

# Coinductive

Andreas Lochbihler  
with contributions by Johannes Hölzl

January 31, 2023

## 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

<b>1</b>	<b>Extended natural numbers as a codatatype</b>	<b>2</b>
1.1	Case operator . . . . .	2
1.2	Corecursion for <i>enat</i> . . . . .	3
1.3	Less as greatest fixpoint . . . . .	6
1.4	Equality as greatest fixpoint . . . . .	7
1.5	Uniqueness of corecursion . . . . .	7
1.6	Setup for <code>partial_function</code> . . . . .	8
1.7	Misc. . . . .	11
<b>2</b>	<b>Coinductive lists and their operations</b>	<b>12</b>
2.1	Auxiliary lemmata . . . . .	12
2.2	Type definition . . . . .	12
2.3	Properties of predefined functions . . . . .	15
2.4	The subset of finite lazy lists <i>lfinite</i> . . . . .	17

2.5	Concatenating two lists: <i>lappend</i> . . . . .	17
2.6	The prefix ordering on lazy lists: <i>lprefix</i> . . . . .	19
2.7	Setup for <i>partial_function</i> . . . . .	21
2.8	Monotonicity and continuity of already defined functions . . .	25
2.9	More function definitions . . . . .	27
2.10	Converting ordinary lists to lazy lists: <i>llist-of</i> . . . . .	30
2.11	Converting finite lazy lists to ordinary lists: <i>list-of</i> . . . . .	31
2.12	The length of a lazy list: <i>llength</i> . . . . .	32
2.13	Taking and dropping from lazy lists: <i>ltake</i> , <i>ldropn</i> , and <i>ldrop</i> .	34
2.14	Taking the <i>n</i> -th element of a lazy list: <i>lnth</i> . . . . .	42
2.15	<i>iterates</i> . . . . .	44
2.16	More on the prefix ordering on lazy lists: $(\sqsubseteq)$ and <i>lstrict-prefix</i>	45
2.17	Length of the longest common prefix . . . . .	47
2.18	Ziping two lazy lists to a lazy list of pairs <i>lzip</i> . . . . .	48
2.19	Taking and dropping from a lazy list: <i>ltakeWhile</i> and <i>ldropWhile</i>	51
2.20	<i>llist-all2</i> . . . . .	55
2.21	The last element <i>llast</i> . . . . .	60
2.22	Distinct lazy lists <i>ldistinct</i> . . . . .	61
2.23	Sortedness <i>lsorted</i> . . . . .	62
2.24	Lexicographic order on lazy lists: <i>llexord</i> . . . . .	65
2.25	The filter functional on lazy lists: <i>lfilter</i> . . . . .	67
2.26	Concatenating all lazy lists in a lazy list: <i>lconcat</i> . . . . .	70
2.27	Sublist view of a lazy list: <i>lnths</i> . . . . .	73
2.28	<i>lsum-list</i> . . . . .	75
2.29	Alternative view on ' <i>a llist</i> as datatype with constructors <i>llist-of</i> and <i>inf-llist</i> . . . . .	75
2.30	Setup for lifting and transfer . . . . .	77
	2.30.1 Relator and predicator properties . . . . .	77
	2.30.2 Transfer rules for the Transfer package . . . . .	78
<b>3</b>	<b>Instantiation of the order type classes for lazy lists</b>	<b>80</b>
3.1	Instantiation of the order type class . . . . .	80
3.2	Prefix ordering as a lower semilattice . . . . .	81
<b>4</b>	<b>Infinite lists as a codatatype</b>	<b>82</b>
4.1	Lemmas about operations from <i>HOL-Library.Stream</i> . . . . .	83
4.2	Link ' <i>a stream</i> to ' <i>a llist</i> . . . . .	84
4.3	Link ' <i>a stream</i> with <i>nat</i> $\Rightarrow$ ' <i>a</i> . . . . .	88
4.4	Function iteration <i>siterate</i> and <i>sconst</i> . . . . .	88
4.5	Counting elements . . . . .	89
4.6	First index of an element . . . . .	90
4.7	<i>stakeWhile</i> . . . . .	90

<b>5</b>	<b>Terminated coinductive lists and their operations</b>	<b>91</b>
5.1	Auxiliary lemmas . . . . .	91
5.2	Type definition . . . . .	91
5.3	Code generator setup . . . . .	92
5.4	Connection with <i>'a llist</i> . . . . .	94
5.5	Library function definitions . . . . .	97
5.6	<i>tfinite</i> . . . . .	98
5.7	The terminal element <i>terminal</i> . . . . .	98
5.8	<i>tmap</i> . . . . .	98
5.9	Appending two terminated lazy lists <i>tappend</i> . . . . .	99
5.10	Appending a terminated lazy list to a lazy list <i>lappendt</i> . . . . .	99
5.11	Filtering terminated lazy lists <i>tfilter</i> . . . . .	100
5.12	Concatenating a terminated lazy list of lazy lists <i>tconcat</i> . . . . .	101
5.13	<i>tllist-all2</i> . . . . .	101
5.14	From a terminated lazy list to a lazy list <i>llist-of-tllist</i> . . . . .	104
5.15	The nth element of a terminated lazy list <i>tnth</i> . . . . .	105
5.16	The length of a terminated lazy list <i>tlength</i> . . . . .	105
5.17	<i>tdropn</i> . . . . .	106
5.18	<i>tset</i> . . . . .	106
5.19	Setup for Lifting/Transfer . . . . .	107
	5.19.1 Relator and predicator properties . . . . .	107
	5.19.2 Transfer rules for the Transfer package . . . . .	107
<b>6</b>	<b>Setup for Isabelle's quotient package for lazy lists</b>	<b>109</b>
6.1	Rules for the Quotient package . . . . .	110
<b>7</b>	<b>Setup for Isabelle's quotient package for terminated lazy lists</b>	<b>112</b>
7.1	Rules for the Quotient package . . . . .	112
<b>8</b>	<b>Code generator setup to implement lazy lists lazily</b>	<b>115</b>
8.1	Lazy lists . . . . .	116
<b>9</b>	<b>Code generator setup to implement terminated lazy lists lazily</b>	<b>122</b>
<b>10</b>	<b>CCPO topologies</b>	<b>126</b>
10.1	The filter <i>at'</i> . . . . .	127
10.2	The type class <i>ccpo-topology</i> . . . . .	127
10.3	Instances for <i>ccpo-topologys</i> and continuity theorems . . . . .	129
<b>11</b>	<b>A CCPO topology on lazy lists with examples</b>	<b>129</b>
11.1	Continuity and closedness of predefined constants . . . . .	130
11.2	Define <i>lfilter</i> as continuous extension . . . . .	133
11.3	Define <i>lconcat</i> as continuous extension . . . . .	134

11.4	Define <i>ldropWhile</i> as continuous extension . . . . .	135
11.5	Define <i>ldrop</i> as continuous extension . . . . .	135
11.6	Define more functions on lazy lists as continuous extensions . . . . .	136
<b>12</b>	<b>Ccpo structure for terminated lazy lists</b>	<b>141</b>
12.1	The ccpo structure . . . . .	142
12.2	Continuity of predefined constants . . . . .	145
12.3	Definition of recursive functions . . . . .	147
<b>13</b>	<b>Example definitions using the CCPO structure on terminated lazy lists</b>	<b>147</b>
<b>14</b>	<b>Example: Koenig's lemma</b>	<b>148</b>
<b>15</b>	<b>Definition of the function lmirror</b>	<b>150</b>
<b>16</b>	<b>The Hamming stream defined as a least fixpoint</b>	<b>152</b>
<b>17</b>	<b>Manual construction of a resumption codatatype</b>	<b>157</b>
17.1	Auxiliary definitions and lemmata similar to <i>HOL-Library.Old-Datatype</i>	157
17.2	Definition for the codatatype universe . . . . .	158
17.3	Definition of the codatatype as a type . . . . .	160

# 1 Extended natural numbers as a codatatype

**theory** *Coinductive-Nat* **imports**

*HOL-Library.Extended-Nat*

*HOL-Library.Complete-Partial-Order2*

**begin**

**lemma** *inj-enat* [*simp*]: *inj-on enat A*

*<proof>*

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

*<proof>*

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

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

**lemma** *enat-add-sub-same*:

**fixes** *a b :: enat* **shows**  $a \neq \infty \implies a + b - a = b$

*<proof>*

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

*<proof>*

**lemma** *enat-min-eq-0-iff*:

**fixes** *a b :: enat*

**shows**  $\text{min } a \ b = 0 \iff a = 0 \vee b = 0$

*<proof>*

**lemma** *enat-le-plus-same*:  $x \leq (x :: \text{enat}) + y \iff x \leq y + x$

*<proof>*

**lemma** *the-enat-0* [*simp*]: *the-enat 0 = 0*

*<proof>*

**lemma** *the-enat-eSuc*:  $n \neq \infty \implies \text{the-enat} (\text{eSuc } n) = \text{Suc} (\text{the-enat } n)$

*<proof>*

**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>*

## 1.1 Case operator

**lemma** *enat-coexhaust*:

**obtains**  $(0) \ n = 0$

|  $(\text{eSuc}) \ n' \ \text{where } n = \text{eSuc } n'$

$\langle proof \rangle$

**locale** *co* **begin**

**free-constructors** (*plugins del: code*) *case-enat for*

*0::enat*

| *eSuc epred*

**where**

*epred 0 = 0*

$\langle proof \rangle$

**end**

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

$\langle proof \rangle$

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

$\langle proof \rangle$

**lemma** *neq-zero-conv-eSuc*:  $n \neq 0 \longleftrightarrow (\exists n'. n = eSuc n')$

$\langle proof \rangle$

**lemma** *enat-cocase-cert*:

**assumes** *CASE*  $\equiv$  *co.case-enat c d*

**shows** (*CASE 0*  $\equiv$  *c*) &&& (*CASE (eSuc n)*  $\equiv$  *d n*)

$\langle proof \rangle$

**lemma** *enat-cosplit-asm*:

$P (co.case-enat c d n) = (\neg (n = 0 \wedge \neg P c \vee (\exists m. n = eSuc m \wedge \neg P (d m))))$

$\langle proof \rangle$

**lemma** *enat-cosplit*:

$P (co.case-enat c d n) = ((n = 0 \longrightarrow P c) \wedge (\forall m. n = eSuc m \longrightarrow P (d m)))$

$\langle proof \rangle$

**abbreviation** *epred* :: *enat*  $\Rightarrow$  *enat* **where** *epred*  $\equiv$  *co.epred*

**lemma** *epred-0* [*simp*]: *epred 0 = 0*  $\langle proof \rangle$

**lemma** *epred-eSuc* [*simp*]: *epred (eSuc n) = n*  $\langle proof \rangle$

**declare** *co.enat.collapse*[*simp*]

**lemma** *epred-conv-minus*: *epred n = n - 1*

$\langle proof \rangle$

## 1.2 Corecursion for *enat*

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

$\langle proof \rangle$

**lemma** *case-enat-0* [*simp*]:  $\text{case-enat } f \ i \ 0 = f \ 0$   
(*proof*)

**lemma** [*simp*]:  
  **shows** *max-eSuc-eSuc*:  $\text{max } (e\text{Suc } n) \ (e\text{Suc } m) = e\text{Suc } (\text{max } n \ m)$   
  **and** *min-eSuc-eSuc*:  $\text{min } (e\text{Suc } n) \ (e\text{Suc } m) = e\text{Suc } (\text{min } n \ m)$   
(*proof*)

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

**lemma** *numeral-eq-eSuc*:  $\text{numeral } k = e\text{Suc } (\text{epred-numeral } k)$   
(*proof*)

**lemma** *epred-numeral-simps* [*simp*]:  
   $\text{epred-numeral } \text{num.One} = 0$   
   $\text{epred-numeral } (\text{num.Bit0 } k) = \text{numeral } (\text{Num.BitM } k)$   
   $\text{epred-numeral } (\text{num.Bit1 } k) = \text{numeral } (\text{num.Bit0 } k)$   
(*proof*)

**lemma** [*simp*]:  
  **shows** *eq-numeral-eSuc*:  $\text{numeral } k = e\text{Suc } n \longleftrightarrow \text{epred-numeral } k = n$   
  **and** *Suc-eq-numeral*:  $e\text{Suc } n = \text{numeral } k \longleftrightarrow n = \text{epred-numeral } k$   
  **and** *less-numeral-Suc*:  $\text{numeral } k < e\text{Suc } n \longleftrightarrow \text{epred-numeral } k < n$   
  **and** *less-eSuc-numeral*:  $e\text{Suc } n < \text{numeral } k \longleftrightarrow n < \text{epred-numeral } k$   
  **and** *le-numeral-eSuc*:  $\text{numeral } k \leq e\text{Suc } n \longleftrightarrow \text{epred-numeral } k \leq n$   
  **and** *le-eSuc-numeral*:  $e\text{Suc } n \leq \text{numeral } k \longleftrightarrow n \leq \text{epred-numeral } k$   
  **and** *diff-eSuc-numeral*:  $e\text{Suc } n - \text{numeral } k = n - \text{epred-numeral } k$   
  **and** *diff-numeral-eSuc*:  $\text{numeral } k - e\text{Suc } n = \text{epred-numeral } k - n$   
  **and** *max-eSuc-numeral*:  $\text{max } (e\text{Suc } n) \ (\text{numeral } k) = e\text{Suc } (\text{max } n \ (\text{epred-numeral } k))$   
  **and** *max-numeral-eSuc*:  $\text{max } (\text{numeral } k) \ (e\text{Suc } n) = e\text{Suc } (\text{max } (\text{epred-numeral } k) \ n)$   
  **and** *min-eSuc-numeral*:  $\text{min } (e\text{Suc } n) \ (\text{numeral } k) = e\text{Suc } (\text{min } n \ (\text{epred-numeral } k))$   
  **and** *min-numeral-eSuc*:  $\text{min } (\text{numeral } k) \ (e\text{Suc } n) = e\text{Suc } (\text{min } (\text{epred-numeral } k) \ n)$   
(*proof*)

**lemma** *enat-cocase-numeral* [*simp*]:  
   $\text{co.case-enat } a \ f \ (\text{numeral } v) = (\text{let } pv = \text{epred-numeral } v \ \text{in } f \ pv)$   
(*proof*)

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

**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*:  $\text{epred } (\text{enat } m) = \text{enat } (m - 1)$

*<proof>*

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

**lemma** *epred-iadd1*:  $a \neq 0 \implies \text{epred } (a + b) = \text{epred } a + b$

*<proof>*

**lemma** *epred-min* [simp]:  $\text{epred } (\text{min } a \ b) = \text{min } (\text{epred } a) \ (\text{epred } b)$

*<proof>*

**lemma** *epred-le-epredI*:  $n \leq m \implies \text{epred } n \leq \text{epred } m$

*<proof>*

**lemma** *epred-minus-epred* [simp]:

$m \neq 0 \implies \text{epred } n - \text{epred } m = n - m$

*<proof>*

**lemma** *eSuc-epred*:  $n \neq 0 \implies \text{eSuc } (\text{epred } n) = n$

*<proof>*

**lemma** *epred-inject*:  $\llbracket x \neq 0; y \neq 0 \rrbracket \implies \text{epred } x = \text{epred } y \longleftrightarrow x = y$

*<proof>*

**lemma** *monotone-fun-eSuc*[*partial-function-mono*]:

$\text{monotone } (\text{fun-ord } (\lambda y \ x. \ x \leq y)) \ (\lambda y \ x. \ x \leq y) \ (\lambda f. \ \text{eSuc } (f \ x))$

*<proof>*

**partial-function** (*gfp*) *enat-unfold* ::  $('a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow \text{enat}$  **where**

*enat-unfold* [code, nitpick-simp]:

$\text{enat-unfold stop next } a = (\text{if stop } a \ \text{then } 0 \ \text{else } \text{eSuc } (\text{enat-unfold stop next } (\text{next } a)))$

**lemma** *enat-unfold-stop* [simp]:  $\text{stop } a \implies \text{enat-unfold stop next } a = 0$

*<proof>*

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

*<proof>*

**lemma** *enat-unfold-eq-0* [simp]:

$\text{enat-unfold stop next } a = 0 \longleftrightarrow \text{stop } a$

*<proof>*

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



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

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

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

**lemma** *finite-imageD2*:  $\llbracket finite\ (f\ 'A); inj-on\ f\ (A - B); finite\ B \rrbracket \implies finite\ A$   
 ⟨proof⟩

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

### 1.3 Less as greatest fixpoint

**coinductive-set** *Le-enat* ::  $(enat \times enat)\ set$

**where**

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

**lemma** *ile-into-Le-enat*:  
 $m \leq n \implies (m, n) \in Le-enat$   
 ⟨proof⟩

**lemma** *Le-enat-imp-ile-enat-k*:  
 $(m, n) \in Le-enat \implies n < enat\ l \implies m < enat\ l$   
 ⟨proof⟩

**lemma** *enat-less-imp-le*:  
**assumes** *k: !!k. n < enat k*  $\implies m < enat\ k$   
**shows**  $m \leq n$   
 ⟨proof⟩

**lemma** *Le-enat-imp-ile*:  
 $(m, n) \in Le-enat \implies m \leq n$   
 ⟨proof⟩

**lemma** *Le-enat-eq-ile*:  
 $(m, n) \in Le-enat \longleftrightarrow m \leq n$   
 ⟨proof⟩

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

$\bigwedge m n. (m, n) \in X$   
 $\implies m = 0 \vee (\exists m' n' k. m = eSuc\ m' \wedge n = n' + enat\ k \wedge k \neq 0 \wedge ((m', n') \in X \vee m' \leq n'))$

**shows**  $m \leq n$

*<proof>*

**lemma** *enat-le-coinduct* [consumes 1, case-names *le*, case-conclusion *le 0 eSuc*]:

**assumes**  $P: P\ m\ n$

**and step:**

$\bigwedge m n. P\ m\ n$

$\implies (n = 0 \longrightarrow m = 0) \wedge$

$(m \neq 0 \longrightarrow n \neq 0 \longrightarrow (\exists k n'. P\ (epred\ m)\ n' \wedge epred\ n = n' + k) \vee$

$epred\ m \leq epred\ n)$

**shows**  $m \leq n$

*<proof>*

## 1.4 Equality as greatest fixpoint

**lemma** *enat-equalityI* [consumes 1, case-names *Eq-enat*, case-conclusion *Eq-enat zero eSuc*]:

**assumes** *major*:  $(m, n) \in X$

**and step:**

$\bigwedge m n. (m, n) \in X$

$\implies m = 0 \wedge n = 0 \vee (\exists m' n'. m = eSuc\ m' \wedge n = eSuc\ n' \wedge ((m', n') \in X$

$\vee m' = n'))$

**shows**  $m = n$

*<proof>*

**lemma** *enat-coinduct* [consumes 1, case-names *Eq-enat*, case-conclusion *Eq-enat zero eSuc*]:

**assumes** *major*:  $P\ m\ n$

**and step:**  $\bigwedge m n. P\ m\ n$

$\implies (m = 0 \longleftrightarrow n = 0) \wedge$

$(m \neq 0 \longrightarrow n \neq 0 \longrightarrow P\ (epred\ m)\ (epred\ n) \vee epred\ m = epred\ n)$

**shows**  $m = n$

*<proof>*

**lemma** *enat-coinduct2* [consumes 1, case-names *zero eSuc*]:

$\llbracket P\ m\ n; \bigwedge m n. P\ m\ n \implies m = 0 \longleftrightarrow n = 0;$

$\bigwedge m n. \llbracket P\ m\ n; m \neq 0; n \neq 0 \rrbracket \implies P\ (epred\ m)\ (epred\ n) \vee epred\ m = epred\ n \rrbracket$

$\implies m = n$

*<proof>*

## 1.5 Uniqueness of corecursion

**lemma** *enat-unfold-unique*:

**assumes** *h*:  $!!x. h\ x = (\text{if stop } x \text{ then } 0 \text{ else } eSuc\ (h\ (\text{next } x)))$

**shows**  $h\ x = \text{enat-unfold stop next } x$

*<proof>*

## 1.6 Setup for partial\_function

**lemma** *enat-diff-cancel-left*:  $\llbracket m \leq x; m \leq y \rrbracket \implies x - m = y - m \longleftrightarrow x = (y$   
 $:: \text{enat})$   
 $\langle \text{proof} \rangle$

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

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

**lemma** *finite-lessThan-enat-iff*:  
 $\text{finite } \{..<m :: \text{enat}\} \longleftrightarrow m \neq \infty$   
 $\langle \text{proof} \rangle$

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

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

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

**lemma** *Sup-enat-minus1*:  
**assumes**  $m \neq \infty$   
**shows**  $\bigsqcup A - m = \bigsqcup ((\lambda n :: \text{enat}. n - m) \ `A)$   
 $\langle \text{proof} \rangle$

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

**lemma** *Sup-image-eadd2*:  
 $Y \neq \{\} \implies \text{Sup } ((\lambda y :: \text{enat}. x + y) \ `Y) = x + \text{Sup } Y$   
 $\langle \text{proof} \rangle$

**lemma** *mono2mono-eSuc* [*THEN* *lfp.mono2mono*, *cont-intro*, *simp*]:  
**shows** *monotone-eSuc*:  $\text{monotone } (\leq) \ (\leq) \ \text{eSuc}$   
 $\langle \text{proof} \rangle$

**lemma** *mcont2mcont-eSuc* [*THEN* *lfp.mcont2mcont*, *cont-intro*, *simp*]:  
**shows** *mcont-eSuc*:  $\text{mcont } \text{Sup } (\leq) \ \text{Sup } (\leq) \ \text{eSuc}$   
 $\langle \text{proof} \rangle$

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

**shows** *monotone-epred*:  $\text{monotone } (\leq) (\leq) \text{ epred}$

*<proof>*

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

**shows** *mcont-epred*:  $\text{mcont Sup } (\leq) \text{ Sup } (\leq) \text{ epred}$

*<proof>*

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

$\llbracket \text{monotone } \text{orda } \text{ordb } \text{zero}; \bigwedge n. \text{monotone } \text{orda } \text{ordb } (\lambda f. \text{esuc } f \ n) \rrbracket$

$\implies \text{monotone } \text{orda } \text{ordb } (\lambda f. \text{co.case-enat } (\text{zero } f) (\text{esuc } f) \ x)$

*<proof>*

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

$\llbracket \text{mcont } \text{luba } \text{orda } \text{lubb } \text{ordb } \text{zero}; \bigwedge n. \text{mcont } \text{luba } \text{orda } \text{lubb } \text{ordb } (\lambda f. \text{esuc } f \ n) \rrbracket$

$\implies \text{mcont } \text{luba } \text{orda } \text{lubb } \text{ordb } (\lambda f. \text{co.case-enat } (\text{zero } f) (\text{esuc } f) \ x)$

*<proof>*

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

$\text{monotone } (\text{fun-ord } (\leq)) (\leq) f \implies \text{monotone } (\text{fun-ord } (\leq)) (\leq) (\lambda x. \text{eSuc } (f \ x))$

*<proof>*

**lemma** *mono2mono-enat-minus1* [THEN lfp.mono2mono, cont-intro, simp]:

**shows** *monotone-enat-minus1*:  $\text{monotone } (\leq) (\leq) (\lambda n. n - m :: \text{enat})$

*<proof>*

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

**shows** *mcont-enat-minus*:  $m \neq \infty \implies \text{mcont Sup } (\leq) \text{ Sup } (\leq) (\lambda n. n - m :: \text{enat})$

*<proof>*

**lemma** *monotone-eadd1*:  $\text{monotone } (\leq) (\leq) (\lambda x. x + y :: \text{enat})$

*<proof>*

**lemma** *monotone-eadd2*:  $\text{monotone } (\leq) (\leq) (\lambda y. x + y :: \text{enat})$

*<proof>*

**lemma** *mono2mono-eadd*[THEN lfp.mono2mono2, cont-intro, simp]:

**shows** *monotone-eadd*:  $\text{monotone } (\text{rel-prod } (\leq) (\leq)) (\leq) (\lambda(x, y). x + y :: \text{enat})$

*<proof>*

**lemma** *mcont-eadd2*:  $\text{mcont Sup } (\leq) \text{ Sup } (\leq) (\lambda y. x + y :: \text{enat})$

*<proof>*

**lemma** *mcont-eadd1*:  $\text{mcont Sup } (\leq) \text{ Sup } (\leq) (\lambda x. x + y :: \text{enat})$

*<proof>*

**lemma** *mcont2mcont-eadd* [cont-intro, simp]:

$$\llbracket \text{mcont lub ord Sup } (\leq) (\lambda x. f x);$$

$$\text{mcont lub ord Sup } (\leq) (\lambda x. g x) \rrbracket$$

$$\implies \text{mcont lub ord Sup } (\leq) (\lambda x. f x + g x :: \text{enat})$$

$$\langle \text{proof} \rangle$$

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

$$\llbracket \text{monotone (fun-ord } (\leq)) (\leq) f; \text{monotone (fun-ord } (\leq)) (\leq) g \rrbracket$$

$$\implies \text{monotone (fun-ord } (\leq)) (\leq) (\lambda x. f x + g x :: \text{enat})$$

$$\langle \text{proof} \rangle$$

**lemma** *monotone-max-enat1*: *monotone* ( $\leq$ ) ( $\leq$ ) ( $\lambda x. \text{max } x y :: \text{enat}$ )  

$$\langle \text{proof} \rangle$$

**lemma** *monotone-max-enat2*: *monotone* ( $\leq$ ) ( $\leq$ ) ( $\lambda y. \text{max } x y :: \text{enat}$ )  

$$\langle \text{proof} \rangle$$

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

$$:: \text{enat}$$
)  

$$\langle \text{proof} \rangle$$

**lemma** *max-Sup-enat2*:  
**assumes**  $Y \neq \{\}$   
**shows**  $\text{max } x (\text{Sup } Y) = \text{Sup } ((\lambda y :: \text{enat}. \text{max } x y) \text{ ` } Y)$   

$$\langle \text{proof} \rangle$$

**lemma** *max-Sup-enat1*:  
 $Y \neq \{\} \implies \text{max } (\text{Sup } Y) x = \text{Sup } ((\lambda y :: \text{enat}. \text{max } y x) \text{ ` } Y)$   

$$\langle \text{proof} \rangle$$

**lemma** *mcont-max-enat1*: *mcont Sup* ( $\leq$ ) *Sup* ( $\leq$ ) ( $\lambda x. \text{max } x y :: \text{enat}$ )  

$$\langle \text{proof} \rangle$$

**lemma** *mcont-max-enat2*: *mcont Sup* ( $\leq$ ) *Sup* ( $\leq$ ) ( $\lambda y. \text{max } x y :: \text{enat}$ )  

$$\langle \text{proof} \rangle$$

**lemma** *mcont2mcont-max-enat* [*cont-intro, simp*]:  

$$\llbracket \text{mcont lub ord Sup } (\leq) (\lambda x. f x);$$

$$\text{mcont lub ord Sup } (\leq) (\lambda x. g x) \rrbracket$$

$$\implies \text{mcont lub ord Sup } (\leq) (\lambda x. \text{max } (f x) (g x) :: \text{enat})$$

$$\langle \text{proof} \rangle$$

**lemma** *max-enat-partial-function-mono* [*partial-function-mono*]:  

$$\llbracket \text{monotone (fun-ord } (\leq)) (\leq) f; \text{monotone (fun-ord } (\leq)) (\leq) g \rrbracket$$

$$\implies \text{monotone (fun-ord } (\leq)) (\leq) (\lambda x. \text{max } (f x) (g x) :: \text{enat})$$

$$\langle \text{proof} \rangle$$

**lemma** *chain-epredI*:  

$$\text{Complete-Partial-Order.chain } (\leq) Y$$

$\implies \text{Complete-Partial-Order.chain } (\leq) \text{ (epred ' (Y } \cap \{x. x \neq 0\})$   
 $\langle \text{proof} \rangle$

**lemma monotone-enat-le-case:**

**fixes** *bot*  
**assumes** *mono*: *monotone*  $(\leq)$  *ord*  $(\lambda x. f x (eSuc x))$   
**and** *ord*:  $\bigwedge x. \text{ord bot } (f x (eSuc x))$   
**and** *bot*: *ord bot bot*  
**shows** *monotone*  $(\leq)$  *ord*  $(\lambda x. \text{case } x \text{ of } 0 \Rightarrow \text{bot} \mid eSuc x' \Rightarrow f x' x)$   
 $\langle \text{proof} \rangle$

**lemma mcont-enat-le-case:**

**fixes** *bot*  
**assumes** *ccpo*: *class.ccpo lub ord (mk-less ord)*  
**and** *mcont*: *mcont Sup*  $(\leq)$  *lub ord*  $(\lambda x. f x (eSuc x))$   
**and** *ord*:  $\bigwedge x. \text{ord bot } (f x (eSuc x))$   
**shows** *mcont Sup*  $(\leq)$  *lub ord*  $(\lambda x. \text{case } x \text{ of } 0 \Rightarrow \text{bot} \mid eSuc x' \Rightarrow f x' x)$   
 $\langle \text{proof} \rangle$

## 1.7 Misc.

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

$\text{enat } x + y < \text{enat } x + z \longleftrightarrow y < z$   
 $\langle \text{proof} \rangle$

**lemma enat-add1-eq** [*simp*]:  $\text{enat } x + y = \text{enat } x + z \longleftrightarrow y = z$

$\langle \text{proof} \rangle$

**lemma enat-add2-eq** [*simp*]:  $y + \text{enat } x = z + \text{enat } x \longleftrightarrow y = z$

$\langle \text{proof} \rangle$

**lemma enat-less-enat-plusI**:  $x < y \implies \text{enat } x < \text{enat } y + z$

$\langle \text{proof} \rangle$

**lemma enat-less-enat-plusI2**:

$\text{enat } y < z \implies \text{enat } (x + y) < \text{enat } x + z$

$\langle \text{proof} \rangle$

**lemma min-enat1-conv-enat**:  $\bigwedge a b. \text{min } (\text{enat } a) b = \text{enat } (\text{case } b \text{ of } \text{enat } b' \Rightarrow$

$\text{min } a b' \mid \infty \Rightarrow a)$

**and** *min-enat2-conv-enat*:  $\bigwedge a b. \text{min } a (\text{enat } b) = \text{enat } (\text{case } a \text{ of } \text{enat } a' \Rightarrow \text{min}$

$a' b \mid \infty \Rightarrow b)$

$\langle \text{proof} \rangle$

**lemma eSuc-le-iff**:  $eSuc x \leq y \longleftrightarrow (\exists y'. y = eSuc y' \wedge x \leq y')$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

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

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

**end**

## 2 Coinductive lists and their operations

**theory** *Coinductive-List*

**imports**

*HOL-Library.Infinite-Set*

*HOL-Library.Sublist*

*HOL-Library.Simps-Case-Conv*

*Coinductive-Nat*

**begin**

### 2.1 Auxiliary lemmata

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

**lemma** *wlog-linorder-le [consumes 0, case-names le symmetry]*:

**assumes** *le*:  $\bigwedge a\ b :: 'a :: linorder.\ a \leq b \Longrightarrow P\ a\ b$

**and** *sym*:  $P\ b\ a \Longrightarrow P\ a\ b$

**shows**  $P\ a\ b$

⟨proof⟩

### 2.2 Type definition

**codatatype**  $(lset: 'a)\ llist =$

*lnull*:  $LNil$

| *LCons*  $(lhd: 'a)\ (ttl: 'a\ llist)$

**for**

*map*:  $lmap$

*rel*:  $llist\ all2$

**where**

*lhd*  $LNil = undefined$

| *ttl*  $LNil = LNil$

Coiterator setup.

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

**where**

$p\ a \Longrightarrow unfold-llist\ p\ g21\ g22\ a = LNil$  |

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

**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*  
 $= \text{unfold-llist IS-LNIL LHD LTL } x$

*<proof>*

lemmas about generated constants

**lemma** *eq-LConsD*:  $xs = LCons \ y \ ys \implies xs \neq LNil \wedge \text{lhs } xs = y \wedge \text{rhs } xs = ys$

*<proof>*

**lemma**

**shows** *LNil-eq-lmap*:  $LNil = \text{lmap } f \ xs \longleftrightarrow xs = LNil$

**and** *lmap-eq-LNil*:  $\text{lmap } f \ xs = LNil \longleftrightarrow xs = LNil$

*<proof>*

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

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

*<proof>*

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

**lemma** *lmap-eq-LCons-conv*:

$\text{lmap } f \ xs = LCons \ y \ ys \longleftrightarrow$

$(\exists x \ xs'. xs = LCons \ x \ xs' \wedge y = f \ x \wedge ys = \text{lmap } f \ xs')$

*<proof>*

**lemma** *lmap-conv-unfold-llist*:

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

*<proof>*

**lemma** *lmap-unfold-llist*:

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

*b*

*<proof>*

**lemma** *lmap-corec-llist*:

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

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



*<proof>*

**lemma** *unfold-llist-ltl-unroll*:

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

*<proof>*

**lemma** *ltl-unfold-llist*:

$ltl\ (unfold-llist\ IS-LNIL\ LHD\ LTL\ a) =$   
*(if IS-LNIL a then LNil else unfold-llist IS-LNIL LHD LTL (LTL a))*

*<proof>*

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

$unfold-llist\ IS-LNIL\ LHD\ LTL\ b = LCons\ x\ xs \longleftrightarrow$   
 $\neg\ IS-LNIL\ b \wedge x = LHD\ b \wedge xs = unfold-llist\ IS-LNIL\ LHD\ LTL\ (LTL\ b)$

*<proof>*

**lemma** *unfold-llist-id* [simp]:  $unfold-llist\ lnull\ lhd\ ltl\ xs = xs$

*<proof>*

**lemma** *lset-eq-empty* [simp]:  $lset\ xs = \{\} \longleftrightarrow lnull\ xs$

*<proof>*

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

**lemma** *lset-ltl*:  $lset\ (ltl\ xs) \subseteq lset\ xs$

*<proof>*

**lemma** *in-lset-ltlD*:  $x \in lset\ (ltl\ xs) \implies x \in lset\ xs$

*<proof>*

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**  $\bigwedge xs\ y. \llbracket \neg lnull\ xs; y \in lset\ (ltl\ xs); P\ y\ (ltl\ xs) \rrbracket \implies P\ y\ xs$   
**shows**  $P\ x\ xs$

*<proof>*

Test quickcheck setup

**lemma**  $\bigwedge xs. xs = LNil$

**quickcheck**[random, expect=counterexample]

**quickcheck**[exhaustive, expect=counterexample]

*<proof>*

**lemma**  $LCons\ x\ xs = LCons\ x\ xs$

**quickcheck**[narrowing, expect=no-counterexample]

*<proof>*

## 2.3 Properties of predefined functions

**lemmas** *lhd-LCons* = *llist.sel*(1)

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

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

**lemma** *lnull-ltlI* [*simp*]:  $lnull\ xs \implies lnull\ (ltl\ xs)$

*<proof>*

**lemma** *neg-LNil-conv*:  $xs \neq LNil \iff (\exists x\ xs'.\ xs = LCons\ x\ xs')$

*<proof>*

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

*<proof>*

**lemma** *lset-LCons*:

$lset\ (LCons\ x\ xs) = insert\ x\ (lset\ xs)$

*<proof>*

**lemma** *lset-intros*:

$x \in lset\ (LCons\ x\ xs)$

$x \in lset\ xs \implies x \in lset\ (LCons\ x'\ xs)$

*<proof>*

**lemma** *lset-cases* [*elim?*]:

**assumes**  $x \in lset\ xs$

**obtains**  $xs'$  **where**  $xs = LCons\ x\ xs'$

|  $x'\ xs'$  **where**  $xs = LCons\ x'\ xs'\ x \in lset\ xs'$

*<proof>*

**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**  $P\ xs$

*<proof>*

**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**  $P\ xs$

*<proof>*

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

**lemma** *lset-lnull*:  $lnull\ xs \implies lset\ xs = \{\}$

*<proof>*

Alternative definition of *lset* for nitpick

**inductive** *lsetp* :: 'a llist  $\Rightarrow$  'a  $\Rightarrow$  bool

**where**

*lsetp* (LCons x xs) x  
| *lsetp* xs x  $\Longrightarrow$  *lsetp* (LCons x' xs) x

**lemma** *lset-into-lsetp*:

$x \in \text{lset } xs \Longrightarrow \text{lsetp } xs \ x$   
<proof>

**lemma** *lsetp-into-lset*:

$\text{lsetp } xs \ x \Longrightarrow x \in \text{lset } xs$   
<proof>

**lemma** *lset-eq-lsetp* [nitpick-unfold]:

$\text{lset } xs = \{x. \text{lsetp } xs \ x\}$   
<proof>

**hide-const** (open) *lsetp*

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

code setup for *lset*

**definition** *gen-lset* :: 'a set  $\Rightarrow$  'a llist  $\Rightarrow$  'a set

**where** *gen-lset* A xs = A  $\cup$  *lset* xs

**lemma** *gen-lset-code* [code]:

*gen-lset* A LNil = A  
*gen-lset* A (LCons x xs) = *gen-lset* (insert x A) xs  
<proof>

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

*lset* = *gen-lset* {}  
<proof>

**definition** *lmember* :: 'a  $\Rightarrow$  'a llist  $\Rightarrow$  bool

**where** *lmember* x xs  $\longleftrightarrow x \in \text{lset } xs$

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

*lmember* x LNil  $\longleftrightarrow$  False  
*lmember* x (LCons y ys)  $\longleftrightarrow x = y \vee \text{lmember } x \ ys$   
<proof>

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

$x \in \text{lset } xs \longleftrightarrow \text{lmember } x \ xs$   
<proof>

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

## 2.4 The subset of finite lazy lists *lfinite*

**inductive** *lfinite* :: 'a llist  $\Rightarrow$  bool

**where**

*lfinite-LNil*: *lfinite* LNil

| *lfinite-LConsI*: *lfinite* xs  $\Longrightarrow$  *lfinite* (LCons x xs)

**declare** *lfinite-LNil* [iff]

**lemma** *lnull-imp-lfinite* [simp]: *lnull* xs  $\Longrightarrow$  *lfinite* xs

*<proof>*

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

*<proof>*

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

*<proof>*

**lemma** *lfinite-code* [code]:

*lfinite* LNil = True

*lfinite* (LCons x xs) = *lfinite* xs

*<proof>*

**lemma** *lfinite-induct* [consumes 1, case-names LNil LCons]:

**assumes** *lfinite*: *lfinite* xs

**and** LNil:  $\bigwedge xs. \text{lnull } xs \Longrightarrow P \text{ } xs$

**and** LCons:  $\bigwedge xs. \llbracket \text{ lfinite } xs; \neg \text{lnull } xs; P \text{ (ltl } xs) \rrbracket \Longrightarrow P \text{ } xs$

**shows**  $P \text{ } xs$

*<proof>*

**lemma** *lfinite-imp-finite-lset*:

**assumes** *lfinite* xs

**shows** *finite* (lset xs)

*<proof>*

## 2.5 Concatenating two lists: *lappend*

**primcorec** *lappend* :: 'a llist  $\Rightarrow$  '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)

*<proof>*

**lemma** *lnull-lappend* [simp]:

$lnull (lappend\ xs\ ys) \longleftrightarrow lnull\ xs \wedge lnull\ ys$   
(proof)

**lemma** *lappend-eq-LNil-iff*:

$lappend\ xs\ ys = LNil \longleftrightarrow xs = LNil \wedge ys = LNil$   
(proof)

**lemma** *LNil-eq-lappend-iff*:

$LNil = lappend\ xs\ ys \longleftrightarrow xs = LNil \wedge ys = LNil$   
(proof)

**lemma** *lappend-LNil2* [simp]:  $lappend\ xs\ LNil = xs$

(proof)

**lemma shows** *lappend-lnull1*:  $lnull\ xs \implies lappend\ xs\ ys = ys$

**and** *lappend-lnull2*:  $lnull\ ys \implies lappend\ xs\ ys = xs$   
(proof)

**lemma** *lappend-assoc*:  $lappend (lappend\ xs\ ys)\ zs = lappend\ xs (lappend\ ys\ zs)$

(proof)

**lemma** *lmap-lappend-distrib*:

$lmap\ f (lappend\ xs\ ys) = lappend (lmap\ f\ xs) (lmap\ f\ ys)$   
(proof)

**lemma** *lappend-snocL1-conv-LCons2*:

$lappend (lappend\ xs (LCons\ y\ LNil))\ ys = lappend\ xs (LCons\ y\ ys)$   
(proof)

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

(proof)

**lemma** *lfinite-lappend* [simp]:

$lfinite (lappend\ xs\ ys) \longleftrightarrow lfinite\ xs \wedge lfinite\ ys$   
(is ?lhs  $\longleftrightarrow$  ?rhs)  
(proof)

**lemma** *lappend-inf*:  $\neg lfinite\ xs \implies lappend\ xs\ ys = xs$

(proof)

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

$lfinite (lmap\ f\ xs) = lfinite\ xs$   
(is ?lhs  $\longleftrightarrow$  ?rhs)  
(proof)

**lemma** *lset-lappend-lfinite* [simp]:

$lfinite\ xs \implies lset (lappend\ xs\ ys) = lset\ xs \cup lset\ ys$

*<proof>*

**lemma** *lset-lappend*:  $lset (lappend\ xs\ ys) \subseteq lset\ xs \cup lset\ ys$   
*<proof>*

**lemma** *lset-lappend1*:  $lset\ xs \subseteq lset (lappend\ xs\ ys)$   
*<proof>*

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

**lemma** *in-lset-lappend-iff*:  $x \in lset (lappend\ xs\ ys) \iff x \in lset\ xs \vee lfinite\ xs \wedge x \in lset\ ys$   
*<proof>*

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

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

## 2.6 The prefix ordering on lazy lists: *lprefix*

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

**where**

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

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

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

**and** *step*:

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

**shows**  $xs \sqsubseteq ys$

*<proof>*

**lemma** *lprefix-coinduct* [*consumes 1*, *case-names lprefix*, *case-conclusion lprefix LNil LCons*, *coinduct pred: lprefix*]:

**assumes** *major*:  $P\ xs\ ys$

**and** *step*:  $\bigwedge xs\ ys.\ P\ xs\ ys$

$\implies (lnull\ ys \longrightarrow lnull\ xs) \wedge$

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

**shows**  $xs \sqsubseteq ys$   
*<proof>*

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

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

**lemma** *lprefix-lnull*:  $lnull\ ys \implies xs \sqsubseteq ys \longleftrightarrow lnull\ xs$   
*<proof>*

**lemma** *lnull-lprefix*:  $lnull\ xs \implies lprefix\ xs\ ys$   
*<proof>*

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

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

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

**lemma** *lprefix-ltlI*:  $xs \sqsubseteq ys \implies ltl\ xs \sqsubseteq ltl\ ys$   
*<proof>*

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

**lemma** *lprefix-lhdD*:  $\llbracket xs \sqsubseteq ys; \neg lnull\ xs \rrbracket \implies lhd\ xs = lhd\ ys$   
*<proof>*

**lemma** *lprefix-lnullD*:  $\llbracket xs \sqsubseteq ys; lnull\ ys \rrbracket \implies lnull\ xs$   
*<proof>*

**lemma** *lprefix-not-lnullD*:  $\llbracket xs \sqsubseteq ys; \neg lnull\ xs \rrbracket \implies \neg lnull\ ys$   
*<proof>*

**lemma** *lprefix-expand*:  
 $(\neg lnull\ xs \implies \neg lnull\ ys \wedge lhd\ xs = lhd\ ys \wedge ltl\ xs \sqsubseteq ltl\ ys) \implies xs \sqsubseteq ys$   
*<proof>*

**lemma** *lprefix-antisym*:  
 $\llbracket xs \sqsubseteq ys; ys \sqsubseteq xs \rrbracket \implies xs = ys$   
 <proof>

**lemma** *lprefix-trans* [*trans*]:  
 $\llbracket xs \sqsubseteq ys; ys \sqsubseteq zs \rrbracket \implies xs \sqsubseteq zs$   
 <proof>

**lemma** *preorder-lprefix* [*cont-intro*]:  
*class.preorder* ( $\sqsubseteq$ ) (*mk-less* ( $\sqsubseteq$ ))  
 <proof>

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

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

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

**lemma** *lprefix-lappend*:  $xs \sqsubseteq lappend\ xs\ ys$   
 <proof>

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

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

## 2.7 Setup for partial\_function

**primcorec** *lSup* :: 'a llist set  $\Rightarrow$  'a llist

**where**

*lSup* *A* =  
 (if  $\forall x \in A. lnull\ x$  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**  $lnull-lSup [simp]$ :  $lnull (lSup A) \longleftrightarrow (\forall x \in A. lnull x)$   
 $\langle proof \rangle$

**lemma**  $lhd-lSup [simp]$ :  $\exists x \in A. \neg lnull x \implies lhd (lSup A) = (THE x. x \in lhd (A \cap \{xs. \neg lnull xs\}))$   
 $\langle proof \rangle$

**lemma**  $ltl-lSup [simp]$ :  $ltl (lSup A) = lSup (ltl (A \cap \{xs. \neg lnull xs\}))$   
 $\langle proof \rangle$

**lemma**  $lhd-lSup-eq$ :  
**assumes**  $chain$ :  $Complete-Partial-Order.chain (\sqsubseteq) Y$   
**shows**  $\llbracket xs \in Y; \neg lnull xs \rrbracket \implies lhd (lSup Y) = lhd xs$   
 $\langle proof \rangle$

**lemma**  $lSup-empty [simp]$ :  $lSup \{\} = LNil$   
 $\langle proof \rangle$

**lemma**  $lSup-singleton [simp]$ :  $lSup \{xs\} = xs$   
 $\langle proof \rangle$

**lemma**  $LCons-image-Int-not-lnull$ :  $(LCons x (A \cap \{xs. \neg lnull xs\})) = LCons x A$   
 $\langle proof \rangle$

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

**lemma**  $lSup-eq-LCons-iff$ :  
 $lSup Y = LCons x xs \longleftrightarrow (\exists x \in Y. \neg lnull x) \wedge x = (THE x. x \in lhd (Y \cap \{xs. \neg lnull xs\})) \wedge xs = lSup (ltl (Y \cap \{xs. \neg lnull xs\}))$   
 $\langle proof \rangle$

**lemma**  $lSup-insert-LNil$ :  $lSup (insert LNil Y) = lSup Y$   
 $\langle proof \rangle$

**lemma**  $lSup-minus-LNil$ :  $lSup (Y - \{LNil\}) = lSup Y$   
 $\langle proof \rangle$

**lemma**  $chain-lprefix-ltl$ :  
**assumes**  $chain$ :  $Complete-Partial-Order.chain (\sqsubseteq) A$   
**shows**  $Complete-Partial-Order.chain (\sqsubseteq) (ltl (A \cap \{xs. \neg lnull xs\}))$   
 $\langle proof \rangle$

**lemma**  $lSup-finite-prefixes$ :  $lSup \{ys. ys \sqsubseteq xs \wedge lfinite ys\} = xs$  (is  $lSup (?C xs) = -$ )

*<proof>*

**lemma** *lSup-finite-gen-prefixes*:

**assumes**  $zs \sqsubseteq xs$  *lfinite*  $zs$

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

*<proof>*

**lemma** *lSup-strict-prefixes*:

$\neg lfinite\ xs \implies lSup \{ys. ys \sqsubseteq xs \wedge ys \neq xs\} = xs$

(**is**  $- \implies lSup\ (?C\ xs) = -$ )

*<proof>*

**lemma** *chain-lprefix-lSup*:

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

$\implies xs \sqsubseteq lSup\ A$

*<proof>*

**lemma** *chain-lSup-lprefix*:

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

$\implies lSup\ A \sqsubseteq zs$

*<proof>*

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

*<proof>*

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

**lemma** *lList-partial-function-definitions*:

*partial-function-definitions*  $(\sqsubseteq)$   $lSup$

*<proof>*

**interpretation** *lList*: *partial-function-definitions*  $(\sqsubseteq)$   $lSup$

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

*<proof>*

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

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

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

*<proof>*

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

**lemma** *lprefixes-chain*:

*Complete-Partial-Order.chain*  $(\sqsubseteq)$   $\{ys. lprefix\ ys\ xs\}$

*<proof>*

**lemma** *lList-gen-induct*:

**assumes** *adm*: *ccpo.admissible lSup* ( $\sqsubseteq$ ) *P*  
**and** *step*:  $\exists zs. zs \sqsubseteq xs \wedge \text{lfinite } zs \wedge (\forall ys. zs \sqsubseteq ys \longrightarrow ys \sqsubseteq xs \longrightarrow \text{lfinite } ys \longrightarrow P \text{ } ys)$   
**shows** *P xs*  
 <proof>

