More Operations on Lazy Lists

Andrei Popescu   Jamie Wright

June 5, 2024

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* ⇒ *bool*) ⇒ '*a list* ⇒ *bool* **where** *never U* ≡ *list-all* (λ *a*. ¬ *U a*)

lemma *never-list-ex*: *never pred xs* ⟷ ¬ *list-ex pred xs*
by (*induction xs*) *auto*

abbreviation *Rcons* (**infix** ## *70*) **where** *xs* ## *x* ≡ *xs* @ [*x*]

lemma *two-singl-Rcons*: [*a*,*b*] = [*a*] ## *b* **by** *auto*

lemma *length-gt-1-Cons-snoc*:
  assumes *length ys* > *1*
  obtains *x1 xs x2* **where** *ys* = *x1* # *xs* ## *x2*
using *assms*
proof (*cases ys*)
  case (*Cons x1 xs*)
    with *assms* **obtain** *xs′ x2* **where** *xs* = *xs′* ## *x2* **by** (*cases xs rule*: *rev-cases*) *auto*
    with *Cons* **show** *thesis* **by** (*intro that*) *auto*
qed *auto*

lemma *right-cons-left*[*simp*]: *i* < *length as* ⟹ (*as* ## *a*)!*i* = *as*!*i*
by (*metis butlast-snoc nth-butlast*)+

1.2 Filtermap

definition *filtermap* :: ('*b* ⇒ *bool*) ⇒ ('*b* ⇒ '*a*) ⇒ '*b list* ⇒ '*a list* **where**
filtermap pred func xs ≡ *map func* (*filter pred xs*)

lemma *filtermap-Nil*[*simp*]:
filtermap pred func [] = []
unfolding *filtermap-def* **by** *auto*

lemma *filtermap-Cons-not*[*simp*]:
¬ *pred x* ⟹ *filtermap pred func* (*x* # *xs*) = *filtermap pred func xs*
unfolding *filtermap-def* **by** *auto*

lemma *filtermap-Cons*[*simp*]:
pred x ⟹ *filtermap pred func* (*x* # *xs*) = *func x* # *filtermap pred func xs*
unfolding *filtermap-def* **by** *auto*

lemma *filtermap-append*: *filtermap pred func* (*xs* @ *xs1*) = *filtermap pred func xs* @ *filtermap pred func xs1*
proof(*induction xs arbitrary*: *xs1*)
  case (*Cons x xs*)
  thus *?case* **by** (*cases pred x*) *auto*
qed *auto*

lemma *filtermap-Nil-list-ex*: *filtermap pred func xs* = [] ⟷ ¬ *list-ex pred xs*
proof(*induction xs*)
  case (*Cons x xs*)
  thus *?case* **by** (*cases pred x*) *auto*
qed *auto*

lemma *filtermap-Nil-never*: *filtermap pred func xs* = [] ⟷ *never pred xs*
proof(*induction xs*)
  case (*Cons x xs*)
  thus *?case* **by** (*cases pred x*) *auto*
qed *auto*

lemma *length-filtermap*: *length* (*filtermap pred func xs*) ≤ *length xs*
proof(*induction xs*)
  case (*Cons x xs*)
  thus *?case* **by** (*cases pred x*) *auto*
qed *auto*

lemma *filtermap-list-all*[*simp*]: *filtermap pred func xs* = *map func xs* ⟷ *list-all pred xs*
proof(*induction xs*)
  case (*Cons x xs*)
  thus *?case* **apply** (*cases pred x*)
  by (*simp-all*) (*metis impossible-Cons length-filtermap length-map*)
qed *auto*

```
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
using assms proof(induction xs arbitrary: a al1)
  case (Cons x xs a al1)
  show ?case
  proof(cases pred x)
    case False
    hence filtermap pred func xs = a # al1 using Cons by simp
    from Cons(1)[OF this] obtain xn xs2 xs1 where
    1: xs = xs2 @ [xn] @ xs1 ∧ never pred xs2 ∧ pred xn ∧ func xn = a ∧
     filtermap pred func xs1 = al1 by blast
    show ?thesis apply(rule exI[of - xn], rule exI[of - x # xs2], rule exI[of - xs1])
    using Cons(2) 1 False by simp
  next
    case True
    hence filtermap pred func xs = al1 using Cons by simp
    show ?thesis apply(rule exI[of - x], rule exI[of - []], rule exI[of - xs])
    using Cons(2) True by simp
  qed
qed auto

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
using assms proof(induction al1 arbitrary: xs)
  case Nil show ?case
  apply (rule exI[of - []], rule exI[of - xs]) using Nil by auto
next
  case (Cons a al1 xs)
  hence filtermap pred func xs = a # (al1 @ al2) by simp
  from filtermap-eq-Cons[OF this] obtain x xs2 xs1
  where xs: xs = xs2 @ [x] @ xs1 and n: never pred xs2 ∧ pred x ∧ func x = a
  and f: filtermap pred func xs1 = al1 @ al2 by blast
  from Cons(1)[OF f] obtain xs11 xs22 where xs1: xs1 = xs11 @ xs22
  and f1: filtermap pred func xs11 = al1 and f2: filtermap pred func xs22 = al2
by blast
  show ?case apply (rule exI[of - xs2 @ [x] @ xs11], rule exI[of - xs22])
  using n filtermap-Nil-never f1 f2 unfolding xs xs1 filtermap-append by auto
qed

lemma holds-filtermap-RCons[simp]:
pred x ⟹ filtermap pred func (xs ## x) = filtermap pred func xs ## func x
proof(induction xs)
  case (Cons x xs)
  thus ?case by (cases pred x) auto
```

qed *auto*

lemma *not-holds-filtermap-RCons*[*simp*]:
¬ *pred x* ⟹ *filtermap pred func* (*xs* ## *x*) = *filtermap pred func xs*
proof(*induction xs*)
  case (*Cons x xs*)
  thus *?case* **by** (*cases pred x*) *auto*
qed *auto*

