More Operations on Lazy Lists

Andrei Popescu Jamie Wright

March 19, 2025

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

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 equqlity of two filtermapped lazy lsits.

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* $\Longrightarrow$ *(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*]:
¬ *pred x* ⟹ *filtermap pred func* (*x* # *xs*) = *filtermap pred func xs*
⟨*proof*⟩

lemma *filtermap-Cons*[*simp*]:
pred x ⟹ *filtermap pred func* (*x* # *xs*) = *func x* # *filtermap pred func xs*
⟨*proof*⟩

lemma *filtermap-append*: *filtermap pred func* (*xs* @ *xs1*) = *filtermap pred func xs* @ *filtermap pred func xs1*
⟨*proof*⟩

lemma *filtermap-Nil-list-ex*: *filtermap pred func xs* = [] ⟷ ¬ *list-ex pred xs*
⟨*proof*⟩

lemma *filtermap-Nil-never*: *filtermap pred func xs* = [] ⟷ *never pred xs*
⟨*proof*⟩

lemma *length-filtermap*: *length* (*filtermap pred func xs*) ≤ *length xs*
⟨*proof*⟩

lemma *filtermap-list-all*[*simp*]: *filtermap pred func xs* = *map func xs* ⟷ *list-all pred xs*
⟨*proof*⟩

lemma *filtermap-eq-Cons*:
assumes *filtermap pred func xs* = *a* # *al1*
shows ∃ *x xs2 xs1*.
xs = *xs2* @ [*x*] @ *xs1* ∧ *never pred xs2* ∧ *pred x* ∧ *func x* = *a* ∧ *filtermap pred func xs1* = *al1*
⟨*proof*⟩

lemma *filtermap-eq-append*:
assumes *filtermap pred func xs* = *al1* @ *al2*
shows ∃ *xs1 xs2*. *xs* = *xs1* @ *xs2* ∧ *filtermap pred func xs1* = *al1* ∧ *filtermap pred func xs2* = *al2*
⟨*proof*⟩

lemma *holds-filtermap-RCons*[*simp*]:
pred x ⟹ *filtermap pred func* (*xs* ## *x*) = *filtermap pred func xs* ## *func x*
⟨*proof*⟩

lemma *not-holds-filtermap-RCons*[*simp*]:
¬ *pred x* ⟹ *filtermap pred func* (*xs* ## *x*) = *filtermap pred func xs*
⟨*proof*⟩

lemma *filtermap-eq-RCons*:
assumes *filtermap pred func xs = al1 ## a*
shows $\exists$ *x xs1 xs2*.
xs = xs1 @ [x] @ xs2 ∧ never pred xs2 ∧ pred x ∧ func x = a ∧ filtermap pred func xs1 = al1
⟨*proof*⟩

lemma *filtermap-eq-Cons-RCons*:
assumes *filtermap pred func xs = a # al1 ## b*
shows $\exists$ *xsa xa xs1 xb xsb*.
xs = xsa @ [xa] @ xs1 @ [xb] @ xsb ∧
never pred xsa ∧
pred xa ∧ func xa = a ∧
filtermap pred func xs1 = al1 ∧
pred xb ∧ func xb = b ∧
never pred xsb
⟨*proof*⟩

lemma *filter-Nil-never*: [] = *filter pred xs* ⟹ *never pred xs*
⟨*proof*⟩

lemma *never-Nil-filter*: *never pred xs* ⟷ [] = *filter pred xs*
⟨*proof*⟩

lemma *set-filtermap*:
set (*filtermap pred func xs*) ⊆ {*func x* | *x* . *x* ∈ *set xs* ∧ *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* ⟹ *enat i < j*
⟨*proof*⟩

abbreviation *LNil-abbr* (‹[]›) **where** *LNil-abbr* ≡ *LNil*

abbreviation *LCons-abbr* (**infixr** ‹$› *65*) **where** *x $ xs* ≡ *LCons x xs*

abbreviation *lnever* :: (*'a* ⇒ *bool*) ⇒ *'a llist* ⇒ *bool* **where** *lnever U* ≡ *llist-all* (λ *a*. ¬ *U a*)

syntax
— llist Enumeration
-llist :: *args* => *'a llist* (‹[[(-)]]›)

syntax-consts
-llist == *LCons*

translations
[[*x*, *xs*]] == *x* $ [[*xs*]]
[[*x*]] == *x* $ [[]]

declare *llist-of-eq-LNil-conv*[*simp*]
declare *lmap-eq-LNil*[*simp*]
declare *llength-ltl*[*simp*]