**lemma** *lList-induct* [*case-names adm LNil LCons, induct type: lList*]:  
**assumes** *adm*: *ccpo.admissible lSup* ( $\sqsubseteq$ ) *P*  
**and** *LNil*: *P LNil*  
**and** *LCons*:  $\bigwedge x xs. [\text{lfinite } xs; P \text{ } xs] \Longrightarrow P (LCons \ x \ xs)$   
**shows** *P xs*  
 <proof>

**lemma** *LCons-mono* [*partial-function-mono, cont-intro*]:  
*mono-lList A*  $\Longrightarrow$  *mono-lList* ( $\lambda f. LCons \ x \ (A \ f)$ )  
 <proof>

**lemma** *mono2mono-LCons* [*THEN lList.mono2mono, simp, cont-intro*]:  
**shows** *monotone-LCons*: *monotone* ( $\sqsubseteq$ ) ( $\sqsubseteq$ ) (*LCons x*)  
 <proof>

**lemma** *mcont2mcont-LCons* [*THEN lList.mcont2mcont, simp, cont-intro*]:  
**shows** *mcont-LCons*: *mcont lSup* ( $\sqsubseteq$ ) *lSup* ( $\sqsubseteq$ ) (*LCons x*)  
 <proof>

**lemma** *mono2mono-ltl* [*THEN lList.mono2mono, simp, cont-intro*]:  
**shows** *monotone-ltl*: *monotone* ( $\sqsubseteq$ ) ( $\sqsubseteq$ ) *ltl*  
 <proof>

**lemma** *cont-ltl*: *cont lSup* ( $\sqsubseteq$ ) *lSup* ( $\sqsubseteq$ ) *ltl*  
 <proof>

**lemma** *mcont2mcont-ltl* [*THEN lList.mcont2mcont, simp, cont-intro*]:  
**shows** *mcont-ltl*: *mcont lSup* ( $\sqsubseteq$ ) *lSup* ( $\sqsubseteq$ ) *ltl*  
 <proof>

**lemma** *lList-case-mono* [*partial-function-mono, cont-intro*]:  
**assumes** *lnil*: *monotone orda ordb lnil*  
**and** *lcons*:  $\bigwedge x xs. \text{monotone } orda \ ordb \ (\lambda f. \text{lcons } f \ x \ xs)$   
**shows** *monotone orda ordb* ( $\lambda f. \text{case-lList } (lnil \ f) \ (\text{lcons } f) \ x$ )  
 <proof>

**lemma** *mcont-lList-case* [*cont-intro, simp*]:  
 $[\text{mcont } luba \ orda \ lubb \ ordb \ (\lambda x. \ f \ x); \bigwedge x xs. \text{mcont } luba \ orda \ lubb \ ordb \ (\lambda y. \ g \ x \ xs \ y)]$   
 $\Longrightarrow \text{mcont } luba \ orda \ lubb \ ordb \ (\lambda y. \text{case } xs \ \text{of } LNil \Rightarrow f \ y \mid LCons \ x \ xs' \Rightarrow g \ x \ xs' \ y)$   
 <proof>

**lemma** *monotone-lprefix-case* [*cont-intro*, *simp*]:  
**assumes** *mono*:  $\bigwedge x. \text{monotone } (\sqsubseteq) (\sqsubseteq) (\lambda xs. f\ x\ xs\ (LCons\ x\ xs))$   
**shows** *monotone*  $(\sqsubseteq) (\sqsubseteq) (\lambda xs. \text{case } xs \text{ of } LNil \Rightarrow LNil \mid LCons\ x\ xs' \Rightarrow f\ x\ xs'$   
*xs*)  
 $\langle \text{proof} \rangle$

**lemma** *mcont-lprefix-case-aux*:  
**fixes** *f bot*  
**defines**  $g \equiv \lambda xs. f\ (lhd\ xs)\ (ltl\ xs)\ (LCons\ (lhd\ xs)\ (ltl\ xs))$   
**assumes** *mcont*:  $\bigwedge x. \text{mcont } lSup\ (\sqsubseteq) \text{ lub } ord\ (\lambda xs. f\ x\ xs\ (LCons\ x\ xs))$   
**and** *ccpo*: *class.ccpo* *lub ord (mk-less ord)*  
**and** *bot*:  $\bigwedge x. \text{ord } bot\ x$   
**shows** *mcont*  $lSup\ (\sqsubseteq) \text{ lub } ord\ (\lambda xs. \text{case } xs \text{ of } LNil \Rightarrow bot \mid LCons\ x\ xs' \Rightarrow f\ x$   
*xs' xs*)  
 $\langle \text{proof} \rangle$

**lemma** *mcont-lprefix-case* [*cont-intro*, *simp*]:  
**assumes**  $\bigwedge x. \text{mcont } lSup\ (\sqsubseteq) lSup\ (\sqsubseteq) (\lambda xs. f\ x\ xs\ (LCons\ x\ xs))$   
**shows** *mcont*  $lSup\ (\sqsubseteq) lSup\ (\sqsubseteq) (\lambda xs. \text{case } xs \text{ of } LNil \Rightarrow LNil \mid LCons\ x\ xs' \Rightarrow f$   
*x xs' xs*)  
 $\langle \text{proof} \rangle$

**lemma** *monotone-lprefix-case-lfp* [*cont-intro*, *simp*]:  
**fixes**  $f :: - \Rightarrow - :: \text{order-bot}$   
**assumes** *mono*:  $\bigwedge x. \text{monotone } (\sqsubseteq) (\leq) (\lambda xs. f\ x\ xs\ (LCons\ x\ xs))$   
**shows** *monotone*  $(\sqsubseteq) (\leq) (\lambda xs. \text{case } xs \text{ of } LNil \Rightarrow \perp \mid LCons\ x\ xs \Rightarrow f\ x\ xs$   
 $(LCons\ x\ xs))$   
 $\langle \text{proof} \rangle$

**lemma** *mcont-lprefix-case-lfp* [*cont-intro*, *simp*]:  
**fixes**  $f :: - \Rightarrow - :: \text{complete-lattice}$   
**assumes**  $\bigwedge x. \text{mcont } lSup\ (\sqsubseteq) Sup\ (\leq) (\lambda xs. f\ x\ xs\ (LCons\ x\ xs))$   
**shows** *mcont*  $lSup\ (\sqsubseteq) Sup\ (\leq) (\lambda xs. \text{case } xs \text{ of } LNil \Rightarrow \perp \mid LCons\ x\ xs \Rightarrow f\ x$   
*xs (LCons x xs)*)  
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

## 2.8 Monotonicity and continuity of already defined functions

**lemma** *fixes f F*  
**defines**  $F \equiv \lambda \text{map } xs. \text{case } xs \text{ of } LNil \Rightarrow LNil \mid LCons\ x\ xs \Rightarrow LCons\ (f\ x)$   
 $(\text{map } xs)$   
**shows** *lmap-conv-fixp*:  $\text{lmap } f \equiv \text{ccpo.fixp } (fun\text{-lub } lSup) (fun\text{-ord } (\sqsubseteq))\ F$  (**is** *?lhs*  
 $\equiv$  *?rhs*)  
**and** *lmap-mono*:  $\bigwedge xs. \text{mono-llist } (\lambda \text{map}. F\ \text{map } xs)$  (**is** *PROP ?mono*)  
 $\langle \text{proof} \rangle$

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

**shows** *monotone-lmap*: *monotone* ( $\sqsubseteq$ ) ( $\sqsubseteq$ ) (*lmap* *f*)  
(*proof*)

**lemma** *mcont2mcont-lmap* [*THEN llist.mcont2mcont, simp, cont-intro*]:  
**shows** *mcont-lmap*: *mcont* *lSup* ( $\sqsubseteq$ ) *lSup* ( $\sqsubseteq$ ) (*lmap* *f*)  
(*proof*)

**lemma** [*partial-function-mono*]: *mono-llist* *F*  $\implies$  *mono-llist* ( $\lambda f. \textit{lmap } g (F f)$ )  
(*proof*)

**lemma** *mono-llist-lappend2* [*partial-function-mono*]:  
*mono-llist* *A*  $\implies$  *mono-llist* ( $\lambda f. \textit{lappend } xs (A f)$ )  
(*proof*)

**lemma** *mono2mono-lappend2* [*THEN llist.mono2mono, cont-intro, simp*]:  
**shows** *monotone-lappend2*: *monotone* ( $\sqsubseteq$ ) ( $\sqsubseteq$ ) (*lappend* *xs*)  
(*proof*)

**lemma** *mcont2mcont-lappend2* [*THEN llist.mcont2mcont, cont-intro, simp*]:  
**shows** *mcont-lappend2*: *mcont* *lSup* ( $\sqsubseteq$ ) *lSup* ( $\sqsubseteq$ ) (*lappend* *xs*)  
(*proof*)

**lemma** *fixes f F*  
**defines** *F*  $\equiv \lambda \textit{lset } xs. \textit{case } xs \textit{ of } LNil \Rightarrow \{\} \mid LCons \ x \ xs \Rightarrow \textit{insert } x \ (\textit{lset } xs)$   
**shows** *lset-conv-fixp*: *lset*  $\equiv \textit{ccpo.fixp } (\textit{fun-lub } Union) (\textit{fun-ord } (\sqsubseteq)) F$  (**is** -  $\equiv$  *?fixp*)  
**and** *lset-mono*:  $\bigwedge x. \textit{monotone } (\textit{fun-ord } (\sqsubseteq)) (\sqsubseteq) (\lambda f. F f x)$  (**is** *PROP ?mono*)  
(*proof*)

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

**lemma** *mcont2mcont-lset* [*THEN mcont2mcont, cont-intro, simp*]:  
**shows** *mcont-lset*: *mcont* *lSup* ( $\sqsubseteq$ ) *Union* ( $\sqsubseteq$ ) *lset*  
(*proof*)

**lemma** *lset-lSup*: *Complete-Partial-Order.chain* ( $\sqsubseteq$ ) *Y*  $\implies \textit{lset } (\textit{lSup } Y) = \bigcup (\textit{lset } ' Y)$   
(*proof*)

**lemma** *lfinite-lSupD*: *lfinite* (*lSup* *A*)  $\implies \forall xs \in A. \textit{lfinite } xs$   
(*proof*)

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

**lemma** *mcont-enat-le-lprefix-case* [*cont-intro, simp*]:  
**assumes** *mcont Sup* ( $\leq$ ) *lSup* ( $\sqsubseteq$ ) ( $\lambda x. f x (eSuc x)$ )  
**shows** *mcont Sup* ( $\leq$ ) *lSup* ( $\sqsubseteq$ ) ( $\lambda x. \text{case } x \text{ of } 0 \Rightarrow LNil \mid eSuc x' \Rightarrow f x' x$ )  
 $\langle \text{proof} \rangle$

**lemma** *compact-LConsI*:  
**assumes** *ccpo.compact lSup* ( $\sqsubseteq$ ) *xs*  
**shows** *ccpo.compact lSup* ( $\sqsubseteq$ ) (*LCons x xs*)  
 $\langle \text{proof} \rangle$

**lemma** *compact-LConsD*:  
**assumes** *ccpo.compact lSup* ( $\sqsubseteq$ ) (*LCons x xs*)  
**shows** *ccpo.compact lSup* ( $\sqsubseteq$ ) *xs*  
 $\langle \text{proof} \rangle$

**lemma** *compact-LCons-iff* [*simp*]:  
*ccpo.compact lSup* ( $\sqsubseteq$ ) (*LCons x xs*)  $\longleftrightarrow$  *ccpo.compact lSup* ( $\sqsubseteq$ ) *xs*  
 $\langle \text{proof} \rangle$

**lemma** *compact-lfiniteI*:  
*lfinite xs*  $\implies$  *ccpo.compact lSup* ( $\sqsubseteq$ ) *xs*  
 $\langle \text{proof} \rangle$

**lemma** *compact-lfiniteD*:  
**assumes** *ccpo.compact lSup* ( $\sqsubseteq$ ) *xs*  
**shows** *lfinite xs*  
 $\langle \text{proof} \rangle$

**lemma** *compact-eq-lfinite* [*simp*]: *ccpo.compact lSup* ( $\sqsubseteq$ ) = *lfinite*  
 $\langle \text{proof} \rangle$

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

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

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

**primcorec** *ltake* :: *enat*  $\Rightarrow$  *'a llist*  $\Rightarrow$  *'a llist*

**where**

$n = 0 \vee \text{lnull } xs \implies \text{lnull } (\text{ltake } n \text{ } xs)$   
|  $\text{lhs } (\text{ltake } n \text{ } xs) = \text{lhs } xs$   
|  $\text{ltl } (\text{ltake } n \text{ } xs) = \text{ltake } (\text{epred } n) (\text{ltl } xs)$

**definition** *ldropn* :: *nat*  $\Rightarrow$  *'a llist*  $\Rightarrow$  *'a llist*

**where** *ldropn* *n* *xs* = (*ltl*  $\overset{\sim}{\sim}$  *n*) *xs*

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

**begin**

**partial-function** (*llist*) *ldrop* :: *enat*  $\Rightarrow$  *'a llist*  $\Rightarrow$  *'a llist*

**where**

$\text{ldrop } n \text{ } xs = (\text{case } n \text{ of } 0 \Rightarrow xs \mid \text{eSuc } n' \Rightarrow \text{case } xs \text{ of } \text{LNil} \Rightarrow \text{LNil} \mid \text{LCons } x \text{ } xs' \Rightarrow \text{ldrop } n' \text{ } xs')$

**end**

**primcorec** *ltakeWhile* :: (*'a*  $\Rightarrow$  *bool*)  $\Rightarrow$  *'a llist*  $\Rightarrow$  *'a llist*

**where**

$\text{lnull } xs \vee \neg P (\text{lhs } xs) \implies \text{lnull } (\text{ltakeWhile } P \text{ } xs)$   
|  $\text{lhs } (\text{ltakeWhile } P \text{ } xs) = \text{lhs } xs$   
|  $\text{ltl } (\text{ltakeWhile } P \text{ } xs) = \text{ltakeWhile } P (\text{ltl } xs)$

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

**notes** *[[function-internals]]*

**begin**

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

**where** *ldropWhile* *xs* = (*case* *xs* *of* *LNil*  $\Rightarrow$  *LNil* | *LCons* *x* *xs'*  $\Rightarrow$  *if* *P* *x* *then* *ldropWhile* *xs'* *else* *xs*)

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

**where** *lfilter* *xs* = (*case* *xs* *of* *LNil*  $\Rightarrow$  *LNil* | *LCons* *x* *xs'*  $\Rightarrow$  *if* *P* *x* *then* *LCons* *x* (*lfilter* *xs'*) *else* *lfilter* *xs'*)

**end**

**primrec** *lnth* :: *'a llist*  $\Rightarrow$  *nat*  $\Rightarrow$  *'a*

**where**

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

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

**primcorec** *lzip* :: *'a llist*  $\Rightarrow$  *'b llist*  $\Rightarrow$  (*'a*  $\times$  *'b*) *llist*

**where**

$lnull\ xs \vee lnull\ ys \implies lnull\ (lzip\ xs\ ys)$   
 $| lhd\ (lzip\ xs\ ys) = (lhd\ xs,\ lhd\ ys)$   
 $| ltl\ (lzip\ xs\ ys) = lzip\ (ltl\ xs)\ (ltl\ ys)$

**definition**  $llast :: 'a\ llist \Rightarrow 'a$

**where**  $[nitpick-simp]$ :

$llast\ xs = (case\ llength\ xs\ of\ enat\ n \Rightarrow (case\ n\ of\ 0 \Rightarrow undefined\ | Suc\ n' \Rightarrow lnth\ xs\ n'))\ | \infty \Rightarrow undefined)$

**coinductive**  $ldistinct :: 'a\ llist \Rightarrow bool$

**where**

$LNil\ [simp]:\ ldistinct\ LNil$

$| LCons: \llbracket x \notin lset\ xs;\ ldistinct\ xs \rrbracket \implies ldistinct\ (LCons\ x\ xs)$

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

**definition**  $inf-llist :: (nat \Rightarrow 'a) \Rightarrow 'a\ llist$

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

**abbreviation**  $repeat :: 'a \Rightarrow 'a\ llist$

**where**  $repeat \equiv iterates\ (\lambda x.\ x)$

**definition**  $lstrict-prefix :: 'a\ llist \Rightarrow 'a\ llist \Rightarrow bool$

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

longest common prefix

**definition**  $llcp :: 'a\ llist \Rightarrow 'a\ llist \Rightarrow enat$

**where**  $[code\ del]:$

$llcp\ xs\ ys =$

$enat-unfold\ (\lambda(xs,\ ys).\ lnull\ xs \vee lnull\ ys \vee lhd\ xs \neq lhd\ ys)\ (map-prod\ ltl\ ltl)\ (xs,\ ys)$

**coinductive**  $llexord :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ llist \Rightarrow 'a\ llist \Rightarrow bool$

**for**  $r :: 'a \Rightarrow 'a \Rightarrow bool$

**where**

$llexord-LCons-eq: llexord\ r\ xs\ ys \implies llexord\ r\ (LCons\ x\ xs)\ (LCons\ x\ ys)$

$| llexord-LCons-less: r\ x\ y \implies llexord\ r\ (LCons\ x\ xs)\ (LCons\ y\ ys)$

$| llexord-LNil\ [simp,\ intro!]:\ llexord\ r\ LNil\ ys$

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

**begin**

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

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

**end**

**definition**  $lhd' :: 'a\ llist \Rightarrow 'a\ option$  **where**



$lhd' xs = (if\ lnull\ xs\ then\ None\ else\ Some\ (lhd\ xs))$

**lemma**  $lhd'$ -simps[simp]:

$lhd'\ LNil = None$

$lhd'\ (LCons\ x\ xs) = Some\ x$

$\langle proof \rangle$

**definition**  $ltl' :: 'a\ llist \Rightarrow 'a\ llist\ option$  **where**

$ltl'\ xs = (if\ lnull\ xs\ then\ None\ else\ Some\ (lhd\ xs))$

**lemma**  $ltl'$ -simps[simp]:

$ltl'\ LNil = None$

$ltl'\ (LCons\ x\ xs) = Some\ xs$

$\langle proof \rangle$

**definition**  $lnths :: 'a\ llist \Rightarrow nat\ set \Rightarrow 'a\ llist$

**where**  $lnths\ xs\ A = lmap\ fst\ (lfilter\ (\lambda(x, y). y \in A)\ (lzip\ xs\ (iterates\ Suc\ 0)))$

**definition** (in *monoid-add*)  $lsum-list :: 'a\ llist \Rightarrow 'a$

**where**  $lsum-list\ xs = (if\ lfinite\ xs\ then\ sum-list\ (list-of\ xs)\ else\ 0)$

## 2.10 Converting ordinary lists to lazy lists: *l*list-of

**lemma**  $lhd$ -*l*list-of [simp]:  $lhd\ (llist-of\ xs) = hd\ xs$

$\langle proof \rangle$

**lemma**  $ltl$ -*l*list-of [simp]:  $ltl\ (llist-of\ xs) = llist-of\ (tl\ xs)$

$\langle proof \rangle$

**lemma**  $lfinite$ -*l*list-of [simp]:  $lfinite\ (llist-of\ xs)$

$\langle proof \rangle$

**lemma**  $lfinite$ -eq-range-*l*list-of:  $lfinite\ xs \longleftrightarrow xs \in range\ llist-of$

$\langle proof \rangle$

**lemma**  $lnull$ -*l*list-of [simp]:  $lnull\ (llist-of\ xs) \longleftrightarrow xs = []$

$\langle proof \rangle$

**lemma**  $l$ list-of-eq-LNil-conv:

$llist-of\ xs = LNil \longleftrightarrow xs = []$

$\langle proof \rangle$

**lemma**  $l$ list-of-eq-LCons-conv:

$llist-of\ xs = LCons\ y\ ys \longleftrightarrow (\exists\ xs'. xs = y \# xs' \wedge ys = llist-of\ xs')$

$\langle proof \rangle$

**lemma**  $lappend$ -*l*list-of-*l*list-of:

$lappend\ (llist-of\ xs)\ (llist-of\ ys) = llist-of\ (xs\ @\ ys)$

$\langle proof \rangle$

**lemma** *lfinite-rev-induct* [*consumes 1, case-names Nil snoc*]:  
**assumes** *fin*: *lfinite xs*  
**and** *Nil*:  $P \text{ LNil}$   
**and** *snoc*:  $\bigwedge x xs. \llbracket \text{lfinite } xs; P \text{ xs} \rrbracket \implies P (\text{lappend } xs (\text{LCons } x \text{ LNil}))$   
**shows**  $P \text{ xs}$   
*<proof>*

**lemma** *lappend-llist-of-LCons*:  
 $\text{lappend } (\text{llist-of } xs) (\text{LCons } y \text{ ys}) = \text{lappend } (\text{llist-of } (xs @ [y])) \text{ ys}$   
*<proof>*

**lemma** *lmap-llist-of* [*simp*]:  
 $\text{lmap } f (\text{llist-of } xs) = \text{llist-of } (\text{map } f \text{ xs})$   
*<proof>*

**lemma** *lset-llist-of* [*simp*]:  $\text{lset } (\text{llist-of } xs) = \text{set } xs$   
*<proof>*

**lemma** *llist-of-inject* [*simp*]:  $\text{llist-of } xs = \text{llist-of } ys \longleftrightarrow xs = ys$   
*<proof>*

**lemma** *inj-llist-of* [*simp*]:  $\text{inj } \text{llist-of}$   
*<proof>*

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

**lemma** *list-of-llist-of* [*simp*]:  $\text{list-of } (\text{llist-of } xs) = xs$   
*<proof>*

**lemma** *llist-of-list-of* [*simp*]:  $\text{lfinite } xs \implies \text{llist-of } (\text{list-of } xs) = xs$   
*<proof>*

**lemma** *list-of-LNil* [*simp, nitpick-simp*]:  $\text{list-of } \text{LNil} = []$   
*<proof>*

**lemma** *list-of-LCons* [*simp*]:  $\text{lfinite } xs \implies \text{list-of } (\text{LCons } x \text{ xs}) = x \# \text{list-of } xs$   
*<proof>*

**lemma** *list-of-LCons-conv* [*nitpick-simp*]:  
 $\text{list-of } (\text{LCons } x \text{ xs}) = (\text{if } \text{lfinite } xs \text{ then } x \# \text{list-of } xs \text{ else undefined})$   
*<proof>*

**lemma** *list-of-lappend*:  
**assumes** *lfinite xs lfinite ys*  
**shows**  $\text{list-of } (\text{lappend } xs \text{ ys}) = \text{list-of } xs @ \text{list-of } ys$   
*<proof>*

**lemma** *list-of-lmap* [*simp*]:

**assumes** *lfinite xs*  
**shows** *list-of (lmap f xs) = map f (list-of xs)*  
 ⟨*proof*⟩

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

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

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

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 []*  
 ⟨*proof*⟩

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

## 2.12 The length of a lazy list: *llength*

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

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

**lemma** *llength-lnull [simp]*: *lnull xs  $\implies$  llength xs = 0*  
 ⟨*proof*⟩

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

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

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

**lemma** *llength-lappend* [simp]:  $llength (lappend\ xs\ ys) = llength\ xs + llength\ ys$   
<proof>

**lemma** *llength-llist-of* [simp]:  
 $llength (l\text{list-of}\ xs) = enat (length\ xs)$   
<proof>

**lemma** *length-list-of*:  
 $lfinite\ xs \implies enat (length (list-of\ xs)) = llength\ xs$   
<proof>

**lemma** *length-list-of-conv-the-enat*:  
 $lfinite\ xs \implies length (list-of\ xs) = the-enat (llength\ xs)$   
<proof>

**lemma** *llength-eq-enat-lfiniteD*:  $llength\ xs = enat\ n \implies lfinite\ xs$   
<proof>

**lemma** *lfinite-llength-enat*:  
assumes  $lfinite\ xs$   
shows  $\exists n. llength\ xs = enat\ n$   
<proof>

**lemma** *lfinite-conv-llength-enat*:  
 $lfinite\ xs \longleftrightarrow (\exists n. llength\ xs = enat\ n)$   
<proof>

**lemma** *not-lfinite-llength*:  
 $\neg lfinite\ xs \implies llength\ xs = \infty$   
<proof>

**lemma** *llength-eq-infty-conv-lfinite*:  
 $llength\ xs = \infty \longleftrightarrow \neg lfinite\ xs$   
<proof>

**lemma** *lfinite-finite-index*:  $lfinite\ xs \implies finite\ \{n. enat\ n < llength\ xs\}$   
<proof>

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]:  
 $gen-llength\ n\ LNil = enat\ n$   
 $gen-llength\ n\ (LCons\ x\ xs) = gen-llength\ (n + 1)\ xs$   
<proof>

**lemma** *llength-code* [code]:  $llength = gen-llength\ 0$   
<proof>

**lemma** fixes  $F$

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

**lemma**  $\text{mono2mono-llength}$ [*THEN lfp.mono2mono, simp, cont-intro*]:

**shows**  $\text{monotone-llength}$ :  $\text{monotone} \ (\sqsubseteq) \ (\leq) \ \text{llength}$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{mcont2mcont-llength}$ [*THEN lfp.mcont2mcont, simp, cont-intro*]:

**shows**  $\text{mcont-llength}$ :  $\text{mcont } lSup \ (\sqsubseteq) \ Sup \ (\leq) \ \text{llength}$   
 $\langle \text{proof} \rangle$

## 2.13 Taking and dropping from lazy lists: $l\text{take}$ , $l\text{dropn}$ , and $l\text{drop}$

**lemma**  $l\text{take-LNil}$  [*simp, code, nitpick-simp*]:  $l\text{take } n \ LNil = LNil$   
 $\langle \text{proof} \rangle$

**lemma**  $l\text{take-0}$  [*simp*]:  $l\text{take } 0 \ xs = LNil$   
 $\langle \text{proof} \rangle$

**lemma**  $l\text{take-eSuc-LCons}$  [*simp*]:  
 $l\text{take} \ (eSuc \ n) \ (LCons \ x \ xs) = LCons \ x \ (l\text{take} \ n \ xs)$   
 $\langle \text{proof} \rangle$

**lemma**  $l\text{take-eSuc}$ :  
 $l\text{take} \ (eSuc \ n) \ xs =$   
 $(\text{case } xs \ \text{of } LNil \Rightarrow LNil \mid LCons \ x \ xs' \Rightarrow LCons \ x \ (l\text{take} \ n \ xs'))$   
 $\langle \text{proof} \rangle$

**lemma**  $l\text{null-ltake}$  [*simp*]:  $l\text{null} \ (l\text{take} \ n \ xs) \longleftrightarrow l\text{null} \ xs \vee n = 0$   
 $\langle \text{proof} \rangle$

**lemma**  $l\text{take-eq-LNil-iff}$ :  $l\text{take} \ n \ xs = LNil \longleftrightarrow xs = LNil \vee n = 0$   
 $\langle \text{proof} \rangle$

**lemma**  $LNil-eq-ltake-iff$  [*simp*]:  $LNil = l\text{take} \ n \ xs \longleftrightarrow xs = LNil \vee n = 0$   
 $\langle \text{proof} \rangle$

**lemma**  $l\text{take-LCons}$  [*code, nitpick-simp*]:  
 $l\text{take} \ n \ (LCons \ x \ xs) =$   
 $(\text{case } n \ \text{of } 0 \Rightarrow LNil \mid eSuc \ n' \Rightarrow LCons \ x \ (l\text{take} \ n' \ xs))$   
 $\langle \text{proof} \rangle$

**lemma** *lhd-ltake* [simp]:  $n \neq 0 \implies \text{lhd} (\text{ltake } n \text{ } xs) = \text{lhd } xs$   
<proof>

**lemma** *ltl-ltake*:  $\text{ltl} (\text{ltake } n \text{ } xs) = \text{ltake} (\text{epred } n) (\text{ltl } xs)$   
<proof>

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

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

**lemma** *ltake-ltl*:  $\text{ltake } n (\text{ltl } xs) = \text{ltl} (\text{ltake} (\text{eSuc } n) \text{ } xs)$   
<proof>

**lemma** *llength-ltake* [simp]:  $\text{llength} (\text{ltake } n \text{ } xs) = \min n (\text{llength } xs)$   
<proof>

**lemma** *ltake-lmap* [simp]:  $\text{ltake } n (\text{lmap } f \text{ } xs) = \text{lmap } f (\text{ltake } n \text{ } xs)$   
<proof>

**lemma** *ltake-ltake* [simp]:  $\text{ltake } n (\text{ltake } m \text{ } xs) = \text{ltake} (\min n m) \text{ } xs$   
<proof>

**lemma** *lset-ltake*:  $\text{lset} (\text{ltake } n \text{ } xs) \subseteq \text{lset } xs$   
<proof>

**lemma** *ltake-all*:  $\text{llength } xs \leq m \implies \text{ltake } m \text{ } xs = xs$   
<proof>

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

**lemma** *lfinite-ltake* [simp]:  
 $\text{lfinite} (\text{ltake } n \text{ } xs) \iff \text{lfinite } xs \vee n < \infty$   
(is ?lhs  $\iff$  ?rhs)  
<proof>

**lemma** *ltake-lappend1*:  $n \leq \text{llength } xs \implies \text{ltake } n (\text{lappend } xs \text{ } ys) = \text{ltake } n \text{ } xs$   
<proof>

**lemma** *ltake-lappend2*:  
 $\text{llength } xs \leq n \implies \text{ltake } n (\text{lappend } xs \text{ } ys) = \text{lappend } xs (\text{ltake} (n - \text{llength } xs) \text{ } ys)$   
<proof>

**lemma** *ltake-lappend*:  
 $\text{ltake } n (\text{lappend } xs \text{ } ys) = \text{lappend} (\text{ltake } n \text{ } xs) (\text{ltake} (n - \text{llength } xs) \text{ } ys)$   
<proof>

**lemma** *take-list-of*:

**assumes** *lfinite xs*

**shows**  $\text{take } n \text{ (list-of } xs) = \text{list-of (ltake (enat } n) xs)$

*<proof>*

**lemma** *ltake-eq-ltake-antimono*:

$\llbracket \text{ltake } n \text{ } xs = \text{ltake } n \text{ } ys; m \leq n \rrbracket \implies \text{ltake } m \text{ } xs = \text{ltake } m \text{ } ys$

*<proof>*

**lemma** *ltake-is-lprefix* [*simp, intro*]:  $\text{ltake } n \text{ } xs \sqsubseteq xs$

*<proof>*

**lemma** *lprefix-ltake-same* [*simp*]:

$\text{ltake } n \text{ } xs \sqsubseteq \text{ltake } m \text{ } xs \iff n \leq m \vee \text{llength } xs \leq m$

(**is** *?lhs*  $\iff$  *?rhs*)

*<proof>*

**lemma** *fixes f F*

**defines**  $F \equiv \lambda \text{ltake } n \text{ } xs. \text{case } xs \text{ of } LNil \Rightarrow LNil \mid LCons \ x \ xs \Rightarrow \text{case } n \text{ of } 0 \Rightarrow LNil \mid eSuc \ n \Rightarrow LCons \ x \ (\text{ltake } n \text{ } xs)$

**shows** *ltake-conv-fixp*:  $\text{ltake} \equiv \text{curry } (\text{ccpo.fixp } (\text{fun-lub } lSup) (\text{fun-ord } (\sqsubseteq))) (\lambda \text{ltake}. \text{case-prod } (F (\text{curry } \text{ltake})))$  (**is** *?lhs*  $\equiv$  *?rhs*)

**and** *ltake-mono*:  $\bigwedge nxs. \text{mono-llist } (\lambda \text{ltake}. \text{case } nxs \text{ of } (n, xs) \Rightarrow F (\text{curry } \text{ltake}) \ n \ xs)$  (**is** *PROP ?mono*)

*<proof>*

**lemma** *monotone-ltake*:  $\text{monotone } (\text{rel-prod } (\leq) (\sqsubseteq)) (\sqsubseteq) (\text{case-prod } \text{ltake})$

*<proof>*

**lemma** *mono2mono-ltake1* [*THEN llist.mono2mono, cont-intro, simp*]:

**shows** *monotone-ltake1*:  $\text{monotone } (\leq) (\sqsubseteq) (\lambda n. \text{ltake } n \text{ } xs)$

*<proof>*

**lemma** *mono2mono-ltake2* [*THEN llist.mono2mono, cont-intro, simp*]:

**shows** *monotone-ltake2*:  $\text{monotone } (\sqsubseteq) (\sqsubseteq) (\text{ltake } n)$

*<proof>*

**lemma** *mcont-ltake*:  $\text{mcont } (\text{prod-lub } Sup \ lSup) (\text{rel-prod } (\leq) (\sqsubseteq)) \ lSup (\sqsubseteq) (\text{case-prod } \text{ltake})$

*<proof>*

**lemma** *mcont2mcont-ltake1* [*THEN llist.mcont2mcont, cont-intro, simp*]:

**shows** *mcont-ltake1*:  $\text{mcont } Sup (\leq) \ lSup (\sqsubseteq) (\lambda n. \text{ltake } n \text{ } xs)$

*<proof>*

**lemma** *mcont2mcont-ltake2* [*THEN llist.mcont2mcont, cont-intro, simp*]:

**shows** *mcont-ltake2*:  $\text{mcont } lSup (\sqsubseteq) \ lSup (\sqsubseteq) (\text{ltake } n)$

*<proof>*

**lemma** [partial-function-mono]:  $\text{mono-list } F \implies \text{mono-list } (\lambda f. \text{ltake } n (F f))$   
 ⟨proof⟩

**lemma** *list-induct2*:

**assumes** *adm*:  $\text{ccpo.admissible } (\text{prod-lub } \text{lSup } \text{lSup}) (\text{rel-prod } (\sqsubseteq) (\sqsubseteq)) (\lambda x. P (\text{fst } x) (\text{snd } x))$

**and** *LNil*:  $P \text{ LNil LNil}$

**and** *LCons1*:  $\bigwedge x \text{ xs}. \llbracket \text{lfinite } x \text{ xs}; P \text{ xs LNil} \rrbracket \implies P (\text{LCons } x \text{ xs}) \text{ LNil}$

**and** *LCons2*:  $\bigwedge y \text{ ys}. \llbracket \text{lfinite } y \text{ ys}; P \text{ LNil } y \text{ ys} \rrbracket \implies P \text{ LNil } (\text{LCons } y \text{ ys})$

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

**shows**  $P \text{ xs } y \text{ ys}$

⟨proof⟩

**lemma** *ldropn-0* [simp]:  $\text{ldropn } 0 \text{ xs} = \text{xs}$

⟨proof⟩

**lemma** *ldropn-LNil* [code, simp]:  $\text{ldropn } n \text{ LNil} = \text{LNil}$

⟨proof⟩

**lemma** *ldropn-lnull*:  $\text{lnull } x \text{ s} \implies \text{ldropn } n \text{ xs} = \text{LNil}$

⟨proof⟩

**lemma** *ldropn-LCons* [code]:

$\text{ldropn } n (\text{LCons } x \text{ xs}) = (\text{case } n \text{ of } 0 \Rightarrow \text{LCons } x \text{ xs} \mid \text{Suc } n' \Rightarrow \text{ldropn } n' \text{ xs})$

⟨proof⟩

**lemma** *ldropn-Suc*:  $\text{ldropn } (\text{Suc } n) \text{ xs} = (\text{case } x \text{ s of } \text{LNil} \Rightarrow \text{LNil} \mid \text{LCons } x \text{ xs}' \Rightarrow \text{ldropn } n \text{ xs}')$

⟨proof⟩

**lemma** *ldropn-Suc-LCons* [simp]:  $\text{ldropn } (\text{Suc } n) (\text{LCons } x \text{ xs}) = \text{ldropn } n \text{ xs}$

⟨proof⟩

**lemma** *ltl-ldropn*:  $\text{ltl } (\text{ldropn } n \text{ xs}) = \text{ldropn } n (\text{ltl } x \text{ s})$

⟨proof⟩

**lemma** *ldrop-simps* [simp]:

**shows** *ldrop-LNil*:  $\text{ldrop } n \text{ LNil} = \text{LNil}$

**and** *ldrop-0*:  $\text{ldrop } 0 \text{ xs} = \text{xs}$

**and** *ldrop-eSuc-LCons*:  $\text{ldrop } (\text{eSuc } n) (\text{LCons } x \text{ xs}) = \text{ldrop } n \text{ xs}$

⟨proof⟩

**lemma** *ldrop-lnull*:  $\text{lnull } x \text{ s} \implies \text{ldrop } n \text{ xs} = \text{LNil}$

⟨proof⟩

**lemma** *fixes f F*

**defines**  $F \equiv \lambda \text{ldropn } x \text{ s}. \text{case } x \text{ s of } \text{LNil} \Rightarrow \lambda \cdot. \text{LNil} \mid \text{LCons } x \text{ xs} \Rightarrow \lambda n. \text{if } n = 0 \text{ then } \text{LCons } x \text{ xs} \text{ else } \text{ldropn } x \text{ s } (n - 1)$



**shows** *ldrop-conv-fixp*:  $(\lambda xs\ n.\ \text{ldropn}\ n\ xs) \equiv \text{ccpo.fixp}\ (\text{fun-lub}\ (\text{fun-lub}\ \text{lSup}))$   
 $(\text{fun-ord}\ (\text{fun-ord}\ \text{lprefix}))\ (\lambda \text{ldrop}.\ F\ \text{ldrop})$  (**is** *?lhs*  $\equiv$  *?rhs*)  
**and** *ldrop-mono*:  $\bigwedge xs.\ \text{mono-llist-lift}\ (\lambda \text{ldrop}.\ F\ \text{ldrop}\ xs)$  (**is** *PROP* *?mono*)  
 $\langle \text{proof} \rangle$

**lemma** *ldropn-fixp-case-conv*:

$(\lambda xs.\ \text{case}\ xs\ \text{of}\ \text{LNil} \Rightarrow \lambda-. \text{LNil} \mid \text{LCons}\ x\ xs \Rightarrow \lambda n.\ \text{if}\ n = 0\ \text{then}\ \text{LCons}\ x\ xs$   
 $\text{else}\ f\ xs\ (n - 1)) =$   
 $(\lambda xs\ n.\ \text{case}\ xs\ \text{of}\ \text{LNil} \Rightarrow \text{LNil} \mid \text{LCons}\ x\ xs \Rightarrow \text{if}\ n = 0\ \text{then}\ \text{LCons}\ x\ xs\ \text{else}\ f$   
 $xs\ (n - 1))$   
 $\langle \text{proof} \rangle$

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

**lemma** *mono2mono-ldropn* [*THEN* *llist.mono2mono*, *cont-intro*, *simp*]:  
**shows** *monotone-ldropn'*:  $\text{monotone}\ \text{lprefix}\ \text{lprefix}\ (\lambda xs.\ \text{ldropn}\ n\ xs)$   
 $\langle \text{proof} \rangle$

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

**lemma** *mcont2mcont-ldropn* [*THEN* *llist.mcont2mcont*, *cont-intro*, *simp*]:  
**shows** *mcont-ldropn*:  $\text{mcont}\ \text{lSup}\ \text{lprefix}\ \text{lSup}\ \text{lprefix}\ (\text{ldropn}\ n)$   
 $\langle \text{proof} \rangle$

**lemma** *monotone-enat-cocase* [*cont-intro*, *simp*]:  
 $\llbracket \bigwedge n.\ \text{monotone}\ (\leq)\ \text{ord}\ (\lambda n.\ f\ n\ (\text{eSuc}\ n));$   
 $\bigwedge n.\ \text{ord}\ a\ (f\ n\ (\text{eSuc}\ n));\ \text{ord}\ a\ a \rrbracket$   
 $\implies \text{monotone}\ (\leq)\ \text{ord}\ (\lambda n.\ \text{case}\ n\ \text{of}\ 0 \Rightarrow a \mid \text{eSuc}\ n' \Rightarrow f\ n'\ n)$   
 $\langle \text{proof} \rangle$

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

**lemma** *mono2mono-ldrop2* [*THEN* *llist.mono2mono*, *cont-intro*, *simp*]:  
**shows** *monotone-ldrop2*:  $\text{monotone}\ (\sqsubseteq)\ (\sqsubseteq)\ (\text{ldrop}\ n)$   
 $\langle \text{proof} \rangle$

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

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

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

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

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

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

**lemma** *lnull-ldrop [simp]*:  $lnull (ldrop n xs) \longleftrightarrow llength xs \leq n$   
(proof)

**lemma** *ldropn-eq-LNil*:  $(ldropn n xs = LNil) = (llength xs \leq enat n)$   
(proof)

**lemma** *ldropn-all*:  $llength xs \leq enat m \implies ldropn m xs = LNil$   
(proof)

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

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

**lemma** *ldrop-eSuc*:  
 $ldrop (eSuc n) xs = (case xs of LNil \Rightarrow LNil \mid LCons x xs' \Rightarrow ldrop n xs')$   
(proof)

**lemma** *ldrop-LCons*:  
 $ldrop n (LCons x xs) = (case n of 0 \Rightarrow LCons x xs \mid eSuc n' \Rightarrow ldrop n' xs)$   
(proof)

**lemma** *ldrop-inf [code, simp]*:  $ldrop \infty xs = LNil$   
(proof)

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

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

**lemma** *lfinite-ldrop [simp]*:  
 $lfinite (ldrop n xs) \longleftrightarrow lfinite xs \vee n = \infty$   
(proof)

**lemma** *ldropn-ltl*:  $ldropn n (ltl xs) = ldropn (Suc n) xs$

*<proof>*

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

**lemma** *lset-ldroprn-subset: lset (ldroprn n xs) ⊆ lset xs*  
*<proof>*

**lemma** *in-lset-ldroprnD: x ∈ lset (ldroprn n xs) ⇒ x ∈ lset xs*  
*<proof>*

**lemma** *lset-ldrop-subset: lset (ldrop n xs) ⊆ lset xs*  
*<proof>*

**lemma** *in-lset-ldropD: x ∈ lset (ldrop n xs) ⇒ x ∈ lset xs*  
*<proof>*

**lemma** *lappend-ltake-ldrop: lappend (ltake n xs) (ldrop n xs) = xs*  
*<proof>*

**lemma** *ldroprn-lappend:*  
*ldroprn n (lappend xs ys) =*  
*(if enat n < llength xs then lappend (ldroprn n xs) ys*  
*else ldroprn (n - the-enat (llength xs)) ys)*  
*<proof>*

**lemma** *ldroprn-lappend2:*  
*llength xs ≤ enat n ⇒ ldroprn n (lappend xs ys) = ldroprn (n - the-enat (llength*  
*xs)) ys*  
*<proof>*

**lemma** *lappend-ltake-enat-ldroprn [simp]: lappend (ltake (enat n) xs) (ldroprn n xs)*  
*= xs*  
*<proof>*

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

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

**lemma** *ldroprn-eq-LConsD:*  
*ldroprn n xs = LCons y ys ⇒ enat n < llength xs*  
*<proof>*

**lemma** *ldrop-eq-LConsD*:

$ldrop\ n\ xs = LCons\ y\ ys \implies n < llength\ xs$   
*<proof>*

**lemma** *ldropn-lmap [simp]*:  $ldropn\ n\ (lmap\ f\ xs) = lmap\ f\ (ldropn\ n\ xs)$   
*<proof>*

**lemma** *ldrop-lmap [simp]*:  $ldrop\ n\ (lmap\ f\ xs) = lmap\ f\ (ldrop\ n\ xs)$   
*<proof>*

**lemma** *ldropn-ldropn [simp]*:  
 $ldropn\ n\ (ldropn\ m\ xs) = ldropn\ (n + m)\ xs$   
*<proof>*

**lemma** *ldrop-ldrop [simp]*:  
 $ldrop\ n\ (ldrop\ m\ xs) = ldrop\ (n + m)\ xs$   
*<proof>*

**lemma** *llength-ldropn [simp]*:  $llength\ (ldropn\ n\ xs) = llength\ xs - enat\ n$   
*<proof>*

**lemma** *enat-llength-ldropn*:  
 $enat\ n \leq llength\ xs \implies enat\ (n - m) \leq llength\ (ldropn\ m\ xs)$   
*<proof>*

**lemma** *ldropn-llist-of [simp]*:  $ldropn\ n\ (llist-of\ xs) = llist-of\ (drop\ n\ xs)$   
*<proof>*

**lemma** *ldrop-llist-of*:  $ldrop\ (enat\ n)\ (llist-of\ xs) = llist-of\ (drop\ n\ xs)$   
*<proof>*

**lemma** *drop-list-of*:  
 $lfinite\ xs \implies drop\ n\ (list-of\ xs) = list-of\ (ldropn\ n\ xs)$   
*<proof>*

**lemma** *llength-ldrop*:  $llength\ (ldrop\ n\ xs) = (if\ n = \infty\ then\ 0\ else\ llength\ xs - n)$   
*<proof>*

**lemma** *ltake-ldropn*:  $ltake\ n\ (ldropn\ m\ xs) = ldropn\ m\ (ltake\ (n + enat\ m)\ xs)$   
*<proof>*

**lemma** *ldropn-ltake*:  $ldropn\ n\ (ltake\ m\ xs) = ltake\ (m - enat\ n)\ (ldropn\ n\ xs)$   
*<proof>*

**lemma** *ltake-ldrop*:  $ltake\ n\ (ldrop\ m\ xs) = ldrop\ m\ (ltake\ (n + m)\ xs)$   
*<proof>*

**lemma** *ldrop-ltake*:  $ldrop\ n\ (ltake\ m\ xs) = ltake\ (m - n)\ (ldrop\ n\ xs)$

*<proof>*

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

**lemma** *lnth-LNil*:

$lnth\ LNil\ n = undefined\ n$

*<proof>*

**lemma** *lnth-0 [simp]*:

$lnth\ (LCons\ x\ xs)\ 0 = x$

*<proof>*

**lemma** *lnth-Suc-LCons [simp]*:

$lnth\ (LCons\ x\ xs)\ (Suc\ n) = lnth\ xs\ n$

*<proof>*

**lemma** *lnth-LCons*:

$lnth\ (LCons\ x\ xs)\ n = (case\ n\ of\ 0 \Rightarrow x \mid Suc\ n' \Rightarrow lnth\ xs\ n')$

*<proof>*

**lemma** *lnth-LCons'*:  $lnth\ (LCons\ x\ xs)\ n = (if\ n = 0\ then\ x\ else\ lnth\ xs\ (n - 1))$

*<proof>*

**lemma** *lhd-conv-lnth*:

$\neg\ lnull\ xs \Longrightarrow lhd\ xs = lnth\ xs\ 0$

*<proof>*

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

**lemma** *lnth-ltl*:  $\neg\ lnull\ xs \Longrightarrow lnth\ (ltl\ xs)\ n = lnth\ xs\ (Suc\ n)$

*<proof>*

**lemma** *lhd-ldropr*:

$enat\ n < llength\ xs \Longrightarrow lhd\ (ldropr\ n\ xs) = lnth\ xs\ n$

*<proof>*

**lemma** *lhd-ldrop*:

**assumes**  $n < llength\ xs$

**shows**  $lhd\ (ldrop\ n\ xs) = lnth\ xs\ (the-enat\ n)$

*<proof>*

**lemma** *lnth-beyond*:

$llength\ xs \leq enat\ n \Longrightarrow lnth\ xs\ n = undefined\ (n - (case\ llength\ xs\ of\ enat\ m \Rightarrow m))$

*<proof>*

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

$enat\ n < llength\ xs \Longrightarrow lnth\ (lmap\ f\ xs)\ n = f\ (lnth\ xs\ n)$

*<proof>*

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

$enat (n + m) < llength\ xs \implies lnth\ (ldropn\ n\ xs)\ m = lnth\ xs\ (m + n)$   
<proof>

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

$n + enat\ m < llength\ xs \implies lnth\ (ldrop\ n\ xs)\ m = lnth\ xs\ (m + the-enat\ n)$   
<proof>

**lemma** *in-lset-conv-lnth*:

$x \in lset\ xs \iff (\exists n. enat\ n < llength\ xs \wedge lnth\ xs\ n = x)$   
(is ?lhs  $\iff$  ?rhs)  
<proof>

**lemma** *lset-conv-lnth*:  $lset\ xs = \{lnth\ xs\ n \mid n. enat\ n < llength\ xs\}$

<proof>

**lemma** *lnth-llist-of* [simp]:  $lnth\ (llist-of\ xs) = nth\ xs$

<proof>

**lemma** *nth-list-of* [simp]:

**assumes** *lfinite xs*

**shows**  $nth\ (list-of\ xs) = lnth\ xs$

<proof>

**lemma** *lnth-lappend1*:

$enat\ n < llength\ xs \implies lnth\ (lappend\ xs\ ys)\ n = lnth\ xs\ n$   
<proof>

