

More Operations on Lazy Lists

Andrei Popescu Jamie Wright

February 6, 2026

Abstract

We formalize some operations and reasoning infrastructure on lazy (coinductive) lists. The operations include: building a lazy list from a function on naturals and an extended natural indicating the intended domain, take-until and drop-until (which are variations of take-while and drop-while), splitting a lazy list into a lazy list of lists with cut points being those elements that satisfy a predicate, and filtermap. The reasoning infrastructure includes: a variation of the corecursion combinator, multi-step (list-based) coinduction for lazy-list equality, and a criterion for the filtermapped equality of two lazy lists.

Contents

1	Filtermap for Lazy Lists	1
1.1	Preliminaries	1
1.2	Filtermap	2
2	Some Operations on Lazy Lists	4
2.1	Preliminaries	4
2.2	More properties of operators from the Coinductive library . .	4
2.3	A convenient adaptation of the lazy-list corecursor	6
2.4	Multi-step coinduction for llist equality	6
2.5	Interval lazy lists	7
2.6	Function builders for lazy lists	7
2.7	The butlast (reverse tail) operation	8
2.8	Consecutive-elements sublists	9
2.9	Take-until and drop-until	10
2.10	Splitting a lazy list according to the points where a predicate is satisfied	12
2.11	The split remainder	13
2.12	The first index for which a predicate holds (if any)	14
2.13	The first index for which the list in a lazy-list of lists is non- empty	14

3	Filtermap for Lazy Lists	15
3.1	Lazy lists filtermap	15
3.2	Coinductive criterion for filtermap equality	18
3.3	A concrete instantiation of the criterion	20

1 Filtermap for Lazy Lists

```

theory List-Filtermap
  imports Main
begin

```

This theory defines the filtermap operator for lazy lists, proves its basic properties, and proves coinductive criteria for the equality of two filtermapped lazy lists.

1.1 Preliminaries

```

hide-const filtermap

```

```

abbreviation never :: ('a  $\Rightarrow$  bool)  $\Rightarrow$  'a list  $\Rightarrow$  bool where never U  $\equiv$  list-all ( $\lambda$ 
a.  $\neg$  U a)

```

```

lemma never-list-ex: never pred xs  $\longleftrightarrow$   $\neg$  list-ex pred xs
<proof>

```

```

abbreviation Rcons (infix <##> 70) where xs ## x  $\equiv$  xs @ [x]

```

```

lemma two-singl-Rcons: [a,b] = [a] ## b <proof>

```

```

lemma length-gt-1-Cons-snoc:
  assumes length ys > 1
  obtains x1 xs x2 where ys = x1 # xs ## x2
<proof>

```

```

lemma right-cons-left[simp]: i < length as  $\implies$  (as ## a)!i = as!i
<proof>

```

1.2 Filtermap

```

definition filtermap :: ('b  $\Rightarrow$  bool)  $\Rightarrow$  ('b  $\Rightarrow$  'a)  $\Rightarrow$  'b list  $\Rightarrow$  'a list where
filtermap pred func xs  $\equiv$  map func (filter pred xs)

```

```

lemma filtermap-Nil[simp]:

```

filtermap pred func [] = []
<proof>

lemma *filtermap-Cons-not[simp]*:
 $\neg \text{pred } x \implies \text{filtermap pred func } (x \# xs) = \text{filtermap pred func } xs$
<proof>

lemma *filtermap-Cons[simp]*:
 $\text{pred } x \implies \text{filtermap pred func } (x \# xs) = \text{func } x \# \text{filtermap pred func } xs$
<proof>

lemma *filtermap-append*: $\text{filtermap pred func } (xs @ xs1) = \text{filtermap pred func } xs @ \text{filtermap pred func } xs1$
<proof>

lemma *filtermap-Nil-list-ex*: $\text{filtermap pred func } xs = [] \iff \neg \text{list-ex pred } xs$
<proof>

lemma *filtermap-Nil-never*: $\text{filtermap pred func } xs = [] \iff \text{never pred } xs$
<proof>

lemma *length-filtermap*: $\text{length } (\text{filtermap pred func } xs) \leq \text{length } xs$
<proof>

lemma *filtermap-list-all[simp]*: $\text{filtermap pred func } xs = \text{map func } xs \iff \text{list-all pred } xs$
<proof>

lemma *filtermap-eq-Cons*:
assumes $\text{filtermap pred func } xs = a \# al1$
shows $\exists x \ xs2 \ xs1. \ xs = xs2 @ [x] @ xs1 \wedge \text{never pred } xs2 \wedge \text{pred } x \wedge \text{func } x = a \wedge \text{filtermap pred func } xs1 = al1$
<proof>

lemma *filtermap-eq-append*:
assumes $\text{filtermap pred func } xs = al1 @ al2$
shows $\exists xs1 \ xs2. \ xs = xs1 @ xs2 \wedge \text{filtermap pred func } xs1 = al1 \wedge \text{filtermap pred func } xs2 = al2$
<proof>

lemma *holds-filtermap-RCons[simp]*:
 $\text{pred } x \implies \text{filtermap pred func } (xs \#\# x) = \text{filtermap pred func } xs \#\# \text{func } x$
<proof>

lemma *not-holds-filtermap-RCons[simp]*:
 $\neg \text{pred } x \implies \text{filtermap pred func } (xs \#\# x) = \text{filtermap pred func } xs$
<proof>

lemma *filtermap-eq-RCons*:

assumes *filtermap pred func xs = all ## a*

shows $\exists x \ xs1 \ xs2.$

$xs = xs1 \ @ \ [x] \ @ \ xs2 \ \wedge \ never \ pred \ xs2 \ \wedge \ pred \ x \ \wedge \ func \ x = a \ \wedge \ filtermap \ pred$

$func \ xs1 = all$

<proof>

lemma *filtermap-eq-Cons-RCons*:

assumes *filtermap pred func xs = a ## all ## b*

shows $\exists \ xsa \ xa \ xs1 \ xb \ xsb.$

$xs = xsa \ @ \ [xa] \ @ \ xs1 \ @ \ [xb] \ @ \ xsb \ \wedge$

$never \ pred \ xsa \ \wedge$

$pred \ xa \ \wedge \ func \ xa = a \ \wedge$

$filtermap \ pred \ func \ xs1 = all \ \wedge$

$pred \ xb \ \wedge \ func \ xb = b \ \wedge$

$never \ pred \ xsb$

<proof>

lemma *filter-Nil-never*: $\square = filter \ pred \ xs \implies never \ pred \ xs$

<proof>

lemma *never-Nil-filter*: $never \ pred \ xs \longleftrightarrow \square = filter \ pred \ xs$

<proof>

lemma *set-filtermap*:

$set \ (filtermap \ pred \ func \ xs) \subseteq \{func \ x \mid x . x \in set \ xs \ \wedge \ pred \ x\}$

<proof>

end

2 Some Operations on Lazy Lists

theory *LazyList-Operations*

imports *Coinductive.Coinductive-List List-Filtermap*

begin

This theory defines some operations for lazy lists, and proves their basic properties.