2.2 More properties of operators from the Coinductive library

lemma *lnth-lconcat*:
assumes *i* < *llength* (*lconcat xss*)
shows ∃ *j*<*llength xss*. ∃ *k*<*llength* (*lnth xss j*). *lnth* (*lconcat xss*) *i* = *lnth* (*lnth xss j*) *k*
⟨*proof*⟩

lemma *lnth-0-lset*: *xs* ≠ [[]] ⟹ *lnth xs 0* ∈ *lset xs*
⟨*proof*⟩

lemma *lconcat-eq-LNil-iff*: *lconcat xss* = [[]] ⟷ (∀ *xs*∈*lset xss*. *xs* = [[]])
⟨*proof*⟩

lemma *llast-last-llist-of*: *lfinite xs* ⟹ *llast xs* = *last* (*list-of xs*)
⟨*proof*⟩

lemma *lappend-llist-of-inj*:
length xs = *length ys* ⟹
lappend (*llist-of xs*) *as* = *lappend* (*llist-of ys*) *bs* ⟷ *xs* = *ys* ∧ *as* = *bs*
⟨*proof*⟩

lemma *llist-all-lnth*: *llist-all P xs* = (∀ *n*<*llength xs*. *P* (*lnth xs n*))
⟨*proof*⟩

lemma *llist-eq-cong*:
assumes *llength xs* = *llength ys* ⋀*i*. *i* < *llength xs* ⟹ *lnth xs i* = *lnth ys i*

shows $xs = ys$
⟨*proof*⟩

lemma *llist-cases*: $llength\ xs = \infty \vee (\exists\, ys.\ xs = llist\text{-}of\ ys)$
⟨*proof*⟩

lemma *llist-all-lappend*: $lfinite\ xs \Longrightarrow$
$llist\text{-}all\ pred\ (lappend\ xs\ ys) \longleftrightarrow llist\text{-}all\ pred\ xs \wedge llist\text{-}all\ pred\ ys$
⟨*proof*⟩

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

lemma *llist-all-conduct*:
$X\ xs \Longrightarrow$
$(\bigwedge xs.\ X\ xs \Longrightarrow \neg\ lnull\ xs \Longrightarrow P\ (lhd\ xs) \wedge (X\ (ltl\ xs) \vee llist\text{-}all\ P\ (ltl\ xs))) \Longrightarrow$
$llist\text{-}all\ P\ xs$
⟨*proof*⟩

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

lemma *ldrop-Suc*: $n < llength\ xs \Longrightarrow ldrop\ (enat\ n)\ xs = LCons\ (lnth\ xs\ n)\ (ldrop\ (enat\ (Suc\ n))\ xs)$
⟨*proof*⟩

lemma *lappend-ltake-lnth-ldrop*: $n < llength\ xs \Longrightarrow$
$lappend\ (ltake\ (enat\ n)\ xs)\ (LCons\ (lnth\ xs\ n)\ (ldrop\ (enat\ (Suc\ n))\ xs)) = xs$
⟨*proof*⟩

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

lemma *ex-llength-infty*:
$\exists\, a.\ llength\ a = \infty \wedge lhd\ a = 0$
⟨*proof*⟩

lemma *repeat-not-Nil*[*simp*]: $repeat\ a \neq [[]]$
⟨*proof*⟩

2.3 A convenient adaptation of the lazy-list corecursor

definition *ccorec-llist* :: $('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b\ llist) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'b\ llist$
where $ccorec\text{-}llist\ isn\ h\ ec\ e\ t \equiv$
$corec\text{-}llist\ isn\ (\lambda a.\ if\ ec\ a\ then\ lhd\ (e\ a)\ else\ h\ a)\ ec\ (\lambda a.\ case\ e\ a\ of\ b\ \$\ a' \Rightarrow a')\ t$

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

lemma *llist-ccorec-LCons*:
¬ *lnull* (*e a*) $\Longrightarrow$ ¬ *isn a* $\Longrightarrow$
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′* $\Longrightarrow$
lxs = *lxs′* ∨
(∃ *ys llxs llxs′*. *ys* ≠ [] ∧
lxs = *lappend* (*llist-of ys*) *llxs* ∧ *lxs′* = *lappend* (*llist-of ys*) *llxs′* ∧
P llxs llxs′)
shows *lxs* = *lxs′*
⟨*proof*⟩