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

$lnth\ (lappend\ (llist-of\ xs)\ ys)\ n =$

(if  $n < length\ xs$  then  $xs\ !\ n$  else  $lnth\ ys\ (n - length\ xs)$ )

<proof>

**lemma** *lnth-lappend2*:

$\llbracket llength\ xs = enat\ k; k \leq n \rrbracket \implies lnth\ (lappend\ xs\ ys)\ n = lnth\ ys\ (n - k)$   
<proof>

**lemma** *lnth-lappend*:

$lnth\ (lappend\ xs\ ys)\ n = (if\ enat\ n < llength\ xs\ then\ lnth\ xs\ n\ else\ lnth\ ys\ (n - the-enat\ (llength\ xs)))$

<proof>

**lemma** *lnth-ltake*:

$enat\ m < n \implies lnth\ (ltake\ n\ xs)\ m = lnth\ xs\ m$   
<proof>

**lemma** *ldropn-Suc-conv-ldropn*:

$enat\ n < llength\ xs \implies LCons\ (lnth\ xs\ n)\ (ldropn\ (Suc\ n)\ xs) = ldropn\ n\ xs$

*<proof>*

**lemma** *ltake-Suc-conv-snoc-lnth*:

*enat m < llength xs  $\implies$  ltake (enat (Suc m)) xs = lappend (ltake (enat m) xs)*  
*(LCons (lnth xs m) LNil)*

*<proof>*

**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>*

## 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>*

**lemma** *lmap-iterates*: *lmap f (iterates f x) = iterates f (f x)*

*<proof>*

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

*<proof>*

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

*<proof>*

**lemma** [*simp*]:

**fixes** *f* :: *'a  $\Rightarrow$  'a*

**shows** *lnull-funpow-lmap*: *lnull ((lmap f  $\overset{\sim}{\sim}$  n) xs)  $\longleftrightarrow$  lnull xs*

**and** *lhd-funpow-lmap*:  $\neg$  *lnull xs  $\implies$  lhd ((lmap f  $\overset{\sim}{\sim}$  n) xs) = (f  $\overset{\sim}{\sim}$  n) (lhd xs)*

**and** *ltl-funpow-lmap*:  $\neg$  *lnull xs  $\implies$  ltl ((lmap f  $\overset{\sim}{\sim}$  n) xs) = (lmap f  $\overset{\sim}{\sim}$  n) (ltl xs)*

*<proof>*

**lemma** *iterates-equality*:

**assumes** *h*:  $\bigwedge x. h x = LCons x (lmap f (h x))$

**shows** *h = iterates f*

*<proof>*

**lemma** *llength-iterates* [*simp*]: *llength (iterates f x) =  $\infty$*

*<proof>*

**lemma** *ldropn-iterates*: *ldropn n (iterates f x) = iterates f ((f  $\overset{\sim}{\sim}$  n) x)*

*<proof>*

**lemma** *ldrop-iterates*:  $ldrop (enat n) (iterates f x) = iterates f ((f \smallfrown n) x)$   
*<proof>*

**lemma** *lnth-iterates [simp]*:  $lnth (iterates f x) n = (f \smallfrown n) x$   
*<proof>*

**lemma** *lset-iterates*:  
 $lset (iterates f x) = \{(f \smallfrown n) x \mid n. True\}$   
*<proof>*

**lemma** *lset-repeat [simp]*:  $lset (repeat x) = \{x\}$   
*<proof>*

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

**lemma** *lstrict-prefix-code [code, simp]*:  
 $lstrict\text{-}prefix\ LNil\ LNil \longleftrightarrow False$   
 $lstrict\text{-}prefix\ LNil\ (LCons\ y\ ys) \longleftrightarrow True$   
 $lstrict\text{-}prefix\ (LCons\ x\ xs)\ LNil \longleftrightarrow False$   
 $lstrict\text{-}prefix\ (LCons\ x\ xs)\ (LCons\ y\ ys) \longleftrightarrow x = y \wedge lstrict\text{-}prefix\ xs\ ys$   
*<proof>*

**lemma** *lmap-lprefix*:  $xs \sqsubseteq ys \implies lmap\ f\ xs \sqsubseteq lmap\ f\ ys$   
*<proof>*

**lemma** *lprefix-llength-eq-imp-eq*:  
 $\llbracket xs \sqsubseteq ys; llength\ xs = llength\ ys \rrbracket \implies xs = ys$   
*<proof>*

**lemma** *lprefix-llength-le*:  $xs \sqsubseteq ys \implies llength\ xs \leq llength\ ys$   
*<proof>*

**lemma** *lstrict-prefix-llength-less*:  
**assumes** *lstrict-prefix xs ys*  
**shows**  $llength\ xs < llength\ ys$   
*<proof>*

**lemma** *lstrict-prefix-lfinite1*:  $lstrict\text{-}prefix\ xs\ ys \implies lfinite\ xs$   
*<proof>*

**lemma** *wfP-lstrict-prefix*:  $wfP\ lstrict\text{-}prefix$   
*<proof>*

**lemma** *lstrict-less-induct [case-names less]*:  
 $(\bigwedge xs. (\bigwedge ys. lstrict\text{-}prefix\ ys\ xs \implies P\ ys) \implies P\ xs) \implies P\ xs$   
*<proof>*



**lemma** *ltake-enat-eq-imp-eq*:  $(\bigwedge n. \text{ltake } (\text{enat } n) \text{ } xs = \text{ltake } (\text{enat } n) \text{ } ys) \implies xs = ys$   
 <proof>

**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  $\text{lprefix } xs \text{ } ys$   
 <proof>

**lemma** *lprefix-conv-lappend*:  $xs \sqsubseteq ys \iff (\exists zs. ys = \text{lappend } xs \text{ } zs)$  (is ?lhs  $\iff$  ?rhs)  
 <proof>

**lemma** *lappend-lprefixE*:  
 assumes  $\text{lappend } xs \text{ } ys \sqsubseteq zs$   
 obtains  $zs'$  where  $zs = \text{lappend } xs \text{ } zs'$   
 <proof>

**lemma** *lprefix-lfiniteD*:  
 $\llbracket xs \sqsubseteq ys; \text{lfinite } ys \rrbracket \implies \text{lfinite } xs$   
 <proof>

**lemma** *lprefix-lappendD*:  
 assumes  $xs \sqsubseteq \text{lappend } ys \text{ } zs$   
 shows  $xs \sqsubseteq ys \vee ys \sqsubseteq xs$   
 <proof>

**lemma** *lstrict-prefix-lappend-conv*:  
 $\text{lstrict-prefix } xs (\text{lappend } xs \text{ } ys) \iff \text{lfinite } xs \wedge \neg \text{lnull } ys$   
 <proof>

**lemma** *lprefix-llist-ofI*:  
 $\exists zs. ys = xs @ zs \implies \text{llist-of } xs \sqsubseteq \text{llist-of } ys$   
 <proof>

**lemma** *lprefix-llist-of [simp]*:  $\text{llist-of } xs \sqsubseteq \text{llist-of } ys \iff \text{prefix } xs \text{ } ys$   
 <proof>

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

**lemma** *lmap-lstrict-prefix*:  
 $\text{lstrict-prefix } xs \text{ } ys \implies \text{lstrict-prefix } (\text{lmap } f \text{ } xs) (\text{lmap } f \text{ } ys)$   
 <proof>

**lemma** *lprefix-lnthD*:

**assumes**  $xs \sqsubseteq ys$  **and**  $enat\ n < \text{length}\ xs$

**shows**  $\text{lnth}\ xs\ n = \text{lnth}\ ys\ n$

*<proof>*

**lemma** *lfinite-lSup-chain*:

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

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

*<proof>*

Setup for  $(\sqsubseteq)$  for Nitpick

**definition** *finite-lprefix* ::  $'a\ \text{list} \Rightarrow 'a\ \text{list} \Rightarrow \text{bool}$

**where** *finite-lprefix* =  $(\sqsubseteq)$

**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*  $xs\ (\text{LCons}\ y\ ys) \longleftrightarrow$

$xs = \text{LNil} \vee (\exists xs'. xs = \text{LCons}\ y\ xs' \wedge \text{finite-lprefix}\ xs'\ ys)$

*<proof>*

**lemma** *lprefix-nitpick-simps* [*nitpick-simp*]:

$xs \sqsubseteq ys = (\text{if}\ \text{lfinite}\ xs\ \text{then}\ \text{finite-lprefix}\ xs\ ys\ \text{else}\ xs = ys)$

*<proof>*

**hide-const** (**open**) *finite-lprefix*

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

## 2.17 Length of the longest common prefix

**lemma** *llcp-simps* [*simp*, *code*, *nitpick-simp*]:

**shows** *llcp-LNil1*:  $\text{llcp}\ \text{LNil}\ ys = 0$

**and** *llcp-LNil2*:  $\text{llcp}\ xs\ \text{LNil} = 0$

**and** *llcp-LCons*:  $\text{llcp}\ (\text{LCons}\ x\ xs)\ (\text{LCons}\ y\ ys) = (\text{if}\ x = y\ \text{then}\ \text{eSuc}\ (\text{llcp}\ xs\ ys)\ \text{else}\ 0)$

*<proof>*

**lemma** *llcp-eq-0-iff*:

$\text{llcp}\ xs\ ys = 0 \longleftrightarrow \text{lnull}\ xs \vee \text{lnull}\ ys \vee \text{lhd}\ xs \neq \text{lhd}\ ys$

*<proof>*

**lemma** *epred-llcp*:

$\llbracket \neg \text{lnull}\ xs; \neg \text{lnull}\ ys; \text{lhd}\ xs = \text{lhd}\ ys \rrbracket$

$\implies \text{epred}\ (\text{llcp}\ xs\ ys) = \text{llcp}\ (\text{ltl}\ xs)\ (\text{ltl}\ ys)$

*<proof>*

**lemma** *llcp-commute*:  $\text{llcp}\ xs\ ys = \text{llcp}\ ys\ xs$

*<proof>*

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

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

**lemma** *llcp-lprefix1* [simp]:  $xs \sqsubseteq ys \implies llcp\ xs\ ys = llength\ xs$   
 ⟨proof⟩

**lemma** *llcp-lprefix2* [simp]:  $ys \sqsubseteq xs \implies llcp\ xs\ ys = llength\ ys$   
 ⟨proof⟩

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

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

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

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

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

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

**lemma** *lnull-lzip* [simp]:  $lnull\ (lzip\ xs\ ys) \longleftrightarrow lnull\ xs \vee lnull\ ys$   
 ⟨proof⟩

**lemma** *lzip-eq-LNil-conv*:  $lzip\ xs\ ys = LNil \longleftrightarrow xs = LNil \vee ys = LNil$   
 ⟨proof⟩

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

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

**lemma** *lzip-lappend*:

$l\text{length } xs = l\text{length } us$   
 $\implies lzip (lappend\ xs\ ys) (lappend\ us\ vs) = lappend (lzip\ xs\ us) (lzip\ ys\ vs)$   
<proof>

**lemma** *llength-lzip* [simp]:

$l\text{length } (lzip\ xs\ ys) = \min (l\text{length } xs) (l\text{length } ys)$   
<proof>

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

<proof>

**lemma** *ldropn-lzip* [simp]:

$ldropn\ n\ (lzip\ xs\ ys) = lzip (ldropn\ n\ xs) (ldropn\ n\ ys)$   
<proof>

**lemma**

**fixes**  $F$

**defines**  $F \equiv \lambda lzip\ (xs,\ ys). \text{ case } xs \text{ of } LNil \Rightarrow LNil \mid LCons\ x\ xs' \Rightarrow \text{ case } ys \text{ of } LNil \Rightarrow LNil \mid LCons\ y\ ys' \Rightarrow LCons\ (x,\ y) (curry\ lzip\ xs'\ ys')$

**shows** *lzip-conv-fixp*:  $lzip \equiv curry (ccpo.fixp (fun-lub\ lSup) (fun-ord (\sqsubseteq)) F)$  (**is** ?lhs  $\equiv$  ?rhs)

**and** *lzip-mono*:  $mono\_l\text{list } (\lambda lzip.\ F\ lzip\ xs)$  (**is** ?mono  $xs$ )  
<proof>

**lemma** *monotone-lzip*:  $monotone (rel\text{-prod } (\sqsubseteq) (\sqsubseteq)) (\sqsubseteq) (case\text{-prod } lzip)$

<proof>

**lemma** *mono2mono-lzip1* [THEN *l\text{list}.mono2mono*, *cont-intro*, *simp*]:

**shows** *monotone-lzip1*:  $monotone (\sqsubseteq) (\sqsubseteq) (\lambda xs.\ lzip\ xs\ ys)$   
<proof>

**lemma** *mono2mono-lzip2* [THEN *l\text{list}.mono2mono*, *cont-intro*, *simp*]:

**shows** *monotone-lzip2*:  $monotone (\sqsubseteq) (\sqsubseteq) (\lambda ys.\ lzip\ xs\ ys)$   
<proof>

**lemma** *mcont-lzip*:  $mcont (prod\text{-lub } lSup\ lSup) (rel\text{-prod } (\sqsubseteq) (\sqsubseteq)) lSup (\sqsubseteq) (case\text{-prod } lzip)$

<proof>

**lemma** *mcont2mcont-lzip1* [THEN *l\text{list}.mcont2mcont*, *cont-intro*, *simp*]:

**shows** *mcont-lzip1*:  $mcont\ lSup (\sqsubseteq) lSup (\sqsubseteq) (\lambda xs.\ lzip\ xs\ ys)$   
<proof>

**lemma** *mcont2mcont-lzip2* [THEN *l\text{list}.mcont2mcont*, *cont-intro*, *simp*]:

**shows** *mcont-lzip2*:  $mcont\ lSup (\sqsubseteq) lSup (\sqsubseteq) (\lambda ys.\ lzip\ xs\ ys)$   
<proof>

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

**lemma** *lzip-iterates*:

$lzip\ (iterates\ f\ x)\ (iterates\ g\ y) = iterates\ (\lambda(x, y). (f\ x, g\ y))\ (x, y)$   
 ⟨*proof*⟩

**lemma** *lzip-llist-of* [*simp*]:

$lzip\ (llist-of\ xs)\ (llist-of\ ys) = llist-of\ (zip\ xs\ ys)$   
 ⟨*proof*⟩

**lemma** *lnth-lzip*:

$\llbracket\ enat\ n < llength\ xs; enat\ n < llength\ ys\ \rrbracket$   
 $\implies lnth\ (lzip\ xs\ ys)\ n = (lnth\ xs\ n, lnth\ ys\ n)$   
 ⟨*proof*⟩

**lemma** *lset-lzip*:

$lset\ (lzip\ xs\ ys) =$   
 $\{(lnth\ xs\ n, lnth\ ys\ n) \mid n. enat\ n < \min\ (llength\ xs)\ (llength\ ys)\}$   
 ⟨*proof*⟩

**lemma** *lset-lzipD1*:  $(x, y) \in lset\ (lzip\ xs\ ys) \implies x \in lset\ xs$   
 ⟨*proof*⟩

**lemma** *lset-lzipD2*:  $(x, y) \in lset\ (lzip\ xs\ ys) \implies y \in lset\ ys$   
 ⟨*proof*⟩

**lemma** *lset-lzip-same* [*simp*]:  $lset\ (lzip\ xs\ xs) = (\lambda x. (x, x))\ `lset\ xs$   
 ⟨*proof*⟩

**lemma** *lfinite-lzip* [*simp*]:

$lfinite\ (lzip\ xs\ ys) \longleftrightarrow lfinite\ xs \vee lfinite\ ys$  (**is** ?*lhs*  $\longleftrightarrow$  ?*rhs*)  
 ⟨*proof*⟩

**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*⟩

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

$lzip\ (lmap\ f\ xs)\ (lmap\ g\ ys) = lmap\ (\lambda(x, y). (f\ x, g\ y))\ (lzip\ xs\ ys)$   
 ⟨*proof*⟩

**lemma** *lzip-lmap1*:

$lzip\ (lmap\ f\ xs)\ ys = lmap\ (\lambda(x, y). (f\ x, y))\ (lzip\ xs\ ys)$   
 ⟨*proof*⟩

**lemma** *lzip-lmap2*:

$lzip\ xs\ (lmap\ f\ ys) = lmap\ (\lambda(x, y). (x, f\ y))\ (lzip\ xs\ ys)$   
(proof)

**lemma** *lmap-fst-lzip-conv-ltake*:

$lmap\ fst\ (lzip\ xs\ ys) = ltake\ (min\ (llength\ xs)\ (llength\ ys))\ xs$   
(proof)

**lemma** *lmap-snd-lzip-conv-ltake*:

$lmap\ snd\ (lzip\ xs\ ys) = ltake\ (min\ (llength\ xs)\ (llength\ ys))\ ys$   
(proof)

**lemma** *lzip-conv-lzip-ltake-min-llength*:

$lzip\ xs\ ys =$   
 $lzip\ (ltake\ (min\ (llength\ xs)\ (llength\ ys))\ xs)$   
 $(ltake\ (min\ (llength\ xs)\ (llength\ ys))\ ys)$   
(proof)

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

**lemma** *ltakeWhile-simps* [*simp*, *code*, *nitpick-simp*]:

**shows** *ltakeWhile-LNil*:  $ltakeWhile\ P\ LNil = LNil$   
**and** *ltakeWhile-LCons*:  $ltakeWhile\ P\ (LCons\ x\ xs) = (if\ P\ x\ then\ LCons\ x\ (ltakeWhile\ P\ xs)\ else\ LNil)$   
(proof)

**lemma** *ldropWhile-simps* [*simp*, *code*]:

**shows** *ldropWhile-LNil*:  $ldropWhile\ P\ LNil = LNil$   
**and** *ldropWhile-LCons*:  $ldropWhile\ P\ (LCons\ x\ xs) = (if\ P\ x\ then\ ldropWhile\ P\ xs\ else\ LCons\ x\ xs)$   
(proof)

**lemma** *fixes f F P*

**defines**  $F \equiv \lambda takeWhile\ xs. case\ xs\ of\ LNil \Rightarrow LNil \mid LCons\ x\ xs \Rightarrow if\ P\ x\ then\ LCons\ x\ (takeWhile\ xs)\ else\ LNil$   
**shows** *ltakeWhile-conv-fixp*:  $ltakeWhile\ P \equiv cppo.fixp\ (fun-lub\ lSup)\ (fun-ord\ lprefix)\ F\ (is\ ?lhs \equiv ?rhs)$   
**and** *ltakeWhile-mono*:  $\bigwedge xs. mono-llist\ (\lambda takeWhile. F\ takeWhile\ xs)\ (is\ PROP\ ?mono)$   
(proof)

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

**shows** *monotone-ltakeWhile*:  $monotone\ lprefix\ lprefix\ (ltakeWhile\ P)$   
(proof)

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

**shows** *mcont-ltakeWhile*:  $mcont\ lSup\ lprefix\ lSup\ lprefix\ (ltakeWhile\ P)$   
(proof)

**lemma** *mono-llist-ltakeWhile* [*partial-function-mono*]:  
 $mono-llist\ F \implies mono-llist\ (\lambda f. ltakeWhile\ P\ (F\ f))$   
 ⟨*proof*⟩

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

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

**lemma** *lnull-ltakeWhile* [*simp*]:  $lnull\ (ltakeWhile\ P\ xs) \longleftrightarrow (\neg\ lnull\ xs \longrightarrow \neg\ P\ (lhd\ xs))$   
 ⟨*proof*⟩

**lemma** *ltakeWhile-eq-LNil-iff*:  $ltakeWhile\ P\ xs = LNil \longleftrightarrow (xs \neq LNil \longrightarrow \neg\ P\ (lhd\ xs))$   
 ⟨*proof*⟩

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

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

**lemma** *lprefix-ltakeWhile*:  $ltakeWhile\ P\ xs \sqsubseteq xs$   
 ⟨*proof*⟩

**lemma** *llength-ltakeWhile-le*:  $llength\ (ltakeWhile\ P\ xs) \leq llength\ xs$   
 ⟨*proof*⟩

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

**lemma** *ltakeWhile-all*:  $\forall x \in lset\ xs. P\ x \implies ltakeWhile\ P\ xs = xs$   
 ⟨*proof*⟩

**lemma** *lset-ltakeWhileD*:  
**assumes**  $x \in lset\ (ltakeWhile\ P\ xs)$   
**shows**  $x \in lset\ xs \wedge P\ x$   
 ⟨*proof*⟩

**lemma** *lset-ltakeWhile-subset*:  
 $lset\ (ltakeWhile\ P\ xs) \subseteq lset\ xs \cap \{x. P\ x\}$   
 ⟨*proof*⟩

**lemma** *ltakeWhile-all-conv*:  $ltakeWhile\ P\ xs = xs \longleftrightarrow lset\ xs \subseteq \{x.\ P\ x\}$   
 ⟨proof⟩

**lemma** *llength-ltakeWhile-all*:  $llength\ (ltakeWhile\ P\ xs) = llength\ xs \longleftrightarrow ltakeWhile\ P\ xs = xs$   
 ⟨proof⟩

**lemma** *ldropWhile-eq-LNil-iff*:  $ldropWhile\ P\ xs = LNil \longleftrightarrow (\forall x \in lset\ xs.\ P\ x)$   
 ⟨proof⟩

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

**lemma** *lset-ldropWhile-subset*:  
 $lset\ (ldropWhile\ P\ xs) \subseteq lset\ xs$   
 ⟨proof⟩

**lemma** *in-lset-ldropWhileD*:  $x \in lset\ (ldropWhile\ P\ xs) \implies x \in lset\ xs$   
 ⟨proof⟩

**lemma** *ltakeWhile-lmap*:  $ltakeWhile\ P\ (lmap\ f\ xs) = lmap\ f\ (ltakeWhile\ (P \circ f)\ xs)$   
 ⟨proof⟩

**lemma** *ldropWhile-lmap*:  $ldropWhile\ P\ (lmap\ f\ xs) = lmap\ f\ (ldropWhile\ (P \circ f)\ xs)$   
 ⟨proof⟩

**lemma** *llength-ltakeWhile-lt-iff*:  $llength\ (ltakeWhile\ P\ xs) < llength\ xs \longleftrightarrow (\exists x \in lset\ xs.\ \neg P\ x)$   
 (is ?lhs  $\longleftrightarrow$  ?rhs)  
 ⟨proof⟩

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

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

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

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

**lemma** *lappend-ltakeWhile-ldropWhile [simp]*:  
 $lappend\ (ltakeWhile\ P\ xs)\ (ldropWhile\ P\ xs) = xs$   
 ⟨proof⟩



**lemma** *ltakeWhile-lappend*:

$ltakeWhile\ P\ (lappend\ xs\ ys) =$   
(if  $\exists x \in lset\ xs.\ \neg\ P\ x$  then  $ltakeWhile\ P\ xs$   
else  $lappend\ xs\ (ltakeWhile\ P\ ys)$ )  
(proof)

**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)

**lemma** *lfinite-ltakeWhile*:

$lfinite\ (ltakeWhile\ P\ xs) \longleftrightarrow lfinite\ xs \vee (\exists x \in lset\ xs.\ \neg\ P\ x)$  (is ?lhs  $\longleftrightarrow$  ?rhs)  
(proof)

**lemma** *llength-ltakeWhile-eq-infinity*:

$llength\ (ltakeWhile\ P\ xs) = \infty \longleftrightarrow \neg\ lfinite\ xs \wedge ltakeWhile\ P\ xs = xs$   
(proof)

**lemma** *llength-ltakeWhile-eq-infinity'*:

$llength\ (ltakeWhile\ P\ xs) = \infty \longleftrightarrow \neg\ lfinite\ xs \wedge (\forall x \in lset\ xs.\ P\ x)$   
(proof)

**lemma** *lzip-ltakeWhile-fst*:  $lzip\ (ltakeWhile\ P\ xs)\ ys = ltakeWhile\ (P \circ fst)\ (lzip\ xs\ ys)$   
(proof)

**lemma** *lzip-ltakeWhile-snd*:  $lzip\ xs\ (ltakeWhile\ P\ ys) = ltakeWhile\ (P \circ snd)\ (lzip\ xs\ ys)$   
(proof)

**lemma** *ltakeWhile-lappend1*:

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

**lemma** *ltakeWhile-lappend2*:

$lset\ xs \subseteq \{x.\ P\ x\}$   
 $\implies ltakeWhile\ P\ (lappend\ xs\ ys) = lappend\ xs\ (ltakeWhile\ P\ ys)$   
(proof)

**lemma** *ltakeWhile-cong [cong, fundef-cong]*:

**assumes**  $xs: xs = ys$   
**and**  $PQ: \bigwedge x. x \in lset\ ys \implies P\ x = Q\ x$   
**shows**  $ltakeWhile\ P\ xs = ltakeWhile\ Q\ ys$   
(proof)

**lemma** *lnth-llength-ltakeWhile*:

**assumes**  $len: llength (ltakeWhile P xs) < llength xs$   
**shows**  $\neg P (lnth xs (the-enat (llength (ltakeWhile P xs))))$   
 $\langle proof \rangle$

**lemma assumes**  $\exists x \in lset xs. \neg P x$   
**shows**  $lhd\text{-}ldropWhile: \neg P (lhd (ldropWhile P xs))$  (**is** *?thesis1*)  
**and**  $lhd\text{-}ldropWhile\text{-}in\text{-}lset: lhd (ldropWhile P xs) \in lset xs$  (**is** *?thesis2*)  
 $\langle proof \rangle$

**lemma**  $ldropWhile\text{-}eq\text{-}ldrop:$   
 $ldropWhile P xs = ldrop (llength (ltakeWhile P xs)) xs$   
(**is** *?lhs = ?rhs*)  
 $\langle proof \rangle$

**lemma**  $ldropWhile\text{-}cong$  [*cong*]:  
 $\llbracket xs = ys; \bigwedge x. x \in lset ys \implies P x = Q x \rrbracket \implies ldropWhile P xs = ldropWhile Q ys$   
 $\langle proof \rangle$

**lemma**  $ltakeWhile\text{-}repeat:$   
 $ltakeWhile P (repeat x) = (if P x then repeat x else LNil)$   
 $\langle proof \rangle$

**lemma**  $ldropWhile\text{-}repeat: ldropWhile P (repeat x) = (if P x then LNil else repeat x)$   
 $\langle proof \rangle$

**lemma**  $lfinite\text{-}ldropWhile: lfinite (ldropWhile P xs) \longleftrightarrow (\exists x \in lset xs. \neg P x) \longrightarrow lfinite xs$   
 $\langle proof \rangle$

**lemma**  $llength\text{-}ldropWhile:$   
 $llength (ldropWhile P xs) =$   
 $(if \exists x \in lset xs. \neg P x then llength xs - llength (ltakeWhile P xs) else 0)$   
 $\langle proof \rangle$

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

## 2.20 *l*list-all2

**lemmas**  $l\text{list}\text{-}all2\text{-}LNil\text{-}LNil = l\text{list}\text{-}rel\text{-}inject(1)$

**lemmas**  $l\text{list}\text{-}all2\text{-}LNil\text{-}LCons = l\text{list}\text{-}rel\text{-}distinct(1)$

**lemmas**  $l\text{list}\text{-}all2\text{-}LCons\text{-}LNil = l\text{list}\text{-}rel\text{-}distinct(2)$

**lemmas**  $l\text{list}\text{-}all2\text{-}LCons\text{-}LCons = l\text{list}\text{-}rel\text{-}inject(2)$

**lemma**  $l\text{list}\text{-}all2\text{-}LNil1$  [*simp*]:  $l\text{list}\text{-}all2 P LNil xs \longleftrightarrow xs = LNil$

$\langle \text{proof} \rangle$

**lemma** *llist-all2-LNil2* [*simp*]:  $\text{llist-all2 } P \text{ } xs \text{ } LNil \longleftrightarrow xs = LNil$   
 $\langle \text{proof} \rangle$

**lemma** *llist-all2-LCons1*:

$\text{llist-all2 } P \text{ } (LCons \ x \ xs) \ ys \longleftrightarrow (\exists y \ ys'. \ ys = LCons \ y \ ys' \wedge P \ x \ y \wedge \text{llist-all2 } P \ xs \ ys')$   
 $\langle \text{proof} \rangle$

**lemma** *llist-all2-LCons2*:

$\text{llist-all2 } P \ xs \ (LCons \ y \ ys) \longleftrightarrow (\exists x \ xs'. \ xs = LCons \ x \ xs' \wedge P \ x \ y \wedge \text{llist-all2 } P \ xs' \ ys)$   
 $\langle \text{proof} \rangle$

**lemma** *llist-all2-llist-of* [*simp*]:

$\text{llist-all2 } P \ (\text{llist-of } \ xs) \ (\text{llist-of } \ ys) = \text{list-all2 } P \ xs \ ys$   
 $\langle \text{proof} \rangle$

**lemma** *llist-all2-conv-lzip*:

$\text{llist-all2 } P \ xs \ ys \longleftrightarrow \text{llength } xs = \text{llength } ys \wedge (\forall (x, y) \in \text{lset } (\text{lzip } \ xs \ ys). \ P \ x \ y)$   
 $\langle \text{proof} \rangle$

**lemma** *llist-all2-llengthD*:

$\text{llist-all2 } P \ xs \ ys \implies \text{llength } xs = \text{llength } ys$   
 $\langle \text{proof} \rangle$

**lemma** *llist-all2-lnullD*:  $\text{llist-all2 } P \ xs \ ys \implies \text{lnull } xs \longleftrightarrow \text{lnull } ys$

$\langle \text{proof} \rangle$

**lemma** *llist-all2-all-lnthI*:

$\llbracket \text{llength } xs = \text{llength } ys; \bigwedge n. \text{enat } n < \text{llength } xs \implies P \ (\text{lnth } \ xs \ n) \ (\text{lnth } \ ys \ n) \rrbracket$   
 $\implies \text{llist-all2 } P \ xs \ ys$   
 $\langle \text{proof} \rangle$

**lemma** *llist-all2-lnthD*:

$\llbracket \text{llist-all2 } P \ xs \ ys; \text{enat } n < \text{llength } xs \rrbracket \implies P \ (\text{lnth } \ xs \ n) \ (\text{lnth } \ ys \ n)$   
 $\langle \text{proof} \rangle$

**lemma** *llist-all2-lnthD2*:

$\llbracket \text{llist-all2 } P \ xs \ ys; \text{enat } n < \text{llength } ys \rrbracket \implies P \ (\text{lnth } \ xs \ n) \ (\text{lnth } \ ys \ n)$   
 $\langle \text{proof} \rangle$

**lemma** *llist-all2-conv-all-lnth*:

$\text{llist-all2 } P \ xs \ ys \longleftrightarrow$   
 $\text{llength } xs = \text{llength } ys \wedge$   
 $(\forall n. \text{enat } n < \text{llength } ys \longrightarrow P \ (\text{lnth } \ xs \ n) \ (\text{lnth } \ ys \ n))$   
 $\langle \text{proof} \rangle$

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

**lemma** *llist-all2-refl*:  
 $(\bigwedge x. x \in \text{lset } xs \implies P \ x \ x) \implies l\text{list-all2 } P \ xs \ xs$   
 ⟨*proof*⟩

**lemma** *llist-all2-lmap1*:  
 $l\text{list-all2 } P \ (\text{lmap } f \ xs) \ ys \longleftrightarrow l\text{list-all2 } (\lambda x. P \ (f \ x)) \ xs \ ys$   
 ⟨*proof*⟩

**lemma** *llist-all2-lmap2*:  
 $l\text{list-all2 } P \ xs \ (\text{lmap } g \ ys) \longleftrightarrow l\text{list-all2 } (\lambda x \ y. P \ x \ (g \ y)) \ xs \ ys$   
 ⟨*proof*⟩

**lemma** *llist-all2-lfiniteD*:  
 $l\text{list-all2 } P \ xs \ ys \implies \text{lfinite } xs \longleftrightarrow \text{lfinite } ys$   
 ⟨*proof*⟩

**lemma** *llist-all2-coinduct*[*consumes 1, case-names LNil LCons, case-conclusion LCons lhd ltl, coinduct pred*]:  
**assumes** *major*:  $X \ xs \ ys$   
**and** *step*:  
 $\bigwedge xs \ ys. X \ xs \ ys \implies \text{lnull } xs \longleftrightarrow \text{lnull } ys$   
 $\bigwedge xs \ ys. \llbracket X \ xs \ ys; \neg \text{lnull } xs; \neg \text{lnull } ys \rrbracket \implies P \ (\text{lhd } xs) \ (\text{lhd } ys) \wedge (X \ (\text{ltl } xs) \ (\text{ltl } ys) \vee l\text{list-all2 } P \ (\text{ltl } xs) \ (\text{ltl } ys))$   
**shows**  $l\text{list-all2 } P \ xs \ ys$   
 ⟨*proof*⟩

**lemma** *llist-all2-cases*[*consumes 1, case-names LNil LCons, cases pred*]:  
**assumes**  $l\text{list-all2 } P \ xs \ ys$   
**obtains**  $(\text{LNil}) \ xs = \text{LNil } ys = \text{LNil}$   
 |  $(\text{LCons}) \ x \ xs' \ y \ ys'$   
**where**  $xs = \text{LCons } x \ xs' \ \text{and } ys = \text{LCons } y \ ys'$   
**and**  $P \ x \ y \ \text{and } l\text{list-all2 } P \ xs' \ ys'$   
 ⟨*proof*⟩

**lemma** *llist-all2-mono*:  
 $\llbracket l\text{list-all2 } P \ xs \ ys; \bigwedge x \ y. P \ x \ y \implies P' \ x \ y \rrbracket \implies l\text{list-all2 } P' \ xs \ ys$   
 ⟨*proof*⟩

**lemma** *llist-all2-left*:  $l\text{list-all2 } (\lambda x - . P \ x) \ xs \ ys \longleftrightarrow \text{length } xs = \text{length } ys \wedge (\forall x \in \text{lset } xs. P \ x)$   
 ⟨*proof*⟩

**lemma** *llist-all2-right*:  $l\text{list-all2 } (\lambda - . P) \ xs \ ys \longleftrightarrow \text{length } xs = \text{length } ys \wedge (\forall x \in \text{lset } ys. P \ x)$

*<proof>*

**lemma** *lsetD1*:  $\llbracket \text{llist-all2 } P \text{ } xs \text{ } ys; x \in \text{lset } xs \rrbracket \implies \exists y \in \text{lset } ys. P \ x \ y$   
*<proof>*

**lemma** *lsetD2*:  $\llbracket \text{llist-all2 } P \text{ } xs \text{ } ys; y \in \text{lset } ys \rrbracket \implies \exists x \in \text{lset } xs. P \ x \ y$   
*<proof>*

**lemma** *conj*:  
 $\text{llist-all2 } (\lambda x \ y. P \ x \ y \wedge Q \ x \ y) \ xs \ ys \longleftrightarrow \text{llist-all2 } P \ xs \ ys \wedge \text{llist-all2 } Q \ xs \ ys$   
*<proof>*

**lemma** *lhdD*:  
 $\llbracket \text{llist-all2 } P \text{ } xs \text{ } ys; \neg \text{lnull } xs \rrbracket \implies P \ (\text{lhd } xs) \ (\text{lhd } ys)$   
*<proof>*

**lemma** *lhdD2*:  
 $\llbracket \text{llist-all2 } P \text{ } xs \text{ } ys; \neg \text{lnull } ys \rrbracket \implies P \ (\text{lhd } xs) \ (\text{lhd } ys)$   
*<proof>*

**lemma** *ltlI*:  
 $\text{llist-all2 } P \text{ } xs \text{ } ys \implies \text{llist-all2 } P \ (\text{ltl } xs) \ (\text{ltl } ys)$   
*<proof>*

**lemma** *lappendI*:  
**assumes** 1:  $\text{llist-all2 } P \text{ } xs \text{ } ys$   
**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>*

**lemma** *lappend1D*:  
**assumes**  $\text{llist-all2 } P \ (\text{lappend } xs \text{ } xs') \ ys$   
**shows**  $\text{llist-all2 } P \ xs \ (\text{ltake } (\text{llength } xs) \ ys)$   
**and**  $\text{lfinite } xs \implies \text{llist-all2 } P \ xs' \ (\text{ldrop } (\text{llength } xs) \ ys)$   
*<proof>*

**lemma** *lmap-eq-lmap-conv-llist-all2*:  
 $\text{lmap } f \text{ } xs = \text{lmap } g \text{ } ys \longleftrightarrow \text{llist-all2 } (\lambda x \ y. f \ x = g \ y) \ xs \ ys$  (**is** ?lhs  $\longleftrightarrow$  ?rhs)  
*<proof>*

**lemma** *expand*:  
 $\llbracket \text{lnull } xs \longleftrightarrow \text{lnull } ys; \llbracket \neg \text{lnull } xs; \neg \text{lnull } ys \rrbracket \implies P \ (\text{lhd } xs) \ (\text{lhd } ys) \wedge \text{llist-all2 } P \ (\text{ltl } xs) \ (\text{ltl } ys) \rrbracket$   
 $\implies \text{llist-all2 } P \ xs \ ys$   
*<proof>*

**lemma** *llength-ltake-WhileD*:  
**assumes** *major*:  $\text{llist-all2 } P \text{ } xs \text{ } ys$   
**and** *Q*:  $\bigwedge x \ y. P \ x \ y \implies Q1 \ x \longleftrightarrow Q2 \ y$

**shows**  $l\text{length } (l\text{takeWhile } Q1\ xs) = l\text{length } (l\text{takeWhile } Q2\ ys)$   
 ⟨proof⟩

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

[[ *l\text{list-all2 } P\ xs\ ys; l\text{list-all2 } P'\ xs'\ ys'* ]]  
 $\implies l\text{list-all2 } (rel\text{-prod } P\ P')\ (l\text{zip } xs\ xs')\ (l\text{zip } ys\ ys')$   
 ⟨proof⟩

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

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

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

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

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

$l\text{list-all2 } P\ xs\ ys \implies l\text{list-all2 } P\ (l\text{drop } n\ xs)\ (l\text{drop } n\ ys)$   
 ⟨proof⟩

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

**assumes** *Complete-Partial-Order.chain* ( $rel\text{-prod } (\sqsubseteq) (\sqsubseteq)$ )  $Y \forall (xs, ys) \in Y. l\text{list-all2 } P\ xs\ ys$   
**shows**  $l\text{list-all2 } P\ (l\text{Sup } (fst \text{ ' } Y))\ (l\text{Sup } (snd \text{ ' } Y))$   
 ⟨proof⟩

**lemma** *admissible-l\text{list-all2}* [*cont-intro, simp*]:

**assumes**  $f: m\text{cont } lub\ ord\ l\text{Sup } (\sqsubseteq) (\lambda x. f\ x)$   
**and**  $g: m\text{cont } lub\ ord\ l\text{Sup } (\sqsubseteq) (\lambda x. g\ x)$   
**shows**  $ccpo.admissible\ lub\ ord (\lambda x. l\text{list-all2 } P\ (f\ x)\ (g\ x))$   
 ⟨proof⟩

**lemmas** [*cont-intro*] =

$ccpo.m\text{cont}2m\text{cont}[OF\ l\text{list-ccpo} - m\text{cont-fst}]$   
 $ccpo.m\text{cont}2m\text{cont}[OF\ l\text{list-ccpo} - m\text{cont-snd}]$

**lemmas** *ldropWhile-fixp-parallel-induct* =

$parallel\text{-fixp-induct-1-1}[OF\ l\text{list-partial-function-definitions } l\text{list-partial-function-definitions}$   
 $ldropWhile.mono\ ldropWhile.mono\ ldropWhile-def\ ldropWhile-def, case-names$   
 $adm\ LNil\ step]$

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

**assumes**  $*$ :  $l\text{list-all2 } P\ xs\ ys$   
**and**  $Q: \bigwedge x\ y. P\ x\ y \implies Q1\ x \longleftrightarrow Q2\ y$   
**shows**  $l\text{list-all2 } P\ (l\text{dropWhile } Q1\ xs)\ (l\text{dropWhile } Q2\ ys)$   
 — cannot prove this with parallel induction over  $xs$  and  $ys$  because  $\lambda x. \neg l\text{list-all2 } P\ (f\ x)\ (g\ x)$  is not admissible.  
 ⟨proof⟩

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

**lemma** *l*list-all2-trans:  
 [ *l*list-all2  $P$   $xs$   $ys$ ; *l*list-all2  $P$   $ys$   $zs$ ; transp  $P$  ]  
 $\implies$  *l*list-all2  $P$   $xs$   $zs$   
 ⟨proof⟩

## 2.21 The last element *llast*

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

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

**lemma** *llast-linfinite*:  $\neg$  *l*finite  $xs \implies$  *llast*  $xs$  = undefined  
 ⟨proof⟩

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

**lemma** *llast-lappend*:  
*llast*  $(lappend xs ys)$  = (if *l*null  $ys$  then *llast*  $xs$  else if *l*finite  $xs$  then *llast*  $ys$  else  
 undefined)  
 ⟨proof⟩

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

**lemma** *llast-ldropn*:  $\text{enat } n < \text{llength } xs \implies$  *llast*  $(ldropn n xs)$  = *llast*  $xs$   
 ⟨proof⟩

**lemma** *llast-ldrop*:  
 assumes  $n < \text{llength } xs$   
 shows *llast*  $(ldrop n xs)$  = *llast*  $xs$   
 ⟨proof⟩

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

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

**lemma** *llast-lmap*:  
 assumes *l*finite  $xs \neg$  *l*null  $xs$

**shows**  $llast (lmap f xs) = f (llast xs)$   
 ⟨proof⟩

## 2.22 Distinct lazy lists *ldistinct*

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

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

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

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

**assumes**  $X xs$   
**and step:**  $\bigwedge xs. [ X xs; \neg lnull xs ]$   
 $\implies lhd xs \notin lset (ltl xs) \wedge (X (ltl xs) \vee ldistinct (ltl xs))$   
**shows**  $ldistinct xs$   
 ⟨proof⟩

**lemma** *ldistinct-lhdD*:  
 $[ ldistinct xs; \neg lnull xs ] \implies lhd xs \notin lset (ltl xs)$   
 ⟨proof⟩

**lemma** *ldistinct-ltlI*:  
 $ldistinct xs \implies ldistinct (ltl xs)$   
 ⟨proof⟩

**lemma** *ldistinct-lSup*:  
 $[ Complete-Partial-Order.chain (\sqsubseteq) Y; \forall xs \in Y. ldistinct xs ]$   
 $\implies ldistinct (lSup Y)$   
 ⟨proof⟩

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

**lemma** *ldistinct-lappend*:  
 $ldistinct (lappend xs ys) \longleftrightarrow ldistinct xs \wedge (lfinite xs \longrightarrow ldistinct ys \wedge lset xs \cap lset ys = \{\})$   
 (is ?lhs = ?rhs)  
 ⟨proof⟩

**lemma** *ldistinct-lprefix*:



$\llbracket \text{ldistinct } xs; ys \sqsubseteq xs \rrbracket \implies \text{ldistinct } ys$   
 <proof>

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

**lemma** *ldistinct-ltake*:  $\text{ldistinct } xs \implies \text{ldistinct } (\text{ltake } n \text{ } xs)$   
 <proof>

**lemma** *ldistinct-ldropn*:  
 $\text{ldistinct } xs \implies \text{ldistinct } (\text{ldropn } n \text{ } xs)$   
 <proof>

**lemma** *ldistinct-ldrop*:  $\text{ldistinct } xs \implies \text{ldistinct } (\text{ldrop } n \text{ } xs)$   
 <proof>

**lemma** *ldistinct-conv-lnth*:  
 $\text{ldistinct } xs \longleftrightarrow (\forall i j. \text{enat } i < \text{llength } xs \longrightarrow \text{enat } j < \text{llength } xs \longrightarrow i \neq j \longrightarrow$   
 $\text{lnth } xs \text{ } i \neq \text{lnth } xs \text{ } j)$   
 (is ?lhs  $\longleftrightarrow$  ?rhs)  
 <proof>

**lemma** *ldistinct-lmap* [*simp*]:  
 $\text{ldistinct } (\text{lmap } f \text{ } xs) \longleftrightarrow \text{ldistinct } xs \wedge \text{inj-on } f \text{ } (\text{lset } xs)$   
 (is ?lhs  $\longleftrightarrow$  ?rhs)  
 <proof>

**lemma** *ldistinct-lzipI1*:  $\text{ldistinct } xs \implies \text{ldistinct } (\text{lzip } xs \text{ } ys)$   
 <proof>

**lemma** *ldistinct-lzipI2*:  $\text{ldistinct } ys \implies \text{ldistinct } (\text{lzip } xs \text{ } ys)$   
 <proof>

## 2.23 Sortedness *lsorted*

**context** *ord begin*

**coinductive** *lsorted* :: 'a list  $\Rightarrow$  bool

**where**

$\text{LNil}$  [*simp*]: *lsorted*  $\text{LNil}$   
 | *Singleton* [*simp*]: *lsorted* ( $\text{LCons } x \text{ } \text{LNil}$ )  
 | *LCons-LCons*:  $\llbracket x \leq y; \text{sorted } (\text{LCons } y \text{ } xs) \rrbracket \implies \text{sorted } (\text{LCons } x \text{ } (\text{LCons } y \text{ } xs))$

**inductive-simps** *lsorted-LCons-LCons* [*simp*]:  
 $\text{sorted } (\text{LCons } x \text{ } (\text{LCons } y \text{ } xs))$

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

$lsorted\ LNil$   
 $lsorted\ (LCons\ x\ LNil)$   
 $lsorted\ (LCons\ x\ (LCons\ y\ xs))$

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

**assumes** *major*:  $X\ xs$   
**and step**:  $\bigwedge xs. \llbracket X\ xs; \neg\ lnull\ xs; \neg\ lnull\ (ltl\ xs) \rrbracket \implies lhd\ xs \leq lhd\ (ltl\ xs) \wedge (X\ (ltl\ xs) \vee lsorted\ (ltl\ xs))$   
**shows** *sorted xs*  
 $\langle proof \rangle$

**lemma** *sorted-ltlI*:  $lsorted\ xs \implies lsorted\ (ltl\ xs)$   
 $\langle proof \rangle$

**lemma** *sorted-lhdD*:  
 $\llbracket lsorted\ xs; \neg\ lnull\ xs; \neg\ lnull\ (ltl\ xs) \rrbracket \implies lhd\ xs \leq lhd\ (ltl\ xs)$   
 $\langle proof \rangle$

**lemma** *sorted-LCons'*:  
 $lsorted\ (LCons\ x\ xs) \longleftrightarrow (\neg\ lnull\ xs \longrightarrow x \leq lhd\ xs \wedge lsorted\ xs)$   
 $\langle proof \rangle$

**lemma** *sorted-lSup*:  
 $\llbracket Complete-Partial-Order.chain\ (\sqsubseteq)\ Y; \forall xs \in Y. lsorted\ xs \rrbracket$   
 $\implies lsorted\ (lSup\ Y)$   
 $\langle proof \rangle$

**lemma** *sorted-lprefixD*:  
 $\llbracket xs \sqsubseteq ys; lsorted\ ys \rrbracket \implies lsorted\ xs$   
 $\langle proof \rangle$

**lemma** *admissible-sorted* [cont-intro, simp]:  
**assumes** *mcont*:  $mcont\ lub\ ord\ lSup\ (\sqsubseteq)\ (\lambda x. f\ x)$   
**and** *ccpo*:  $class.ccpo\ lub\ ord\ (mk-less\ ord)$   
**shows**  $ccpo.admissible\ lub\ ord\ (\lambda x. lsorted\ (f\ x))$   
 $\langle proof \rangle$

**lemma** *admissible-not-sorted*[THEN *admissible-subst*, cont-intro, simp]:  
 $ccpo.admissible\ lSup\ (\sqsubseteq)\ (\lambda xs. \neg\ lsorted\ xs)$   
 $\langle proof \rangle$

**lemma** *sorted-ltake* [simp]:  $lsorted\ xs \implies lsorted\ (ltake\ n\ xs)$   
 $\langle proof \rangle$

**lemma** *sorted-ldropn* [simp]:  $lsorted\ xs \implies lsorted\ (ldropn\ n\ xs)$   
 $\langle proof \rangle$

**lemma** *sorted-ldrop* [simp]:  $lsorted\ xs \implies lsorted\ (ldrop\ n\ xs)$

*<proof>*

**end**

**declare**

*ord.lsorted-code* [*code*]