2.1 Preliminaries

lemma *enat-ls-minius-1*: $enat \ i < j - 1 \implies enat \ i < j$

<proof>

abbreviation *LNil-abbr* ($\langle [] \rangle$) **where** *LNil-abbr* $\equiv LNil$

abbreviation *LCons-abbr* (**infixr** $\langle \$ \rangle$ 65) **where** $x \$ xs \equiv LCons\ x\ xs$

abbreviation *lnever* :: ($'a \Rightarrow bool$) $\Rightarrow 'a\ llist \Rightarrow bool$ **where** $lnever\ U \equiv llist\ all\ (\lambda\ a.\ \neg\ U\ a)$

syntax

— *l*list Enumeration
*-l*list :: $args \Rightarrow 'a\ llist$ ($\langle [[(-)]] \rangle$)

syntax-consts

*-l*list == *LCons*

translations

$[[x, xs]] == x \$ [[xs]]$
 $[[x]] == x \$ [[]]$

declare *l*list-of-eq-LNil-conv[*simp*]

declare *l*map-eq-LNil[*simp*]

declare *l*length-*l*tl[*simp*]

2.2 More properties of operators from the Coinductive library

lemma *l*nth-*l*concat:

assumes $i < llength\ (lconcat\ xss)$

shows $\exists j < llength\ xss.\ \exists k < llength\ (lnth\ xss\ j).\ lnth\ (lconcat\ xss)\ i = lnth\ (lnth\ xss\ j)\ k$

<proof>

lemma *l*nth-0-*l*set: $xs \neq [] \implies lnth\ xs\ 0 \in lset\ xs$

<proof>

lemma *l*concat-eq-LNil-iff: $lconcat\ xss = [] \longleftrightarrow (\forall xs \in lset\ xss.\ xs = [])$

<proof>

lemma *l*last-last-*l*list-of: $lfinite\ xs \implies llast\ xs = last\ (list\ of\ xs)$

<proof>

lemma *l*append-*l*list-of-inj:

$length\ xs = length\ ys \implies$

$lappend\ (llist\ of\ xs)\ as = lappend\ (llist\ of\ ys)\ bs \longleftrightarrow xs = ys \wedge as = bs$

<proof>

lemma *l*list-all-*l*nth: $llist\ all\ P\ xs = (\forall n < llength\ xs.\ P\ (lnth\ xs\ n))$

<proof>

lemma *l*list-eq-cong:

assumes $length\ xs = length\ ys \wedge i < length\ xs \implies lnth\ xs\ i = lnth\ ys\ i$

shows $xs = ys$
<proof>

lemma *llist-cases*: $l\text{length } xs = \infty \vee (\exists ys. xs = \text{llist-of } ys)$
<proof>

lemma *llist-all-lappend*: $l\text{finite } xs \implies$
 $l\text{list-all } \text{pred } (\text{lappend } xs \text{ } ys) \longleftrightarrow l\text{list-all } \text{pred } xs \wedge l\text{list-all } \text{pred } ys$
<proof>

lemma *llist-all-lappend-llist-of*:
 $l\text{list-all } \text{pred } (\text{lappend } (\text{llist-of } xs) \text{ } ys) \longleftrightarrow l\text{list-all } \text{pred } xs \wedge l\text{list-all } \text{pred } ys$
<proof>

lemma *llist-all-conduct*:
 $X \text{ } xs \implies$
 $(\bigwedge xs. X \text{ } xs \implies \neg l\text{null } xs \implies P (\text{lhs } xs) \wedge (X (\text{lhs } xs) \vee l\text{list-all } P (\text{lhs } xs))) \implies$
 $l\text{list-all } P \text{ } xs$
<proof>

lemma *lfilter-lappend-llist-of*:
 $l\text{filter } P (\text{lappend } (\text{llist-of } xs) \text{ } ys) = \text{lappend } (\text{llist-of } (\text{filter } P \text{ } xs)) (l\text{filter } P \text{ } ys)$
<proof>

lemma *ldrop-Suc*: $n < l\text{length } xs \implies l\text{drop } (\text{enat } n) \text{ } xs = L\text{Cons } (\text{lnth } xs \text{ } n) (l\text{drop } (\text{enat } (\text{Suc } n)) \text{ } xs)$
<proof>

lemma *lappend-ltake-lnth-ldrop*: $n < l\text{length } xs \implies$
 $\text{lappend } (\text{ltake } (\text{enat } n) \text{ } xs) (L\text{Cons } (\text{lnth } xs \text{ } n) (l\text{drop } (\text{enat } (\text{Suc } n)) \text{ } xs)) = xs$
<proof>

lemma *ltake-eq-LNil*: $ltake \ i \ tr = [] \longleftrightarrow i = 0 \vee tr = []$
<proof>

lemma *ex-llength-infity*:
 $\exists a. l\text{length } a = \infty \wedge \text{lhs } a = 0$
<proof>

lemma *repeat-not-Nil[simp]*: $\text{repeat } a \neq []$
<proof>

2.3 A convenient adaptation of the lazy-list corecursor

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

where *ccorec-llist is n h ec e t* \equiv

ccorec-llist is n $(\lambda a. \text{if } \text{ec } a \text{ then } \text{lhs } (e \ a) \text{ else } h \ a) \text{ ec } (\lambda a. \text{case } e \ a \text{ of } b \ \$ \ a' \Rightarrow a') \ t$

lemma *llist-ccorec-LNil*: $isn\ a \implies ccorec-llist\ isn\ h\ ec\ e\ t\ a = []$
 ⟨proof⟩

lemma *llist-ccorec-LCons*:
 $\neg\ lnull\ (e\ a) \implies \neg\ isn\ a \implies$
 $ccorec-llist\ isn\ h\ ec\ e\ t\ a = (if\ ec\ a\ then\ e\ a\ else\ h\ a\ \$\ ccorec-llist\ isn\ h\ ec\ e\ t\ (t\ a))$
 ⟨proof⟩

lemmas *llist-ccorec* = *llist-ccorec-LNil* *llist-ccorec-LCons*

2.4 Multi-step coinduction for llist equality

In this principle, the coinductive step can consume any non-empty list, not just single elements.

