y* *xs*))

inductive-simps *lsorted-code* [*code*]:

lsorted *LNil*
lsorted (*LCons* *x* *LNil*)
lsorted (*LCons* *x* (*LCons* *y* *xs*))

lemma *lsorted-coinduct'* [*consumes* 1, *case-names* *lsorted*, *case-conclusion* *lsorted* *lhd* *ltl*, *coinduct* *pred*: *lsorted*]:

assumes *major*: $X \textit{ } xs$
and *step*: $\bigwedge xs. \llbracket X \textit{ } xs; \neg \textit{lnull } xs; \neg \textit{lnull} (\textit{ltl } xs) \rrbracket \Longrightarrow \textit{lhd } xs \leq \textit{lhd} (\textit{ltl } xs) \wedge (X (\textit{ltl } xs) \vee \textit{lsorted} (\textit{ltl } xs))$

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* \Longrightarrow *lsorted* (*ltl* *xs*)

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

lemma *lsorted-lhdD*:

$\llbracket \textit{lsorted } xs; \neg \textit{lnull } xs; \neg \textit{lnull} (\textit{ltl } xs) \rrbracket \Longrightarrow \textit{lhd } xs \leq \textit{lhd} (\textit{ltl } xs)$
by(*auto* *elim*: *lsorted.cases*)

lemma *lsorted-LCons'*:

lsorted (*LCons* *x* *xs*) $\longleftrightarrow (\neg \textit{lnull } xs \longrightarrow x \leq \textit{lhd } xs \wedge \textit{lsorted } xs)$
by(*cases* *xs*) *auto*

lemma *lsorted-lSup*:

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

proof(*coinduction* *arbitrary*: *Y*)

case (*lsorted* *Y*)

hence *sorted*: $\bigwedge xs. xs \in Y \Longrightarrow \textit{lsorted } xs$ **by** *blast*

note *chain* = $\langle \textit{Complete-Partial-Order.chain } op \sqsubseteq Y \rangle$

from $\langle \neg \textit{lnull} (\textit{lSup } Y) \rangle \langle \neg \textit{lnull} (\textit{ltl} (\textit{lSup } Y)) \rangle$

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

hence $\textit{lhd} (\textit{lSup } Y) = \textit{lhd } xs \textit{lhd} (\textit{ltl} (\textit{lSup } Y)) = \textit{lhd} (\textit{ltl } xs) \textit{lhd } xs \leq \textit{lhd} (\textit{ltl } xs)$

using *chain* *sorted* **by**(*auto* *intro*: *lhd-lSup-eq* *chain-lprefix-ltl* *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*:
 $\llbracket xs \sqsubseteq ys; lsorted\ ys \rrbracket \implies lsorted\ xs$
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 $lhd\ ys \leq lhd\ (ltl\ ys)$ **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 op* $\sqsubseteq (\lambda x. f\ x)$
and *ccpo: class.ccpo lub ord (mk-less ord)*
shows *ccpo.admissible lub ord* $(\lambda x. lsorted\ (f\ x))$
proof(*rule ccpo.admissibleI*)
fix *Y*
assume *chain: Complete-Partial-Order.chain ord Y*
and *sorted: $\forall x \in Y. lsorted\ (f\ x)$*
and $Y \neq \{\}$
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 op $\sqsubseteq (\lambda xs. \neg lsorted\ xs)$
by(*rule ccpo.admissibleI*)(*auto dest: lsorted-lprefixD[rotated] intro: chain-lprefix-lSup*)

lemma *lsorted-ltake* [*simp*]: *lsorted xs* $\implies lsorted\ (ltake\ n\ xs)$
by(*rule lsorted-lprefixD*)(*rule ltake-is-lprefix*)

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

lemma *lsorted-ldrop* [*simp*]: *lsorted xs* $\implies lsorted\ (ldrop\ n\ xs)$
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 ($\forall y \in \text{lset } xs. x \leq y$) (is *?lhs* \longleftrightarrow *?rhs*)

proof

assume *?lhs*

{ **fix** *y*

assume $y \in \text{lset } xs$

hence $x \leq y$ **using** $\langle ?lhs \rangle$

by(*induct arbitrary: x*)(*auto intro: order-trans*) }

with $\langle ?lhs \rangle$ **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: $\bigwedge xs. \llbracket X xs; \neg \text{lnull } xs \rrbracket \implies (\forall x \in \text{lset } (\text{ltl } xs). \text{lhd } xs \leq x) \wedge (X (\text{ltl } xs) \vee \text{lsorted } (\text{ltl } xs))$*

shows *lsorted xs*

using *major* **by**(*coinduct rule: lsorted-coinduct'*)(*auto dest: step*)

lemma *lsortedD*: $\llbracket \text{lsorted } xs; \neg \text{lnull } xs; y \in \text{lset } (\text{ltl } xs) \rrbracket \implies \text{lhd } xs \leq y$
by(*clarsimp 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 $\langle \text{ord.lsorted } orda \ xs \rangle$

by(*coinduction arbitrary: xs rule: ord.lsorted-coinduct'*)(*auto intro: monotoneD[OF $\langle \text{monotone } orda \ ordb \ f \rangle$ ord.lsorted-lhdD ord.lsorted-ltII*)

lemma *lsorted-lmap*:

assumes *lsorted xs monotone op \leq op \leq f*

shows *lsorted (lmap f xs)*

using $\langle \text{lsorted } xs \rangle$

by(*coinduction arbitrary: xs rule: lsorted-coinduct'*)(*auto intro: monotoneD[OF $\langle \text{monotone } op \leq op \leq f \rangle$ lsorted-lhdD lsorted-ltII*)

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-ltII ldistinct-ltII 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 sorted-Cons*)

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: $\bigwedge xs ys. \llbracket X \ xs \ ys; \neg \text{lnull } xs \rrbracket$
 $\implies \neg \text{lnull } ys \wedge$
 $(\neg \text{lnull } ys \longrightarrow r \ (\text{lhd } xs) \ (\text{lhd } ys) \vee$
 $\text{lhd } xs = \text{lhd } ys \wedge (X \ (\text{ltl } xs) \ (\text{ltl } ys) \vee \text{llexord } r \ (\text{ltl } xs) \ (\text{ltl } ys)))$
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 –
def *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*]:
 $lnull\ ys \implies llexord\ r\ xs\ ys \iff lnull\ xs$
by(*auto elim: llexord.cases*)

lemma *llexord-LCons-left*:
 $llexord\ r\ (LCons\ x\ xs)\ ys \iff$
 $(\exists\ y\ ys'.\ ys = LCons\ y\ ys' \wedge (x = y \wedge llexord\ r\ xs\ ys' \vee r\ x\ y))$
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*:
 $\llbracket lfinite\ xs; \forall x.\ x \in lset\ xs \implies \neg r\ x\ x \rrbracket$
 $\implies llexord\ r\ (lappend\ xs\ ys)\ (lappend\ xs\ zs) \iff llexord\ r\ ys\ zs$
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:
   $\llbracket llexord\ r\ xs\ ys; llexord\ r\ ys\ xs; \!$ 
     $\llbracket r\ a\ b; r\ b\ a \rrbracket \implies False \rrbracket$ 
   $\implies xs = ys$ 
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:  $\llbracket r\ a\ b; r\ b\ c \rrbracket \implies r\ a\ c$ 
  shows llexord r xs zs
proof –
  from 1 2 have  $\exists ys. llexord\ r\ xs\ ys \wedge llexord\ r\ ys\ zs$  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:  $\llbracket x\ y. r\ x\ y \vee x = y \vee r\ y\ x$ 
  shows llexord r xs ys  $\vee$  llexord r ys xs
proof(rule disjCI)
  assume  $\neg llexord\ r\ ys\ xs$ 
  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  $\langle \neg llexord\ r\ ys\ xs \rangle$  obtain y ys'

```

```

    where ys: ys = LCons y ys'  $\neg$  r y x y  $\neq$  x  $\vee$   $\neg$  llexord r ys' xs'
    by(cases ys) auto
    with ( $\neg$  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  $\vee$  x = y  $\wedge$  llexord r xs ys)
by auto

lemma llexord-conv:
  llexord r xs ys  $\longleftrightarrow$ 
  xs = ys  $\vee$ 
  ( $\exists$  zs xs' y ys'. lfinite zs  $\wedge$  xs = lappend zs xs'  $\wedge$  ys = lappend zs (LCons y ys')  $\wedge$ 
    (xs' = LNil  $\vee$  r (lhd xs') y))
  (is ?lhs  $\longleftrightarrow$  ?rhs)
proof
  assume ?lhs
  show ?rhs (is -  $\vee$  ?prefix)
  proof(rule disjCI)
    assume  $\neg$  ?prefix
    with (?lhs) show xs = ys
  proof(coinduction arbitrary: xs ys)
    case (Eq-llist xs ys)
    hence llexord r xs ys
    and prefix:  $\bigwedge$ zs xs' y ys'.  $\llbracket$  lfinite zs; xs = lappend zs xs';
      ys = lappend zs (LCons y ys')  $\rrbracket$ 
       $\implies$  xs'  $\neq$  LNil  $\wedge$   $\neg$  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''  $\neq$  LNil  $\wedge$   $\neg$  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  $\exists zs\ xs'\ y\ ys'.\ lfinite\ zs \wedge xs = lappend\ zs\ xs' \wedge$ 
             $ys = lappend\ zs\ (LCons\ y\ ys') \wedge$ 
             $(xs' = LNil \vee r\ (lhd\ xs')\ y)$ 
          then obtain zs xs' y' ys''
            where lfinite zs xs = lappend zs xs'
            and ys = lappend zs (LCons y' ys'')
            and xs' = LNil  $\vee$  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
  qed

```

lemma *llexord-conv-ltake-index*:

$$\begin{aligned}
 & llexord\ r\ xs\ ys \iff \\
 & (llength\ xs \leq llength\ ys \wedge ltake\ (llength\ xs)\ ys = xs) \vee \\
 & (\exists n. enat\ n < \min\ (llength\ xs)\ (llength\ ys) \wedge \\
 & \quad ltake\ (enat\ n)\ xs = ltake\ (enat\ n)\ ys \wedge r\ (lnth\ xs\ n)\ (lnth\ ys\ n))
 \end{aligned}$$

```

(is ?lhs  $\longleftrightarrow$  ?rhs)
proof(rule iffI)
  assume ?lhs
  thus ?rhs (is ?A  $\vee$  ?B) unfolding llexord-conv
proof
  assume xs = ys thus ?thesis by(simp add: ltake-all)
next
  assume  $\exists$  zs xs' y ys'. lfinite zs  $\wedge$  xs = lappend zs xs'  $\wedge$ 
    ys = lappend zs (LCons y ys')  $\wedge$ 
    (xs' = LNil  $\vee$  r (lhd xs') y)
  then obtain zs xs' y ys'
    where lfinite zs xs' = LNil  $\vee$  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  $\vee$  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  $\vee$  ?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 takers: 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 takers len
have  $ys = lappend (ltake (enat\ n)\ xs) (LCons (lnth\ ys\ n) (ldrop (enat\ (Suc\ n))\ ys))$ 
  by(simp add: ldropsn-Suc-conv-ldropsn 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*:

```

 $llexord\ r\ (llist-of\ xs)\ (llist-of\ ys) \longleftrightarrow$ 
 $xs = ys \vee (xs, ys) \in llexord\ \{(x, y). r\ x\ y\}$ 
(is ?lhs  $\longleftrightarrow$  ?rhs)

```

proof

```

assume ?lhs
{ fix  $zs\ xs'\ y\ ys'$ 
  assume lfinite zs  $llist-of\ xs = lappend\ zs\ xs'$ 
  and  $llist-of\ ys = lappend\ zs\ (LCons\ y\ ys')$ 
  and  $xs' = LNil \vee r\ (lhd\ xs')\ y$ 
from  $\langle lfinite\ zs \rangle$  obtain  $zs'$  where [simp]:  $zs = llist-of\ zs'$ 
  unfolding lfinite-eq-range-llist-of by blast
have lfinite (llist-of xs) by simp
hence lfinite xs' unfolding  $\langle llist-of\ xs = lappend\ zs\ xs' \rangle$  by simp
then obtain  $XS'$  where [simp]:  $xs' = llist-of\ XS'$ 
  unfolding lfinite-eq-range-llist-of by blast
from  $\langle llist-of\ xs = lappend\ zs\ xs' \rangle$  have [simp]:  $xs = zs' @ XS'$ 
  by(simp add: lappend-llist-of-llist-of)
have lfinite (llist-of ys) by simp
hence lfinite ys' unfolding  $\langle llist-of\ ys = lappend\ zs\ (LCons\ y\ ys') \rangle$  by simp
then obtain  $YS'$  where [simp]:  $ys' = llist-of\ YS'$ 
  unfolding lfinite-eq-range-llist-of by blast
from  $\langle llist-of\ ys = lappend\ zs\ (LCons\ y\ ys') \rangle$  have [simp]:  $ys = zs' @ y \# YS'$ 
  by(auto simp add: llist-of.simps(2)[symmetric] lappend-llist-of-llist-of simp
del: llist-of.simps(2))
have  $(\exists a\ ys'. ys = xs @ a \# ys') \vee$ 
   $(\exists zs\ a\ b. r\ a\ b \wedge (\exists xs'. xs = zs @ a \# xs') \wedge (\exists ys'. ys = zs @ b \# ys'))$ 
(is ?A  $\vee$  ?B)
proof(cases xs')
  case LNil thus ?thesis by(auto simp add: llist-of-eq-LNil-conv)
next
  case  $(LCons\ x\ xs'')$ 
  with  $\langle xs' = LNil \vee r\ (lhd\ xs')\ y \rangle$ 
  have  $r\ (lhd\ xs')\ y$  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 op ⊆ op ⊆ (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 op ⊆ lSup op ⊆ (lfilter P)
by(rule llist.fixp-preserves-mcont1 [OF lfilter.mono lfilter-def]) simp

```

lemma *lfilter-mono* [*partial-function-mono*]:

```

mono-llist A ⇒ mono-llist (λf. lfilter P (A f))
by(rule mono2mono-lfilter)

```

lemma *lfilter-LCons-seek*: $\sim (p\ x) \implies lfilter\ p\ (LCons\ x\ l) = lfilter\ p\ l$

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 $lfilter\ P\ xs \sqsubseteq LNil$

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*]: $lhd\ (lfilter\ P\ xs) = lhd\ (ldropWhile\ (Not \circ P)\ xs)$

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*: $ltl\ (lfilter\ P\ xs) = lfilter\ P\ (ltl\ (ldropWhile\ (Not \circ P)\ xs))$
by(*induct xs*) *simp-all*

lemma *lfilter-eq-LCons*:

$lfilter\ P\ xs = LCons\ x\ xs' \implies$

$\exists xs''. xs' = lfilter\ P\ xs'' \wedge ldropWhile\ (Not \circ P)\ xs = LCons\ x\ xs''$

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*]: $lfilter\ (\lambda-. False)\ xs = LNil$

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*]: $lset\ (lfilter\ P\ xs) = \{x \in lset\ xs.\ P\ x\}$

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 $lset\ (lfilter\ P\ xs) = lset\ xs \cap \{x.\ P\ x\}$ (**is** *?lhs = ?rhs*)

proof

show *?lhs* \subseteq *?rhs*

by(*induct arbitrary: xs rule: lfilter.fixp-induct*)(*auto split: llist.split*)

show *?rhs* \subseteq *?lhs*

proof

fix *x*

assume $x \in ?rhs$

hence $x \in lset\ xs\ P\ x$ **by** *simp-all*

thus $x \in ?lhs$ **by** *induction simp-all*

qed

qed

end

lemma *lfilter-lfilter*: $lfilter\ P\ (lfilter\ Q\ xs) = lfilter\ (\lambda x.\ P\ x \wedge Q\ x)\ xs$

by(*induction xs*) *simp-all*

notepad begin — show *lfilter-lfilter* by fixpoint induction

fix *P Q xs*

have $\forall xs.\ lfilter\ P\ (lfilter\ Q\ xs) \sqsubseteq lfilter\ (\lambda x.\ P\ x \wedge Q\ x)\ xs$

by(*rule lfilter.fixp-induct*)(*auto split: llist.split*)

moreover have $\forall xs.\ lfilter\ (\lambda x.\ P\ x \wedge Q\ x)\ xs \sqsubseteq lfilter\ P\ (lfilter\ Q\ xs)$

by(*rule lfilter.fixp-induct*)(*auto split: llist.split*)

ultimately have $lfilter\ P\ (lfilter\ Q\ xs) = lfilter\ (\lambda x.\ P\ x \wedge Q\ x)\ xs$

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*]:

$lfilter\ P\ (l\text{list-of}\ xs) = l\text{list-of}\ (filter\ P\ xs)$
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 $\langle lfinite\ (lfilter\ P\ xs) \rangle$
have $finite\ \{n. enat\ n < llength\ xs \wedge P\ (lnth\ xs\ n)\}$
proof(*induct ys \equiv 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 $\langle \neg lnull\ (lfilter\ P\ xs) \rangle$
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 $finite\ ((\lambda m. n + 1 + m) \text{ ` } (\{n. enat\ n < llength\ ?xs\} \cap \{n. P\ (lnth\ ?xs\ n)\}))$
(is $finite\ (?f \text{ ` } -)$)
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  $l\text{nth } ?xs' m \in l\text{set } ?xs'$  using  $n$ 
    unfolding  $in\text{-lset-conv-lnth}$  by auto
    hence  $\neg P (l\text{nth } ?xs' m)$  by (auto dest:  $l\text{set-ltakeWhileD}$ ) }
    hence  $\{n. \text{enat } n < l\text{length } xs \wedge P (l\text{nth } xs n)\} \subseteq \text{insert } (the\text{-enat } (l\text{length } ?xs')) ( ?f ' ?A)$ 
    using  $n$  by (subst (1 2)  $xs$ ) (cases  $l\text{length } ?xs$ , auto simp add:  $l\text{nth-lappend not-less not-le } l\text{nth-LCons}' e\text{Suc-enat split: if-split-asm intro!: rev-image-eqI}$ )
    ultimately show  $?case$  by (auto intro:  $finite\text{-subset}$ )
  qed }
  thus  $l\text{finite } xs \vee \text{finite } \{n. \text{enat } n < l\text{length } xs \wedge P (l\text{nth } xs n)\}$  by blast
next
assume  $l\text{finite } xs \vee \text{finite } \{n. \text{enat } n < l\text{length } xs \wedge P (l\text{nth } xs n)\}$ 
moreover {
  assume  $l\text{finite } xs$ 
  with  $l\text{length-lfilter-ile}[of P xs]$  have  $l\text{finite } (l\text{filter } P xs)$ 
  by (auto simp add:  $l\text{finite-eq-range-llist-of}$ )
} moreover {
  assume  $n\text{fin}: \neg l\text{finite } xs$ 
  hence  $len: l\text{length } xs = \infty$  by (rule  $not\text{-}l\text{finite-llength}$ )
  assume  $fin: \text{finite } \{n. \text{enat } n < l\text{length } xs \wedge P (l\text{nth } xs n)\}$ 
  then obtain  $m$  where  $\{n. \text{enat } n < l\text{length } xs \wedge P (l\text{nth } xs n)\} \subseteq \{..<m\}$ 
  unfolding  $finite\text{-nat-iff-bounded}$  by auto
  hence  $\bigwedge n. \llbracket m \leq n; \text{enat } n < l\text{length } xs \rrbracket \implies \neg P (l\text{nth } xs n)$  by auto
  hence  $l\text{finite } (l\text{filter } P xs)$ 
  proof (induct  $m$  arbitrary:  $xs$ )
    case 0
    hence  $l\text{null } (l\text{filter } P xs)$  by (auto simp add:  $in\text{-lset-conv-lnth}$ )
    thus  $?case$  by simp
  next
  case (Suc  $m$ )
  { fix  $n$ 
    assume  $m \leq n$   $\text{enat } n < l\text{length } (l\text{tl } xs)$ 
    hence  $\text{Suc } m \leq \text{Suc } n$   $\text{enat } (\text{Suc } n) < l\text{length } xs$ 
    by (case-tac [!]  $xs$ ) (auto simp add:  $\text{Suc-ile-eq}$ )
    hence  $\neg P (l\text{nth } xs (\text{Suc } n))$  by (rule  $\text{Suc}$ )
    hence  $\neg P (l\text{nth } (l\text{tl } xs) n)$ 
    using  $\langle \text{enat } n < l\text{length } (l\text{tl } xs) \rangle$  by (cases  $xs$ ) (simp-all) }
    hence  $l\text{finite } (l\text{filter } P (l\text{tl } xs))$  by (rule  $\text{Suc}$ )
    thus  $?case$  by (cases  $xs$ ) simp-all
  }
  qed }
  ultimately show  $l\text{finite } (l\text{filter } P xs)$  by blast
qed

```

lemma $l\text{filter-eq-LConsD}$:

assumes $l\text{filter } P ys = L\text{Cons } x xs$

shows $\exists us \text{ vs. } ys = l\text{append } us (L\text{Cons } x us) \wedge l\text{finite } us \wedge$
 $(\forall u \in l\text{set } us. \neg P u) \wedge P x \wedge xs = l\text{filter } P vs$

proof –

let $?us = l\text{takeWhile } (Not \circ P) ys$

and $?vs = \text{ltl } (\text{ldropWhile } (\text{Not } \circ P) \text{ } ys)$
from $assms$ **have** $\neg \text{lnull } (\text{lfilter } P \text{ } ys)$ **by** $auto$
hence $exP: \exists x \in \text{lset } ys. P \text{ } x$ **by** $simp$
from $eq-LConsD[OF \text{ } assms]$
have $x: x = \text{lhs } (\text{ldropWhile } (\text{Not } \circ P) \text{ } ys)$
and $xs: xs = \text{ltl } (\text{lfilter } P \text{ } ys)$ **by** $auto$
from exP
have $ys = \text{lappend } ?us \text{ } (LCons \text{ } x \text{ } ?vs)$ **unfolding** x **by** $simp$
moreover **have** $\text{lfinite } ?us$ **using** exP **by** $(simp \text{ add: } \text{lfinite-ltakeWhile})$
moreover **have** $\forall u \in \text{lset } ?us. \neg P \text{ } u$ **by** $(auto \text{ dest: } \text{lset-ltakeWhileD})$
moreover **have** $P \text{ } x$ **unfolding** x $o\text{-def}$
using exP **by** $(rule \text{ lhs-ldropWhile}[\text{where } P = \text{Not } \circ P, \text{ simplified}])$
moreover **have** $xs = \text{lfilter } P \text{ } ?vs$ **unfolding** xs **by** $(simp \text{ add: } \text{ltl-lfilter})$
ultimately show $?thesis$ **by** $blast$
qed

lemma $\text{lfilter-eq-lappend-lfiniteD}$:

assumes $\text{lfilter } P \text{ } xs = \text{lappend } ys \text{ } zs$ **and** $\text{lfinite } ys$
shows $\exists us \text{ } vs. xs = \text{lappend } us \text{ } vs \wedge \text{lfinite } us \wedge$
 $ys = \text{lfilter } P \text{ } us \wedge zs = \text{lfilter } P \text{ } vs$
using $\langle \text{lfinite } ys \rangle \langle \text{lfilter } P \text{ } xs = \text{lappend } ys \text{ } zs \rangle$
proof $(induct \text{ arbitrary: } xs \text{ } zs)$
case lfinite-LNil
hence $xs = \text{lappend } LNil \text{ } xs \text{ } LNil = \text{lfilter } P \text{ } LNil \text{ } zs = \text{lfilter } P \text{ } xs$
by $simp\text{-all}$
thus $?case$ **by** $blast$

next

case $(\text{lfinite-LConsI } ys \text{ } y)$
note $IH = \langle \bigwedge xs \text{ } zs. \text{lfilter } P \text{ } xs = \text{lappend } ys \text{ } zs \implies$
 $\exists us \text{ } vs. xs = \text{lappend } us \text{ } vs \wedge \text{lfinite } us \wedge$
 $ys = \text{lfilter } P \text{ } us \wedge zs = \text{lfilter } P \text{ } vs \rangle$
from $\langle \text{lfilter } P \text{ } xs = \text{lappend } (LCons \text{ } y \text{ } ys) \text{ } zs \rangle$
have $\text{lfilter } P \text{ } xs = LCons \text{ } y \text{ } (\text{lappend } ys \text{ } zs)$ **by** $simp$
from $\text{lfilter-eq-LConsD}[OF \text{ } this]$ **obtain** $us \text{ } vs$
where $xs = \text{lappend } us \text{ } (LCons \text{ } y \text{ } vs)$ $\text{lfinite } us$
 $P \text{ } y \forall u \in \text{lset } us. \neg P \text{ } u$
and $vs: \text{lfilter } P \text{ } vs = \text{lappend } ys \text{ } zs$ **by** $auto$
from $IH[OF \text{ } vs]$ **obtain** $us' \text{ } vs'$ **where** $vs = \text{lappend } us' \text{ } vs' \text{ } \text{lfinite } us'$
and $ys = \text{lfilter } P \text{ } us' \text{ } zs = \text{lfilter } P \text{ } vs'$ **by** $blast$
with xs **show** $?case$
by $(fastforce \text{ simp add: } \text{lappend-snocL1-conv-LCons2}[\text{symmetric}, \text{ where } ys = \text{lappend}$
 $us' \text{ } vs']$
 $\text{lappend-assoc}[\text{symmetric}])$

qed

lemma $\text{ldistinct-lfilterI}$: $\text{ldistinct } xs \implies \text{ldistinct } (\text{lfilter } P \text{ } xs)$

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

notepad begin

```

fix P xs
assume *: ldistinct xs
from * have ldistinct (lfilter P xs)  $\wedge$  lset (lfilter P xs)  $\subseteq$  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:

\llbracket ldistinct (lfilter P xs); enat n < llength xs; enat m < llength xs; P a; lnth xs n = a; lnth xs m = a $\rrbracket \implies m = n$

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 \langle Suc n \leq m \rangle **obtain** m' **where** m [simp]: m = Suc m' **by**(cases m) simp

with \langle Suc n \leq m \rangle **have** n \leq m' **by** simp

moreover from \langle enat (Suc n) < llength xs \rangle

obtain x xs' **where** xs [simp]: xs = LCons x xs' **by**(cases xs) simp

from \langle ldistinct (lfilter P xs) \rangle **have** ldistinct (lfilter P xs') **by**(simp split: if-split-asm)

moreover from \langle enat (Suc n) < llength xs \rangle \langle enat m < llength xs \rangle

have enat n < llength xs' enat m' < llength xs' **by**(simp-all add: Suc-ile-eq)

moreover note \langle P a \rangle

moreover have lnth xs' n = a lnth xs' m' = a

using \langle lnth xs (Suc n) = a \rangle \langle lnth xs m = a \rangle **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 *lfilter-all2-lfilterI*:

assumes *: *lfilter-all2 P xs ys*

and *Q*: $\bigwedge x y. P x y \implies Q1 x \longleftrightarrow Q2 y$

shows *lfilter-all2 P (lfilter Q1 xs) (lfilter Q2 ys)*

using * **by**(*induction arbitrary: xs ys rule: lfilter-fixp-parallel-induct*)(*auto split: lfilter.split dest: Q*)

lemma *distinct-filterD*:

$\llbracket \text{distinct } (lfilter P xs); n < length\ xs; m < length\ xs; P\ x; xs\ !\ n = x; xs\ !\ m = x \rrbracket \implies m = n$

using *ldistinct-lfilterD[of P lfilter of xs n m x]* **by** *simp*

lemma *lprefix-lfilterI*:

$xs \sqsubseteq ys \implies lfilter\ P\ xs \sqsubseteq lfilter\ P\ ys$

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*: $lfilter\ P\ xs = xs \longleftrightarrow (\forall x \in lset\ xs. P\ x)$ (**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 op* \sqsubseteq *op* \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 op* \sqsubseteq *lSup op* \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*]:
 $\llbracket \neg \text{lnull } xss; \neg \text{lnull } (\text{lhd } xss) \rrbracket \implies \text{lhd } (\text{lconcat } xss) = \text{lhd } (\text{lhd } xss)$
by(*clarsimp simp add: not-lnull-conv*)

lemma *ltl-lconcat* [*simp*]:
 $\llbracket \neg \text{lnull } xss; \neg \text{lnull } (\text{lhd } xss) \rrbracket \implies \text{ltl } (\text{lconcat } xss) = \text{lappend } (\text{ltl } (\text{lhd } xss))$
(*lconcat (ltl xss)*)
by(*clarsimp 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
 $(\exists x s' x s s''. x s s = \text{lappend } (\text{list-of } x s s') (\text{LCons } (\text{LCons } x x s') x s s'') \wedge$
 $x s = \text{lappend } x s' (\text{lconcat } x s s'') \wedge \text{set } x s s' \subseteq \{x s. \text{lnull } x s\})$
(**is** *?lhs* \longleftrightarrow *?rhs*)

proof
assume *?rhs*
then obtain *x s' x s s' x s s''*
where *x s s = lappend (list-of x s s') (LCons (LCons x x s') x s s'')*
and *x s = lappend x s' (lconcat x s s'')*

and $set\ xss' \subseteq \{xs.\ lnull\ xs\}$ **by** *blast*
moreover from $\langle set\ xss' \subseteq \{xs.\ lnull\ xs\} \rangle$
have $lnull\ (lconcat\ (l\text{-}list\text{-}of\ xss'))$ **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 $lset\ us \subseteq \{xs.\ lnull\ xs\} \neg\ lnull\ y$
and $ys:\ ys = lfilter\ (\lambda xs.\ \neg\ lnull\ xs)\ vs$ **by** *blast*
from $\langle lfinite\ us \rangle$ **obtain** us' **where** $[simp]: us = llist\text{-}of\ us'$
unfolding *lfinite-eq-range-llist-of* **by** *blast*
from $\langle lset\ us \subseteq \{xs.\ lnull\ xs\} \rangle$ **have** $us:\ lnull\ (lconcat\ us)$
unfolding *lnull-lconcat* .
from $\langle \neg\ lnull\ y \rangle$ **obtain** $y'\ ys'$ **where** $y:\ y = LCons\ y'\ ys'$
unfolding *not-lnull-conv* **by** *blast*
from $\langle ?lhs \rangle us$ **have** $[simp]: y' = x\ xs = lappend\ ys'\ (lconcat\ vs)$
unfolding $xss\ y$ **by** $(simp\ \text{-}all\ add:\ lappend\ lnull1)$
from $\langle lset\ us \subseteq \{xs.\ lnull\ xs\} \rangle ys$ **show** $?rhs$ **unfolding** $xss\ y$ **by** *simp blast*
qed

lemma *llength-lconcat-lfinite-conv-sum*:
assumes *lfinite xss*
shows $llength\ (lconcat\ xss) = (\sum\ i \mid\ enat\ i < llength\ xss.\ llength\ (l\text{-}nth\ xss\ i))$
using *assms*
proof $(induct)$
case *lfinite-LNil* **thus** $?case$ **by** *simp*
next
case $(lfinite\ LConsI\ xss\ xs)$
have $\{i.\ enat\ i \leq llength\ xss\} = insert\ 0\ \{Suc\ i \mid i.\ enat\ i < llength\ xss\}$
by $(auto\ simp\ add:\ zero\ \text{-}enat\ \text{-}def[symmetric]\ Suc\ \text{-}ile\ \text{-}eq\ gr0\ \text{-}conv\ \text{-}Suc)$
also have $\dots = insert\ 0\ (Suc\ \text{'}\ \{i.\ enat\ i < llength\ xss\})$ **by** *auto*
also have $0 \notin Suc\ \text{'}\ \{i.\ enat\ i < llength\ xss\}$ **by** *auto*
moreover from $\langle lfinite\ xss \rangle$ **have** *finite* $\{i.\ enat\ i < llength\ xss\}$
by $(rule\ lfinite\ \text{-}finite\ \text{-}index)$
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 $(induct\ xss)(simp\ \text{-}all\ add:\ lappend\ lnull1)$

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 *lalist-all2-lconcatI*:

lalist-all2 (lalist-all2 A) xss yss
 \implies *lalist-all2 A (lconcat xss) (lconcat yss)*

by(*induct arbitrary: xss yss rule: lconcat-fixp-parallel-induct*)(*auto split: lalist.split*
intro: lalist-all2-lappendI)

lemma *lalist-length-lconcat-eqI*:

fixes *xss :: 'a llist llist and yss :: 'b llist llist*
assumes *lalist-all2 ($\lambda xs ys. \text{length } xs = \text{length } ys$) xss yss*
shows *lalist-length (lconcat xss) = lalist-length (lconcat yss)*
apply(*rule lalist-all2-lengthD*[**where** *P= $\lambda - . \text{True}$*])
apply(*rule lalist-all2-lconcatI*)
apply(*simp add: lalist-all2-True*[*abs-def*] *assms*)
done

lemma *lset-lconcat-lfinite*:

$\forall xs \in \text{lset } xss. \text{lfinite } xs \implies \text{lset } (\text{lconcat } xss) = (\bigcup xs \in \text{lset } xss. \text{lset } xs)$
by(*induction xss*) *auto*

lemma *lconcat-ltake*:

lconcat (ltake (enat n) xss) = ltake ($\sum i < n. \text{length } (\text{lth } xss \ i)$) (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 ($\sum i < n. \text{length } (\text{lth } xss' \ i)$) (lconcat xss')* **by**(*rule Suc*)
also have *lappend xs ... = ltake (length xs + ($\sum i < n. \text{length } (\text{lth } xss' \ i)$)) (lappend xs (lconcat xss'))*
by(*cases length xs*)(*simp-all add: ltake-plus-conv-lappend ltake-lappend1 ltake-all ldropn-lappend2 lappend-inf lfinite-conv-length-enat ldrop-enat*)
also have ($\sum i < n. \text{length } (\text{lth } xss' \ i)$) = ($\sum i = 1 .. < \text{Suc } n. \text{length } (\text{lth } xss \ i)$)
by (*rule sum.reindex-cong* [*symmetric, of Suc*])
(auto simp add: LCons image-iff less-Suc-eq-0-disj)
also have *length xs + ... = ($\sum i < \text{Suc } n. \text{length } (\text{lth } xss \ i)$)*

unfolding *atLeast0LessThan[symmetric] LCons*
by (*subst (2) sum-head-upt-Suc*) *simp-all*
finally show *?thesis using LCons by simp*
qed
qed

lemma *lnth-lconcat-conv*:
assumes *enat n < llength (lconcat xss)*
shows $\exists m n'. \text{lnth } (lconcat \ xss) \ n = \text{lnth } (\text{lnth } \ xss \ m) \ n' \wedge \text{enat } n' < \text{llength } (\text{lnth } \ xss \ m) \wedge$
 $\text{enat } m < \text{llength } \ xss \wedge \text{enat } n = (\sum_{i < m} . \text{llength } (\text{lnth } \ xss \ i)) + \text{enat } n'$
using *assms*
proof (*induct n arbitrary: xss*)
case *0*
hence $\neg \text{lnull } (lconcat \ xss)$ **by** *auto*
then obtain *x xs* **where** *concat-xss: lconcat xss = LCons x xs*
unfolding *not-lnull-conv* **by** *blast*
then obtain *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' \subseteq {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 $(\sum_{i < \text{length } xss'} . \text{llength } (\text{lnth } \ xss \ i)) = (\sum_{i < \text{length } xss'} . 0)$
proof (*rule sum.cong*)
show $\{.. < \text{length } xss'\} = \{.. < \text{length } xss'\}$ **by** *simp*
next
fix *i*
assume *i \in {.. < length xss'}*
hence *xss' ! i \in set xss'* **unfolding** *in-set-conv-nth* **by** *blast*
with *LNil* **have** *xss' ! i = LNil* **by** *auto*
moreover from $\langle i \in \{.. < \text{length } xss'\} \rangle$ **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 = ($\sum_{i < \text{length } xss'} . \text{llength } (\text{lnth } \ xss \ i)) + \text{enat } 0$*
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-llnull-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-llnull1)
  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'. llength (lnth xss i)) =
  (∑ i < length xss'. llength (lnth xss i)) +
  (∑ i = length xss'..<m + length xss'. llength (lnth xss i))
by (subst (1 2) atLeast0LessThan[symmetric]) (subst sum-add-nat-ivl, simp-all)
also from lnth-prefix have (∑ i < length xss'. llength (lnth xss i)) = 0 by
simp
also have {length xss'..<m + length xss'} = {0+length xss'..<m+length xss'}
by auto
also have (∑ i = 0 + length xss'..<m + length xss'. llength (lnth xss i)) =
  (∑ i = 0..<m. llength (lnth xss (i + length xss')))
by (rule sum-shift-bounds-nat-ivl)
also have ... = (∑ i = 0..<m. llength (lnth (LCons (LCons x xs') xss'') i))
unfolding xss by (subst lnth-lappend2) simp+
also have ... = eSuc (llength xs') + (∑ i = Suc 0..<m. llength (lnth (LCons
(LCons x xs') xss'') i))
by (subst sum-head-upt-Suc) simp-all
also {
fix i
assume i ∈ {Suc 0..<m}
then obtain i' where i = Suc i' by (cases i) auto
hence llength (lnth (LCons (LCons x xs') xss'') i) = llength (lnth (LCons
xs' xss'') i)
by simp }
hence (∑ i = Suc 0..<m. llength (lnth (LCons (LCons x xs') xss'') i)) =
  (∑ i = Suc 0..<m. llength (lnth (LCons xs' xss'') i)) by (simp)
also have eSuc (llength xs') + ... = 1 + (llength (lnth (LCons xs' xss'') 0)
+ ...)
by (simp add: eSuc-plus-1 ac-simps)
also note sum-head-upt-Suc[symmetric, OF ‹0 < m›]
finally have enat (Suc n) = (∑ i < m + length xss'. llength (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') < llength (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 $lnth\text{-}prefix$ **have** $(\sum i < length\ xss'. llength\ (lnth\ xss\ i)) = 0$
by $simp$
hence $enat\ (Suc\ n) = (\sum i < length\ xss'. llength\ (lnth\ xss\ i)) + enat\ (Suc\ n')$
using $n\text{-}eq$ **by** $simp$
ultimately show $?thesis$ **by** $blast$
qed
qed

lemma $lprefix\text{-}lconcatI$:
 $xss \sqsubseteq yss \implies lconcat\ xss \sqsubseteq lconcat\ yss$
by($rule\ monotoneD[OF\ monotone\text{-}lconcat]$)

lemma $lnth\text{-}lconcat\text{-}ltake$:
assumes $enat\ w < llength\ (lconcat\ (ltake\ (enat\ n)\ xss))$
shows $lnth\ (lconcat\ (ltake\ (enat\ n)\ xss))\ w = lnth\ (lconcat\ xss)\ w$
using $assms$ **by**($auto\ intro: lprefix\text{-}lnthD\ lprefix\text{-}lconcatI$)

lemma $lfinite\text{-}lconcat$ [$simp$]:
 $lfinite\ (lconcat\ xss) \longleftrightarrow lfinite\ (lfilter\ (\lambda xs. \neg\ lnull\ xs)\ xss) \wedge (\forall xs \in lset\ xss.$
 $lfinite\ xs)$
(is $?lhs \longleftrightarrow ?rhs$ **)**

proof
assume $?lhs$
thus $?rhs$ **(is** $?concl\ xss$ **)**
proof($induct\ lconcat\ xss\ arbitrary: xss$)
case [$symmetric$]: $lfinite\text{-}LNil$
moreover hence $lnull\ (lfilter\ (\lambda xs. \neg\ lnull\ xs)\ xss)$
by($auto\ simp\ add: lconcat\text{-}eq\text{-}LNil$)
ultimately show $?case$ **by**($auto$)
next
case ($lfinite\text{-}LConsI\ xs\ x$)
from $\langle LCons\ x\ xs = lconcat\ xss \rangle [symmetric]$
obtain $xs'\ xss'\ xss''$ **where** $xss [simp]: xss = lappend\ (lset\ of\ xs')\ (LCons$
 $(LCons\ x\ xs')\ xss'')$
and $xs [simp]: xs = lappend\ xs'\ (lconcat\ xss'')$
and $xss': set\ xss' \subseteq \{xs.\ lnull\ xs\}$ **unfolding** $lconcat\text{-}eq\text{-}LCons\text{-}conv$ **by** $blast$
have $xs = lconcat\ (LCons\ xs'\ xss'')$ **by** $simp$
hence $?concl\ (LCons\ xs'\ xss'')$ **by**($rule\ lfinite\text{-}LConsI$)
thus $?case$ **using** $\langle lfinite\ xs \rangle\ xss'$ **by**($auto\ split: if\text{-}split\text{-}asm$)
qed
next
assume $?rhs$
then obtain $lfinite\ (lfilter\ (\lambda xs. \neg\ lnull\ xs)\ xss)$
and $\forall xs \in lset\ xss. lfinite\ xs ..$
thus $?lhs$
proof($induct\ lfilter\ (\lambda xs. \neg\ lnull\ xs)\ xss\ arbitrary: xss$)
case $lfinite\text{-}LNil$

from $\langle LNil = lfilter (\lambda xs. \neg lnull xs) xss \rangle [symmetric]$
have $lset xss \subseteq \{xs. lnull xs\}$ **unfolding** $lfilter\text{-empty}\text{-conv}$ **by** $blast$
hence $lnull (lconcat xss)$ **by** $(simp)$
thus $?case$ **by** $(simp)$
next
case $(lfinite\text{-}LConsI\ x\ x)$
from $lfilter\text{-eq}\text{-}LConsD [OF \langle LCons\ x\ x = lfilter (\lambda xs. \neg lnull xs) xss \rangle [symmetric]]$
obtain $xss'\ xss''$ **where** $xss [simp]: xss = lappend\ xss' (LCons\ x\ xss'')$
and $xss': lfinite\ xss' \wedge lset\ xss' \subseteq \{xs. lnull xs\}$
and $x: \neg lnull\ x$
and $x [simp]: x = lfilter (\lambda xs. \neg lnull xs) xss''$ **by** $blast$
moreover
from $\langle \forall xs \in lset\ xss. lfinite\ xs \rangle xss'$ **have** $\forall xs \in lset\ xss''. lfinite\ xs$ **by** $auto$
with x **have** $lfinite (lconcat\ xss'')$ **by** $(rule\ lfinite\text{-}LConsI)$
ultimately show $?case$ **using** $\langle \forall xs \in lset\ xss. lfinite\ xs \rangle$ **by** $(simp)$
qed
qed

lemma $list\text{-of}\text{-}lconcat$:
assumes $lfinite\ xss$
and $\forall xs \in lset\ xss. lfinite\ xs$
shows $list\text{-of} (lconcat\ xss) = concat (list\text{-of} (lmap\ list\text{-of}\ xss))$
using $assms$ **by** $induct (simp\text{-all}\ add: list\text{-of}\text{-}lappend)$

lemma $lfilter\text{-}lconcat\text{-}lfinite$:
 $\forall xs \in lset\ xss. lfinite\ xs$
 $\implies lfilter\ P (lconcat\ xss) = lconcat (lmap (lfilter\ P) xss)$
by $(induct\ xss) simp\text{-all}$

lemma $lconcat\text{-}repeat\text{-}LNil [simp]$: $lconcat (repeat\ LNil) = LNil$
by $(simp\ add: lconcat\text{-}eq\text{-}LNil)$

lemma $lconcat\text{-}lmap\text{-}singleton [simp]$: $lconcat (lmap (\lambda x. LCons (f\ x) LNil) xs) = lmap\ f\ xs$
by $(induct\ xs) simp\text{-all}$

lemma $lset\text{-}lconcat\text{-}subset$: $lset (lconcat\ xss) \subseteq (\bigcup xs \in lset\ xss. lset\ xs)$
by $(induct\ xss) (auto\ dest: subsetD [OF\ lset\text{-}lappend])$

lemma $ldistinct\text{-}lconcat$:
 $\llbracket ldistinct\ xss; \bigwedge ys. ys \in lset\ xss \implies ldistinct\ ys; \bigwedge ys\ zs. \llbracket ys \in lset\ xss; zs \in lset\ xss; ys \neq zs \rrbracket \implies lset\ ys \cap lset\ zs = \{\} \rrbracket$
 $\implies ldistinct (lconcat\ xss)$
apply $(induction\ arbitrary: xss\ rule: lconcat.\text{fixp}\text{-}induct)$
apply $(auto\ simp\ add: ldistinct\text{-}lappend\ fun\text{-}ord\text{-}def\ split: llist.\text{split})$
apply $(blast\ dest!: subsetD [OF\ lprefix\text{-}lsetD] subsetD [OF\ lset\text{-}lconcat\text{-}subset])$
done

2.27 Sublist view of a lazy list: *lnths*

lemma *lnths-empty* [*simp*]: $lnths\ xs\ \{\} = LNil$
by(*auto simp add: lnths-def split-def lfilter-empty-conv*)

lemma *lnths-LNil* [*simp*]: $lnths\ LNil\ A = LNil$
by(*simp add: lnths-def*)

lemma *lnths-LCons*:
 $lnths\ (LCons\ x\ xs)\ A =$
(if $0 \in A$ *then* $LCons\ x\ (lnths\ xs\ \{n.\ Suc\ n \in A\})$ *else* $lnths\ xs\ \{n.\ Suc\ n \in A\}$)

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*:

$lset\ (lnths\ xs\ I) = \{lnth\ xs\ i \mid i.\ enat\ i < llength\ xs \wedge i \in I\}$

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*: $lset\ (lnths\ xs\ I) \subseteq lset\ xs$

by(*auto simp add: lset-lnths in-lset-conv-lnth*)

lemma *lnths-singleton* [*simp*]:

$lnths\ (LCons\ x\ LNil)\ A = (if\ 0 : A\ then\ LCons\ x\ LNil\ else\ LNil)$

by (*simp add: lnths-LCons*)

lemma *lnths-upt-eq-ltake* [*simp*]:

$lnths\ xs\ \{.. $n\} = ltake\ (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*]:

$l\text{nth} (l\text{list-of } xs) A = l\text{list-of } (n\text{th} xs A)$
by(*induct xs arbitrary: A*)(*simp-all add: lnth-LCons nth-Cons*)

lemma *llength-lnth-ile*: $llength (lnth xs A) \leq llength xs$

proof –

have $llength (lfilter (\lambda(x, y). y \in A) (lzip xs (iterates Suc 0))) \leq$
 $llength (lzip xs (iterates Suc 0))$

by(*rule llength-lfilter-ile*)

thus *?thesis* **by**(*simp add: lnth-def*)

qed

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

$lnth (lmap f xs) A = lmap f (lnth xs A)$

by(*simp add: lnth-def lzip-lmap1 llist.map-comp lfilter-lmap o-def split-def*)

lemma *lfilter-conv-lnth*:

$lfilter P xs = lnth xs \{n. enat n < llength xs \wedge P (lnth xs n)\}$

proof –

have $lnth xs \{n. enat n < llength xs \wedge P (lnth xs n)\} =$

$lmap fst (lfilter (\lambda(x, y). enat y < llength xs \wedge P (lnth xs y))$
 $(lzip xs (iterates Suc 0)))$

by(*simp add: lnth-def*)

also have $\forall (x, y) \in lset (lzip xs (iterates Suc 0)). enat y < llength xs \wedge x = lnth$
 $xs y$

by(*auto simp add: lset-lzip*)

hence $lfilter (\lambda(x, y). enat y < llength xs \wedge P (lnth xs y)) (lzip xs (iterates Suc$
 $0)) =$

$lfilter (P \circ fst) (lzip xs (iterates Suc 0))$

by –(*rule lfilter-cong[OF refl], auto*)

also have $lmap fst (lfilter (P \circ fst) (lzip xs (iterates Suc 0))) =$

$lfilter P (lmap fst (lzip xs (iterates Suc 0)))$

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*: $l\text{length } xs = \text{enat } k$

shows $lnths (lappend\ xs\ ys)\ A =$

$lappend (lnths\ xs\ A) (lnths\ ys\ \{n.\ n + k \in A\})$

proof –

let *?it* = iterates *Suc*

from *assms* have *fin*: *lfinite* *xs* by(rule *llength-eq-enat-lfiniteD*)

have $lnths (lappend\ xs\ ys)\ A =$

$lmap\ fst (lfilter (\lambda(x, y).\ y \in A) (lzip (lappend\ xs\ ys) (?it\ 0)))$

by(*simp add: lnths-def*)

also have $?it\ 0 = lappend (l\text{take } (\text{enat } k) (?it\ 0)) (l\text{drop } (\text{enat } k) (?it\ 0))$

by(*simp only: lappend-ltake-l\text{drop}*)

also note *lzip-lappend*

also note *lfilter-lappend-lfinite*

also note *lmap-lappend-distrib*

also have $lzip\ xs (l\text{take } (\text{enat } k) (?it\ 0)) = lzip\ xs (?it\ 0)$

using *len* by(*subst (1 2) lzip-conv-lzip-ltake-min-l\text{length}*) *simp*

also note *lnths-def[symmetric]*

also have $l\text{drop } (\text{enat } k) (?it\ 0) = ?it\ k$

by(*simp add: l\text{drop-iterates}*)

also { **fix** *n m*

have $?it (n + m) = lmap (\lambda n.\ n + m) (?it\ n)$

by(*coinduction arbitrary: n*)(*force*)+ }

from *this*[of *0 k*] have $?it\ k = lmap (\lambda n.\ n + k) (?it\ 0)$ by *simp*

also note *lzip-lmap2*

also note *lfilter-lmap*

also note *l\text{list.map-comp}*

also have $fst \circ (\lambda(x, y).\ (x, y + k)) = fst$

by(*simp add: o-def split-def*)

also have $(\lambda(x, y).\ y \in A) \circ (\lambda(x, y).\ (x, y + k)) = (\lambda(x, y).\ y \in \{n.\ n + k \in A\})$

by(*simp add: fun-eq-iff*)

also note *lnths-def[symmetric]*

finally show ?thesis using *len* by *simp*

qed

lemma *lnths-split*:

$lnths\ xs\ A =$

$lappend (lnths (l\text{take } (\text{enat } n)\ xs)\ A) (lnths (l\text{dropn } n\ xs)\ \{m.\ n + m \in A\})$

proof(*cases enat n ≤ l\text{length } xs*)

case *False* thus ?thesis by(*auto simp add: l\text{take-all l\text{dropn-all}*)

next

case *True*

have $xs = lappend (l\text{take } (\text{enat } n)\ xs) (l\text{drop } (\text{enat } n)\ xs)$

by(*simp only: lappend-ltake-l\text{drop}*)

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** $A: \bigwedge n. enat\ n < llength\ ys \implies n \in A \longleftrightarrow n \in B$
shows $lnths\ xs\ A = lnths\ ys\ B$

proof –

have $lfilter (\lambda(x, y). y \in A) (lzip\ ys\ (iterates\ Suc\ 0)) =$
 $lfilter (\lambda(x, y). y \in B) (lzip\ ys\ (iterates\ Suc\ 0))$
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 $n: enat\ n < llength\ xs$
shows $lnths\ xs\ (insert\ n\ A) =$
 $lappend (lnths (ltake (enat n) xs) A) (LCons (lnth\ xs\ n)$
 $(lnths (ldropn (Suc\ n) xs) \{m. Suc\ (n + m) \in A\}))$

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 $ldropn\ n\ xs = LCons\ X\ XS$

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 $ldropn\ n\ xs = LCons (lnth\ xs\ n) (ldropn (Suc\ n) xs)$ **by** *simp*

hence $lnths (ldropn\ n\ xs) \{m. n + m \in insert\ n\ A\} =$

$LCons (lnth\ xs\ n) (lnths (ldropn (Suc\ n) xs) \{m. Suc\ (n + m) \in A\})$

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 $lfinite\ xs \vee$

$finite \{n. enat\ n < llength\ xs \wedge (\lambda(x, y). y \in A) (lnth (lzip\ xs\ (iterates\ Suc$

```

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 llist-sum

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*: $\llbracket \text{lfinite } xs; \text{lfinite } ys \rrbracket \implies \text{lsum-list } (\text{lappend } xs \ ys) = \text{lsum-list } xs + \text{lsum-list } ys$

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* $\implies \text{lsum-list } (\text{LCons } x \ xs) = x + \text{lsum-list } xs$

by(*simp add: lsum-list-def*)

lemma *lsum-list-inf* [simp]: $\neg \text{lfinite } xs \implies \text{lsum-list } xs = 0$

by(*simp add: lsum-list-def*)

end

lemma *lsum-list-mono*:

fixes *f* :: 'a \Rightarrow 'b :: {*monoid-add, ordered-ab-semigroup-add*}

assumes $\bigwedge x. x \in \text{lset } xs \implies f \ x \leq g \ x$

shows *lsum-list* (*lmap f xs*) \leq *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 *l*list-of and *inf*-l*l*ist

lemma *lnull-inf-llist* [*simp*]: $\neg \text{lnull } (\text{inf-llist } f)$
by (*simp add: inf-llist-def*)

lemma *inf-llist-neq-LNil*: $\text{inf-llist } f \neq \text{LNil}$
using *lnull-inf-llist unfolding lnull-def* .

lemmas *LNil-neq-inf-llist = inf-llist-neq-LNil*[*symmetric*]

lemma *lhd-inf-llist* [*simp*]: $\text{lhd } (\text{inf-llist } f) = f\ 0$
by (*simp add: inf-llist-def*)

lemma *ltl-inf-llist* [*simp*]: $\text{ltl } (\text{inf-llist } f) = \text{inf-llist } (\lambda n. f\ (\text{Suc } n))$

by (*simp add: inf-llist-def lmap-iterates*[*symmetric*] *l*list.*map-comp*)

lemma *inf-llist-rec* [*code, nitpick-simp*]:
 $\text{inf-llist } f = \text{LCons } (f\ 0) (\text{inf-llist } (\lambda n. f\ (\text{Suc } n)))$
by (*rule llist.expand simp-all*)

lemma *lfinite-inf-llist* [*iff*]: $\neg \text{lfinite } (\text{inf-llist } f)$
by (*simp add: inf-llist-def*)

lemma *iterates-conv-inf-llist*:
 $\text{iterates } f\ a = \text{inf-llist } (\lambda n. (f\ \wedge\ n)\ a)$
by (*coinduction arbitrary: a*)(*auto simp add: funpow-swap1*)

lemma *inf-llist-neq-llist-of* [*simp*]:
 $\text{llist-of } xs \neq \text{inf-llist } f$
 $\text{inf-llist } f \neq \text{llist-of } xs$
using *lfinite-llist-of*[*of xs*] *lfinite-inf-llist*[*of f*] **by** *fastforce*+

lemma *lnth-inf-llist* [*simp*]: $\text{lnth } (\text{inf-llist } f)\ n = f\ n$

proof (*induct n arbitrary: f*)

case 0 **thus** *?case* **by** (*subst inf-llist-rec simp*)

next

case (*Suc n*)

from *Suc*[*of* $\lambda n. f\ (\text{Suc } n)$] **show** *?case*

by (*subst inf-llist-rec simp*)

qed

lemma *inf-llist-lprefix* [*simp*]: $\text{inf-llist } f \sqsubseteq xs \longleftrightarrow xs = \text{inf-llist } f$
by (*auto simp add: not-lfinite-lprefix-conv-eq*)

lemma *llength-inf-llist* [*simp*]: $\text{llength } (\text{inf-llist } f) = \infty$
by (*simp add: inf-llist-def*)

lemma *lset-inf-llist* [*simp*]: $\text{lset } (\text{inf-llist } f) = \text{range } f$

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 *lzip-inf-llist-inf-llist* [*simp*]:

$lzip\ (inf-llist\ f)\ (inf-llist\ g) = inf-llist\ (\lambda n. (f\ n,\ g\ n))$

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 $map\ f [0..<length\ (x\ \# \ xs)] = f\ 0\ \# \ map\ (\lambda n. f\ (Suc\ n)) [0..<length\ xs]$
 by(*simp add: upt-conv-Cons map-Suc-upt[symmetric] del: upt-Suc*)
 with Cons[of $\lambda n. f\ (Suc\ n)$]
 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 $map\ f [0..<length\ (x\ \# \ xs)] = f\ 0\ \# \ map\ (\lambda n. f\ (Suc\ n)) [0..<length\ xs]$
 by(*simp add: upt-conv-Cons map-Suc-upt[symmetric] del: upt-Suc*)
 with Cons[of $\lambda n. f\ (Suc\ n)$]
 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-inflist*: $lsum-list (inf-llist\ f) = 0$
by simp

2.30 Setup for lifting and transfer

2.30.1 Relator and predicator properties

abbreviation *llist-all == pred-llist*

2.30.2 Transfer rules for the Transfer package

context *includes lifting-syntax*
begin

lemma *set1-pre-llist-transfer* [*transfer-rule*]:
 (*rel-pre-llist A B* \implies *rel-set A*) *set1-pre-llist set1-pre-llist*
by(*auto simp add: rel-pre-llist-def vimage2p-def rel-fun-def set1-pre-llist-def rel-set-def collect-def sum-set-defs prod-set-defs elim: rel-sum.cases split: sum.split-asm*)

lemma *set2-pre-llist-transfer* [*transfer-rule*]:
 (*rel-pre-llist A B* \implies *rel-set B*) *set2-pre-llist set2-pre-llist*
by(*auto simp add: rel-pre-llist-def vimage2p-def rel-fun-def set2-pre-llist-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* \implies *llist-all2 A* \implies *llist-all2 A*) *LCons LCons*
unfolding *rel-fun-def* **by** *simp*

lemma *case-llist-transfer* [*transfer-rule*]:
 (*B* \implies (*A* \implies *llist-all2 A* \implies *B*) \implies *llist-all2 A* \implies *B*)
case-llist case-llist
unfolding *rel-fun-def* **by** (*simp split: llist.split*)

lemma *unfold-llist-transfer* [*transfer-rule*]:
 ((*A* \implies *op* =) \implies (*A* \implies *B*) \implies (*A* \implies *A*) \implies *A* \implies *llist-all2 B*) *unfold-llist unfold-llist*
proof(*rule rel-funI*)
fix *IS-LNIL1* :: '*a* \Rightarrow *bool* **and** *IS-LNIL2 LHD1 LHD2 LTL1 LTL2 x y*
assume *rel: (A* \implies *op* =) *IS-LNIL1 IS-LNIL2 (A* \implies *B*) *LHD1 LHD2 (A* \implies *A*) *LTL1 LTL2*
and *A x y*
from $\langle A x y \rangle$
show *llist-all2 B (unfold-llist IS-LNIL1 LHD1 LTL1 x) (unfold-llist 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* \implies *op* =) \implies (*A* \implies *B*) \implies (*A* \implies *op* =) \implies (*A* \implies *llist-all2 B*) \implies (*A* \implies *A*) \implies *A* \implies *llist-all2 B*) *corec-llist corec-llist*
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* \implies *op* =) *IS-LNIL1 IS-LNIL2 (A* \implies *B*) *LHD1 LHD2 (A* \implies *op* =) *STOP1 STOP2 (A* \implies *llist-all2 B*) *MORE1 MORE2 (A* \implies *A*) *LTL1 LTL2*