*ord.admissible-lsorted* [*cont-intro, simp*]

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

**context** *preorder* **begin**

**lemma** *lsorted-LCons*:

*lsorted (LCons x xs)  $\longleftrightarrow$  lsorted xs  $\wedge$  ( $\forall y \in \text{lset } xs. x \leq y$ ) (is ?lhs  $\longleftrightarrow$  ?rhs)*  
*<proof>*

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

**assumes** *major*:  $X \text{ xs}$

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

**shows** *lsorted xs*

*<proof>*

**lemma** *lsortedD*:  $\llbracket \text{lsorted } xs; \neg \text{lnull } xs; y \in \text{lset } (\text{ltl } xs) \rrbracket \implies \text{lhd } xs \leq y$   
*<proof>*

**end**

**lemma** *lsorted-lmap'*:

**assumes** *ord.lsorted* *orda xs monotone* *orda ordb f*

**shows** *ord.lsorted* *ordb (lmap f xs)*

*<proof>*

**lemma** *lsorted-lmap*:

**assumes** *lsorted xs monotone* ( $\leq$ ) ( $\leq$ ) *f*

**shows** *lsorted (lmap f xs)*

*<proof>*

**context** *linorder* **begin**

**lemma** *lsorted-ldistinct-lset-unique*:

$\llbracket \text{lsorted } xs; \text{ldistinct } xs; \text{lsorted } ys; \text{ldistinct } ys; \text{lset } xs = \text{lset } ys \rrbracket$   
 $\implies xs = ys$

*<proof>*

**end**

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

## 2.24 Lexicographic order on lazy lists: *llexord*

**lemma** *llexord-coinduct* [consumes 1, case-names *llexord*, coinduct pred: *llexord*]:

**assumes**  $X: X\ xs\ ys$

**and step:**  $\bigwedge xs\ ys. \llbracket X\ xs\ ys; \neg\ lnull\ xs \rrbracket$

$\implies \neg\ lnull\ ys \wedge$

$(\neg\ lnull\ ys \longrightarrow r\ (lhd\ xs)\ (lhd\ ys) \vee$

$lhd\ xs = lhd\ ys \wedge (X\ (ltl\ xs)\ (ltl\ ys) \vee llexord\ r\ (ltl\ xs)\ (ltl\ ys)))$

**shows**  $llexord\ r\ xs\ ys$

*<proof>*

**lemma** *llexord-reft* [simp, intro!]:

$llexord\ r\ xs\ xs$

*<proof>*

**lemma** *llexord-LCons-LCons* [simp]:

$llexord\ r\ (LCons\ x\ xs)\ (LCons\ y\ ys) \longleftrightarrow (x = y \wedge llexord\ r\ xs\ ys \vee r\ x\ y)$

*<proof>*

**lemma** *lnull-llexord* [simp]:  $lnull\ xs \implies llexord\ r\ xs\ ys$

*<proof>*

**lemma** *llexord-LNil-right* [simp]:

$lnull\ ys \implies llexord\ r\ xs\ ys \longleftrightarrow lnull\ xs$

*<proof>*

**lemma** *llexord-LCons-left*:

$llexord\ r\ (LCons\ x\ xs)\ ys \longleftrightarrow$

$(\exists y\ ys'. ys = LCons\ y\ ys' \wedge (x = y \wedge llexord\ r\ xs\ ys' \vee r\ x\ y))$

*<proof>*

**lemma** *lprefix-imp-llexord*:

**assumes**  $xs \sqsubseteq ys$

**shows**  $llexord\ r\ xs\ ys$

*<proof>*

**lemma** *llexord-empty*:

$llexord\ (\lambda x\ y. False)\ xs\ ys = xs \sqsubseteq ys$

*<proof>*

**lemma** *llexord-append-right*:

$llexord\ r\ xs\ (lappend\ xs\ ys)$

*<proof>*

**lemma** *llexord-lappend-leftI*:

**assumes**  $llexord\ r\ ys\ zs$

**shows**  $llexord\ r\ (lappend\ xs\ ys)\ (lappend\ xs\ zs)$

*<proof>*

**lemma** *llexord-lappend-leftD*:

**assumes**  $lex: llexord\ r\ (lappend\ xs\ ys)\ (lappend\ xs\ zs)$   
**and**  $fin: lfinite\ xs$   
**and**  $irrefl: !!x. x \in lset\ xs \implies \neg r\ x\ x$   
**shows**  $llexord\ r\ ys\ zs$   
 $\langle proof \rangle$

**lemma**  $llexord\ lappend\ left$ :  
 $\llbracket lfinite\ xs; !!x. x \in lset\ xs \implies \neg r\ x\ x \rrbracket$   
 $\implies llexord\ r\ (lappend\ xs\ ys)\ (lappend\ xs\ zs) \longleftrightarrow llexord\ r\ ys\ zs$   
 $\langle proof \rangle$

**lemma**  $antisym\ llexord$ :  
**assumes**  $r: antisymp\ r$   
**and**  $irrefl: \bigwedge x. \neg r\ x\ x$   
**shows**  $antisymp\ (llexord\ r)$   
 $\langle proof \rangle$

**lemma**  $llexord\ antisym$ :  
 $\llbracket llexord\ r\ xs\ ys; llexord\ r\ ys\ xs; !!a\ b. \llbracket r\ a\ b; r\ b\ a \rrbracket \implies False \rrbracket$   
 $\implies xs = ys$   
 $\langle proof \rangle$

**lemma**  $llexord\ trans$ :  
**assumes**  $1: llexord\ r\ xs\ ys$   
**and**  $2: llexord\ r\ ys\ zs$   
**and**  $trans: !!a\ b\ c. \llbracket r\ a\ b; r\ b\ c \rrbracket \implies r\ a\ c$   
**shows**  $llexord\ r\ xs\ zs$   
 $\langle proof \rangle$

**lemma**  $trans\ llexord$ :  
 $transp\ r \implies transp\ (llexord\ r)$   
 $\langle proof \rangle$

**lemma**  $llexord\ linear$ :  
**assumes**  $linear: !!x\ y. r\ x\ y \vee x = y \vee r\ y\ x$   
**shows**  $llexord\ r\ xs\ ys \vee llexord\ r\ ys\ xs$   
 $\langle proof \rangle$

**lemma**  $llexord\ code$  [code]:  
 $llexord\ r\ LNil\ ys = True$   
 $llexord\ r\ (LCons\ x\ xs)\ LNil = False$   
 $llexord\ r\ (LCons\ x\ xs)\ (LCons\ y\ ys) = (r\ x\ y \vee x = y \wedge llexord\ r\ xs\ ys)$   
 $\langle proof \rangle$

**lemma**  $llexord\ conv$ :  
 $llexord\ r\ xs\ ys \longleftrightarrow$   
 $xs = ys \vee$   
 $(\exists zs\ xs'\ y\ ys'. lfinite\ zs \wedge xs = lappend\ zs\ xs' \wedge ys = lappend\ zs\ (LCons\ y\ ys') \wedge$

$(xs' = LNil \vee r (\text{lhs } xs') y)$

**(is ?lhs  $\longleftrightarrow$  ?rhs)**  
 $\langle \text{proof} \rangle$

**lemma llexord-conv-ltake-index:**

$llexord\ r\ xs\ ys \longleftrightarrow$   
 $(l\text{length } xs \leq l\text{length } ys \wedge ltake\ (l\text{length } xs)\ ys = xs) \vee$   
 $(\exists n. \text{enat } n < \min\ (l\text{length } xs)\ (l\text{length } ys) \wedge$   
 $ltake\ (\text{enat } n)\ xs = ltake\ (\text{enat } n)\ ys \wedge r\ (\text{lnth } xs\ n)\ (\text{lnth } ys\ n))$   
**(is ?lhs  $\longleftrightarrow$  ?rhs)**  
 $\langle \text{proof} \rangle$

**lemma llexord-llist-of:**

$llexord\ r\ (l\text{list-of } xs)\ (l\text{list-of } ys) \longleftrightarrow$   
 $xs = ys \vee (xs, ys) \in llexord\ \{(x, y). r\ x\ y\}$   
**(is ?lhs  $\longleftrightarrow$  ?rhs)**  
 $\langle \text{proof} \rangle$

## 2.25 The filter functional on lazy lists: *lfilter*

**lemma lfilter-code** [*simp, code*]:

**shows** *lfilter-LNil*:  $lfilter\ P\ LNil = LNil$   
**and** *lfilter-LCons*:  $lfilter\ P\ (LCons\ x\ xs) = (\text{if } P\ x\ \text{then } LCons\ x\ (lfilter\ P\ xs)$   
 $\text{else } lfilter\ P\ xs)$   
 $\langle \text{proof} \rangle$

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

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

**shows** *monotone-lfilter*:  $\text{monotone } (\sqsubseteq) (\sqsubseteq) (lfilter\ P)$   
 $\langle \text{proof} \rangle$

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

**shows** *mcont-lfilter*:  $\text{mcont } lSup\ (\sqsubseteq) lSup\ (\sqsubseteq) (lfilter\ P)$   
 $\langle \text{proof} \rangle$

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

$\text{mono-llist } A \implies \text{mono-llist } (\lambda f. lfilter\ P\ (A\ f))$   
 $\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

**lemma lfilter-LCons-found**:

$P\ x \implies lfilter\ P\ (LCons\ x\ xs) = LCons\ x\ (lfilter\ P\ xs)$   
 $\langle \text{proof} \rangle$

**lemma lfilter-eq-LNil**:  $lfilter\ P\ xs = LNil \longleftrightarrow (\forall x \in lset\ xs. \neg P\ x)$

$\langle \text{proof} \rangle$

**notepad begin**

*<proof>*

**end**

**lemma** *diverge-lfilter-LNil* [simp]:  $\forall x \in \text{lset } xs. \neg P x \implies \text{lfilter } P \text{ } xs = \text{LNil}$   
*<proof>*

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

**lemma** *lnull-lfilter* [simp]:  $\text{lnull } (\text{lfilter } P \text{ } xs) \iff (\forall x \in \text{lset } xs. \neg P x)$   
*<proof>*

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

**lemma** *lhd-lfilter* [simp]:  $\text{lhd } (\text{lfilter } P \text{ } xs) = \text{lhd } (\text{ldropWhile } (\text{Not} \circ P) \text{ } xs)$   
*<proof>*

**lemma** *ltl-lfilter*:  $\text{ltl } (\text{lfilter } P \text{ } xs) = \text{lfilter } P \text{ } (\text{ltl } (\text{ldropWhile } (\text{Not} \circ P) \text{ } xs))$   
*<proof>*

**lemma** *lfilter-eq-LCons*:

$\text{lfilter } P \text{ } xs = \text{LCons } x \text{ } xs' \implies$

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

*<proof>*

**lemma** *lfilter-K-True* [simp]:  $\text{lfilter } (\%-. \text{True}) \text{ } xs = xs$   
*<proof>*

**lemma** *lfilter-K-False* [simp]:  $\text{lfilter } (\lambda-. \text{False}) \text{ } xs = \text{LNil}$   
*<proof>*

**lemma** *lfilter-lappend-lfinite* [simp]:

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

*<proof>*

**lemma** *lfinite-lfilterI* [simp]:  $\text{lfinite } xs \implies \text{lfinite } (\text{lfilter } P \text{ } xs)$   
*<proof>*

**lemma** *lset-lfilter* [simp]:  $\text{lset } (\text{lfilter } P \text{ } xs) = \{x \in \text{lset } xs. P x\}$   
*<proof>*

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

*<proof>*

**end**

**lemma** *lfilter-lfilter*:  $\text{lfilter } P \text{ } (\text{lfilter } Q \text{ } xs) = \text{lfilter } (\lambda x. P x \wedge Q x) \text{ } xs$   
*<proof>*

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

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

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

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

**lemma** *lfilter-cong* [*cong*]:  
**assumes** *xsys*:  $xs = ys$   
**and** *set*:  $\bigwedge x. x \in lset\ ys \implies P\ x = Q\ x$   
**shows**  $lfilter\ P\ xs = lfilter\ Q\ ys$   
 ⟨*proof*⟩

**lemma** *llength-lfilter-ile*:  
 $llength\ (lfilter\ P\ xs) \leq llength\ xs$   
 ⟨*proof*⟩

**lemma** *lfinite-lfilter*:  
 $lfinite\ (lfilter\ P\ xs) \longleftrightarrow$   
 $lfinite\ xs \vee finite\ \{n. enat\ n < llength\ xs \wedge P\ (lnth\ xs\ n)\}$   
 ⟨*proof*⟩

**lemma** *lfilter-eq-LConsD*:  
**assumes**  $lfilter\ P\ ys = LCons\ x\ xs$   
**shows**  $\exists us\ vs. ys = lappend\ us\ (LCons\ x\ vs) \wedge lfinite\ us \wedge$   
 $(\forall u \in lset\ us. \neg P\ u) \wedge P\ x \wedge xs = lfilter\ P\ vs$   
 ⟨*proof*⟩

**lemma** *lfilter-eq-lappend-lfiniteD*:  
**assumes**  $lfilter\ P\ xs = lappend\ ys\ zs$  **and**  $lfinite\ ys$   
**shows**  $\exists us\ vs. xs = lappend\ us\ vs \wedge lfinite\ us \wedge$   
 $ys = lfilter\ P\ us \wedge zs = lfilter\ P\ vs$   
 ⟨*proof*⟩

**lemma** *ldistinct-lfilterI*:  $ldistinct\ xs \implies ldistinct\ (lfilter\ P\ xs)$   
 ⟨*proof*⟩

**notepad begin**  
 ⟨*proof*⟩  
**end**

**lemma** *ldistinct-lfilterD*:



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

**lemmas** *lfilter-fixp-parallel-induct* =  
*parallel-fixp-induct-1-1* [OF *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*:  $\bigwedge x y. P \text{ } x \text{ } y \implies Q1 \text{ } x \longleftrightarrow Q2 \text{ } y$   
**shows** *lfilter-all2* *P* (*lfilter* *Q1* *xs*) (*lfilter* *Q2* *ys*)  
 <proof>

**lemma** *distinct-filterD*:  
 $\llbracket \text{distinct } (\text{filter } P \text{ } xs); n < \text{length } xs; m < \text{length } xs; P \text{ } x; xs ! n = x; xs ! m = x \rrbracket \implies m = n$   
 <proof>

**lemma** *lprefix-lfilterI*:  
 $xs \sqsubseteq ys \implies \text{lfilter } P \text{ } xs \sqsubseteq \text{lfilter } P \text{ } ys$   
 <proof>

**context** *preorder* **begin**

**lemma** *lsorted-lfilterI*:  
 $\text{lsorted } xs \implies \text{lsorted } (\text{lfilter } P \text{ } xs)$   
 <proof>

**lemma** *lsorted-lfilter-same*:  
 $\text{lsorted } (\text{lfilter } (\lambda x. x = c) \text{ } xs)$   
 <proof>

**end**

**lemma** *lfilter-id-conv*:  $\text{lfilter } P \text{ } xs = xs \longleftrightarrow (\forall x \in \text{lset } xs. P \text{ } x)$  (is ?lhs  $\longleftrightarrow$  ?rhs)  
 <proof>

**lemma** *lfilter-repeat* [simp]:  $\text{lfilter } P \text{ } (\text{repeat } x) = (\text{if } P \text{ } x \text{ then repeat } x \text{ else LNil})$   
 <proof>

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

**lemma** *lconcat-simps* [simp, code]:  
**shows** *lconcat-LNil*:  $\text{lconcat } \text{LNil} = \text{LNil}$   
**and** *lconcat-LCons*:  $\text{lconcat } (\text{LCons } xs \text{ } xss) = \text{lappend } xs \text{ } (\text{lconcat } xss)$   
 <proof>

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

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

**shows** *monotone-lconcat*: *monotone* ( $\sqsubseteq$ ) ( $\sqsubseteq$ ) *lconcat*  
(*proof*)

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

**shows** *mcont-lconcat*: *mcont* *lSup* ( $\sqsubseteq$ ) *lSup* ( $\sqsubseteq$ ) *lconcat*  
(*proof*)

**lemma** *lconcat-eq-LNil*: *lconcat* *xss* = *LNil*  $\longleftrightarrow$  *lset* *xss*  $\subseteq$  {*LNil*} (**is** ?*lhs*  $\longleftrightarrow$  ?*rhs*)

(*proof*)

**lemma** *lnull-lconcat* [*simp*]: *lnull* (*lconcat* *xss*)  $\longleftrightarrow$  *lset* *xss*  $\subseteq$  {*xs.lnull* *xs*}

(*proof*)

**lemma** *lconcat-llist-of*:

*lconcat* (*llist-of* (*map* *llist-of* *xs*)) = *llist-of* (*concat* *xs*)  
(*proof*)

**lemma** *lhd-lconcat* [*simp*]:

$\llbracket \neg \text{lnull } xss; \neg \text{lnull } (\text{lhd } xss) \rrbracket \implies \text{lhd } (\text{lconcat } xss) = \text{lhd } (\text{lhd } xss)$   
(*proof*)

**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*))  
(*proof*)

**lemma** *lmap-lconcat*:

*lmap* *f* (*lconcat* *xss*) = *lconcat* (*lmap* (*lmap* *f*) *xss*)  
(*proof*)

**lemma** *lconcat-lappend* [*simp*]:

**assumes** *lfinite* *xss*  
**shows** *lconcat* (*lappend* *xss* *yss*) = *lappend* (*lconcat* *xss*) (*lconcat* *yss*)  
(*proof*)

**lemma** *lconcat-eq-LCons-conv*:

*lconcat* *xss* = *LCons* *x* *xs*  $\longleftrightarrow$   
( $\exists$  *xs'* *xss'* *xss''*. *xss* = *lappend* (*llist-of* *xss'*) (*LCons* (*LCons* *x* *xs'*) *xss''*)  $\wedge$   
*xs* = *lappend* *xs'* (*lconcat* *xss''*)  $\wedge$  *set* *xss'*  $\subseteq$  {*xs.lnull* *xs*})  
(**is** ?*lhs*  $\longleftrightarrow$  ?*rhs*)  
(*proof*)

**lemma** *llength-lconcat-lfinite-conv-sum*:

**assumes** *lfinite* *xss*  
**shows** *llength* (*lconcat* *xss*) = ( $\sum$  *i* | *enat* *i* < *llength* *xss*. *llength* (*lnth* *xss* *i*))  
(*proof*)

**lemma** *lconcat-lfilter-neq-LNil*:

$lconcat (lfilter (\lambda xs. \neg lnull xs) xss) = lconcat xss$   
<proof>

**lemmas** *lconcat-fixp-parallel-induct =*

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

**lemma** *llist-all2-lconcatI*:

$llist-all2 (llist-all2 A) xss yss$   
 $\implies llist-all2 A (lconcat xss) (lconcat yss)$   
<proof>

**lemma** *llength-lconcat-eqI*:

**fixes**  $xss :: 'a\ list\ list$  **and**  $yss :: 'b\ list\ list$   
**assumes**  $llist-all2 (\lambda xs\ ys. llength\ xs = llength\ ys) xss\ yss$   
**shows**  $llength (lconcat xss) = llength (lconcat yss)$   
<proof>

**lemma** *lset-lconcat-lfinite*:

$\forall xss \in lset\ xss. lfinite\ xss \implies lset (lconcat xss) = (\bigcup_{xs \in lset\ xss} lset\ xs)$   
<proof>

**lemma** *lconcat-ltake*:

$lconcat (ltake (enat n) xss) = ltake (\sum i < n. llength (lnth xss i)) (lconcat xss)$   
<proof>

**lemma** *lnth-lconcat-conv*:

**assumes**  $enat\ n < llength (lconcat xss)$   
**shows**  $\exists m\ n'. lnth (lconcat xss) n = lnth (lnth xss m) n' \wedge enat\ n' < llength (lnth xss m) \wedge$   
 $enat\ m < llength\ xss \wedge enat\ n = (\sum i < m . llength (lnth xss i)) + enat\ n'$   
<proof>

**lemma** *lprefix-lconcatI*:

$xss \sqsubseteq yss \implies lconcat\ xss \sqsubseteq lconcat\ yss$   
<proof>

**lemma** *lnth-lconcat-ltake*:

**assumes**  $enat\ w < llength (lconcat (ltake (enat n) xss))$   
**shows**  $lnth (lconcat (ltake (enat n) xss)) w = lnth (lconcat xss) w$   
<proof>

**lemma** *lfinite-lconcat [simp]*:

$lfinite (lconcat xss) \longleftrightarrow lfinite (lfilter (\lambda xs. \neg lnull xs) xss) \wedge (\forall xs \in lset\ xss. lfinite\ xs)$   
(is ?lhs  $\longleftrightarrow$  ?rhs)

*<proof>*

**lemma** *list-of-lconcat*:

**assumes** *lfinite xss*

**and**  $\forall xs \in \text{lset } xss. \text{lfinite } xs$

**shows**  $\text{list-of } (\text{lconcat } xss) = \text{concat } (\text{list-of } (\text{lmap list-of } xss))$

*<proof>*

**lemma** *lfilter-lconcat-lfinite*:

$\forall xss \in \text{lset } xss. \text{lfinite } xss$

$\implies \text{lfilter } P (\text{lconcat } xss) = \text{lconcat } (\text{lmap } (\text{lfilter } P) xss)$

*<proof>*

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

*<proof>*

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

*<proof>*

**lemma** *lset-lconcat-subset*:  $\text{lset } (\text{lconcat } xss) \subseteq (\bigcup xs \in \text{lset } xss. \text{lset } xs)$

*<proof>*

**lemma** *ldistinct-lconcat*:

$\llbracket \text{ldistinct } xss; \bigwedge ys. ys \in \text{lset } xss \implies \text{ldistinct } ys; \bigwedge ys zs. \llbracket ys \in \text{lset } xss; zs \in \text{lset } xss; ys \neq zs \rrbracket \implies \text{lset } ys \cap \text{lset } zs = \{\} \rrbracket$

$\implies \text{ldistinct } (\text{lconcat } xss)$

*<proof>*

## 2.27 Sublist view of a lazy list: *lnths*

**lemma** *lnths-empty* [*simp*]:  $\text{lnths } xs \{\} = LNil$

*<proof>*

**lemma** *lnths-LNil* [*simp*]:  $\text{lnths } LNil A = LNil$

*<proof>*

**lemma** *lnths-LCons*:

$\text{lnths } (LCons x xs) A =$

$(\text{if } 0 \in A \text{ then } LCons x (\text{lnths } xs \{n. Suc n \in A\}) \text{ else } \text{lnths } xs \{n. Suc n \in A\})$

*<proof>*

**lemma** *lset-lnths*:

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

*<proof>*

**lemma** *lset-lnths-subset*:  $\text{lset } (\text{lnths } xs I) \subseteq \text{lset } xs$

*<proof>*

**lemma** *lnths-singleton* [simp]:  
 $lnths (LCons x LNil) A = (if\ 0 : A\ then\ LCons\ x\ LNil\ else\ LNil)$   
⟨proof⟩

**lemma** *lnths-upt-eq-ltake* [simp]:  
 $lnths\ xs\ \{..  
⟨proof⟩$

**lemma** *lnths-llist-of* [simp]:  
 $lnths\ (llist-of\ xs)\ A = llist-of\ (lnths\ xs\ A)$   
⟨proof⟩

**lemma** *llength-lnths-ile*:  $llength\ (lnths\ xs\ A) \leq llength\ xs$   
⟨proof⟩

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

**lemma** *lfilter-conv-lnths*:  
 $lfilter\ P\ xs = lnths\ xs\ \{n.\ enat\ n < llength\ xs \wedge P\ (lnth\ xs\ n)\}$   
⟨proof⟩

**lemma** *ltake-iterates-Suc*:  
 $ltake\ (enat\ n)\ (iterates\ Suc\ m) = llist-of\ [m..  
⟨proof⟩$

**lemma** *lnths-lappend-lfinite*:  
**assumes**  $len: llength\ xs = enat\ k$   
**shows**  $lnths\ (lappend\ xs\ ys)\ A =$   
 $lappend\ (lnths\ xs\ A)\ (lnths\ ys\ \{n.\ n + k \in A\})$   
⟨proof⟩

**lemma** *lnths-split*:  
 $lnths\ xs\ A =$   
 $lappend\ (lnths\ (ltake\ (enat\ n)\ xs)\ A)\ (lnths\ (ldropn\ n\ xs)\ \{m.\ n + m \in A\})$   
⟨proof⟩

**lemma** *lnths-cong*:  
**assumes**  $xs = ys$  **and**  $A: \bigwedge n.\ enat\ n < llength\ xs \implies n \in A \iff n \in B$   
**shows**  $lnths\ xs\ A = lnths\ ys\ B$   
⟨proof⟩

**lemma** *lnths-insert*:  
**assumes**  $n: enat\ n < llength\ xs$   
**shows**  $lnths\ xs\ (insert\ n\ A) =$   
 $lappend\ (lnths\ (ltake\ (enat\ n)\ xs)\ A)\ (LCons\ (lnth\ xs\ n)$   
 $(lnths\ (ldropn\ (Suc\ n)\ xs)\ \{m.\ Suc\ (n + m) \in A\}))$   
⟨proof⟩

**lemma** *lfinite-lnth* [simp]:  
 $lfinite (lnth\ xs\ A) \longleftrightarrow lfinite\ xs \vee finite\ A$   
 ⟨proof⟩

## 2.28 *lsum-list*

**context** *monoid-add* **begin**

**lemma** *lsum-list-0* [simp]:  $lsum-list (lmap (\lambda-. 0) xs) = 0$   
 ⟨proof⟩

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

**lemma** *lsum-list-lappend*:  $\llbracket lfinite\ xs; lfinite\ ys \rrbracket \implies lsum-list (lappend\ xs\ ys) = lsum-list\ xs + lsum-list\ ys$   
 ⟨proof⟩

**lemma** *lsum-list-LNil* [simp]:  $lsum-list\ LNil = 0$   
 ⟨proof⟩

**lemma** *lsum-list-LCons* [simp]:  $lfinite\ xs \implies lsum-list (LCons\ x\ xs) = x + lsum-list\ xs$   
 ⟨proof⟩

**lemma** *lsum-list-inf* [simp]:  $\neg lfinite\ xs \implies lsum-list\ xs = 0$   
 ⟨proof⟩

**end**

**lemma** *lsum-list-mono*:  
 fixes  $f :: 'a \Rightarrow 'b :: \{monoid-add, ordered-ab-semigroup-add\}$   
 assumes  $\bigwedge x. x \in lset\ xs \implies f\ x \leq g\ x$   
 shows  $lsum-list (lmap\ f\ xs) \leq lsum-list (lmap\ g\ xs)$   
 ⟨proof⟩

## 2.29 Alternative view on 'a llist as datatype with constructors *llist-of* and *inf-llist*

**lemma** *lnull-inf-llist* [simp]:  $\neg lnull (inf-llist\ f)$   
 ⟨proof⟩

**lemma** *inf-llist-neq-LNil*:  $inf-llist\ f \neq LNil$   
 ⟨proof⟩

**lemmas**  $LNil-neq-inf-llist = inf-llist-neq-LNil[symmetric]$

**lemma** *lhd-inf-llist* [simp]:  $lhd (inf-llist\ f) = f\ 0$

$\langle \text{proof} \rangle$

**lemma** *ltl-inf-llist* [*simp*]:  $ltl (inf-llist f) = inf-llist (\lambda n. f (Suc n))$

$\langle \text{proof} \rangle$

**lemma** *inf-llist-rec* [*code*, *nitpick-simp*]:  
 $inf-llist f = LCons (f 0) (inf-llist (\lambda n. f (Suc n)))$   
 $\langle \text{proof} \rangle$

**lemma** *lfinite-inf-llist* [*iff*]:  $\neg lfinite (inf-llist f)$   
 $\langle \text{proof} \rangle$

**lemma** *iterates-conv-inf-llist*:  
 $iterates f a = inf-llist (\lambda n. (f \hat{\sim} n) a)$   
 $\langle \text{proof} \rangle$

**lemma** *inf-llist-neq-llist-of* [*simp*]:  
 $llist-of xs \neq inf-llist f$   
 $inf-llist f \neq llist-of xs$   
 $\langle \text{proof} \rangle$

**lemma** *lnth-inf-llist* [*simp*]:  $lnth (inf-llist f) n = f n$   
 $\langle \text{proof} \rangle$

**lemma** *inf-llist-lprefix* [*simp*]:  $inf-llist f \sqsubseteq xs \longleftrightarrow xs = inf-llist f$   
 $\langle \text{proof} \rangle$

**lemma** *llength-inf-llist* [*simp*]:  $llength (inf-llist f) = \infty$   
 $\langle \text{proof} \rangle$

**lemma** *lset-inf-llist* [*simp*]:  $lset (inf-llist f) = range f$   
 $\langle \text{proof} \rangle$

**lemma** *inf-llist-inj* [*simp*]:  
 $inf-llist f = inf-llist g \longleftrightarrow f = g$   
 $\langle \text{proof} \rangle$

**lemma** *inf-llist-lnth* [*simp*]:  $\neg lfinite xs \implies inf-llist (lnth xs) = xs$   
 $\langle \text{proof} \rangle$

**lemma** *llist-exhaust*:  
**obtains**  $(llist-of) ys$  **where**  $xs = llist-of ys$   
|  $(inf-llist) f$  **where**  $xs = inf-llist f$   
 $\langle \text{proof} \rangle$

**lemma** *lappend-inf-llist* [*simp*]:  $lappend (inf-llist f) xs = inf-llist f$   
 $\langle \text{proof} \rangle$

**lemma** *lmap-inf-llist* [*simp*]:

$lmap\ f\ (inf\text{-}llist\ g) = inf\text{-}llist\ (f\ o\ g)$   
*<proof>*

**lemma** *ltake-enat-inf-llist* [*simp*]:

$ltake\ (enat\ n)\ (inf\text{-}llist\ f) = llist\text{-}of\ (map\ f\ [0..<n])$   
*<proof>*

**lemma** *ldropn-inf-llist* [*simp*]:

$ldropn\ n\ (inf\text{-}llist\ f) = inf\text{-}llist\ (\lambda m. f\ (m + n))$   
*<proof>*

**lemma** *ldrop-enat-inf-llist*:

$ldrop\ (enat\ n)\ (inf\text{-}llist\ f) = inf\text{-}llist\ (\lambda m. f\ (m + n))$   
*<proof>*

**lemma** *lzip-inf-llist-inf-llist* [*simp*]:

$lzip\ (inf\text{-}llist\ f)\ (inf\text{-}llist\ g) = inf\text{-}llist\ (\lambda n. (f\ n,\ g\ n))$   
*<proof>*

**lemma** *lzip-llist-of-inf-llist* [*simp*]:

$lzip\ (llist\text{-}of\ xs)\ (inf\text{-}llist\ f) = llist\text{-}of\ (zip\ xs\ (map\ f\ [0..<length\ xs]))$   
*<proof>*

**lemma** *lzip-inf-llist-llist-of* [*simp*]:

$lzip\ (inf\text{-}llist\ f)\ (llist\text{-}of\ xs) = llist\text{-}of\ (zip\ (map\ f\ [0..<length\ xs])\ xs)$   
*<proof>*

**lemma** *llist-all2-inf-llist* [*simp*]:

$llist\text{-}all2\ P\ (inf\text{-}llist\ f)\ (inf\text{-}llist\ g) \longleftrightarrow (\forall n. P\ (f\ n)\ (g\ n))$   
*<proof>*

**lemma** *llist-all2-llist-of-inf-llist* [*simp*]:

$\neg\ llist\text{-}all2\ P\ (llist\text{-}of\ xs)\ (inf\text{-}llist\ f)$   
*<proof>*

**lemma** *llist-all2-inf-llist-llist-of* [*simp*]:

$\neg\ llist\text{-}all2\ P\ (inf\text{-}llist\ f)\ (llist\text{-}of\ xs)$   
*<proof>*

**lemma** (*in monoid-add*) *lsum-list-inflist*:  $lsum\text{-}list\ (inf\text{-}llist\ f) = 0$

*<proof>*

## 2.30 Setup for lifting and transfer

### 2.30.1 Relator and predicator properties

**abbreviation** *llist-all* == *pred-llist*



## 2.30.2 Transfer rules for the Transfer package

**context** includes *lifting-syntax*  
**begin**

**lemma** *set1-pre-llist-transfer* [*transfer-rule*]:  
(*rel-pre-llist A B*  $\implies$  *rel-set A*) *set1-pre-llist set1-pre-llist*  
<*proof*>

**lemma** *set2-pre-llist-transfer* [*transfer-rule*]:  
(*rel-pre-llist A B*  $\implies$  *rel-set B*) *set2-pre-llist set2-pre-llist*  
<*proof*>

**lemma** *LNil-transfer* [*transfer-rule*]: *llist-all2 P LNil LNil*  
<*proof*>

**lemma** *LCons-transfer* [*transfer-rule*]:  
(*A*  $\implies$  *llist-all2 A*  $\implies$  *llist-all2 A*) *LCons LCons*  
<*proof*>

**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*  
<*proof*>

**lemma** *unfold-llist-transfer* [*transfer-rule*]:  
((*A*  $\implies$  (=))  $\implies$  (*A*  $\implies$  *B*)  $\implies$  (*A*  $\implies$  *A*)  $\implies$  *A*  $\implies$  *llist-all2 B*) *unfold-llist unfold-llist*  
<*proof*>

**lemma** *llist-corec-transfer* [*transfer-rule*]:  
((*A*  $\implies$  (=))  $\implies$  (*A*  $\implies$  *B*)  $\implies$  (*A*  $\implies$  (=))  $\implies$  (*A*  $\implies$  *llist-all2 B*)  $\implies$  (*A*  $\implies$  *A*)  $\implies$  *A*  $\implies$  *llist-all2 B*) *corec-llist corec-llist*  
<*proof*>

**lemma** *ltl-transfer* [*transfer-rule*]:  
(*llist-all2 A*  $\implies$  *llist-all2 A*) *ltl ltl*  
<*proof*>

**lemma** *lset-transfer* [*transfer-rule*]:  
(*llist-all2 A*  $\implies$  *rel-set A*) *lset lset*  
<*proof*>

**lemma** *lmap-transfer* [*transfer-rule*]:  
((*A*  $\implies$  *B*)  $\implies$  *llist-all2 A*  $\implies$  *llist-all2 B*) *lmap lmap*  
<*proof*>

**lemma** *lappend-transfer* [*transfer-rule*]:  
(*llist-all2 A*  $\implies$  *llist-all2 A*  $\implies$  *llist-all2 A*) *lappend lappend*

$\langle proof \rangle$

**lemma** *iterates-transfer* [transfer-rule]:

$((A \implies A) \implies A \implies \text{llist-all2 } A) \text{ iterates iterates}$

$\langle proof \rangle$

**lemma** *lfinite-transfer* [transfer-rule]:

$(\text{llist-all2 } A \implies (=)) \text{ lfinite lfinite}$

$\langle proof \rangle$

**lemma** *lalist-of-transfer* [transfer-rule]:

$(\text{list-all2 } A \implies \text{llist-all2 } A) \text{ lalist-of lalist-of}$

$\langle proof \rangle$

**lemma** *llength-transfer* [transfer-rule]:

$(\text{llist-all2 } A \implies (=)) \text{ llength llength}$

$\langle proof \rangle$

**lemma** *ltake-transfer* [transfer-rule]:

$(=) \implies \text{llist-all2 } A \implies \text{llist-all2 } A) \text{ ltake ltake}$

$\langle proof \rangle$

**lemma** *ldropn-transfer* [transfer-rule]:

$(=) \implies \text{llist-all2 } A \implies \text{llist-all2 } A) \text{ ldropn ldropn}$

$\langle proof \rangle$

**lemma** *ldrop-transfer* [transfer-rule]:

$(=) \implies \text{llist-all2 } A \implies \text{llist-all2 } A) \text{ ldrop ldrop}$

$\langle proof \rangle$

**lemma** *ltakeWhile-transfer* [transfer-rule]:

$((A \implies (=)) \implies \text{llist-all2 } A \implies \text{llist-all2 } A) \text{ ltakeWhile ltakeWhile}$

$\langle proof \rangle$

**lemma** *ldropWhile-transfer* [transfer-rule]:

$((A \implies (=)) \implies \text{llist-all2 } A \implies \text{llist-all2 } A) \text{ ldropWhile ldropWhile}$

$\langle proof \rangle$

**lemma** *lzip-ltransfer* [transfer-rule]:

$(\text{llist-all2 } A \implies \text{llist-all2 } B \implies \text{llist-all2 } (\text{rel-prod } A \ B)) \text{ lzip lzip}$

$\langle proof \rangle$

**lemma** *inf-llist-transfer* [transfer-rule]:

$((=) \implies A \implies \text{llist-all2 } A) \text{ inf-llist inf-llist}$

$\langle proof \rangle$

**lemma** *lfilter-transfer* [transfer-rule]:

$((A \implies (=)) \implies \text{llist-all2 } A \implies \text{llist-all2 } A) \text{ lfilter lfilter}$

$\langle proof \rangle$

**lemma** *lconcat-transfer* [*transfer-rule*]:  
 (*llist-all2* (*llist-all2* *A*)  $\implies$  *llist-all2* *A*) *lconcat* *lconcat*  
 <*proof*>

**lemma** *lnths-transfer* [*transfer-rule*]:  
 (*llist-all2* *A*  $\implies$  (=)  $\implies$  *llist-all2* *A*) *lnths* *lnths*  
 <*proof*>

**lemma** *llist-all-transfer* [*transfer-rule*]:  
 ((*A*  $\implies$  (=))  $\implies$  *llist-all2* *A*  $\implies$  (=)) *llist-all* *llist-all*  
 <*proof*>

**lemma** *llist-all2-rsp*:  
 assumes *r*:  $\forall x y. R\ x\ y \longrightarrow (\forall a b. R\ a\ b \longrightarrow S\ x\ a = T\ y\ b)$   
 and *l1*: *llist-all2* *R* *x* *y*  
 and *l2*: *llist-all2* *R* *a* *b*  
 shows *llist-all2* *S* *x* *a* = *llist-all2* *T* *y* *b*  
 <*proof*>

**lemma** *llist-all2-transfer* [*transfer-rule*]:  
 ((*R*  $\implies$  *R*  $\implies$  (=))  $\implies$  *llist-all2* *R*  $\implies$  *llist-all2* *R*  $\implies$  (=))  
*llist-all2* *llist-all2*  
 <*proof*>

**end**

**no-notation** *lprefix* (**infix**  $\sqsubseteq$  65)

**end**

### 3 Instantiation of the order type classes for lazy lists

**theory** *Coinductive-List-Prefix* **imports**  
*Coinductive-List*  
*HOL-Library.Prefix-Order*  
**begin**

#### 3.1 Instantiation of the order type class

**instantiation** *llist* :: (*type*) *order* **begin**

**definition** [*code-unfold*]:  $xs \leq ys = \textit{lprefix}\ xs\ ys$

**definition** [*code-unfold*]:  $xs < ys = \textit{lstrict-prefix}\ xs\ ys$

**instance**

*<proof>*

**end**

**lemma** *le-llist-conv-lprefix* [iff]:  $(\leq) = \text{lprefix}$   
*<proof>*

**lemma** *less-llist-conv-lstrict-prefix* [iff]:  $(<) = \text{lstrict-prefix}$   
*<proof>*

**instantiation** *llist* :: (type) *order-bot* **begin**

**definition** *bot* = *LNil*

**instance**  
*<proof>*

**end**

**lemma** *llist-of-lprefix-llist-of* [simp]:  
 $\text{lprefix} (\text{llist-of } xs) (\text{llist-of } ys) \longleftrightarrow xs \leq ys$   
*<proof>*

### 3.2 Prefix ordering as a lower semilattice

**instantiation** *llist* :: (type) *semilattice-inf* **begin**

**definition** [code del]:  
 $\text{inf } xs \ ys =$   
 $\text{unfold-llist } (\lambda(xs, ys). xs \neq \text{LNil} \longrightarrow ys \neq \text{LNil} \longrightarrow \text{lhd } xs \neq \text{lhd } ys)$   
 $(\text{lhd} \circ \text{snd}) (\text{map-prod } \text{ttl } \text{ttl}) (xs, ys)$

**lemma** *llist-inf-simps* [simp, code, nitpick-simp]:  
 $\text{inf } \text{LNil } xs = \text{LNil}$   
 $\text{inf } xs \ \text{LNil} = \text{LNil}$   
 $\text{inf } (\text{LCons } x \ xs) (\text{LCons } y \ ys) = (\text{if } x = y \ \text{then } \text{LCons } x \ (\text{inf } xs \ ys) \ \text{else } \text{LNil})$   
*<proof>*

**lemma** *llist-inf-eq-LNil* [simp]:  
 $\text{lnull } (\text{inf } xs \ ys) \longleftrightarrow (xs \neq \text{LNil} \longrightarrow ys \neq \text{LNil} \longrightarrow \text{lhd } xs \neq \text{lhd } ys)$   
*<proof>*

**lemma** [simp]: **assumes**  $xs \neq \text{LNil}$   $ys \neq \text{LNil}$   $\text{lhd } xs = \text{lhd } ys$   
**shows** *lhd-llist-inf*:  $\text{lhd } (\text{inf } xs \ ys) = \text{lhd } ys$   
**and** *ttl-llist-inf*:  $\text{ttl } (\text{inf } xs \ ys) = \text{inf } (\text{ttl } xs) (\text{ttl } ys)$   
*<proof>*

**instance**  
*<proof>*

**end**

**lemma** *llength-inf* [*simp*]:  $llength (inf\ xs\ ys) = llcp\ xs\ ys$   
*<proof>*

**instantiation** *llist* :: (*type*) *ccpo*  
**begin**

**definition** *Sup A* = *lSup A*

**instance**  
*<proof>*

**end**

**end**

## 4 Infinite lists as a codatatype

**theory** *Coinductive-Stream*

**imports**

*HOL-Library.Stream*

*HOL-Library.Linear-Temporal-Logic-on-Streams*

*Coinductive-List*

**begin**

**lemma** *eq-onpI*:  $P\ x \implies eq\text{-onp}\ P\ x\ x$   
*<proof>*

**primcorec** *unfold-stream* :: ( $'a \Rightarrow 'b$ )  $\Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'b$  **stream** **where**  
*unfold-stream g1 g2 a = g1 a ## unfold-stream g1 g2 (g2 a)*

The following setup should be done by the BNF package.

congruence rule

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

lemmas about generated constants

**lemma** *eq-SConsD*:  $xs = SCons\ y\ ys \implies shd\ xs = y \wedge stl\ xs = ys$   
*<proof>*

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

**lemma** *smap-eq-SCons-conv*:

$smap\ f\ xs = y\ ##\ ys \longleftrightarrow$

$(\exists\ x\ xs'. xs = x\ ##\ xs' \wedge y = f\ x \wedge ys = smap\ f\ xs')$

*<proof>*

**lemma** *smap-unfold-stream*:

$smap\ f\ (unfold-stream\ SHD\ STL\ b) = unfold-stream\ (f \circ SHD)\ STL\ b$   
*<proof>*

**lemma** *smap-corec-stream*:

$smap\ f\ (corec-stream\ SHD\ endORmore\ STL-end\ STL-more\ b) =$   
 $corec-stream\ (f \circ SHD)\ endORmore\ (smap\ f \circ STL-end)\ STL-more\ b$   
*<proof>*

**lemma** *unfold-stream-ltl-unroll*:

$unfold-stream\ SHD\ STL\ (STL\ b) = unfold-stream\ (SHD \circ STL)\ STL\ b$   
*<proof>*

**lemma** *unfold-stream-eq-SCons* [simp]:

$unfold-stream\ SHD\ STL\ b = x \#\# xs \longleftrightarrow$   
 $x = SHD\ b \wedge xs = unfold-stream\ SHD\ STL\ (STL\ b)$   
*<proof>*

**lemma** *unfold-stream-id* [simp]:  $unfold-stream\ shd\ stl\ xs = xs$

*<proof>*

**lemma** *sset-neq-empty* [simp]:  $sset\ xs \neq \{\}$

*<proof>*

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

**lemma** *sset-stl*:  $sset\ (stl\ xs) \subseteq sset\ xs$

*<proof>*

induction rules

**lemmas** *stream-set-induct = sset-induct*

## 4.1 Lemmas about operations from *HOL-Library.Stream*

**lemma** *szip-iterates*:

$szip\ (siterate\ f\ a)\ (siterate\ g\ b) = siterate\ (map-prod\ f\ g)\ (a,\ b)$   
*<proof>*

**lemma** *szip-smap1*:  $szip\ (smap\ f\ xs)\ ys = smap\ (apfst\ f)\ (szip\ xs\ ys)$

*<proof>*

**lemma** *szip-smap2*:  $szip\ xs\ (smap\ g\ ys) = smap\ (apsnd\ g)\ (szip\ xs\ ys)$

*<proof>*

**lemma** *szip-smap* [simp]:  $szip\ (smap\ f\ xs)\ (smap\ g\ ys) = smap\ (map-prod\ f\ g)\ (szip\ xs\ ys)$

*<proof>*

**lemma** *smap-fst-szip* [simp]:  $smap\ fst\ (szip\ xs\ ys) = xs$

*<proof>*

**lemma** *smap-snd-szip* [simp]:  $smap\ snd\ (szip\ xs\ ys) = ys$   
*<proof>*

**lemma** *snth-shift*:  $snth\ (shift\ xs\ ys)\ n = (if\ n < length\ xs\ then\ xs\ !\ n\ else\ snth\ ys\ (n - length\ xs))$   
*<proof>*

**declare** *szip-unfold* [simp, nitpick-simp]

**lemma** *szip-shift*:  
 $length\ xs = length\ us$   
 $\implies szip\ (xs\ @- ys)\ (us\ @- zs) = zip\ xs\ us\ @- szip\ ys\ zs$   
*<proof>*

## 4.2 Link 'a stream to 'a llist

**definition** *llist-of-stream* :: 'a stream  $\Rightarrow$  'a llist  
**where** *llist-of-stream* = *unfold-llist* ( $\lambda$ -. 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]:  $\neg\ lnull\ (llist-of-stream\ xs)$   
*<proof>*

**lemma** *ltl-llist-of-stream* [simp]:  $ltl\ (llist-of-stream\ xs) = llist-of-stream\ (stl\ xs)$   
*<proof>*

**lemma** *stl-stream-of-llist* [simp]:  $stl\ (stream-of-llist\ xs) = stream-of-llist\ (ltl\ xs)$   
*<proof>*

**lemma** *shd-stream-of-llist* [simp]:  $shd\ (stream-of-llist\ xs) = lhd\ xs$   
*<proof>*

**lemma** *lhd-llist-of-stream* [simp]:  $lhd\ (llist-of-stream\ xs) = shd\ xs$   
*<proof>*

**lemma** *stream-of-llist-llist-of-stream* [simp]:  
 $stream-of-llist\ (llist-of-stream\ xs) = xs$   
*<proof>*

**lemma** *llist-of-stream-stream-of-llist* [simp]:  
 $\neg\ lfinite\ xs \implies llist-of-stream\ (stream-of-llist\ xs) = xs$   
*<proof>*

**lemma** *lfinite-llist-of-stream* [simp]:  $\neg\ lfinite\ (llist-of-stream\ xs)$   
*<proof>*

**lemma** *stream-from-llist*: type-definition *llist-of-stream stream-of-llist* {*xs*.  $\neg$  *lfinite xs*}  
 ⟨*proof*⟩

**interpretation** *stream*: type-definition *llist-of-stream stream-of-llist* {*xs*.  $\neg$  *lfinite xs*}  
 ⟨*proof*⟩

**declare** *stream.exhaust*[*cases type: stream*]

**locale** *stream-from-llist-setup*  
**begin**  
**setup-lifting** *stream-from-llist*  
**end**

**context**  
**begin**

**interpretation** *stream-from-llist-setup* ⟨*proof*⟩

**lemma** *cr-streamI*:  $\neg$  *lfinite xs*  $\implies$  *cr-stream xs* (*stream-of-llist xs*)  
 ⟨*proof*⟩

**lemma** *llist-of-stream-unfold-stream* [*simp*]:  
*llist-of-stream* (*unfold-stream SHD STL x*) = *unfold-llist* ( $\lambda$ -. *False*) *SHD STL x*  
 ⟨*proof*⟩

**lemma** *llist-of-stream-corec-stream* [*simp*]:  
*llist-of-stream* (*corec-stream SHD endORmore STL-more STL-end x*) =  
*corec-llist* ( $\lambda$ -. *False*) *SHD endORmore (llist-of-stream  $\circ$  STL-more) STL-end x*  
 ⟨*proof*⟩