2.5 Interval lazy lists

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

primcorec *betw* :: *nat* ⇒ *enat* ⇒ *nat llist* **where**
betw i n = (*if i* ≥ *n then LNil else i* $ *betw* (*Suc i*) *n*)

lemma *betw-more-simps*:
¬ *n* ≤ *i* $\Longrightarrow$ *betw i n* = *i* $ *betw* (*Suc i*) *n*
⟨*proof*⟩

lemma *lhd-betw*: *i* < *n* $\Longrightarrow$ *lhd* (*betw i n*) = *i*
⟨*proof*⟩

lemma *not-lfinite-betw-infty*: ¬ *lfinite* (*betw i* ∞)
⟨*proof*⟩

lemma *llength-betw-infty*[*simp*]: *llength* (*betw i* ∞) = ∞
⟨*proof*⟩

lemma *llength-betw*: *llength* (*betw i n*) = *n* − *i*

⟨*proof*⟩

lemma *lfinite-betw-not-infty*: $n < \infty \Longrightarrow$ *lfinite* (*betw i n*)
⟨*proof*⟩

lemma *lfinite-betw-enat*: *lfinite* (*betw i* (*enat n*))
⟨*proof*⟩

lemma *lnth-betw*: *enat j* $<$ *n* $-$ *i* $\Longrightarrow$ *lnth* (*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* $\equiv$ *lmap f* (*betw 0 n*)

lemma *llength-build*[*simp*]: *llength* (*build n f*) $=$ *n*
⟨*proof*⟩

lemma *lnth-build*[*simp*]: *i* $<$ *n* $\Longrightarrow$ *lnth* (*build n f*) *i* $=$ *f i*
⟨*proof*⟩

lemma *build-lnth*[*simp*]: *build* (*llength xs*) (*lnth xs*) $=$ *xs*
⟨*proof*⟩

lemma *build-eq-LNil*[*simp*]: *build n f* $=$ [[]] $\longleftrightarrow$ *n* $=$ *0*
⟨*proof*⟩

2.7 The butlast (reverse tail) operation

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

lemma *llength-lbutlast-lfinite*[*simp*]:
sl $\neq$ [[]] $\Longrightarrow$ *lfinite sl* $\Longrightarrow$ *llength* (*lbutlast sl*) $=$ *llength sl* $-$ *1*
⟨*proof*⟩

lemma *llength-lbutlast-not-lfinite*[*simp*]:
$\neg$ *lfinite sl* $\Longrightarrow$ *llength* (*lbutlast sl*) $= \infty$
⟨*proof*⟩

lemma *lbutlast-LNil*[*simp*]:
lbutlast [[]] $=$ [[]]
⟨*proof*⟩

lemma *lbutlast-singl*[*simp*]:
lbutlast [[*s*]] $=$ [[]]
⟨*proof*⟩

lemma *lbutlast-lfinite*[*simp*]:
lfinite sl $\Longrightarrow$ *lbutlast sl* = *llist-of* (*butlast* (*list-of sl*))
⟨*proof*⟩

lemma *lbutlast-Cons*[*simp*]: *tr* $\neq$ [[]] $\Longrightarrow$ *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* $\Longrightarrow$ *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* $\Longrightarrow$ *butlast* (*list-of xs*) = *list-of* (*lbutlast xs*)
⟨*proof*⟩

lemma *butlast-length-le1*[*simp*]: *llength xs* $\leq$ *Suc 0* $\Longrightarrow$ *lbutlast xs* = [[]]
⟨*proof*⟩

lemma *llength-lbutlast*[*simp*]: *llength* (*lbutlast tr*) = *llength tr* − *1*
⟨*proof*⟩

lemma *lnth-lbutlast*: *i* < *llength xs* − *1* $\Longrightarrow$ *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 llist-of xss*) **and** *Suc i* < *llength xss*
shows *lsublist* (*llist-of* (*append* (*lnth xss i*) (*lnth xss* (*Suc i*)))) *xs*
⟨*proof*⟩