```

assume  $A\ x\ y$ 
thus  $l\text{list-all2}\ B\ (\text{corec-llist}\ IS\text{-LNIL1}\ LHD1\ STOP1\ MORE1\ LTL1\ x)\ (\text{corec-llist}\ IS\text{-LNIL2}\ LHD2\ STOP2\ MORE2\ LTL2\ y)$ 
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] =  $\langle A\ x\ y \rangle$ 
  have  $?lhd$  by(simp add: LCons[simplified]) transfer-prover
  moreover
  have  $STOP1\ x = STOP2\ y\ l\text{list-all2}\ B\ (MORE1\ x)\ (MORE2\ y)\ A\ (LTL1\ x)\ (LTL2\ y)$  by transfer-prover+
  hence  $?ltl$  by(auto simp add: LCons[simplified])
  ultimately show  $?case\ ..$ 
qed
qed

```

```

lemma ltl-transfer [transfer-rule]:
  ( $l\text{list-all2}\ A\ ==>\ l\text{list-all2}\ A$ ) ltl ltl
unfolding ltl-def[abs-def] by transfer-prover

```

```

lemma lset-transfer [transfer-rule]:
  ( $l\text{list-all2}\ A\ ==>\ \text{rel-set}\ A$ ) 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]:
  ( $(A\ ==>\ B)\ ==>\ l\text{list-all2}\ A\ ==>\ l\text{list-all2}\ B$ ) lmap lmap
by(auto simp add: rel-fun-def llist-all2-lmap1 llist-all2-lmap2 elim: llist-all2-mono)

```

```

lemma lappend-transfer [transfer-rule]:
  ( $l\text{list-all2}\ A\ ==>\ l\text{list-all2}\ A\ ==>\ l\text{list-all2}\ A$ ) lappend lappend
by(auto simp add: rel-fun-def intro: llist-all2-lappendI)

```

```

lemma iterates-transfer [transfer-rule]:
  ( $(A\ ==>\ A)\ ==>\ A\ ==>\ l\text{list-all2}\ A$ ) iterates iterates
unfolding iterates-def split-def corec-llist-never-stop by transfer-prover

```

```

lemma lfinite-transfer [transfer-rule]:
  ( $l\text{list-all2}\ A\ ==>\ \text{op}\ =$ ) lfinite lfinite
by(auto dest: llist-all2-lfiniteD)

```

```

lemma l\text{list-of-transfer} [transfer-rule]:
  ( $l\text{list-all2}\ A\ ==>\ l\text{list-all2}\ A$ ) l\text{list-of}\ l\text{list-of}
unfolding l\text{list-of-def} by transfer-prover

```

```

lemma l\text{length-transfer} [transfer-rule]:
  ( $l\text{list-all2}\ A\ ==>\ \text{op}\ =$ ) l\text{length}\ l\text{length}

```

by(*auto dest: llist-all2-llengthD*)

lemma *ltake-transfer* [*transfer-rule*]:

(*op* = \implies *llist-all2 A* \implies *llist-all2 A*) *ltake ltake*
by(*auto intro: llist-all2-ltakeI*)

lemma *ldropn-transfer* [*transfer-rule*]:

(*op* = \implies *llist-all2 A* \implies *llist-all2 A*) *ldropn ldrown*
unfolding *ldrown-def*[*abs-def*] **by** *transfer-prover*

lemma *ldrop-transfer* [*transfer-rule*]:

(*op* = \implies *llist-all2 A* \implies *llist-all2 A*) *ldrop ldrown*
by(*auto intro: llist-all2-ldropI*)

lemma *ltakeWhile-transfer* [*transfer-rule*]:

((*A* \implies *op* =) \implies *llist-all2 A* \implies *llist-all2 A*) *ltakeWhile ltakeWhile*
proof(*rule rel-funI*)+

fix *P* :: '*a* \Rightarrow *bool* **and** *Q* :: '*b* \Rightarrow *bool* **and** *xs* :: '*a* *l*list **and** *ys* :: '*b* *l*list

assume *PQ*: (*A* \implies *op* =) *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-null-conv llist-all2-LCons2*
llist-all2-LCons1)

qed

lemma *ldropWhile-transfer* [*transfer-rule*]:

((*A* \implies *op* =) \implies *llist-all2 A* \implies *llist-all2 A*) *ldropWhile ldrownWhile*
unfolding *ldrownWhile-eq-ldrown*[*abs-def*] **by** *transfer-prover*

lemma *lzip-ltransfer* [*transfer-rule*]:

(*llist-all2 A* \implies *llist-all2 B* \implies *llist-all2 (rel-prod A B)*) *lzip lzip*
by(*auto intro: llist-all2-lzipI*)

lemma *inf-llist-transfer* [*transfer-rule*]:

((*op* = \implies *A*) \implies *llist-all2 A*) *inf-llist inf-llist*
unfolding *inf-llist-def*[*abs-def*] **by** *transfer-prover*

lemma *lfilter-transfer* [*transfer-rule*]:

((*A* \implies *op* =) \implies *llist-all2 A* \implies *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)* \implies *llist-all2 A*) *lconcat lconcat*
by(*auto intro: llist-all2-lconcatI*)

lemma *lnths-transfer* [*transfer-rule*]:

(*llist-all2 A* \implies *op* = \implies *llist-all2 A*) *lnths lnths*
unfolding *lnths-def*[*abs-def*] **by** *transfer-prover*

lemma *l1ist-all-transfer* [*transfer-rule*]:
 $((A \text{====>} op =) \text{====>} \text{l1ist-all2 } A \text{====>} op =) \text{l1ist-all l1ist-all}$
unfolding *pred-l1ist-def*[*abs-def*] **by** *transfer-prover*

lemma *l1ist-all2-rsp*:
assumes *r*: $\forall x y. R x y \longrightarrow (\forall a b. R a b \longrightarrow S x a = T y b)$
and *l1*: *l1ist-all2* *R* *x* *y*
and *l2*: *l1ist-all2* *R* *a* *b*
shows *l1ist-all2* *S* *x* *a* = *l1ist-all2* *T* *y* *b*
proof(*cases* *l1* *length* *x* = *length* *a*)
case *True*
thus ?*thesis* **using** *l1* *l2*
by(*simp* *add*: *l1ist-all2-conv-all-lnth*)(*blast* *dest*: *r*[*rule-format*])
next
case *False*
with *l1ist-all2-lengthD*[*OF* *l1*] *l1ist-all2-lengthD*[*OF* *l2*]
show ?*thesis* **by**(*simp* *add*: *l1ist-all2-conv-all-lnth*)
qed

lemma *l1ist-all2-transfer* [*transfer-rule*]:
 $((R \text{====>} R \text{====>} op =) \text{====>} \text{l1ist-all2 } R \text{====>} \text{l1ist-all2 } R \text{====>} op =)$
 $\text{l1ist-all2 l1ist-all2}$
by (*simp* *add*: *l1ist-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 *l1ist* :: (*type*) *order* **begin**

definition [*code-unfold*]: $xs \leq ys = \text{lprefix } xs \ ys$

definition [*code-unfold*]: $xs < ys = \text{lstrict-prefix } xs \ ys$

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]: op ≤ = lprefix
by(simp add: less-eq-llist-def fun-eq-iff)

```

```

lemma less-llist-conv-lstrict-prefix [iff]: op < = 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 *l*list-*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*: $P\ x \implies eq\text{-onp}\ P\ x\ x$

by (*simp add: eq-onp-def*)

primcorec *unfold-stream* :: $('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'b\ stream$ **where**

unfold-stream $g1\ g2\ a = g1\ a\ \#\#\ unfold\text{-stream}\ g1\ g2\ (g2\ a)$

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 $f\ xs = y\ \#\#\ ys \longleftrightarrow$

$(\exists\ xs'. xs = x\ \#\#\ xs' \wedge y = f\ x \wedge ys = smap\ f\ xs')$

by (*cases xs*)(*auto*)

lemma *smap-unfold-stream*:

smap $f\ (unfold\text{-stream}\ SHD\ STL\ b) = unfold\text{-stream}\ (f \circ SHD)\ STL\ b$

by (*coinduction arbitrary: b*) *auto*

lemma *smap-corec-stream*:

smap $f\ (corec\text{-stream}\ SHD\ endORmore\ STL\text{-end}\ STL\text{-more}\ b) =$

$corec\text{-stream}\ (f \circ SHD)\ endORmore\ (smap\ f \circ STL\text{-end})\ STL\text{-more}\ b$

by (*coinduction arbitrary: b rule: stream.coinduct-strong*) *auto*

lemma *unfold-stream-ltl-unroll*:

unfold-stream $SHD\ STL\ (STL\ b) = unfold\text{-stream}\ (SHD \circ STL)\ STL\ b$

by (*coinduction arbitrary: b*) *auto*

lemma *unfold-stream-eq-SCons* [*simp*]:

unfold-stream $SHD\ STL\ b = x\ \#\#\ xs \longleftrightarrow$

$x = SHD\ b \wedge xs = unfold\text{-stream}\ SHD\ STL\ (STL\ b)$

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.set-sel(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 *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 *szip (xs @- ys) (us @- zs) = zip xs us @- szip ys zs*
by(*induct xs arbitrary: us*)(*auto simp add: Suc-length-conv*)

4.2 Link 'a stream to 'a llist

definition *lstream-of-stream* :: 'a stream \Rightarrow 'a llist

where *llist-of-stream* = *unfold-llist* (λ -. *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 *lnull* (*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 *lfinite xs* \implies *llist-of-stream (stream-of-llist xs) = xs*
by(*coinduction arbitrary: xs auto*)

lemma *lfinite-llist-of-stream* [*simp*]: \neg *lfinite* (*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 lfinite xs}*
by(*unfold-locales simp-all*)

interpretation *stream*: *type-definition llist-of-stream stream-of-llist {xs. \neg 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 *lfinite* *xs* \implies *cr-stream* *xs* (*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* (λ -. *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 (λ -. *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* \in ?*lhs*

thus *x* \in ?*rhs*

by(*induct* *llist-of-stream* *xs* *arbitrary*: *xs* *rule*: *llist-set-induct*)
(*auto* *dest*: *stream.set-sel*(2))

next

fix *x*

assume *x* \in ?*rhs*

thus *x* \in ?*lhs*

proof(*induct*)

case (*shd* *xs*)

thus ?*case* **using** *llist.set-sel*(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*]:

$l\text{nth } (\text{l\textit{list-of-stream } } xs) = s\text{nth } xs$
proof
fix n
show $l\text{nth } (\text{l\textit{list-of-stream } } xs) n = s\text{nth } xs n$
by($\textit{induction } n \textit{ arbitrary: } xs$)($\textit{simp-all add: l\textit{nth-0-conv-lhd } l\textit{nth-ll}[symmetric]}$)
qed

lemma $\textit{l\textit{list-of-stream-siterates } [simp]: l\textit{list-of-stream } (s\textit{iterate } f x) = i\textit{terates } f x}$
by($\textit{coinduction arbitrary: } x$) \textit{auto}

lemma $\textit{l\textit{append-l\textit{list-of-stream-conv-shift } [simp]:}$
 $\textit{l\textit{append } (l\textit{list-of } } xs) (l\textit{list-of-stream } } ys) = l\textit{list-of-stream } (xs @- ys)$
by($\textit{induct } xs$) $\textit{simp-all}$

lemma $\textit{l\textit{zip-l\textit{list-of-stream } [simp]:}$
 $\textit{l\textit{zip } (l\textit{list-of-stream } } xs) (l\textit{list-of-stream } } ys) = l\textit{list-of-stream } (s\textit{zip } xs ys)$
by($\textit{coinduction arbitrary: } xs \textit{ } ys$) \textit{auto}

context includes $\textit{lifting-syntax}$
begin

lemma $\textit{l\textit{map-infinite-transfer } [transfer-rule]:}$
 $(op = \textit{====>} eq\textit{-onp } (\lambda xs. \neg l\textit{finite } xs) \textit{====>} eq\textit{-onp } (\lambda xs. \neg l\textit{finite } xs)) \textit{lmap}$
 \textit{lmap}
by($\textit{simp add: rel-fun-def eq-onp-def}$)

lemma $\textit{l\textit{set-infinite-transfer } [transfer-rule]:}$
 $(eq\textit{-onp } (\lambda xs. \neg l\textit{finite } xs) \textit{====>} op =) \textit{lset lset}$
by($\textit{simp add: rel-fun-def eq-onp-def}$)

lemma $\textit{unfold-stream-transfer } [transfer-rule]:$
 $(op = \textit{====>} op = \textit{====>} op = \textit{====>} p\textit{cr-stream } op =) (\textit{unfold-l\textit{list } } (\lambda-. \textit{False})) \textit{unfold-stream}$
by($\textit{auto simp add: stream.pcr-cr-eq cr-stream-def intro!: rel-funI}$)

lemma $\textit{corec-stream-transfer } [transfer-rule]:$
 $(op = \textit{====>} op = \textit{====>} (op = \textit{====>} p\textit{cr-stream } op =) \textit{====>} op = \textit{====>} op = \textit{====>} p\textit{cr-stream } op =)$
 $(\textit{corec-l\textit{list } } (\lambda-. \textit{False})) \textit{corec-stream}$
apply($\textit{auto intro!: rel-funI simp add: cr-stream-def stream.pcr-cr-eq}$)
apply($\textit{rule fun-cong}$) **back**
apply($\textit{rule-tac } x=yc \textit{ in fun-cong}$)
apply($\textit{rule-tac } x=xb \textit{ in arg-cong}$)
apply($\textit{auto elim: rel-funE}$)
done

lemma $\textit{shd-transfer } [transfer-rule]: (p\textit{cr-stream } A \textit{====>} A) \textit{lhd shd}$
by($\textit{auto simp add: pcr-stream-def cr-stream-def intro!: rel-funI relcomppI}(f\textit{rule l\textit{list-all2-lhdD, auto dest: l\textit{list-all2-lnullD})}$)

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 *lstream-transfer* [*transfer-rule*]: (*pcr-stream op =* ====> *op =*) *id*
lstream
by(*simp add: rel-fun-def stream.pcr-cr-eq cr-stream-def*)

lemma *stream-of-llist-transfer* [*transfer-rule*]:
(*eq-onp* ($\lambda xs. \neg \text{lfinite } xs$) ====> *pcr-stream op =*) ($\lambda xs. xs$) *stream-of-llist*
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 op ##*
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 op =* ====> *op =*) *lnth snth*
by(*rule rel-funI*)(*clarsimp simp add: stream.pcr-cr-eq cr-stream-def fun-eq-iff*)

lemma *siterate-transfer* [*transfer-rule*]:
(*op =* ====> *op =* ====> *pcr-stream op =*) *iterates siterate*
by(*rule rel-funI*)+(*clarsimp simp add: stream.pcr-cr-eq cr-stream-def*)

context

fixes *xs*

assumes *inf*: $\neg \text{lfinite } xs$

notes [*transfer-rule*] = *eq-onpI*[**where** $P = \lambda xs. \neg \text{lfinite } xs$, *OF inf*]

begin

lemma *smap-stream-of-llist* [*simp*]:
shows *smap f* (*stream-of-llist xs*) = *stream-of-llist* (*lmap f xs*)
by *transfer simp*

lemma *sset-stream-of-llist* [*simp*]:
assumes $\neg \text{lfinite } xs$
shows *sset* (*stream-of-llist xs*) = *lset xs*
by *transfer simp*

end

lemma *l1ist-all2-l1ist-of-stream* [*simp*]:

l1ist-all2 *P* (*l1ist-of-stream* *xs*) (*l1ist-of-stream* *ys*) = *stream-all2* *P* *xs* *ys*
apply(*cases* *xs* *ys* *rule*: *stream.Abs-cases*[*case-product* *stream.Abs-cases*])
apply(*simp* *add*: *l1ist-all2-def* *stream-all2-def*)
apply(*safe* *elim*!: *GrpE*)
apply(*rule-tac* *b*=*stream-of-l1ist* *b* **in** *relcomppI*; *auto* *intro*!: *GrpI*)
apply(*rule-tac* *b*=*l1ist-of-stream* *b* **in** *relcomppI*; *auto* *intro*!: *GrpI*)
done

lemma *stream-all2-transfer* [*transfer-rule*]:

(*op* = \implies *pcr-stream* *op* = \implies *pcr-stream* *op* = \implies *op* =) *l1ist-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 $\bigwedge xs\ ys. X\ xs\ ys \implies P\ (shd\ xs)\ (shd\ ys) \wedge (X\ (stl\ xs)\ (stl\ ys) \vee stream-all2\ P\ (stl\ xs)\ (stl\ ys))$
shows *stream-all2* *P* *xs* *ys*
using *assms*
apply *transfer*
apply(*rule-tac* *X*= $\lambda xs\ ys. \neg\ lfinite\ xs \wedge \neg\ lfinite\ ys \wedge X\ xs\ ys$ **in** *l1ist-all2-coinduct*)
apply *auto*
done

lemma *shift-transfer* [*transfer-rule*]:

(*op* = \implies *pcr-stream* *op* = \implies *pcr-stream* *op* =) (*lappend* \circ *l1ist-of*) *shift*
by(*clarsimp* *simp* *add*: *rel-fun-def* *stream.pcr-cr-eq* *cr-stream-def*)

lemma *szip-transfer* [*transfer-rule*]:

(*pcr-stream* *op* = \implies *pcr-stream* *op* = \implies *pcr-stream* *op* =) *lzip* *szip*
by(*simp* *add*: *stream.pcr-cr-eq* *cr-stream-def* *rel-fun-def*)

4.3 Link 'a stream with nat \Rightarrow 'a

lift-definition *of-seq* :: (*nat* \Rightarrow 'a) \Rightarrow 'a stream **is** *inf-l1ist* **by** *simp*

lemma *of-seq-rec* [*code*]: *of-seq* *f* = *f* 0 ## *of-seq* (*f* \circ *Suc*)

by *transfer* (*subst* *inf-l1ist-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 \Rightarrow *x* | *Suc* *n'* \Rightarrow *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*: $(x \#\# xs) !! \text{Suc } n = xs !! n$
by(*simp-all add: snth-SCons*)

lemma *of-seq-snth* [*simp*]: $\text{of-seq } (\text{snth } xs) = xs$
by *transfer simp*

lemma *shd-of-seq* [*simp*]: $\text{shd } (\text{of-seq } f) = f 0$
by *transfer simp*

lemma *stl-of-seq* [*simp*]: $\text{stl } (\text{of-seq } f) = \text{of-seq } (\lambda n. f (\text{Suc } n))$
by *transfer simp*

lemma *sset-of-seq* [*simp*]: $\text{sset } (\text{of-seq } f) = \text{range } f$
by *transfer simp*

lemma *smap-of-seq* [*simp*]: $\text{smap } f (\text{of-seq } g) = \text{of-seq } (f \circ g)$
by *transfer simp*
end

4.4 Function iteration *siterate* and *sconst*

lemmas *siterate* [*nitpick-simp*] = *siterate.code*

lemma *smap-iterates*: $\text{smap } f (\text{siterate } f x) = \text{siterate } f (f x)$
by *transfer (rule lmap-iterates)*

lemma *siterate-smap*: $\text{siterate } f x = x \#\# (\text{smap } f (\text{siterate } f x))$
by *transfer (rule iterates-lmap)*

lemma *siterate-conv-of-seq*: $\text{siterate } f a = \text{of-seq } (\lambda n. (f \hat{\ }^n) a)$
by *transfer (rule iterates-conv-inf-llist)*

lemma *sconst-conv-of-seq*: $\text{sconst } a = \text{of-seq } (\lambda -. a)$
by(*simp add: siterate-conv-of-seq*)

lemma *szip-sconst1* [*simp*]: $\text{szip } (\text{sconst } a) xs = \text{smap } (\text{Pair } a) xs$
by(*coinduction arbitrary: xs*) *auto*

lemma *szip-sconst2* [*simp*]: $\text{szip } xs (\text{sconst } b) = \text{smap } (\lambda x. (x, b)) xs$
by(*coinduction arbitrary: xs*) *auto*

end

4.5 Counting elements

partial-function (*lfp*) *scount* :: $('s \text{ stream} \Rightarrow \text{bool}) \Rightarrow 's \text{ stream} \Rightarrow \text{enat}$ **where**
scount $P \omega = (\text{if } P \omega \text{ then } e\text{Suc } (\text{scount } P (\text{stl } \omega)) \text{ else } \text{scount } P (\text{stl } \omega))$

lemma *scount-simps*:

```

P ω ⇒ scout P ω = eSuc (scout P (stl ω))
¬ P ω ⇒ scout P ω = scout P (stl ω)
using scout.simps[of P ω] by auto

lemma scout-eq-0I: alw (not P) ω ⇒ scout P ω = 0
  by (induct arbitrary: ω rule: scout.fixp-induct)
  (auto simp: bot-enat-def intro!: admissible-all admissible-imp admissible-eq-mcontI
mcont-const)

lemma scout-eq-0D: scout P ω = 0 ⇒ alw (not P) ω
proof (induction rule: alw.coinduct)
  case (alw ω) with scout.simps[of P ω] show ?case
    by (simp split: if-split-asm)
qed

lemma scout-eq-0-iff: scout P ω = 0 ⇔ alw (not P) ω
  by (metis scout-eq-0D scout-eq-0I)

lemma
  assumes ev (alw (not P)) ω
  shows scout-eq-card: scout 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 {}) ∪ (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: scout.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: scout-eq-0I enat-0) }
  { case 2 show ?case by simp }
qed

lemma scout-finite: ev (alw (not P)) ω ⇒ scout P ω < ∞
  using scout-eq-card[of P ω] by auto

lemma scout-infinite:
  alw (ev P) ω ⇒ scout 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 $\omega = \infty \longleftrightarrow$ alw (ev P) ω*
by (*metis enat-ord-simps(4) not-alw-not scount-finite scount-infinite*)

lemma *scount-eq:*
scount P $\omega =$ (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*) *sfirst :: ('s stream \Rightarrow bool) \Rightarrow 's stream \Rightarrow enat where*
sfirst P $\omega =$ (if P ω then 0 else eSuc (sfirst P (stl ω)))

lemma *sfirst-eq-0: sfirst P $\omega = 0 \longleftrightarrow$ P ω*
by (*subst sfirst.simps*) *auto*

lemma *sfirst-0[simp]: P $\omega \Longrightarrow$ sfirst P $\omega = 0$*
by (*subst sfirst.simps*) *auto*

lemma *sfirst-eSuc[simp]: \neg P $\omega \Longrightarrow$ sfirst P $\omega =$ eSuc (sfirst P (stl ω))*
by (*subst sfirst.simps*) *auto*

lemma *less-sfirstD:*
fixes *n :: nat*
assumes *enat n < sfirst P ω shows \neg P (sdrop n ω)*
using *assms*
proof (*induction n arbitrary: ω*)
case (*Suc n*) **then show** *?case*
by (*auto simp: sfirst.simps[of - ω] eSuc-enat[symmetric] split: if-split-asm*)
qed (*simp add: enat-0 sfirst-eq-0*)

lemma *sfirst-finite: sfirst P $\omega < \infty \longleftrightarrow$ ev P ω*

proof
assume *sfirst P $\omega < \infty$*
then obtain *n where sfirst P $\omega =$ 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] sfirst.simps[of P ω] split: if-split-asm*)
qed (*auto simp add: enat-0 sfirst-eq-0*)
next
assume *ev P ω then show sfirst P $\omega < \infty$*

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*: $(not\ P\ until\ (alw\ P))\ \omega \implies enat\ n < sfirst\ P\ \omega \longleftrightarrow \neg\ P\ (sdrop\ n\ \omega)$

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.prem*s have **: $P\ \omega \implies P\ (stl\ \omega)$

by (auto elim: *UNTIL.cases*)

have *: $\neg\ P\ (sdrop\ n\ (stl\ \omega)) \longleftrightarrow enat\ n < sfirst\ P\ (stl\ \omega)$

using *Suc.prem*s by (intro *Suc.IH[symmetric]*) (auto intro: *UNTIL.intros elim: UNTIL.cases*)

show ?case

unfolding *sdrop.simps* * by (cases $P\ \omega$) (simp-all add: ** *eSuc-enat[symmetric]*)

qed

lemma *sfirst-eq-Inf*: $sfirst\ P\ \omega = Inf\ \{enat\ i \mid i.\ P\ (sdrop\ i\ \omega)\}$

proof (rule *antisym*)

show $sfirst\ P\ \omega \leq \sqcap\ \{enat\ i \mid i.\ P\ (sdrop\ i\ \omega)\}$

proof (safe intro!: *Inf-greatest*)

fix $\omega\ i$ assume $P\ (sdrop\ i\ \omega)$ then show $sfirst\ P\ \omega \leq enat\ i$