**lemma** *LCons-llist-of-stream* [*simp*]: *LCons x (llist-of-stream xs)* = *llist-of-stream*  
 (*x ## xs*)  
 ⟨*proof*⟩

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

**lemma** *lset-llist-of-stream* [*simp*]: *lset (llist-of-stream xs)* = *sset xs* (**is** ?*lhs* = ?*rhs*)  
 ⟨*proof*⟩

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

**lemma** *llist-of-stream-siterates* [*simp*]: *llist-of-stream (siterate f x)* = *iterates f x*



*<proof>*

**lemma** *lappend-llist-of-stream-conv-shift* [*simp*]:

$lappend (llist-of\ xs) (lstream\ ys) = lstream\ (xs @- ys)$

*<proof>*

**lemma** *lzip-llist-of-stream* [*simp*]:

$lzip (lstream\ xs) (lstream\ ys) = lstream\ (szip\ xs\ ys)$

*<proof>*

**context includes** *lifting-syntax*

**begin**

**lemma** *lmap-infinite-transfer* [*transfer-rule*]:

$((=) ==> eq\_onp\ (\lambda xs. \neg\ lfinite\ xs) ==> eq\_onp\ (\lambda xs. \neg\ lfinite\ xs))\ lmap$

*lmap*

*<proof>*

**lemma** *lset-infinite-transfer* [*transfer-rule*]:

$(eq\_onp\ (\lambda xs. \neg\ lfinite\ xs) ==> (=))\ lset\ lset$

*<proof>*

**lemma** *unfold-stream-transfer* [*transfer-rule*]:

$((=) ==> (=) ==> (=) ==> pcr\_stream\ (=))\ (unfold\_llist\ (\lambda-. False))$

*unfold-stream*

*<proof>*

**lemma** *corec-stream-transfer* [*transfer-rule*]:

$((=) ==> (=) ==> ((=) ==> pcr\_stream\ (=)) ==> (=) ==> (=)$

$==> pcr\_stream\ (=))$

$(corec\_llist\ (\lambda-. False))\ corec\_stream$

*<proof>*

**lemma** *shd-transfer* [*transfer-rule*]:  $(pcr\_stream\ A ==> A)\ lhd\ shd$

*<proof>*

**lemma** *stl-transfer* [*transfer-rule*]:  $(pcr\_stream\ A ==> pcr\_stream\ A)\ ltl\ stl$

*<proof>*

**lemma** *llist-of-stream-transfer* [*transfer-rule*]:  $(pcr\_stream\ (=) ==> (=))\ id\ llist-of-stream$

*<proof>*

**lemma** *stream-of-llist-transfer* [*transfer-rule*]:

$(eq\_onp\ (\lambda xs. \neg\ lfinite\ xs) ==> pcr\_stream\ (=))\ (\lambda xs. xs)\ stream-of-llist$

*<proof>*

**lemma** *SCons-transfer* [*transfer-rule*]:

$(A ==> pcr\_stream\ A ==> pcr\_stream\ A)\ LCons\ (\#\#)$

*<proof>*

**lemma** *sset-transfer* [*transfer-rule*]: (*pcr-stream*  $A$   $====>$  *rel-set*  $A$ ) *lset* *sset*  
 ⟨*proof*⟩

**lemma** *smap-transfer* [*transfer-rule*]:  
 (( $A$   $====>$   $B$ )  $====>$  *pcr-stream*  $A$   $====>$  *pcr-stream*  $B$ ) *lmap* *smap*  
 ⟨*proof*⟩

**lemma** *snth-transfer* [*transfer-rule*]: (*pcr-stream*  $(=)$   $====>$   $(=)$ ) *lnth* *snth*  
 ⟨*proof*⟩

**lemma** *siterate-transfer* [*transfer-rule*]:  
 (( $=$ )  $====>$   $(=)$   $====>$  *pcr-stream*  $(=)$ ) *iterates* *siterate*  
 ⟨*proof*⟩

**context**

**fixes**  $xs$

**assumes**  $inf$ :  $\neg$  *lfinite*  $xs$

**notes** [*transfer-rule*] = *eq-onpI*[**where**  $P = \lambda xs. \neg$  *lfinite*  $xs$ , *OF inf*]

**begin**

**lemma** *smap-stream-of-llist* [*simp*]:  
**shows** *smap*  $f$  (*stream-of-llist*  $xs$ ) = *stream-of-llist* (*lmap*  $f$   $xs$ )  
 ⟨*proof*⟩

**lemma** *sset-stream-of-llist* [*simp*]:  
**assumes**  $\neg$  *lfinite*  $xs$   
**shows** *sset* (*stream-of-llist*  $xs$ ) = *lset*  $xs$   
 ⟨*proof*⟩

**end**

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

**lemma** *stream-all2-transfer* [*transfer-rule*]:  
 (( $=$ )  $====>$  *pcr-stream*  $(=)$   $====>$  *pcr-stream*  $(=)$   $====>$   $(=)$ ) *llist-all2* *stream-all2*  
 ⟨*proof*⟩

**lemma** *stream-all2-coinduct*:  
**assumes**  $X$   $xs$   $ys$   
**and**  $\bigwedge xs$   $ys. X$   $xs$   $ys \implies P$  (*shd*  $xs$ ) (*shd*  $ys$ )  $\wedge$  ( $X$  (*stl*  $xs$ ) (*stl*  $ys$ )  $\vee$  *stream-all2*  $P$  (*stl*  $xs$ ) (*stl*  $ys$ ))  
**shows** *stream-all2*  $P$   $xs$   $ys$   
 ⟨*proof*⟩

**lemma** *shift-transfer* [*transfer-rule*]:  
 (( $=$ )  $====>$  *pcr-stream*  $(=)$   $====>$  *pcr-stream*  $(=)$ ) (*lappend*  $\circ$  *llist-of*) *shift*

*<proof>*

**lemma** *szip-transfer* [*transfer-rule*]:

*(pcr-stream (=) ==> pcr-stream (=) ==> pcr-stream (=)) lzip szip*  
*<proof>*

### 4.3 Link 'a stream with $\text{nat} \Rightarrow 'a$

**lift-definition** *of-seq* ::  $(\text{nat} \Rightarrow 'a) \Rightarrow 'a$  stream is *inf-llist* *<proof>*

**lemma** *of-seq-rec* [*code*]: *of-seq f = f 0 ## of-seq (f o Suc)*  
*<proof>*

**lemma** *snth-of-seq* [*simp*]: *snth (of-seq f) = f*  
*<proof>*

**lemma** *snth-SCons*: *snth (x ## xs) n = (case n of 0  $\Rightarrow$  x | Suc n'  $\Rightarrow$  snth xs n')*  
*<proof>*

**lemma** *snth-SCons-simps* [*simp*]:  
  **shows** *snth-SCons-0*:  $(x ## xs) !! 0 = x$   
  **and** *snth-SCons-Suc*:  $(x ## xs) !! \text{Suc } n = xs !! n$   
*<proof>*

**lemma** *of-seq-snth* [*simp*]: *of-seq (snth xs) = xs*  
*<proof>*

**lemma** *shd-of-seq* [*simp*]: *shd (of-seq f) = f 0*  
*<proof>*

**lemma** *stl-of-seq* [*simp*]: *stl (of-seq f) = of-seq ( $\lambda n. f (\text{Suc } n)$ )*  
*<proof>*

**lemma** *sset-of-seq* [*simp*]: *sset (of-seq f) = range f*  
*<proof>*

**lemma** *smap-of-seq* [*simp*]: *smap f (of-seq g) = of-seq (f o g)*  
*<proof>*

**end**

### 4.4 Function iteration *siterate* and *sconst*

**lemmas** *siterate* [*nitpick-simp*] = *siterate.code*

**lemma** *smap-iterates*: *smap f (siterate f x) = siterate f (f x)*  
*<proof>*

**lemma** *siterate-smap*: *siterate f x = x ## (smap f (siterate f x))*  
*<proof>*

**lemma** *siterate-conv-of-seq*:  $siterate\ f\ a = of\text{-}seq\ (\lambda n. (f \overset{\sim}{\sim} n)\ a)$   
 ⟨proof⟩

**lemma** *sconst-conv-of-seq*:  $sconst\ a = of\text{-}seq\ (\lambda -. a)$   
 ⟨proof⟩

**lemma** *szip-sconst1* [simp]:  $szip\ (sconst\ a)\ xs = smap\ (Pair\ a)\ xs$   
 ⟨proof⟩

**lemma** *szip-sconst2* [simp]:  $szip\ xs\ (sconst\ b) = smap\ (\lambda x. (x, b))\ xs$   
 ⟨proof⟩

**end**

## 4.5 Counting elements

**partial-function** (*lfp*) *scount* :: ('s stream  $\Rightarrow$  bool)  $\Rightarrow$  's stream  $\Rightarrow$  enat **where**  
 $scount\ P\ \omega = (if\ P\ \omega\ then\ eSuc\ (scount\ P\ (stl\ \omega))\ else\ scount\ P\ (stl\ \omega))$

**lemma** *scount-simps*:  
 $P\ \omega \implies scount\ P\ \omega = eSuc\ (scount\ P\ (stl\ \omega))$   
 $\neg P\ \omega \implies scount\ P\ \omega = scount\ P\ (stl\ \omega)$   
 ⟨proof⟩

**lemma** *scount-eq-0I*:  $alw\ (not\ P)\ \omega \implies scount\ P\ \omega = 0$   
 ⟨proof⟩

**lemma** *scount-eq-0D*:  $scount\ P\ \omega = 0 \implies alw\ (not\ P)\ \omega$   
 ⟨proof⟩

**lemma** *scount-eq-0-iff*:  $scount\ P\ \omega = 0 \longleftrightarrow alw\ (not\ P)\ \omega$   
 ⟨proof⟩

**lemma**  
**assumes**  $ev\ (alw\ (not\ P))\ \omega$   
**shows** *scount-eq-card*:  $scount\ P\ \omega = enat\ (card\ \{i. P\ (sdrop\ i\ \omega)\})$   
**and** *ev-alw-not-HLD-finite*:  $finite\ \{i. P\ (sdrop\ i\ \omega)\}$   
 ⟨proof⟩

**lemma** *scount-finite*:  $ev\ (alw\ (not\ P))\ \omega \implies scount\ P\ \omega < \infty$   
 ⟨proof⟩

**lemma** *scount-infinite*:  
 $alw\ (ev\ P)\ \omega \implies scount\ P\ \omega = \infty$   
 ⟨proof⟩

**lemma** *scount-infinite-iff*:  $scount\ P\ \omega = \infty \longleftrightarrow alw\ (ev\ P)\ \omega$   
 ⟨proof⟩

**lemma** *scount-eq*:

$scount\ P\ \omega = (if\ alw\ (ev\ P)\ \omega\ then\ \infty\ else\ enat\ (card\ \{i.\ P\ (sdrop\ i\ \omega)\}))$   
 $\langle proof \rangle$

## 4.6 First index of an element

**partial-function** (*gfp*) *sfirst* :: ('s stream  $\Rightarrow$  bool)  $\Rightarrow$  's stream  $\Rightarrow$  enat **where**  
 $sfirst\ P\ \omega = (if\ P\ \omega\ then\ 0\ else\ eSuc\ (sfirst\ P\ (stl\ \omega)))$

**lemma** *sfirst-eq-0*:  $sfirst\ P\ \omega = 0 \longleftrightarrow P\ \omega$   
 $\langle proof \rangle$

**lemma** *sfirst-0[simp]*:  $P\ \omega \Longrightarrow sfirst\ P\ \omega = 0$   
 $\langle proof \rangle$

**lemma** *sfirst-eSuc[simp]*:  $\neg P\ \omega \Longrightarrow sfirst\ P\ \omega = eSuc\ (sfirst\ P\ (stl\ \omega))$   
 $\langle proof \rangle$

**lemma** *less-sfirstD*:

**fixes**  $n :: nat$

**assumes**  $enat\ n < sfirst\ P\ \omega$  **shows**  $\neg P\ (sdrop\ n\ \omega)$

$\langle proof \rangle$

**lemma** *sfirst-finite*:  $sfirst\ P\ \omega < \infty \longleftrightarrow ev\ P\ \omega$   
 $\langle proof \rangle$

**lemma** *sfirst-Stream*:  $sfirst\ P\ (s\ \#\#\ x) = (if\ P\ (s\ \#\#\ x)\ then\ 0\ else\ eSuc\ (sfirst\ P\ x))$   
 $\langle proof \rangle$

**lemma** *less-sfirst-iff*:  $(not\ P\ until\ (alw\ P))\ \omega \Longrightarrow enat\ n < sfirst\ P\ \omega \longleftrightarrow \neg P\ (sdrop\ n\ \omega)$   
 $\langle proof \rangle$

**lemma** *sfirst-eq-Inf*:  $sfirst\ P\ \omega = Inf\ \{enat\ i \mid i.\ P\ (sdrop\ i\ \omega)\}$   
 $\langle proof \rangle$

**lemma** *sfirst-eq-enat-iff*:  $sfirst\ P\ \omega = enat\ n \longleftrightarrow ev-at\ P\ n\ \omega$   
 $\langle proof \rangle$

## 4.7 stakeWhile

**definition** *stakeWhile* :: ('a  $\Rightarrow$  bool)  $\Rightarrow$  'a stream  $\Rightarrow$  'a llist  
**where**  $stakeWhile\ P\ xs = ltakeWhile\ P\ (lstream\ xs)$

**lemma** *stakeWhile-SCons* [*simp*]:

$stakeWhile\ P\ (x\ \#\#\ xs) = (if\ P\ x\ then\ LCons\ x\ (stakeWhile\ P\ xs)\ else\ LNil)$   
 $\langle proof \rangle$

**lemma** *lnull-stakeWhile* [*simp*]:  $lnull\ (stakeWhile\ P\ xs) \longleftrightarrow \neg P\ (shd\ xs)$

*<proof>*

**lemma** *lhd-stakeWhile* [simp]:  $P \text{ (shd } xs) \implies \text{lhd (stakeWhile } P \text{ } xs) = \text{shd } xs$   
*<proof>*

**lemma** *ltl-stakeWhile* [simp]:  
 $\text{ltl (stakeWhile } P \text{ } xs) = (\text{if } P \text{ (shd } xs) \text{ then stakeWhile } P \text{ (stl } xs) \text{ else LNil})$   
*<proof>*

**lemma** *stakeWhile-K-False* [simp]:  $\text{stakeWhile } (\lambda\_. \text{False}) \text{ } xs = \text{LNil}$   
*<proof>*

**lemma** *stakeWhile-K-True* [simp]:  $\text{stakeWhile } (\lambda\_. \text{True}) \text{ } xs = \text{llist-of-stream } xs$   
*<proof>*

**lemma** *stakeWhile-smap*:  $\text{stakeWhile } P \text{ (smap } f \text{ } xs) = \text{lmap } f \text{ (stakeWhile } (P \circ f) \text{ } xs)$   
*<proof>*

**lemma** *lfinite-stakeWhile* [simp]:  $\text{lfinite (stakeWhile } P \text{ } xs) \longleftrightarrow (\exists x \in \text{sset } xs. \neg P \text{ } x)$   
*<proof>*

**end**

## 5 Terminated coinductive lists and their operations

**theory** *TLList* imports

*Coinductive-List*

**begin**

Terminated coinductive lists  $('a, 'b)$  *tllist* are the codatatype defined by the constructors *TNil* of type  $'b \Rightarrow ('a, 'b)$  *tllist* and *TCons* of type  $'a \Rightarrow ('a, 'b)$  *tllist*  $\Rightarrow ('a, 'b)$  *tllist*.

### 5.1 Auxiliary lemmas

**lemma** *split-fst*:  $R \text{ (fst } p) = (\forall x \ y. p = (x, y) \longrightarrow R \text{ } x)$   
*<proof>*

**lemma** *split-fst-asm*:  $R \text{ (fst } p) \longleftrightarrow (\neg (\exists x \ y. p = (x, y) \wedge \neg R \text{ } x))$   
*<proof>*

### 5.2 Type definition

**consts** *terminal0* ::  $'a$

```

codatatype (tset: 'a, 'b) tllist =
  TNil (terminal : 'b)
  | TCons (thd : 'a) (ttl : ('a, 'b) tllist)
for
  map: tmap
  rel: tllist-all2
where
  thd (TNil _) = undefined
  | ttl (TNil b) = TNil b
  | terminal (TCons _ xs) = terminal0 xs

overloading
  terminal0 == terminal0::('a, 'b) tllist ⇒ 'b
begin

partial-function (tailrec) terminal0
where terminal0 xs = (if is-TNil xs then case-tllist id undefined xs else terminal0
  (ttl xs))

end

lemma terminal0-terminal [simp]: terminal0 = terminal
  ⟨proof⟩

lemmas terminal-TNil [code, nitpick-simp] = tllist.sel(1)

lemma terminal-TCons [simp, code, nitpick-simp]: terminal (TCons x xs) = ter-
  minal xs
  ⟨proof⟩

declare tllist.sel(2) [simp del]

primcorec unfold-tllist :: ('a ⇒ bool) ⇒ ('a ⇒ 'b) ⇒ ('a ⇒ 'c) ⇒ ('a ⇒ 'a) ⇒
  'a ⇒ ('c, 'b) tllist where
  p a ⇒⇒ unfold-tllist p g1 g21 g22 a = TNil (g1 a) |
  - ⇒⇒ unfold-tllist p g1 g21 g22 a =
    TCons (g21 a) (unfold-tllist p g1 g21 g22 (g22 a))

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

```

### 5.3 Code generator setup

Test quickcheck setup

```

lemma xs = TNil x
quickcheck[random, expect=counterexample]
quickcheck[exhaustive, expect=counterexample]
  ⟨proof⟩

```

**lemma**  $TCons\ x\ xs = TCons\ x\ xs$   
**quickcheck**[*narrowing, expect=no-counterexample*]  
 ⟨*proof*⟩

More lemmas about generated constants

**lemma** *tll-unfold-tllist*:  
 $tll\ (unfold\text{-}tl\ list\ IS\ \text{TNIL}\ TNIL\ THD\ TTL\ a) =$   
*(if IS-TNIL a then TNil (TNIL a) else unfold-tl list IS-TNIL TNIL THD TTL*  
*(TTL a))*  
 ⟨*proof*⟩

**lemma** *is-TNil-ttl [simp]*:  $is\ \text{TNil}\ xs \implies is\ \text{TNil}\ (ttl\ xs)$   
 ⟨*proof*⟩

**lemma** *terminal-ttl [simp]*:  $terminal\ (ttl\ xs) = terminal\ xs$   
 ⟨*proof*⟩

**lemma** *unfold-tl list-eq-TNil [simp]*:  
 $unfold\text{-}tl\ list\ IS\ \text{TNIL}\ TNIL\ THD\ TTL\ a = \text{TNil}\ b \iff IS\ \text{TNIL}\ a \wedge b = \text{TNIL}$   
 $a$   
 ⟨*proof*⟩

**lemma** *TNil-eq-unfold-tl list [simp]*:  
 $\text{TNil}\ b = unfold\text{-}tl\ list\ IS\ \text{TNIL}\ TNIL\ THD\ TTL\ a \iff IS\ \text{TNIL}\ a \wedge b = \text{TNIL}$   
 $a$   
 ⟨*proof*⟩

**lemma** *tmap-is-TNil*:  $is\ \text{TNil}\ xs \implies tmap\ f\ g\ xs = \text{TNil}\ (g\ (terminal\ xs))$   
 ⟨*proof*⟩

**declare** *tllist.map-sel(2)*[*simp*]

**lemma** *tll-tmap [simp]*:  $tll\ (tmap\ f\ g\ xs) = tmap\ f\ g\ (tll\ xs)$   
 ⟨*proof*⟩

**lemma** *tmap-eq-TNil-conv*:  
 $tmap\ f\ g\ xs = \text{TNil}\ y \iff (\exists\ y'.\ xs = \text{TNil}\ y' \wedge g\ y' = y)$   
 ⟨*proof*⟩

**lemma** *TNil-eq-tmap-conv*:  
 $\text{TNil}\ y = tmap\ f\ g\ xs \iff (\exists\ y'.\ xs = \text{TNil}\ y' \wedge g\ y' = y)$   
 ⟨*proof*⟩

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

**lemma** *tset-tll*:  $tset\ (tll\ xs) \subseteq tset\ xs$   
 ⟨*proof*⟩



**lemma** *in-tset-ttlD*:  $x \in \text{tset } (\text{ttl } xs) \implies x \in \text{tset } xs$   
 ⟨*proof*⟩

**theorem** *tllist-set-induct*[*consumes 1, case-names find step*]:  
**assumes**  $x \in \text{tset } xs$  **and**  $\bigwedge xs. \neg \text{is-TNil } xs \implies P (\text{thd } xs) \ xs$   
**and**  $\bigwedge xs y. \llbracket \neg \text{is-TNil } xs; y \in \text{tset } (\text{ttl } xs); P y (\text{ttl } xs) \rrbracket \implies P y \ xs$   
**shows**  $P x \ xs$   
 ⟨*proof*⟩

**theorem** *set2-tllist-induct*[*consumes 1, case-names find step*]:  
**assumes**  $x \in \text{set2-tllist } xs$  **and**  $\bigwedge xs. \text{is-TNil } xs \implies P (\text{terminal } xs) \ xs$   
**and**  $\bigwedge xs y. \llbracket \neg \text{is-TNil } xs; y \in \text{set2-tllist } (\text{ttl } xs); P y (\text{ttl } xs) \rrbracket \implies P y \ xs$   
**shows**  $P x \ xs$   
 ⟨*proof*⟩

## 5.4 Connection with 'a llist

**context** fixes  $b :: 'b$  **begin**

**primcorec** *tllist-of-llist* ::  $'a \ \text{llist} \Rightarrow ('a, 'b) \ \text{tllist}$  **where**  
 $\text{tllist-of-llist } xs = (\text{case } xs \ \text{of } \text{LNil} \Rightarrow \text{TNil } b \mid \text{LCons } x \ xs' \Rightarrow \text{TCons } x (\text{tllist-of-llist } xs'))$   
**end**

**primcorec** *llist-of-tllist* ::  $('a, 'b) \ \text{tllist} \Rightarrow 'a \ \text{llist}$   
**where**  $\text{llist-of-tllist } xs = (\text{case } xs \ \text{of } \text{TNil} \Rightarrow \text{LNil} \mid \text{TCons } x \ xs' \Rightarrow \text{LCons } x (\text{llist-of-tllist } xs'))$

**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*]:  
 $\text{lnull } xs \implies \text{terminal } (\text{tllist-of-llist } b \ xs) = b$   
 ⟨*proof*⟩

**declare** *tllist-of-llist.sel*(1)[*simp del*]

**lemma** *lhd-LNil*:  $\text{lhd } \text{LNil} = \text{undefined}$   
 ⟨*proof*⟩

**lemma** *thd-TNil*:  $\text{thd } (\text{TNil } b) = \text{undefined}$   
 ⟨*proof*⟩

**lemma** *thd-tllist-of-llist* [*simp*]:  $\text{thd } (\text{tllist-of-llist } b \ xs) = \text{lhd } xs$   
 ⟨*proof*⟩

**lemma** *ttl-tllist-of-llist* [*simp*]:  $\text{ttl } (\text{tllist-of-llist } b \ xs) = \text{tllist-of-llist } b (\text{ttl } xs)$   
 ⟨*proof*⟩

**lemma** *llist-of-tllist-eq-LNil*:

$llist\text{-of-tllist } xs = LNil \longleftrightarrow is\text{-TNil } xs$   
*<proof>*

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

**lemmas** *llist-of-tllist-TNil* = *llist-of-tllist-simps*(1)  
**and** *llist-of-tllist-TCons* = *llist-of-tllist-simps*(2)

**declare** *llist-of-tllist.sel* [*simp del*]

**lemma** *lhd-llist-of-tllist* [*simp*]:  $\neg is\text{-TNil } xs \implies lhd (llist\text{-of-tllist } xs) = thd xs$   
*<proof>*

**lemma** *ttl-llist-of-tllist* [*simp*]:  
 $ttl (llist\text{-of-tllist } xs) = llist\text{-of-tllist } (ttl xs)$   
*<proof>*

**lemma** *tlldist-of-llist-cong* [*cong*]:  
**assumes**  $xs = xs' \wedge lfinite\ xs' \implies b = b'$   
**shows**  $tlldist\text{-of-llist } b\ xs = tlldist\text{-of-llist } b'\ xs'$   
*<proof>*

**lemma** *llist-of-tlldist-inverse* [*simp*]:  
 $tlldist\text{-of-llist } (terminal\ b) (llist\text{-of-tlldist } b) = b$   
*<proof>*

**lemma** *tlldist-of-llist-eq* [*simp*]:  $tlldist\text{-of-llist } b'\ xs = TNil\ b \longleftrightarrow b = b' \wedge xs = LNil$   
*<proof>*

**lemma** *TNil-eq-tlldist-of-llist* [*simp*]:  $TNil\ b = tlldist\text{-of-llist } b'\ xs \longleftrightarrow b = b' \wedge xs = LNil$   
*<proof>*

**lemma** *tlldist-of-llist-inject* [*simp*]:  
 $tlldist\text{-of-llist } b\ xs = tlldist\text{-of-llist } c\ ys \longleftrightarrow xs = ys \wedge (lfinite\ ys \longrightarrow b = c)$   
**(is ?lhs  $\longleftrightarrow$  ?rhs)**  
*<proof>*

**lemma** *tlldist-of-llist-inverse* [*simp*]:  
 $llist\text{-of-tlldist } (tlldist\text{-of-llist } b\ xs) = xs$   
*<proof>*

**definition** *cr-tlldist* ::  $('a\ llist \times 'b) \Rightarrow ('a, 'b)\ tlldist \Rightarrow bool$   
**where**  $cr\text{-tlldist} \equiv (\lambda(xs, b)\ ys.\ tlldist\text{-of-llist } b\ xs = ys)$

**lemma** *Quotient-tlldist*:  
 $Quotient\ (\lambda(xs, a)\ (ys, b).\ xs = ys \wedge (lfinite\ ys \longrightarrow a = b))$

$(\lambda(xs, a). \text{tllist-of-llist } a \text{ } xs) (\lambda ys. (\text{llist-of-tllist } ys, \text{terminal } ys)) \text{ cr-tllist}$   
 $\langle \text{proof} \rangle$

**lemma** *reflp-tllist*:  $\text{reflp } (\lambda(xs, a) (ys, b). xs = ys \wedge (\text{lfinite } ys \longrightarrow a = b))$   
 $\langle \text{proof} \rangle$

**setup-lifting** *Quotient-tllist reflp-tllist*

**context includes** *lifting-syntax*  
**begin**

**lemma** *TNil-transfer* [*transfer-rule*]:  
 $(B \implies \text{pcr-tllist } A \ B) (\text{Pair } LNil) \ TNil$   
 $\langle \text{proof} \rangle$

**lemma** *TCons-transfer* [*transfer-rule*]:  
 $(A \implies \text{pcr-tllist } A \ B \implies \text{pcr-tllist } A \ B) (\text{apfst} \circ LCons) \ TCons$   
 $\langle \text{proof} \rangle$

**lemma** *tmap-tllist-of-llist*:  
 $\text{tmap } f \ g (\text{tllist-of-llist } b \ xs) = \text{tllist-of-llist } (g \ b) (\text{lmap } f \ xs)$   
 $\langle \text{proof} \rangle$

**lemma** *tmap-transfer* [*transfer-rule*]:  
 $((=) \implies (=) \implies \text{pcr-tllist } (=) (=) \implies \text{pcr-tllist } (=) (=)) (\text{map-prod}$   
 $\circ \text{lmap}) \ \text{tmap}$   
 $\langle \text{proof} \rangle$

**lemma** *lset-llist-of-tllist* [*simp*]:  
 $\text{lset } (\text{llist-of-tllist } xs) = \text{tset } xs \ (\text{is } ?lhs = ?rhs)$   
 $\langle \text{proof} \rangle$

**lemma** *tset-tllist-of-llist* [*simp*]:  
 $\text{tset } (\text{tllist-of-llist } b \ xs) = \text{lset } xs$   
 $\langle \text{proof} \rangle$

**lemma** *tset-transfer* [*transfer-rule*]:  
 $(\text{pcr-tllist } (=) (=) \implies (=)) (\text{lset} \circ \text{fst}) \ \text{tset}$   
 $\langle \text{proof} \rangle$

**lemma** *is-TNil-transfer* [*transfer-rule*]:  
 $(\text{pcr-tllist } (=) (=) \implies (=)) (\lambda(xs, b). \text{lnull } xs) \ \text{is-TNil}$   
 $\langle \text{proof} \rangle$

**lemma** *thd-transfer* [*transfer-rule*]:  
 $(\text{pcr-tllist } (=) (=) \implies (=)) (\text{lhs} \circ \text{fst}) \ \text{thd}$   
 $\langle \text{proof} \rangle$

**lemma** *tll-transfer* [*transfer-rule*]:

$(\text{pcr-tl}list\ A\ B\ ==\!>\ \text{pcr-tl}list\ A\ B)\ (\text{apfst}\ \text{ttl})\ \text{ttl}$   
 $\langle \text{proof} \rangle$

**lemma** *l}list-of-tl}list-transfer* [*transfer-rule*]:  
 $(\text{pcr-tl}list\ (=)\ B\ ==\!>\ (=))\ \text{fst}\ \text{l}list\text{-of-tl}list$   
 $\langle \text{proof} \rangle$

**lemma** *tl}list-of-l}list-transfer* [*transfer-rule*]:  
 $((=)\ ==\!>\ (=)\ ==\!>\ \text{pcr-tl}list\ (=)\ (=))\ (\lambda b\ xs.\ (xs,\ b))\ \text{tl}list\text{-of-l}list$   
 $\langle \text{proof} \rangle$

**lemma** *terminal-tl}list-of-l}list-l}finite* [*simp*]:  
 $\text{l}finite\ xs\ \implies\ \text{terminal}\ (\text{tl}list\text{-of-l}list\ b\ xs) = b$   
 $\langle \text{proof} \rangle$

**lemma** *set2-tl}list-tl}list-of-l}list* [*simp*]:  
 $\text{set2-tl}list\ (\text{tl}list\text{-of-l}list\ b\ xs) = (\text{if}\ \text{l}finite\ xs\ \text{then}\ \{b\}\ \text{else}\ \{\})$   
 $\langle \text{proof} \rangle$

**lemma** *set2-tl}list-transfer* [*transfer-rule*]:  
 $(\text{pcr-tl}list\ A\ B\ ==\!>\ \text{rel-set}\ B)\ (\lambda(xs,\ b).\ \text{if}\ \text{l}finite\ xs\ \text{then}\ \{b\}\ \text{else}\ \{\})\ \text{set2-tl}list$   
 $\langle \text{proof} \rangle$

**lemma** *tl}list-all2-transfer* [*transfer-rule*]:  
 $((=)\ ==\!>\ (=)\ ==\!>\ \text{pcr-tl}list\ (=)\ (=)\ ==\!>\ \text{pcr-tl}list\ (=)\ (=)\ ==\!>\ (=))$   
 $(\lambda P\ Q\ (xs,\ b)\ (ys,\ b')).\ \text{l}list\text{-all2}\ P\ xs\ ys\ \wedge\ (\text{l}finite\ xs\ \longrightarrow\ Q\ b\ b'))\ \text{tl}list\text{-all2}$   
 $\langle \text{proof} \rangle$

## 5.5 Library function definitions

We lift the constants from  $'a$  *l}list* to  $('a,\ 'b)$  *tl}list* using the lifting package. This way, many results are transferred easily.

**lift-definition** *tappend* ::  $('a,\ 'b)\ \text{tl}list \Rightarrow ('b \Rightarrow ('a,\ 'c)\ \text{tl}list) \Rightarrow ('a,\ 'c)\ \text{tl}list$   
**is**  $\lambda(xs,\ b)\ f.\ \text{apfst}\ (\text{lappend}\ xs)\ (f\ b)$   
 $\langle \text{proof} \rangle$

**lift-definition** *lappendt* ::  $'a\ \text{l}list \Rightarrow ('a,\ 'b)\ \text{tl}list \Rightarrow ('a,\ 'b)\ \text{tl}list$   
**is**  $\text{apfst} \circ \text{lappend}$   
 $\langle \text{proof} \rangle$

**lift-definition** *tfilter* ::  $'b \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow ('a,\ 'b)\ \text{tl}list \Rightarrow ('a,\ 'b)\ \text{tl}list$   
**is**  $\lambda b\ P\ (xs,\ b').\ (\text{lfilter}\ P\ xs,\ \text{if}\ \text{l}finite\ xs\ \text{then}\ b'\ \text{else}\ b)$   
 $\langle \text{proof} \rangle$

**lift-definition** *tconcat* ::  $'b \Rightarrow ('a\ \text{l}list,\ 'b)\ \text{tl}list \Rightarrow ('a,\ 'b)\ \text{tl}list$   
**is**  $\lambda b\ (xss,\ b').\ (\text{lconcat}\ xss,\ \text{if}\ \text{l}finite\ xss\ \text{then}\ b'\ \text{else}\ b)$   
 $\langle \text{proof} \rangle$

**lift-definition** *tnth* ::  $('a,\ 'b)\ \text{tl}list \Rightarrow \text{nat} \Rightarrow 'a$

is  $lnth \circ fst \langle proof \rangle$

**lift-definition**  $tlength :: ('a, 'b) tlist \Rightarrow enat$   
is  $llength \circ fst \langle proof \rangle$

**lift-definition**  $tdropn :: nat \Rightarrow ('a, 'b) tlist \Rightarrow ('a, 'b) tlist$   
is  $apfst \circ ldropn \langle proof \rangle$

**abbreviation**  $tfinite :: ('a, 'b) tlist \Rightarrow bool$   
where  $tfinite\ xs \equiv lfinite\ (l\text{-of-}tlist\ xs)$

## 5.6 $tfinite$

**lemma**  $tfinite\text{-induct}$  [*consumes 1, case-names TNil TCons*]:  
assumes  $tfinite\ xs$   
and  $\bigwedge y. P\ (TNil\ y)$   
and  $\bigwedge x\ xs. \llbracket tfinite\ xs; P\ xs \rrbracket \Longrightarrow P\ (TCons\ x\ xs)$   
shows  $P\ xs$   
 $\langle proof \rangle$

**lemma**  $is\text{-}TNil\text{-}tfinite$  [*simp*]:  $is\text{-}TNil\ xs \Longrightarrow tfinite\ xs$   
 $\langle proof \rangle$

## 5.7 The terminal element $terminal$

**lemma**  $terminal\text{-}tfinite$ :  
assumes  $\neg tfinite\ xs$   
shows  $terminal\ xs = undefined$   
 $\langle proof \rangle$

**lemma**  $terminal\text{-}tlist\text{-of-}l\text{-list}$ :  
 $terminal\ (tlist\text{-of-}l\text{-list}\ y\ xs) = (if\ lfinite\ xs\ then\ y\ else\ undefined)$   
 $\langle proof \rangle$

**lemma**  $terminal\text{-}transfer$  [*transfer-rule*]:  
 $(pcr\text{-}tlist\ A\ (=) \Longrightarrow (=))\ (\lambda(xs, b). if\ lfinite\ xs\ then\ b\ else\ undefined)\ terminal$   
 $\langle proof \rangle$

**lemma**  $terminal\text{-}tmap$  [*simp*]:  $tfinite\ xs \Longrightarrow terminal\ (tmap\ f\ g\ xs) = g\ (terminal\ xs)$   
 $\langle proof \rangle$

## 5.8 $tmap$

**lemma**  $tmap\text{-}eq\text{-}TCons\text{-}conv$ :  
 $tmap\ f\ g\ xs = TCons\ y\ ys \longleftrightarrow$   
 $(\exists z\ zs. xs = TCons\ z\ zs \wedge f\ z = y \wedge tmap\ f\ g\ zs = ys)$   
 $\langle proof \rangle$

**lemma**  $TCons\text{-}eq\text{-}tmap\text{-}conv$ :

$TCons\ y\ ys = tmap\ f\ g\ xs \longleftrightarrow$   
 $(\exists z\ zs.\ xs = TCons\ z\ zs \wedge f\ z = y \wedge tmap\ f\ g\ zs = ys)$   
 ⟨proof⟩

## 5.9 Appending two terminated lazy lists *tappend*

**lemma** *tappend-TNil* [*simp*, *code*, *nitpick-simp*]:

$tappend\ (TNil\ b)\ f = f\ b$   
 ⟨proof⟩

**lemma** *tappend-TCons* [*simp*, *code*, *nitpick-simp*]:

$tappend\ (TCons\ a\ tr)\ f = TCons\ a\ (tappend\ tr\ f)$   
 ⟨proof⟩

**lemma** *tappend-TNil2* [*simp*]:

$tappend\ xs\ TNil = xs$   
 ⟨proof⟩

**lemma** *tappend-assoc*:  $tappend\ (tappend\ xs\ f)\ g = tappend\ xs\ (\lambda b.\ tappend\ (f\ b)\ g)$

⟨proof⟩

**lemma** *terminal-tappend*:

$terminal\ (tappend\ xs\ f) = (if\ tfinite\ xs\ then\ terminal\ (f\ (terminal\ xs))\ else\ terminal\ xs)$   
 ⟨proof⟩

**lemma** *tfinite-tappend*:  $tfinite\ (tappend\ xs\ f) \longleftrightarrow tfinite\ xs \wedge tfinite\ (f\ (terminal\ xs))$

⟨proof⟩

**lift-definition** *tcast* ::  $('a, 'b)\ tllist \Rightarrow ('a, 'c)\ tllist$

**is**  $\lambda(xs, a).\ (xs, undefined)$  ⟨proof⟩

**lemma** *tappend-inf*:  $\neg tfinite\ xs \Longrightarrow tappend\ xs\ f = tcast\ xs$

⟨proof⟩

*tappend* is the monadic bind on  $('a, 'b)\ tllist$

**lemmas** *tllist-monad* = *tappend-TNil* *tappend-TNil2* *tappend-assoc*

## 5.10 Appending a terminated lazy list to a lazy list *lappendt*

**lemma** *lappendt-LNil* [*simp*, *code*, *nitpick-simp*]:  $lappendt\ LNil\ tr = tr$

⟨proof⟩

**lemma** *lappendt-LCons* [*simp*, *code*, *nitpick-simp*]:

$lappendt\ (LCons\ x\ xs)\ tr = TCons\ x\ (lappendt\ xs\ tr)$   
 ⟨proof⟩

**lemma** *terminal-lappendt-lfinite* [simp]:  
 $lfinite\ xs \implies terminal\ (lappendt\ xs\ ys) = terminal\ ys$   
 ⟨proof⟩

**lemma** *tlist-of-llist-eq-lappendt-conv*:  
 $tlist\ of\ llist\ a\ xs = lappendt\ ys\ zs \iff$   
 $(\exists\ xs'\ a'.\ xs = lappend\ ys\ xs' \wedge zs = tlist\ of\ llist\ a'\ xs' \wedge (lfinite\ ys \implies a = a'))$   
 ⟨proof⟩

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

## 5.11 Filtering terminated lazy lists *tfilter*

**lemma** *tfilter-TNil* [simp]:  
 $tfilter\ b'\ P\ (TNil\ b) = TNil\ b$   
 ⟨proof⟩

**lemma** *tfilter-TCons* [simp]:  
 $tfilter\ b\ P\ (TCons\ a\ tr) = (if\ P\ a\ then\ TCons\ a\ (tfilter\ b\ P\ tr)\ else\ tfilter\ b\ P\ tr)$   
 ⟨proof⟩

**lemma** *is-TNil-tfilter*[simp]:  
 $is\ TNil\ (tfilter\ y\ P\ xs) \iff (\forall\ x \in tset\ xs.\ \neg\ P\ x)$   
 ⟨proof⟩

**lemma** *tfilter-empty-conv*:  
 $tfilter\ y\ P\ xs = TNil\ y' \iff (\forall\ x \in tset\ xs.\ \neg\ P\ x) \wedge (if\ tfinite\ xs\ then\ terminal\ xs = y'\ else\ y = y')$   
 ⟨proof⟩

**lemma** *tfilter-eq-TConsD*:  
 $tfilter\ a\ P\ ys = TCons\ x\ xs \implies$   
 $\exists\ us\ vs.\ ys = lappendt\ us\ (TCons\ x\ vs) \wedge lfinite\ us \wedge (\forall\ u \in lset\ us.\ \neg\ P\ u) \wedge P\ x \wedge xs = tfilter\ a\ P\ vs$   
 ⟨proof⟩

Use a version of *tfilter* for code generation that does not evaluate the first argument

**definition** *tfilter'* ::  $(unit \Rightarrow 'b) \Rightarrow ('a \Rightarrow bool) \Rightarrow ('a, 'b)\ tlist \Rightarrow ('a, 'b)\ tlist$   
**where** [simp, code del]:  $tfilter'\ b = tfilter\ (b\ ())$

**lemma** *tfilter-code* [code, code-unfold]:  
 $tfilter = (\lambda b.\ tfilter'\ (\lambda \_.\ b))$   
 ⟨proof⟩

**lemma** *tfilter'-code* [code]:  
 $tfilter'\ b'\ P\ (TNil\ b) = TNil\ b$

$tfilter' b' P (TCons a tr) = (if P a then TCons a (tfilter' b' P tr) else tfilter' b' P tr)$   
 $\langle proof \rangle$

**end**

**hide-const (open) tfilter'**

## 5.12 Concatenating a terminated lazy list of lazy lists *tconcat*

**lemma** *tconcat-TNil* [*simp*]:  $tconcat b (TNil b') = TNil b'$   
 $\langle proof \rangle$

**lemma** *tconcat-TCons* [*simp*]:  $tconcat b (TCons a tr) = lappendt a (tconcat b tr)$   
 $\langle proof \rangle$

Use a version of *tconcat* for code generation that does not evaluate the first argument

**definition** *tconcat'* ::  $(unit \Rightarrow 'b) \Rightarrow ('a\ list, 'b)\ tllist \Rightarrow ('a, 'b)\ tllist$   
**where** [*simp, code del*]:  $tconcat' b = tconcat (b ())$

**lemma** *tconcat-code* [*code, code-unfold*]:  $tconcat = (\lambda b. tconcat' (\lambda-. b))$   
 $\langle proof \rangle$

**lemma** *tconcat'-code* [*code*]:  
 $tconcat' b (TNil b') = TNil b'$   
 $tconcat' b (TCons a tr) = lappendt a (tconcat' b tr)$   
 $\langle proof \rangle$

**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*:  $tllist-all2 P Q (TNil b) ts \longleftrightarrow (\exists b'. ts = TNil b' \wedge Q b b')$   
 $\langle proof \rangle$

**lemma** *tllist-all2-TNil2*:  $tllist-all2 P Q ts (TNil b') \longleftrightarrow (\exists b. ts = TNil b \wedge Q b b')$   
 $\langle proof \rangle$

**lemma** *tllist-all2-TCons1*:  
 $tllist-all2 P Q (TCons x ts) ts' \longleftrightarrow (\exists x' ts''. ts' = TCons x' ts'' \wedge P x x' \wedge tllist-all2 P Q ts ts'')$   
 $\langle proof \rangle$



**lemma** *tllist-all2-TCons2*:

$tllist-all2\ P\ Q\ ts'\ (TCons\ x\ ts) \longleftrightarrow (\exists x'\ ts''.\ ts' = TCons\ x'\ ts'' \wedge P\ x'\ x \wedge tllist-all2\ P\ Q\ ts''\ ts)$   
 ⟨proof⟩

**lemma** *tllist-all2-coinduct* [consumes 1, case-names *tllist-all2*, case-conclusion *tllist-all2 is-TNil TNil TCons*, coinduct pred: *tllist-all2*]:

**assumes**  $X\ xs\ ys$   
**and**  $\bigwedge xs\ ys.\ X\ xs\ ys \implies$   
 (*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*  $P\ R\ xs\ ys$   
 ⟨proof⟩

**lemma** *tllist-all2-cases*[consumes 1, case-names *TNil TCons*, cases pred]:

**assumes** *tllist-all2*  $P\ Q\ xs\ ys$   
**obtains** (*TNil*)  $b\ b'$  **where**  $xs = TNil\ b\ ys = TNil\ b'\ Q\ b\ b'$   
 | (*TCons*)  $x\ xs'\ y\ ys'$   
**where**  $xs = TCons\ x\ xs'\$  **and**  $ys = TCons\ y\ ys'$   
**and**  $P\ x\ y$  **and** *tllist-all2*  $P\ Q\ xs'\ ys'$   
 ⟨proof⟩

**lemma** *tllist-all2-tmap1*:

$tllist-all2\ P\ Q\ (tmap\ f\ g\ xs)\ ys \longleftrightarrow tllist-all2\ (\lambda x.\ P\ (f\ x))\ (\lambda x.\ Q\ (g\ x))\ xs\ ys$   
 ⟨proof⟩

**lemma** *tllist-all2-tmap2*:

$tllist-all2\ P\ Q\ xs\ (tmap\ f\ g\ ys) \longleftrightarrow tllist-all2\ (\lambda x\ y.\ P\ x\ (f\ y))\ (\lambda x\ y.\ Q\ x\ (g\ y))\ xs\ ys$   
 ⟨proof⟩

**lemma** *tllist-all2-mono*:

$\llbracket tllist-all2\ P\ Q\ xs\ ys; \bigwedge x\ y.\ P\ x\ y \implies P'\ x\ y; \bigwedge x\ y.\ Q\ x\ y \implies Q'\ x\ y \rrbracket$   
 $\implies tllist-all2\ P'\ Q'\ xs\ ys$   
 ⟨proof⟩

**lemma** *tllist-all2-tlengthD*: *tllist-all2*  $P\ Q\ xs\ ys \implies tlength\ xs = tlength\ ys$

⟨proof⟩

**lemma** *tllist-all2-tfiniteD*: *tllist-all2*  $P\ Q\ xs\ ys \implies tfinite\ xs = tfinite\ ys$

⟨proof⟩

**lemma** *tllist-all2-tfinite1-terminalD*:

$\llbracket tllist-all2\ P\ Q\ xs\ ys; tfinite\ xs \rrbracket \implies Q\ (terminal\ xs)\ (terminal\ ys)$   
 ⟨proof⟩

**lemma** *tllist-all2-tfinite2-terminalD*:

$\llbracket \text{tllist-all2 } P \ Q \ xs \ ys; \text{tfinite } ys \rrbracket \Longrightarrow Q \ (\text{terminal } xs) \ (\text{terminal } ys)$   
 <proof>

**lemma** *tllist-all2D-llist-all2-llist-of-tllist*:

$\text{tllist-all2 } P \ Q \ xs \ ys \Longrightarrow \text{llist-all2 } P \ (\text{llist-of-tllist } xs) \ (\text{llist-of-tllist } ys)$   
 <proof>

**lemma** *tllist-all2-is-TNilD*:

$\text{tllist-all2 } P \ Q \ xs \ ys \Longrightarrow \text{is-TNil } xs \longleftrightarrow \text{is-TNil } ys$   
 <proof>

**lemma** *tllist-all2-thdD*:

$\llbracket \text{tllist-all2 } P \ Q \ xs \ ys; \neg \text{is-TNil } xs \vee \neg \text{is-TNil } ys \rrbracket \Longrightarrow P \ (\text{thd } xs) \ (\text{thd } ys)$   
 <proof>

**lemma** *tllist-all2-ttl*:

$\llbracket \text{tllist-all2 } P \ Q \ xs \ ys; \neg \text{is-TNil } xs \vee \neg \text{is-TNil } ys \rrbracket \Longrightarrow \text{tllist-all2 } P \ Q \ (\text{ttl } xs)$   
 <proof>

**lemma** *tllist-all2-refl*:

$\text{tllist-all2 } P \ Q \ xs \ xs \longleftrightarrow (\forall x \in \text{tset } xs. P \ x \ x) \wedge (\text{tfinite } xs \longrightarrow Q \ (\text{terminal } xs))$   
 <proof>

**lemma** *tllist-all2-reflI*:

$\llbracket \bigwedge x. x \in \text{tset } xs \Longrightarrow P \ x \ x; \text{tfinite } xs \Longrightarrow Q \ (\text{terminal } xs) \ (\text{terminal } xs) \rrbracket$   
 $\Longrightarrow \text{tllist-all2 } P \ Q \ xs \ xs$   
 <proof>