lemma *filtermap-eq-RCons*:
assumes *filtermap pred func xs* = *al1* ## *a*
shows ∃ *x xs1 xs2*.
  xs = *xs1* @ [*x*] @ *xs2* ∧ *never pred xs2* ∧ *pred x* ∧ *func x* = *a* ∧ *filtermap pred func xs1* = *al1*
using *assms* **proof**(*induction xs arbitrary*: *a al1 rule*: *rev-induct*)
  case (*snoc x xs a al1*)
  show *?case*
  proof(*cases pred x*)
    case *False*
    hence *filtermap pred func xs* = *al1* ## *a* **using** *snoc* **by** *simp*
    from *snoc*(*1*)[*OF this*] **obtain** *xn xs2 xs1* **where**
    1: *xs* = *xs1* @ [*xn*] @ *xs2* ∧ *never pred xs2* ∧ *pred xn* ∧ *func xn* = *a* ∧
     filtermap pred func xs1 = *al1* **by** *blast*
    show *?thesis* **apply**(*rule exI*[*of - xn*], *rule exI*[*of - xs1*], *rule exI*[*of - xs2* ## *x*])
    using *snoc*(*2*) *1 False* **by** *simp*
  next
    case *True*
    hence *filtermap pred func xs* = *al1* **using** *snoc* **by** *simp*
    show *?thesis* **apply**(*rule exI*[*of - x*], *rule exI*[*of - xs*], *rule exI*[*of -* []])
    using *snoc*(*2*) *True* **by** *simp*
  qed
qed *auto*

lemma *filtermap-eq-Cons-RCons*:
assumes *filtermap pred func xs* = *a* # *al1* ## *b*
shows ∃ *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−
  from *filtermap-eq-Cons*[*OF assms*] **obtain** *xa xsa xs2*
  where *0*: *xs* = *xsa* @ [*xa*] @ *xs2* ∧ *never pred xsa* ∧ *pred xa* ∧ *func xa* = *a*
  and *1*: *filtermap pred func xs2* = *al1* ## *b* **by** *auto*
  from *filtermap-eq-RCons*[*OF 1*] **obtain** *xb xs1 xsb* **where**
  2: *xs2* = *xs1* @ [*xb*] @ *xsb* ∧ *never pred xsb* ∧

pred xb ∧ *func xb* = *b* ∧ *filtermap pred func xs1* = *al1* **by** *blast*
show *?thesis* **apply**(*rule exI*[*of - xsa*], *rule exI*[*of - xa*], *rule exI*[*of - xs1*],
rule exI[*of - xb*], *rule exI*[*of - xsb*])
using *2 0* **by** *auto*
qed

lemma *filter-Nil-never*: [] = *filter pred xs* ⟹ *never pred xs*
by (*induction xs*) (*auto split*: *if-splits*)

lemma *never-Nil-filter*: *never pred xs* ⟷ [] = *filter pred xs*
by (*induction xs*) (*auto split*: *if-splits*)

lemma *set-filtermap*:
set (*filtermap pred func xs*) ⊆ {*func x* | *x* . *x* ∈ *set xs* ∧ *pred x*}
proof(*induction xs*)
case (*Cons x xs*)
thus *?case* **by** (*cases pred x*) *auto*
qed *auto*

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*
by (*metis co.enat.exhaust eSuc-minus-1 idiff-0 iless-Suc-eq less-imp-le*)

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*   ([[(-)]])

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 $\exists$ *j*<*llength xss*. $\exists$ *k*<*llength* (*lnth xss j*). *lnth* (*lconcat xss*) *i* = *lnth* (*lnth xss j*) *k*
using *assms lnth-lconcat-conv* **by** *blast*

lemma *lnth-0-lset*: *xs* $\neq$ [[]] $\Longrightarrow$ *lnth xs 0* $\in$ *lset xs*
by (*metis llist.set-sel*(*1*) *lnth-0-conv-lhd lnull-def*)

lemma *lconcat-eq-LNil-iff*: *lconcat xss* = [[]] $\longleftrightarrow$ ($\forall$ *xs*$\in$*lset xss*. *xs* = [[]])
by (*metis lnull-def lnull-lconcat mem-Collect-eq subset-eq*)

lemma *llast-last-llist-of*: *lfinite xs* $\Longrightarrow$ *llast xs* = *last* (*list-of xs*)
by (*metis llast-llist-of llist-of-list-of*)

lemma *lappend-llist-of-inj*:
length xs = *length ys* $\Longrightarrow$
lappend (*llist-of xs*) *as* = *lappend* (*llist-of ys*) *bs* $\longleftrightarrow$ *xs* = *ys* $\wedge$ *as* = *bs*
apply(*induct xs ys arbitrary*: *as bs rule*: *list-induct2*) **by** *auto*

lemma *llist-all-lnth*: *llist-all P xs* = ($\forall$ *n*<*llength xs*. *P* (*lnth xs n*))
by (*metis in-lset-conv-lnth llist.pred-set*)

lemma *llist-eq-cong*:
assumes *llength xs* = *llength ys* $\bigwedge$*i*. *i* < *llength xs* $\Longrightarrow$ *lnth xs i* = *lnth ys i*
shows *xs* = *ys*
proof−
have *llength xs* = *llength ys* $\wedge$ ($\forall$ *i*. *i* < *llength xs* $\longrightarrow$ *lnth xs i* = *lnth ys i*)
using *assms* **by** *auto*
thus *?thesis* **apply**(*coinduct rule*: *llist.coinduct*)
by *simp* (*metis lhd-conv-lnth linorder-not-less llength-eq-0 lnth-beyond lnth-ltl*)
qed

lemma *llist-cases*: *llength xs* = ∞ $\vee$ ($\exists$ *ys*. *xs* = *llist-of ys*)
by (*metis llist-of-list-of not-lfinite-llength*)

lemma *llist-all-lappend*: *lfinite xs* $\Longrightarrow$

llist-all pred (*lappend xs ys*) ⟷ *llist-all pred xs* ∧ *llist-all pred ys*
unfolding *llist.pred-set* **by** (*auto simp add*: *in-lset-lappend-iff*)

lemma *llist-all-lappend-llist-of*:
llist-all pred (*lappend* (*llist-of xs*) *ys*) ⟷ *list-all pred xs* ∧ *llist-all pred ys*
by (*metis lfinite-llist-of list-all-iff llist.pred-set llist-all-lappend lset-llist-of*)

lemma *llist-all-conduct*:
X xs ⟹
(⋀*xs*. *X xs* ⟹ ¬ *lnull xs* ⟹ *P* (*lhd xs*) ∧ (*X* (*ltl xs*) ∨ *llist-all P* (*ltl xs*))) ⟹
llist-all P xs
unfolding *llist.pred-rel* **apply**(*coinduct rule*: *llist-all2-coinduct*[*of* λ*xs ys*. *X xs* ∧ *xs* = *ys*])
by (*auto simp*: *eq-onp-same-args*)