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 $\sqcap\ \{enat\ i \mid i.\ P\ (sdrop\ i\ \omega)\} \leq sfirst\ P\ \omega$

proof (induction arbitrary: ω rule: *sfirst.fixp-induct*)

case (3 *f*)

have $\{enat\ i \mid i.\ P\ (sdrop\ i\ \omega)\} = (if\ P\ \omega\ then\ \{0\}\ else\ \{\}) \cup eSuc\ \{enat\ i \mid i.\ P\ (sdrop\ i\ (stl\ \omega))\}$

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*: $sfirst\ P\ \omega = enat\ n \longleftrightarrow ev-at\ P\ n\ \omega$

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 (lstream 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-lstream[symmetric] del: LCons-lstream*)

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 = lstream 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*: $R (fst p) = (\forall x y. p = (x, y) \longrightarrow R x)$
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**

$p\ a \implies \text{unfold-tl}list\ p\ g1\ g21\ g22\ a = TNil\ (g1\ a) \mid$
 $- \implies \text{unfold-tl}list\ p\ g1\ g21\ g22\ a =$
 $TCons\ (g21\ a)\ (\text{unfold-tl}list\ p\ g1\ g21\ g22\ (g22\ a))$

declare

$\text{unfold-tl}list.ctr(1)\ [simp]$
 $\text{tl}list.corec(1)\ [simp]$

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 $\text{ttl-unfold-tl}list$:
 $\text{ttl}\ (\text{unfold-tl}list\ IS-TNIL\ TNIL\ THD\ TTL\ a) =$
 $(if\ IS-TNIL\ a\ then\ TNil\ (TNIL\ a)\ else\ \text{unfold-tl}list\ IS-TNIL\ TNIL\ THD\ TTL$
 $(TTL\ a))$
by $(simp)$

lemma $is-TNil\ \text{ttl}\ [simp]$: $is-TNil\ xs \implies is-TNil\ (\text{ttl}\ xs)$
by $(cases\ xs)\ simp-all$

lemma $terminal\ \text{ttl}\ [simp]$: $terminal\ (\text{ttl}\ xs) = terminal\ xs$
by $(cases\ xs)\ simp-all$

lemma $\text{unfold-tl}list\ eq\ TNil\ [simp]$:
 $\text{unfold-tl}list\ IS-TNIL\ TNIL\ THD\ TTL\ a = TNil\ b \iff IS-TNIL\ a \wedge b = TNIL$
 a
by $(auto\ simp\ add:\ \text{unfold-tl}list.code)$

lemma $TNil\ eq\ \text{unfold-tl}list\ [simp]$:
 $TNil\ b = \text{unfold-tl}list\ IS-TNIL\ TNIL\ THD\ TTL\ a \iff IS-TNIL\ a \wedge b = TNIL$
 a
by $(auto\ simp\ add:\ \text{unfold-tl}list.code)$

lemma tmap-is-TNil : $is-TNil\ xs \implies \text{tmap}\ f\ g\ xs = TNil\ (g\ (terminal\ xs))$
by $(clarsimp\ simp\ add:\ is-TNil-def)$

declare $\text{tl}list.map\ sel(2)\ [simp]$

lemma $\text{ttl-tmap}\ [simp]$: $\text{ttl}\ (\text{tmap}\ f\ g\ xs) = \text{tmap}\ f\ g\ (\text{ttl}\ xs)$

by(cases xs) simp-all

lemma tmap-eq-TNil-conv:

$tmap\ f\ g\ xs = TNil\ y \iff (\exists\ y'.\ xs = TNil\ y' \wedge g\ y' = y)$

by(cases xs) simp-all

lemma TNil-eq-tmap-conv:

$TNil\ y = tmap\ f\ g\ xs \iff (\exists\ y'.\ xs = TNil\ y' \wedge g\ y' = y)$

by(cases xs) auto

declare tllist.set-sel(1)[simp]

lemma tset-ttl: tset (ttl xs) \subseteq tset xs

by(cases xs) auto

lemma in-tset-ttlD: $x \in tset\ (ttl\ xs) \implies x \in tset\ xs$

using tset-ttl[of xs] **by** auto

theorem tllist-set-induct[consumes 1, case-names find step]:

assumes $x \in tset\ xs$ **and** $\bigwedge xs.\ \neg\ is-TNil\ xs \implies P\ (thd\ xs)\ xs$

and $\bigwedge xs\ y.\ [\neg\ is-TNil\ xs; y \in tset\ (ttl\ xs); P\ y\ (ttl\ xs)] \implies P\ y\ xs$

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 $x \in set2-tllist\ xs$ **and** $\bigwedge xs.\ is-TNil\ xs \implies P\ (terminal\ xs)\ xs$

and $\bigwedge xs\ y.\ [\neg\ is-TNil\ xs; y \in set2-tllist\ (ttl\ xs); P\ y\ (ttl\ xs)] \implies P\ y\ xs$

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**

$tllist-of-llist\ xs = (case\ xs\ of\ LNil \Rightarrow TNil\ b \mid LCons\ x\ xs' \Rightarrow TCons\ x\ (tllist-of-llist\ xs'))$

end

primcorec llist-of-tllist :: ('a, 'b) tllist \Rightarrow 'a llist

where llist-of-tllist xs = (case xs of TNil - \Rightarrow LNil \mid 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 tll-tllist-of-llist [simp]: tll (tllist-of-llist b xs) = tllist-of-llist b (tll 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-iff(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 (ltl 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  $\langle$ xs = xs $\rangle$ )
  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*]:
 $(B \implies pcr\ tllist\ A\ B)\ (Pair\ LNil)\ TNil$
by(*force simp add: pcr-tllist-def cr-tllist-def*)

lemma *TCons-transfer* [*transfer-rule*]:
 $(A \implies pcr\ tllist\ A\ B \implies pcr\ tllist\ A\ B)\ (apfst \circ LCons)\ TCons$
by(*force simp add: pcr-tllist-def llist-all2-LCons1 cr-tllist-def*)

lemma *tmap-tllist-of-llist*:

$tmap\ f\ g\ (tlist\ of\ llist\ b\ xs) = tlist\ of\ llist\ (g\ b)\ (lmap\ f\ xs)$
by(*coinduction arbitrary: xs*)(*auto simp add: tmap-is-TNil*)

lemma *tmap-transfer* [*transfer-rule*]:

$(op = \implies op = \implies pcr\ tlist\ op = op = \implies pcr\ tlist\ op = op =)$
 $(map\ prod \circ lmap)\ tmap$
by(*force simp add: cr-tlist-def tlist.pcr-cr-eq tmap-tlist-of-llist*)

lemma *lset-llist-of-tlist* [*simp*]:

$lset\ (llist\ of\ tlist\ xs) = tset\ xs\ (is\ ?lhs = ?rhs)$

proof(*intro set-eqI iffI*)

fix x

assume $x \in ?lhs$

thus $x \in ?rhs$

by(*induct llist-of-tlist xs arbitrary: xs rule: llist-set-induct*)(*auto simp: tlist.set-sel(2)*)

next

fix x

assume $x \in ?rhs$

thus $x \in ?lhs$

proof(*induct rule: tlist-set-induct*)

case (*find xs*)

thus $?case$ **by**(*cases xs*) *auto*

next

case *step*

thus $?case$

by(*auto simp add: ltl-llist-of-tlist[symmetric] simp del: ltl-llist-of-tlist dest: in-lset-ltlD*)

qed

qed

lemma *tset-tlist-of-llist* [*simp*]:

$tset\ (tlist\ of\ llist\ b\ xs) = lset\ xs$

by(*simp add: lset-llist-of-tlist[symmetric] del: lset-llist-of-tlist*)

lemma *tset-transfer* [*transfer-rule*]:

$(pcr\ tlist\ op = op = \implies op =) (lset \circ fst)\ tset$

by(*auto simp add: cr-tlist-def tlist.pcr-cr-eq*)

lemma *is-TNil-transfer* [*transfer-rule*]:

$(pcr\ tlist\ op = op = \implies op =) (\lambda(xs, b). lnull\ xs)\ is\ TNil$

by(*auto simp add: tlist.pcr-cr-eq cr-tlist-def*)

lemma *thd-transfer* [*transfer-rule*]:

$(pcr\ tlist\ op = op = \implies op =) (lhd \circ fst)\ thd$

by(*auto simp add: cr-tlist-def tlist.pcr-cr-eq*)

lemma *ttl-transfer* [*transfer-rule*]:

$(pcr\ tlist\ A\ B \implies pcr\ tlist\ A\ B) (apfst\ ttl)\ ttl$

by(force simp add: pcr-tllist-def cr-tllist-def intro: llist-all2-llI)

lemma *llist-of-tllist-transfer* [transfer-rule]:
 (pcr-tllist op = B ===> op =) fst llist-of-tllist
by(auto simp add: pcr-tllist-def cr-tllist-def llist.rel-eq)

lemma *tllist-of-llist-transfer* [transfer-rule]:
 (op ===> op ===> pcr-tllist op = op =) ($\lambda b xs. (xs, b)$) tllist-of-llist
by(auto simp add: tllist.pcr-cr-eq cr-tllist-def)

lemma *terminal-tllist-of-llist-lfinite* [simp]:
 lfinite xs \implies 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 $x \in \text{set2-tllist (tllist-of-llist b xs)}$
 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) ($\lambda(xs, b). \text{if lfinite xs then } \{b\} \text{ else } \{\}$) 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]:
 (op ===> op ===> pcr-tllist op = op = ===> pcr-tllist op = op =
 ===> op =)
 ($\lambda P Q (xs, b) (ys, b'). \text{llist-all2 } P \text{ xs ys} \wedge (\text{lfinite xs} \implies Q \text{ b } b')$) tllist-all2
unfolding tllist.pcr-cr-eq
apply(rule rel-funI)+
apply(clarsimp 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* \Rightarrow *('b \Rightarrow ('a, 'c) tllist) \Rightarrow ('a, 'c) tllist
is $\lambda(xs, b) f. \text{apfst } (\text{lappend } xs) (f b)$
by(*auto simp add: split-def lappend-inf*)*

lift-definition *lappendt* :: *'a llist* \Rightarrow *('a, 'b) tllist* \Rightarrow *('a, 'b) tllist*
is *apfst* \circ *lappend*
by(*simp add: split-def*)

lift-definition *tfilter* :: *'b \Rightarrow ('a \Rightarrow bool) \Rightarrow ('a, 'b) tllist \Rightarrow ('a, 'b) tllist*
is $\lambda b P (xs, b'). (\text{lfilter } P \text{ } xs, \text{ if } \text{lfinite } xs \text{ then } b' \text{ else } b)$
by(*simp add: split-beta*)

lift-definition *tconcat* :: *'b \Rightarrow ('a llist, 'b) tllist \Rightarrow ('a, 'b) tllist*
is $\lambda b (xss, b'). (\text{lconcat } xss, \text{ if } \text{lfinite } xss \text{ then } b' \text{ else } b)$
by(*simp add: split-beta*)

lift-definition *tnth* :: *('a, 'b) tllist \Rightarrow nat \Rightarrow 'a*
is *lnth* \circ *fst* **by**(*auto*)

lift-definition *tlength* :: *('a, 'b) tllist \Rightarrow enat*
is *llength* \circ *fst* **by** *auto*

lift-definition *tdropn* :: *nat \Rightarrow ('a, 'b) tllist \Rightarrow ('a, 'b) tllist*
is *apfst* \circ *ldropn* **by** *auto*

abbreviation *tfinite* :: *('a, 'b) tllist \Rightarrow bool*
where *tfinite* *xs* \equiv *lfinite (llist-of-tllist xs)*

5.6 *tfinite*

lemma *tfinite-induct* [*consumes 1, case-names TNil TCons*]:

assumes *tfinite* *xs*
and $\bigwedge y. P (TNil y)$
and $\bigwedge x xs. \llbracket \text{tfinite } xs; P \text{ } xs \rrbracket \Longrightarrow P (TCons x xs)$
shows *P* *xs*

using *assms*

by *transfer (clarsimp, erule lfinite.induct)*

lemma *is-TNil-tfinite* [*simp*]: *is-TNil* *xs* \Longrightarrow *tfinite* *xs*
by *transfer clarsimp*

5.7 The terminal element *terminal*

lemma *terminal-tinfinite*:

assumes \neg *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 $terminal-transfer\ [transfer-rule]$:
 $(pcr-tllist\ A\ op = ==> op =) (\lambda(xs, b). if\ lfinite\ xs\ then\ b\ else\ undefined)$
 $terminal$
by($force\ simp\ add: cr-tllist-def\ pcr-tllist-def\ terminal-tllist-of-llist\ dest: llist-all2-lfiniteD$)

lemma $terminal-tmap\ [simp]$: $tfinite\ xs \implies terminal\ (tmap\ f\ g\ xs) = g\ (terminal\ xs)$
by($induct\ rule: tfinite-induct$) $simp-all$

5.8 $tmap$

lemma $tmap-eq-TCons-conv$:
 $tmap\ f\ g\ xs = TCons\ y\ ys \longleftrightarrow$
 $(\exists z\ zs. xs = TCons\ z\ zs \wedge f\ z = y \wedge tmap\ f\ g\ zs = ys)$
by($cases\ xs$) $simp-all$

lemma $TCons-eq-tmap-conv$:
 $TCons\ y\ ys = tmap\ f\ g\ xs \longleftrightarrow$
 $(\exists z\ zs. xs = TCons\ z\ zs \wedge f\ z = y \wedge tmap\ f\ g\ zs = ys)$
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 $tappend-assoc$: $tappend\ (tappend\ xs\ f)\ g = tappend\ xs\ (\lambda b. tappend\ (f\ b)\ g)$
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** *clarsimp*

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 *tllist-monad = tappend-TNil tappend-TNil2 tappend-assoc*

5.10 Appending a terminated lazy list to a lazy list *lappendt*

lemma *lappendt-LNil* [*simp, code, nitpick-simp*]: *lappendt LNil tr = tr*

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 *tllist-of-llist-eq-lappendt-conv*:

tllist-of-llist a xs = lappendt ys zs \longleftrightarrow

($\exists xs' a'. xs = lappend ys xs' \wedge zs = tllist-of-llist a' xs' \wedge (lfinite ys \longrightarrow a = a')$)

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 ($\forall x \in \text{tset } xs. \neg P x$)

by *transfer auto*

lemma *tfilter-empty-conv*:

tfilter *y P xs* = *TNil* *y'* \longleftrightarrow ($\forall x \in \text{tset } xs. \neg P x$) \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* \implies

$\exists us vs. ys = \text{lappendt } us (\text{TCons } x vs) \wedge \text{lfinite } us \wedge (\forall u \in \text{lset } us. \neg P u) \wedge P x \wedge xs = \text{tfilter } a P vs$

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 = ($\lambda b. \text{tfilter}' (\lambda-. b)$)

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* = ($\lambda b. \text{tconcat}' (\lambda-. b)$)

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*: $tllist-all2 P Q (TNil b) ts \longleftrightarrow (\exists b'. ts = TNil b' \wedge Q b b')$

by *transfer auto*

lemma *tllist-all2-TNil2*: $tllist-all2 P Q ts (TNil b') \longleftrightarrow (\exists b. ts = TNil b \wedge Q b b')$

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 tllist-all2 is-TNil TNil TCons*, *coinduct pred: tllist-all2*]:

assumes $X xs ys$

and $\bigwedge xs ys. X xs ys \implies$

$(is-TNil xs \longleftrightarrow is-TNil ys) \wedge$

$(is-TNil xs \longrightarrow is-TNil ys \longrightarrow R (terminal xs) (terminal ys)) \wedge$

$(\neg is-TNil xs \longrightarrow \neg is-TNil ys \longrightarrow P (thd xs) (thd ys) \wedge (X (ttl xs) (ttl ys) \vee$

$tllist-all2 P R (ttl xs) (ttl ys)))$

shows $tllist-all2 P R xs ys$

using *assms*

apply(*transfer fixing: P R*)

apply *clarsimp*

apply(*rule conjI*)

apply(*erule llist-all2-coinduct, blast, blast*)

apply (*rule impI*)

subgoal premises *prems* **for** $X xs b ys c$

proof –

from $\langle \text{lfinite } xs \rangle \langle X (xs, b) (ys, c) \rangle$
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 $(\text{TNil}) b b'$ **where** $xs = \text{TNil } b \text{ } ys = \text{TNil } b' \text{ } Q b b'$
 $| (\text{TCons}) x xs' y ys'$
where $xs = \text{TCons } x \text{ } xs' \text{ and } ys = \text{TCons } y \text{ } ys'$
and $P x y \text{ and } \text{tllist-all2 } P \text{ } Q \text{ } xs' \text{ } ys'$
using *assms*
by(*cases xs*)(*fastforce simp add: tllist-all2-TCons1 tllist-all2-TNil1*)+

lemma *tllist-all2-tmap1*:
 $\text{tllist-all2 } P \text{ } Q (tmap f g xs) ys \longleftrightarrow \text{tllist-all2 } (\lambda x. P (f x)) (\lambda x. Q (g x)) xs ys$
by(*transfer*)(*auto simp add: llist-all2-lmap1*)

lemma *tllist-all2-tmap2*:
 $\text{tllist-all2 } P \text{ } Q xs (tmap f g ys) \longleftrightarrow \text{tllist-all2 } (\lambda x y. P x (f y)) (\lambda x y. Q x (g y)) xs ys$
by(*transfer*)(*auto simp add: llist-all2-lmap2*)

lemma *tllist-all2-mono*:
 $\llbracket \text{tllist-all2 } P \text{ } Q \text{ } xs \text{ } ys; \bigwedge x y. P x y \implies P' x y; \bigwedge x y. Q x y \implies Q' x y \rrbracket$
 $\implies \text{tllist-all2 } P' \text{ } Q' \text{ } xs \text{ } ys$
by *transfer*(*auto elim!: llist-all2-mono*)

lemma *tllist-all2-tlengthD*: $\text{tllist-all2 } P \text{ } Q \text{ } xs \text{ } ys \implies \text{tlength } xs = \text{tlength } ys$
by(*transfer*)(*auto dest: llist-all2-llengthD*)

lemma *tllist-all2-tfiniteD*: $\text{tllist-all2 } P \text{ } Q \text{ } xs \text{ } ys \implies \text{tfinite } xs = \text{tfinite } ys$
by(*transfer*)(*auto dest: llist-all2-lfiniteD*)

lemma *tllist-all2-tfinite1-terminalD*:
 $\llbracket \text{tllist-all2 } P \text{ } Q \text{ } xs \text{ } ys; \text{tfinite } xs \rrbracket \implies Q (\text{terminal } xs) (\text{terminal } ys)$
by(*frule tllist-all2-tfiniteD*)(*transfer, auto*)

lemma *tllist-all2-tfinite2-terminalD*:
 $\llbracket \text{tllist-all2 } P \text{ } Q \text{ } xs \text{ } ys; \text{tfinite } ys \rrbracket \implies Q (\text{terminal } xs) (\text{terminal } ys)$
by(*metis tllist-all2-tfinite1-terminalD tllist-all2-tfiniteD*)

lemma *tllist-all2D-llist-all2-llist-of-tllist*:
 $\text{tllist-all2 } P \text{ } Q \text{ } xs \text{ } ys \implies \text{llist-all2 } P (\text{llist-of-tllist } xs) (\text{llist-of-tllist } ys)$
by(*transfer*) *auto*

lemma *tllist-all2-is-TNilD*:
 $\text{tllist-all2 } P \text{ } Q \text{ } xs \text{ } ys \implies \text{is-TNil } xs \longleftrightarrow \text{is-TNil } ys$

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: tllist-all2-conv-all-tnth split: if-split-asm dest: lfinite-llength-enat not-lfinite-llength)

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(clarsimp simp add: lmap-eq-lmap-conv-tllist-all2)

apply(auto dest: tllist-all2-lfiniteD)

done

lemma *tllist-all2-trans*:

$\llbracket \text{tllist-all2 } P \ Q \ xs \ ys; \text{tllist-all2 } P \ Q \ ys \ zs; \text{transp } P; \text{transp } Q \rrbracket$
 $\implies \text{tllist-all2 } P \ Q \ xs \ zs$

by *transfer*(*auto elim: llist-all2-trans dest: llist-all2-lfiniteD transpD*)

lemma *tllist-all2-tappendI*:

$\llbracket \text{tllist-all2 } P \ Q \ xs \ ys;$
 $\llbracket \text{tfinite } xs; \text{tfinite } ys; Q \ (\text{terminal } xs) \ (\text{terminal } ys) \rrbracket$
 $\implies \text{tllist-all2 } P \ R \ (xs' \ (\text{terminal } xs)) \ (ys' \ (\text{terminal } ys)) \rrbracket$
 $\implies \text{tllist-all2 } P \ R \ (\text{tappend } xs \ xs') \ (\text{tappend } ys \ ys')$

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*:

$\text{tllist-all2 } A \ B \ xs \ ys \implies \text{llist-all2 } A \ (\text{llist-of-tllist } xs) \ (\text{llist-of-tllist } ys)$

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]*:

$\text{tllist-all2 } A \ B \ (\text{tllist-of-llist } b \ xs) \ (\text{tllist-of-llist } c \ ys) \longleftrightarrow$
 $\text{llist-all2 } A \ xs \ ys \wedge (\text{lfinite } xs \longrightarrow B \ b \ c)$

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*:
 $llist\text{-of-tllist } xs = lappend\ us\ vs \longleftrightarrow$
 $(\exists ys. xs = lappendt\ us\ ys \wedge vs = llist\text{-of-tllist } ys \wedge terminal\ xs = terminal\ ys)$
by *transfer auto*

5.15 The n th element of a terminated lazy list $tnth$

lemma *tnth-TNil* [*nitpick-simp*]:
 $tnth\ (TNil\ b)\ n = undefined\ n$
by(*transfer*)(*simp add: lnth-LNil*)

lemma *tnth-TCons*:
 $tnth\ (TCons\ x\ xs)\ n = (case\ n\ of\ 0 \Rightarrow x \mid Suc\ n' \Rightarrow tnth\ xs\ n')$
by(*transfer*)(*auto simp add: lnth-LCons split: nat.split*)

lemma *tnth-code* [*simp, nitpick-simp, code*]:
shows *tnth-0*: $tnth\ (TCons\ x\ xs)\ 0 = x$
and *tnth-Suc-TCons*: $tnth\ (TCons\ x\ xs)\ (Suc\ n) = tnth\ xs\ n$
by(*simp-all add: tnth-TCons*)

lemma *lnth-llist-of-tllist* [*simp*]:
 $lnth\ (llist\text{-of-tllist } xs) = tnth\ xs$
by(*transfer*)(*auto*)

lemma *tnth-tmap* [*simp*]: $enat\ n < tlength\ xs \implies tnth\ (tmap\ f\ g\ xs)\ n = f\ (tnth\ xs\ n)$
by *transfer simp*

5.16 The length of a terminated lazy list $tlength$

lemma [*simp, nitpick-simp*]:
shows *tlength-TNil*: $tlength\ (TNil\ b) = 0$
and *tlength-TCons*: $tlength\ (TCons\ x\ xs) = eSuc\ (tlength\ xs)$
apply(*transfer, simp*)
apply(*transfer, auto*)
done

lemma *llength-llist-of-tllist* [*simp*]: $llength\ (llist\text{-of-tllist } xs) = tlength\ xs$
by *transfer auto*

lemma *tlength-tmap* [*simp*]: $tlength\ (tmap\ f\ g\ xs) = tlength\ xs$
by *transfer simp*

definition *gen-tlength* :: $nat \Rightarrow ('a, 'b)\ tlist \Rightarrow enat$
where *gen-tlength* $n\ xs = enat\ n + tlength\ xs$

lemma *gen-tlength-code* [*code*]:
 $gen\text{-tlength } n\ (TNil\ b) = enat\ n$
 $gen\text{-tlength } n\ (TCons\ x\ xs) = gen\text{-tlength } (n + 1)\ xs$
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-tl* *xs*)) (*tdropn* *n* *xs*) = *xs*
by *transfer (auto)*

lemma *l*list-of-tl *tdropn* [*simp*]:
*l*list-of-tl (*tdropn* *n* *xs*) = *ldropn* *n* (*l*list-of-tl *xs*)
by *transfer auto*

lemma *tdropn-Suc-conv-tdropn*:
enat *n* < *tlength* *xs* \implies *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* \implies *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*. \llbracket *x* \in *tset* *xs*; *x* \neq *x'*; *P* *xs* $\rrbracket \implies$ *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-lnth*)

lemma *in-tset-conv-tnth*: $x \in \text{tset } xs \iff (\exists n. \text{enat } n < \text{length } xs \wedge \text{tnth } xs \ n = x)$

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*]:

(*rel-pre-tllist* *A B C* ==> *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*]:

(*rel-pre-tllist* *A B C* ==> *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*]:

(*rel-pre-tllist* *A B C* ==> *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* ==> *tllist-all2* *A B*) *TNil* *TNil*

by *auto*

declare *TNil-transfer* [*transfer-rule*]

lemma *TCons-transfer2* [*transfer-rule*]:

(*A* ==> *tllist-all2* *A B* ==> *tllist-all2* *A B*) *TCons* *TCons*

unfolding *rel-fun-def* by *simp*

declare *TCons-transfer* [*transfer-rule*]

lemma *case-tllist-transfer* [*transfer-rule*]:

((*B* ==> *C*) ==> (*A* ==> *tllist-all2* *A B* ==> *C*) ==> *tllist-all2* *A B* ==> *C*)

case-tllist *case-tllist*

unfolding *rel-fun-def*

by (*simp add: tllist-all2-TNil1 tllist-all2-TNil2 split: tllist.split*)

lemma *unfold-tllist-transfer* [*transfer-rule*]:

```

  ((A ==> op =) ==> (A ==> B) ==> (A ==> C) ==> (A ==>
A) ==> A ==> 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 ==> op =) 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 ==> op =) ==> (A ==> B) ==> (A ==> C) ==> (A ==>
op =) ==> (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 ==> op =) IS-TNIL1 IS-TNIL2 (A ==> B) TERMINAL1
TERMINAL2
    (A ==> C) THD1 THD2 (A ==> op =) 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 tlist-all2-tmap1 tlist-all2-tmap2 elim: tlist-all2-mono*)
declare *tmap-transfer* [*transfer-rule*]

lemma *is-TNil-transfer2* [*transfer-rule*]:
(tlist-all2 A B ==> op =) is-TNil is-TNil
by(*auto dest: tlist-all2-is-TNilD*)
declare *is-TNil-transfer* [*transfer-rule*]

lemma *tappend-transfer* [*transfer-rule*]:
(tlist-all2 A B ==> (B ==> tlist-all2 A C) ==> tlist-all2 A C) tappend
tappend
by(*auto intro: tlist-all2-tappendI elim: rel-funE*)
declare *tappend.transfer* [*transfer-rule*]

lemma *lappendt-transfer* [*transfer-rule*]:
(llist-all2 A ==> tlist-all2 A B ==> tlist-all2 A B) lappendt lappendt
unfolding *rel-fun-def*
by *transfer(auto intro: llist-all2-lappendI)*
declare *lappendt.transfer* [*transfer-rule*]

lemma *llist-of-tlist-transfer2* [*transfer-rule*]:
(tlist-all2 A B ==> llist-all2 A) llist-of-tlist llist-of-tlist
by(*auto intro: llist-all2-tlist-of-llistI*)
declare *llist-of-tlist-transfer* [*transfer-rule*]

lemma *tlist-of-llist-transfer2* [*transfer-rule*]:
(B ==> llist-all2 A ==> tlist-all2 A B) tlist-of-llist tlist-of-llist
by(*auto intro!: rel-funI*)
declare *tlist-of-llist-transfer* [*transfer-rule*]

lemma *tlength-transfer* [*transfer-rule*]:
(tlist-all2 A B ==> op =) tlength tlength
by(*auto dest: tlist-all2-tlengthD*)
declare *tlength.transfer* [*transfer-rule*]

lemma *tdropn-transfer* [*transfer-rule*]:
(op = ==> tlist-all2 A B ==> tlist-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 ==> (A ==> op =) ==> tlist-all2 A B ==> tlist-all2 A B) tfilter
tfilter
unfolding *rel-fun-def*
by *transfer(auto intro: llist-all2-tfilterI dest: llist-all2-lfiniteD)*
declare *tfilter.transfer* [*transfer-rule*]

lemma *tconcat-transfer* [*transfer-rule*]:

```

  (B ===> tllist-all2 (llist-all2 A) B ===> tllist-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:  $\forall x y. R1\ x\ y \longrightarrow (\forall a b. R1\ a\ b \longrightarrow S\ x\ a = T\ y\ b)$ 
  and R2:  $\forall x y. R2\ x\ y \longrightarrow (\forall a b. R2\ a\ b \longrightarrow S'\ x\ a = T'\ y\ b)$ 
  and xsys: tllist-all2 R1 R2 xs ys
  and xs'ys': tllist-all2 R1 R2 xs' ys'
  shows tllist-all2 S S' xs xs' = tllist-all2 T T' ys ys'
proof
  assume tllist-all2 S S' xs xs'
  with xsys 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 xsys 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 ===> op =) ===> (R2 ===> R2 ===> op =) ===>
  tllist-all2 R1 R2 ===> tllist-all2 R1 R2 ===> op =) 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*: $\llbracket \text{transp } R; R \ a \ b; R \ b \ c \rrbracket \implies R \ a \ c$
by (*erule transpE*) *blast*

lemma *id-respect* [*quot-respect*]:
 $(R \implies R) \ id \ id$
by (*fact id-rsp*)

lemma *id-preserve* [*quot-preserve*]:
assumes *Quotient3 R Abs Rep*
shows $(Rep \dashrightarrow Abs) \ id = id$
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 *lmap-id0 [id-simps]*

lemma *reflp-llist-all2*: $reflp \ R \implies reflp \ (llist-all2 \ R)$
by (*rule reflpI*) (*auto simp add: llist-all2-conv-all-lnth elim: reflpE*)

lemma *symp-llist-all2*: $symp \ R \implies symp \ (llist-all2 \ R)$
by (*rule sympI*) (*auto simp add: llist-all2-conv-all-lnth elim: sympE*)

lemma *transp-llist-all2*: $transp \ R \implies transp \ (llist-all2 \ R)$
by (*rule transpI*) (*rule llist-all2-trans*)

lemma *llist-equiv* [*quot-equiv*]:
 $equivp \ R \implies equivp \ (llist-all2 \ R)$
by (*simp add: equivp-reflp-symp-transp reflp-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 $((Abs1 \dashrightarrow id) \dashrightarrow (Abs1 \dashrightarrow Rep2) \dashrightarrow (Abs1 \dashrightarrow Rep1) \dashrightarrow Rep1 \dashrightarrow lmap \ Abs2) \ unfold-llist = unfold-llist$
(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] Quotient3-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-reflI)
  apply(clarsimp 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-reflI)
  apply(clarsimp 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(clarsimp 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*]:
 $Quotient3\ R\ Abs\ Rep \implies Quotient3\ (l\text{list-all2}\ R)\ (l\text{map}\ Abs)\ (l\text{map}\ Rep)$
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*]:
 $l\text{map}\ Abs\ LNil = LNil$
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* ==== > *op* =) ==== > (*llist-all2* *R1*) ==== > *op* =) *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 $q1: \text{Quotient3 } R1 \text{ Abs1 Rep1}$

and $q2: \text{Quotient3 } R2 \text{ Abs2 Rep2}$

shows $((\text{Abs1} \text{ ----} \rightarrow \text{id}) \text{ ----} \rightarrow (\text{Abs1} \text{ ----} \rightarrow \text{Rep2}) \text{ ----} \rightarrow (\text{Abs1} \text{ ----} \rightarrow \text{id}) \text{ ----} \rightarrow$

$(\text{Abs1} \text{ ----} \rightarrow \text{lmap Rep2}) \text{ ----} \rightarrow (\text{Abs1} \text{ ----} \rightarrow \text{Rep1}) \text{ ----} \rightarrow \text{Rep1}$

$\text{----} \rightarrow \text{lmap Abs2}) \text{ corec-llist} = \text{corec-llist}$

(is ?lhs = ?rhs)

proof(intro ext)

fix $IS\text{-LNIL } LHD \text{ endORmore } LTL\text{-end } LTL\text{-more } b$

show $?lhs \text{ IS-LNIL } LHD \text{ endORmore } LTL\text{-end } LTL\text{-more } b = ?rhs \text{ IS-LNIL } LHD \text{ endORmore } LTL\text{-end } LTL\text{-more } b$

by (coinduction arbitrary: b rule: llist.coinduct-strong)

(auto simp add: Quotient3-abs-rep[OF q1] Quotient3-abs-rep[OF q2] Quotient-lmap-Abs-Rep[OF q2])

qed

end

7 Setup for Isabelle's quotient package for terminated lazy lists

theory Quotient-TLList imports

TLList

HOL-Library.Quotient-Product

HOL-Library.Quotient-Sum

HOL-Library.Quotient-Set

begin

7.1 Rules for the Quotient package

lemma tmap-id-id [id-simps]:

tmap id id = id

by (simp add: fun-eq-iff tllist.map-id)

declare tllist-all2-eq[id-simps]

lemma case-sum-preserve [quot-preserve]:

assumes $q1: \text{Quotient3 } R1 \text{ Abs1 Rep1}$

and $q2: \text{Quotient3 } R2 \text{ Abs2 Rep2}$

and $q3: \text{Quotient3 } R3 \text{ Abs3 Rep3}$

shows $((\text{Abs1} \text{ ----} \rightarrow \text{Rep2}) \text{ ----} \rightarrow (\text{Abs3} \text{ ----} \rightarrow \text{Rep2}) \text{ ----} \rightarrow \text{map-sum Rep1}$

$\text{Rep3} \text{ ----} \rightarrow \text{Abs2}) \text{ case-sum} = \text{case-sum}$

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:  $\llbracket \text{symp } R; \text{symp } S \rrbracket \implies \text{symp } (\text{tllist-all2 } R \ S)$ 
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:  $\llbracket \text{transp } R; \text{transp } S \rrbracket \implies \text{transp } (\text{tllist-all2 } R \ S)$ 

```

```

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])

```

```

done

from ⟨?lhs⟩ show tllist-all2 R1 R2 s s
  apply –
  apply(rule tllist-all2-reflI)
  apply(clarsimp 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(clarsimp 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 Quotient-llist[quot-map]:
  assumes Quotient R1 Abs1 Rep1 T1
  and Quotient R2 Abs2 Rep2 T2
  shows Quotient (tllist-all2 R1 R2) (tmap Abs1 Abs2) (tmap Rep1 Rep2) (tllist-all2
T1 T2)
unfolding Quotient-alt-def
proof(intro conjI strip)
  from assms have 1:  $\bigwedge x y. T1\ x\ y \implies Abs1\ x = y$ 
  and 2:  $\bigwedge x y. T2\ x\ y \implies Abs2\ x = y$ 
  unfolding Quotient-alt-def by simp-all
  fix xs ys
  assume tllist-all2 T1 T2 xs ys
  thus tmap Abs1 Abs2 xs = ys
  by(coinduction arbitrary: xs ys)(auto simp add: 1 2 dest: tllist-all2-is-TNilD
tllist-all2-tfinite1-terminalD tllist-all2-thdD intro: tllist-all2-ttI)
next
  from assms have 1:  $\bigwedge x. T1\ (Rep1\ x)\ x$ 

```

```

    and 2:  $\bigwedge x. T2 (Rep2 x) x$ 
    unfolding Quotient-alt-def by simp-all
  fix xs
  show tllist-all2 T1 T2 (tmap Rep1 Rep2 xs) xs
    by(simp add: tllist-all2-tmap1 1 2 tllist-all2-refl)
next
  from assms have 1:  $R1 = (\lambda x y. T1 x (Abs1 x) \wedge T1 y (Abs1 y) \wedge Abs1 x = Abs1 y)$ 
  and 2:  $R2 = (\lambda x y. T2 x (Abs2 x) \wedge T2 y (Abs2 y) \wedge Abs2 x = Abs2 y)$ 
  unfolding Quotient-alt-def by(simp-all add: fun-eq-iff)
  fix xs ys
  show tllist-all2 R1 R2 xs ys
     $\longleftrightarrow$  tllist-all2 T1 T2 xs (tmap Abs1 Abs2 xs)  $\wedge$ 
    tllist-all2 T1 T2 ys (tmap Abs1 Abs2 ys)  $\wedge$ 
    tmap Abs1 Abs2 xs = tmap Abs1 Abs2 ys
  unfolding 1 2 tmap-eq-tmap-conv-tllist-all2
  by(auto 4 3 simp add: tllist-all2-conv-all-tnth dest: lfinite-llength-enat not-lfinite-llength)
qed

```

```

lemma TCons-preserve [quot-preserve]:
  assumes q1: Quotient3 R1 Abs1 Rep1
  and q2: Quotient3 R2 Abs2 Rep2
  shows (Rep1  $\dashrightarrow$  (tmap Rep1 Rep2)  $\dashrightarrow$  (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  $\dashrightarrow$  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  $\dashrightarrow$  Abs1  $\dashrightarrow$  id) P) ((Abs2  $\dashrightarrow$  Abs2  $\dashrightarrow$ 
  id) Q)
    (tmap Rep1 Rep2 ts) (tmap Rep1 Rep2 ts')
     $\longleftrightarrow$  tllist-all2 P Q ts ts'
  (is ?lhs  $\longleftrightarrow$  ?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-ttll)
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)) = (op =)
  by (simp add: fun-eq-iff map-fun-def comp-def Quotient3-rel-rep[OF q1] Quotient3-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: Quotient3-abs-rep[OF
q1] Quotient3-abs-rep[OF q2] Quotient3-abs-rep[OF q3] Quotient3-tmap-Abs-Rep[OF
q3 q2])
qed

```

```

end

theory Coinductive imports
  Coinductive-List-Prefix
  Coinductive-Stream
  TLList
  Quotient-Coinductive-List
  Quotient-TLList
begin

end

```

8 Code generator setup to implement lazy lists lazily

```

theory Lazy-LList imports
  Coinductive-List
begin

```

8.1 Lazy lists

```

code-identifier code-module Lazy-LList  $\rightarrow$ 
  (SML) Coinductive-List and
  (OCaml) Coinductive-List and
  (Haskell) Coinductive-List and
  (Scala) Coinductive-List

```

```

definition Lazy-llist :: (unit  $\Rightarrow$  ('a  $\times$  'a llist) option)  $\Rightarrow$  'a llist
where [simp]:
  Lazy-llist xs = (case xs () of None  $\Rightarrow$  LNil | Some (x, ys)  $\Rightarrow$  LCons x ys)

```

```

definition force :: 'a llist  $\Rightarrow$  ('a  $\times$  'a llist) option
where [simp, code del]: force xs = (case xs of LNil  $\Rightarrow$  None | LCons x ys  $\Rightarrow$  Some
(x, ys))

```

```

code-datatype Lazy-llist

```

```

declare — Restore consistency in code equations between partial-term-of and narrowing for 'a llist
  [[code drop: partial-term-of :: - llist itself => -]]

```

```

lemma partial-term-of-llist-code [code]:
  fixes tytok :: 'a :: partial-term-of llist itself shows
  partial-term-of tytok (Quickcheck-Narrowing.Narrowing-variable p tt)  $\equiv$ 
    Code-Evaluation.Free (STR "-") (Typerep.typerep TYPE('a llist))
  partial-term-of tytok (Quickcheck-Narrowing.Narrowing-constructor 0 [])  $\equiv$ 
    Code-Evaluation.Const (STR "Coinductive-List.llist.LNil") (Typerep.typerep
TYPE('a llist))

```

```

partial-term-of tytok (Quickcheck-Narrowing.Narrowing-constructor 1 [head, tail])
≡
Code-Evaluation.App
  (Code-Evaluation.App
    (Code-Evaluation.Const
      (STR "Coinductive-List.llist.LCons'")
      (Typerep.typerep TYPE('a ⇒ 'a llist ⇒ 'a llist)))
    (partial-term-of TYPE('a) head))
  (partial-term-of TYPE('a llist) tail)
by-(rule partial-term-of-anything)+

declare option.splits [split]

lemma Lazy-llist-inject [simp]:
  Lazy-llist xs = Lazy-llist ys ↔ xs = ys
by(auto simp add: fun-eq-iff)

lemma Lazy-llist-inverse [code, simp]:
  force (Lazy-llist xs) = xs ()
by(auto)

lemma force-inverse [simp]:
  Lazy-llist (λ-. force xs) = xs
by(auto split: llist.split)

lemma LNil-Lazy-llist [code]: LNil = Lazy-llist (λ-. None)
by(simp)

lemma LCons-Lazy-llist [code, code-unfold]: LCons x xs = Lazy-llist (λ-. Some
(x, xs))
by simp

lemma lnull-lazy [code]: lnull = Option.is-none ∘ 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 ⇒ -]]

lemma equal-llist-Lazy-llist [code]:
  equal-class.equal (Lazy-llist xs) (Lazy-llist ys) ↔
  (case xs () of None ⇒ (case ys () of None ⇒ True | - ⇒ False)
  | Some (x, xs') ⇒
    (case ys () of None ⇒ False
    | Some (y, ys') ⇒ if x = y then equal-class.equal xs' ys' else False))
by(auto simp add: equal-llist-def)

declare [[code drop: corec-llist]]

lemma corec-llist-Lazy-llist [code]:

```

```

    corec-llist IS-LNIL LHD endORmore LTL-end LTL-more b =
    Lazy-llist (λ-. if IS-LNIL b then None
    else Some (LHD b,
    if endORmore b then LTL-end b
    else corec-llist IS-LNIL LHD endORmore LTL-end LTL-more (LTL-more b)))
by(subst llist.corec-code) simp

declare [[code drop: unfold-llist]]

lemma unfold-llist-Lazy-llist [code]:
    unfold-llist IS-LNIL LHD LTL b =
    Lazy-llist (λ-. if IS-LNIL b then None else Some (LHD b, unfold-llist IS-LNIL
    LHD LTL (LTL b)))
by(subst unfold-llist.code) simp

declare [[code drop: case-llist]]

lemma case-llist-Lazy-llist [code]:
    case-llist n c (Lazy-llist xs) = (case xs () of None ⇒ n | Some (x, ys) ⇒ c x ys)
by simp

declare [[code drop: lappend]]

lemma lappend-Lazy-llist [code]:
    lappend (Lazy-llist xs) ys =
    Lazy-llist (λ-. case xs () of None ⇒ force ys | Some (x, xs') ⇒ Some (x, lappend
    xs' ys))
by(auto split: llist.splits)

declare [[code drop: lmap]]

lemma lmap-Lazy-llist [code]:
    lmap f (Lazy-llist xs) = Lazy-llist (λ-. map-option (map-prod f (lmap f)) (xs ()))
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-llist [code]:
  lzip (Lazy-llist xs) (Lazy-llist ys) =
  Lazy-llist (λ-. Option.bind (xs ()) (λ(x, xs'). map-option (λ(y, ys'). ((x, y), lzip
xs' ys')) (ys ())))
by auto

declare [[code drop: gen-lset]]

```

```

lemma lset-Lazy-llist [code]:
  gen-lset A (Lazy-llist xs) =
    (case xs () of None  $\Rightarrow$  A | Some (y, ys)  $\Rightarrow$  gen-lset (insert y A) ys)
by(auto simp add: gen-lset-code)

declare [[code drop: lmember]]

lemma lmember-Lazy-llist [code]:
  lmember x (Lazy-llist xs) =
    (case xs () of None  $\Rightarrow$  False | Some (y, ys)  $\Rightarrow$  x = y  $\vee$  lmember x ys)
by(simp add: lmember-def)

declare [[code drop: lalist-all2]]

lemma lalist-all2-Lazy-llist [code]:
  lalist-all2 P (Lazy-llist xs) (Lazy-llist ys) =
    (case xs () of None  $\Rightarrow$  ys () = None
      | Some (x, xs')  $\Rightarrow$  (case ys () of None  $\Rightarrow$  False
        | Some (y, ys')  $\Rightarrow$  P x y  $\wedge$  lalist-all2 P xs' ys'))
by auto

declare [[code drop: lhd]]

lemma lhd-Lazy-llist [code]:
  lhd (Lazy-llist xs) = (case xs () of None  $\Rightarrow$  undefined | Some (x, xs')  $\Rightarrow$  x)
by(simp add: lhd-def)

declare [[code drop: ltl]]

lemma ltl-Lazy-llist [code]:
  ltl (Lazy-llist xs) = Lazy-llist ( $\lambda$ -. case xs () of None  $\Rightarrow$  None | Some (x, ys)  $\Rightarrow$ 
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 ldistinct-Lazy-llist [code]:
  ldistinct (Lazy-llist xs) =
    (case xs () of None  $\Rightarrow$  True | Some (x, ys)  $\Rightarrow$  x  $\notin$  lset ys  $\wedge$  ldistinct 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 \Rightarrow *True*
 | *Some* (*x*, *xs'*) \Rightarrow
 (*case ys* () of *None* \Rightarrow *False* | *Some* (*y*, *ys'*) \Rightarrow *r* *x* *y* \vee *x* = *y* \wedge *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* \Rightarrow *None*
 | *Some* (*x*, *ys*) \Rightarrow 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 ( $\lambda$ -. case xss () of None  $\Rightarrow$  None | Some (xs, xss')  $\Rightarrow$  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-TLList imports
  TLList
  Lazy-LList
begin

code-identifier code-module Lazy-TLList  $\rightarrow$ 
  (SML) TLList and
  (OCaml) TLList and
  (Haskell) TLList and
  (Scala) TLList

```


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* "TLList.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* "TLList.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* (λ -. *Inr* *b*)

by *simp*

lemma *TCons-Lazy-tllist* [code, code-unfold]:

TCons *x* *xs* = *Lazy-tllist* (λ -. *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 ⇒ -]]`

lemma `equal-tllist-Lazy-tllist [code]:`

`equal-class.equal (Lazy-tllist xs) (Lazy-tllist ys) =`
`(case xs () of`
`Inr b ⇒ (case ys () of Inr b' ⇒ b = b' | - ⇒ False)`
`| Inl (x, xs') ⇒`
`(case ys () of Inr b' ⇒ False | Inl (y, ys') ⇒ if x = y then equal-class.equal`
`xs' ys' else False))`
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) ⟷`
`(case xs () of Inl - ⇒ False | Inr - ⇒ 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`
`(λ-. if IS-TNIL b then Inr (TNIL b)`
`else Inl (THD b, if endORMore b then TTL-end b else corec-tllist IS-TNIL`
`TNIL THD endORMore TTL-end TTL-more (TTL-more b)))`
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 ( $\lambda$ -. case xs () of None  $\Rightarrow$  Inr b | Some (x, ys)  $\Rightarrow$  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)  $\Rightarrow$  terminal ys | Inr b  $\Rightarrow$  b)
by simp

declare [[code drop: tmap]]

lemma tmap-Lazy-tllist [code]:
  tmap f g (Lazy-tllist xs) =
    Lazy-tllist ( $\lambda$ -. case xs () of Inl (x, ys)  $\Rightarrow$  Inl (f x, tmap f g ys) | Inr b  $\Rightarrow$  Inr (g
    b))
by simp

declare [[code drop: tappend]]

lemma tappend-Lazy-tllist [code]:
  tappend (Lazy-tllist xs) ys =
    Lazy-tllist ( $\lambda$ -. case xs () of Inl (x, xs')  $\Rightarrow$  Inl (x, tappend xs' ys) | Inr b  $\Rightarrow$  force
    (ys b))
by(auto split: tllist.split)

declare [[code drop: lappendt]]

lemma lappendt-Lazy-llist [code]:
  lappendt (Lazy-llist xs) ys =
    Lazy-tllist ( $\lambda$ -. case xs () of None  $\Rightarrow$  force ys | Some (x, xs')  $\Rightarrow$  Inl (x, lappendt
    xs' ys))
by(auto simp add: Lazy-llist-def split: option.split tllist.split)

declare [[code drop: TLList.tfilter']]

lemma tfilter'-Lazy-tllist [code]:
  TLList.tfilter' b P (Lazy-tllist xs) =
    Lazy-tllist ( $\lambda$ -. case xs () of Inl (x, xs')  $\Rightarrow$  if P x then Inl (x, TLList.tfilter' b P
    xs') else force (TLList.tfilter' b P xs') | Inr b'  $\Rightarrow$  Inr b')
by(simp split: tllist.split)

declare [[code drop: TLList.tconcat']]

lemma tconcat-Lazy-tllist [code]:

```

$TLList.tconcat' b (Lazy-tl\text{list } xss) =$
 $Lazy-tl\text{list } (\lambda-. \text{case } xss () \text{ of } Inr b' \Rightarrow Inr b' \mid Inl (x, xss') \Rightarrow \text{force } (lappendt\ xs$
 $(TLList.tconcat' b\ xss'))$
 $\text{by}(simp\ split: tl\text{list.split})$

declare $[[code\ drop: tl\text{list-all2}]]$

lemma $tl\text{list-all2-Lazy-tl\text{list}}$ $[code]:$

$tl\text{list-all2 } P\ Q\ (Lazy-tl\text{list } xs)\ (Lazy-tl\text{list } ys) \longleftrightarrow$
 $(\text{case } xs () \text{ of}$
 $\quad Inr b \Rightarrow (\text{case } ys () \text{ of } Inr b' \Rightarrow Q\ b\ b' \mid Inl - \Rightarrow False)$
 $\quad \mid Inl (x, xs') \Rightarrow (\text{case } ys () \text{ of } Inr - \Rightarrow False \mid Inl (y, ys') \Rightarrow P\ x\ y \wedge tl\text{list-all2}$
 $\quad P\ Q\ xs'\ ys'))$
 $\text{by}(simp\ add: tl\text{list-all2-TNil1 } tl\text{list-all2-TNil2})$

declare $[[code\ drop: llist-of-tl\text{list}]]$

lemma $llist-of-tl\text{list-Lazy-tl\text{list}}$ $[code]:$

$llist-of-tl\text{list } (Lazy-tl\text{list } xs) =$
 $Lazy-llist (\lambda-. \text{case } xs () \text{ of } Inl (x, ys) \Rightarrow \text{Some } (x, llist-of-tl\text{list } ys) \mid Inr b \Rightarrow$
 $\text{None})$
 $\text{by}(simp\ add: Lazy-llist-def)$

declare $[[code\ drop: tnth]]$

lemma $tnth-Lazy-tl\text{list}$ $[code]:$

$tnth (Lazy-tl\text{list } xs) n =$
 $(\text{case } xs () \text{ of } Inr b \Rightarrow \text{undefined } n \mid Inl (x, ys) \Rightarrow \text{if } n = 0 \text{ then } x \text{ else } tnth\ ys$
 $(n - 1))$
 $\text{by}(cases\ n)(auto\ simp\ add: tnth-TNil)$

declare $[[code\ drop: gen-tlength]]$

lemma $gen-tlength-Lazy-tl\text{list}$ $[code]:$

$gen-tlength\ n\ (Lazy-tl\text{list } xs) =$
 $(\text{case } xs () \text{ of } Inr b \Rightarrow \text{enat } n \mid Inl (-, xs') \Rightarrow gen-tlength\ (n + 1)\ xs')$
 $\text{by}(simp\ add: gen-tlength-code)$

declare $[[code\ drop: tdropn]]$

lemma $tdropn-Lazy-tl\text{list}$ $[code]:$

$tdropn\ n\ (Lazy-tl\text{list } xs) =$
 $Lazy-tl\text{list } (\lambda-. \text{if } n = 0 \text{ then } xs () \text{ else } \text{case } xs () \text{ of } Inr b \Rightarrow Inr b \mid Inl (x, xs')$
 $\Rightarrow \text{force } (tdropn\ (n - 1)\ xs'))$
 $\text{by}(cases\ n)(auto\ split: tl\text{list.split})$

declare $Lazy-tl\text{list-def}$ $[simp\ del]$

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 $\forall x \in \text{set } xs. P x$ then dropWhile P ys else dropWhile P xs @ ys)

by *auto*

lemma *dropWhile-False:* $(\bigwedge x. x \in \text{set } xs \implies P x) \implies \text{dropWhile } P \text{ } xs = []$

by *simp*

abbreviation (**in order**) *chain* \equiv *Complete-Partial-Order.chain op \leq*

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 + 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 ≤ t⟩, auto)
qed

```

```

lemma tendsto-add-ereal:
  0 ≤ x ⇒ 0 ≤ y ⇒ (f ⟶ y) F ⇒ ((λz. x + f z :: ereal) ⟶ 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={- ∞ <..}])
  done

```

```

lemma tendsto-LimI: (f ⟶ y) F ⇒ (f ⟶ Lim F f) F
  by (metis tendsto-Lim tendsto-bot)

```

10.1 The filter at'

abbreviation (in *ccpo*) *compact-element* \equiv *ccpo.compact Sup op ≤*

```

lemma tendsto-unique-eventually:
  fixes x x' :: 'a :: t2-space
  shows F ≠ bot ⇒ eventually (λx. f x = g x) F ⇒ (f ⟶ x) F ⇒ (g
  ⟶ x') F ⇒ x = x'
  by (metis tendsto-unique filterlim-cong)

```

```

lemma (in cppo) cppo-Sup-upper2: chain C ⇒ x ∈ C ⇒ y ≤ x ⇒ y ≤ Sup C
  by (blast intro: cppo-Sup-upper order-trans)

```

```

lemma tendsto-open-vimage: (⋀B. open B ⇒ open (f -' B)) ⇒ f -l→ f l
  using continuous-on-open-vimage[of UNIV f] continuous-on-def[of UNIV f] by
  simp

```

```

lemma open-vimageI: (⋀x. f -x→ f x) ⇒ open A ⇒ 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 ⟷ 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'*: $(f \longrightarrow f x) (at' x) \iff f -x \rightarrow f x$
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 \ll *Sign.add-const-constraint*
 $(@{\text{const-name open}}, \text{SOME } @\{\text{typ 'a}::\text{open set} \Rightarrow \text{bool}\}) \gg$

class *ccpo-topology* = *open* + *ccpo* +
assumes *open-ccpo*: $\text{open } A \iff (\forall C. \text{chain } C \longrightarrow C \neq \{\} \longrightarrow \text{Sup } C \in A \longrightarrow C \cap A \neq \{\})$

begin

lemma *open-ccpoD*:
assumes *open A chain C C ≠ {} Sup C ∈ 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 $\langle \text{chain } C \rangle \langle C \neq \{\} \rangle$ **have** *chain (C - A) C - A ≠ {}*
by (*auto intro: chain-Diff*)
moreover have $\text{Sup } C = \text{Sup } (C - A)$
proof (*safe intro!: antisym ccpo-Sup-least <chain C> chain-Diff*)
fix *c* **assume** $c \in C$
with $*$ **obtain** *c'* **where** $c' \in C \ c \leq c' \ c' \notin A$
by *auto*
with $\langle c \in C \rangle$ **show** $c \leq \bigsqcup (C - A)$
by (*intro ccpo-Sup-upper2 <chain (C - A)> auto*)
qed (*auto intro: <chain C> ccpo-Sup-upper*)
ultimately show *False*
using $\langle \text{open } A \rangle \langle \text{Sup } C \in A \rangle$ **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 order-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 C: chain C C ≠ {} and  $\sqcup C \in S \cap T$ 
    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 *:  $\forall D \in K. \text{open } D$ 
show open ( $\bigcup K$ )
    unfolding open-ccpo
proof (intro allI impI)
    fix C assume chain C C ≠ {}  $\sqcup C \in (\bigcup K)$ 
    with * obtain D where D ∈ K  $\sqcup C \in D$  C ∩ D ≠ {}
        by (auto simp: open-ccpo)
    then show C ∩ ( $\bigcup K$ ) ≠ {}
        by auto
qed
qed

lemma closed-ccpo: closed A  $\longleftrightarrow (\forall C. \text{chain } C \longrightarrow C \neq \{\} \longrightarrow C \subseteq A \longrightarrow \text{Sup } C \in A)$ 
    unfolding closed-def open-ccpo by auto

lemma closed-admissible: closed {x. P x}  $\longleftrightarrow \text{ccpo.admissible } \text{Sup } \text{op } \leq P$ 
    unfolding closed-ccpo ccpo.admissible-def by auto

lemma open-singletonI-compact: compact-element x  $\implies \text{open } \{x\}$ 
    using admissible-compact-neq[of Sup op ≤ x]
    by (simp add: closed-admissible[symmetric] open-closed Collect-neq-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  $\exists U V. \text{open } U \wedge \text{open } V \wedge x \in U \wedge y \in V \wedge U \cap V = \{\}$ 
    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 (λx. False) F by eventually-elim (auto intro: order-trans)
  with F show False by (auto simp: eventually-False)
next
  assume ¬ x ≤ y
  with ⟨¬ y ≤ x⟩ 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)
  with topological-tendstoD[OF x, of {...x} ∩ − {...y}]
    topological-tendstoD[OF y, of {...y} ∩ − {...x}]
  have eventually (λz. f z ≤ x ∧ ¬ f z ≤ y) F eventually (λz. g z ≤ y ∧ ¬ g z
  ≤ x) F

```