**lemma** *tllist-all2-conv-all-tnth*:

$\text{tllist-all2 } P \ Q \ xs \ ys \longleftrightarrow$   
 $\text{tlength } xs = \text{tlength } ys \wedge$   
 $(\forall n. \text{enat } n < \text{tlength } xs \longrightarrow P \ (\text{tnth } xs \ n) \ (\text{tnth } ys \ n)) \wedge$   
 $(\text{tfinite } xs \longrightarrow Q \ (\text{terminal } xs) \ (\text{terminal } ys))$   
 <proof>

**lemma** *tllist-all2-tnthD*:

$\llbracket \text{tllist-all2 } P \ Q \ xs \ ys; \text{enat } n < \text{tlength } xs \rrbracket$   
 $\Longrightarrow P \ (\text{tnth } xs \ n) \ (\text{tnth } ys \ n)$   
 <proof>

**lemma** *tllist-all2-tnthD2*:

$\llbracket \text{tllist-all2 } P \ Q \ xs \ ys; \text{enat } n < \text{tlength } ys \rrbracket$   
 $\Longrightarrow P \ (\text{tnth } xs \ n) \ (\text{tnth } ys \ n)$   
 <proof>

**lemmas** *tllist-all2-eq = tllist.rel-eq*

**lemma** *tmap-eq-tmap-conv-tllist-all2*:

$tmap\ f\ g\ xs = tmap\ f'\ g'\ ys \longleftrightarrow$   
 $tllist\ all2\ (\lambda x\ y.\ f\ x = f'\ y)\ (\lambda x\ y.\ g\ x = g'\ y)\ xs\ ys$   
 ⟨proof⟩

**lemma** *tllist-all2-trans*:

$\llbracket tllist\ all2\ P\ Q\ xs\ ys;\ tllist\ all2\ P\ Q\ ys\ zs;\ transp\ P;\ transp\ Q \rrbracket$   
 $\implies tllist\ all2\ P\ Q\ xs\ zs$   
 ⟨proof⟩

**lemma** *tllist-all2-tappendI*:

$\llbracket tllist\ all2\ P\ Q\ xs\ ys;$   
 $\llbracket tfinite\ xs;\ tfinite\ ys;\ Q\ (terminal\ xs)\ (terminal\ ys) \rrbracket$   
 $\implies tllist\ all2\ P\ R\ (xs'\ (terminal\ xs))\ (ys'\ (terminal\ ys)) \rrbracket$   
 $\implies tllist\ all2\ P\ R\ (tappend\ xs\ xs')\ (tappend\ ys\ ys')$   
 ⟨proof⟩

**lemma** *llist-all2-tllist-of-llistI*:

$tllist\ all2\ A\ B\ xs\ ys \implies llist\ all2\ A\ (llist\ of\ tllist\ xs)\ (llist\ of\ tllist\ ys)$   
 ⟨proof⟩

**lemma** *tllist-all2-tllist-of-llist [simp]*:

$tllist\ all2\ A\ B\ (tllist\ of\ llist\ b\ xs)\ (tllist\ of\ llist\ c\ ys) \longleftrightarrow$   
 $llist\ all2\ A\ xs\ ys \wedge (lfinite\ xs \longrightarrow B\ b\ c)$   
 ⟨proof⟩

## 5.14 From a terminated lazy list to a lazy list *llist-of-tllist*

**lemma** *llist-of-tllist-tmap [simp]*:

$llist\ of\ tllist\ (tmap\ f\ g\ xs) = lmap\ f\ (llist\ of\ tllist\ xs)$   
 ⟨proof⟩

**lemma** *llist-of-tllist-tappend*:

$llist\ of\ tllist\ (tappend\ xs\ f) = lappend\ (llist\ of\ tllist\ xs)\ (llist\ of\ tllist\ (f\ (terminal\ xs)))$   
 ⟨proof⟩

**lemma** *llist-of-tllist-lappendt [simp]*:

$llist\ of\ tllist\ (lappendt\ xs\ tr) = lappend\ xs\ (llist\ of\ tllist\ tr)$   
 ⟨proof⟩

**lemma** *llist-of-tllist-tfilter [simp]*:

$llist\ of\ tllist\ (tfilter\ b\ P\ tr) = lfilter\ P\ (llist\ of\ tllist\ tr)$   
 ⟨proof⟩

**lemma** *llist-of-tllist-tconcat*:

$llist\ of\ tllist\ (tconcat\ b\ trs) = lconcat\ (llist\ of\ tllist\ trs)$   
 ⟨proof⟩

**lemma** *llist-of-tllist-eq-lappend-conv*:  
 $llist\text{-of-tllist } xs = lappend\ us\ vs \longleftrightarrow$   
 $(\exists\ ys.\ xs = lappendt\ us\ ys \wedge vs = llist\text{-of-tllist } ys \wedge terminal\ xs = terminal\ ys)$   
 $\langle proof \rangle$

### 5.15 The $n$ th element of a terminated lazy list $tnth$

**lemma** *tnth-TNil* [*nitpick-simp*]:  
 $tnth\ (TNil\ b)\ n = undefined\ n$   
 $\langle proof \rangle$

**lemma** *tnth-TCons*:  
 $tnth\ (TCons\ x\ xs)\ n = (case\ n\ of\ 0 \Rightarrow x \mid Suc\ n' \Rightarrow tnth\ xs\ n')$   
 $\langle proof \rangle$

**lemma** *tnth-code* [*simp*, *nitpick-simp*, *code*]:  
**shows** *tnth-0*:  $tnth\ (TCons\ x\ xs)\ 0 = x$   
**and** *tnth-Suc-TCons*:  $tnth\ (TCons\ x\ xs)\ (Suc\ n) = tnth\ xs\ n$   
 $\langle proof \rangle$

**lemma** *lnth-llist-of-tllist* [*simp*]:  
 $lnth\ (llist\text{-of-tllist } xs) = tnth\ xs$   
 $\langle proof \rangle$

**lemma** *tnth-tmap* [*simp*]:  $enat\ n < tlength\ xs \implies tnth\ (tmap\ f\ g\ xs)\ n = f\ (tnth\ xs\ n)$   
 $\langle proof \rangle$

### 5.16 The length of a terminated lazy list $tlength$

**lemma** [*simp*, *nitpick-simp*]:  
**shows** *tlength-TNil*:  $tlength\ (TNil\ b) = 0$   
**and** *tlength-TCons*:  $tlength\ (TCons\ x\ xs) = eSuc\ (tlength\ xs)$   
 $\langle proof \rangle$

**lemma** *tlength-llist-of-tllist* [*simp*]:  $tlength\ (llist\text{-of-tllist } xs) = tlength\ xs$   
 $\langle proof \rangle$

**lemma** *tlength-tmap* [*simp*]:  $tlength\ (tmap\ f\ g\ xs) = tlength\ xs$   
 $\langle proof \rangle$

**definition** *gen-tlength* ::  $nat \Rightarrow ('a, 'b)\ tlist \Rightarrow enat$   
**where** *gen-tlength*  $n\ xs = enat\ n + tlength\ xs$

**lemma** *gen-tlength-code* [*code*]:  
 $gen\text{-tlength } n\ (TNil\ b) = enat\ n$   
 $gen\text{-tlength } n\ (TCons\ x\ xs) = gen\text{-tlength } (n + 1)\ xs$   
 $\langle proof \rangle$

**lemma** *tlength-code* [*code*]:  $tlength = gen\text{-tlength } 0$

*<proof>*

### 5.17 *tdropn*

**lemma** *tdropn-0* [*simp, code, nitpick-simp*]:  $tdropn\ 0\ xs = xs$   
*<proof>*

**lemma** *tdropn-TNil* [*simp, code*]:  $tdropn\ n\ (TNil\ b) = (TNil\ b)$   
*<proof>*

**lemma** *tdropn-Suc-TCons* [*simp, code*]:  $tdropn\ (Suc\ n)\ (TCons\ x\ xs) = tdropn\ n\ xs$   
*<proof>*

**lemma** *tdropn-Suc* [*nitpick-simp*]:  $tdropn\ (Suc\ n)\ xs = (case\ xs\ of\ TNil\ b \Rightarrow\ TNil\ b \mid\ TCons\ x\ xs' \Rightarrow\ tdropn\ n\ xs')$   
*<proof>*

**lemma** *lappendt-ltake-tdropn*:  
 $lappendt\ (ltake\ (enat\ n)\ (llist-of-tl\ list\ xs))\ (tdropn\ n\ xs) = xs$   
*<proof>*

**lemma** *l\list-of-tl\list-tdropn* [*simp*]:  
 $l\list-of-tl\list\ (tdropn\ n\ xs) = ldropn\ n\ (l\list-of-tl\list\ xs)$   
*<proof>*

**lemma** *tdropn-Suc-conv-tdropn*:  
 $enat\ n < tlength\ xs \Longrightarrow TCons\ (tnth\ xs\ n)\ (tdropn\ (Suc\ n)\ xs) = tdropn\ n\ xs$   
*<proof>*

**lemma** *tlength-tdropn* [*simp*]:  $tlength\ (tdropn\ n\ xs) = tlength\ xs - enat\ n$   
*<proof>*

**lemma** *tnth-tdropn* [*simp*]:  $enat\ (n + m) < tlength\ xs \Longrightarrow tnth\ (tdropn\ n\ xs)\ m = tnth\ xs\ (m + n)$   
*<proof>*

### 5.18 *tset*

**lemma** *tset-induct* [*consumes 1, case-names find step*]:  
  **assumes**  $x \in tset\ xs$   
  **and**  $\bigwedge xs. P\ (TCons\ x\ xs)$   
  **and**  $\bigwedge x' xs. [x \in tset\ xs; x \neq x'; P\ xs] \Longrightarrow P\ (TCons\ x'\ xs)$   
  **shows**  $P\ xs$   
*<proof>*

**lemma** *tset-conv-tnth*:  $tset\ xs = \{tnth\ xs\ n \mid n . enat\ n < tlength\ xs\}$   
*<proof>*

**lemma** *in-tset-conv-tnth*:  $x \in \text{tset } xs \iff (\exists n. \text{enat } n < \text{length } xs \wedge \text{tnth } xs \ n = x)$   
 ⟨*proof*⟩

## 5.19 Setup for Lifting/Transfer

### 5.19.1 Relator and predicator properties

**abbreviation** *tllist-all* == *pred-tllist*

### 5.19.2 Transfer rules for the Transfer package

**context** includes *lifting-syntax*  
**begin**

**lemma** *set1-pre-tllist-transfer* [*transfer-rule*]:  
 (*rel-pre-tllist* *A B C* ==> *rel-set* *A*) *set1-pre-tllist set1-pre-tllist*  
 ⟨*proof*⟩

**lemma** *set2-pre-tllist-transfer* [*transfer-rule*]:  
 (*rel-pre-tllist* *A B C* ==> *rel-set* *B*) *set2-pre-tllist set2-pre-tllist*  
 ⟨*proof*⟩

**lemma** *set3-pre-tllist-transfer* [*transfer-rule*]:  
 (*rel-pre-tllist* *A B C* ==> *rel-set* *C*) *set3-pre-tllist set3-pre-tllist*  
 ⟨*proof*⟩

**lemma** *TNil-transfer2* [*transfer-rule*]: (*B* ==> *tllist-all2* *A B*) *TNil TNil*  
 ⟨*proof*⟩  
**declare** *TNil-transfer* [*transfer-rule*]

**lemma** *TCons-transfer2* [*transfer-rule*]:  
 (*A* ==> *tllist-all2* *A B* ==> *tllist-all2* *A B*) *TCons TCons*  
 ⟨*proof*⟩  
**declare** *TCons-transfer* [*transfer-rule*]

**lemma** *case-tllist-transfer* [*transfer-rule*]:  
 ((*B* ==> *C*) ==> (*A* ==> *tllist-all2* *A B* ==> *C*) ==> *tllist-all2* *A B* ==> *C*)  
*case-tllist case-tllist*  
 ⟨*proof*⟩

**lemma** *unfold-tllist-transfer* [*transfer-rule*]:  
 ((*A* ==> (=)) ==> (*A* ==> *B*) ==> (*A* ==> *C*) ==> (*A* ==> *A*) ==> *tllist-all2* *C B*) *unfold-tllist unfold-tllist*  
 ⟨*proof*⟩

**lemma** *corec-tllist-transfer* [*transfer-rule*]:  
 ((*A* ==> (=)) ==> (*A* ==> *B*) ==> (*A* ==> *C*) ==> (*A* ==> (=)) ==> (*A* ==> *tllist-all2* *C B*) ==> (*A* ==> *A*) ==> *A* ==>

*tllist-all2 C B) corec-tllist corec-tllist*  
 <proof>

**lemma** *tll-transfer2* [*transfer-rule*]:  
 (*tllist-all2 A B*  $\implies$  *tllist-all2 A B*) *tll tll*  
 <proof>  
**declare** *tll-transfer* [*transfer-rule*]

**lemma** *tset-transfer2* [*transfer-rule*]:  
 (*tllist-all2 A B*  $\implies$  *rel-set A*) *tset tset*  
 <proof>

**lemma** *tmap-transfer2* [*transfer-rule*]:  
 ((*A*  $\implies$  *B*)  $\implies$  (*C*  $\implies$  *D*)  $\implies$  *tllist-all2 A C*  $\implies$  *tllist-all2 B D*) *tmap tmap*  
 <proof>  
**declare** *tmap-transfer* [*transfer-rule*]

**lemma** *is-TNil-transfer2* [*transfer-rule*]:  
 (*tllist-all2 A B*  $\implies$  (=)) *is-TNil is-TNil*  
 <proof>  
**declare** *is-TNil-transfer* [*transfer-rule*]

**lemma** *tappend-transfer* [*transfer-rule*]:  
 (*tllist-all2 A B*  $\implies$  (*B*  $\implies$  *tllist-all2 A C*)  $\implies$  *tllist-all2 A C*) *tappend tappend*  
 <proof>  
**declare** *tappend.transfer* [*transfer-rule*]

**lemma** *lappendt-transfer* [*transfer-rule*]:  
 (*llist-all2 A*  $\implies$  *tllist-all2 A B*  $\implies$  *tllist-all2 A B*) *lappendt lappendt*  
 <proof>  
**declare** *lappendt.transfer* [*transfer-rule*]

**lemma** *lalist-of-tllist-transfer2* [*transfer-rule*]:  
 (*tllist-all2 A B*  $\implies$  *lalist-all2 A*) *lalist-of-tllist lalist-of-tllist*  
 <proof>  
**declare** *lalist-of-tllist-transfer* [*transfer-rule*]

**lemma** *tllist-of-llist-transfer2* [*transfer-rule*]:  
 (*B*  $\implies$  *lalist-all2 A*  $\implies$  *tllist-all2 A B*) *tllist-of-llist tllist-of-llist*  
 <proof>  
**declare** *tllist-of-llist-transfer* [*transfer-rule*]

**lemma** *tlength-transfer* [*transfer-rule*]:  
 (*tllist-all2 A B*  $\implies$  (=)) *tlength tlength*  
 <proof>  
**declare** *tlength.transfer* [*transfer-rule*]

```

lemma tdropn-transfer [transfer-rule]:
  ((=) ==> tllist-all2 A B ==> tllist-all2 A B) tdropn tdropn
  <proof>
declare tdropn.transfer [transfer-rule]

lemma tfilter-transfer [transfer-rule]:
  (B ==> (A ==> (=)) ==> tllist-all2 A B ==> tllist-all2 A B) tfilter
  tfilter
  <proof>
declare tfilter.transfer [transfer-rule]

lemma tconcat-transfer [transfer-rule]:
  (B ==> tllist-all2 (llist-all2 A) B ==> tllist-all2 A B) tconcat tconcat
  <proof>
declare tconcat.transfer [transfer-rule]

lemma tllist-all2-rsp:
  assumes R1:  $\forall x y. R1\ x\ y \longrightarrow (\forall a\ b. R1\ a\ b \longrightarrow S\ x\ a = T\ y\ b)$ 
  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>

lemma tllist-all2-transfer2 [transfer-rule]:
  ((R1 ==> R1 ==> (=)) ==> (R2 ==> R2 ==> (=)) ==>
   tllist-all2 R1 R2 ==> tllist-all2 R1 R2 ==> (=)) tllist-all2
  <proof>
declare tllist-all2-transfer [transfer-rule]

end

Delete lifting rules for ('a, 'b) tllist because the parametricity rules take
precedence over most of the transfer rules. They can be restored by including
the bundle tllist.lifting.

lifting-update tllist.lifting
lifting-forget tllist.lifting

end

```

## 6 Setup for Isabelle's quotient package for lazy lists

```

theory Quotient-Coinductive-List imports
  HOL-Library.Quotient-List
  HOL-Library.Quotient-Set
  Coinductive-List
begin

```



## 6.1 Rules for the Quotient package

**declare** *llist.rel-eq[id-simps]*

**lemma** *transpD*:  $\llbracket \text{transp } R; R \ a \ b; R \ b \ c \rrbracket \implies R \ a \ c$   
 $\langle \text{proof} \rangle$

**lemma** *id-respect* [*quot-respect*]:  
 $(R \implies R) \text{ id id}$   
 $\langle \text{proof} \rangle$

**lemma** *id-preserve* [*quot-preserve*]:  
**assumes** *Quotient3* *R Abs Rep*  
**shows**  $(Rep \dashrightarrow Abs) \text{ id} = \text{id}$   
 $\langle \text{proof} \rangle$

**functor** *lmap*: *lmap*  
 $\langle \text{proof} \rangle$

**declare** *lmap-id0* [*id-simps*]

**lemma** *reflp-llist-all2*:  $\text{reflp } R \implies \text{reflp } (\text{llist-all2 } R)$   
 $\langle \text{proof} \rangle$

**lemma** *symp-llist-all2*:  $\text{symp } R \implies \text{symp } (\text{llist-all2 } R)$   
 $\langle \text{proof} \rangle$

**lemma** *transp-llist-all2*:  $\text{transp } R \implies \text{transp } (\text{llist-all2 } R)$   
 $\langle \text{proof} \rangle$

**lemma** *llist-equivp* [*quot-equiv*]:  
 $\text{equivp } R \implies \text{equivp } (\text{llist-all2 } R)$   
 $\langle \text{proof} \rangle$

**lemma** *unfold-llist-preserve* [*quot-preserve*]:  
**assumes** *q1*: *Quotient3* *R1 Abs1 Rep1*  
**and** *q2*: *Quotient3* *R2 Abs2 Rep2*  
**shows**  $((Abs1 \dashrightarrow id) \dashrightarrow (Abs1 \dashrightarrow Rep2) \dashrightarrow (Abs1 \dashrightarrow Rep1) \dashrightarrow Rep1 \dashrightarrow lmap \ Abs2) \text{ unfold-llist} = \text{unfold-llist}$   
 $(\text{is } ?lhs = ?rhs)$   
 $\langle \text{proof} \rangle$

**lemma** *Quotient-lmap-Abs-Rep*:  
 $\text{Quotient3 } R \ Abs \ Rep \implies \text{lmap } Abs (\text{lmap } Rep \ a) = a$   
 $\langle \text{proof} \rangle$

**lemma** *llist-all2-rel*:  
**assumes** *Quotient3* *R Abs Rep*  
**shows**  $\text{llist-all2 } R \ r \ s \iff \text{llist-all2 } R \ r \ r \ \wedge \ \text{llist-all2 } R \ s \ s \ \wedge \ (\text{lmap } Abs \ r = \text{lmap } Abs \ s)$

(is ?lhs  $\longleftrightarrow$  ?rhs)  
<proof>

**lemma** *Quotient-llist-all2-lmap-Rep*:

*Quotient3 R Abs Rep  $\implies$  llist-all2 R (lmap Rep a) (lmap Rep a)*  
<proof>

**lemma** *llist-quotient [quot-thm]*:

*Quotient3 R Abs Rep  $\implies$  Quotient3 (llist-all2 R) (lmap Abs) (lmap Rep)*  
<proof>

**declare** [[*mapQ3 llist = (llist-all2, llist-quotient)*]]

**lemma** *LCons-preserve [quot-preserve]*:

**assumes** *Quotient3 R Abs Rep*  
**shows** (*Rep  $\dashrightarrow$  (lmap Rep)  $\dashrightarrow$  (lmap Abs)*) *LCons = LCons*  
<proof>

**lemmas** *LCons-respect [quot-respect] = LCons-transfer*

**lemma** *LNil-preserve [quot-preserve]*:

*lmap Abs LNil = LNil*  
<proof>

**lemmas** *LNil-respect [quot-respect] = LNil-transfer*

**lemma** *lmap-preserve [quot-preserve]*:

**assumes** *a: Quotient3 R1 abs1 rep1*  
**and** *b: Quotient3 R2 abs2 rep2*  
**shows** ((*abs1  $\dashrightarrow$  rep2*)  $\dashrightarrow$  (*lmap rep1*)  $\dashrightarrow$  (*lmap abs2*)) *lmap = lmap*  
**and** ((*abs1  $\dashrightarrow$  id*)  $\dashrightarrow$  *lmap rep1*  $\dashrightarrow$  *id*) *lmap = lmap*  
<proof>

**lemma** *lmap-respect [quot-respect]*:

**shows** ((*R1  $\implies$  R2*)  $\implies$  (*llist-all2 R1*)  $\implies$  *llist-all2 R2*) *lmap lmap*  
**and** ((*R1  $\implies$  (=)*)  $\implies$  (*llist-all2 R1*)  $\implies$  (=)) *lmap lmap*  
<proof>

**lemmas** *llist-all2-respect [quot-respect] = llist-all2-transfer*

**lemma** *llist-all2-preserve [quot-preserve]*:

**assumes** *Quotient3 R Abs Rep*  
**shows** ((*Abs  $\dashrightarrow$  Abs*)  $\dashrightarrow$  *id*)  $\dashrightarrow$  *lmap Rep*  $\dashrightarrow$  *lmap Rep*  $\dashrightarrow$  *id*) *llist-all2 = llist-all2*  
<proof>

**lemma** *llist-all2-preserve2 [quot-preserve]*:

**assumes** *Quotient3 R Abs Rep*

**shows** ( $l\text{list-all2 } ((\text{Rep} \dashrightarrow \text{Rep} \dashrightarrow \text{id}) R) l m) = (l = m)$ )  
 <proof>

**lemma** *corec-llist-preserve* [*quot-preserve*]:  
**assumes**  $q1: \text{Quotient3 } R1 \text{ Abs1 } \text{Rep1}$   
**and**  $q2: \text{Quotient3 } R2 \text{ Abs2 } \text{Rep2}$   
**shows**  $((\text{Abs1} \dashrightarrow \text{id}) \dashrightarrow (\text{Abs1} \dashrightarrow \text{Rep2}) \dashrightarrow (\text{Abs1} \dashrightarrow \text{id}) \dashrightarrow$   
 $(\text{Abs1} \dashrightarrow \text{lmap } \text{Rep2}) \dashrightarrow (\text{Abs1} \dashrightarrow \text{Rep1}) \dashrightarrow \text{Rep1}$   
 $\dashrightarrow \text{lmap } \text{Abs2}) \text{corec-llist} = \text{corec-llist}$   
 (**is**  $?lhs = ?rhs$ )  
 <proof>

**end**

## 7 Setup for Isabelle's quotient package for terminated lazy lists

**theory** *Quotient-TLList* **imports**

*TLList*  
*HOL-Library.Quotient-Product*  
*HOL-Library.Quotient-Sum*  
*HOL-Library.Quotient-Set*

**begin**

### 7.1 Rules for the Quotient package

**lemma** *tmap-id-id* [*id-simps*]:

$tmap \text{id } \text{id} = \text{id}$   
 <proof>

**declare** *tllist-all2-eq*[*id-simps*]

**lemma** *case-sum-preserve* [*quot-preserve*]:

**assumes**  $q1: \text{Quotient3 } R1 \text{ Abs1 } \text{Rep1}$   
**and**  $q2: \text{Quotient3 } R2 \text{ Abs2 } \text{Rep2}$   
**and**  $q3: \text{Quotient3 } R3 \text{ Abs3 } \text{Rep3}$   
**shows**  $((\text{Abs1} \dashrightarrow \text{Rep2}) \dashrightarrow (\text{Abs3} \dashrightarrow \text{Rep2}) \dashrightarrow \text{map-sum } \text{Rep1}$   
 $\text{Rep3} \dashrightarrow \text{Abs2}) \text{case-sum} = \text{case-sum}$   
 <proof>

**lemma** *case-sum-preserve2* [*quot-preserve*]:

**assumes**  $q: \text{Quotient3 } R \text{ Abs } \text{Rep}$   
**shows**  $((\text{id} \dashrightarrow \text{Rep}) \dashrightarrow (\text{id} \dashrightarrow \text{Rep}) \dashrightarrow \text{id} \dashrightarrow \text{Abs}) \text{case-sum}$   
 $= \text{case-sum}$   
 <proof>

**lemma** *case-prod-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**  $((\text{Abs1} \text{ ----> } \text{Abs2} \text{ ----> } \text{Rep3}) \text{ ----> } \text{map-prod } \text{Rep1 } \text{Rep2} \text{ ----> } \text{Abs3}) \text{ case-prod} = \text{case-prod}$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{case-prod-preserve2}$  [ $\text{quot-preserve}$ ]:  
**assumes**  $q: \text{Quotient3 } R \text{ Abs Rep}$   
**shows**  $((\text{id} \text{ ----> } \text{id} \text{ ----> } \text{Rep}) \text{ ----> } \text{id} \text{ ----> } \text{Abs}) \text{ case-prod} = \text{case-prod}$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{id-preserve}$  [ $\text{quot-preserve}$ ]:  
**assumes**  $\text{Quotient3 } R \text{ Abs Rep}$   
**shows**  $(\text{Rep} \text{ ----> } \text{Abs}) \text{id} = \text{id}$   
 $\langle \text{proof} \rangle$

**functor**  $\text{tmap}: \text{tmap}$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{reflp-tllist-all2}$ :  
**assumes**  $R: \text{reflp } R$  **and**  $Q: \text{reflp } Q$   
**shows**  $\text{reflp } (\text{tllist-all2 } R \text{ } Q)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{symp-tllist-all2}$ :  $\llbracket \text{symp } R; \text{symp } S \rrbracket \implies \text{symp } (\text{tllist-all2 } R \text{ } S)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{transp-tllist-all2}$ :  $\llbracket \text{transp } R; \text{transp } S \rrbracket \implies \text{transp } (\text{tllist-all2 } R \text{ } S)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{tllist-equiv}$  [ $\text{quot-equiv}$ ]:  
 $\llbracket \text{equivp } R; \text{equivp } S \rrbracket \implies \text{equivp } (\text{tllist-all2 } R \text{ } S)$   
 $\langle \text{proof} \rangle$

**declare**  $\text{tllist-all2-eq}$  [ $\text{simp}$ ,  $\text{id-simps}$ ]

**lemma**  $\text{tmap-preserve}$  [ $\text{quot-preserve}$ ]:  
**assumes**  $q1: \text{Quotient3 } R1 \text{ Abs1 Rep1}$   
**and**  $q2: \text{Quotient3 } R2 \text{ Abs2 Rep2}$   
**and**  $q3: \text{Quotient3 } R3 \text{ Abs3 Rep3}$   
**and**  $q4: \text{Quotient3 } R4 \text{ Abs4 Rep4}$   
**shows**  $((\text{Abs1} \text{ ----> } \text{Rep2}) \text{ ----> } (\text{Abs3} \text{ ----> } \text{Rep4}) \text{ ----> } \text{tmap } \text{Rep1 } \text{Rep3} \text{ ----> } \text{tmap } \text{Abs2 } \text{Abs4}) \text{tmap} = \text{tmap}$   
**and**  $((\text{Abs1} \text{ ----> } \text{id}) \text{ ----> } (\text{Abs2} \text{ ----> } \text{id}) \text{ ----> } \text{tmap } \text{Rep1 } \text{Rep2} \text{ ----> } \text{id}) \text{tmap} = \text{tmap}$   
 $\langle \text{proof} \rangle$

**lemmas**  $\text{tmap-respect}$  [ $\text{quot-respect}$ ] =  $\text{tmap-transfer2}$

**lemma** *Quotient3-tmap-Abs-Rep*:

[[*Quotient3 R1 Abs1 Rep1*; *Quotient3 R2 Abs2 Rep2*]]  
⇒ *tmap Abs1 Abs2 (tmap Rep1 Rep2 ts) = ts*

⟨*proof*⟩

**lemma** *Quotient3-tllist-all2-tmap-tmapI*:

**assumes** *q1: Quotient3 R1 Abs1 Rep1*

**and** *q2: Quotient3 R2 Abs2 Rep2*

**shows** *tllist-all2 R1 R2 (tmap Rep1 Rep2 ts) (tmap Rep1 Rep2 ts)*

⟨*proof*⟩

**lemma** *tllist-all2-rel*:

**assumes** *q1: Quotient3 R1 Abs1 Rep1*

**and** *q2: Quotient3 R2 Abs2 Rep2*

**shows** *tllist-all2 R1 R2 r s ⟷ (tllist-all2 R1 R2 r r ∧ tllist-all2 R1 R2 s s ∧*  
*tmap Abs1 Abs2 r = tmap Abs1 Abs2 s)*

(**is** *?lhs ⟷ ?rhs*)

⟨*proof*⟩

**lemma** *tllist-quotient [quot-thm]*:

[[*Quotient3 R1 Abs1 Rep1*; *Quotient3 R2 Abs2 Rep2*]]

⇒ *Quotient3 (tllist-all2 R1 R2) (tmap Abs1 Abs2) (tmap Rep1 Rep2)*

⟨*proof*⟩

**declare** [[*mapQ3 tllist = (tllist-all2, tllist-quotient)*]]

**lemma** *TCons-preserve [quot-preserve]*:

**assumes** *q1: Quotient3 R1 Abs1 Rep1*

**and** *q2: Quotient3 R2 Abs2 Rep2*

**shows** (*Rep1 ----> (tmap Rep1 Rep2) ----> (tmap Abs1 Abs2)*) *TCons =*  
*TCons*

⟨*proof*⟩

**lemmas** *TCons-respect [quot-respect] = TCons-transfer2*

**lemma** *TNil-preserve [quot-preserve]*:

**assumes** *Quotient3 R2 Abs2 Rep2*

**shows** (*Rep2 ----> tmap Abs1 Abs2*) *TNil = TNil*

⟨*proof*⟩

**lemmas** *TNil-respect [quot-respect] = TNil-transfer2*

**lemmas** *tllist-all2-respect [quot-respect] = tllist-all2-transfer*

**lemma** *tllist-all2-prs*:

**assumes** *q1: Quotient3 R1 Abs1 Rep1*

**and** *q2: Quotient3 R2 Abs2 Rep2*

**shows** *tllist-all2 ((Abs1 ----> Abs1 ----> id) P) ((Abs2 ----> Abs2 ---->*

```

id) Q)
      (tmap Rep1 Rep2 ts) (tmap Rep1 Rep2 ts')
       $\longleftrightarrow$  tllist-all2 P Q ts ts'
      (is ?lhs  $\longleftrightarrow$  ?rhs)
      <proof>

lemma tllist-all2-preserve [quot-preserve]:
  assumes Quotient3 R1 Abs1 Rep1
  and Quotient3 R2 Abs2 Rep2
  shows ((Abs1 ----> Abs1 ----> id) ----> (Abs2 ----> Abs2 ----> id)
  ---->
  tmap Rep1 Rep2 ----> tmap Rep1 Rep2 ----> id) tllist-all2 = tllist-all2
  <proof>

lemma tllist-all2-preserve2 [quot-preserve]:
  assumes q1: Quotient3 R1 Abs1 Rep1
  and q2: Quotient3 R2 Abs2 Rep2
  shows (tllist-all2 ((Rep1 ----> Rep1 ----> id) R1) ((Rep2 ----> Rep2
  ----> id) R2)) = (=)
  <proof>

lemma corec-tllist-preserve [quot-preserve]:
  assumes q1: Quotient3 R1 Abs1 Rep1
  and q2: Quotient3 R2 Abs2 Rep2
  and q3: Quotient3 R3 Abs3 Rep3
  shows ((Abs1 ----> id) ----> (Abs1 ----> Rep2) ----> (Abs1 ---->
  Rep3) ----> (Abs1 ----> id) ----> (Abs1 ----> tmap Rep3 Rep2) ---->
  (Abs1 ----> Rep1) ----> Rep1 ----> tmap Abs3 Abs2) corec-tllist = corec-tllist
  (is ?lhs = ?rhs)
  <proof>

end

theory Coinductive imports
  Coinductive-List-Prefix
  Coinductive-Stream
  TLList
  Quotient-Coinductive-List
  Quotient-TLList
begin

end

```

## 8 Code generator setup to implement lazy lists lazily

```

theory Lazy-LList imports
  Coinductive-List

```

begin

## 8.1 Lazy lists

**code-identifier code-module** *Lazy-LList*  $\rightarrow$   
(*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])  
 $\equiv$

Code-Evaluation.App

(Code-Evaluation.App

(Code-Evaluation.Const

(STR "Coinductive-List.llist.LCons")

(Typerep.typerep TYPE('a  $\Rightarrow$  'a llist  $\Rightarrow$  'a llist)))

(*partial-term-of* TYPE('a) head))

(*partial-term-of* TYPE('a llist) tail)

*<proof>*)

**declare** *option.splits* [*split*]

**lemma** *Lazy-llist-inject* [*simp*]:

*Lazy-llist* xs = *Lazy-llist* ys  $\longleftrightarrow$  xs = ys

*<proof>*)

**lemma** *Lazy-llist-inverse* [*code*, *simp*]:

*force (Lazy-llist xs) = xs ()*  
 ⟨proof⟩

**lemma** *force-inverse [simp]:*  
*Lazy-llist (λ-. force xs) = xs*  
 ⟨proof⟩

**lemma** *LNil-Lazy-llist [code]: LNil = Lazy-llist (λ-. None)*  
 ⟨proof⟩

**lemma** *LCons-Lazy-llist [code, code-unfold]: LCons x xs = Lazy-llist (λ-. Some (x, xs))*  
 ⟨proof⟩

**lemma** *lnull-lazy [code]: lnull = Option.is-none ∘ force*  
 ⟨proof⟩

**declare** [[code drop: equal-class.equal :: 'a :: equal llist ⇒ -]]

**lemma** *equal-llist-Lazy-llist [code]:*  
*equal-class.equal (Lazy-llist xs) (Lazy-llist ys) ↔*  
*(case xs () of None ⇒ (case ys () of None ⇒ True | - ⇒ False)*  
*| Some (x, xs') ⇒*  
*(case ys () of None ⇒ False*  
*| Some (y, ys') ⇒ if x = y then equal-class.equal xs' ys' else False))*  
 ⟨proof⟩

**declare** [[code drop: corec-llist]]

**lemma** *corec-llist-Lazy-llist [code]:*  
*corec-llist IS-LNIL LHD endORMore LTL-end LTL-more b =*  
*Lazy-llist (λ-. if IS-LNIL b then None*  
*else Some (LHD b,*  
*if endORMore b then LTL-end b*  
*else corec-llist IS-LNIL LHD endORMore LTL-end LTL-more (LTL-more b)))*  
 ⟨proof⟩

**declare** [[code drop: unfold-llist]]

**lemma** *unfold-llist-Lazy-llist [code]:*  
*unfold-llist IS-LNIL LHD LTL b =*  
*Lazy-llist (λ-. if IS-LNIL b then None else Some (LHD b, unfold-llist IS-LNIL*  
*LHD LTL (LTL b)))*  
 ⟨proof⟩

**declare** [[code drop: case-llist]]

**lemma** *case-llist-Lazy-llist [code]:*  
*case-llist n c (Lazy-llist xs) = (case xs () of None ⇒ n | Some (x, ys) ⇒ c x ys)*



*<proof>*

**declare** *[[code drop: lappend]]*

**lemma** *lappend-Lazy-llist [code]:*

*lappend (Lazy-llist xs) ys =  
Lazy-llist (λ-. case xs () of None ⇒ force ys | Some (x, xs') ⇒ Some (x, lappend  
xs' ys))*  
*<proof>*

**declare** *[[code drop: lmap]]*

**lemma** *lmap-Lazy-llist [code]:*

*lmap f (Lazy-llist xs) = Lazy-llist (λ-. map-option (map-prod f (lmap f)) (xs ()))*  
*<proof>*

**declare** *[[code drop: lfinite]]*

**lemma** *lfinite-Lazy-llist [code]:*

*lfinite (Lazy-llist xs) = (case xs () of None ⇒ True | Some (x, ys) ⇒ lfinite ys)*  
*<proof>*

**declare** *[[code drop: list-of-aux]]*

**lemma** *list-of-aux-Lazy-llist [code]:*

*list-of-aux xs (Lazy-llist ys) =  
(case ys () of None ⇒ rev xs | Some (y, ys) ⇒ list-of-aux (y # xs) ys)*  
*<proof>*

**declare** *[[code drop: gen-llength]]*

**lemma** *gen-llength-Lazy-llist [code]:*

*gen-llength n (Lazy-llist xs) = (case xs () of None ⇒ enat n | Some (-, ys) ⇒  
gen-llength (n + 1) ys)*  
*<proof>*

**declare** *[[code drop: ltake]]*

**lemma** *ltake-Lazy-llist [code]:*

*ltake n (Lazy-llist xs) =  
Lazy-llist (λ-. if n = 0 then None else case xs () of None ⇒ None | Some (x, ys)  
⇒ Some (x, ltake (n - 1) ys))*  
*<proof>*

**declare** *[[code drop: ldropn]]*

**lemma** *ldropn-Lazy-llist [code]:*

*ldropn n (Lazy-llist xs) =  
Lazy-llist (λ-. if n = 0 then xs () else*

*case xs () of None  $\Rightarrow$  None | Some (x, ys)  $\Rightarrow$  force (ldropn (n - 1) ys)*

*<proof>*

**declare** *[[code drop: ltakeWhile]]*

**lemma** *ltakeWhile-Lazy-llist [code]:*

*ltakeWhile P (Lazy-llist xs) =*  
*Lazy-llist ( $\lambda$ -. case xs () of None  $\Rightarrow$  None | Some (x, ys)  $\Rightarrow$  if P x then Some (x, ltakeWhile P ys) else None)*  
*<proof>*

**declare** *[[code drop: ldropWhile]]*

**lemma** *ldropWhile-Lazy-llist [code]:*

*ldropWhile P (Lazy-llist xs) =*  
*Lazy-llist ( $\lambda$ -. case xs () of None  $\Rightarrow$  None | Some (x, ys)  $\Rightarrow$  if P x then force (ldropWhile P ys) else Some (x, ys))*  
*<proof>*

**declare** *[[code drop: lzip]]*

**lemma** *lzip-Lazy-llist [code]:*

*lzip (Lazy-llist xs) (Lazy-llist ys) =*  
*Lazy-llist ( $\lambda$ -. Option.bind (xs ()) ( $\lambda$ (x, xs'). map-option ( $\lambda$ (y, ys'). ((x, y), lzip xs' ys')) (ys ())))*  
*<proof>*

**declare** *[[code drop: gen-lset]]*

**lemma** *lset-Lazy-llist [code]:*

*gen-lset A (Lazy-llist xs) =*  
*(case xs () of None  $\Rightarrow$  A | Some (y, ys)  $\Rightarrow$  gen-lset (insert y A) ys)*  
*<proof>*

**declare** *[[code drop: lmember]]*

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

*lmember x (Lazy-llist xs) =*  
*(case xs () of None  $\Rightarrow$  False | Some (y, ys)  $\Rightarrow$  x = y  $\vee$  lmember x ys)*  
*<proof>*

**declare** *[[code drop: llist-all2]]*

**lemma** *llist-all2-Lazy-llist [code]:*

*llist-all2 P (Lazy-llist xs) (Lazy-llist ys) =*  
*(case xs () of None  $\Rightarrow$  ys () = None*  
*| Some (x, xs')  $\Rightarrow$  (case ys () of None  $\Rightarrow$  False*  
*| Some (y, ys')  $\Rightarrow$  P x y  $\wedge$  llist-all2 P xs' ys'))*

$\langle proof \rangle$

**declare**  $[[code\ drop:\ lhd]]$

**lemma** *lhd-Lazy-llist* [code]:

$lhd\ (Lazy-llist\ xs) = (case\ xs\ ()\ of\ None\ \Rightarrow\ undefined\ |\ Some\ (x,\ xs')\ \Rightarrow\ x)$   
 $\langle proof \rangle$

**declare**  $[[code\ drop:\ ltl]]$

**lemma** *ltl-Lazy-llist* [code]:

$ltl\ (Lazy-llist\ xs) = Lazy-llist\ (\lambda.\ case\ xs\ ()\ of\ None\ \Rightarrow\ None\ |\ Some\ (x,\ ys)\ \Rightarrow\ force\ ys)$   
 $\langle proof \rangle$

**declare**  $[[code\ drop:\ llast]]$

**lemma** *llast-Lazy-llist* [code]:

$llast\ (Lazy-llist\ xs) =$   
 $(case\ xs\ ()\ of$   
 $\quad None\ \Rightarrow\ undefined$   
 $\quad | Some\ (x,\ xs')\ \Rightarrow$   
 $\quad (case\ force\ xs'\ of\ None\ \Rightarrow\ x\ |\ Some\ (x',\ xs'')\ \Rightarrow\ llast\ (LCons\ x'\ xs'')))$   
 $\langle proof \rangle$

**declare**  $[[code\ drop:\ ldistinct]]$

**lemma** *ldistinct-Lazy-llist* [code]:

$ldistinct\ (Lazy-llist\ xs) =$   
 $(case\ xs\ ()\ of\ None\ \Rightarrow\ True\ |\ Some\ (x,\ ys)\ \Rightarrow\ x\ \notin\ lset\ ys\ \wedge\ ldistinct\ ys)$   
 $\langle proof \rangle$

**declare**  $[[code\ drop:\ lprefix]]$

**lemma** *lprefix-Lazy-llist* [code]:

$lprefix\ (Lazy-llist\ xs)\ (Lazy-llist\ ys) =$   
 $(case\ xs\ ()\ of$   
 $\quad None\ \Rightarrow\ True$   
 $\quad | Some\ (x,\ xs')\ \Rightarrow$   
 $\quad (case\ ys\ ()\ of\ None\ \Rightarrow\ False\ |\ Some\ (y,\ ys')\ \Rightarrow\ x = y\ \wedge\ lprefix\ xs'\ ys'))$   
 $\langle proof \rangle$

**declare**  $[[code\ drop:\ lstrict-prefix]]$

**lemma** *lstrict-prefix-Lazy-llist* [code]:

$lstrict-prefix\ (Lazy-llist\ xs)\ (Lazy-llist\ ys) \longleftrightarrow$   
 $(case\ ys\ ()\ of$   
 $\quad None\ \Rightarrow\ False$   
 $\quad | Some\ (y,\ ys')\ \Rightarrow$

(*case xs () of None  $\Rightarrow$  True | Some (x, xs')  $\Rightarrow$  x = y  $\wedge$  lstrict-prefix xs' ys')*)  
 <proof>

**declare** [[code drop: llcp]]

**lemma** llcp-Lazy-llist [code]:

llcp (Lazy-llist xs) (Lazy-llist ys) =  
 (*case xs () of None  $\Rightarrow$  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)*)

<proof>

**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')*)

<proof>

**declare** [[code drop: lfilter]]

**lemma** lfilter-Lazy-llist [code]:

lfilter P (Lazy-llist xs) =  
 Lazy-llist ( $\lambda$ -. *case xs () of None  $\Rightarrow$  None*  
 | *Some (x, ys)  $\Rightarrow$  if P x then Some (x, lfilter P ys) else force (lfilter*  
*P ys)*)

<proof>

**declare** [[code drop: lconcat]]

**lemma** lconcat-Lazy-llist [code]:

lconcat (Lazy-llist xss) =  
 Lazy-llist ( $\lambda$ -. *case xss () of None  $\Rightarrow$  None | Some (xs, xss')  $\Rightarrow$  force (lappend xs*  
*(lconcat xss')*)

<proof>

**declare** option.splits [split del]

**declare** Lazy-llist-def [simp del]

Simple ML test for laziness

<ML>

**hide-const** (open) force

**end**

## 9 Code generator setup to implement terminated lazy lists lazily

**theory** *Lazy-TLList* imports

*TLList*

*Lazy-LList*

**begin**

**code-identifier code-module** *Lazy-TLList*  $\rightarrow$

(*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*))

*<proof>*)

**declare** *Lazy-tllist-def* [*simp*]

```

declare sum.splits [split]

lemma TNil-Lazy-tllist [code]:
  TNil b = Lazy-tllist ( $\lambda$ -. Inr b)
  <proof>

lemma TCons-Lazy-tllist [code, code-unfold]:
  TCons x xs = Lazy-tllist ( $\lambda$ -. Inl (x, xs))
  <proof>

lemma Lazy-tllist-inverse [simp, code]:
  force (Lazy-tllist xs) = xs ()
  <proof>

declare [[code drop: equal-class.equal :: (-, -) tllist  $\Rightarrow$  -]]

lemma equal-tllist-Lazy-tllist [code]:
  equal-class.equal (Lazy-tllist xs) (Lazy-tllist ys) =
  (case xs () of
    Inr b  $\Rightarrow$  (case ys () of Inr b'  $\Rightarrow$  b = b' | -  $\Rightarrow$  False)
    | Inl (x, xs')  $\Rightarrow$ 
    (case ys () of Inr b'  $\Rightarrow$  False | Inl (y, ys')  $\Rightarrow$  if x = y then equal-class.equal xs'
    ys' else False))
  <proof>

declare
  [[code drop: thd ttl]
  thd-def [code]
  ttl-def [code]]

declare [[code drop: is-TNil]]

lemma is-TNil-code [code]:
  is-TNil (Lazy-tllist xs)  $\longleftrightarrow$ 
  (case xs () of Inl -  $\Rightarrow$  False | Inr -  $\Rightarrow$  True)
  <proof>

declare [[code drop: corec-tllist]]

lemma corec-tllist-Lazy-tllist [code]:
  corec-tllist IS-TNIL TNIL THD endORmore TTL-end TTL-more b = Lazy-tllist
  ( $\lambda$ -. if IS-TNIL b then Inr (TNIL b)
    else Inl (THD b, if endORmore b then TTL-end b else corec-tllist IS-TNIL
    TNIL THD endORmore TTL-end TTL-more (TTL-more b)))
  <proof>

declare [[code drop: unfold-tllist]]

lemma unfold-tllist-Lazy-tllist [code]:

```

$unfold\_tllist\ IS\ \text{TNIL}\ TNIL\ THD\ TTL\ b = Lazy\_tllist$   
 $(\lambda-. \text{if}\ IS\ \text{TNIL}\ b\ \text{then}\ Inr\ (TNIL\ b)$   
 $\quad \text{else}\ Inl\ (THD\ b,\ unfold\_tllist\ IS\ \text{TNIL}\ TNIL\ THD\ TTL\ (TTL\ b)))$   
 $\langle proof \rangle$

**declare**  $[[code\ drop:\ case\_tllist]]$

**lemma**  $case\_tllist\ Lazy\_tllist$   $[code]:$   
 $case\_tllist\ n\ c\ (Lazy\_tllist\ xs) =$   
 $(case\ xs\ ()\ of\ Inl\ (x,\ ys) \Rightarrow c\ x\ ys\ | Inr\ b \Rightarrow n\ b)$   
 $\langle proof \rangle$

**declare**  $[[code\ drop:\ tllist\_of\_llist]]$

**lemma**  $tllist\_of\_llist\ Lazy\_llist$   $[code]:$   
 $tllist\_of\_llist\ b\ (Lazy\_llist\ xs) =$   
 $Lazy\_tllist\ (\lambda-. \text{case}\ xs\ ()\ of\ None \Rightarrow Inr\ b\ | Some\ (x,\ ys) \Rightarrow Inl\ (x,\ tllist\_of\_llist$   
 $\quad b\ ys))$   
 $\langle proof \rangle$

**declare**  $[[code\ drop:\ terminal]]$

**lemma**  $terminal\ Lazy\_tllist$   $[code]:$   
 $terminal\ (Lazy\_tllist\ xs) =$   
 $(case\ xs\ ()\ of\ Inl\ (-,\ ys) \Rightarrow terminal\ ys\ | Inr\ b \Rightarrow b)$   
 $\langle proof \rangle$

**declare**  $[[code\ drop:\ tmap]]$

**lemma**  $tmap\ Lazy\_tllist$   $[code]:$   
 $tmap\ f\ g\ (Lazy\_tllist\ xs) =$   
 $Lazy\_tllist\ (\lambda-. \text{case}\ xs\ ()\ of\ Inl\ (x,\ ys) \Rightarrow Inl\ (f\ x,\ tmap\ f\ g\ ys)\ | Inr\ b \Rightarrow Inr\ (g$   
 $\quad b))$   
 $\langle proof \rangle$