proposition *llist-lappend-coind*:

assumes $P: P\ lxs\ lxs'$

and *lappend*:

$\bigwedge\ lxs\ lxs'. P\ lxs\ lxs' \implies$

$lxs = lxs' \vee$

$(\exists\ ys\ lxs\ lxs'. ys \neq [] \wedge$

$lxs = lappend\ (l\!list\!of\ ys)\ lxs \wedge lxs' = lappend\ (l\!list\!of\ ys)\ lxs' \wedge$

$P\ lxs\ lxs')$

shows $lxs = lxs'$

⟨proof⟩

2.5 Interval lazy lists

The list of all naturals between a natural and an extended-natural

primcorec *betw* :: $nat \Rightarrow enat \Rightarrow nat\ llist$ **where**

betw $i\ n = (if\ i \geq n\ then\ LNil\ else\ i\ \$\ betw\ (Suc\ i)\ n)$

lemma *betw-more-simps*:

$\neg\ n \leq i \implies betw\ i\ n = i\ \$\ betw\ (Suc\ i)\ n$

⟨proof⟩

lemma *lhd-betw*: $i < n \implies lhd\ (betw\ i\ n) = i$

⟨proof⟩

lemma *not-lfinite-betw-infty*: $\neg\ lfinite\ (betw\ i\ \infty)$

⟨proof⟩

lemma *llength-betw-infty[simp]*: $llength\ (betw\ i\ \infty) = \infty$

⟨proof⟩

lemma *llength-betw*: $llength\ (betw\ i\ n) = n - i$

<proof>

lemma *lfinite-betw-not-infty*: $n < \infty \implies \text{lfinite } (\text{betw } i \ n)$
<proof>

lemma *lfinite-betw-enat*: $\text{lfinite } (\text{betw } i \ (\text{enat } n))$
<proof>

lemma *lnth-betw*: $\text{enat } j < n - i \implies \text{lnth } (\text{betw } i \ n) \ j = i + j$
<proof>

2.6 Function builders for lazy lists

Building an llist from a function, more precisely from its values between 0 and a given extended natural n

definition *build* $n \ f \equiv \text{lmap } f \ (\text{betw } 0 \ n)$

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

lemma *lnth-build[simp]*: $i < n \implies \text{lnth } (\text{build } n \ f) \ i = f \ i$
<proof>

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

lemma *build-eq-LNil[simp]*: $\text{build } n \ f = [] \longleftrightarrow n = 0$
<proof>

2.7 The butlast (reverse tail) operation

definition *lbutlast* :: 'a llist \Rightarrow 'a llist **where**
lbutlast $sl \equiv \text{if } \text{lfinite } sl \text{ then } \text{llist-of } (\text{butlast } (\text{list-of } sl)) \text{ else } sl$

lemma *llength-lbutlast-lfinite[simp]*:
 $sl \neq [] \implies \text{lfinite } sl \implies \text{llength } (\text{lbutlast } sl) = \text{llength } sl - 1$
<proof>

lemma *llength-lbutlast-not-lfinite[simp]*:
 $\neg \text{lfinite } sl \implies \text{llength } (\text{lbutlast } sl) = \infty$
<proof>

lemma *lbutlast-LNil[simp]*:
lbutlast $[] = []$
<proof>

lemma *lbutlast-singl[simp]*:
lbutlast $[[s]] = []$
<proof>

lemma *lbutlast-lfinite*[simp]:

lfinite sl \implies *lbutlast sl* = *llist-of* (*butlast* (*list-of sl*))
<proof>

lemma *lbutlast-Cons*[simp]: *tr* \neq [] \implies *lbutlast* (*s* \$ *tr*) = *s* \$ *lbutlast tr*
<proof>

lemma *llist-of-butlast*: *llist-of* (*butlast xs*) = *lbutlast* (*llist-of xs*)
<proof>

lemma *lprefix-lbutlast*: *lprefix xs ys* \implies *lprefix* (*lbutlast xs*) (*lbutlast ys*)
<proof>

lemma *lbutlast-lappend*:

assumes (*ys*::'a *llist*) \neq [] **shows** *lbutlast* (*lappend xs ys*) = *lappend xs* (*lbutlast ys*)
<proof>

lemma *lbutlast-llist-of*: *lbutlast* (*llist-of xs*) = *llist-of* (*butlast xs*)
<proof>

lemma *butlast-list-of*: *lfinite xs* \implies *butlast* (*list-of xs*) = *list-of* (*lbutlast xs*)
<proof>

lemma *butlast-length-le1*[simp]: *llength xs* \leq *Suc 0* \implies *lbutlast xs* = []
<proof>

lemma *llength-lbutlast*[simp]: *llength* (*lbutlast tr*) = *llength tr* - 1
<proof>

lemma *lnth-lbutlast*: *i* < *llength xs* - 1 \implies *lnth* (*lbutlast xs*) *i* = *lnth xs i*
<proof>