```

    by auto
  with ev have eventually ( $\lambda x. False$ ) 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 ( $\bigwedge C. chain\ C \Longrightarrow C \neq \{\}$   $\Longrightarrow chain\ (f\ 'C) \wedge f\ (Sup\ C) = Sup\ (f\ 'C)$ )
 $\Longrightarrow f\ -x \rightarrow f\ x$ 
  by (intro tendsto-open-vimage) (auto simp: open-ccpo)

```

```

lemma tendsto-mcont:
  assumes mcont: mcont Sup op  $\leq Sup\ op \leq (f :: 'a :: ccpo-topology \Rightarrow 'b :: ccpo-topology)$ 
  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=op  $\leq$ ] 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  $\Rightarrow bool$  where
  open-set 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-set-def)

```

```

end

```

```

instantiation enat :: ccpo-topology
begin

```

```

instance
proof
  fix A :: enat set
  show open A = ( $\forall C. chain\ C \longrightarrow C \neq \{\} \longrightarrow \bigsqcup C \in A \longrightarrow C \cap A \neq \{\}$ )
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  $\langle \bigsqcup C \in A \rangle$  (open A) obtain  $n$  where  $\{ \text{enat } n < .. \} \subseteq A$ 
    unfolding open-enat-iff by auto
  with  $\langle \bigsqcup C = \infty \rangle$  Sup-eq-top-iff[of C] show ?thesis
    by (auto simp: top-enat-def)
next
  assume  $\bigsqcup C \neq \infty$ 
  then obtain  $n$  where  $C \subseteq \{ .. \text{enat } n \}$ 
    unfolding Sup-eq-top-iff top-enat-def[symmetric] by (auto simp: not-less top-enat-def)
  moreover have finite  $\{ .. \text{enat } n \}$ 
    by (auto intro: finite-enat-bounded)
  ultimately have finite  $C$ 
    by (auto intro: finite-subset)
  from in-chain-finite[OF  $\langle \text{chain } C \rangle$   $\langle \text{finite } C \rangle$   $\langle C \neq \{ \} \rangle$   $\langle \bigsqcup C \in A \rangle$  show
?thesis
    by auto
  qed
next
  assume  $C: \forall C. \text{chain } C \longrightarrow C \neq \{ \} \longrightarrow \bigsqcup C \in A \longrightarrow C \cap A \neq \{ \}$ 
  show open A
    unfolding open-enat-iff
  proof safe
    assume  $\infty \in A$ 
    { fix  $C :: \text{enat set}$  assume infinite C
      then have  $\bigsqcup C = \infty$ 
        by (auto simp: Sup-enat-def)
      with  $\langle \text{infinite } C \rangle$  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  $\neg (\exists x. \{ \text{enat } x < .. \} \subseteq A)$ 
    with  $\langle \infty \in A \rangle$  have  $\bigwedge x. \exists y > x. \text{enat } y \notin A$ 
      by (simp add: subset-eq Bex-def) (metis enat.exhaust enat-ord-simps(2))
    then have infinite  $\{ n. \text{enat } n \notin A \}$ 
      unfolding infinite-nat-iff-unbounded by auto
    then have infinite (enat ‘  $\{ n. \text{enat } n \notin A \}$ )
      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 ($\lambda x. x \sqcap y$) ($z :: - :: \{\text{ccpo-topology, complete-distrib-lattice}\}$)
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 $x \implies isCont$ ($\lambda x. \text{if } Q \text{ then } f \text{ else } g$
 x) 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 $g :: 'a :: t2\text{-space} \Rightarrow 'b::t2\text{-space}$

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 $A \longleftrightarrow (\forall C. \text{chain } C \longrightarrow C \neq \{\} \longrightarrow \text{Sup } C \in A \longrightarrow C \cap A \neq \{\})$

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* $f \implies f \text{ --}l\text{--} f$ 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 $\cap \{l. \neg lnull l \wedge lhd l = x\}$)*
by (*intro open-Int open-vimageI open-lhd A tendsto-intros*)
also have $(ltl -' A \cap \{l. \neg lnull l \wedge lhd l = x\}) = LCons x ' A$
by *force*
finally show *?thesis* .
qed

lemma *open-Ici*: *lfinite xs \implies 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 \implies 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} \longleftrightarrow 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 \wedge ys \neq x}*
assume *inf: \neg 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 \neq {}*
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: $\bigwedge xs\ ys. Q\ xs \implies lprefix\ ys\ xs \implies Q\ ys$*
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 ∈ 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 ∧ ldistinct xs}
  by (rule closure-eq-lfinite[OF closed-ldistinct ldistinct-lprefix])

lemma closed-ldistinct': (∧x. isCont f x) ⇒ closed {xs. ldistinct (f xs)}
  using continuous-closed-preimage-univ[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 ∧ lsorted xs}
  by (rule closure-eq-lfinite[OF closed-lsorted lsorted-lprefixD])

lemma closed-lsorted': (∧x. isCont f x) ⇒ closed {xs. lsorted (f xs)}
  using continuous-closed-preimage-univ[of - f, OF closed-lsorted] by auto

lemma closed-in-lset: closed {l. x ∈ 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 *: ∧A B. open A ⇒ open B ⇒ a ∈ A ⇒ b ∈ B ⇒
    (∃x∈A. ∃y∈B. 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 ∃ a' ∈ A. ∃ b' ∈ B. 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: ∧x. isCont f x and g: ∧x. isCont g x
  shows closed {x. llist-all2 R (f x) (g x)}
  using isCont-closed-vimage[OF isCont-Pair[OF f g] closed-llist-all2, of R]
  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) ⇔ (∃ xs ≤ l. lfinite xs ∧ (∀ ys ≥ xs. ys ≤ l ⇒ P ys))
  unfolding eventually-nhds

```

```

proof safe
  let ?F = {l'. lprefix l' l ∧ lfinite l'}
  fix A assume open A l ∈ A ∀ l ∈ A. P l
  moreover have chain ?F ?F ≠ {}
  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 ∃ xs. xs ∈ ?F ∧ (∀ ys ≥ xs. ys ∈ ?F ⇒ P ys)
  using open-ccpoD[of A ?F] by auto

```


then show $\exists xs \leq l. \text{lfinite } xs \wedge (\forall ys \geq xs. ys \leq l \longrightarrow P \text{ } ys)$
by (*metis (lifting) (l ∈ A) (∀ l ∈ A. P l) le-llist-conv-lprefix mem-Collect-eq not-lfinite-lprefix-conv-eq*)
next
fix *xs* **assume** $xs \leq l \text{ lfinite } xs \forall ys \geq xs. ys \leq l \longrightarrow P \text{ } ys$
then show $\exists S. \text{open } S \wedge l \in S \wedge \text{Ball } S \text{ } P$
by (*intro exI[of - {xs ..} ∩ {.. l}] conjI open-Int open-Ici open-ccpo-Ici*) *auto*
qed

lemma *nhds-lfinite*: $\text{lfinite } l \implies \text{nhds } l = \text{principal } \{l\}$
unfolding *filter-eq-iff eventually-principal eventually-nhds-llist*
by (*auto simp del: le-llist-conv-lprefix*)

lemma *eventually-at'-llist*:
 $\text{eventually } P \text{ (at' } l) \iff (\exists xs \leq l. \text{lfinite } xs \wedge (\forall ys \geq xs. \text{lfinite } ys \longrightarrow ys \leq l \longrightarrow P \text{ } ys))$
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 $\neg \text{lfinite } l$
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*: $(\bigwedge xs. \text{lfinite } xs \implies xs \leq l \implies P \text{ } xs) \implies \text{eventually } P \text{ (at' } l)$
by (*auto simp: eventually-at'-llist*)

lemma *Lim-at'-lfinite*: $\text{lfinite } xs \implies \text{Lim (at' } xs) f = f \text{ } xs$
by (*rule tendsto-Lim[OF at'-bot] (auto simp add: at'-def topological-tendstoI eventually-principal)*)

lemma *filterlim-at'-list*:
 $(f \longrightarrow y) \text{ (at' } (x :: 'a \text{ llist})) \implies f \text{ } -x \rightarrow y$
unfolding *at'-def* **by** (*auto split: if-split-asm simp: open-singleton-iff-lfinite*)

lemma *tendsto-mcont-llist'*: $\text{mcont } l\text{Sup } l\text{prefix } l\text{Sup } l\text{prefix } f \implies (f \longrightarrow f \text{ } x) \text{ (at' } (x :: 'a \text{ llist}))$
by (*auto simp add: at'-def nhds-lfinite[symmetric] open-singleton-iff-lfinite tendsto-at-iff-tendsto-nhds[symmetric] intro: tendsto-mcont-llist*)

lemma *tendsto-closed*:
assumes *eq: closed {x. P x}*
assumes *ev: (∀ ys. lfinite ys ⟹ ys ≤ x ⟹ 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'-l1istI*)
qed (rule *assms tendsto-id-at' at'-bot*) +
then show *?thesis*
by *simp*
qed

lemma *tendsto-Sup-at'*:
fixes $f :: 'a \text{ l1ist} \Rightarrow 'b::\text{ccpo-topology}$
assumes $f: \bigwedge x y. x \leq y \Longrightarrow \text{lfinite } x \Longrightarrow \text{lfinite } y \Longrightarrow f x \leq f y$
shows $(f \longrightarrow (\text{Sup } (f' \{xs. \text{lfinite } xs \wedge xs \leq l\})))$ (at' l)
proof (rule *topological-tendstoI*)
let $?F = \{xs. \text{lfinite } xs \wedge xs \leq l\}$

have $ch\text{-}F: \text{chain } (f' ?F) f' ?F \neq \{\}$
by (rule *chain-imageI[OF chain-subset, OF lprefixes-chain]*) (auto *simp: f*)

fix A **assume** $A: \text{open } A \text{ Sup } (f' ?F) \in A$ **then show** *eventually* $(\lambda x. f x \in A)$
(at' l)
using *open-ccpoD[OF - ch-F, OF A]* **by** (auto *simp: eventually-at'-l1ist f simp*
del: le-l1ist-conv-lprefix)
qed

lemma *tendsto-Lim-at'*:
fixes $f :: 'a \text{ l1ist} \Rightarrow 'b::\text{ccpo-topology}$
assumes $f: \bigwedge l. f l = \text{Lim } (at' l) f'$
assumes *mono*: $\bigwedge x y. x \leq y \Longrightarrow \text{lfinite } x \Longrightarrow \text{lfinite } y \Longrightarrow f' x \leq f' y$
shows $(f \longrightarrow f l)$ (at' l)
unfolding $f[\text{abs-def}]$
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*) ($\lambda xs. \text{llist-of } (\text{filter } P (\text{list-of } xs))$)

lemma *tendsto-lfilter*: (*lfilter'* *P* \longrightarrow *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* \circ *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'* ($\lambda x. Q\ x \wedge P\ x$) *xs*
by (*rule tendsto-closed*[**where** *x=xs*, *OF closed-Collect-eq-isCont*]) (*auto intro!: isCont-lfilter*)

lemma *lfilter'-LNil-I*[*simp*]: ($\forall x \in \text{lset } xs. \neg P\ x$) \implies *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* \cap $\{x. P\ x\}$
by(*rule tendsto-closed*[*OF closed-Collect-eq-isCont*])
(auto 4 3 intro: isCont-lset isCont-lfilter isCont-inf2)

lemma *lfilter'-eq-LNil-iff*: $lfilter' P xs = LNil \iff (\forall x \in lset xs. \neg P x)$
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 $ys :: 'a\ list$
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 $lconcat' xs = Lim (at' xs) (\lambda xs. foldr lappend (list-of xs) LNil)$

lemma *tendsto-lconcat'*: $(lconcat' \longrightarrow lconcat' xss) (at' xss)$
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*]: $lfinite xs \implies lconcat' xs = foldr lappend (list-of xs) LNil$
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*: $lmap f (lconcat' xss) = lconcat' (lmap (lmap f) (xss::'a\ llist\ llist))$

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]: $ldropWhile'\ P\ (LCons\ l\ xs) = (if\ P\ l\ then\ ldropWhile'\ P\ xs\ else\ LCons\ l\ xs)$

by (rule tendsto-closed[**where** $x=xs$, OF closed-Collect-eq-isCont])

(auto intro!: isCont-ldropWhile' isCont-If isCont-LCons)

lemma *ldropWhile'* $P\ (lmap\ f\ xs) = lmap\ f\ (ldropWhile'\ (P \circ f)\ xs)$

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** $x \in lset xs$?lhs **then show** $P x$ **by** *induct (simp-all split: if-split-asm)*

qed *simp*

lemma *lhd-lfilter'*: $lhd (lfilter' P xs) = lhd (ldropWhile' (Not \circ P) xs)$

proof *cases*

assume $\exists x \in lset xs. P x$

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*

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*: $xs \leq ys \Longrightarrow edrop\ n\ xs \leq edrop\ n\ ys$

using *mono-edrop[of n xs]* **by** (*auto simp add: less-eq-list-def prefix-def*)

definition *ldrop'* $n\ xs = Lim\ (at'\ xs)\ (l\list-of\ \circ\ edrop\ n\ \circ\ list-of)$

lemma *ldrop'-lfinite[simp]*: $lfinite\ xs \Longrightarrow ldrop'\ n\ xs = l\list-of\ (edrop\ n\ (list-of\ xs))$

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-l\list-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 \Rightarrow LCons x xs | eSuc n \Rightarrow 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* $:: 'a :: order \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**

$up\ a\ [] = []$

| $up\ a\ (x\ \# \ xs) = (if\ a < x\ then\ x\ \# \ up\ x\ xs\ else\ up\ a\ xs)$

lemma *set-upD*: $x \in set\ (up\ y\ xs) \implies x \in set\ xs \wedge y < x$
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*: $xs \leq ys \implies up\ a\ xs \leq up\ a\ ys$
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 sorted.Cons*)

11.6 Define more functions on lazy lists as continuous extensions

definition $lup\ a\ xs = Lim\ (at'\ xs)\ (\lambda xs.\ llist-of\ (up\ a\ (list-of\ xs)))$

lemma *tendsto-lup*: $(lup\ a \longrightarrow lup\ a\ xs)\ (at'\ xs)$
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*]: $lfinite\ xs \implies lup\ a\ xs = llist-of\ (up\ a\ (list-of\ xs))$
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*: $lset\ (lup\ x\ xs) \subseteq lset\ xs \cap \{y.\ x < y\}$
by (*rule tendsto-le-ccpo[where g= $\lambda xs.\ lset\ (lup\ x\ xs)$ and $f = \lambda xs.\ lset\ xs \cap \{y.\ x < y\}$ 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 op = 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 op = 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

simps-of-case lup'-simps [simp]: lup'.simps

lemma lset-lup'-subset:
  fixes x :: - :: preorder
  shows lset (lup' x xs) ⊆ lset xs ∩ {y. x < y}
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 in-
tro: 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 (*l*list) *iterate* :: 'a ⇒ 'a *l*list
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 *lxtup* *i* *l* = *Lim* (*at*' *l*) (*l*list-of ∘ *extup* *i* ∘ *list-of*)

definition *lxtdown* *i* *l* = *Lim* (*at*' *l*) (*l*list-of ∘ *extdown* *i* ∘ *list-of*)

lemma *tendsto-lxtup*[*tendsto-intros*]: (*lxtup* *i* ⟶ *lxtup* *i* *xs*) (*at*' *xs*)
by (*rule* *tendsto-Lim-at*'[*OF lxtup-def*]) (*auto simp*: *lfinite-eq-range-llist-of mono-ext*)

lemma *tendsto-lxtdown*[*tendsto-intros*]: (*lxtdown* *i* ⟶ *lxtdown* *i* *xs*) (*at*' *xs*)
by (*rule* *tendsto-Lim-at*'[*OF lxtdown-def*]) (*auto simp*: *lfinite-eq-range-llist-of mono-ext*)

lemma *isCont-lxtup*[*THEN isCont-o2*[*rotated*]]: *isCont* (*lxtup* *a*) *l*
by (*simp* *add*: *isCont-def filterlim-at*'-*list tendsto-lxtup*)

lemma *isCont-lxtdown*[*THEN isCont-o2*[*rotated*]]: *isCont* (*lxtdown* *a*) *l*
by (*simp* *add*: *isCont-def filterlim-at*'-*list tendsto-lxtdown*)

lemma *lxtup-lfinite*[*simp*]: *lfinite* *xs* ⟹ *lxtup* *i* *xs* = *l*list-of (*extup* *i* (*list-of* *xs*))
by (*simp* *add*: *lxtup-def Lim-at*'-*lfinite*)

lemma *lxtdown-lfinite*[*simp*]: *lfinite* *xs* ⟹ *lxtdown* *i* *xs* = *l*list-of (*extdown* *i* (*list-of* *xs*))

by (*simp add: lextdown-def Lim-at'-lfinite*)

lemma *lxtup i LNil = LNil lextdown i LNil = LNil*
by *simp-all*

lemma *lxtup i (LCons x xs) = (if i ≤ x then lxtup 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-lxtup isCont-If isCont-LCons)

lemma *lextdown i (LCons x xs) = (if x ≤ i then lextdown x xs else LCons i (lxtup x xs))*
by (*rule tendsto-closed[where x=xs, OF closed-Collect-eq-isCont]*)
(auto intro!: isCont-lextdown isCont-lxtup isCont-If isCont-LCons)

lemma *lset (lxtup a xs) ⊆ {a} ∪ lset xs*
apply (*rule tendsto-le-ccpo[where g=λxs. lset (lxtup 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 set-mp]*)
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 set-mp]*)
done

lemma *distinct-ext:*
assumes *distinct xs a ∉ set xs*
shows *distinct (lxtup 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 ⇒ a ∉ lset xs ⇒ ldistinct (lxtup a xs)*
by (*rule-tac tendsto-closed[OF closed-ldistinct', OF isCont-lxtup]*)
(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 ∘ 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*: $esum\text{-list } LNil = 0$
by *simp*

context
fixes $xs :: \text{ereal } llist$
assumes $xs: \bigwedge x. x \in lset\ xs \implies 0 \leq x$
begin

lemma *esum-list-tendsto-SUP*:
 $((sum\text{-list} \circ list\text{-of}) \longrightarrow (SUP\ ys : \{ys. lfinite\ ys \wedge ys \leq xs\}. esum\text{-list}\ ys)) (at'\ xs)$
(is $(- \longrightarrow ?y) -$
proof (*rule order-tendstoI*)
fix a **assume** $a < ?y$
then obtain ys **where** $l\text{list-of } ys \leq xs$ $a < sum\text{-list } ys$
by (*auto simp: less-SUP-iff lfinite-eq-range-llist-of*)
moreover
{ **fix** zs **assume** $ys \leq zs$ $l\text{list-of } zs \leq xs$
then obtain ys' **where** $zs: zs = ys @ ys'$
by (*auto simp: prefix-def less-eq-list-def*)
with $\langle l\text{list-of } zs \leq xs \rangle$ **have** $nonneg: 0 \leq sum\text{-list } ys'$
using xs **by** (*auto simp: lprefix-conv-lappend sum-list-sum-nth intro: sum-nonneg*)
note $\langle a < sum\text{-list } ys \rangle$
also have $sum\text{-list } ys \leq sum\text{-list } zs$
using zs *ereal-add-mono*[*OF order-refl nonneg*] **by** *auto*
finally have $a < sum\text{-list } zs$ **}**
ultimately show *eventually* $(\lambda x. a < (sum\text{-list} \circ list\text{-of})\ x) (at'\ xs)$
unfolding *eventually-at'-llist* **by** (*auto simp: lfinite-eq-range-llist-of*)
next
fix a **assume** $?y < a$ **then show** *eventually* $(\lambda x. (sum\text{-list} \circ list\text{-of})\ x < a) (at'\ xs)$
by (*auto intro!: eventually-at'-llistI dest: SUP-lessD*)
qed

lemma *tendsto-esum-list*: $(esum\text{-list} \longrightarrow esum\text{-list } xs) (at'\ xs)$
apply (*rule filterlim-cong*[**where** $g = sum\text{-list} \circ list\text{-of}$, *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* $esum\text{-list } xs$
by (*simp add: isCont-def filterlim-at'-list tendsto-esum-list*)

end

lemma *esum-list-nonneg*:

$(\bigwedge x. x \in \text{lset } xs \implies 0 \leq x) \implies 0 \leq \text{esum-list } xs$
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 $x: 0 \leq x \bigwedge x. x \in \text{lset } xs \implies 0 \leq x$ **shows** $\text{esum-list } (\text{LCons } x \text{ } xs) = x + \text{esum-list } xs$

proof (rule tendsto-unique-eventually[OF at'-bot])

from x **show** $((\lambda xs. \text{esum-list } (\text{LCons } x \text{ } xs)) \longrightarrow \text{esum-list } (\text{LCons } x \text{ } xs))$ (at' xs)

by (intro tendsto-compose[OF filterlim-at'-list[OF tendsto-esum-list] tendsto-LCons])
auto

show eventually $(\lambda xa. \text{esum-list } (\text{LCons } x \text{ } xa) = x + \text{esum-list } xa)$ (at' xs)

using eventually-lfinite **by** eventually-elim simp

from x **show** $((\lambda xa. x + \text{esum-list } xa) \longrightarrow x + \text{esum-list } xa)$ (at' xs)

by (intro esum-list-nonneg tendsto-esum-list tendsto-add-ereal) auto

qed

lemma *esum-list-lfilter'*:

assumes $nn: \bigwedge x. x \in \text{lset } xs \implies 0 \leq x$ **shows** $\text{esum-list } (\text{lfilter}' (\lambda x. x \neq 0) xs) = \text{esum-list } xs$

proof (rule tendsto-unique-eventually[OF at'-bot])

show $(\text{esum-list} \longrightarrow \text{esum-list } xs)$ (at' xs)

using nn **by** (rule tendsto-esum-list)

from nn **show** $((\lambda xs. \text{esum-list } (\text{lfilter}' (\lambda x. x \neq 0) xs)) \longrightarrow \text{esum-list } (\text{lfilter}' (\lambda x. x \neq 0) xs))$ (at' xs)

by (intro tendsto-compose[OF filterlim-at'-list[OF tendsto-esum-list] tendsto-lfilter])
(auto simp: lset-lfilter')

show eventually $(\lambda xs. \text{esum-list } (\text{lfilter}' (\lambda x. x \neq 0) xs) = \text{esum-list } xs)$ (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 $f:: \text{nat list} \Rightarrow \text{nat list}$ **where**

$f [] = []$

| $f (x \# xs) = (x * 2) \# f (f xs)$

by auto pat-completeness

termination f

proof (relation inv-image natLess length)

fix $x \text{ } xs$ **assume** $f\text{-dom } xs$ **then show** $(f \text{ } xs, x \# xs) \in \text{inv-image } \text{natLess } \text{length}$

by induct (auto simp: f.psimps natLess-def)

qed (auto intro: wf-less simp add: natLess-def)

lemma *length-f[simp]*: $\text{length } (f \text{ } xs) = \text{length } xs$

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** $x=xs$, *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 ($rel\text{-}set A \implies op =$) *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*: $\llbracket monotone\ orda\ ordb\ g; monotone\ ordb\ ordc\ f \rrbracket \implies$
 $monotone\ orda\ ordc\ (f \circ g)$
by(*rule monotoneI*)(*simp add: monotoneD*)

lemma *cont-comp*: $\llbracket mcont\ luba\ orda\ lubb\ ordb\ g; cont\ lubb\ ordb\ lubc\ ordc\ f \rrbracket \implies$
 $cont\ luba\ orda\ lubc\ ordc\ (f \circ g)$
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:  $\llbracket$  mcont luba orda lubb ordb g; mcont lubb ordb lubc ordc f  $\rrbracket$ 
 $\implies$  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  $((A \implies A \implies op =) \implies (B \implies B \implies op =) \implies$ 
 $(A \implies B) \implies op =)$  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  $((rel\text{-set } A \implies A) \implies (A \implies A \implies op =) \implies (rel\text{-set}$ 
 $B \implies B) \implies (B \implies B \implies op =) \implies (A \implies B) \implies$ 
 $op =)$  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  $((rel\text{-set } A \implies A) \implies (A \implies A \implies op =) \implies (rel\text{-set}$ 
 $B \implies B) \implies (B \implies B \implies op =) \implies (A \implies B) \implies$ 
 $op =)$  mcont mcont
unfolding mcont-def[abs-def] by transfer-prover

```

```

end

```

```

lemma (in ccpo) Sup-Un-less:
  assumes chain: Complete-Partial-Order.chain op  $\leq$  (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 antisym)
  from chain have chain': Complete-Partial-Order.chain op  $\leq$  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  $\langle x \leq y \rangle$ 
    also from  $\text{chain}' \langle y \in B \rangle$  have  $y \leq \text{Sup } B$  by(rule ccpo-Sup-upper)
    finally show ?thesis .
  qed(rule ccpo-Sup-upper[OF chain'])
qed
show  $\text{Sup } B \leq \text{Sup } (A \cup B)$ 
  using  $\text{chain } \text{chain}'$  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* $\Rightarrow \text{bool}$
 is $\lambda(xs1, b1) (xs2, b2)$. if *lfinite* $xs1$ then $b1 = b \wedge \text{lprefix } xs1 \ xs2 \vee xs1 = xs2 \wedge$
flat-ord $b \ b1 \ b2$ else $xs1 = xs2$
 by *auto*

lift-definition *tSup* :: $('a, 'b)$ *tllist set* $\Rightarrow ('a, 'b)$ *tllist*
 is λA . (*lSup* (*fst* ' A), *flat-lub* b (*snd* ' ($A \cap \{(xs, -). \text{lfinite } xs\}$)))
proof *goal-cases*
 case (1 $A1 \ A2$)
 hence *fst* ' $A1 = \text{fst}$ ' $A2$ *snd* ' ($A1 \cap \{(xs, -). \text{lfinite } xs\}$) = *snd* ' ($A2 \cap \{(xs,$
 $-). \text{lfinite } xs\}$)
 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*: $\llbracket \text{tllist-ord } xs \ ys; \text{tllist-ord } ys \ xs \rrbracket \Longrightarrow xs = ys$
 by *transfer*(*auto simp add: flat-ord-def split: if-split-asm intro: lprefix-antisym*)

lemma *tllist-ord-trans* [*trans*]: $\llbracket \text{tllist-ord } xs \ ys; \text{tllist-ord } ys \ zs \rrbracket \Longrightarrow \text{tllist-ord } xs$
 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 ∧ 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(clarsimp 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 $A \subseteq \{b, z\}$ **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 ($\forall x \in A. is-TNil x$)*
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 $\langle tfinite \ xs \rangle$ **have** *lfinite (lSup (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 $\langle tfinite \ xs \rangle$ +
ultimately show *?thesis* **using** *True* **by**(simp add: tllist-ord.rep-eq)
next
case *False*

hence $\exists ys \in A. \neg \text{tllist-ord } ys \ xs$
by(*rule contrapos- \neg*)(*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** $ys \in A \neg \text{tllist-ord } ys \ xs$ **by** *blast*
with *A* **have** *tllist-ord xs ys xs \neq 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: $\bigwedge xs'. xs' \in A \implies \text{tllist-ord } xs' \ xs$*
shows *tllist-ord (tSup A) xs*
proof –
have $\bigwedge xs'. xs' \in \text{llist-of-tllist } 'A \implies \text{lprefix } xs' (\text{llist-of-tllist } xs)$
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 \in A \wedge 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 lprefix-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-llist-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:
  ∀ xs ∈ Y. is-TNil xs ⇒ terminal (tSup Y) = flat-lub b (terminal ' Y)
including tllist.lifting
apply(transfer)
apply(clarsimp simp add: split-def)
apply(rule arg-cong[where f=flat-lub b])
apply(auto simp add: split-def)
done