lemma *lfilter-lappend-llist-of*:
lfilter P (*lappend* (*llist-of xs*) *ys*) = *lappend* (*llist-of* (*filter P xs*)) (*lfilter P ys*)
by *simp*

lemma *ldrop-Suc*: *n* < *llength xs* ⟹ *ldrop* (*enat n*) *xs* = *LCons* (*lnth xs n*) (*ldrop* (*enat* (*Suc n*)) *xs*)
apply(*rule llist-eq-cong*)
subgoal apply(*subst llength-ldrop*) **apply** *simp* **apply**(*subst llength-ldrop*)
using *llist-cases*[*of xs*] **by** (*auto simp*: *eSuc-enat*)
subgoal for *i* **apply**(*subst lnth-ldrop*)
subgoal by (*metis add.commute ldrop-eq-LNil ldrop-ldrop linorder-not-less*)
subgoal apply(*subst lnth-LCons*)
by (*metis* ‹⟦*enat n* < *llength xs*; *enat i* < *llength* (*ldrop* (*enat n*) *xs*)⟧ ⟹ *enat n* + *enat i* < *llength xs*› *ldrop-enat ldropn-Suc-conv-ldropn lnth-LCons lnth-ldrop*) **. .**

lemma *lappend-ltake-lnth-ldrop*: *n* < *llength xs* ⟹
lappend (*ltake* (*enat n*) *xs*) (*LCons* (*lnth xs n*) (*ldrop* (*enat* (*Suc n*)) *xs*)) = *xs*
by (*simp add*: *ldrop-enat ldropn-Suc-conv-ldropn*)

lemma *ltake-eq-LNil*: *ltake i tr* = [[]] ⟷ *i* = *0* ∨ *tr* = [[]]
by (*metis LNil-eq-ltake-iff*)

lemma *ex-llength-infty*:
∃ *a*. *llength a* = ∞ ∧ *lhd a* = *0*
by (*meson lhd-iterates llength-iterates*)

lemma *repeat-not-Nil*[*simp*]: *repeat a* ≠ [[]]
by (*metis lfinite-LNil lfinite-iterates*)

2.3 A convenient adaptation of the lazy-list corecursor

definition *ccorec-llist* :: (*'a* ⇒ *bool*) ⇒ (*'a* ⇒ *'b*) ⇒ (*'a* ⇒ *bool*) ⇒ (*'a* ⇒ *'b llist*) ⇒ (*'a* ⇒ *'a*) ⇒ *'a* ⇒ *'b llist*

where *ccorec-llist isn h ec e t* ≡
corec-llist isn (*λa. if ec a then lhd* (*e a*) *else h a*) *ec* (*λa. case e a of b* $ *a′* ⇒ *a′*) *t*

lemma *llist-ccorec-LNil*: *isn a* ⟹ *ccorec-llist isn h ec e t a* = [[]]
unfolding *ccorec-llist-def llist.corec*(*1*) **by** *auto*

lemma *llist-ccorec-LCons*:
¬ *lnull* (*e a*) ⟹ ¬ *isn a* ⟹
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*))
unfolding *ccorec-llist-def llist.corec*(*2*)
by (*cases e a*, *auto simp*: *lnull-def*)

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*:
⋀*lxs lxs′*. *P lxs lxs′* ⟹
  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−
 have *l1*: *llength lxs* ≤ *llength lxs′*
 proof(*cases lfinite lxs′*)
  case *False* **thus** *?thesis* **by** (*simp add*: *not-lfinite-llength*)
 next
  case *True*
  then obtain *xs′* **where** *lxs′*: *lxs′* = *llist-of xs′*
  by (*metis llist-of-list-of*)
   show *?thesis* **using** *P* **unfolding** *lxs′* **proof**(*induct xs′ arbitrary*: *lxs rule*: *length-induct*)
   case (*1 xs′ lxs*)
   show *?case* **using** *lappend*[*OF 1*(*2*)] **apply**(*elim disjE exE conjE*)
    subgoal by *simp*
    subgoal for *ys llxs llxs′* **using** *1*(*1*)[*rule-format*, *of list-of llxs′ llxs*]
    by *simp* (*metis length-append length-greater-0-conv less-add-same-cancel2*
     lfinite-lappend lfinite-llist-of list-of-lappend list-of-llist-of llength-llist-of llist-of-list-of) **.**
  qed
 qed

```
  have l2: llength lxs' ≤ llength lxs
  proof(cases lfinite lxs)
    case False thus ?thesis by (simp add: not-lfinite-llength)
  next
    case True
    then obtain xs where lxs: lxs = llist-of xs
    by (metis llist-of-list-of)
     show ?thesis using P unfolding lxs proof(induct xs arbitrary: lxs' rule:
length-induct)
      case (1 xs lxs')
      show ?case using lappend[OF 1(2)] apply(elim disjE exE conjE)
        subgoal by simp
        subgoal for ys llxs llxs' using 1(1)[rule-format, of list-of llxs llxs']
        by simp (metis length-append length-greater-0-conv less-add-same-cancel2
            lfinite-lappend lfinite-llist-of list-of-lappend list-of-llist-of llength-llist-of
llist-of-list-of) .
    qed
  qed

  from l1 l2 have l: llength lxs = llength lxs' by simp
  show ?thesis proof(rule llist-eq-cong)
    show llength lxs = llength lxs' by fact
  next
    fix i assume i: enat i < llength lxs

    show lnth lxs i = lnth lxs' i
    using P l i proof(induct i arbitrary: lxs lxs' rule: less-induct)
      case (less i lxs lxs')
      show ?case using lappend[OF less(2)] proof(elim disjE exE conjE)
        fix ys llxs llxs'
        assume ys: ys ≠ [] and P:  P llxs llxs'
        and lxs: lxs = lappend (llist-of ys) llxs
        and lxs': lxs' = lappend (llist-of ys) llxs'

        show lnth lxs i = lnth lxs' i
        proof(cases i < length ys)
          case True
          hence lnth lxs i = ys ! i lnth lxs' i = ys ! i unfolding lxs lxs'
          by (auto simp: lnth-lappend-llist-of)
          thus ?thesis by simp
        next
          case False
          then obtain j where i: i = length ys + j
          using le-Suc-ex not-le-imp-less by blast
          hence j: j < llength lxs j < llength lxs'
          by (metis dual-order.strict-trans enat-ord-simps(2)
                  length-greater-0-conv less.prems(2,3) less-add-same-cancel2 ys)+
         hence 0: lnth lxs i = lnth llxs j lnth lxs' i = lnth llxs' j unfolding lxs lxs'
```