lemma *lnth-lconcat-lconcat-lsublist*:
assumes *xs*: *xs* = *lappend* (*lconcat* (*lmap llist-of xss*)) *ys* **and** *i* < *llength xss*
shows *lsublist* (*llist-of* (*lnth xss i*)) *xs*
⟨*proof*⟩

lemma *lnth-lconcat-lconcat-lsublist2*:
assumes *xs*: *xs* = *lappend* (*lconcat* (*lmap llist-of xss*)) *ys* **and** *Suc i* < *llength xss*
shows *lsublist* (*llist-of* (*append* (*lnth xss i*) (*lnth xss* (*Suc i*)))) *xs*
⟨*proof*⟩

lemma *su-lset-lconcat-llist-of*:
assumes *xs* ∈ *lset xss*
shows *set xs* ⊆ *lset* (*lconcat* (*lmap llist-of xss*))
⟨*proof*⟩

lemma *lsublist-lnth-lconcat*: *i* < *llength tr1s* ⟹ *lsublist* (*llist-of* (*lnth tr1s i*)) (*lconcat* (*lmap llist-of tr1s*))
⟨*proof*⟩

lemma *lsublist-lset*:
lsublist xs ys ⟹ *lset xs* ⊆ *lset ys*
⟨*proof*⟩

lemma *lsublist-LNil*:
lsublist xs ys ⟹ *ys* = *LNil* ⟹ *xs* = *LNil*
⟨*proof*⟩

2.9 Take-until and drop-until

definition *ltakeUntil* :: ('*a* ⇒ *bool*) ⇒ '*a llist* ⇒ '*a list* **where**
ltakeUntil pred xs ≡
list-of (*lappend* (*ltakeWhile* (λ*x*. ¬ *pred x*) *xs*) [[*lhd* (*ldropWhile* (λ*x*. ¬ *pred x*) *xs*)]])

definition *ldropUntil* :: ('*a* ⇒ *bool*) ⇒ '*a llist* ⇒ '*a llist* **where**
ldropUntil pred xs ≡ *ltl* (*ldropWhile* (λ*x*. ¬ *pred x*) *xs*)

lemma *lappend-ltakeUntil-ldropUntil*:
∃ *x*∈*lset xs*. *pred x* ⟹ *lappend* (*llist-of* (*ltakeUntil pred xs*)) (*ldropUntil pred xs*) = *xs*
⟨*proof*⟩

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

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

lemma *ltakeUntil-never-butlast*:
assumes $\exists\, x{\in}lset\ xs.\ pred\ x$
shows $never\ pred\ (butlast\ (ltakeUntil\ pred\ xs))$
⟨*proof*⟩

lemma *ltakeUntil-last*:
assumes $\exists\, x{\in}lset\ xs.\ pred\ x$
shows $pred\ (last\ (ltakeUntil\ pred\ xs))$
⟨*proof*⟩

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

lemma *ltakeUntil-LCons1*[*simp*]: $\exists\, x{\in}lset\ xs.\ pred\ x \Longrightarrow \neg\ pred\ x \Longrightarrow ltakeUntil\ pred\ (LCons\ x\ xs) = x\ \#\ ltakeUntil\ pred\ xs$
⟨*proof*⟩

lemma *ldropUntil-LCons1*[*simp*]: $\exists\, x{\in}lset\ xs.\ pred\ x \Longrightarrow \neg\ pred\ x \Longrightarrow$
$ldropUntil\ pred\ (LCons\ x\ xs) = ldropUntil\ pred\ xs$
⟨*proof*⟩

lemma *ltakeUntil-LCons2*[*simp*]: $\exists\, x{\in}lset\ xs.\ pred\ x \Longrightarrow pred\ x \Longrightarrow ltakeUntil\ pred\ (LCons\ x\ xs) = [x]$
⟨*proof*⟩

lemma *ldropUntil-LCons2*[*simp*]: $\exists\, x{\in}lset\ xs.\ pred\ x \Longrightarrow pred\ x \Longrightarrow ldropUntil\ pred\ (LCons\ x\ xs) = xs$
⟨*proof*⟩

lemma *ltakeUntil-tl1*[*simp*]:
$\exists\, x{\in}lset\ xs.\ pred\ x \Longrightarrow \neg\ pred\ (lhd\ xs) \Longrightarrow ltakeUntil\ pred\ (ltl\ xs) = tl\ (ltakeUntil\ pred\ xs)$
⟨*proof*⟩

lemma *ldropUntil-tl1*[*simp*]:

$\exists x{\in}lset\ xs.\ pred\ x \Longrightarrow \neg\ pred\ (lhd\ xs) \Longrightarrow ldropUntil\ pred\ (ltl\ xs) = ldropUntil\ pred\ xs$
⟨*proof*⟩

lemma *ltakeUntil-tl2*[*simp*]:
$xs \neq [[]] \Longrightarrow pred\ (lhd\ xs) \Longrightarrow tl\ (ltakeUntil\ pred\ xs) = []$
⟨*proof*⟩

lemma *ldropUntil-tl2*[*simp*]:
$xs \neq [[]] \Longrightarrow pred\ (lhd\ xs) \Longrightarrow ldropUntil\ pred\ xs = ltl\ xs$
⟨*proof*⟩

lemma *LCons-lfilter-ldropUntil*: $y\ \$\ ys = lfilter\ pred\ xs \Longrightarrow ys = lfilter\ pred\ (ldropUntil\ pred\ xs)$
⟨*proof*⟩

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

lemma *length-ltakeUntil-eq-1*:
assumes $\exists x{\in}lset\ xs.\ pred\ x$
shows $length\ (ltakeUntil\ pred\ xs) = Suc\ 0 \longleftrightarrow pred\ (lhd\ xs)$
⟨*proof*⟩

lemma *length-ltakeUntil-Suc*:
assumes $\exists x{\in}lset\ xs.\ pred\ x\ \neg\ pred\ (lhd\ xs)$
shows $length\ (ltakeUntil\ pred\ xs) = Suc\ (length\ (ltakeUntil\ pred\ (ltl\ xs)))$
⟨*proof*⟩

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

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

declare *lsplit.ctr*[*simp*]

lemma *infinite-split*[*simp*]:
$infinite\ \{x \in lset\ xs.\ pred\ x\} \Longrightarrow lsplit\ pred\ xs = LCons\ (ltakeUntil\ pred\ xs)\ (lsplit\ pred\ (ldropUntil\ pred\ xs))$
⟨*proof*⟩

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

⟨*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 ∃ *ys*. *xs* = *lappend* (*lconcat* (*lmap llist-of* (*lsplit pred xs*))) *ys* ∧ (∀ *y*∈*lset ys*. ¬ *pred y*)
⟨*proof*⟩

lemma *lconcat-lsplit*:
∃ *ys*. *xs* = *lappend* (*lconcat* (*lmap llist-of* (*lsplit pred xs*))) *ys* ∧ (∀ *y*∈*lset ys*. ¬ *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*:
llist-all (λ*zs*. *zs* ≠ [] ∧ *list-all* (λ*z*. ¬ *pred z*) (*butlast zs*) ∧ *pred* (*last zs*))
(*lsplit pred xs*)
⟨*proof*⟩

lemma *lsplit-main-lset*:
assumes *ys* ∈ *lset* (*lsplit pred xs*)
shows *ys* ≠ [] ∧
list-all (λ*z*. ¬ *pred z*) (*butlast ys*) ∧
pred (*last ys*)
⟨*proof*⟩

lemma *lsplit-main-lnth*:
assumes *i* < *llength* (*lsplit pred xs*)
shows *lnth* (*lsplit pred xs*) *i* ≠ [] ∧
list-all (λ*z*. ¬ *pred z*) (*butlast* (*lnth* (*lsplit pred xs*) *i*)) ∧
pred (*last* (*lnth* (*lsplit pred xs*) *i*))
⟨*proof*⟩

lemma *hd-lhd-lsplit*: ∃ *x*∈*lset xs*. *pred x* ⟹ *hd* (*lhd* (*lsplit pred xs*)) = *lhd xs*

⟨*proof*⟩

lemma *lprefix-lsplit*:
assumes ∃ *x*∈*lset xs*. *pred x*
shows *lprefix* (*llist-of* (*lhd* (*lsplit pred xs*))) *xs*
⟨*proof*⟩

lemma *lprefix-lsplit-lbutlast*:
assumes ∃ *x*∈*lset xs*. *pred x*
shows *lprefix* (*llist-of* (*butlast* (*lhd* (*lsplit pred xs*)))) (*lbutlast xs*)
⟨*proof*⟩

lemma *set-lset-lsplit*:
assumes *ys* ∈ *lset* (*lsplit pred xs*)
shows *set ys* ⊆ *lset xs*
⟨*proof*⟩

lemma *set-lnth-lsplit*:
assumes *i* < *llength* (*lsplit pred xs*)
shows *set* (*lnth* (*lsplit pred xs*) *i*) ⊆ *lset xs*
⟨*proof*⟩