```

```

lemma thd-tSup:

```

```

   $\exists xs \in Y. \neg is-TNil\ xs$ 
   $\implies thd\ (tSup\ Y) = (THE\ x.\ x \in thd\ ' (Y \cap \{xs.\ \neg is-TNil\ xs\}))$ 
apply(simp add: tSup-def image-image)
apply(rule arg-cong[where f=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)  $\implies$  op =) ( $\lambda Y.\ \exists (xs, b) \in Y.\ \neg lnull\ xs$ )
( $\lambda Y.\ \exists (xs, b) \in Y.\ \neg lnull\ xs$ )
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 \implies 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 ($\exists xs \in Y.\ \neg is-TNil\ xs$)
by transfer auto

lemma *retain-TCons-raw-parametric*:

```

includes lifting-syntax
shows (rel-set (rel-prod (llist-all2 A) B)  $\implies$  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 \implies ('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 \cap {xs. $\neg is-TNil\ xs$ }
by(auto simp add: retain-TCons-def intro: rev-image-eqI)

lemma *ttl-tSup*:

```

 $\llbracket$  Complete-Partial-Order.chain tllist-ord Y;  $\exists xs \in Y.\ \neg is-TNil\ xs$   $\rrbracket$ 
 $\implies$  ttl (tSup Y) = tSup (ttl ' (Y  $\cap$  {xs.  $\neg is-TNil\ xs$ }))

```

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') \in Y \neg lnull xs **by** blast

note chain = prems(1)

have flat-lub b (snd ' (Y \cap {(xs, -). lfinite xs})) = flat-lub b (insert b (snd ' (Y \cap {(xs, -). lfinite xs})))

by(auto simp add: flat-lub-def)

also have insert b (snd ' (Y \cap {(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-llnull intro!: rev-image-eqI)
also have flat-lub b ... = flat-lub b (snd ‘ (apfst ltl ‘ (Y ∩ {(xs, b). ¬ lnull xs})
∩ {(xs, -). lfinite xs}))
  by(auto simp add: flat-lub-def)
also
  have ltl ‘ (fst ‘ Y ∩ {xs. ¬ lnull xs}) = fst ‘ apfst ltl ‘ (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: ¬ is-TNil xs ⇔ (∃ 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: ∀x. monotone tllist-ord ord (λxs. f x xs (TCons x xs))
  and ord: class.preorder ord (mk-less ord)
  and bot: ∀x. ord (bot b) x
  shows monotone tllist-ord ord (λxs. case xs of TNil b ⇒ bot b | TCons x xs' ⇒
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 ≡ λxs. f (thd xs) (ttl xs) (TCons (thd xs) (ttl xs))
assumes mcont: ∧x. mcont tSup tllist-ord lub ord (λxs. f x xs (TCons x xs))
and ccpo: class.ccpo lub ord (mk-less ord)
and bot: ∧x. ord (bot b) x
and cont-bot: cont (flat-lub b) (flat-ord b) lub ord bot
shows mcont tSup tllist-ord lub ord (λxs. case xs of TNil b ⇒ bot b | TCons x
xs' ⇒ 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 ≠ {}

```

```

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.strong-SUP-cong)

```

```

next

```

```

case False

```

```

hence eq: tSup Y = TCons (thd (tSup Y)) (ttl (tSup Y)) by simp

```

```

have eq':

```

```

?f ' Y =

```

```

(λx. bot (terminal x)) ' (Y ∩ {xs. is-TNil xs}) ∪

```

```

(λxs. f (thd xs) (ttl xs) xs) ' (Y ∩ {xs. ¬ is-TNil xs})

```

```

by(auto 4 3 split: tllist.splits intro: rev-image-eqI)

```

```

from False obtain xs where xs: xs ∈ Y ∩ is-TNil xs by auto
{ fix ys

```

```

assume  $ys \in Y$  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  $lub: lub (?f \text{ ' } Y) = lub ((\lambda xs. f (thd\ xs) (ttl\ xs) xs) \text{ ' } (Y \cap \{xs. \neg is-TNil\ xs\}))$ 
  using  $xs$  chain' unfolding  $eq'$ 
  by  $-(erule\ cppo.Sup-Un-less$ [OF cppo], auto simp add: intro!: bot)
  { fix  $xs$ 
    assume  $xs: xs \in Y \cap \{xs. \neg is-TNil\ xs\}$ 
    hence (THE  $x. x \in thd \text{ ' } (Y \cap \{xs. \neg is-TNil\ xs\}) = thd\ xs$ 
      by(auto dest: chainD[OF chain] simp add: not-is-TNil-conv intro!: the-equality)
    )
    hence  $f (THE\ x. x \in thd \text{ ' } (Y \cap \{xs. \neg is-TNil\ xs\})) (ttl\ xs) (TCons (THE\ x. x \in thd \text{ ' } (Y \cap \{xs. \neg is-TNil\ xs\})) (ttl\ xs)) = f (thd\ xs) (ttl\ xs) xs$ 
      using  $xs$  by simp }
    moreover have Complete-Partial-Order.chain tllist-ord ( $ttl \text{ ' } (Y \cap \{xs. \neg is-TNil\ xs\})$ )
      using chain by(rule chain-imageI[OF chain-subset])(auto simp add: tllist-ord-ttlI)
    moreover from False have  $Y \cap \{xs. \neg is-TNil\ xs\} \neq \{\}$  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] image-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 op  $\subseteq$  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 op  $\subseteq$  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:
   $(\bigwedge x. A (f x) (g x)) \implies (op = \implies A) f g$ 
by(simp add: rel-fun-def)

lemma tllist-ord-transfer [transfer-rule]:
   $(op = \implies pcr-tllist op = op = \implies pcr-tllist op = op = \implies op =)$ 
   $(\lambda b (xs1, b1) (xs2, b2). \text{if } lfinite\ xs1 \text{ then } b1 = b \wedge lprefix\ xs1\ xs2 \vee xs1 =$ 
   $xs2 \wedge flat-ord\ b\ b1\ b2 \text{ else } xs1 = xs2)$ 
  tllist-ord
by(rule rel-fun-lift)(rule tllist-ord.transfer)

lemma tSup-transfer [transfer-rule]:
   $(op = \implies rel-set (pcr-tllist op = op =) \implies pcr-tllist op = op =)$ 
   $(\lambda b A. (lSup (fst ' A), flat-lub b (snd ' (A \cap \{(xs, -). lfinite\ xs\}))))$ 
  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*: $\bigwedge b. \text{monotone } \text{orda } \text{ordb } (\lambda f. \text{tnil } f \ b)$

and *tcons*: $\bigwedge x \ xs. \text{monotone } \text{orda } \text{ordb } (\lambda f. \text{tcons } f \ x \ xs)$

shows *monotone orda ordb* $(\lambda f. \text{case-tllist } (\text{tnil } f) (\text{tcons } f) \ xs)$

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 \ll *Partial-Function.init tllist* $\@$ {*term tllist.fix-fun b*}

$\@$ {*term tllist.mono-body b*} $\@$ {*thm tllist.fix-rule-uc[where b=b]*} $\@$ {*thm tllist.fix-induct-uc[where b=b]*} *NONE* \gg

abbreviation *mono-tllist where mono-tllist* \equiv *monotone (fun-ord (tllist-ord b))*
(*tllist-ord b*)

lemma *LCons-mono* [*partial-function-mono, cont-intro*]:

mono-tllist A \implies *mono-tllist* $(\lambda f. \text{TCons } x \ (A \ f))$

by(*rule monotoneI*)(*auto dest: monotoneD*)

end

lemma *mono-tllist-lappend2* [*partial-function-mono*]:

tllist-pf.mono-tllist b A \implies *tllist-pf.mono-tllist b* $(\lambda f. \text{lappend } xs \ (A \ f))$

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 $\bigwedge y. \text{tllist-pf.mono-tllist } b \ (C \ y)$

shows *tllist-pf.mono-tllist b* $(\lambda f. \text{tappend } xs \ (\lambda y. C \ y \ f))$

apply(*cases tfinite xs*)

apply(*rule monotoneI*)

apply(*drule monotoneD[OF assms[where y=terminal xs]]*)

including *tllist.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 TLList-CCPO-Examples imports
  ../TLList-CCPO
begin

```

```

context fixes b :: 'b begin
interpretation tllist-pf b .

```

```

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

```

```

partial-function (tllist) tfilter :: ('a, 'b) tllist  $\Rightarrow$  ('a, 'b) tllist
where

```

```

  tfilter xs = (case xs of TNil b'  $\Rightarrow$  TNil b' | TCons x xs'  $\Rightarrow$  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)  $\longleftrightarrow$  ( $\forall x \in \text{tset } xs. \neg P x$ ) (is ?lhs
 $\longleftrightarrow$  ?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: tllist.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

```

```

simps-of-case tconcat2-simps [simp]: tconcat.simps

```

```

end

```

14 Example: Koenig's lemma

```

theory Koenigslemma imports
  ../Coinductive-List
begin

type-synonym 'node graph = 'node  $\Rightarrow$  '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:  $\llbracket$  graph x y; LCons y xs  $\in$  paths graph  $\rrbracket \Longrightarrow$  LCons x (LCons y xs)  $\in$ 
  paths graph

definition connected :: 'node graph  $\Rightarrow$  bool
where connected graph  $\longleftrightarrow$   $(\forall n n'. \exists xs. \text{llist-of } (n \# xs @ [n']) \in \text{paths graph})$ 

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  $\llbracket$  LCons n xs  $\in$  paths graph; n'  $\in$  lset xs; lset xs  $\subseteq$  ns  $\rrbracket \Longrightarrow$  n'  $\in$  reachable-via
  graph ns n

lemma connectedD: connected graph  $\Longrightarrow$   $\exists xs. \text{llist-of } (n \# xs @ [n']) \in \text{paths}$ 
  graph
unfolding connected-def by blast

lemma paths-LConsD:
  assumes LCons x xs  $\in$  paths graph
  shows xs  $\in$  paths graph
using assms
by (coinduct)(fastforce elim: paths.cases del: disjCI)

lemma paths-lappendD1:
  assumes lappend xs ys  $\in$  paths graph
  shows xs  $\in$  paths graph
using assms
apply coinduct
apply(erule paths.cases)
  apply(simp add: lappend-eq-LNil-iff)
  apply(case-tac x)
  apply simp
  apply(simp add: lappend-eq-LNil-iff)
apply(case-tac x)
  apply simp
apply(case-tac x2)

```

apply *simp*
apply *simp*
done

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 $\exists xs'. LCons n xs' \in paths\ graph \wedge lset\ xs' \subseteq lset\ xs \wedge x \in lset\ xs' \wedge n \notin lset\ xs'$
proof –
from *set* **obtain** *xs' xs''* **where** *lfinite xs'*
and *xs: xs = lappend xs' (LCons x xs'')*
and $x \notin lset\ xs'$
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** $lappend\ (LCons\ n\ (lappend\ xs'\ (LCons\ x\ LNil)))\ xs'' \in paths\ graph$
by(*simp add: lappend-assoc*)
hence $LCons\ n\ ?xs' \in paths\ graph$ **by**(*rule paths-lappendD1*)
moreover **have** $x \in lset\ ?xs' \wedge lset\ ?xs' \subseteq lset\ xs \wedge n \notin lset\ ?xs'$
using *xs False lfinite xs' n-neq-x* **by** *auto*
ultimately **show** *?thesis* **by** *blast*
next
case *True*
from $\langle lfinite\ xs' \rangle$ **obtain** *XS'* **where** $xs' = llist-of\ XS'$
unfolding *lfinite-eq-range-llist-of* **by** *blast*
with *True* **have** $n \in set\ XS'$ **by** *simp*
from *split-list-last[OF this]*
obtain *ys zs* **where** $XS' = ys @ n \# zs$
and $n \notin set\ zs$ **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 $lappend\ (LCons\ n\ (llist-of\ ys))\ (lappend\ (LCons\ n\ ?xs')\ xs'') \in paths\ graph$
by(*simp add: lappend-assoc*)(*metis lappend-assoc lappend-llist-of-LCons lappend-llist-of-llist-of llist-of.simps(2)*)
ultimately **have** $lappend\ (LCons\ n\ ?xs')\ xs'' \in paths\ graph$

by(rule paths-lappendD2)
 hence $LCons\ n\ ?xs' \in paths\ graph$ by(rule paths-lappendD1)
 moreover have $x \in lset\ ?xs' \wedge lset\ ?xs' \subseteq lset\ xs \wedge n \notin lset\ ?xs'$
 using $xs\ xs'\ XS'\ \langle lfinite\ xs' \rangle\ n\ neq\ x\ \langle n \notin set\ zs \rangle$ by auto
 ultimately show ?thesis by blast
 qed
 qed

lemma *reachable-via-subset-unfold*:
 $reachable\ via\ graph\ ns\ n \subseteq (\bigcup n' \in \{n'.\ graph\ n\ n'\} \cap ns.\ insert\ n' (reachable\ via\ graph\ (ns - \{n'\})\ n'))$
 (is ?lhs \subseteq ?rhs)
proof(rule subsetI)
 fix x
 assume $x \in ?lhs$
 then obtain xs where $path: LCons\ n\ xs \in paths\ graph$
 and $x \in lset\ xs \wedge lset\ xs \subseteq ns$ by cases

from $\langle x \in lset\ xs \rangle$ 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 $\langle lset\ xs \subseteq ns \rangle$ xs have $n' \in ns$ by simp
 ultimately have $n' \in \{n'.\ graph\ n\ n'\} \cap ns$ by simp
 thus $x \in ?rhs$
proof(rule UN-I)
 show $x \in insert\ n' (reachable\ via\ graph\ (ns - \{n'\})\ n')$
proof(cases $x = n'$)
 case True thus ?thesis by simp
 next
 case False
 with $xs\ \langle x \in lset\ xs \rangle$ have $x \in lset\ xs'$ by simp
 with path xs have $path': LCons\ n'\ xs' \in paths\ graph$ by cases simp-all
 from $\langle lset\ xs \subseteq ns \rangle$ xs have $lset\ xs' \subseteq ns$ by simp

from *path-avoid-node*[OF $path'\ \langle x \in lset\ xs' \rangle$] False
 obtain xs'' where $path'': LCons\ n'\ xs'' \in paths\ graph$
 and $lset\ xs'' \subseteq lset\ xs' \wedge x \in lset\ xs'' \wedge n' \notin lset\ xs''$ by blast
 with False $\langle lset\ xs \subseteq ns \rangle$ 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*: $\bigwedge n.\ finite\ \{n'.\ graph\ n\ n'\}$
 shows $\exists xs \in paths\ graph.\ n \in lset\ xs \wedge \neg lfinite\ xs \wedge ldistinct\ xs$
proof(intro *beI conjI*)

```

let ?P =  $\lambda(n, ns) n'. \text{graph } n \ n' \wedge \text{infinite } (\text{reachable-via graph } (- \text{insert } n \text{ (insert } n' \ ns)) \ n') \wedge n' \notin \text{insert } n \ ns$ 
def LTL ==  $\lambda(n, ns). \text{let } n' = \text{SOME } n'. ?P \ (n, ns) \ n' \text{ in } (n', \text{insert } n \ ns)$ 
def f == unfold-llist ( $\lambda-. \text{False}$ ) fst LTL
def ns == {} :: 'node set

{ fix n ns
  assume infinite (reachable-via graph (- insert n ns) n)
  hence  $\exists n'. ?P \ (n, ns) \ n'$ 
  proof(rule contrapos-np)
    assume  $\neg ?thesis$ 
    hence fin:  $\bigwedge n'. \llbracket \text{graph } n \ n'; n' \notin \text{insert } n \ ns \rrbracket \implies \text{finite } (\text{reachable-via graph } (- \text{insert } n \ (\text{insert } n' \ ns)) \ n')$ 
      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 ( $\{n'. \text{graph } n \ n'\} \cap - \text{insert } n \ ns$ ) by blast
    next
      fix n'
      assume  $n' \in \{n'. \text{graph } n \ n'\} \cap - \text{insert } n \ ns$ 
      hence graph n n' n'  $\notin \text{insert } n \ ns$  by simp-all
      from fin[OF this] have finite (reachable-via graph (- insert n (insert n' ns)) n') .
      moreover have  $- \text{insert } n \ (\text{insert } n' \ ns) = - \text{insert } n \ ns - \{n'\}$  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  $\neg \text{lnull } (f \ (n, ns))$ 
    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  $x \in \text{lset } (f \ (n, ns)) \ n \notin ns$ 
  and finite ns infinite (reachable-via graph (- insert n ns) n)
  hence  $x \notin ns$ 
  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.set-sel(1)[OF f-simps(1), of n ns] by(simp del: llist.set-sel(1))

show ¬ lfinite (f (n, ns))
proof
  assume lfinite (f (n, ns))
  thus False by(induct f (n, ns) arbitrary: n ns rule: lfinite-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** $xs\text{-def}: xs = f\ (n,\ ns)$
and $finite\ ns$ **and** $infinite\ (reachable\text{-via}\ graph\ (-\ insert\ n\ ns)\ n)$ **by** $blast$
from $\langle infinite\ (reachable\text{-via}\ graph\ (-\ insert\ n\ ns)\ n) \rangle$
have $\exists n'. ?P\ (n,\ ns)\ n'$ **by** $(rule\ ex\text{-}P\text{-}I)$
hence $P: ?P\ (n,\ ns)\ (Eps\ (?P\ (n,\ ns)))$ **by** $(rule\ someI\text{-}ex)$
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 $xs\text{-def}$ **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 $(rule\ llist.expand)\ simp\text{-}all$
 moreover have $finite\ ?ns'$ **using** $\langle finite\ ns \rangle$ **by** $simp$
 moreover have $insert\ ?n'\ ?ns' = insert\ n\ (insert\ ?n'\ ns)$ **by** $auto$
 hence $infinite\ (reachable\text{-via}\ graph\ (-\ insert\ ?n'\ ?ns')\ ?n')$ **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 $\langle infinite\ (reachable\text{-via}\ graph\ (-\ \{n\})\ n) \rangle$
have $infinite\ (reachable\text{-via}\ graph\ (-\ insert\ n\ ns)\ n) \wedge finite\ ns$
by $(simp\ add: ns\text{-}def)$
thus $ldistinct\ (f\ (n,\ ns))$
proof $(coinduction\ arbitrary: n\ ns)$
 case $(ldistinct\ n\ ns)$
 then obtain $finite\ ns$
 and $infinite\ (reachable\text{-via}\ graph\ (-\ insert\ n\ ns)\ n)$ **by** $simp$
 from $\langle infinite\ (reachable\text{-via}\ graph\ (-\ insert\ n\ ns)\ n) \rangle$
 have $\exists n'. ?P\ (n,\ ns)\ n'$ **by** $(rule\ ex\text{-}P\text{-}I)$
 hence $P: ?P\ (n,\ ns)\ (Eps\ (?P\ (n,\ ns)))$ **by** $(rule\ someI\text{-}ex)$
 let $?n' = Eps\ (?P\ (n,\ ns))$
 let $?ns' = insert\ n\ ns$
 have $eq: insert\ ?n'\ ?ns' = insert\ n\ (insert\ ?n'\ ns)$ **by** $auto$
 hence $n \notin lset\ (f\ (?n',\ ?ns'))$
 using $P\ \langle finite\ ns \rangle$ **by** $(auto\ dest: lset)$
 moreover from $\langle finite\ ns \rangle$ $P\ eq$
 have $infinite\ (reachable\text{-via}\ graph\ (-\ insert\ ?n'\ ?ns')\ ?n')$ **by** $simp$
 ultimately show $?case$ **using** $\langle finite\ ns \rangle$ **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` \Rightarrow `'a llist` \Rightarrow `'a llist`

where

`lmirror-aux` `acc` `xs` = (case `xs` of `LNil` \Rightarrow `acc` | `LCons` `x` `xs'` \Rightarrow `LCons` `x` (`lmirror-aux` (`LCons` `x` `acc`) `xs'`))

definition `lmirror` :: `'a llist` \Rightarrow `'a llist`

where `lmirror` = `lmirror-aux` `LNil`

simps-of-case `lmirror-aux-simps` [`simp`]: `lmirror-aux.code`

lemma `lnull-lmirror-aux` [`simp`]:

`lnull` (`lmirror-aux` `acc` `xs`) = (`lnull` `xs` \wedge `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`(\exists)[`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 ?lhs **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 (hide-lams, 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 *lnull xs* \implies *ltl (lmirror xs) = lappend (lmirror (ltl xs)) (LCons (lhd xs) LNil)*

by(*clarsimp 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*:
 $\llbracket llist\ all2\ P\ acc\ acc'; llist\ all2\ P\ xs\ xs' \rrbracket$
 $\implies llist\ all2\ P\ (lmirror\ aux\ acc\ xs)\ (lmirror\ aux\ acc'\ xs')$
by(*coinduction arbitrary: acc acc' xs xs'*)(*auto simp add: lhd-lmirror-aux ltl-lmirror-aux elim: llist-all2-lhdD intro!: llist-all2-ltl exI dest: llist-all2-lnullD*)

lemma *enat-mult-cancel1* [*simp*]:
 $k * m = k * n \iff m = n \vee k = 0 \vee k = (\infty :: enat) \wedge n \neq 0 \wedge m \neq 0$
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*:
 $\llbracket llist\ all2\ P\ (lmirror\ aux\ acc\ xs)\ (lmirror\ aux\ acc'\ xs'); llist\ all2\ P\ acc\ acc'; lfinite\ acc \rrbracket$
 $\implies llist\ all2\ P\ xs\ xs'$

proof(*coinduction arbitrary: xs xs' acc acc'*)
case (*LNil xs xs' acc acc'*)
from $\langle llist\ all2\ P\ (lmirror\ aux\ acc\ xs)\ (lmirror\ aux\ acc'\ xs') \rangle$
have $llength\ (lmirror\ aux\ acc\ xs) = llength\ (lmirror\ aux\ acc'\ xs')$
by(*rule llist-all2-llengthD*)
moreover from $\langle llist\ all2\ P\ acc\ acc' \rangle$
have $llength\ acc = llength\ acc'$ **by**(*rule llist-all2-llengthD*)
ultimately have $llength\ xs = llength\ xs'$
using $\langle lfinite\ acc \rangle$ **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 (lmirror-aux acc xs) (lmirror-aux acc' xs')⟩ ⟨¬ lnull xs⟩ ⟨¬ lnull xs'⟩
  have ?lhd by(auto dest: llist-all2-lhdD simp add: lhd-lmirror-aux)
  thus ?case using LCons ⟨llist-all2 P (lmirror-aux acc xs) (lmirror-aux acc' xs')⟩[THEN llist-all2-ltI]
  by(auto 4 3 simp add: lt-lmirror-aux)
qed

```

```

lemma llist-all2-lmirrorI:
  llist-all2 P xs ys  $\implies$  llist-all2 P (lmirror xs) (lmirror ys)
by(simp add: lmirror-def llist-all2-lmirror-aux)

```

```

lemma llist-all2-lmirrorD:
  llist-all2 P (lmirror xs) (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 (lmirror xs) (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  $\implies$  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  $\infty \sqcap n = n \sqcap \infty = n$ 
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: $\llbracket \neg$ lnull xs; \neg lnull ys $\rrbracket \Longrightarrow$ lhd (lmerge xs ys) = (if lhd xs < lhd ys then lhd xs else lhd ys)
 by (auto split: llist.split)

lemma ltl-lmerge:
 $\llbracket \neg$ lnull xs; \neg lnull ys $\rrbracket \Longrightarrow$
 ltl (lmerge xs ys) =
 (if lhd xs < lhd ys then lmerge (ltl xs) ys
 else if lhd ys < lhd xs then lmerge xs (ltl ys)
 else lmerge (ltl xs) (ltl ys))
 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:
 \llbracket lprefix xs xs'; lprefix ys ys' \rrbracket
 \Longrightarrow 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-llist* F **and** G : *mono-llist* G

shows *mono-llist* $(\lambda f. \text{lmerge } (F f) (G f))$

by(*blast intro: monotoneI lprefix-lmergeI monotoneD[OF F] monotoneD[OF G]*)

lemma *in-lset-lmergeD*: $x \in \text{lset } (\text{lmerge } xs \ ys) \implies x \in \text{lset } xs \vee x \in \text{lset } ys$

by(*induct zs \equiv lmerge xs ys arbitrary: xs ys rule: llist-set-induct*)(*auto simp add:*

lhd-lmerge ltl-lmerge split: if-split-asm dest: in-lset-ltlD)

lemma *lset-lmerge*: $\text{lset } (\text{lmerge } xs \ ys) \subseteq \text{lset } xs \cup \text{lset } ys$

by(*auto dest: in-lset-lmergeD*)

lemma *lfinite-lmergeD*: $\text{lfinite } (\text{lmerge } xs \ ys) \implies \text{lfinite } xs \vee \text{lfinite } ys$

by(*induct zs \equiv lmerge xs ys arbitrary: xs ys rule: lfinite-induct*)(*fastforce simp add:*

ltl-lmerge if-pull2 split: if-split-asm)**+**

lemma *fixes F*

defines $F \equiv \lambda \text{lmerge } (xs, ys). \text{case } xs \text{ of } LNil \Rightarrow LNil \mid LCons \ x \ xs' \Rightarrow \text{case } ys \text{ of } LNil \Rightarrow LNil \mid LCons \ y \ ys' \Rightarrow (\text{if } x < y \text{ then } LCons \ x \ (\text{curry } \text{lmerge } xs' \ ys) \text{ else if } y < x \text{ then } LCons \ y \ (\text{curry } \text{lmerge } xs \ ys') \text{ else } LCons \ y \ (\text{curry } \text{lmerge } xs' \ ys'))$

shows *lmerge-conv-fixp*: $\text{lmerge} \equiv \text{curry } (\text{ccpo.fixp } (\text{fun-lub } \text{lSup}) (\text{fun-ord } \text{lprefix}) F)$ (**is** *?lhs \equiv ?rhs*)

and *lmerge-mono*: *mono-llist* $(\lambda \text{lmerge}. F \ \text{lmerge } xs)$ (**is** *?mono xs*)

proof(*intro eq-reflection ext*)

show *mono*: $\bigwedge xs. \ ?mono \ xs$ **unfolding** $F\text{-def}$ **by**(*tactic* $\langle\langle \text{Partial-Function.mono-tac} \ @\{\text{context}\} \ 1 \ \rangle\rangle$)

fix $xs \ ys$

show $\text{lmerge } xs \ ys = \ ?rhs \ xs \ ys$

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* $(\text{rel-prod } \text{lprefix } \text{lprefix}) \ \text{lprefix} \ (\text{case-prod } \text{lmerge})$

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* $\text{lprefix } \text{lprefix} \ (\lambda xs. \ \text{lmerge } xs \ ys)$

by(*simp add: monotone-lmerge[simplified]*)

lemma *mono2mono-lmerge2* [*THEN llist.mono2mono, cont-intro, simp*]:

shows *monotone-lmerge2*: *monotone* $\text{lprefix } \text{lprefix} \ (\lambda ys. \ \text{lmerge } xs \ ys)$