**declare**  $[[code\ drop:\ tappend]]$

**lemma**  $tappend\ Lazy\_tllist$   $[code]:$   
 $tappend\ (Lazy\_tllist\ xs)\ ys =$   
 $Lazy\_tllist\ (\lambda-. \text{case}\ xs\ ()\ of\ Inl\ (x,\ xs') \Rightarrow Inl\ (x,\ tappend\ xs'\ ys)\ | Inr\ b \Rightarrow force$   
 $\quad (ys\ b))$   
 $\langle proof \rangle$

**declare**  $[[code\ drop:\ lappendt]]$

**lemma**  $lappendt\ Lazy\_llist$   $[code]:$   
 $lappendt\ (Lazy\_llist\ xs)\ ys =$   
 $Lazy\_tllist\ (\lambda-. \text{case}\ xs\ ()\ of\ None \Rightarrow force\ ys\ | Some\ (x,\ xs') \Rightarrow Inl\ (x,\ lappendt$   
 $\quad xs'\ ys))$

$\langle proof \rangle$

**declare**  $[[code\ drop:\ TLList.tfilter']]$

**lemma** *tfilter'-Lazy-tllist*  $[code]:$

$TLList.tfilter' b P (Lazy-tllist\ xs) =$   
 $Lazy-tllist\ (\lambda-. case\ xs\ ()\ of\ Inl\ (x,\ xs') \Rightarrow if\ P\ x\ then\ Inl\ (x,\ TLList.tfilter'\ b\ P$   
 $xs')\ else\ force\ (TLList.tfilter'\ b\ P\ xs')\ |\ Inr\ b' \Rightarrow Inr\ b')$   
 $\langle proof \rangle$

**declare**  $[[code\ drop:\ TLList.tconcat']]$

**lemma** *tconcat-Lazy-tllist*  $[code]:$

$TLList.tconcat' b (Lazy-tllist\ xss) =$   
 $Lazy-tllist\ (\lambda-. case\ xss\ ()\ of\ Inr\ b' \Rightarrow Inr\ b' |\ Inl\ (xs,\ xss') \Rightarrow force\ (lappendt\ xs$   
 $(TLList.tconcat'\ b\ xss')))$   
 $\langle proof \rangle$

**declare**  $[[code\ drop:\ tlolist-all2]]$

**lemma** *tlolist-all2-Lazy-tlolist*  $[code]:$

$tlolist-all2\ P\ Q (Lazy-tlolist\ xs) (Lazy-tlolist\ ys) \longleftrightarrow$   
 $(case\ xs\ ()\ of$   
 $Inr\ b \Rightarrow (case\ ys\ ()\ of\ Inr\ b' \Rightarrow Q\ b\ b' |\ Inl\ - \Rightarrow False)$   
 $|\ Inl\ (x,\ xs') \Rightarrow (case\ ys\ ()\ of\ Inr\ - \Rightarrow False |\ Inl\ (y,\ ys') \Rightarrow P\ x\ y \wedge tlolist-all2\ P$   
 $Q\ xs'\ ys'))$   
 $\langle proof \rangle$

**declare**  $[[code\ drop:\ llist-of-tlolist]]$

**lemma** *llist-of-tlolist-Lazy-tlolist*  $[code]:$

$llist-of-tlolist (Lazy-tlolist\ xs) =$   
 $Lazy-llist\ (\lambda-. case\ xs\ ()\ of\ Inl\ (x,\ ys) \Rightarrow Some\ (x,\ llist-of-tlolist\ ys) |\ Inr\ b \Rightarrow$   
 $None)$   
 $\langle proof \rangle$

**declare**  $[[code\ drop:\ tnth]]$

**lemma** *tnth-Lazy-tlolist*  $[code]:$

$tnth (Lazy-tlolist\ xs) n =$   
 $(case\ xs\ ()\ of\ Inr\ b \Rightarrow undefined\ n |\ Inl\ (x,\ ys) \Rightarrow if\ n = 0\ then\ x\ else\ tnth\ ys\ (n$   
 $- 1))$   
 $\langle proof \rangle$

**declare**  $[[code\ drop:\ gen-tlength]]$

**lemma** *gen-tlength-Lazy-tlolist*  $[code]:$

$gen-tlength\ n (Lazy-tlolist\ xs) =$   
 $(case\ xs\ ()\ of\ Inr\ b \Rightarrow enat\ n |\ Inl\ (-,\ xs') \Rightarrow gen-tlength\ (n + 1)\ xs')$



*<proof>*

**declare** [[code drop: tdropn]]

**lemma** *tdropn-Lazy-tllist* [code]:

*tdropn n (Lazy-tllist xs) =*  
*Lazy-tllist (λ-. if n = 0 then xs () else case xs () of Inr b ⇒ Inr b | Inl (x, xs') ⇒ force (tdropn (n - 1) xs'))*  
*<proof>*

**declare** *Lazy-tllist-def* [simp del]

**declare** *sum.splits* [split del]

Simple ML test for laziness

*<ML>*

**hide-const** (**open**) *force*

**end**

## 10 CCPO topologies

**theory** *CCPO-Topology*

**imports**

*HOL-Analysis.Extended-Real-Limits*

*../Coinductive-Nat*

**begin**

**lemma** *dropWhile-append*:

*dropWhile P (xs @ ys) = (if ∀ x ∈ set xs. P x then dropWhile P ys else dropWhile P xs @ ys)*  
*<proof>*

**lemma** *dropWhile-False*:  $(\bigwedge x. x \in \text{set } xs \implies P x) \implies \text{dropWhile } P \text{ } xs = []$

*<proof>*

**abbreviation** (**in order**) *chain*  $\equiv$  *Complete-Partial-Order.chain* ( $\leq$ )

**lemma** (**in linorder**) *chain-linorder*: *chain C*

*<proof>*

**lemma** *continuous-add-ereal*:

**assumes**  $0 \leq t$

**shows** *continuous-on*  $\{-\infty::ereal <..\}$   $(\lambda x. t + x)$

*<proof>*

**lemma** *tendsto-add-ereal*:

$0 \leq x \implies 0 \leq y \implies (f \longrightarrow y) F \implies ((\lambda z. x + f z :: ereal) \longrightarrow x + y) F$

*<proof>*

**lemma** *tendsto-LimI*:  $(f \longrightarrow y) F \implies (f \longrightarrow \text{Lim } F f) F$   
 ⟨proof⟩

## 10.1 The filter $at'$

**abbreviation** (in *ccpo*) *compact-element*  $\equiv$  *ccpo.compact Sup* ( $\leq$ )

**lemma** *tendsto-unique-eventually*:

**fixes**  $x x' :: 'a :: t2\text{-space}$

**shows**  $F \neq \text{bot} \implies \text{eventually } (\lambda x. f x = g x) F \implies (f \longrightarrow x) F \implies (g \longrightarrow x') F \implies x = x'$

⟨proof⟩

**lemma** (in *ccpo*) *ccpo-Sup-upper2*:  $\text{chain } C \implies x \in C \implies y \leq x \implies y \leq \text{Sup } C$   
 ⟨proof⟩

**lemma** *tendsto-open-vimage*:  $(\bigwedge B. \text{open } B \implies \text{open } (f -' B)) \implies f -l \rightarrow f l$   
 ⟨proof⟩

**lemma** *open-vimageI*:  $(\bigwedge x. f -x \rightarrow f x) \implies \text{open } A \implies \text{open } (f -' A)$   
 ⟨proof⟩

**lemma** *principal-bot*:  $\text{principal } x = \text{bot} \iff x = \{\}$   
 ⟨proof⟩

**definition**  $at' x = (\text{if } \text{open } \{x\} \text{ then } \text{principal } \{x\} \text{ else } at\ x)$

**lemma** *at'-bot*:  $at' x \neq \text{bot}$   
 ⟨proof⟩

**lemma** *tendsto-id-at'*[*simp, intro*]:  $((\lambda x. x) \longrightarrow x) (at' x)$   
 ⟨proof⟩

**lemma** *cont-at'*:  $(f \longrightarrow f x) (at' x) \iff f -x \rightarrow f x$   
 ⟨proof⟩

## 10.2 The type class *ccpo-topology*

Temporarily relax type constraints for *open*.

⟨ML⟩

**class** *ccpo-topology* = *open* + *ccpo* +

**assumes** *open-ccpo*:  $\text{open } A \iff (\forall C. \text{chain } C \longrightarrow C \neq \{\} \longrightarrow \text{Sup } C \in A \longrightarrow C \cap A \neq \{\})$

**begin**

**lemma** *open-ccpoD*:

**assumes**  $\text{open } A \text{ chain } C \ C \neq \{\} \ \text{Sup } C \in A$

**shows**  $\exists c \in C. \forall c' \in C. c \leq c' \longrightarrow c' \in A$   
 ⟨proof⟩

**lemma** *open-ccpo-Ici*: *open* {.. b}  
 ⟨proof⟩

**subclass** *topological-space*  
 ⟨proof⟩

**lemma** *closed-ccpo*: *closed* A  $\longleftrightarrow$  ( $\forall C. \text{chain } C \longrightarrow C \neq \{\} \longrightarrow C \subseteq A \longrightarrow \text{Sup } C \in A$ )  
 ⟨proof⟩

**lemma** *closed-admissible*: *closed* {x. P x}  $\longleftrightarrow$  *ccpo.admissible* Sup ( $\leq$ ) P  
 ⟨proof⟩

**lemma** *open-singletonI-compact*: *compact-element* x  $\implies$  *open* {x}  
 ⟨proof⟩

**lemma** *closed-Ici*: *closed* {.. b}  
 ⟨proof⟩

**lemma** *closed-Iic*: *closed* {b ..}  
 ⟨proof⟩

*ccpo-topologys* are also *t2-spaces*. This is necessary to have a unique continuous extension.

**subclass** *t2-space*  
 ⟨proof⟩

**end**

**lemma** *tendsto-le-ccpo*:  
**fixes** f g :: 'a  $\Rightarrow$  'b::*ccpo-topology*  
**assumes** F:  $\neg$  *trivial-limit* F  
**assumes** x: (f  $\longrightarrow$  x) F **and** y: (g  $\longrightarrow$  y) F  
**assumes** ev: *eventually* ( $\lambda x. g\ x \leq f\ x$ ) F  
**shows** y  $\leq$  x  
 ⟨proof⟩

**lemma** *tendsto-ccpoI*:  
**fixes** f :: 'a::*ccpo-topology*  $\Rightarrow$  'b::*ccpo-topology*  
**shows** ( $\bigwedge C. \text{chain } C \implies C \neq \{\} \implies \text{chain } (f\ 'C) \wedge f\ (\text{Sup } C) = \text{Sup } (f\ 'C)$ )  
 $\implies f\ -x \rightarrow f\ x$   
 ⟨proof⟩

**lemma** *tendsto-mcont*:  
**assumes** *mcont*: *mcont* Sup ( $\leq$ ) Sup ( $\leq$ ) (f :: 'a :: *ccpo-topology*  $\Rightarrow$  'b :: *ccpo-topology*)  
**shows** f  $-l \rightarrow f\ l$

*<proof>*

### 10.3 Instances for *ccpo-topologys* and continuity theorems

**instantiation** *set* :: (*type*) *ccpo-topology*  
**begin**

**definition** *open-set* :: 'a *set set*  $\Rightarrow$  *bool* **where**  
*open-set* *A*  $\longleftrightarrow (\forall C. \text{chain } C \longrightarrow C \neq \{\} \longrightarrow \text{Sup } C \in A \longrightarrow C \cap A \neq \{\})$

**instance**  
*<proof>*

**end**

**instantiation** *enat* :: *ccpo-topology*  
**begin**

**instance**  
*<proof>*

**end**

**lemmas** *tendsto-inf2*[*THEN tendsto-compose, tendsto-intros*] =  
*tendsto-mcont*[*OF mcont-inf2*]

**lemma** *isCont-inf2*[*THEN isCont-o2*[*rotated*]]:  
*isCont* ( $\lambda x. x \sqcap y$ ) (*z* :: - :: {*ccpo-topology, complete-distrib-lattice*})  
*<proof>*

**lemmas** *tendsto-sup1*[*THEN tendsto-compose, tendsto-intros*] =  
*tendsto-mcont*[*OF mcont-sup1*]

**lemma** *isCont-If*: *isCont* *f* *x*  $\Longrightarrow$  *isCont* *g* *x*  $\Longrightarrow$  *isCont* ( $\lambda x. \text{if } Q \text{ then } f \ x \text{ else } g \ x$ ) *x*  
*<proof>*

**lemma** *isCont-enat-case*: *isCont* (*f* (*epred* *n*)) *x*  $\Longrightarrow$  *isCont* *g* *x*  $\Longrightarrow$  *isCont* ( $\lambda x. \text{co.case-enat } (g \ x) (\lambda n. f \ n \ x) \ n$ ) *x*  
*<proof>*

**end**

## 11 A CCPO topology on lazy lists with examples

**theory** *LList-CCPO-Topology* **imports**  
*CCPO-Topology*  
*../Coinductive-List-Prefix*  
**begin**

**lemma** *closed-Collect-eq-isCont*:  
**fixes**  $f\ g :: 'a :: t2\text{-space} \Rightarrow 'b :: t2\text{-space}$   
**assumes**  $f: \bigwedge x. \text{isCont } f\ x$  **and**  $g: \bigwedge x. \text{isCont } g\ x$   
**shows** *closed*  $\{x. f\ x = g\ x\}$   
 $\langle \text{proof} \rangle$

**instantiation**  $l\text{list} :: (\text{type})\ \text{ccpo-topology}$   
**begin**

**definition** *open-llist*  $:: 'a\ \text{llist}\ \text{set} \Rightarrow \text{bool}$  **where**  
 $\text{open-llist } A \longleftrightarrow (\forall C. \text{chain } C \longrightarrow C \neq \{\} \longrightarrow \text{Sup } C \in A \longrightarrow C \cap A \neq \{\})$

**instance**  
 $\langle \text{proof} \rangle$

**end**

## 11.1 Continuity and closedness of predefined constants

**lemma** *tendsto-mcont-llist*:  $m\text{cont } l\text{Sup } l\text{prefix } l\text{Sup } l\text{prefix } f \Longrightarrow f\ -l \rightarrow f\ l$   
 $\langle \text{proof} \rangle$

**lemma** *tendsto-ltl*[*THEN tendsto-compose, tendsto-intros*]:  $l\text{tl } -l \rightarrow l\text{tl } l$   
 $\langle \text{proof} \rangle$

**lemma** *tendsto-lappend2*[*THEN tendsto-compose, tendsto-intros*]:  $l\text{append } l\ -l' \rightarrow l\text{append } l\ l'$   
 $\langle \text{proof} \rangle$

**lemma** *tendsto-LCons*[*THEN tendsto-compose, tendsto-intros*]:  $L\text{Cons } x\ -l \rightarrow L\text{Cons } x\ l$   
 $\langle \text{proof} \rangle$

**lemma** *tendsto-lmap*[*THEN tendsto-compose, tendsto-intros*]:  $l\text{map } f\ -l \rightarrow l\text{map } f\ l$   
 $\langle \text{proof} \rangle$

**lemma** *tendsto-llength*[*THEN tendsto-compose, tendsto-intros*]:  $l\text{length } -l \rightarrow l\text{length } l$   
 $\langle \text{proof} \rangle$

**lemma** *tendsto-lset*[*THEN tendsto-compose, tendsto-intros*]:  $l\text{set } -l \rightarrow l\text{set } l$   
 $\langle \text{proof} \rangle$

**lemma** *open-lhd*:  $\text{open } \{l. \neg l\text{null } l \wedge l\text{hd } l = x\}$   
 $\langle \text{proof} \rangle$

**lemma** *open-LCons'*: **assumes**  $A: \text{open } A$  **shows**  $\text{open } (L\text{Cons } x\ 'A)$

*<proof>*

**lemma** *open-Ici*:  $lfinite\ xs \implies open\ \{xs\ ..\}$   
*<proof>*

**lemma** *open-lfinite[simp]*:  $lfinite\ x \implies open\ \{x\}$   
*<proof>*

**lemma** *open-singleton-iff-lfinite*:  $open\ \{x\} \longleftrightarrow lfinite\ x$   
*<proof>*

**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>*

**lemma** *closure-lfinite*:  $closure\ \{xs.\ lfinite\ xs\} = UNIV$   
*<proof>*

**lemma** *closed-ldistinct*:  $closed\ \{xs.\ ldistinct\ xs\}$   
*<proof>*

**lemma** *ldistinct-closure*:  $\{xs.\ ldistinct\ xs\} = closure\ \{xs.\ lfinite\ xs \wedge ldistinct\ xs\}$   
*<proof>*

**lemma** *closed-ldistinct'*:  $(\bigwedge x.\ isCont\ f\ x) \implies closed\ \{xs.\ ldistinct\ (f\ xs)\}$   
*<proof>*

**lemma** *closed-lsorted*:  $closed\ \{xs.\ lsorted\ xs\}$   
*<proof>*

**lemma** *lsorted-closure*:  $\{xs.\ lsorted\ xs\} = closure\ \{xs.\ lfinite\ xs \wedge lsorted\ xs\}$   
*<proof>*

**lemma** *closed-lsorted'*:  $(\bigwedge x.\ isCont\ f\ x) \implies closed\ \{xs.\ lsorted\ (f\ xs)\}$   
*<proof>*

**lemma** *closed-in-lset*:  $closed\ \{l.\ x \in lset\ l\}$   
*<proof>*

**lemma** *closed-llist-all2*:  
   $closed\ \{(x, y).\ llist-all2\ R\ x\ y\}$   
*<proof>*

**lemma** *closed-list-all2*:  
  **fixes**  $f\ g :: 'b::t2-space \Rightarrow 'a\ llist$   
  **assumes**  $f: \bigwedge x.\ isCont\ f\ x$  **and**  $g: \bigwedge x.\ isCont\ g\ x$   
  **shows**  $closed\ \{x.\ llist-all2\ R\ (f\ x)\ (g\ x)\}$

*<proof>*

**lemma** *at-botI-lfinite[simp]*:  $lfinite\ l \implies at\ l = bot$   
*<proof>*

**lemma** *at-eq-lfinite*:  $at\ l = (if\ lfinite\ l\ then\ bot\ else\ at'\ l)$   
*<proof>*

**lemma** *eventually-lfinite*:  $eventually\ lfinite\ (at'\ x)$   
*<proof>*

**lemma** *eventually-nhds-llist*:  
 $eventually\ P\ (nhds\ l) \longleftrightarrow (\exists\ xs \leq l. lfinite\ xs \wedge (\forall\ ys \geq xs. ys \leq l \longrightarrow P\ ys))$   
*<proof>*

**lemma** *nhds-lfinite*:  $lfinite\ l \implies nhds\ l = principal\ \{l\}$   
*<proof>*

**lemma** *eventually-at'-llist*:  
 $eventually\ P\ (at'\ l) \longleftrightarrow (\exists\ xs \leq l. lfinite\ xs \wedge (\forall\ ys \geq xs. lfinite\ ys \longrightarrow ys \leq l \longrightarrow P\ ys))$   
*<proof>*

**lemma** *eventually-at'-llistI*:  $(\bigwedge xs. lfinite\ xs \implies xs \leq l \implies P\ xs) \implies eventually\ P\ (at'\ l)$   
*<proof>*

**lemma** *Lim-at'-lfinite*:  $lfinite\ xs \implies Lim\ (at'\ xs)\ f = f\ xs$   
*<proof>*

**lemma** *filterlim-at'-list*:  
 $(f \longrightarrow y)\ (at'\ (x::'a\ llist)) \implies f\ -x \rightarrow y$   
*<proof>*

**lemma** *tendsto-mcont-llist'*:  $mcont\ lSup\ lprefix\ lSup\ lprefix\ f \implies (f \longrightarrow f\ x)\ (at'\ (x::'a\ llist))$   
*<proof>*

**lemma** *tendsto-closed*:  
**assumes**  $eq: closed\ \{x. P\ x\}$   
**assumes**  $ev: \bigwedge ys. lfinite\ ys \implies ys \leq x \implies P\ ys$   
**shows**  $P\ x$   
*<proof>*

**lemma** *tendsto-Sup-at'*:  
**fixes**  $f :: 'a\ llist \Rightarrow 'b::ccpo-topology$   
**assumes**  $f: \bigwedge x\ y. x \leq y \implies lfinite\ x \implies lfinite\ y \implies f\ x \leq f\ y$   
**shows**  $(f \longrightarrow (Sup\ (f'\{xs. lfinite\ xs \wedge xs \leq l\})))\ (at'\ l)$

*<proof>*

**lemma** *tendsto-Lim-at'*:

**fixes**  $f :: 'a \text{ llist} \Rightarrow 'b::\text{ccpo-topology}$

**assumes**  $f: \bigwedge l. f\ l = \text{Lim}\ (at'\ l)\ f'$

**assumes** *mono*:  $\bigwedge x\ y. x \leq y \implies \text{lfinite}\ x \implies \text{lfinite}\ y \implies f'\ x \leq f'\ y$

**shows**  $(f \longrightarrow f\ l)\ (at'\ l)$

*<proof>*

**lemma** *isCont-LCons*[*THEN isCont-o2*[*rotated*]]:  $\text{isCont}\ (LCons\ x)\ l$

*<proof>*

**lemma** *isCont-lmap*[*THEN isCont-o2*[*rotated*]]:  $\text{isCont}\ (lmap\ f)\ l$

*<proof>*

**lemma** *isCont-lappend*[*THEN isCont-o2*[*rotated*]]:  $\text{isCont}\ (lappend\ xs)\ ys$

*<proof>*

**lemma** *isCont-lset*[*THEN isCont-o2*[*rotated*]]:  $\text{isCont}\ \text{lset}\ xs$

*<proof>*

## 11.2 Define *lfilter* as continuous extension

**definition**  $\text{lfilter}'\ P\ l = \text{Lim}\ (at'\ l)\ (\lambda xs. \text{llist-of}\ (\text{filter}\ P\ (\text{list-of}\ xs)))$

**lemma** *tendsto-lfilter*:  $(\text{lfilter}'\ P \longrightarrow \text{lfilter}'\ P\ xs)\ (at'\ xs)$

*<proof>*

**lemma** *isCont-lfilter*[*THEN isCont-o2*[*rotated*]]:  $\text{isCont}\ (\text{lfilter}'\ P)\ l$

*<proof>*

**lemma** *lfilter'-lfinite*[*simp*]:  $\text{lfinite}\ xs \implies \text{lfilter}'\ P\ xs = \text{llist-of}\ (\text{filter}\ P\ (\text{list-of}\ xs))$

*<proof>*

**lemma** *lfilter'-LNil*:  $\text{lfilter}'\ P\ LNil = LNil$

*<proof>*

**lemma** *lfilter'-LCons* [*simp*]:  $\text{lfilter}'\ P\ (LCons\ a\ xs) = (\text{if}\ P\ a\ \text{then}\ LCons\ a\ (\text{lfilter}'\ P\ xs)\ \text{else}\ \text{lfilter}'\ P\ xs)$

*<proof>*

**lemma** *ldistinct-lfilter'*:  $\text{ldistinct}\ l \implies \text{ldistinct}\ (\text{lfilter}'\ P\ l)$

*<proof>*

**lemma** *lfilter'-lmap*:  $\text{lfilter}'\ P\ (lmap\ f\ xs) = lmap\ f\ (\text{lfilter}'\ (P \circ f)\ xs)$

*<proof>*



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

**lemma** *lfilter'-LNil-I[simp]*:  $(\forall x \in lset xs. \neg P x) \implies lfilter' P xs = LNil$   
 ⟨proof⟩

**lemma** *lset-lfilter'*:  $lset (lfilter' P xs) = lset xs \cap \{x. P x\}$   
 ⟨proof⟩

**lemma** *lfilter'-eq-LNil-iff*:  $lfilter' P xs = LNil \iff (\forall x \in lset xs. \neg P x)$   
 ⟨proof⟩

**lemma** *lfilter'-eq-lfilter*:  $lfilter' P xs = lfilter P xs$   
 ⟨proof⟩

### 11.3 Define *lconcat* as continuous extension

**definition** *lconcat'*  $xs = Lim (at' xs) (\lambda xs. foldr lappend (list-of xs) LNil)$

**lemma** *tendsto-lconcat'*:  $(lconcat' \longrightarrow lconcat' xss) (at' xss)$   
 ⟨proof⟩

**lemma** *isCont-lconcat'* [THEN *isCont-o2[rotated]*]: *isCont* *lconcat' l*  
 ⟨proof⟩

**lemma** *lconcat'-lfinite[simp]*:  $lfinite xs \implies lconcat' xs = foldr lappend (list-of xs) LNil$   
 ⟨proof⟩

**lemma** *lconcat'-LNil*:  $lconcat' LNil = LNil$   
 ⟨proof⟩

**lemma** *lconcat'-LCons [simp]*:  $lconcat' (LCons l xs) = lappend l (lconcat' xs)$   
 ⟨proof⟩

**lemma** *lmap-lconcat*:  $lmap f (lconcat' xss) = lconcat' (lmap (lmap f) (xss::'a llist llist))$   
 ⟨proof⟩

**lemmas** *tendsto-Sup* [THEN *tendsto-compose, tendsto-intros*] =  
*mcont-SUP* [OF *mcont-id' mcont-const, THEN tendsto-mcont*]

**lemma**

**assumes** *fin*:  $\forall xs \in lset xss. lfinite xs$

**shows**  $lset (lconcat' xss) = (\bigcup xs \in lset xss. lset xs)$  (**is** *?lhs* = *?rhs*)

⟨proof⟩

## 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'*:

$(ldropWhile' P \longrightarrow ldropWhile' P xs) (at' xs)$   
 $\langle proof \rangle$

**lemma** *isCont-ldropWhile'[THEN isCont-o2[rotated]]*:  $isCont (ldropWhile' P) l$

$\langle proof \rangle$

**lemma** *ldropWhile'-lfinite[simp]*:  $lfinite xs \implies ldropWhile' P xs = llist-of (dropWhile P (list-of xs))$

$\langle proof \rangle$

**lemma** *ldropWhile'-LNil*:  $ldropWhile' P LNil = LNil$

$\langle proof \rangle$

**lemma** *ldropWhile'-LCons [simp]*:  $ldropWhile' P (LCons l xs) = (if P l then ldropWhile' P xs else LCons l xs)$

$\langle proof \rangle$

**lemma** *ldropWhile' P (lmap f xs) = lmap f (ldropWhile' (P o f) xs)*

$\langle proof \rangle$

**lemma** *ldropWhile'-LNil-I[simp]*:  $\forall x \in lset xs. P x \implies ldropWhile' P xs = LNil$

$\langle proof \rangle$

**lemma** *lnull-ldropWhile'*:  $lnull (ldropWhile' P xs) \longleftrightarrow (\forall x \in lset xs. P x) \text{ (is ?lhs } \longleftrightarrow \cdot)$

$\langle proof \rangle$

**lemma** *lhd-lfilter'*:  $lhd (lfilter' P xs) = lhd (ldropWhile' (Not o P) xs)$

$\langle proof \rangle$

## 11.5 Define *ldrop* as continuous extension

**primrec** *edrop where*

$edrop n [] = []$   
 $| edrop n (x \# xs) = (case n of eSuc n \Rightarrow edrop n xs \mid 0 \Rightarrow x \# xs)$

**lemma** *mono-edrop*:  $edrop n xs \leq edrop n (xs @ ys)$

$\langle proof \rangle$

**lemma** *edrop-mono*:  $xs \leq ys \implies edrop n xs \leq edrop n ys$

$\langle proof \rangle$

**definition**  $ldrop' n xs = Lim (at' xs) (llist-of \circ edrop n \circ list-of)$

**lemma** *ldrop'-lfinite[simp]*:  $lfinite\ xs \implies ldrop'\ n\ xs = llist-of\ (edrop\ n\ (list-of\ xs))$   
 ⟨proof⟩

**lemma** *tendsto-ldrop'*:  $(ldrop'\ n \longrightarrow ldrop'\ n\ l)\ (at'\ l)$   
 ⟨proof⟩

**lemma** *isCont-ldrop'[THEN isCont-o2[rotated]]*:  $isCont\ (ldrop'\ n)\ l$   
 ⟨proof⟩

**lemma** *ldrop' n LNil = LNil*  
 ⟨proof⟩

**lemma** *ldrop' n (LCons x xs) = (case n of 0  $\Rightarrow$  LCons x xs | eSuc n  $\Rightarrow$  ldrop' n xs)*  
 ⟨proof⟩

**primrec** *up* :: 'a :: order  $\Rightarrow$  'a list  $\Rightarrow$  'a list **where**  
 $up\ a\ [] = []$   
 $| up\ a\ (x \# xs) = (if\ a < x\ then\ x \# up\ a\ xs\ else\ up\ a\ xs)$

**lemma** *set-upD*:  $x \in set\ (up\ y\ xs) \implies x \in set\ xs \wedge y < x$   
 ⟨proof⟩

**lemma** *prefix-up*:  $prefix\ (up\ a\ xs)\ (up\ a\ (xs\ @\ ys))$   
 ⟨proof⟩

**lemma** *mono-up*:  $xs \leq ys \implies up\ a\ xs \leq up\ a\ ys$   
 ⟨proof⟩

**lemma** *sorted-up*:  $sorted\ (up\ a\ xs)$   
 ⟨proof⟩

## 11.6 Define more functions on lazy lists as continuous extensions

**definition** *lup a xs = Lim (at' xs) ( $\lambda xs. llist-of\ (up\ a\ (list-of\ xs))$ )*

**lemma** *tendsto-lup*:  $(lup\ a \longrightarrow lup\ a\ xs)\ (at'\ xs)$   
 ⟨proof⟩

**lemma** *isCont-lup[THEN isCont-o2[rotated]]*:  $isCont\ (lup\ a)\ l$   
 ⟨proof⟩

**lemma** *lup-lfinite[simp]*:  $lfinite\ xs \implies lup\ a\ xs = llist-of\ (up\ a\ (list-of\ xs))$   
 ⟨proof⟩

**lemma** *lup-LNil*:  $lup\ a\ LNil = LNil$   
 ⟨proof⟩

**lemma** *lup-LCons* [*simp*]:  $\text{lup } a \text{ (LCons } x \text{ } xs) = (\text{if } a < x \text{ then LCons } x \text{ (lup } x \text{ } xs) \text{ else lup } a \text{ } xs)$   
 ⟨*proof*⟩

**lemma** *lset-lup*:  $\text{lset (lup } x \text{ } xs) \subseteq \text{lset } xs \cap \{y. x < y\}$   
 ⟨*proof*⟩

**lemma** *lsorted-lup*:  $\text{lsorted (lup (a::'a::linorder) l)}$   
 ⟨*proof*⟩

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

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

**end**

**declare** *lup'.mono*[*cont-intro*]

**lemma** *monotone-lup'*:  $\text{monotone (rel-prod (=) lprefix) lprefix } (\lambda(a, xs). \text{lup}' a \text{ } xs)$   
 ⟨*proof*⟩

**lemma** *mono2mono-lup'2*[*THEN llist.mono2mono, simp, cont-intro*]:  
**shows** *monotone-lup'2*:  $\text{monotone lprefix lprefix (lup}' a)$   
 ⟨*proof*⟩

**lemma** *mcont-lup'*:  $\text{mcont (prod-lub the-Sup lSup) (rel-prod (=) lprefix) lSup lprefix } (\lambda(a, xs). \text{lup}' a \text{ } xs)$   
 ⟨*proof*⟩

**lemma** *mcont2mcont-lup'2*[*THEN llist.mcont2mcont, simp, cont-intro*]:  
**shows** *mcont-lup'2*:  $\text{mcont lSup lprefix lSup lprefix (lup}' a)$   
 ⟨*proof*⟩

**simps-of-case** *lup'-simps* [*simp*]: *lup'.simps*

**lemma** *lset-lup'-subset*:  
**fixes** x :: - :: preorder  
**shows**  $\text{lset (lup}' x \text{ } xs) \subseteq \text{lset } xs \cap \{y. x < y\}$   
 ⟨*proof*⟩

**lemma** *in-lset-lup'D*:  
**fixes** x :: - :: preorder  
**assumes**  $y \in \text{lset (lup}' x \text{ } xs)$   
**shows**  $y \in \text{lset } xs \wedge x < y$   
 ⟨*proof*⟩

**lemma** *lsorted-lup'*:  
**fixes**  $x :: - :: \text{preorder}$   
**shows** *lsorted* (*lup'*  $x$   $xs$ )  
 $\langle \text{proof} \rangle$

**lemma** *ldistinct-lup'*:  
**fixes**  $x :: - :: \text{preorder}$   
**shows** *ldistinct* (*lup'*  $x$   $xs$ )  
 $\langle \text{proof} \rangle$

**context** **fixes**  $f :: 'a \Rightarrow 'a$  **begin**

**partial-function** (*llist*) *iterate* ::  $'a \Rightarrow 'a \text{ llist}$   
**where** *iterate*  $x = LCons$   $x$  (*iterate* ( $f$   $x$ ))

**lemma** *lmap-iterate*: *lmap*  $f$  (*iterate*  $x$ ) = *iterate* ( $f$   $x$ )  
 $\langle \text{proof} \rangle$

**end**

**fun** *extup* *extdown* ::  $\text{int} \Rightarrow \text{int list} \Rightarrow \text{int list}$  **where**  
*extup*  $i$  [] = []  
| *extup*  $i$  ( $x \# xs$ ) = (if  $i \leq x$  then *extup*  $x$   $xs$  else  $i \#$  *extdown*  $x$   $xs$ )  
| *extdown*  $i$  [] = []  
| *extdown*  $i$  ( $x \# xs$ ) = (if  $i \geq x$  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*  $a$  ( $xs$  @  $ys$ ))  
 $\langle \text{proof} \rangle$

**lemma** *mono-ext*: **assumes**  $xs \leq ys$  **shows** *extup*  $a$   $xs \leq$  *extup*  $a$   $ys$  *extdown*  $a$   $xs$   
 $\leq$  *extdown*  $a$   $ys$   
 $\langle \text{proof} \rangle$

**lemma** *set-ext*: *set* (*extup*  $a$   $xs$ )  $\subseteq$   $\{a\} \cup \text{set } xs$  *set* (*extdown*  $a$   $xs$ )  $\subseteq$   $\{a\} \cup \text{set } xs$   
 $\langle \text{proof} \rangle$

**definition** *lxtup*  $i$   $l = Lim$  ( $at'$   $l$ ) (*llist-of*  $\circ$  *extup*  $i$   $\circ$  *list-of*)

**definition** *lxtdown*  $i$   $l = Lim$  ( $at'$   $l$ ) (*llist-of*  $\circ$  *extdown*  $i$   $\circ$  *list-of*)

**lemma** *tendsto-lxtup*[*tendsto-intros*]: (*lxtup*  $i$   $\longrightarrow$  *lxtup*  $i$   $xs$ ) ( $at'$   $xs$ )  
 $\langle \text{proof} \rangle$

**lemma** *tendsto-lxtdown*[*tendsto-intros*]: (*lxtdown*  $i$   $\longrightarrow$  *lxtdown*  $i$   $xs$ ) ( $at'$   $xs$ )  
 $\langle \text{proof} \rangle$

**lemma** *isCont-lxtup*[*THEN isCont-o2*[*rotated*]]: *isCont* (*lxtup*  $a$ )  $l$   
 $\langle \text{proof} \rangle$

**lemma** *isCont-lextdown*[*THEN isCont-o2*[*rotated*]]: *isCont* (*lextdown a*) *l*  
⟨*proof*⟩

**lemma** *lextup-lfinite*[*simp*]: *lfinite xs*  $\implies$  *lextup i xs* = *llist-of* (*extup i* (*list-of xs*))  
⟨*proof*⟩

**lemma** *lextdown-lfinite*[*simp*]: *lfinite xs*  $\implies$  *lextdown i xs* = *llist-of* (*extdown i* (*list-of xs*))  
⟨*proof*⟩

**lemma** *lextup i LNil* = *LNil* *lextdown i LNil* = *LNil*  
⟨*proof*⟩

**lemma** *lextup i (LCons x xs)* = (*if i*  $\leq$  *x* *then* *lextup x xs* *else* *LCons i* (*lextdown x xs*))  
⟨*proof*⟩

**lemma** *lextdown i (LCons x xs)* = (*if x*  $\leq$  *i* *then* *lextdown x xs* *else* *LCons i* (*lextup x xs*))  
⟨*proof*⟩

**lemma** *lset (lextup a xs)*  $\subseteq$   $\{a\} \cup$  *lset xs*  
⟨*proof*⟩

**lemma** *lset (lextdown a xs)*  $\subseteq$   $\{a\} \cup$  *lset xs*  
⟨*proof*⟩

**lemma** *distinct-ext*:  
**assumes** *distinct xs a*  $\notin$  *set xs*  
**shows** *distinct (extup a xs)* *distinct (extdown a xs)*  
⟨*proof*⟩

**lemma** *ldistinct xs*  $\implies$  *a*  $\notin$  *lset xs*  $\implies$  *ldistinct (lextup a xs)*  
⟨*proof*⟩

**definition** *esum-list* :: *ereal llist*  $\Rightarrow$  *ereal* **where**  
*esum-list xs* = *Lim (at' xs)* (*sum-list*  $\circ$  *list-of*)

**lemma** *esum-list-lfinite*[*simp*]: *lfinite xs*  $\implies$  *esum-list xs* = *sum-list* (*list-of xs*)  
⟨*proof*⟩

**lemma** *esum-list-LNil*: *esum-list LNil* = 0  
⟨*proof*⟩

**context**  
**fixes** *xs* :: *ereal llist*  
**assumes** *xs*:  $\bigwedge x. x \in$  *lset xs*  $\implies$   $0 \leq x$   
**begin**

**lemma** *esum-list-tendsto-SUP*:

$((\text{sum-list} \circ \text{list-of}) \longrightarrow (\text{SUP } ys \in \{ys. \text{lfinite } ys \wedge ys \leq xs\}. \text{esum-list } ys)) \text{ (at' } xs)$   
 $(\text{is } (- \longrightarrow ?y) -)$   
 $\langle \text{proof} \rangle$

**lemma** *tendsto-esum-list*:  $(\text{esum-list} \longrightarrow \text{esum-list } xs) \text{ (at' } xs)$   
 $\langle \text{proof} \rangle$

**lemma** *isCont-esum-list*:  $\text{isCont } \text{esum-list } xs$   
 $\langle \text{proof} \rangle$

**end**

**lemma** *esum-list-nonneg*:

$(\bigwedge x. x \in \text{lset } xs \implies 0 \leq x) \implies 0 \leq \text{esum-list } xs$   
 $\langle \text{proof} \rangle$

**lemma** *esum-list-LCons*:

**assumes**  $x: 0 \leq x \wedge \bigwedge x. x \in \text{lset } xs \implies 0 \leq x$  **shows**  $\text{esum-list } (\text{LCons } x \ xs) = x + \text{esum-list } xs$   
 $\langle \text{proof} \rangle$

**lemma** *esum-list-lfilter'*:

**assumes**  $nn: \bigwedge x. x \in \text{lset } xs \implies 0 \leq x$  **shows**  $\text{esum-list } (\text{lfilter}' (\lambda x. x \neq 0) \ xs) = \text{esum-list } xs$   
 $\langle \text{proof} \rangle$

**function**  $f:: \text{nat list} \Rightarrow \text{nat list}$  **where**

$f \ [] = []$   
 $| f \ (x \# xs) = (x * 2) \# f \ (f \ xs)$   
 $\langle \text{proof} \rangle$

**termination**  $f$

$\langle \text{proof} \rangle$

**lemma** *length-f[simp]*:  $\text{length } (f \ xs) = \text{length } xs$

$\langle \text{proof} \rangle$

**lemma** *f-mono'*:  $\exists ys'. f \ (xs \ @ \ ys) = f \ xs \ @ \ ys'$

$\langle \text{proof} \rangle$

**lemma** *f-mono*:  $xs \leq ys \implies f \ xs \leq f \ ys$

$\langle \text{proof} \rangle$

**definition**  $f' \ l = \text{Lim } (\text{at' } l) (\lambda l. \text{llist-of } (f \ (\text{list-of } l)))$

**lemma** *f'-lfinite[simp]*:  $\text{lfinite } xs \implies f' \ xs = \text{llist-of } (f \ (\text{list-of } xs))$

*<proof>*

**lemma** *tendsto-f'*:  $(f' \longrightarrow f' l) (at' l)$   
*<proof>*

**lemma** *isCont-f'*[*THEN isCont-o2[rotated]*]: *isCont f' l*  
*<proof>*

**lemma** *f' LNil = LNil*  
*<proof>*

**lemma** *f' (LCons x xs) = LCons (x \* 2) (f' (f' xs))*  
*<proof>*

**end**

## 12 Ccpo structure for terminated lazy lists

**theory** *TLList-CCPO* **imports** *TLList* **begin**

**lemma** *Set-is-empty-parametric* [*transfer-rule*]:  
**includes** *lifting-syntax*  
**shows**  $(rel\text{-set } A \text{ ===== } (=)) \text{ Set.is-empty Set.is-empty}$   
*<proof>*

**lemma** *monotone-comp*:  $\llbracket \text{monotone } orda \text{ ordb } g; \text{monotone } ordb \text{ ordc } f \rrbracket \implies$   
*monotone orda ordc (f o g)*  
*<proof>*

**lemma** *cont-comp*:  $\llbracket \text{mcont } luba \text{ orda } lubb \text{ ordb } g; \text{cont } lubb \text{ ordb } lubc \text{ ordc } f \rrbracket \implies$   
*cont luba orda lubc ordc (f o g)*  
*<proof>*

**lemma** *mcont-comp*:  $\llbracket \text{mcont } luba \text{ orda } lubb \text{ ordb } g; \text{mcont } lubb \text{ ordb } lubc \text{ ordc } f \rrbracket$   
 $\implies \text{mcont } luba \text{ orda } lubc \text{ ordc (f o g)}$   
*<proof>*

**context** **includes** *lifting-syntax*  
**begin**

**lemma** *monotone-parametric* [*transfer-rule*]:  
**assumes** [*transfer-rule*]: *bi-total A*  
**shows**  $((A \text{ ===== } A \text{ ===== } (=)) \text{ ===== } (B \text{ ===== } B \text{ ===== } (=)) \text{ ===== } (A$   
 $\text{ ===== } B) \text{ ===== } (=)) \text{ monotone monotone}$   
*<proof>*

**lemma** *cont-parametric* [*transfer-rule*]:  
**assumes** [*transfer-rule*]: *bi-total A bi-unique B*  
**shows**  $((rel\text{-set } A \text{ ===== } A) \text{ ===== } (A \text{ ===== } A \text{ ===== } (=)) \text{ ===== } (rel\text{-set } B$



====> B) ====> (B ====> B ====> (=)) ====> (A ====> B) ====> (=))  
 cont cont  
 <proof>

**lemma** *mcont-parametric* [*transfer-rule*]:  
**assumes** [*transfer-rule*]: *bi-total A bi-unique B*  
**shows** ((*rel-set A* ====> A) ====> (A ====> A ====> (=)) ====> (*rel-set B*  
 ====> B) ====> (B ====> B ====> (=)) ====> (A ====> B) ====> (=))  
 mcont mcont  
 <proof>

**end**

**lemma** (in *ccpo*) *Sup-Un-less*:  
**assumes** *chain*: *Complete-Partial-Order.chain* ( $\leq$ ) ( $A \cup B$ )  
**and** *AB*:  $\forall x \in A. \exists y \in B. x \leq y$   
**shows** *Sup* ( $A \cup B$ ) = *Sup B*  
 <proof>

## 12.1 The ccpo structure

**context includes** *tllist.lifting* **fixes** *b* :: 'b **begin**

**lift-definition** *tllist-ord* :: ('a, 'b) *tllist*  $\Rightarrow$  ('a, 'b) *tllist*  $\Rightarrow$  *bool*  
**is**  $\lambda(xs1, b1) (xs2, b2). \text{if } \text{lfinite } xs1 \text{ then } b1 = b \wedge \text{lprefix } xs1 \text{ } xs2 \vee xs1 = xs2 \wedge$   
*flat-ord* *b* *b1* *b2* **else**  $xs1 = xs2$   
 <proof>

**lift-definition** *tSup* :: ('a, 'b) *tllist set*  $\Rightarrow$  ('a, 'b) *tllist*  
**is**  $\lambda A. (\text{lSup } (\text{fst } ' A), \text{flat-lub } b (\text{snd } ' (A \cap \{(xs, -). \text{lfinite } xs\})))$   
 <proof>

**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*  
 <proof>

**lemma** *tllist-ord-refl* [*simp*]: *tllist-ord* *xs* *xs*  
 <proof>

**lemma** *tllist-ord-antisym*:  $\llbracket \text{tllist-ord } xs \text{ } ys; \text{tllist-ord } ys \text{ } xs \rrbracket \Longrightarrow xs = ys$   
 <proof>

**lemma** *tllist-ord-trans* [*trans*]:  $\llbracket \text{tllist-ord } xs \text{ } ys; \text{tllist-ord } ys \text{ } zs \rrbracket \Longrightarrow \text{tllist-ord } xs$   
*zs*  
 <proof>

**lemma** *chain-tllist-llist-of-tllist*:

**assumes** *Complete-Partial-Order.chain tllist-ord A*

**shows** *Complete-Partial-Order.chain lprefix (llist-of-tllist ‘ A)*

*<proof>*

**lemma** *chain-tllist-terminal*:

**assumes** *Complete-Partial-Order.chain tllist-ord A*

**shows** *Complete-Partial-Order.chain (flat-ord b) {terminal xs|xs. xs ∈ A ∧ tfinite xs}*

*<proof>*

**lemma** *flat-ord-chain-finite*:

**assumes** *Complete-Partial-Order.chain (flat-ord b) A*

**shows** *finite A*

*<proof>*

**lemma** *tSup-empty [simp]: tSup {} = TNil b*

*<proof>*

**lemma** *is-TNil-tSup [simp]: is-TNil (tSup A) ⟷ (∀ x∈A. is-TNil x)*

*<proof>*

**lemma** *chain-tllist-ord-tSup*:

**assumes** *chain: Complete-Partial-Order.chain tllist-ord A*

**and** *A: xs ∈ A*

**shows** *tllist-ord xs (tSup A)*

*<proof>*

**lemma** *chain-tSup-tllist-ord*:

**assumes** *chain: Complete-Partial-Order.chain tllist-ord A*

**and** *lub: ∧xs'. xs' ∈ A ⟹ tllist-ord xs' xs*

**shows** *tllist-ord (tSup A) xs*

*<proof>*

**lemma** *tllist-ord-ccpo [simp, cont-intro]*:

*class.ccpo tSup tllist-ord (mk-less tllist-ord)*

*<proof>*

**lemma** *tllist-ord-partial-function-definitions: partial-function-definitions tllist-ord tSup*

*<proof>*

**interpretation** *tllist: partial-function-definitions tllist-ord tSup*

*<proof>*

**lemma** *admissible-mcont-is-TNil [THEN admissible-subst, cont-intro, simp]*:

**shows** *admissible-is-TNil: ccpo.admissible tSup tllist-ord is-TNil*

*<proof>*

**lemma** *terminal-tSup*:

$\forall xs \in Y. is-TNil\ xs \implies terminal\ (tSup\ Y) = flat-lub\ b\ (terminal\ 'Y)$   
**including** *tlist.lifting*  $\langle proof \rangle$

**lemma** *thd-tSup*:

$\exists xs \in Y. \neg is-TNil\ xs$   
 $\implies thd\ (tSup\ Y) = (THE\ x. x \in thd\ ' (Y \cap \{xs. \neg is-TNil\ xs\}))$   
 $\langle proof \rangle$

**lemma** *ex-TCons-raw-parametric*:

**includes** *lifting-syntax*  
**shows**  $(rel-set\ (rel-prod\ (llist-all2\ A)\ B) \implies\ (=))\ (\lambda Y. \exists (xs, b) \in Y. \neg lnull\ xs)$   
 $(\lambda Y. \exists (xs, b) \in Y. \neg lnull\ xs)$   
 $\langle proof \rangle$

**lift-definition** *ex-TCons* ::  $('a, 'b)\ tlist\ set \Rightarrow bool$

**is**  $\lambda Y. \exists (xs, b) \in Y. \neg lnull\ xs$  **parametric** *ex-TCons-raw-parametric*  
 $\langle proof \rangle$

**lemma** *ex-TCons-iff*:  $ex-TCons\ Y \longleftrightarrow (\exists xs \in Y. \neg is-TNil\ xs)$   
 $\langle proof \rangle$