2.11 The split remainder

definition *lsplitRemainder pred xs* ≡ *SOME ys*. *xs* = *lappend* (*lconcat* (*lmap llist-of* (*lsplit pred xs*))) *ys* ∧ (∀ *y*∈*lset ys*. ¬ *pred y*)

lemma *lsplitRemainder*:
xs = *lappend* (*lconcat* (*lmap llist-of* (*lsplit pred xs*))) (*lsplitRemainder pred xs*) ∧ (∀ *y*∈*lset* (*lsplitRemainder pred xs*). ¬ *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 ≡ *length* (*ltakeUntil pred xs*) − *1*

lemma *firstHolds-eq-0*:
assumes ∃ *x*∈*lset xs*. *pred x*
shows *firstHolds pred xs* = *0* ⟷ *pred* (*lhd xs*)
⟨*proof*⟩

lemma *firstHolds-eq-0′*:
assumes ¬ *lnever pred xs*
shows *firstHolds pred xs* = *0* ⟷ *pred* (*lhd xs*)
⟨*proof*⟩

lemma *firstHolds-Suc*:
assumes $\exists\, x{\in}lset\ xs.\ pred\ x$ **and** $\neg\ pred\ (lhd\ xs)$
shows $firstHolds\ pred\ xs = Suc\ (firstHolds\ pred\ (ltl\ xs))$
⟨*proof*⟩

lemma *firstHolds-Suc'*:
assumes $\neg\ lnever\ pred\ xs$ **and** $\neg\ pred\ (lhd\ xs)$
shows $firstHolds\ pred\ xs = Suc\ (firstHolds\ pred\ (ltl\ xs))$
⟨*proof*⟩

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

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

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

lemma *firstNC-eq-0*:
assumes $\exists\, xs{\in}lset\ xss.\ xs \neq []$
shows $firstNC\ xss = 0 \longleftrightarrow lhd\ xss \neq []$
⟨*proof*⟩

lemma *firstNC-Suc*:
assumes $\exists\, xs{\in}lset\ xss.\ xs \neq []$ **and** $lhd\ xss = []$
shows $firstNC\ xss = Suc\ (firstNC\ (ltl\ xss))$
⟨*proof*⟩

lemma *firstNC-LCons-notNil*: $xs \neq [] \Longrightarrow firstNC\ (xs\ \$\ xss) = 0$
⟨*proof*⟩

lemma *firstNC-LCons-Nil*:
$(\exists\, ys{\in}lset\ xss.\ ys \neq []) \Longrightarrow xs = [] \Longrightarrow firstNC\ (xs\ \$\ xss) = Suc\ (firstNC\ xss)$
⟨*proof*⟩

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*: $lfinite\ tr \Longrightarrow lfiltermap\ pred\ func\ (lappend\ tr\ tr1) =$
$lappend\ (lfiltermap\ pred\ func\ tr)\ (lfiltermap\ pred\ func\ tr1)$
⟨*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*]: $lfinite\ tr \Longrightarrow lfiltermap\ pred\ func\ tr = lmap\ func\ tr \longleftrightarrow llist\text{-}all\ pred\ tr$
⟨*proof*⟩

lemma *lfilter-LNil-lnever*: $[[]] = lfilter\ pred\ xs \Longrightarrow lnever\ pred\ xs$
⟨*proof*⟩

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

lemma *lfilter-LNil-lnever'*: $lfilter\ pred\ xs = [[]] \Longrightarrow lnever\ pred\ xs$
⟨*proof*⟩