unfolding *i* **by** (*auto simp*: *lnth-lappend-llist-of*)
show *?thesis* **unfolding** *0* **apply**(*rule less*(*1*)[*rule-format*, *of j llxs llxs'*])
subgoal by (*simp add*: *i ys*)
subgoal by (*simp add*: *P*)
subgoal using *less.prems*(*2*) *lxs lxs'* **by** *auto*
subgoal by (*metis enat-add-mono i less.prems*(*3*) *llength-lappend llength-llist-of lxs plus-enat-simps*(*1*)) **.**
qed
qed *auto*
qed
qed
qed

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*:
$\neg\ n \leq i \Longrightarrow betw\ i\ n = i\ \$\ betw\ (Suc\ i)\ n$
using *betw.ctr*(*2*) *enat-ord-simps*(*1*) **by** *blast*

lemma *lhd-betw*: $i < n \Longrightarrow lhd\ (betw\ i\ n) = i$
by *fastforce*

lemma *not-lfinite-betw-infty*: $\neg\ lfinite\ (betw\ i\ \infty)$
proof−
{**fix** *js* **assume** $lfinite\ js\ js = betw\ i\ \infty$
hence *False*
apply (*induct arbitrary*: *i*)
subgoal by (*metis betw.disc*(*2*) *enat-ord-code*(*5*) *llist.disc*(*1*))
subgoal by (*metis betw.sel*(*2*) *enat-ord-code*(*5*) *llist.sel*(*3*)) **.**
}
thus *?thesis* **by** *auto*
qed

lemma *llength-betw-infty*[*simp*]: $llength\ (betw\ i\ \infty) = \infty$
using *not-lfinite-betw-infty not-lfinite-llength* **by** *blast*

lemma *llength-betw*: $llength\ (betw\ i\ n) = n - i$
apply(*cases n*)
subgoal for *nn* **apply** *simp* **apply**(*induct nn−i arbitrary*: *i*, *auto*)
apply (*simp add*: *zero-enat-def*)
by (*metis betw.ctr*(*2*) *diff-Suc-1 diff-Suc-eq-diff-pred diff-commute eSuc-enat enat-ord-simps*(*1*) *less-le-not-le llength-LCons zero-less-Suc zero-less-diff*)
subgoal by *simp* **.**

lemma *lfinite-betw-not-infty*: $n < \infty \Longrightarrow$ *lfinite* (*betw i n*)
using *lfinite-conv-llength-enat llength-betw* **by** *fastforce*

lemma *lfinite-betw-enat*: *lfinite* (*betw i* (*enat n*))
using *lfinite-conv-llength-enat llength-betw* **by** *fastforce*

lemma *lnth-betw*: *enat* $j < n - i \Longrightarrow$ *lnth* (*betw i n*) $j = i + j$
apply(*induct j arbitrary*: *i*, *auto*)
apply (*metis betw.ctr*(*1*) *betw.disc-iff*(*1*) *betw.simps*(*3*) *enat-0 llength-LNil llength-betw lnth-0-conv-lhd zero-less-iff-neq-zero*)
by (*metis Suc-ile-eq add-Suc betw.ctr*(*1*) *betw.ctr*(*2*) *betw.disc*(*2*) *betw.sel*(*2*) *iless-Suc-eq*
llength-LCons llength-LNil llength-betw lnth-ltl not-less-zero)

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*
by (*simp add*: *build-def llength-betw*)

lemma *lnth-build*[*simp*]: $i < n \Longrightarrow$ *lnth* (*build n f*) *i* = *f i*
by (*simp add*: *build-def llength-betw lnth-betw*)

lemma *build-lnth*[*simp*]: *build* (*llength xs*) (*lnth xs*) = *xs*
by (*metis* (*mono-tags*, *lifting*) *llength-build llist.rel-eq llist-all2-all-lnthI lnth-build*)

lemma *build-eq-LNil*[*simp*]: *build n f* = [[]] $\longleftrightarrow$ *n* = *0*
by (*metis llength-build llength-eq-0 lnull-def*)

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*
unfolding *lbutlast-def*
by *simp* (*metis One-nat-def idiff-enat-enat length-list-of one-enat-def*)

lemma *llength-lbutlast-not-lfinite*[*simp*]:
$\neg$ *lfinite sl* $\Longrightarrow$ *llength* (*lbutlast sl*) = ∞
unfolding *lbutlast-def* **using** *not-lfinite-llength* **by** *auto*

lemma *lbutlast-LNil*[*simp*]:
lbutlast [[]] = [[]]
unfolding *lbutlast-def* **by** *auto*

```
lemma lbutlast-singl[simp]:
lbutlast [[s]] = [[]]
unfolding lbutlast-def by auto

lemma lbutlast-lfinite[simp]:
lfinite sl ⟹ lbutlast sl = llist-of (butlast (list-of sl))
unfolding lbutlast-def by auto

lemma lbutlast-Cons[simp]: tr ≠ [[]] ⟹ lbutlast (s $ tr) = s $ lbutlast tr
unfolding lbutlast-def using llist-of-list-of by fastforce

lemma llist-of-butlast: llist-of (butlast xs) = lbutlast (llist-of xs)
by simp

lemma lprefix-lbutlast: lprefix xs ys ⟹ lprefix (lbutlast xs) (lbutlast ys)
apply(cases lfinite xs)
  subgoal apply(cases lfinite ys)
    subgoal
    by simp (smt (verit, ccfv-threshold) butlast-append lfinite-lappend list-of-lappend

      lprefix-conv-lappend prefix-append prefix-order.eq-iff prefixeq-butlast)
      subgoal by (metis lbutlast-def llist-of-list-of lprefix-llist-of lprefix-trans pre-
fixeq-butlast) .
  by (simp add: not-lfinite-lprefix-conv-eq)

lemma lbutlast-lappend:
assumes (ys::'a llist) ≠ [[]] shows lbutlast (lappend xs ys) = lappend xs (lbutlast
ys)
proof−
  {fix us vs :: 'a llist
   assume ∃ xs ys. ys ≠ [[]] ∧ us = lbutlast (lappend xs ys) ∧ vs = lappend xs
(lbutlast ys)
   hence us = vs
   apply(coinduct rule: llist.coinduct)
   by (smt (z3) eq-LConsD lappend.disc-iff(2) lappend-code(2) lappend-eq-LNil-iff
lappend-lnull1
    lappend-snocL1-conv-LCons2 lbutlast-Cons lbutlast-singl lhd-LCons lhd-LCons-ltl
lnull-def
      lnull-lprefix lprefix-code(2) ltl-simps(1) ltl-simps(2) not-lnull-conv)
  }
  thus ?thesis using assms by blast
qed

lemma lbutlast-llist-of: lbutlast (llist-of xs) = llist-of (butlast xs)
by auto

lemma butlast-list-of: lfinite xs ⟹ butlast (list-of xs) = list-of (lbutlast xs)
by simp
```