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* ($\lambda x s. lmerge\ x\ s$)
by(*simp* *add*: *mcont-lmerge*[*simplified*])

lemma *mcont2mcont-lmerge2* [*THEN* *llist.mcont2mcont*, *cont-intro*, *simp*]:
shows *mcont-lmerge2*: *mcont* *lSup* *lprefix* *lSup* *lprefix* ($\lambda y s. lmerge\ x\ s$)
by(*simp* *add*: *mcont-lmerge*[*simplified*])

lemma *lfinite-lmergeI* [*simp*]: $\llbracket lfinite\ x s; lfinite\ y s \rrbracket \implies lfinite\ (lmerge\ x\ s\ y s)$
proof(*induction* *arbitrary*: *ys* *rule*: *lfinite-induct*)
case (*LCons* *xs*)
from *LCons.prem*s *LCons.hyps* *LCons.IH*
show *?case* **by** *induct*(*clarsimp* *simp* *add*: *not-lnull-conv* *lmerge-simps*) +
qed *simp*

lemma *linfinite-lmerge* [*simp*]: $\llbracket \neg lfinite\ x s; \neg lfinite\ y s \rrbracket \implies \neg lfinite\ (lmerge\ x\ s\ y s)$
by(*auto* *dest*: *lfinite-lmergeD*)

lemma *llength-lmerge-above*: *llength* *xs* \sqcap *llength* *ys* \leq *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*:
 $\llbracket x \in lset\ x s; lsorted\ x s; \neg lfinite\ y s; \exists y \in lset\ y s. x \leq y \rrbracket$
 $\implies x \in lset\ (lmerge\ x\ s\ y s)$
proof(*induction* *arbitrary*: *ys* *rule*: *lset-induct*)
case (*find* *xs*)
then **obtain** *y* **where** *y* \in *lset* *ys* *x* \leq *y* **by** *blast*
thus *?case* **by** *induct*(*auto* *simp* *add*: *lmerge-simps* *not-less*)
next

case (*step* $x' xs$)
then obtain y **where** $y \in \text{lset } ys$ $x \leq y$ **by** *blast*
moreover from $\langle \text{lsorted } (LCons\ x'\ xs) \rangle \langle x \in \text{lset } xs \rangle \langle x \neq x' \rangle$
have $x' < x$ *lsorted* xs **by**(*auto simp add: lsorted-LCons*)
ultimately show ?*case* **using** $\langle \neg \text{lfinite } ys \rangle$
by *induct*(*auto 4 3 simp add: lmerge-simps* $\langle x \neq x' \rangle$ *not-less intro: step.IH dest:*
in-lset-lmergeD)
qed

lemma *in-lset-lmergeI2*:

$\llbracket x \in \text{lset } ys; \text{lsorted } ys; \neg \text{lfinite } xs; \exists y \in \text{lset } xs. x \leq y \rrbracket$
 $\implies x \in \text{lset } (\text{lmerge } xs\ ys)$

proof(*induction arbitrary: xs rule: lset-induct*)

case (*find* ys)

then obtain y **where** $y \in \text{lset } xs$ $x \leq y$ **by** *blast*

thus ?*case* **by** *induct*(*auto simp add: lmerge-simps not-less*)

next

case (*step* $x' ys$)

then obtain y **where** $y \in \text{lset } xs$ $x \leq y$ **by** *blast*

moreover from $\langle \text{lsorted } (LCons\ x'\ ys) \rangle \langle x \in \text{lset } ys \rangle \langle x \neq x' \rangle$

have $x' < x$ *lsorted* ys **by**(*auto simp add: lsorted-LCons*)

ultimately show ?*case* **using** $\langle \neg \text{lfinite } xs \rangle$

by *induct*(*auto 4 3 simp add: lmerge-simps* $\langle x \neq x' \rangle$ *not-less intro: step.IH dest:*
in-lset-lmergeD)

qed

lemma *lsorted-lmerge*: $\llbracket \text{lsorted } xs; \text{lsorted } ys \rrbracket \implies \text{lsorted } (\text{lmerge } xs\ ys)$

proof(*coinduction arbitrary: xs ys*)

case (*lsorted* $xs\ ys$)

have ?*lhd* **using** *lsorted*

by(*auto 4 3 simp add: not-null-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*:

$\llbracket \text{lsorted } xs; \text{lsorted } ys; \text{ldistinct } xs; \text{ldistinct } ys \rrbracket$

$\implies \text{ldistinct } (\text{lmerge } xs\ ys)$

by(*coinduction arbitrary: xs ys*)(*auto 4 3 simp add: lhd-lmerge ltl-lmerge not-null-conv*
lsorted-LCons not-less dest!: in-lset-lmergeD dest: antisym)

end

partial-function (*llist*) *hamming'* :: *unit* \Rightarrow *nat llist*

where

hamming' - =

$LCons\ 1\ (\text{lmerge } (\text{lmap } (\text{op } * 2)\ (\text{hamming}'\ ()))\ (\text{lmerge } (\text{lmap } (\text{op } * 3)$

(*hamming'* ()) (lmap (op * 5) (*hamming'* ())))

definition *hamming* :: nat llist
where *hamming* = *hamming'* ()

lemma *lnull-hamming* [*simp*]: \neg *lnull hamming*
unfolding *hamming-def* **by**(subst *hamming'.simps*) *simp*

lemma *hamming-eq-LNil-iff* [*simp*]: *hamming* = LNil \longleftrightarrow False
using *lnull-hamming* **unfolding** *lnull-def* **by** *simp*

lemma *lhd-hamming* [*simp*]: *lhd hamming* = 1
unfolding *hamming-def* **by**(subst *hamming'.simps*) *simp*

lemma *ltl-hamming* [*simp*]:
 ltl hamming = *lmerge* (*lmap* (op * 2) *hamming*) (*lmerge* (*lmap* (op * 3) *hamming*) (*lmap* (op * 5) *hamming*))
unfolding *hamming-def* **by**(subst *hamming'.simps*) *simp*

lemma *hamming-unfold*:
 hamming = LCons 1 (*lmerge* (*lmap* (op * 2) *hamming*) (*lmerge* (*lmap* (op * 3) *hamming*) (*lmap* (op * 5) *hamming*)))
by(rule *llist.expand*) *simp-all*

definition *smooth* :: nat \Rightarrow bool
where *smooth* n \longleftrightarrow ($\forall p$. *prime* p \longrightarrow p *dvd* n \longrightarrow p \leq 5)

lemma *smooth-0* [*simp*]: \neg *smooth* 0
by(auto *simp* add: *smooth-def* intro: *exI*[**where** x=7])

lemma *smooth-Suc0* [*simp*]: *smooth* (*Suc* 0)
by(auto *simp* add: *smooth-def*)

lemma *smooth-gt0*: *smooth* n \Longrightarrow n > 0
by(cases n) *simp-all*

lemma *smooth-ge-Suc0*: *smooth* n \Longrightarrow n \geq *Suc* 0
by(cases n) *simp-all*

lemma *prime-nat-dvdD*: *prime* p \Longrightarrow (n :: nat) *dvd* p \Longrightarrow n = 1 \vee n = p
unfolding *prime-nat-iff* **by** *simp*

lemma *smooth-times* [*simp*]: *smooth* (x * y) \longleftrightarrow *smooth* x \wedge *smooth* y
by(auto *simp* add: *smooth-def* *prime-dvd-mult-iff*)

lemma *smooth2* [*simp*]: *smooth* 2
by(auto *simp* add: *smooth-def* dest: *prime-nat-dvdD*[of 2, *simplified*])

lemma *smooth3* [*simp*]: *smooth* 3

by(*auto simp add: smooth-def dest: prime-nat-dvdD[of 3, simplified]*)

lemma *smooth5* [*simp*]: *smooth 5*

by(*auto simp add: smooth-def dest: prime-nat-dvdD[of 5, simplified]*)

lemma *hamming-in-smooth*: *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 *lfinite hamming*

proof

assume *lfinite hamming*

then obtain *n* **where** *n*: *llength hamming = enat n* **unfolding** *lfinite-conv-llength-enat*

by *fastforce*

have *llength (lmap (op * 3) hamming) \sqcap llength (lmap (op * 5) hamming) \leq llength (lmerge (lmap (op * 3) hamming) (lmap (op * 5) hamming))*

by(*rule llength-lmerge-above*)

hence *llength hamming $\leq \dots$* **by** *simp*

moreover have *llength (lmap (op * 2) hamming) $\sqcap \dots \leq$*

*llength (lmerge (lmap (op * 2) hamming) (lmerge (lmap (op * 3) hamming) (lmap (op * 5) hamming)))*

by(*rule llength-lmerge-above*)

ultimately have *llength hamming $\leq \dots$* **by**(*simp add: inf.absorb1*)

also from *n* **have** $\dots < \text{enat } n$

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 \wedge ldistinct hamming \wedge lset hamming \subseteq $\{1..\}$*

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 ldistinct-lmerge inj-onI*)

thus *lsorted hamming ldistinct hamming* **by** *simp-all*

qed

lemma *smooth-hamming*:

assumes *smooth n*

shows $n \in \text{lset hamming}$

using *assms*

proof(*induction n rule: less-induct*)

have [*simp*]:

*monotone op \leq op \leq (op * 2 :: nat \Rightarrow nat)*

*monotone op \leq op \leq (op * 3 :: nat \Rightarrow nat)*

*monotone op \leq op \leq (op * 5 :: nat \Rightarrow 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* $xs.\ lsorted\ xs \wedge\ ldistinct\ xs \wedge\ lset\ xs = \{n.\ smooth\ n\}$) = *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 *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*: $Old-Datatype.Lim\ (\lambda x.\ \bigcup y.\ f\ x\ y) = (\bigcup y.\ Old-Datatype.Lim\ (\lambda x.\ f\ x\ y))$

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* :: $(('b \Rightarrow ('a, 'b)\ Old-Datatype.dtree) \Rightarrow 'c) \Rightarrow ('a, 'b)\ Old-Datatype.dtree \Rightarrow 'c$

where *DTBranch* $f\ M = (THE\ u.\ \exists x.\ M = Old-Datatype.Lim\ x \wedge\ u = f\ x)$

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* :: $'a \Rightarrow ('c + 'b + 'a, 'd) Old-Datatype.dtree$

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*: $TERMINAL\ a = TERMINAL\ a' \iff a = a'$
and *LINEAR-inject*: $LINEAR\ b\ r = LINEAR\ b'\ r' \iff b = b' \wedge r = r'$
and *BRANCH-inject*: $BRANCH\ c\ rs = BRANCH\ c'\ rs' \iff c = c' \wedge rs = rs'$
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*: $r \subseteq r' \implies LINEAR\ b\ r \subseteq LINEAR\ b\ r'$
by(*simp add: LINEAR-def In1-mono In0-mono Scons-mono*)

lemma *BRANCH-mono*: $(\bigwedge d. rs\ d \subseteq rs'\ d) \implies BRANCH\ c\ rs \subseteq BRANCH\ c\ rs'$
by(*simp add: BRANCH-def In1-mono Scons-mono Lim-mono*)

lemma *LINEAR-UN*: $LINEAR\ b\ (\bigcup x. f\ x) = (\bigcup x. LINEAR\ b\ (f\ x))$
by (*simp add: LINEAR-def In1-UN1 In0-UN1 Scons-UN1-y*)

lemma *BRANCH-UN*: $BRANCH\ b\ (\lambda d. \bigcup x. f\ d\ x) = (\bigcup x. BRANCH\ b\ (\lambda d. f\ d\ x))$
by (*simp add: BRANCH-def Lim-UN1 In1-UN1 In0-UN1 Scons-UN1-y*)

The codatatype universe

coinductive-set *resumption* :: $('c + 'b + 'a, 'd)\ Old-Datatype.dtree\ set$
where
resumption-TERMINAL:
 $TERMINAL\ a \in resumption$
| *resumption-LINEAR*:
 $r \in resumption \implies LINEAR\ b\ r \in resumption$
| *resumption-BRANCH*:

$(\bigwedge d. rs\ d \in \text{resumption}) \implies \text{BRANCH } c\ rs \in \text{resumption}$

17.3 Definition of the codatatype as a type

typedef ('a,'b,'c,'d) *resumption* = *resumption* :: ('c + 'b + 'a, 'd) *Old-Datatype.dtree set*

proof

show *TERMINAL undefined* \in ?*resumption* **by**(blast intro: *resumption.intros*)
qed

Constructors

definition *Terminal* :: 'a \Rightarrow ('a,'b,'c,'d) *resumption*
where *Terminal a* = *Abs-resumption (TERMINAL a)*

definition *Linear* :: 'b \Rightarrow ('a,'b,'c,'d) *resumption* \Rightarrow ('a,'b,'c,'d) *resumption*
where *Linear b r* = *Abs-resumption (LINEAR b (Rep-resumption r))*

definition *Branch* :: 'c \Rightarrow ('d \Rightarrow ('a,'b,'c,'d) *resumption*) \Rightarrow ('a,'b,'c,'d) *resumption*
where *Branch c rs* = *Abs-resumption (BRANCH c ($\lambda d.$ *Rep-resumption (rs d)*))*

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'* \iff *a* = *a'*
and *Linear-inject*: *Linear b r* = *Linear b' r'* \iff *b* = *b' \wedge r = r'*
and *Branch-inject*: *Branch c rs* = *Branch c' rs'* \iff *c* = *c' \wedge 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-iff, 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 ($\lambda d.$ *Rep-resumption (rs d)*)*
by(*simp-all add: Terminal-def Linear-def Branch-def Abs-resumption-inverse resumption.intros Rep-resumption*)

Case operator

definition *case-resumption* :: ('a \Rightarrow 'e) \Rightarrow ('b \Rightarrow ('a,'b,'c,'d) *resumption* \Rightarrow 'e) \Rightarrow

```

                                ('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  $\text{--}(rule\ Linear, simp\ add: Linear-def)$ 
next
case (resumption-BRANCH rs c)
from  $\langle \wedge d. rs\ d \in resumption \rangle$ 
have eq: rs =  $(\lambda d. Rep-resumption\ (Abs-resumption\ (rs\ d)))$ 
by (subst Abs-resumption-inverse) blast+
show ?thesis using  $\langle y = BRANCH\ c\ rs \rangle$ 
by  $\text{--}(rule\ Branch, simp\ add: Branch-def, subst\ eq, simp)$ 
qed
qed

```

lemma *resumption-split*:

```

P (case-resumption t l br r)  $\longleftrightarrow$ 
 $(\forall a. r = Terminal\ a \longrightarrow P\ (t\ a)) \wedge$ 
 $(\forall b\ r'. r = Linear\ b\ r' \longrightarrow P\ (l\ b\ r')) \wedge$ 
 $(\forall c\ rs. r = Branch\ c\ rs \longrightarrow P\ (br\ c\ rs))$ 
by (cases r) simp-all

```

lemma *resumption-split-asm*:

```

P (case-resumption t l br r)  $\longleftrightarrow$ 
 $\neg ((\exists a. r = Terminal\ a \wedge \neg P\ (t\ a)) \vee$ 
 $(\exists b\ r'. r = Linear\ b\ r' \wedge \neg P\ (l\ b\ r')) \vee$ 
 $(\exists c\ rs. r = Branch\ c\ rs \wedge \neg P\ (br\ c\ rs)))$ 
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  $\Rightarrow$  'e
  | Resumption-corec ('a, 'b, 'c, 'd) resumption

```

```

primrec RESUMPTION-corec-aux :: nat  $\Rightarrow$  ('e  $\Rightarrow$  ('a,'b,'c,'d,'e) resumption-corec)
 $\Rightarrow$  'e  $\Rightarrow$  ('c + 'b + 'a,'d) Old-Datatype.dtree

```

where

```

RESUMPTION-corec-aux 0 f e = {}
| RESUMPTION-corec-aux (Suc n) f e =
  (case f e of Terminal-corec a  $\Rightarrow$  TERMINAL a
  | Linear-corec b e'  $\Rightarrow$  LINEAR b (RESUMPTION-corec-aux n f e')
  | Branch-corec c es  $\Rightarrow$  BRANCH c  $(\lambda d. RESUMPTION-corec-aux\ n\ f$ 
(es d))
  | Resumption-corec r  $\Rightarrow$  Rep-resumption r)

```

```

definition RESUMPTION-corec :: ('e  $\Rightarrow$  ('a,'b,'c,'d,'e) resumption-corec)  $\Rightarrow$  'e
 $\Rightarrow$  ('c + 'b + 'a,'d) Old-Datatype.dtree

```

where

```

RESUMPTION-corec f e = ( $\bigcup$  n. RESUMPTION-corec-aux n f e)

```

lemma *RESUMPTION-corec* [nitpick-simp]:
RESUMPTION-corec $f\ e =$
(case $f\ e$ of *Terminal-corec* $a \Rightarrow$ *TERMINAL* a
| *Linear-corec* $b\ e' \Rightarrow$ *LINEAR* b (*RESUMPTION-corec* $f\ e'$)
| *Branch-corec* $c\ es \Rightarrow$ *BRANCH* c ($\lambda d.$ *RESUMPTION-corec* f ($es\ d$))
| *Resumption-corec* $r \Rightarrow$ *Rep-resumption* r)
(is $?lhs = ?rhs$)
proof
show $?lhs \subseteq ?rhs$ **unfolding** *RESUMPTION-corec-def*
proof(rule *UN-least*)
fix n
show *RESUMPTION-corec-aux* $n\ f\ e$
 \subseteq (case $f\ e$ of *Terminal-corec* $a \Rightarrow$ *TERMINAL* a
| *Linear-corec* $b\ e' \Rightarrow$ *LINEAR* b ($\bigcup n.$ *RESUMPTION-corec-aux* $n\ f\ e'$)
| *Branch-corec* $c\ es \Rightarrow$ *BRANCH* c ($\lambda d.$ $\bigcup n.$ *RESUMPTION-corec-aux* n
 f ($es\ d$))
| *Resumption-corec* $r \Rightarrow$ *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 $?rhs \subseteq ?lhs$ **unfolding** *RESUMPTION-corec-def*
apply(*simp split: resumption-corec.split add: LINEAR-UN BRANCH-UN*)
by *safe*(rule-tac $a = \text{Suc } n$ **for** n **in** *UN-I*, rule *UNIV-I*, *simp*)
qed

lemma *RESUMPTION-corec-type*: *RESUMPTION-corec* $f\ e \in$ *resumption*
proof –
have $\exists x.$ *RESUMPTION-corec* $f\ e =$ *RESUMPTION-corec* $f\ x$ **by** *blast*
thus $?thesis$
proof *coinduct*
case (*resumption* x)
then obtain e **where** $x =$ *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* :: $('e \Rightarrow ('a, 'b, 'c, 'd, 'e)$ *resumption-corec*) $\Rightarrow 'e \Rightarrow$
 $('a, 'b, 'c, 'd)$ *resumption*
where

$resumption\text{-corec } f e = Abs\text{-resumption } (RESUMPTION\text{-corec } f e)$

lemma *resumption-corec*:

$resumption\text{-corec } f e =$

(*case* $f e$ of *Terminal-corec* $a \Rightarrow Terminal a$

| *Linear-corec* $b e' \Rightarrow Linear b (resumption\text{-corec } f e')$

| *Branch-corec* $c es \Rightarrow Branch c (\lambda d. resumption\text{-corec } f (es d))$

| *Resumption-corec* $r \Rightarrow r$)

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* \Rightarrow ('c+'b+'a,
'd) *Old-Datatype.dtree* \Rightarrow *bool*

where

EqTERMINAL: *Eq-RESUMPTION* (*TERMINAL* a) (*TERMINAL* a)

| *EqLINEAR*: *Eq-RESUMPTION* $r r' \Longrightarrow Eq\text{-RESUMPTION } (LINEAR b r)$
(*LINEAR* $b r'$)

| *EqBRANCH*: ($\bigwedge d. Eq\text{-RESUMPTION } (rs d) (rs' d)$) $\Longrightarrow Eq\text{-RESUMPTION}$
(*BRANCH* $c rs$) (*BRANCH* $c rs'$)

lemma *Eq-RESUMPTION-implies-ntrunc-equality*:

$Eq\text{-RESUMPTION } r r' \Longrightarrow ntrunc k r = ntrunc k r'$

proof(*induct* k *arbitrary*: $r r'$ *rule*: *less-induct*)

case (*less* k)

note $IH = \langle \bigwedge k' r r'. \llbracket k' < k; Eq\text{-RESUMPTION } r r' \rrbracket \Longrightarrow ntrunc k' r = ntrunc$
 $k' r' \rangle$

from ($Eq\text{-RESUMPTION } r r'$) **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** $f = Old\text{-Datatype.Lim}$])

```

    apply(rule ext)
    apply(simp add: IH)
  done
qed
qed

```

```

lemma Eq-RESUMPTION-refl:
  assumes  $r \in \text{resumption}$ 
  shows Eq-RESUMPTION  $r r$ 
proof -
  def  $r' == r$ 
  with assms have  $r = r' \wedge r \in \text{resumption}$  by auto
  thus Eq-RESUMPTION  $r r'$ 
  proof(coinduct)
    case (Eq-RESUMPTION  $r r'$ )
    hence [simp]:  $r = r'$  and  $r \in \text{resumption}$  by auto
    from  $\langle r \in \text{resumption} \rangle$  show ?case
    by(cases rule: resumption.cases) auto
  qed
qed

```

```

lemma Eq-RESUMPTION-into-resumption:
  assumes Eq-RESUMPTION  $r r$ 
  shows  $r \in \text{resumption}$ 
using assms
proof(coinduct)
  case resumption thus ?case by cases auto
qed

```

```

lemma Eq-RESUMPTION-eq:
  Eq-RESUMPTION  $r r' \longleftrightarrow r = r' \wedge r \in \text{resumption}$ 
proof(rule iffI)
  assume Eq-RESUMPTION  $r r'$ 
  hence  $\bigwedge k. \text{ntrunc } k r = \text{ntrunc } k r'$  by(rule Eq-RESUMPTION-implies-ntrunc-equality)
  hence  $r = r'$  by(rule ntrunc-equality)
  moreover with  $\langle \text{Eq-RESUMPTION } r r' \rangle$  have  $r \in \text{resumption}$ 
  by(auto intro: Eq-RESUMPTION-into-resumption)
  ultimately show  $r = r' \wedge r \in \text{resumption} ..$ 
next
  assume  $r = r' \wedge r \in \text{resumption}$ 
  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:  $\bigwedge r r'. X r r' \implies$ 
    ( $\exists a. r = \text{TERMINAL } a \wedge r' = \text{TERMINAL } a$ )  $\vee$ 
    ( $\exists R R' b. r = \text{LINEAR } b R \wedge r' = \text{LINEAR } b R' \wedge (X R R' \vee$ 

```

$Eq\text{-RESUMPTION } R R')$ \vee
 $(\exists rs rs' c. r = \text{BRANCH } c rs \wedge r' = \text{BRANCH } c rs' \wedge (\forall d. X (rs d)$
 $(rs' d) \vee Eq\text{-RESUMPTION } (rs d) (rs' d)))$
shows $r = r'$
proof –
from $\langle X r r' \rangle$ **have** $Eq\text{-RESUMPTION } r r'$
by $(\text{coinduct})(\text{rule step})$
thus $?thesis$ **by** $(\text{simp add: } Eq\text{-RESUMPTION-eq})$
qed

lemma *resumption-equalityI* [*consumes 1, case-names Eq-resumption, case-conclusion*
Eq-resumption EqTerminal EqLinear EqBranch]:

assumes $X r r'$
and *step*: $\bigwedge r r'. X r r' \implies$
 $(\exists a. r = \text{Terminal } a \wedge r' = \text{Terminal } a) \vee$
 $(\exists R R' b. r = \text{Linear } b R \wedge r' = \text{Linear } b R' \wedge (X R R' \vee R = R')) \vee$
 $(\exists rs rs' c. r = \text{Branch } c rs \wedge r' = \text{Branch } c rs' \wedge (\forall d. X (rs d) (rs'$
 $d) \vee rs d = rs' d))$
shows $r = r'$
proof –
def $M == \text{Rep-resumption } r$ **and** $N == \text{Rep-resumption } r'$
with $\langle X r r' \rangle$ **have** $\exists r r'. M = \text{Rep-resumption } r \wedge N = \text{Rep-resumption } r' \wedge$
 $X r r'$ **by** *blast*
hence $M = N$
proof $(\text{coinduct rule: } Eq\text{-RESUMPTION-I})$
case $(Eq\text{-RESUMPTION } M N)$
then obtain $r r'$ **where** $[\text{simp}]$: $M = \text{Rep-resumption } r \wedge N = \text{Rep-resumption}$
 r'
and $X r r'$ **by** *blast*
{ assume $\exists a. r = \text{Terminal } a \wedge r' = \text{Terminal } a$
hence $?EqTerminal$ **by** $(\text{auto simp add: } \text{Rep-resumption-constructors})$
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 $(\text{clarsimp simp add: } \text{Rep-resumption-constructors } Eq\text{-RESUMPTION-eq}$
 $\text{Rep-resumption-inject } \text{Rep-resumption})$
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 $(\text{clarsimp simp add: } \text{Rep-resumption-constructors } Eq\text{-RESUMPTION-eq}$
 $\text{Rep-resumption-inject } \text{Rep-resumption})$
hence $?case$ **by** *blast* **}**
ultimately show $?case$ **using** *step* $[OF \langle X r r' \rangle]$ **by** *blast*
qed
thus $?thesis$ **unfolding** $M\text{-def } N\text{-def}$ **by** $(\text{simp add: } \text{Rep-resumption-inject})$
qed

Finality of *resumption*: Uniqueness of functions defined by corecursion.

lemma *equals-RESUMPTION-corec*:

assumes $h: \bigwedge x. h\ x = (\text{case } f\ x \text{ of } \text{Terminal-corec } a \Rightarrow \text{TERMINAL } a$
 $\quad \quad \quad | \text{Linear-corec } b\ x' \Rightarrow \text{LINEAR } b\ (h\ x')$
 $\quad \quad \quad | \text{Branch-corec } c\ xs \Rightarrow \text{BRANCH } c\ (\lambda d. h\ (xs\ d))$
 $\quad \quad \quad | \text{Resumption-corec } r \Rightarrow \text{Rep-resumption } r)$

shows $h = \text{RESUMPTION-corec } f$

proof

fix x

def $h' == \text{RESUMPTION-corec } f$

have $h': \bigwedge x. h'\ x = (\text{case } f\ x \text{ of } \text{Terminal-corec } a \Rightarrow \text{TERMINAL } a$
 $\quad \quad \quad | \text{Linear-corec } b\ x' \Rightarrow \text{LINEAR } b\ (h'\ x')$
 $\quad \quad \quad | \text{Branch-corec } c\ xs \Rightarrow \text{BRANCH } c\ (\lambda d. h'\ (xs\ d))$
 $\quad \quad \quad | \text{Resumption-corec } r \Rightarrow \text{Rep-resumption } r)$

unfolding h' -def **by** (*simp add: RESUMPTION-corec*)

def $M == h\ x$ **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 $h: \bigwedge x. h\ x = (\text{case } f\ x \text{ of } \text{Terminal-corec } a \Rightarrow \text{Terminal } a$
 $\quad \quad \quad | \text{Linear-corec } b\ x' \Rightarrow \text{Linear } b\ (h\ x')$
 $\quad \quad \quad | \text{Branch-corec } c\ xs \Rightarrow \text{Branch } c\ (\lambda d. h\ (xs\ d))$
 $\quad \quad \quad | \text{Resumption-corec } r \Rightarrow r)$

shows $h = \text{resumption-corec } f$

```

proof(rule ext)
  fix x
  { fix x
    from h[of x]
    have Rep-resumption (h x) =
      (case f x of Terminal-corec a  $\Rightarrow$  TERMINAL a
        | Linear-corec b x'  $\Rightarrow$  LINEAR b (Rep-resumption (h x^))
        | Branch-corec c xs  $\Rightarrow$  BRANCH c ( $\lambda$ d. Rep-resumption (h (xs d)))
        | Resumption-corec r  $\Rightarrow$  Rep-resumption r)
    by(auto split: resumption-corec.split simp add: Rep-resumption-constructors)
  }
  hence eq: ( $\lambda$ x. 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
  TLList-CCPO-Examples
  Koenigslemma
  LMirror
  Hamming-Stream
  Resumption

```

```

begin

```

```

end

```