2.8 Consecutive-elements sublists

definition *lsublist xs ys* \equiv \exists *us vs*. *lfinite us* \wedge *ys* = *lappend us* (*lappend xs vs*)

lemma *lsublist-refl*: *lsublist xs xs*
<proof>

lemma *lsublist-trans*:

assumes *lsublist xs ys* **and** *lsublist ys zs* **shows** *lsublist xs zs*
<proof>

lemma *lnth-lconcat-lsublist*:

assumes *xs*: *xs* = *lconcat* (*lmap llist-of xss*) **and** *i* < *llength xss*
shows *lsublist* (*llist-of* (*lnth xss i*)) *xs*
<proof>

lemma *lnth-lconcat-lsublist2*:

assumes xs : $xs = lconcat (lmap \text{ llist-of } xss)$ **and** $Suc\ i < llength\ xss$
shows $lsublist (\text{ llist-of } (\text{ append } (lnth\ xss\ i) (lnth\ xss\ (Suc\ i))))\ xs$
(*proof*)

lemma *lnth-lconcat-lconcat-lsublist*:

assumes xs : $xs = lappend (lconcat (lmap \text{ llist-of } xss))\ ys$ **and** $i < llength\ xss$
shows $lsublist (\text{ llist-of } (lnth\ xss\ i))\ xs$
(*proof*)

lemma *lnth-lconcat-lconcat-lsublist2*:

assumes xs : $xs = lappend (lconcat (lmap \text{ llist-of } xss))\ ys$ **and** $Suc\ i < llength\ xss$
shows $lsublist (\text{ llist-of } (\text{ append } (lnth\ xss\ i) (lnth\ xss\ (Suc\ i))))\ xs$
(*proof*)

lemma *su-lset-lconcat-llist-of*:

assumes $xs \in lset\ xss$
shows $set\ xs \subseteq lset (lconcat (lmap \text{ llist-of } xss))$
(*proof*)

lemma *lsublist-lnth-lconcat*: $i < llength\ tr1s \implies lsublist (\text{ llist-of } (lnth\ tr1s\ i))$
 $(lconcat (lmap \text{ llist-of } tr1s))$
(*proof*)

lemma *lsublist-lset*:

$lsublist\ xs\ ys \implies lset\ xs \subseteq lset\ ys$
(*proof*)

lemma *lsublist-LNil*:

$lsublist\ xs\ ys \implies ys = LNil \implies xs = LNil$
(*proof*)

2.9 Take-until and drop-until

definition *ltakeUntil* :: $('a \Rightarrow bool) \Rightarrow 'a\ \text{ llist} \Rightarrow 'a\ \text{ llist}$ **where**

$ltakeUntil\ pred\ xs \equiv$

$list\text{-of } (lappend (ltakeWhile (\lambda x. \neg pred\ x))\ xs) \ [[lhd (ldropWhile (\lambda x. \neg pred\ x))\ xs]]]$

definition *ldropUntil* :: $('a \Rightarrow bool) \Rightarrow 'a\ \text{ llist} \Rightarrow 'a\ \text{ llist}$ **where**

$ldropUntil\ pred\ xs \equiv ltl (ldropWhile (\lambda x. \neg pred\ x))\ xs$

lemma *lappend-ltakeUntil-ldropUntil*:

$\exists x \in lset\ xs. pred\ x \implies lappend (\text{ llist-of } (ltakeUntil\ pred\ xs)) (ldropUntil\ pred\ xs)$
 $= xs$
(*proof*)

lemma *ltakeUntil-not-Nil*:
assumes $\exists x \in \text{lset } xs. \text{pred } x$
shows $\text{ltakeUntil pred } xs \neq []$
 $\langle \text{proof} \rangle$

lemma *ltakeUntil-ex-butlast*:
assumes $\exists x \in \text{lset } xs. \text{pred } x \ y \in \text{set } (\text{butlast } (\text{ltakeUntil pred } xs))$
shows $\neg \text{pred } y$
 $\langle \text{proof} \rangle$

lemma *ltakeUntil-never-butlast*:
assumes $\exists x \in \text{lset } xs. \text{pred } x$
shows $\text{never pred } (\text{butlast } (\text{ltakeUntil pred } xs))$
 $\langle \text{proof} \rangle$

lemma *ltakeUntil-last*:
assumes $\exists x \in \text{lset } xs. \text{pred } x$
shows $\text{pred } (\text{last } (\text{ltakeUntil pred } xs))$
 $\langle \text{proof} \rangle$

lemma *ltakeUntil-last-butlast*:
assumes $\exists x \in \text{lset } xs. \text{pred } x$
shows $\text{ltakeUntil pred } xs = \text{append } (\text{butlast } (\text{ltakeUntil pred } xs)) [\text{last } (\text{ltakeUntil pred } xs)]$
 $\langle \text{proof} \rangle$

lemma *ltakeUntil-LCons1[simp]*: $\exists x \in \text{lset } xs. \text{pred } x \implies \neg \text{pred } x \implies \text{ltakeUntil pred } (\text{LCons } x \ xs) = x \ \# \ \text{ltakeUntil pred } xs$
 $\langle \text{proof} \rangle$

lemma *ldropUntil-LCons1[simp]*: $\exists x \in \text{lset } xs. \text{pred } x \implies \neg \text{pred } x \implies \text{ldropUntil pred } (\text{LCons } x \ xs) = \text{ldropUntil pred } xs$
 $\langle \text{proof} \rangle$

lemma *ltakeUntil-LCons2[simp]*: $\exists x \in \text{lset } xs. \text{pred } x \implies \text{pred } x \implies \text{ltakeUntil pred } (\text{LCons } x \ xs) = [x]$
 $\langle \text{proof} \rangle$

lemma *ldropUntil-LCons2[simp]*: $\exists x \in \text{lset } xs. \text{pred } x \implies \text{pred } x \implies \text{ldropUntil pred } (\text{LCons } x \ xs) = xs$
 $\langle \text{proof} \rangle$

lemma *ltakeUntil-tl1[simp]*:
 $\exists x \in \text{lset } xs. \text{pred } x \implies \neg \text{pred } (\text{lhs } xs) \implies \text{ltakeUntil pred } (\text{ttl } xs) = \text{tl } (\text{ltakeUntil pred } xs)$
 $\langle \text{proof} \rangle$

lemma *ldropUntil-tl1[simp]*:

$\exists x \in \text{lset } xs. \text{pred } x \implies \neg \text{pred } (\text{lhs } xs) \implies \text{ldropUntil pred } (\text{ttl } xs) = \text{ldropUntil pred } xs$
 <proof>

lemma *ltakeUntil-tl2[simp]*:
 $xs \neq [] \implies \text{pred } (\text{lhs } xs) \implies \text{tl } (\text{ltakeUntil pred } xs) = []$
 <proof>

lemma *ldropUntil-tl2[simp]*:
 $xs \neq [] \implies \text{pred } (\text{lhs } xs) \implies \text{ldropUntil pred } xs = \text{ttl } xs$
 <proof>

lemma *LCons-lfilter-ldropUntil*: $y \$ ys = \text{lfilter pred } xs \implies ys = \text{lfilter pred } (\text{ldropUntil pred } xs)$
 <proof>

lemma *length-ltakeUntil-ge-0*:
assumes $\exists x \in \text{lset } xs. \text{pred } x$
shows $\text{length } (\text{ltakeUntil pred } xs) > 0$
 <proof>

lemma *length-ltakeUntil-eq-1*:
assumes $\exists x \in \text{lset } xs. \text{pred } x$
shows $\text{length } (\text{ltakeUntil pred } xs) = \text{Suc } 0 \iff \text{pred } (\text{lhs } xs)$
 <proof>

lemma *length-ltakeUntil-Suc*:
assumes $\exists x \in \text{lset } xs. \text{pred } x \wedge \neg \text{pred } (\text{lhs } xs)$
shows $\text{length } (\text{ltakeUntil pred } xs) = \text{Suc } (\text{length } (\text{ltakeUntil pred } (\text{ttl } xs)))$
 <proof>

2.10 Splitting a lazy list according to the points where a predicate is satisfied

primcorec *lsplit* :: $('a \Rightarrow \text{bool}) \Rightarrow 'a \text{ llist} \Rightarrow 'a \text{ list llist}$ **where**
 $\text{lsplit pred } xs =$
 (if $(\exists x \in \text{lset } xs. \text{pred } x)$
 then $LCons (\text{ltakeUntil pred } xs) (\text{lsplit pred } (\text{ldropUntil pred } xs))$
 else $[]$)

declare *lsplit.ctr[simp]*

lemma *infinite-split[simp]*:
 $\text{infinite } \{x \in \text{lset } xs. \text{pred } x\} \implies \text{lsplit pred } xs = LCons (\text{ltakeUntil pred } xs) (\text{lsplit pred } (\text{ldropUntil pred } xs))$
 <proof>

lemma *lconcat-lsplit-not-lfinite*:
 $\neg \text{lfinite } (\text{lfilter pred } xs) \implies xs = \text{lconcat } (\text{lmap llist-of } (\text{lsplit pred } xs))$

<proof>

lemma *lfinite-lsplit:*

assumes *lfinite (lfilter pred xs)*

shows *lfinite (lsplit pred xs)*

<proof>

lemma *lconcat-lsplit-lfinite:*

assumes *lfinite (lfilter pred xs)*

shows $\exists ys. xs = lappend (lconcat (lmap llist-of (lsplit pred xs))) ys \wedge (\forall y \in lset ys. \neg pred y)$

<proof>

lemma *lconcat-lsplit:*

$\exists ys. xs = lappend (lconcat (lmap llist-of (lsplit pred xs))) ys \wedge (\forall y \in lset ys. \neg pred y)$

<proof>

lemma *lsublist-lsplit:*

assumes $i < llength (lsplit pred xs)$

shows *lsublist (llist-of (lnth (lsplit pred xs) i)) xs*

<proof>

lemma *lsublist-lsplit2:*

assumes $Suc\ i < llength (lsplit pred xs)$

shows *lsublist (llist-of (append (lnth (lsplit pred xs) i) (lnth (lsplit pred xs) (Suc i)))) xs*

<proof>

lemma *lsplit-main:*

$l\text{list-all } (\lambda zs. zs \neq [] \wedge l\text{list-all } (\lambda z. \neg pred\ z) (butlast\ zs) \wedge pred\ (last\ zs)) (lsplit\ pred\ xs)$

<proof>

lemma *lsplit-main-lset:*

assumes $ys \in lset (lsplit pred xs)$

shows $ys \neq [] \wedge$

$l\text{list-all } (\lambda z. \neg pred\ z) (butlast\ ys) \wedge pred\ (last\ ys)$

<proof>

lemma *lsplit-main-lnth:*

assumes $i < llength (lsplit pred xs)$

shows $lnth (lsplit pred xs) i \neq [] \wedge$

$l\text{list-all } (\lambda z. \neg pred\ z) (butlast (lnth (lsplit pred xs) i)) \wedge pred (last (lnth (lsplit pred xs) i))$

<proof>

lemma *hd-lhd-lsplit:* $\exists x \in lset xs. pred\ x \implies hd (lhd (lsplit pred xs)) = lhd\ xs$

<proof>

lemma *lprefix-lsplit*:

assumes $\exists x \in \text{lset } xs. \text{pred } x$

shows *lprefix* (*llist-of* (*lhd* (*lsplit pred xs*))) *xs*

<proof>

lemma *lprefix-lsplit-lbutlast*:

assumes $\exists x \in \text{lset } xs. \text{pred } x$

shows *lprefix* (*llist-of* (*lbutlast* (*lhd* (*lsplit pred xs*)))) (*lbutlast xs*)

<proof>

lemma *set-lset-lsplit*:

assumes $ys \in \text{lset } (\text{lsplit pred } xs)$

shows $\text{set } ys \subseteq \text{lset } xs$

<proof>

lemma *set-lnth-lsplit*:

assumes $i < \text{llength } (\text{lsplit pred } xs)$

shows $\text{set } (\text{lnth } (\text{lsplit pred } xs) i) \subseteq \text{lset } xs$

<proof>

2.11 The split remainder

definition *lsplitRemainder pred xs* $\equiv \text{SOME } ys. xs = \text{lappend } (\text{lconcat } (\text{lmap } \text{llist-of } (\text{lsplit pred } xs))) ys \wedge (\forall y \in \text{lset } ys. \neg \text{pred } y)$

lemma *lsplitRemainder*:

$xs = \text{lappend } (\text{lconcat } (\text{lmap } \text{llist-of } (\text{lsplit pred } xs))) (\text{lsplitRemainder pred } xs) \wedge$

$(\forall y \in \text{lset } (\text{lsplitRemainder pred } xs). \neg \text{pred } y)$

<proof>

lemmas *lsplit-lsplitRemainder* = *lsplitRemainder*[*THEN conjunct1*]

lemmas *lset-lsplitRemainder* = *lsplitRemainder*[*THEN conjunct2, rule-format*]