lemma *butlast-length-le1* [*simp*]: *llength xs* ≤ *Suc 0* ⟹ *lbutlast xs* = [[]]
by (*metis One-nat-def antisym-conv2 enat-ile epred-1 epred-conv-minus iless-Suc-eq lbutlast-LNil le-zero-eq lfinite-conv-llength-enat llength-eq-0 llength-lbutlast-lfinite lnull-def one-eSuc one-enat-def*)

lemma *llength-lbutlast*[*simp*]: *llength* (*lbutlast tr*) = *llength tr* − *1*
by (*metis idiff-0 idiff-infinity lbutlast-LNil llength-LNil llength-lbutlast-lfinite llength-lbutlast-not-lfinite not-lfinite-llength*)

lemma *lnth-lbutlast*: *i* < *llength xs* − *1* ⟹ *lnth* (*lbutlast xs*) *i* = *lnth xs i*
unfolding *lbutlast-def*
by *simp* (*metis enat-ord-simps*(*2*) *llength-lbutlast llength-llist-of llist-of-butlast llist-of-list-of nth-butlast nth-list-of*)

2.8 Consecutive-elements sublists

definition *lsublist xs ys* ≡ ∃ *us vs*. *lfinite us* ∧ *ys* = *lappend us* (*lappend xs vs*)

lemma *lsublist-refl*: *lsublist xs xs*
by (*metis lappend-LNil2 lappend-code*(*1*) *lfinite-LNil lsublist-def*)

lemma *lsublist-trans*:
assumes *lsublist xs ys* **and** *lsublist ys zs* **shows** *lsublist xs zs*
using *assms* **unfolding** *lsublist-def*
by (*metis lappend-assoc lfinite-lappend*)

lemma *lnth-lconcat-lsublist*:
assumes *xs*: *xs* = *lconcat* (*lmap llist-of xss*) **and** *i* < *llength xss*
shows *lsublist* (*llist-of* (*lnth xss i*)) *xs*
unfolding *lsublist-def*
apply(*rule exI*[*of - lconcat* (*lmap llist-of* (*ltake i xss*))])
apply(*rule exI*[*of - lconcat* (*lmap llist-of* (*ldrop* (*Suc i*) *xss*))])
apply *simp*
by (*metis* (*no-types*, *lifting*) *assms lappend-inf lappend-ltake-ldrop lconcat-lappend lconcat-simps*(*2*) *ldrop-enat ldrop-lmap ldropn-Suc-conv-ldropn linorder-neq-iff llength-lmap llength-ltake lmap-lappend-distrib*

lnth-lmap min-def order-less-imp-le)

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 −
have *llen-Suc*: ‹*enat* (*Suc i*) < *llength xss*›
by (*simp add*: *assms*(*2*))
then have *unfold-Suc-llist-of*: ‹*llist-of* (*lnth xss* (*Suc i*)) = *lnth* (*lmap llist-of xss*) (*Suc i*)›
by (*rule lnth-lmap*[*symmetric*])

have *ldropn-Suc-complex*:‹(*llist-of* (*lnth xss* (*Suc i*)) $ *ldrop* (*enat* (*Suc* (*Suc i*))) (*lmap llist-of xss*)) = *ldropn* (*Suc i*) (*lmap llist-of xss*)›
unfolding *unfold-Suc-llist-of*
unfolding *ldrop-enat lconcat-simps*(*2*)[*symmetric*]
apply (*rule ldropn-Suc-conv-ldropn*)
by (*simp add*: *llen-Suc*)

have *llen*: ‹*enat i* < *llength xss*›
using *llen-Suc Suc-ile-eq nless-le* **by** *auto*
then have *unfold-llist-of*: ‹*llist-of* (*lnth xss i*) = *lnth* (*lmap llist-of xss*) *i*›
by (*rule lnth-lmap*[*symmetric*])
have *ldropn-complex*:‹(*llist-of* (*lnth xss i*) $ *ldropn* (*Suc i*) (*lmap llist-of xss*)) = *ldropn i* (*lmap llist-of xss*)›
unfolding *unfold-llist-of*
unfolding *ldrop-enat lconcat-simps*(*2*)[*symmetric*]
apply (*rule ldropn-Suc-conv-ldropn*)
by (*simp add*: *llen*)

show *?thesis*
unfolding *lsublist-def*
apply(*rule exI*[*of - lconcat* (*lmap llist-of* (*ltake i xss*))])
apply(*rule exI*[*of - lconcat* (*lmap llist-of* (*ldrop* (*Suc* (*Suc i*)) *xss*))])
unfolding *xs*
by *simp* (*metis enat.simps*(*3*) *enat-ord-simps*(*4*) *lappend-assoc lappend-llist-of-llist-of lappend-ltake-enat-ldropn lconcat-LCons lconcat-lappend ldrop-lmap ldropn-Suc-complex ldropn-complex lfinite-ltake ltake-lmap*)
qed

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*
by (*metis assms lappend-assoc lnth-lconcat-lsublist lsublist-def xs*)