**lemma** *retain-TCons-raw-parametric*:

**includes** *lifting-syntax*  
**shows**  $(rel-set\ (rel-prod\ (llist-all2\ A)\ B) \implies\ rel-set\ (rel-prod\ (llist-all2\ A)\ B))$   
 $(\lambda A. A \cap \{(xs, b). \neg lnull\ xs\})\ (\lambda A. A \cap \{(xs, b). \neg lnull\ xs\})$   
 $\langle proof \rangle$

**lift-definition** *retain-TCons* ::  $('a, 'b)\ tlist\ set \Rightarrow ('a, 'b)\ tlist\ set$

**is**  $\lambda A. A \cap \{(xs, b). \neg lnull\ xs\}$  **parametric** *retain-TCons-raw-parametric*  
 $\langle proof \rangle$

**lemma** *retain-TCons-conv*:  $retain-TCons\ A = A \cap \{xs. \neg is-TNil\ xs\}$   
 $\langle proof \rangle$

**lemma** *tll-tSup*:

$\llbracket Complete-Partial-Order.chain\ tllist-ord\ Y; \exists xs \in Y. \neg is-TNil\ xs \rrbracket$   
 $\implies tll\ (tSup\ Y) = tSup\ (tll\ ' (Y \cap \{xs. \neg is-TNil\ xs\}))$   
 $\langle proof \rangle$

**lemma** *tSup-TCons*:  $A \neq \{\}$   $\implies tSup\ (TCons\ x\ 'A) = TCons\ x\ (tSup\ A)$   
 $\langle proof \rangle$

**lemma** *tllist-ord-terminalD*:

$\llbracket tllist-ord\ xs\ ys; is-TNil\ ys \rrbracket \implies flat-ord\ b\ (terminal\ xs)\ (terminal\ ys)$   
 $\langle proof \rangle$

**lemma** *tllist-ord-bot* [*simp*]: *tllist-ord* (TNil b) *xs*  
 ⟨*proof*⟩

**lemma** *tllist-ord-ttlI*:  
*tllist-ord xs ys*  $\implies$  *tllist-ord* (ttl *xs*) (ttl *ys*)  
 ⟨*proof*⟩

**lemma** *not-is-TNil-conv*:  $\neg$  *is-TNil xs*  $\longleftrightarrow$  ( $\exists x xs'$ . *xs* = TCons *x xs'*)  
 ⟨*proof*⟩

## 12.2 Continuity of predefined constants

**lemma** *mono-tllist-ord-case*:  
 fixes *bot*  
 assumes *mono*:  $\bigwedge x$ . *monotone tllist-ord ord* ( $\lambda xs$ . *f x xs* (TCons *x xs*))  
 and *ord*: *class.preorder ord* (*mk-less ord*)  
 and *bot*:  $\bigwedge x$ . *ord* (*bot b*) *x*  
 shows *monotone tllist-ord ord* ( $\lambda xs$ . *case xs of* TNil *b*  $\implies$  *bot b* | TCons *x xs'*  $\implies$   
*f x xs' xs*)  
 ⟨*proof*⟩

**lemma** *mcont-lprefix-case-aux*:  
 fixes *f bot ord*  
 defines *g*  $\equiv$   $\lambda xs$ . *f* (*thd xs*) (ttl *xs*) (TCons (*thd xs*) (ttl *xs*))  
 assumes *mcont*:  $\bigwedge x$ . *mcont tSup tllist-ord lub ord* ( $\lambda xs$ . *f x xs* (TCons *x xs*))  
 and *ccpo*: *class.ccpo lub ord* (*mk-less ord*)  
 and *bot*:  $\bigwedge x$ . *ord* (*bot b*) *x*  
 and *cont-bot*: *cont* (*flat-lub b*) (*flat-ord b*) *lub ord bot*  
 shows *mcont tSup tllist-ord lub ord* ( $\lambda xs$ . *case xs of* TNil *b*  $\implies$  *bot b* | TCons *x*  
*xs'*  $\implies$  *f x xs' xs*)  
 (is *mcont* - - - ?*f*)  
 ⟨*proof*⟩

**lemma** *cont-TNil* [*simp*, *cont-intro*]: *cont* (*flat-lub b*) (*flat-ord b*) *tSup tllist-ord*  
 TNil  
 ⟨*proof*⟩

**lemma** *monotone-TCons*: *monotone tllist-ord tllist-ord* (TCons *x*)  
 ⟨*proof*⟩

**lemmas** *mono2mono-TCons*[*cont-intro*] = *monotone-TCons*[THEN *tllist.mono2mono*]

**lemma** *mcont-TCons*: *mcont tSup tllist-ord tSup tllist-ord* (TCons *x*)  
 ⟨*proof*⟩

**lemmas** *mcont2mcont-TCons*[*cont-intro*] = *mcont-TCons*[THEN *tllist.mcont2mcont*]

**lemmas** [*transfer-rule del*] = *tllist-ord.transfer tSup.transfer*

**lifting-update** *tllist.lifting*  
**lifting-forget** *tllist.lifting*

**lemmas** [*transfer-rule*] = *tllist-ord.transfer tSup.transfer*

**lemma** *mono2mono-tset* [*THEN lfp.mono2mono, cont-intro*]:  
**shows** *smonotone-tset: monotone tllist-ord ( $\subseteq$ ) tset*  
**including** *tllist.lifting*  
 $\langle$ *proof* $\rangle$

**lemma** *mcont2mcont-tset* [*THEN lfp.mcont2mcont, cont-intro*]:  
**shows** *mcont-tset: mcont tSup tllist-ord Union ( $\subseteq$ ) tset*  
**including** *tllist.lifting*  
 $\langle$ *proof* $\rangle$

**end**

**context includes** *lifting-syntax*  
**begin**

**lemma** *rel-fun-lift*:  
 $(\bigwedge x. A (f x) (g x)) \implies ((=) \implies A) f g$   
 $\langle$ *proof* $\rangle$

**lemma** *tllist-ord-transfer* [*transfer-rule*]:  
 $((=) \implies \text{pcr-tllist } (=) (=) \implies \text{pcr-tllist } (=) (=) \implies (=))$   
 $(\lambda b (xs1, b1) (xs2, b2). \text{if } \text{lfinite } xs1 \text{ then } b1 = b \wedge \text{lprefix } xs1 \text{ } xs2 \vee xs1 =$   
 $xs2 \wedge \text{flat-ord } b \text{ } b1 \text{ } b2 \text{ else } xs1 = xs2)$   
*tllist-ord*  
 $\langle$ *proof* $\rangle$

**lemma** *tSup-transfer* [*transfer-rule*]:  
 $((=) \implies \text{rel-set } (\text{pcr-tllist } (=) (=)) \implies \text{pcr-tllist } (=) (=))$   
 $(\lambda b A. (\text{lSup } (\text{fst } 'A), \text{flat-lub } b (\text{snd } '(A \cap \{(xs, -). \text{lfinite } xs\}))))$   
*tSup*  
 $\langle$ *proof* $\rangle$

**end**

**lifting-update** *tllist.lifting*  
**lifting-forget** *tllist.lifting*

**interpretation** *tllist: partial-function-definitions tllist-ord b tSup b for b*  
 $\langle$ *proof* $\rangle$

**lemma** *tllist-case-mono* [*partial-function-mono, cont-intro*]:  
**assumes** *tnil:  $\bigwedge b. \text{monotone } orda \text{ } ordb (\lambda f. \text{tnil } f \text{ } b)$*   
**and** *tcons:  $\bigwedge x \text{ } xs. \text{monotone } orda \text{ } ordb (\lambda f. \text{tcons } f \text{ } x \text{ } xs)$*   
**shows** *monotone orda ordb ( $\lambda f. \text{case-tllist } (\text{tnil } f) (\text{tcons } f) \text{ } xs)$*

*<proof>*

### 12.3 Definition of recursive functions

**locale** *tllist-pf* = **fixes** *b* :: 'b  
**begin**

*<ML>*

**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* ( $\lambda f. TCons\ x\ (A\ f)$ )  
*<proof>*

**end**

**lemma** *mono-tllist-lappendt2* [*partial-function-mono*]:  
*tllist-pf.mono-tllist* *b* *A*  $\implies$  *tllist-pf.mono-tllist* *b* ( $\lambda f. lappendt\ xs\ (A\ f)$ )  
*<proof>*

**lemma** *mono-tllist-tappend2* [*partial-function-mono*]:  
**assumes**  $\bigwedge y. tllist-pf.mono-tllist\ b\ (C\ y)$   
**shows** *tllist-pf.mono-tllist* *b* ( $\lambda f. tappend\ xs\ (\lambda y. C\ y\ f)$ )  
*<proof>*  
**including** *tllist.lifting*  
*<proof>*

**end**

## 13 Example definitions using the CCPO structure on terminated lazy lists

**theory** *TLList-CCPO-Examples* **imports**  
*../TLList-CCPO*  
**begin**

**context** **fixes** *b* :: 'b **begin**  
**interpretation** *tllist-pf* *b* *<proof>*

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

**partial-function** (*tllist*) *tfilter* :: ('a, 'b) *tllist*  $\Rightarrow$  ('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)  $\longleftrightarrow$  ( $\forall x \in tset\ xs. \neg P\ x$ ) (is ?lhs  $\longleftrightarrow$  ?rhs)*  
<proof>

**end**

**lemma** *mcont2mcont-tfilter*[*THEN tlist.mcont2mcont, simp, cont-intro*]:  
shows *mcont-tfilter*: *mcont (tSup b) (tlist-ord b) (tSup b) (tlist-ord b) (tfilter b P)*  
<proof>

**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>

**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>

**context** *fixes b :: 'b* **begin**

**interpretation** *tlist-pf b* <proof>

**partial-function** (*tlist*) *tconcat* :: (*'a llist, 'b*) *tlist*  $\Rightarrow$  (*'a, 'b*) *tlist*  
**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*:  $LNil \in \text{paths graph}$   
 | *Single*:  $LCons\ x\ LNil \in \text{paths graph}$   
 | *LCons*:  $\llbracket \text{graph } x\ y; LCons\ y\ xs \in \text{paths graph} \rrbracket \Longrightarrow LCons\ x\ (LCons\ y\ xs) \in \text{paths graph}$

**definition** *connected* :: 'node graph  $\Rightarrow$  bool  
**where** *connected graph*  $\longleftrightarrow (\forall n\ n'. \exists xs. \text{lset-of } (n \# xs @ [n']) \in \text{paths graph})$

**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**  $\llbracket LCons\ n\ xs \in \text{paths graph}; n' \in \text{lset } xs; \text{lset } xs \subseteq ns \rrbracket \Longrightarrow n' \in \text{reachable-via graph ns n}$

**lemma** *connectedD*: *connected graph*  $\Longrightarrow \exists xs. \text{lset-of } (n \# xs @ [n']) \in \text{paths graph}$   
 <proof>

**lemma** *paths-LConsD*:  
**assumes**  $LCons\ x\ xs \in \text{paths graph}$   
**shows**  $x \in \text{paths graph}$   
 <proof>

**lemma** *paths-lappendD1*:  
**assumes**  $\text{lappend } xs\ ys \in \text{paths graph}$   
**shows**  $x \in \text{paths graph}$   
 <proof>

**lemma** *paths-lappendD2*:  
**assumes**  $\text{lfinite } xs$   
**and**  $\text{lappend } xs\ ys \in \text{paths graph}$   
**shows**  $ys \in \text{paths graph}$   
 <proof>

**lemma** *path-avoid-node*:  
**assumes** *path*:  $LCons\ n\ xs \in \text{paths graph}$   
**and** *set*:  $x \in \text{lset } xs$   
**and** *n-neq-x*:  $n \neq x$   
**shows**  $\exists xs'. LCons\ n\ xs' \in \text{paths graph} \wedge \text{lset } xs' \subseteq \text{lset } xs \wedge x \in \text{lset } xs' \wedge n \notin \text{lset } xs'$   
 <proof>

**lemma** *reachable-via-subset-unfold*:  
*reachable-via graph ns n*  $\subseteq (\bigcup n' \in \{n'. \text{graph } n\ n'\} \cap ns. \text{insert } n' (\text{reachable-via graph } (ns - \{n'\})\ n'))$   
 (is ?lhs  $\subseteq$  ?rhs)



*<proof>*

**theorem** *koenigslemma*:

**fixes** *graph* :: 'node graph

**and** *n* :: 'node

**assumes** *connected*: connected graph

**and** *infinite*: infinite (UNIV :: 'node set)

**and** *finite-branching*:  $\bigwedge n. \text{finite } \{n'. \text{graph } n \ n'\}$

**shows**  $\exists xs \in \text{paths graph}. n \in \text{lset } xs \wedge \neg \text{lfinite } xs \wedge \text{ldistinct } xs$

*<proof>*

**end**

## 15 Definition of the function *lmirror*

**theory** *LMirror* **imports** *../Coinductive-List* **begin**

This theory defines a function *lmirror*.

**primcorec** *lmirror-aux* :: 'a llist  $\Rightarrow$  '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)

*<proof>*

**lemma** *ltl-lmirror-aux*:

*ltl* (*lmirror-aux* acc xs) = (if *lnull* xs then *ltl* acc else *lmirror-aux* (LCons (lhd xs) acc) (*ltl* xs))

*<proof>*

**lemma** *lhd-lmirror-aux*:

*lhd* (*lmirror-aux* acc xs) = (if *lnull* xs then *lhd* acc else *lhd* xs)

*<proof>*

**declare** *lmirror-aux.sel*[*simp del*]

**lemma** *lfinite-lmirror-aux* [*simp*]:

*lfinite* (*lmirror-aux* acc xs)  $\longleftrightarrow$  *lfinite* xs  $\wedge$  *lfinite* acc

(**is** ?lhs  $\longleftrightarrow$  ?rhs)

*<proof>*

**lemma** *lmirror-aux-inf*:

$\neg \text{lfinite } xs \implies \text{lmirror-aux acc } xs = xs$

*<proof>*

**lemma** *lmirror-aux-acc*:

$\text{lmirror-aux (lappend } ys \ zs) \ xs = \text{lappend (lmirror-aux } ys \ xs) \ zs$

*<proof>*

**lemma** *lmirror-aux-LCons*:

$\text{lmirror-aux acc (LCons } x \ xs) = \text{LCons } x \ (\text{lappend (lmirror-aux LNil } xs) \ (\text{LCons } x \ \text{acc}))$

*<proof>*

**lemma** *llength-lmirror-aux*:  $\text{llength (lmirror-aux acc } xs) = 2 * \text{llength } xs + \text{llength acc}$

*<proof>*

**lemma** *lnull-lmirror [simp]*:  $\text{lnull (lmirror } xs) = \text{lnull } xs$

*<proof>*

**lemma** *lmirror-LNil [simp]*:  $\text{lmirror LNil} = \text{LNil}$

*<proof>*

**lemma** *lmirror-LCons [simp]*:  $\text{lmirror (LCons } x \ xs) = \text{LCons } x \ (\text{lappend (lmirror } xs) \ (\text{LCons } x \ \text{LNil}))$

*<proof>*

**lemma** *ltl-lmirror [simp]*:

$\neg \text{lnull } xs \implies \text{ltl (lmirror } xs) = \text{lappend (lmirror (ltl } xs)) \ (\text{LCons (lhd } xs) \ \text{LNil})$

*<proof>*

**lemma** *lmap-lmirror-aux*:  $\text{lmap } f \ (\text{lmirror-aux acc } xs) = \text{lmirror-aux (lmap } f \ \text{acc}) \ (\text{lmap } f \ xs)$

*<proof>*

**lemma** *lmap-lmirror*:  $\text{lmap } f \ (\text{lmirror } xs) = \text{lmirror (lmap } f \ xs)$

*<proof>*

**lemma** *lset-lmirror-aux*:  $\text{lset (lmirror-aux acc } xs) = \text{lset (lappend } xs \ \text{acc})$

*<proof>*

**lemma** *lset-lmirror [simp]*:  $\text{lset (lmirror } xs) = \text{lset } xs$

*<proof>*

**lemma** *llength-lmirror [simp]*:  $\text{llength (lmirror } xs) = 2 * \text{llength } xs$

*<proof>*

**lemma** *lmirror-llist-of [simp]*:  $\text{lmirror (llist-of } xs) = \text{llist-of } (xs \ @ \ \text{rev } xs)$

*<proof>*

**lemma** *list-of-lmirror* [*simp*]:  $lfinite\ xs \implies list-of\ (lmirror\ xs) = list-of\ xs\ @\ rev\ (list-of\ xs)$   
 <proof>

**lemma** *llist-all2-lmirror-aux*:  
 $\llbracket llist-all2\ P\ acc\ acc';\ llist-all2\ P\ xs\ xs' \rrbracket$   
 $\implies llist-all2\ P\ (lmirror-aux\ acc\ xs)\ (lmirror-aux\ acc'\ xs')$   
 <proof>

**lemma** *enat-mult-cancel1* [*simp*]:  
 $k * m = k * n \iff m = n \vee k = 0 \vee k = (\infty :: enat) \wedge n \neq 0 \wedge m \neq 0$   
 <proof>

**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>

**lemma** *llist-all2-lmirrorI*:  
 $llist-all2\ P\ xs\ ys \implies llist-all2\ P\ (lmirror\ xs)\ (lmirror\ ys)$   
 <proof>

**lemma** *llist-all2-lmirrorD*:  
 $llist-all2\ P\ (lmirror\ xs)\ (lmirror\ ys) \implies llist-all2\ P\ xs\ ys$   
 <proof>

**lemma** *llist-all2-lmirror* [*simp*]:  
 $llist-all2\ P\ (lmirror\ xs)\ (lmirror\ ys) \iff llist-all2\ P\ xs\ ys$   
 <proof>

**lemma** *lmirror-parametric* [*transfer-rule*]:  
**includes** *lifting-syntax*  
**shows**  $(llist-all2\ A \implies llist-all2\ A)\ lmirror\ lmirror$   
 <proof>

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**  $n :: \text{enat}$   
**shows**  $\infty \sqcap n = n \sqcap \infty = n$   
 $\langle \text{proof} \rangle$

**lemma**  $e\text{Suc-inf-eSuc}$  [*simp*]:  $e\text{Suc } n \sqcap e\text{Suc } m = e\text{Suc } (n \sqcap m)$   
 $\langle \text{proof} \rangle$

**lemma**  $if\text{-pull2}$ :  $(if\ b\ then\ f\ x\ x'\ else\ f\ y\ y') = f\ (if\ b\ then\ x\ else\ y)\ (if\ b\ then\ x'\ else\ y')$   
 $\langle \text{proof} \rangle$

**context**  $ord$  **begin**

**primcorec**  $lmerge :: 'a\ list \Rightarrow 'a\ llist \Rightarrow 'a\ llist$   
**where**

$lmerge\ xs\ ys =$   
 $(case\ xs\ of\ LNil \Rightarrow LNil \mid LCons\ x\ xs' \Rightarrow$   
 $case\ ys\ of\ LNil \Rightarrow LNil \mid LCons\ y\ ys' \Rightarrow$   
 $if\ lhd\ xs < lhd\ ys\ then\ LCons\ x\ (lmerge\ xs'\ ys)$   
 $else\ LCons\ y\ (if\ lhd\ ys < lhd\ xs\ then\ lmerge\ xs\ ys'\ else\ lmerge\ xs'\ ys'))$

**lemma**  $lnull\text{-lmerge}$  [*simp*]:  $lnull\ (lmerge\ xs\ ys) \longleftrightarrow (lnull\ xs \vee lnull\ ys)$   
 $\langle \text{proof} \rangle$

**lemma**  $lmerge\text{-eq-LNil-iff}$ :  $lmerge\ xs\ ys = LNil \longleftrightarrow (xs = LNil \vee ys = LNil)$   
 $\langle \text{proof} \rangle$

**lemma**  $lhd\text{-lmerge}$ :  $\llbracket \neg\ lnull\ xs; \neg\ lnull\ ys \rrbracket \Longrightarrow lhd\ (lmerge\ xs\ ys) = (if\ lhd\ xs < lhd\ ys\ then\ lhd\ xs\ else\ lhd\ ys)$   
 $\langle \text{proof} \rangle$

**lemma**  $ltl\text{-lmerge}$ :  
 $\llbracket \neg\ lnull\ xs; \neg\ lnull\ ys \rrbracket \Longrightarrow$   
 $ltl\ (lmerge\ xs\ ys) =$   
 $(if\ lhd\ xs < lhd\ ys\ then\ lmerge\ (ltl\ xs)\ ys$   
 $else\ if\ lhd\ ys < lhd\ xs\ then\ lmerge\ xs\ (ltl\ ys)$   
 $else\ lmerge\ (ltl\ xs)\ (ltl\ ys))$   
 $\langle \text{proof} \rangle$

**declare**  $lmerge.sel$  [*simp del*]

**lemma**  $lmerge\text{-simps}$ :  
 $lmerge\ (LCons\ x\ xs)\ (LCons\ y\ ys) =$   
 $(if\ x < y\ then\ LCons\ x\ (lmerge\ xs\ (LCons\ y\ ys))$   
 $else\ if\ y < x\ then\ LCons\ y\ (lmerge\ (LCons\ x\ xs)\ ys)$   
 $else\ LCons\ y\ (lmerge\ xs\ ys))$   
 $\langle \text{proof} \rangle$

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

$lmerge\ LNil\ ys = LNil$

$lmerge\ xs\ LNil = LNil$

$\langle proof \rangle$

**lemma** *lprefix-lmergeI*:

$\llbracket lprefix\ xs\ xs';\ lprefix\ ys\ ys' \rrbracket$

$\implies lprefix\ (lmerge\ xs\ ys)\ (lmerge\ xs'\ ys')$

$\langle proof \rangle$

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

**assumes**  $F$ : *mono-llist*  $F$  **and**  $G$ : *mono-llist*  $G$

**shows** *mono-llist*  $(\lambda f. lmerge\ (F\ f)\ (G\ f))$

$\langle proof \rangle$

**lemma** *in-lset-lmergeD*:  $x \in lset\ (lmerge\ xs\ ys) \implies x \in lset\ xs \vee x \in lset\ ys$

$\langle proof \rangle$

**lemma** *lset-lmerge*:  $lset\ (lmerge\ xs\ ys) \subseteq lset\ xs \cup lset\ ys$

$\langle proof \rangle$

**lemma** *lfinite-lmergeD*:  $lfinite\ (lmerge\ xs\ ys) \implies lfinite\ xs \vee lfinite\ ys$

$\langle proof \rangle$

**lemma** *fixes F*

**defines**  $F \equiv \lambda lmerge\ (xs, ys). case\ xs\ of\ LNil \Rightarrow LNil \mid LCons\ x\ xs' \Rightarrow case\ ys$   
*of*  $LNil \Rightarrow LNil \mid LCons\ y\ ys' \Rightarrow (if\ x < y\ then\ LCons\ x\ (curry\ lmerge\ xs'\ ys)\ else$   
*if*  $y < x\ then\ LCons\ y\ (curry\ lmerge\ xs\ ys')\ else\ LCons\ y\ (curry\ lmerge\ xs'\ ys'))$

**shows** *lmerge-conv-fixp*:  $lmerge \equiv curry\ (ccpo.fixp\ (fun-lub\ lSup)\ (fun-ord\ lprefix))$   
 $F$  (**is**  $?lhs \equiv ?rhs$ )

**and** *lmerge-mono*: *mono-llist*  $(\lambda lmerge. F\ lmerge\ xs)$  (**is**  $?mono\ xs$ )

$\langle proof \rangle$

**lemma** *monotone-lmerge*: *monotone*  $(rel-prod\ lprefix\ lprefix)\ lprefix\ (case-prod\ lmerge)$

$\langle proof \rangle$

**lemma** *mono2mono-lmerge1* [*THEN* *llist.mono2mono*, *cont-intro*, *simp*]:

**shows** *monotone-lmerge1*: *monotone*  $lprefix\ lprefix\ (\lambda xs. lmerge\ xs\ ys)$

$\langle proof \rangle$

**lemma** *mono2mono-lmerge2* [*THEN* *llist.mono2mono*, *cont-intro*, *simp*]:

**shows** *monotone-lmerge2*: *monotone*  $lprefix\ lprefix\ (\lambda ys. lmerge\ xs\ ys)$

$\langle proof \rangle$

**lemma** *mcont-lmerge*: *mcont*  $(prod-lub\ lSup\ lSup)\ (rel-prod\ lprefix\ lprefix)\ lSup$   
 $lprefix\ (case-prod\ lmerge)$

$\langle proof \rangle$

**lemma** *mcont2mcont-lmerge1* [*THEN* *llist.mcont2mcont*, *cont-intro*, *simp*]:

**shows** *mcont-lmerge1*: *mcont lSup lprefix lSup lprefix* ( $\lambda x.s. \text{lmerge } x s \text{ } y s$ )  
<proof>

**lemma** *mcont2mcont-lmerge2* [*THEN llist.mcont2mcont, cont-intro, simp*]:  
**shows** *mcont-lmerge2*: *mcont lSup lprefix lSup lprefix* ( $\lambda y.s. \text{lmerge } x s \text{ } y s$ )  
<proof>

**lemma** *lfinite-lmergeI* [*simp*]:  $\llbracket \text{lfinite } x s; \text{lfinite } y s \rrbracket \implies \text{lfinite } (\text{lmerge } x s \text{ } y s)$   
<proof>

**lemma** *linfinite-lmerge* [*simp*]:  $\llbracket \neg \text{lfinite } x s; \neg \text{lfinite } y s \rrbracket \implies \neg \text{lfinite } (\text{lmerge } x s \text{ } y s)$   
<proof>

**lemma** *llength-lmerge-above*:  $\text{llength } x s \sqcap \text{llength } y s \leq \text{llength } (\text{lmerge } x s \text{ } y s)$   
<proof>

**end**

**context** *linorder begin*

**lemma** *in-lset-lmergeI1*:  
 $\llbracket x \in \text{lset } x s; \text{lsorted } x s; \neg \text{lfinite } y s; \exists y \in \text{lset } y s. x \leq y \rrbracket$   
 $\implies x \in \text{lset } (\text{lmerge } x s \text{ } y s)$   
<proof>

**lemma** *in-lset-lmergeI2*:  
 $\llbracket x \in \text{lset } y s; \text{lsorted } y s; \neg \text{lfinite } x s; \exists y \in \text{lset } x s. x \leq y \rrbracket$   
 $\implies x \in \text{lset } (\text{lmerge } x s \text{ } y s)$   
<proof>

**lemma** *lsorted-lmerge*:  $\llbracket \text{lsorted } x s; \text{lsorted } y s \rrbracket \implies \text{lsorted } (\text{lmerge } x s \text{ } y s)$   
<proof>

**lemma** *ldistinct-lmerge*:  
 $\llbracket \text{lsorted } x s; \text{lsorted } y s; \text{ldistinct } x s; \text{ldistinct } y s \rrbracket$   
 $\implies \text{ldistinct } (\text{lmerge } x s \text{ } y s)$   
<proof>

**end**

**partial-function** (*llist*) *hamming'* :: *unit*  $\Rightarrow$  *nat llist*

**where**

*hamming'* - =  
*LCons* 1 (*lmerge* (*lmap* ((\* 2) (*hamming'* ())) (*lmerge* (*lmap* ((\* 3) (*hamming'* ())) (*lmap* ((\* 5) (*hamming'* ())))))

**definition** *hamming* :: *nat llist*

**where**  $hamming = hamming' ()$

**lemma**  $lnull-hamming$  [simp]:  $\neg lnull\ hamming$   
<proof>

**lemma**  $hamming-eq-LNil-iff$  [simp]:  $hamming = LNil \longleftrightarrow False$   
<proof>

**lemma**  $lhd-hamming$  [simp]:  $lhd\ hamming = 1$   
<proof>

**lemma**  $ltl-hamming$  [simp]:  
 $ltl\ hamming = lmerge\ (lmap\ ((*)\ 2)\ hamming)\ (lmerge\ (lmap\ ((*)\ 3)\ hamming)\ (lmap\ ((*)\ 5)\ hamming))$   
<proof>

**lemma**  $hamming-unfold$ :  
 $hamming = LCons\ 1\ (lmerge\ (lmap\ ((*)\ 2)\ hamming)\ (lmerge\ (lmap\ ((*)\ 3)\ hamming)\ (lmap\ ((*)\ 5)\ hamming)))$   
<proof>

**definition**  $smooth :: nat \Rightarrow bool$   
**where**  $smooth\ n \longleftrightarrow (\forall p.\ prime\ p \longrightarrow p\ dvd\ n \longrightarrow p \leq 5)$

**lemma**  $smooth-0$  [simp]:  $\neg smooth\ 0$   
<proof>

**lemma**  $smooth-Suc0$  [simp]:  $smooth\ (Suc\ 0)$   
<proof>

**lemma**  $smooth-gt0$ :  $smooth\ n \Longrightarrow n > 0$   
<proof>

**lemma**  $smooth-ge-Suc0$ :  $smooth\ n \Longrightarrow n \geq Suc\ 0$   
<proof>

**lemma**  $prime-nat-dvdD$ :  $prime\ p \Longrightarrow (n :: nat)\ dvd\ p \Longrightarrow n = 1 \vee n = p$   
<proof>

**lemma**  $smooth-times$  [simp]:  $smooth\ (x * y) \longleftrightarrow smooth\ x \wedge smooth\ y$   
<proof>

**lemma**  $smooth2$  [simp]:  $smooth\ 2$   
<proof>

**lemma**  $smooth3$  [simp]:  $smooth\ 3$   
<proof>

**lemma**  $smooth5$  [simp]:  $smooth\ 5$

*<proof>*

**lemma** *hamming-in-smooth*:  $lset\ hamming \subseteq \{n.\ smooth\ n\}$   
*<proof>*

**lemma** *lfinite-hamming* [simp]:  $\neg\ lfinite\ hamming$   
*<proof>*

**lemma** *lsorted-hamming* [simp]: *lsorted hamming*  
**and** *ldistinct-hamming* [simp]: *ldistinct hamming*  
*<proof>*

**lemma** *smooth-hamming*:  
  **assumes** *smooth n*  
  **shows**  $n \in lset\ hamming$   
*<proof>*

**corollary** *hamming-smooth*:  $lset\ hamming = \{n.\ smooth\ n\}$   
*<proof>*

**lemma** *hamming-THE*:  
  (*THE xs. lsorted xs  $\wedge$  ldistinct xs  $\wedge$  lset xs = {n. smooth n}*) = *hamming*  
*<proof>*

**end**

## 17 Manual construction of a resumption codatatype

**theory** *Resumption* **imports**  
  *HOL-Library.Old-Datatype*  
**begin**

This theory defines the following codatatype:

```
codatatype ('a,'b,'c,'d) resumption =  
  Terminal 'a  
  | Linear 'b "('a,'b,'c,'d) resumption"  
  | Branch 'c "'d => ('a,'b,'c,'d) resumption"
```

### 17.1 Auxiliary definitions and lemmata similar to *HOL-Library.Old-Datatype*

**lemma** *Lim-mono*:  $(\bigwedge d.\ rs\ d \subseteq rs'\ d) \implies Old-Datatype.Lim\ rs \subseteq Old-Datatype.Lim\ rs'$   
*<proof>*

**lemma** *Lim-UN1*:  $Old-Datatype.Lim\ (\lambda x.\ \bigcup y.\ f\ x\ y) = (\bigcup y.\ Old-Datatype.Lim\ (\lambda x.\ f\ x\ y))$



*<proof>*

Inverse for *Old-Datatype.Lim* like *Old-Datatype.Split* and *Old-Datatype.Case* for *Scons* and *In0/In1*

**definition** *DTBranch* :: ('b  $\Rightarrow$  ('a, 'b) *Old-Datatype.dtree*)  $\Rightarrow$  'c  $\Rightarrow$  ('a, 'b) *Old-Datatype.dtree*  $\Rightarrow$  'c

**where** *DTBranch* f M = (THE u.  $\exists x. M = \text{Old-Datatype.Lim } x \wedge u = f x$ )

**lemma** *DTBranch-Lim [simp]*: *DTBranch* f (*Old-Datatype.Lim* M) = f M

*<proof>*

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

*<proof>*

**lemma** *ndepth-Push-Node-Inl*:

*ndepth* (*Push-Node* (*Inl* a) n) = *Suc* (*ndepth* n)

*<proof>*

**lemma** *ntrunc-Lim [simp]*: *ntrunc* (*Suc* k) (*Old-Datatype.Lim* rs) = *Old-Datatype.Lim* ( $\lambda x. \text{ntrunc } k (rs x)$ )

*<proof>*

## 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*:  $TERMINAL\ a = TERMINAL\ a' \longleftrightarrow a = a'$   
**and** *LINEAR-inject*:  $LINEAR\ b\ r = LINEAR\ b'\ r' \longleftrightarrow b = b' \wedge r = r'$   
**and** *BRANCH-inject*:  $BRANCH\ c\ rs = BRANCH\ c'\ rs' \longleftrightarrow c = c' \wedge rs = rs'$   
 <proof>

**lemma** *case-RESUMPTION-simps* [simp]:

**shows** *case-RESUMPTION-TERMINAL*:  $case-RESUMPTION\ t\ l\ br\ (TERMINAL\ a) = t\ a$   
**and** *case-RESUMPTION-LINEAR*:  $case-RESUMPTION\ t\ l\ br\ (LINEAR\ b\ r) = l\ b\ r$   
**and** *case-RESUMPTION-BRANCH*:  $case-RESUMPTION\ t\ l\ br\ (BRANCH\ c\ rs) = br\ c\ rs$   
 <proof>

**lemma** *LINEAR-mono*:  $r \subseteq r' \implies LINEAR\ b\ r \subseteq LINEAR\ b\ r'$   
 <proof>

**lemma** *BRANCH-mono*:  $(\bigwedge d. rs\ d \subseteq rs'\ d) \implies BRANCH\ c\ rs \subseteq BRANCH\ c\ rs'$   
 <proof>

**lemma** *LINEAR-UN*:  $LINEAR\ b\ (\bigcup x. f\ x) = (\bigcup x. LINEAR\ b\ (f\ x))$   
 <proof>

**lemma** *BRANCH-UN*:  $BRANCH\ b\ (\lambda d. \bigcup x. f\ d\ x) = (\bigcup x. BRANCH\ b\ (\lambda d. f\ d\ x))$   
 <proof>

The codatatype universe

**coinductive-set** *resumption* :: ('c + 'b + 'a, 'd) Old-Datatype.dtree set

**where**

*resumption-TERMINAL*:

$TERMINAL\ a \in resumption$

| *resumption-LINEAR*:

$r \in resumption \implies LINEAR\ b\ r \in resumption$

| *resumption-BRANCH*:

$(\bigwedge d. rs\ d \in resumption) \implies BRANCH\ c\ rs \in resumption$

### 17.3 Definition of the codatatype as a type

**typedef**  $(\text{'a}, \text{'b}, \text{'c}, \text{'d})$  *resumption* = *resumption* ::  $(\text{'c} + \text{'b} + \text{'a}, \text{'d})$  *Old-Datatype.dtree*  
*set*  
 ⟨*proof*⟩

Constructors

**definition** *Terminal* ::  $\text{'a} \Rightarrow (\text{'a}, \text{'b}, \text{'c}, \text{'d})$  *resumption*  
**where** *Terminal*  $a = \text{Abs-resumption} (\text{TERMINAL } a)$

**definition** *Linear* ::  $\text{'b} \Rightarrow (\text{'a}, \text{'b}, \text{'c}, \text{'d})$  *resumption*  $\Rightarrow (\text{'a}, \text{'b}, \text{'c}, \text{'d})$  *resumption*  
**where** *Linear*  $b\ r = \text{Abs-resumption} (\text{LINEAR } b (\text{Rep-resumption } r))$

**definition** *Branch* ::  $\text{'c} \Rightarrow (\text{'d} \Rightarrow (\text{'a}, \text{'b}, \text{'c}, \text{'d})$  *resumption*)  $\Rightarrow (\text{'a}, \text{'b}, \text{'c}, \text{'d})$  *resumption*  
**where** *Branch*  $c\ rs = \text{Abs-resumption} (\text{BRANCH } c (\lambda d. \text{Rep-resumption } (rs\ d)))$

**lemma** [*iff*]:

**shows** *Terminal-not-Linear*: *Terminal*  $a \neq \text{Linear } b\ r$   
**and** *Linear-not-Terminal*: *Linear*  $b\ r \neq \text{Terminal } a$   
**and** *Terminal-not-Branch*: *Terminal*  $a \neq \text{Branch } c\ rs$   
**and** *Branch-not-Terminal*: *Branch*  $c\ rs \neq \text{Terminal } a$   
**and** *Linear-not-Branch*: *Linear*  $b\ r \neq \text{Branch } c\ rs$   
**and** *Branch-not-Linear*: *Branch*  $c\ rs \neq \text{Linear } b\ r$   
**and** *Terminal-inject*: *Terminal*  $a = \text{Terminal } a' \longleftrightarrow a = a'$   
**and** *Linear-inject*: *Linear*  $b\ r = \text{Linear } b'\ r' \longleftrightarrow b = b' \wedge r = r'$   
**and** *Branch-inject*: *Branch*  $c\ rs = \text{Branch } c'\ rs' \longleftrightarrow c = c' \wedge rs = rs'$

⟨*proof*⟩

**lemma** *Rep-resumption-constructors*:

**shows** *Rep-resumption-Terminal*: *Rep-resumption* (*Terminal*  $a$ ) = *TERMINAL*  $a$   
**and** *Rep-resumption-Linear*: *Rep-resumption* (*Linear*  $b\ r$ ) = *LINEAR*  $b$  (*Rep-resumption*  $r$ )  
**and** *Rep-resumption-Branch*: *Rep-resumption* (*Branch*  $c\ rs$ ) = *BRANCH*  $c$  ( $\lambda d. \text{Rep-resumption } (rs\ d)$ )  
 ⟨*proof*⟩

Case operator

**definition** *case-resumption* ::  $(\text{'a} \Rightarrow \text{'e}) \Rightarrow (\text{'b} \Rightarrow (\text{'a}, \text{'b}, \text{'c}, \text{'d})$  *resumption*  $\Rightarrow \text{'e}) \Rightarrow$   
 $(\text{'c} \Rightarrow (\text{'d} \Rightarrow (\text{'a}, \text{'b}, \text{'c}, \text{'d})$  *resumption*)  $\Rightarrow \text{'e}) \Rightarrow (\text{'a}, \text{'b}, \text{'c}, \text{'d})$   
*resumption*  $\Rightarrow \text{'e}$

**where** [*code del*]:

*case-resumption*  $t\ l\ br\ r =$   
*case-RESUMPTION*  $t (\lambda b\ r. l\ b (\text{Abs-resumption } r)) (\lambda c\ rs. br\ c (\lambda d. \text{Abs-resumption } (rs\ d))) (\text{Rep-resumption } r)$

**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*:  $\text{case-resumption } t \text{ l br } (\text{Branch } c \text{ rs}) = \text{br } c \text{ rs}$   
 $\langle \text{proof} \rangle$

**declare** [[*case-translation case-resumption Terminal Linear Branch*]]

**lemma** *case-resumption-cert*:

**assumes**  $\text{CASE} \equiv \text{case-resumption } t \text{ l br}$   
**shows**  $(\text{CASE } (\text{Terminal } a) \equiv t \text{ a}) \ \&\&\& \ (\text{CASE } (\text{Linear } b \text{ r}) \equiv l \text{ b r}) \ \&\&\& \ (\text{CASE } (\text{Branch } c \text{ rs}) \equiv \text{br } c \text{ rs})$   
 $\langle \text{proof} \rangle$

**code-datatype** *Terminal Linear Branch*

$\langle \text{ML} \rangle$

**lemma** *resumption-exhaust* [*cases type: resumption*]:

**obtains**  $(\text{Terminal}) \ a$  **where**  $x = \text{Terminal } a$   
 $| (\text{Linear}) \ b \ r$  **where**  $x = \text{Linear } b \ r$   
 $| (\text{Branch}) \ c \ \text{rs}$  **where**  $x = \text{Branch } c \ \text{rs}$   
 $\langle \text{proof} \rangle$

**lemma** *resumption-split*:

$P (\text{case-resumption } t \text{ l br } r) \longleftrightarrow$   
 $(\forall a. r = \text{Terminal } a \longrightarrow P (t \ a)) \ \wedge$   
 $(\forall b \ r'. r = \text{Linear } b \ r' \longrightarrow P (l \ b \ r')) \ \wedge$   
 $(\forall c \ \text{rs}. r = \text{Branch } c \ \text{rs} \longrightarrow P (\text{br } c \ \text{rs}))$   
 $\langle \text{proof} \rangle$

**lemma** *resumption-split-asm*:

$P (\text{case-resumption } t \text{ l br } r) \longleftrightarrow$   
 $\neg ((\exists a. r = \text{Terminal } a \ \wedge \neg P (t \ a)) \vee$   
 $(\exists b \ r'. r = \text{Linear } b \ r' \ \wedge \neg P (l \ b \ r')) \vee$   
 $(\exists c \ \text{rs}. r = \text{Branch } c \ \text{rs} \ \wedge \neg P (\text{br } c \ \text{rs})))$   
 $\langle \text{proof} \rangle$

**lemmas** *resumption-splits = resumption-split resumption-split-asm*

corecursion operator

**datatype**  $(\text{dead } 'a, \text{dead } 'b, \text{dead } 'c, \text{dead } 'd, \text{dead } 'e) \text{ resumption-corec} =$   
 $\text{Terminal-corec } 'a$   
 $| \text{Linear-corec } 'b \ 'e$   
 $| \text{Branch-corec } 'c \ 'd \Rightarrow 'e$   
 $| \text{Resumption-corec } ('a, 'b, 'c, 'd) \text{ resumption}$

**primrec** *RESUMPTION-corec-aux* ::  $\text{nat} \Rightarrow ('e \Rightarrow ('a, 'b, 'c, 'd, 'e) \text{ resumption-corec})$   
 $\Rightarrow 'e \Rightarrow ('c + 'b + 'a, 'd) \text{ Old-Datatype.dtree}$

**where**

$\text{RESUMPTION-corec-aux } 0 \ f \ e = \{\}$   
 $| \text{RESUMPTION-corec-aux } (\text{Suc } n) \ f \ e =$

(case  $f e$  of Terminal-corec  $a \Rightarrow \text{TERMINAL } a$   
 | Linear-corec  $b e' \Rightarrow \text{LINEAR } b (\text{RESUMPTION-corec-aux } n f e')$   
 | Branch-corec  $c es \Rightarrow \text{BRANCH } c (\lambda d. \text{RESUMPTION-corec-aux } n f$   
 ( $es d$ )  
 | Resumption-corec  $r \Rightarrow \text{Rep-resumption } r$ )

**definition**  $\text{RESUMPTION-corec} :: ('e \Rightarrow ('a, 'b, 'c, 'd, 'e) \text{resumption-corec}) \Rightarrow 'e \Rightarrow ('c + 'b + 'a, 'd) \text{Old-Datatype.dtree}$

**where**

$\text{RESUMPTION-corec } f e = (\bigcup n. \text{RESUMPTION-corec-aux } n f e)$

**lemma**  $\text{RESUMPTION-corec}$  [*nitpick-simp*]:

$\text{RESUMPTION-corec } f e =$

(case  $f e$  of Terminal-corec  $a \Rightarrow \text{TERMINAL } a$   
 | Linear-corec  $b e' \Rightarrow \text{LINEAR } b (\text{RESUMPTION-corec } f e')$   
 | Branch-corec  $c es \Rightarrow \text{BRANCH } c (\lambda d. \text{RESUMPTION-corec } f (es d))$   
 | Resumption-corec  $r \Rightarrow \text{Rep-resumption } r$ )

(**is**  $?lhs = ?rhs$ )

$\langle \text{proof} \rangle$

**lemma**  $\text{RESUMPTION-corec-type}$ :  $\text{RESUMPTION-corec } f e \in \text{resumption}$

$\langle \text{proof} \rangle$

corecursion operator for the resumption type

**definition**  $\text{resumption-corec} :: ('e \Rightarrow ('a, 'b, 'c, 'd, 'e) \text{resumption-corec}) \Rightarrow 'e \Rightarrow ('a, 'b, 'c, 'd) \text{resumption}$

**where**

$\text{resumption-corec } f e = \text{Abs-resumption } (\text{RESUMPTION-corec } f e)$

**lemma**  $\text{resumption-corec}$ :

$\text{resumption-corec } f e =$

(case  $f e$  of Terminal-corec  $a \Rightarrow \text{Terminal } a$   
 | Linear-corec  $b e' \Rightarrow \text{Linear } b (\text{resumption-corec } f e')$   
 | Branch-corec  $c es \Rightarrow \text{Branch } c (\lambda d. \text{resumption-corec } f (es d))$   
 | Resumption-corec  $r \Rightarrow r$ )

$\langle \text{proof} \rangle$

Equality as greatest fixpoint

**coinductive**  $\text{Eq-RESUMPTION} :: ('c + 'b + 'a, 'd) \text{Old-Datatype.dtree} \Rightarrow ('c + 'b + 'a, 'd) \text{Old-Datatype.dtree} \Rightarrow \text{bool}$

**where**

$\text{EqTERMINAL}$ :  $\text{Eq-RESUMPTION } (\text{TERMINAL } a) (\text{TERMINAL } a)$   
 |  $\text{EqLINEAR}$ :  $\text{Eq-RESUMPTION } r r' \Longrightarrow \text{Eq-RESUMPTION } (\text{LINEAR } b r) (\text{LINEAR } b r')$   
 |  $\text{EqBRANCH}$ :  $(\bigwedge d. \text{Eq-RESUMPTION } (rs d) (rs' d)) \Longrightarrow \text{Eq-RESUMPTION } (\text{BRANCH } c rs) (\text{BRANCH } c rs')$

**lemma**  $\text{Eq-RESUMPTION-implies-ntrunc-equality}$ :

$\text{Eq-RESUMPTION } r r' \Longrightarrow \text{ntrunc } k r = \text{ntrunc } k r'$

$\langle proof \rangle$

**lemma** *Eq-RESUMPTION-refl*:

**assumes**  $r \in resumption$

**shows** *Eq-RESUMPTION*  $r$   $r$

$\langle proof \rangle$

**lemma** *Eq-RESUMPTION-into-resumption*:

**assumes** *Eq-RESUMPTION*  $r$   $r$

**shows**  $r \in resumption$

$\langle proof \rangle$

**lemma** *Eq-RESUMPTION-eq*:

*Eq-RESUMPTION*  $r$   $r'$   $\longleftrightarrow$   $r = r' \wedge r \in resumption$

$\langle proof \rangle$

**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-RESUMPTION*  $R R')) \vee$

$(\exists rs rs' c. r = \text{BRANCH } c rs \wedge r' = \text{BRANCH } c rs' \wedge (\forall d. X (rs d)$

$(rs' d) \vee \text{Eq-RESUMPTION } (rs d) (rs' d)))$

**shows**  $r = r'$

$\langle proof \rangle$

**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**  $r = r'$

$\langle proof \rangle$

Finality of *resumption*: Uniqueness of functions defined by corecursion.

**lemma** *equals-RESUMPTION-corec*:

**assumes**  $h: \bigwedge x. h$   $x = (\text{case } f$   $x$  of *Terminal-corec*  $a \Rightarrow \text{TERMINAL } a$

$| \text{Linear-corec } b$   $x' \Rightarrow \text{LINEAR } b$   $(h$   $x')$

$| \text{Branch-corec } c$   $xs \Rightarrow \text{BRANCH } c$   $(\lambda d. h$   $(xs$   $d))$

$| \text{Resumption-corec } r \Rightarrow \text{Rep-resumption } r)$

**shows**  $h = \text{RESUMPTION-corec } f$

$\langle proof \rangle$

**lemma** *equals-resumption-corec*:

```

assumes  $h: \bigwedge x. h\ x = (\text{case } f\ x \text{ of } \text{Terminal-corec } a \Rightarrow \text{Terminal } a$ 
          |  $\text{Linear-corec } b\ x' \Rightarrow \text{Linear } b\ (h\ x')$ 
          |  $\text{Branch-corec } c\ xs \Rightarrow \text{Branch } c\ (\lambda d. h\ (xs\ d))$ 
          |  $\text{Resumption-corec } r \Rightarrow r)$ 
shows  $h = \text{resumption-corec } f$ 
<proof>

end

```

```

theory Coinductive-Examples imports
  LList-CCPO-Topology
  TLList-CCPO-Examples
  Koenigslemma
  LMirror
  Hamming-Stream
  Resumption
begin

end

```