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

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

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

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

$\Longrightarrow$ *pred x* $\longleftrightarrow$ *pred y*
$\Longrightarrow$ *pred x* $\longrightarrow$ *func x* = *func y*
$\Longrightarrow$ *lfiltermap pred func* (*x* $ *xs*) = *lfiltermap pred func* (*y* $ *ys*)
⟨*proof*⟩

lemma *set-lfiltermap*:
lset (*lfiltermap pred func xs*) ⊆ {*func x* | *x* . *x* ∈ *lset xs* ∧ *pred x*}
⟨*proof*⟩

lemma *lfinite-lfiltermap-filtermap*:
lfinite xs $\Longrightarrow$ *lfiltermap pred func xs* = *llist-of* (*filtermap pred func* (*list-of xs*))
⟨*proof*⟩

lemma *lfiltermap-llist-of-filtermap*:
lfiltermap pred func(*llist-of xs*) = *llist-of* (*filtermap pred func xs*)
⟨*proof*⟩

lemma *filtermap-butlast*: *xs* ≠ [] $\Longrightarrow$
¬ *pred* (*last xs*) $\Longrightarrow$
filtermap pred func xs = *filtermap pred func* (*butlast xs*)
⟨*proof*⟩

lemma *filtermap-butlast'*:
xs ≠ [] $\Longrightarrow$ *pred* (*last xs*) $\Longrightarrow$
filtermap pred func xs = *filtermap pred func* (*butlast xs*) @ [*func* (*last xs*)]
⟨*proof*⟩

lemma *lfinite-lfiltermap-butlast*: *xs* ≠ [[]] $\Longrightarrow$ (*lfinite xs* $\Longrightarrow$ ¬ *pred* (*llast xs*)) $\Longrightarrow$

lfiltermap pred func xs = *lfiltermap pred func* (*lbutlast xs*)
⟨*proof*⟩

lemma *last-filtermap*: *xs* ≠ [] $\Longrightarrow$ *pred* (*last xs*) $\Longrightarrow$
filtermap pred func xs ≠ [] ∧ *last* (*filtermap pred func xs*) = *func* (*last xs*)
⟨*proof*⟩

lemma *filtermap-ltakeUntil*[*simp*]:
∃ *x*∈*lset xs*. *pred x* $\Longrightarrow$ *filtermap pred func* (*ltakeUntil pred xs*) = [*func* (*last* (*ltakeUntil pred xs*))]
⟨*proof*⟩

lemma *last-ltakeUntil-filtermap*[*simp*]:
∃ *x*∈*lset xs*. *pred x* $\Longrightarrow$ *func* (*last* (*ltakeUntil pred xs*)) = *lhd* (*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)*
⟨*proof*⟩

lemma *lfiltermap-lfinite-lsplit*:
assumes *lfiltermap pred func xs = lfiltermap pred func ys*
shows *lfinite (lsplit pred xs) ⟷ lfinite (lsplit pred ys)*
⟨*proof*⟩

lemma *lfiltermap-lsplitRemainder*[*simp*]: *lfiltermap pred func (lsplitRemainder pred xs) = [[]]*
⟨*proof*⟩

lemma *lfiltermap-lconcat-lsplit*:
lfiltermap pred func xs =
lfiltermap pred func (lconcat (lmap llist-of (lsplit pred xs)))
⟨*proof*⟩

lemma *lfilter-lconcat-lfinite'*: *(⋀i. i < llength yss ⟹ lfinite (lnth yss i)) ⟹ lfilter pred (lconcat yss) = lconcat (lmap (lfilter pred) yss)*
⟨*proof*⟩

lemma *lfilter-lconcat-llist-of*:
lfilter pred (lconcat (lmap llist-of yss)) = lconcat (lmap (lfilter pred) (lmap llist-of yss))
⟨*proof*⟩

lemma *lfiltermap-lconcat-lmap-llist-of*:
lfiltermap pred func (lconcat (lmap llist-of yss)) =
lconcat (lmap (llist-of o filtermap pred func) yss)
⟨*proof*⟩

lemma *filtermap-noteq-imp-lsplit*:
assumes *len*: *llength (lsplit pred xs) = llength (lsplit pred xs')*
and *l*: *lfiltermap pred func xs ≠ lfiltermap pred func xs'*
shows ∃ *i0*<*llength (lsplit pred xs)*.
filtermap pred func (lnth (lsplit pred xs) i0) ≠
filtermap pred func (lnth (lsplit pred xs') i0)
⟨*proof*⟩