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*
by (*metis assms lappend-assoc lnth-lconcat-lsublist2 lsublist-def xs*)

lemma *su-lset-lconcat-llist-of*:
assumes *xs* ∈ *lset xss*
shows *set xs* ⊆ *lset* (*lconcat* (*lmap llist-of xss*))
using *in-lset-lappend-iff*
by (*metis assms in-lset-conv-lnth lnth-lconcat-lsublist lset-llist-of lsublist-def subsetI*)

lemma *lsublist-lnth-lconcat*: *i* < *llength tr1s* ⟹ *lsublist* (*llist-of* (*lnth tr1s i*)) (*lconcat* (*lmap llist-of tr1s*))
by (*meson lnth-lconcat-lsublist*)

lemma *lsublist-lset*:
lsublist xs ys $\Longrightarrow$ *lset xs* $\subseteq$ *lset ys*
by (*metis in-lset-lappend-iff lsublist-def subsetI*)

lemma *lsublist-LNil*:
lsublist xs ys $\Longrightarrow$ *ys* = *LNil* $\Longrightarrow$ *xs* = *LNil*
by (*metis LNil-eq-lappend-iff lsublist-def*)

2.9 Take-until and drop-until

definition *ltakeUntil* :: ($'a \Rightarrow bool$) $\Rightarrow$ $'a$ *llist* $\Rightarrow$ $'a$ *list* **where**
ltakeUntil pred xs $\equiv$
list-of (*lappend* (*ltakeWhile* ($\lambda x.$ $\neg$ *pred x*) *xs*) [[*lhd* (*ldropWhile* ($\lambda x.$ $\neg$ *pred x*) *xs*)]])

definition *ldropUntil* :: ($'a \Rightarrow bool$) $\Rightarrow$ $'a$ *llist* $\Rightarrow$ $'a$ *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* $\Longrightarrow$ *lappend* (*llist-of* (*ltakeUntil pred xs*)) (*ldropUntil pred xs*) = *xs*
by (*simp add*: *lappend-snocL1-conv-LCons2 ldropUntil-def lfinite-ltakeWhile ltakeUntil-def*)

lemma *ltakeUntil-not-Nil*:
assumes $\exists x \in$*lset xs. pred x*
shows *ltakeUntil pred xs* $\neq$ []
by (*simp add*: *assms lfinite-ltakeWhile list-of-lappend ltakeUntil-def*)

lemma *ltakeUntil-ex-butlast*:
assumes $\exists x \in$*lset xs. pred x y* $\in$ *set* (*butlast* (*ltakeUntil pred xs*))
shows $\neg$ *pred y*
using *assms* **unfolding** *ltakeUntil-def*
by (*metis* (*mono-tags*, *lifting*) *butlast-snoc lfinite-LConsI lfinite-LNil lfinite-ltakeWhile*

list-of-LCons list-of-LNil list-of-lappend llist-of-list-of lset-llist-of lset-ltakeWhileD)

lemma *ltakeUntil-never-butlast*:
assumes $\exists x \in$*lset xs. pred x*
shows *never pred* (*butlast* (*ltakeUntil pred xs*))
using *Ball-set assms ltakeUntil-ex-butlast* **by** *fastforce*

lemma *ltakeUntil-last*:
assumes $\exists x \in$*lset xs. pred x*
shows *pred* (*last* (*ltakeUntil pred xs*))

using *assms* **unfolding** *ltakeUntil-def*
by (*metis lfinite-LConsI lfinite-LNil lfinite-lappend lfinite-ltakeWhile lhd-ldropWhile*

llast-lappend-LCons llast-llist-of llast-singleton llist-of-list-of)

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*)]
by (*simp add*: *assms ltakeUntil-not-Nil*)

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*
unfolding *ltakeUntil-def*
by *simp* (*metis lfinite-LConsI lfinite-LNil lfinite-lappend lfinite-ltakeWhile list-of-LCons*)

lemma *ldropUntil-LCons1* [*simp*]: $\exists x \in lset\ xs.\ pred\ x \Longrightarrow \neg\ pred\ x \Longrightarrow$
ldropUntil pred (*LCons x xs*) = *ldropUntil pred xs*
by (*simp add*: *ldropUntil-def*)

lemma *ltakeUntil-LCons2* [*simp*]: $\exists x \in lset\ xs.\ pred\ x \Longrightarrow pred\ x \Longrightarrow$ *ltakeUntil pred* (*LCons x xs*) = [*x*]
unfolding *ltakeUntil-def* **by** *auto*

lemma *ldropUntil-LCons2* [*simp*]: $\exists x \in lset\ xs.\ pred\ x \Longrightarrow pred\ x \Longrightarrow$ *ldropUntil pred* (*LCons x xs*) = *xs*
unfolding *ldropUntil-def* **by** *auto*

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*)
by (*smt* (*verit*, *ccfv-SIG*) *eq-LConsD list.sel*(*3*) *lset-cases ltakeUntil-LCons1*)

lemma *ldropUntil-tl1* [*simp*]:
$\exists x \in lset\ xs.\ pred\ x \Longrightarrow \neg\ pred$ (*lhd xs*) $\Longrightarrow$ *ldropUntil pred* (*ltl xs*) = *ldropUntil pred xs*
by (*metis bex-empty ldropUntil-def ldropWhile-LCons llist.exhaust-sel llist.set*(*1*))

lemma *ltakeUntil-tl2* [*simp*]:
xs $\neq$ [[]] $\Longrightarrow$ *pred* (*lhd xs*) $\Longrightarrow$ *tl* (*ltakeUntil pred xs*) = []
by (*metis lappend-code*(*1*) *lfinite-LNil list.sel*(*3*) *list-of-LCons list-of-LNil ltakeUntil-def ltakeWhile-eq-LNil-iff*)

lemma *ldropUntil-tl2* [*simp*]:
xs $\neq$ [[]] $\Longrightarrow$ *pred* (*lhd xs*) $\Longrightarrow$ *ldropUntil pred xs* = *ltl xs*
by (*metis lappend-code*(*1*) *lappend-ltakeWhile-ldropWhile ldropUntil-def ltakeWhile-eq-LNil-iff*)