2.12 The first index for which a predicate holds (if any)

definition *firstHolds where*

firstHolds pred xs $\equiv \text{length } (\text{ltakeUntil pred } xs) - 1$

lemma *firstHolds-eq-0*:

assumes $\exists x \in \text{lset } xs. \text{pred } x$

shows $\text{firstHolds pred } xs = 0 \iff \text{pred } (\text{lhd } xs)$

<proof>

lemma *firstHolds-eq-0'*:

assumes $\neg \text{lnever pred } xs$

shows $\text{firstHolds pred } xs = 0 \iff \text{pred } (\text{lhd } xs)$

<proof>

lemma *firstHolds-Suc*:
assumes $\exists x \in \text{lset } xs. \text{pred } x$ **and** $\neg \text{pred } (\text{lhd } xs)$
shows $\text{firstHolds pred } xs = \text{Suc } (\text{firstHolds pred } (\text{ttl } xs))$
 $\langle \text{proof} \rangle$

lemma *firstHolds-Suc'*:
assumes $\neg \text{lnever pred } xs$ **and** $\neg \text{pred } (\text{lhd } xs)$
shows $\text{firstHolds pred } xs = \text{Suc } (\text{firstHolds pred } (\text{ttl } xs))$
 $\langle \text{proof} \rangle$

lemma *firstHolds-append*:
assumes $\neg \text{lnever pred } xs$ **and** $\text{never pred } ys$
shows $\text{firstHolds pred } (\text{lappend } (\text{llist-of } ys) xs) = \text{length } ys + \text{firstHolds pred } xs$
 $\langle \text{proof} \rangle$

2.13 The first index for which the list in a lazy-list of lists is non-empty

definition *firstNC* **where**
 $\text{firstNC } xss \equiv \text{firstHolds } (\lambda xs. xs \neq []) xss$

lemma *firstNC-eq-0*:
assumes $\exists xs \in \text{lset } xss. xs \neq []$
shows $\text{firstNC } xss = 0 \iff \text{lhd } xss \neq []$
 $\langle \text{proof} \rangle$

lemma *firstNC-Suc*:
assumes $\exists xs \in \text{lset } xss. xs \neq []$ **and** $\text{lhd } xss = []$
shows $\text{firstNC } xss = \text{Suc } (\text{firstNC } (\text{ttl } xss))$
 $\langle \text{proof} \rangle$

lemma *firstNC-LCons-notNil*: $xs \neq [] \implies \text{firstNC } (xs \$ xss) = 0$
 $\langle \text{proof} \rangle$

lemma *firstNC-LCons-Nil*:
 $(\exists ys \in \text{lset } xss. ys \neq []) \implies xs = [] \implies \text{firstNC } (xs \$ xss) = \text{Suc } (\text{firstNC } xss)$
 $\langle \text{proof} \rangle$

end

3 Filtermap for Lazy Lists

theory *LazyList-Filtermap*
imports *LazyList-Operations List-Filtermap*
begin

This theory defines the filtermap operator for lazy lists, proves its basic properties, and proves a coinductive criterion for the euqlity of two filtermapped

lazy lsits.

3.1 Lazy lists filtermap

definition *lfiltermap* ::
(*'trans* \Rightarrow *bool*) \Rightarrow (*'trans* \Rightarrow *'a*) \Rightarrow *'trans llist* \Rightarrow *'a llist*
where
lfiltermap pred func tr \equiv *lmap func (lfilter pred tr)*

lemmas *lfiltermap-lmap-lfilter* = *lfiltermap-def*

lemma *lfiltermap-lappend*: *lfinite tr* \implies *lfiltermap pred func (lappend tr tr1)* =
lappend (lfiltermap pred func tr) (lfiltermap pred func tr1)
(*proof*)

lemma *lfiltermap-LNil-never*: *lfiltermap pred func tr* = $[\]$ \longleftrightarrow *lnever pred tr*
(*proof*)

lemma *llength-lfiltermap*: *llength (lfiltermap pred func tr)* \leq *llength tr*
(*proof*)

lemma *lfiltermap-llist-all[simp]*: *lfinite tr* \implies *lfiltermap pred func tr* = *lmap func tr*
 \longleftrightarrow *llist-all pred tr*
(*proof*)

lemma *lfilter-LNil-never*: $[\]$ = *lfilter pred xs* \implies *lnever pred xs*
(*proof*)

lemma *lnever-LNil-lfilter*: *lnever pred xs* \longleftrightarrow $[\]$ = *lfilter pred xs*
(*proof*)

lemma *lfilter-LNil-never'*: *lfilter pred xs* = $[\]$ \implies *lnever pred xs*
(*proof*)

lemma *lnever-LNil-lfilter'*: *lnever pred xs* \longleftrightarrow *lfilter pred xs* = $[\]$
(*proof*)

lemma *lfiltermap-LCons2-eq*:
lfiltermap pred func [[x, x']] = *lfiltermap pred func [[y, y']]*
 \implies *lfiltermap pred func (x \$ x' \$ zs)* = *lfiltermap pred func (y \$ y' \$ zs)*
(*proof*)

lemma *lfiltermap-LCons-cong*:
lfiltermap pred func xs = *lfiltermap pred func ys*
 \implies *lfiltermap pred func (x \$ xs)* = *lfiltermap pred func (x \$ ys)*
(*proof*)

lemma *lfiltermap-LCons-eq*:
lfiltermap pred func xs = *lfiltermap pred func ys*

$\implies \text{pred } x \longleftrightarrow \text{pred } y$
 $\implies \text{pred } x \longrightarrow \text{func } x = \text{func } y$
 $\implies \text{lfiltermap pred func } (x \$ xs) = \text{lfiltermap pred func } (y \$ ys)$
 <proof>

lemma *set-lfiltermap*:

$\text{lset } (\text{lfiltermap pred func } xs) \subseteq \{\text{func } x \mid x . x \in \text{lset } xs \wedge \text{pred } x\}$
 <proof>

lemma *lfinite-lfiltermap-filtermap*:

$\text{lfinite } xs \implies \text{lfiltermap pred func } xs = \text{llist-of } (\text{filtermap pred func } (\text{list-of } xs))$
 <proof>

lemma *lfiltermap-llist-of-filtermap*:

$\text{lfiltermap pred func}(\text{llist-of } xs) = \text{llist-of } (\text{filtermap pred func } xs)$
 <proof>

lemma *filtermap-butlast*: $xs \neq [] \implies$

$\neg \text{pred } (\text{last } xs) \implies$
 $\text{filtermap pred func } xs = \text{filtermap pred func } (\text{butlast } xs)$
 <proof>

lemma *filtermap-butlast'*:

$xs \neq [] \implies \text{pred } (\text{last } xs) \implies$
 $\text{filtermap pred func } xs = \text{filtermap pred func } (\text{butlast } xs) @ [\text{func } (\text{last } xs)]$
 <proof>

lemma *lfinite-lfiltermap-butlast*: $xs \neq [[]] \implies (\text{lfinite } xs \implies \neg \text{pred } (\text{llast } xs)) \implies$

$\text{lfiltermap pred func } xs = \text{lfiltermap pred func } (\text{lbutlast } xs)$
 <proof>

lemma *last-filtermap*: $xs \neq [] \implies \text{pred } (\text{last } xs) \implies$

$\text{filtermap pred func } xs \neq [] \wedge \text{last } (\text{filtermap pred func } xs) = \text{func } (\text{last } xs)$
 <proof>

lemma *filtermap-ltakeUntil[simp]*:

$\exists x \in \text{lset } xs. \text{pred } x \implies \text{filtermap pred func } (\text{ltakeUntil pred } xs) = [\text{func } (\text{last } (\text{ltakeUntil pred } xs))]$
 <proof>

lemma *last-ltakeUntil-filtermap[simp]*:

$\exists x \in \text{lset } xs. \text{pred } x \implies \text{func } (\text{last } (\text{ltakeUntil pred } xs)) = \text{lhd } (\text{lfiltermap pred func } xs)$
 <proof>

lemma *lfiltermap-lmap-filtermap-lsplit*:

assumes $lfiltermap\ pred\ func\ xs = lfiltermap\ pred\ func\ ys$
shows $lmap\ (filtermap\ pred\ func)\ (lsplit\ pred\ xs) = lmap\ (filtermap\ pred\ func)\ (lsplit\ pred\ ys)$
 $\langle proof \rangle$

lemma $lfiltermap-lfinite-lsplit$:
assumes $lfiltermap\ pred\ func\ xs = lfiltermap\ pred\ func\ ys$
shows $lfinite\ (lsplit\ pred\ xs) \longleftrightarrow lfinite\ (lsplit\ pred\ ys)$
 $\langle proof \rangle$

lemma $lfiltermap-lsplitRemainder[simp]$: $lfiltermap\ pred\ func\ (lsplitRemainder\ pred\ xs) = []$
 $\langle proof \rangle$

lemma $lfiltermap-lconcat-lsplit$:
 $lfiltermap\ pred\ func\ xs =$
 $lfiltermap\ pred\ func\ (lconcat\ (lmap\ llist-of\ (lsplit\ pred\ xs)))$
 $\langle proof \rangle$

lemma $lfilter-lconcat-lfinite'$: $(\bigwedge i. i < llength\ yss \implies lfinite\ (lnth\ yss\ i))$
 $\implies lfilter\ pred\ (lconcat\ yss) = lconcat\ (lmap\ (lfilter\ pred)\ yss)$
 $\langle proof \rangle$

lemma $lfilter-lconcat-llist-of$:
 $lfilter\ pred\ (lconcat\ (lmap\ llist-of\ yss)) = lconcat\ (lmap\ (lfilter\ pred)\ (lmap\ llist-of\ yss))$
 $\langle proof \rangle$

lemma $lfiltermap-lconcat-lmap-llist-of$:
 $lfiltermap\ pred\ func\ (lconcat\ (lmap\ llist-of\ yss)) =$
 $lconcat\ (lmap\ (lfiltermap\ pred\ func)\ (lmap\ llist-of\ yss))$
 $\langle proof \rangle$

lemma $filtermap-noteq-imp-lsplit$:
assumes $len: llength\ (lsplit\ pred\ xs) = llength\ (lsplit\ pred\ xs')$
and $l: lfiltermap\ pred\ func\ xs \neq lfiltermap\ pred\ func\ xs'$
shows $\exists i0 < llength\ (lsplit\ pred\ xs).$
 $filtermap\ pred\ func\ (lnth\ (lsplit\ pred\ xs)\ i0) \neq$
 $filtermap\ pred\ func\ (lnth\ (lsplit\ pred\ xs')\ i0)$
 $\langle proof \rangle$

3.2 Coinductive criterion for filtermap equality

We work in a locale that fixes two function-predicate pairs, for performing two instances of filtermap. We will give criteria for when the two filtermap applications to two lazy lists are equal.

locale $TwoFuncPred =$
fixes $pred :: 'a \Rightarrow bool$ **and** $pred' :: 'a' \Rightarrow bool$
and $func :: 'a \Rightarrow 'b$ **and** $func' :: 'a' \Rightarrow 'b$

begin

lemma *LCons-eq-lmap-lfilter*:

assumes *LCons* *b* *bss* = *lmap* *func* (*lfilter* *pred* *as*)

shows \exists *as1* *a* *ass*.

$as = \text{lappend} (\text{llist-of } as1) (LCons\ a\ ass) \wedge$

$\text{never } pred\ as1 \wedge pred\ a \wedge func\ a = b \wedge$

$bss = \text{lmap } func\ (\text{lfilter } pred\ ass)$

<proof>

lemma *LCons-eq-lmap-lfilter'*:

assumes *LCons* *b* *bss* = *lmap* *func'* (*lfilter* *pred'* *as*)

shows \exists *as1* *a* *ass*.

$as = \text{lappend} (\text{llist-of } as1) (LCons\ a\ ass) \wedge$

$\text{never } pred'\ as1 \wedge pred'\ a \wedge func'\ a = b \wedge$

$bss = \text{lmap } func'\ (\text{lfilter } pred'\ ass)$

<proof>

lemma *lmap-lfilter-lappend-lnever*:

assumes *P*: *P* *lxs* *lxs'*

and *lappend*:

\bigwedge *lxs* *lxs'*. *P* *lxs* *lxs'* \implies

$\text{lmap } func\ (\text{lfilter } pred\ lxs) = \text{lmap } func'\ (\text{lfilter } pred'\ lxs') \vee$

$(\exists$ *lxs* *lxs'* *lxs''*.