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* ⇒ *bool* **and** *pred'* :: *'a'* ⇒ *bool*
and *func* :: *'a* ⇒ *'b* **and** *func'* :: *'a'* ⇒ *'b*

begin

lemma *LCons-eq-lmap-lfilter*:
assumes *LCons b bss* = *lmap func* (*lfilter pred as*)
shows ∃ *as1 a ass*.
as = *lappend* (*llist-of as1*) (*LCons a ass*) ∧
never pred as1 ∧ *pred a* ∧ *func a* = *b* ∧
bss = *lmap func* (*lfilter pred ass*)
⟨*proof*⟩

lemma *LCons-eq-lmap-lfilter'*:
assumes *LCons b bss* = *lmap func'* (*lfilter pred' as*)
shows ∃ *as1 a ass*.
as = *lappend* (*llist-of as1*) (*LCons a ass*) ∧
never pred' as1 ∧ *pred' a* ∧ *func' a* = *b* ∧
bss = *lmap func'* (*lfilter pred' ass*)
⟨*proof*⟩

lemma *lmap-lfilter-lappend-lnever*:
assumes *P*: *P lxs lxs'*
and *lappend*:
⋀*lxs lxs'*. *P lxs lxs'* ⟹
lmap func (*lfilter pred lxs*) = *lmap func'* (*lfilter pred' lxs'*) ∨
(∃ *ys llxs ys' llxs'*.
ys ≠ [] ∧ *ys'* ≠ [] ∧
map func (*filter pred ys*) = *map func'* (*filter pred' ys'*) ∧
lxs = *lappend* (*llist-of ys*) *llxs* ∧ *lxs'* = *lappend* (*llist-of ys'*) *llxs'* ∧
P llxs llxs')
shows *lnever pred lxs* = *lnever pred' lxs'*
⟨*proof*⟩

lemma *lmap-lfilter-lappend-makeStronger*:
assumes *lappend*:
⋀*lxs lxs'*. *P lxs lxs'* ⟹
lmap func (*lfilter pred lxs*) = *lmap func'* (*lfilter pred' lxs'*) ∨
(∃ *ys llxs ys' llxs'*.
ys ≠ [] ∧ *ys'* ≠ [] ∧
map func (*filter pred ys*) = *map func'* (*filter pred' ys'*) ∧
lxs = *lappend* (*llist-of ys*) *llxs* ∧ *lxs'* = *lappend* (*llist-of ys'*) *llxs'* ∧
P llxs llxs')
and *P*: *P lxs lxs'*
shows *lmap func* (*lfilter pred lxs*) = *lmap func'* (*lfilter pred' lxs'*) ∨
(∃ *ys llxs ys' llxs'*.
map func (*filter pred ys*) ≠ [] ∧
map func (*filter pred ys*) = *map func'* (*filter pred' ys'*) ∧
lxs = *lappend* (*llist-of ys*) *llxs* ∧ *lxs'* = *lappend* (*llist-of ys'*) *llxs'* ∧
P llxs llxs')
⟨*proof*⟩

proposition *lmap-lfilter-lappend-coind*:
assumes *P*: *P lxs lxs'*
and *lappend*:
$\bigwedge$*lxs lxs'*. *P lxs lxs'* $\Longrightarrow$
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* (*llist-of ys*) *llxs* $\wedge$ *lxs'* = *lappend* (*llist-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'* $\Longrightarrow$
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* (*llist-of ys*) *llxs* $\wedge$ *lxs'* = *lappend* (*llist-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'* $\Longrightarrow$
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* (*llist-of ys*) *llxs* $\wedge$ *lxs'* = *lappend* (*llist-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* ⇒ *enat* ⇒ *'a llist* ⇒ *'a' llist* ⇒ *bool* **where**
LNil:
sameFM wL wR [[]] [[]]
|
Singl:
(*pred a* ⟷ *pred' a'*) ⟹ (*pred a* ⟶ *func a* = *func' a'*) ⟹ *sameFM wL wR* [[*a*]]
[[*a'*]]
|
lappend:
(*xs* ≠ [] ∨ *vL* < *wL*) ⟹ (*xs'* ≠ [] ∨ *vR* < *wR*) ⟹
map func (*filter pred xs*) = *map func'* (*filter pred' xs'*) ⟹
sameFM vL vR as as'
⟹ *sameFM wL wR* (*lappend* (*llist-of xs*) *as*) (*lappend* (*llist-of xs'*) *as'*)
|
lmap-lfilter:
lmap func (*lfilter pred as*) = *lmap func'* (*lfilter pred' as'*) ⟹
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