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

unfolding *ldropUntil-def*
by (*metis* (*mono-tags*, *lifting*) *comp-apply eq-LConsD ldropWhile-cong lfilter-eq-LCons*)

lemma *length-ltakeUntil-ge-0*:
assumes ∃ *x*∈*lset xs*. *pred x*
shows *length* (*ltakeUntil pred xs*) > *0*
using *ltakeUntil-not-Nil*[*OF assms*] **by** *auto*

lemma *length-ltakeUntil-eq-1*:
assumes ∃ *x*∈*lset xs*. *pred x*
shows *length* (*ltakeUntil pred xs*) = *Suc 0* ⟷ *pred* (*lhd xs*)
proof−
obtain *x xss* **where** *xs*: *xs* = *LCons x xss*
using *assms* **by** (*cases xs*, *auto*)
hence *x*: *x* = *lhd xs* **by** *auto*
show *?thesis* **unfolding** *xs*
using *ltakeUntil-LCons2*[*OF assms*, *of x*] *x*
by (*smt* (*verit*, *del-insts*) *assms diff-Suc-1 eq-LConsD lappend-ltakeUntil-ldropUntil*

length-Suc-conv-rev length-butlast length-tl length-ltakeUntil-ge-0 list.size(*3*) *lnth-0*

lnth-lappend-llist-of ltakeUntil-last ltakeUntil-last-butlast ltakeUntil-tl2 nth-append-length xs)
qed

lemma *length-ltakeUntil-Suc*:
assumes ∃ *x*∈*lset xs*. *pred x* ¬ *pred* (*lhd xs*)
shows *length* (*ltakeUntil pred xs*) = *Suc* (*length* (*ltakeUntil pred* (*ltl xs*)))
proof−
obtain *x xss* **where** *xs*: *xs* = *LCons x xss*
using *assms* **by** (*cases xs*, *auto*)
hence *x*: *x* = *lhd xs* **and** *xss*: *xss* = *ltl xs* **by** *auto*
hence *0*: ∃ *x*∈*lset xss*. *pred x*
by (*metis assms*(*1*) *assms*(*2*) *insertE lset-LCons xs*)
show *?thesis* **unfolding** *xs*
unfolding *ltakeUntil-LCons1*[*OF 0 assms*(*2*), *unfolded x*[*symmetric*]] **by** *simp*
qed

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

primcorec *lsplit* :: (*'a* ⇒ *bool*) ⇒ *'a llist* ⇒ *'a list llist* **where**
lsplit pred xs =
(*if* (∃ *x*∈*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* ∈ *lset xs. pred x*} ⟹ *lsplit pred xs* = *LCons* (*ltakeUntil pred xs*) (*lsplit pred* (*ldropUntil pred xs*))
using *lsplit.ctr*(*2*) *not-finite-existsD* **by** *force*

lemma *lconcat-lsplit-not-lfinite*:
¬ *lfinite* (*lfilter pred xs*) ⟹ *xs* = *lconcat* (*lmap llist-of* (*lsplit pred xs*))
apply(*coinduction arbitrary*: *xs*) **apply** *safe*
subgoal by *simp*
subgoal by *simp* (*metis* (*full-types*) *image-subset-iff llist.set-sel*(*1*) *lnull-imp-lfinite lnull-lfilter*
lnull-llist-of lsplit.simps(*2*) *lsplit.simps*(*3*) *ltakeUntil-not-Nil mem-Collect-eq*)
subgoal by (*smt* (*verit*) *lappend-ltakeUntil-ldropUntil lhd-lappend lhd-lconcat llist.map-disc-iff*
llist.map-sel(*1*) *llist-of.simps*(*1*) *llist-of-inject*
lnull-def lnull-imp-lfinite lnull-lfilter lsplit.simps(*2*) *lsplit.simps*(*3*) *ltakeUntil-not-Nil*)
subgoal for *xs* **apply**(*subgoal-tac xs* ≠ [[]])
subgoal apply(*subgoal-tac* ¬ *lfinite* (*lfilter pred* (*ltl xs*)) ∧ (∃ *x*∈*lset xs. pred x*))
subgoal apply(*cases pred* (*lhd xs*))
subgoal by *simp* (*meson ltakeUntil-not-Nil*)
subgoal apply(*rule exI*[*of - ltl xs*], *safe*)
subgoal apply(*subst ltl-lconcat*)
subgoal by *auto*
subgoal by (*metis llist.map-sel*(*1*) *lnull-llist-of lsplit.disc*(*2*) *lsplit.simps*(*3*) *ltakeUntil-not-Nil*)
subgoal unfolding *ltl-lmap* **apply**(*subst lsplit.sel*(*2*))
subgoal by *auto*
subgoal using *ltakeUntil-tl1 ltl-lappend ltl-lconcat ltl-llist-of ltl-lmap ltl-simps*(*2*)
by (*smt* (*verit*) *lconcat-LCons ldropUntil-tl1 lhd-LCons-ltl*
llist.map-disc-iff llist.map-sel(*1*) *lnull-imp-lfinite lnull-lfilter*
lsplit.ctr(*2*) *lsplit.disc-iff*(*2*) *lsplit.simps*(*3*)) **. . . .**
subgoal by (*metis diverge-lfilter-LNil lfilter-LCons lfinite.simps lhd-LCons-ltl*)
.
subgoal by *auto* **. .**

lemma *lfinite-lsplit*:
assumes *lfinite* (*lfilter pred xs*)
shows *lfinite* (*lsplit pred xs*)
proof−
{**fix** *ys* **assume** *lfinite ys ys* = *lfilter pred xs*
hence *?thesis* **proof**(*induct arbitrary*: *xs*)
case *lfinite-LNil*
then show *?case* **by** (*metis lfilter-empty-conv lnull-imp-lfinite lsplit.disc*(*1*))
next
case (*lfinite-LConsI ys y xs*)
then show *?case* **apply**(*cases* ∃ *x*∈*lset xs. pred x*)
subgoal by *simp* (*meson LCons-lfilter-ldropUntil*)

subgoal using *lnull-imp-lfinite lsplit.disc*(*1*) **by** *blast* **.**
qed
}
thus *?thesis* **using** *assms* **by** *auto*
qed