$lxs \neq [] \wedge lxs' \neq [] \wedge$

$\text{map } func\ (\text{filter } pred\ lxs) = \text{map } func'\ (\text{filter } pred'\ lxs') \wedge$

$lxs = \text{lappend} (\text{llist-of } lxs) lxs'' \wedge lxs' = \text{lappend} (\text{llist-of } lxs') lxs'' \wedge$

$P\ lxs\ lxs''$)

shows *lnever* *pred* *lxs* = *lnever* *pred'* *lxs'*

<proof>

lemma *lmap-lfilter-lappend-makeStronger*:

assumes *lappend*:

\bigwedge *lxs* *lxs'*. *P* *lxs* *lxs'* \implies

$\text{lmap } func\ (\text{lfilter } pred\ lxs) = \text{lmap } func'\ (\text{lfilter } pred'\ lxs') \vee$

$(\exists$ *lxs* *lxs'* *lxs''*.

$lxs \neq [] \wedge lxs' \neq [] \wedge$

$\text{map } func\ (\text{filter } pred\ lxs) = \text{map } func'\ (\text{filter } pred'\ lxs') \wedge$

$lxs = \text{lappend} (\text{llist-of } lxs) lxs'' \wedge lxs' = \text{lappend} (\text{llist-of } lxs') lxs'' \wedge$

$P\ lxs\ lxs''$)

and *P*: *P* *lxs* *lxs'*

shows *lmap* *func* (*lfilter* *pred* *lxs*) = *lmap* *func'* (*lfilter* *pred'* *lxs'*) \vee

$(\exists$ *lxs* *lxs'* *lxs''*.

$\text{map } func\ (\text{filter } pred\ lxs) \neq [] \wedge$

$\text{map } func\ (\text{filter } pred\ lxs) = \text{map } func'\ (\text{filter } pred'\ lxs') \wedge$

$lxs = \text{lappend} (\text{llist-of } lxs) lxs'' \wedge lxs' = \text{lappend} (\text{llist-of } lxs') lxs'' \wedge$

$P\ lxs\ lxs''$)

<proof>

proposition *lmap-lfilter-lappend-coind*:

assumes $P: P\ lxs\ lxs'$

and *lappend*:

$\bigwedge lxs\ lxs'. P\ lxs\ lxs' \implies$

$lmap\ func\ (lfilter\ pred\ lxs) = lmap\ func'\ (lfilter\ pred'\ lxs') \vee$

$(\exists\ ys\ llxs\ ys'\ llxs'.$

$ys \neq [] \wedge ys' \neq [] \wedge$

$map\ func\ (filter\ pred\ ys) = map\ func'\ (filter\ pred'\ ys') \wedge$

$lxs = lappend\ (l\ list\ of\ ys)\ llxs \wedge lxs' = lappend\ (l\ list\ of\ ys')\ llxs' \wedge$

$P\ llxs\ llxs')$

shows $lmap\ func\ (lfilter\ pred\ lxs) = lmap\ func'\ (lfilter\ pred'\ lxs')$

<proof>

proposition *lmap-lfilter-lappend-coind-wf*:

assumes $W: wf\ W$ **and** $P: P\ w\ lxs\ lxs'$

and *lappend*:

$\bigwedge w\ lxs\ lxs'. P\ w\ lxs\ lxs' \implies$

$lmap\ func\ (lfilter\ pred\ lxs) = lmap\ func'\ (lfilter\ pred'\ lxs') \vee$

$(\exists\ v\ ys\ llxs\ ys'\ llxs'.$

$ys \neq [] \wedge ys' \neq [] \vee (v, w) \in W) \wedge$

$map\ func\ (filter\ pred\ ys) = map\ func'\ (filter\ pred'\ ys') \wedge$

$lxs = lappend\ (l\ list\ of\ ys)\ llxs \wedge lxs' = lappend\ (l\ list\ of\ ys')\ llxs' \wedge$

$P\ v\ llxs\ llxs')$

shows $lmap\ func\ (lfilter\ pred\ lxs) = lmap\ func'\ (lfilter\ pred'\ lxs')$

<proof>

proposition *lmap-lfilter-lappend-coind-wf2*:

assumes $W1: wf\ (W1::'a1\ rel)$ **and** $W2: wf\ (W2::'a2\ rel)$

and $P: P\ w1\ w2\ lxs\ lxs'$

and *lappend*:

$\bigwedge w1\ w2\ lxs\ lxs'. P\ w1\ w2\ lxs\ lxs' \implies$

$lmap\ func\ (lfilter\ pred\ lxs) = lmap\ func'\ (lfilter\ pred'\ lxs') \vee$

$(\exists\ v1\ v2\ ys\ llxs\ ys'\ llxs'.$

$((v1, w1) \in W1 \vee ys \neq []) \wedge ((v2, w2) \in W2 \vee ys' \neq []) \wedge$

$map\ func\ (filter\ pred\ ys) = map\ func'\ (filter\ pred'\ ys') \wedge$

$lxs = lappend\ (l\ list\ of\ ys)\ llxs \wedge lxs' = lappend\ (l\ list\ of\ ys')\ llxs' \wedge$

$P\ v1\ v2\ llxs\ llxs')$

shows $lmap\ func\ (lfilter\ pred\ lxs) = lmap\ func'\ (lfilter\ pred'\ lxs')$

<proof>

3.3 A concrete instantiation of the criterion

coinductive *sameFM* :: *enat* \Rightarrow *enat* \Rightarrow 'a *l*list \Rightarrow 'a' *l*list \Rightarrow *bool* **where**

LNil:

sameFM *wL* *wR* [] []

|

Singl:

(*pred* *a* \longleftrightarrow *pred'* *a'*) \implies (*pred* *a* \longrightarrow *func* *a* = *func'* *a'*) \implies *sameFM* *wL* *wR* [[*a*]]
[[*a'*]]

|

lappend:

(*xs* \neq [] \vee *vL* < *wL*) \implies (*xs'* \neq [] \vee *vR* < *wR*) \implies
map func (filter pred xs) = *map func' (filter pred' xs')* \implies
sameFM vL vR as as'
 \implies *sameFM wL wR (lappend (l*list-of *xs*) *as*) (lappend (llist-of *xs'*) *as')*

|

lmap-lfilter:

lmap func (lfilter pred as) = *lmap func' (lfilter pred' as')* \implies
sameFM wL wR as as'

proposition *sameFM-lmap-lfilter*:

assumes *sameFM wL wR as as'*

shows *lmap func (lfilter pred as)* = *lmap func' (lfilter pred' as')*

<proof>

end

end