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−
{fix *ys* **assume** *lfinite ys ys* = *lfilter pred xs*
hence *?thesis* **proof**(*induct arbitrary*: *xs*)
case *lfinite-LNil*
then show *?case*
by (*metis lappend-code*(*1*) *lconcat-LNil llist.disc*(*1*) *llist.simps*(*12*) *lnull-lfilter lsplit.ctr*(*1*))
next
case (*lfinite-LConsI ys y xs*)
then show *?case* **apply**(*cases* ∃ *x*∈*lset xs. pred x*)
subgoal by *simp* (*smt* (*verit*) *LCons-lfilter-ldropUntil lappend-assoc lappend-ltakeUntil-ldropUntil*)
subgoal by *simp* **.**
qed
}
thus *?thesis* **using** *assms* **by** *auto*
qed

lemma *lconcat-lsplit*:
∃ *ys. xs* = *lappend* (*lconcat* (*lmap llist-of* (*lsplit pred xs*))) *ys* ∧ (∀ *y*∈*lset ys.* ¬ *pred y*)
proof(*cases lfinite* (*lfilter pred xs*))
case *True*
show *?thesis* **using** *lconcat-lsplit-lfinite*[*OF True*] **.**
next
case *False*
show *?thesis* **apply**(*rule exI*[*of - LNil*])
using *lconcat-lsplit-not-lfinite*[*OF False*] **by** *simp*
qed

lemma *lsublist-lsplit*:
assumes *i* < *llength* (*lsplit pred xs*)
shows *lsublist* (*llist-of* (*lnth* (*lsplit pred xs*) *i*)) *xs*
by (*metis assms lconcat-lsplit lnth-lconcat-lconcat-lsublist*)

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*

by (*metis assms lconcat-lsplit lnth-lconcat-lconcat-lsublist2*)

lemma *lsplit-main*:
llist-all (*λzs. zs* ≠ [] ∧ *list-all* (*λz.* ¬ *pred z*) (*butlast zs*) ∧ *pred* (*last zs*))
(*lsplit pred xs*)
proof−
{**fix** *ys* **assume** ∃ *xs. ys* = (*lsplit pred xs*)
hence *llist-all* (*λzs. zs* ≠ [] ∧ *list-all* (*λz.* ¬ *pred z*) (*butlast zs*) ∧ *pred* (*last zs*)) *ys*
apply(*coinduct rule*: *llist-all-conduct*[**where** *X* = *λys.* ∃ *xs. ys* = (*lsplit pred xs*)])
apply *safe*
subgoal by *simp* (*meson ltakeUntil-not-Nil*)
subgoal by *simp* (*metis ltakeUntil-never-butlast*)
subgoal by *simp* (*meson ltakeUntil-last*)
subgoal by *auto* **.**
}
thus *?thesis* **by** *auto*
qed

lemma *lsplit-main-lset*:
assumes *ys* ∈ *lset* (*lsplit pred xs*)
shows *ys* ≠ [] ∧
list-all (*λz.* ¬ *pred z*) (*butlast ys*) ∧
pred (*last ys*)
using *assms lsplit-main*[*of pred*] **unfolding** *llist.pred-set* **by** *auto*

lemma *lsplit-main-lnth*:
assumes *i* < *llength* (*lsplit pred xs*)
shows *lnth* (*lsplit pred xs*) *i* ≠ [] ∧
list-all (*λz.* ¬ *pred z*) (*butlast* (*lnth* (*lsplit pred xs*) *i*)) ∧
pred (*last* (*lnth* (*lsplit pred xs*) *i*))
using *assms lsplit-main*[*of pred*] **unfolding** *llist-all-lnth* **by** *auto*

lemma *hd-lhd-lsplit*: ∃ *x*∈*lset xs. pred x* ⟹ *hd* (*lhd* (*lsplit pred xs*)) = *lhd xs*
by (*metis lappend-ltakeUntil-ldropUntil lhd-lappend lhd-llist-of lnull-llist-of lsplit.simps*(*3*) *ltakeUntil-not-Nil*)

lemma *lprefix-lsplit*:
assumes ∃ *x*∈*lset xs. pred x*
shows *lprefix* (*llist-of* (*lhd* (*lsplit pred xs*))) *xs*
by (*metis assms lappend-ltakeUntil-ldropUntil lprefix-lappend lsplit.simps*(*3*))

lemma *lprefix-lsplit-lbutlast*:
assumes ∃ *x*∈*lset xs. pred x*
shows *lprefix* (*llist-of* (*butlast* (*lhd* (*lsplit pred xs*)))) (*lbutlast xs*)
using *lprefix-lsplit*[*OF assms*] **unfolding** *llist-of-butlast*
using *lprefix-lbutlast* **by** *blast*

lemma *set-lset-lsplit*:
assumes *ys* ∈ *lset* (*lsplit pred xs*)
shows *set ys* ⊆ *lset xs*
proof−
  have *set ys* ⊆ *lset* (*lconcat* (*lmap llist-of* (*lsplit pred xs*)))
  using *su-lset-lconcat-llist-of*[*OF assms*] **.**
  also have . . . ⊆ *lset xs*
  by (*metis lconcat-lsplit lset-lappend1*)
  finally show *?thesis* **.**
qed

lemma *set-lnth-lsplit*:
assumes *i* < *llength* (*lsplit pred xs*)
shows *set* (*lnth* (*lsplit pred xs*) *i*) ⊆ *lset xs*
by (*meson assms in-lset-conv-lnth set-lset-lsplit*)

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*)
unfolding *lsplitRemainder-def* **apply**(*rule someI-ex*) **using** *lconcat-lsplit* **.**

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*)
using *assms* **unfolding** *firstHolds-def*
by (*metis Suc-pred diff-Suc-1 length-ltakeUntil-eq-1 length-ltakeUntil-ge-0*)

lemma *firstHolds-eq-0′*:
assumes ¬ *lnever pred xs*
shows *firstHolds pred xs* = *0* ⟷ *pred* (*lhd xs*)
apply(*rule firstHolds-eq-0*)
using *assms* **by** (*simp add*: *llist.pred-set*)

lemma *firstHolds-Suc*:
assumes ∃ *x*∈*lset xs*. *pred x* **and** ¬ *pred* (*lhd xs*)
shows *firstHolds pred xs* = *Suc* (*firstHolds pred* (*ltl xs*))
using *assms* **unfolding** *firstHolds-def*

by (*smt* (*verit*, *best*) *Suc-pred diff-Suc-1 length-greater-0-conv length-ltakeUntil-Suc length-ltakeUntil-eq-1 list.size*(*3*))

lemma *firstHolds-Suc′*:
assumes ¬ *lnever pred xs* **and** ¬ *pred* (*lhd xs*)
shows *firstHolds pred xs* = *Suc* (*firstHolds pred* (*ltl xs*))
apply(*rule firstHolds-Suc*) **using** *assms* **by** (*auto simp*: *llist.pred-set*)

lemma *firstHolds-append*:
assumes ¬ *lnever pred xs* **and** *never pred ys*
shows *firstHolds pred* (*lappend* (*llist-of ys*) *xs*) = *length ys* + *firstHolds pred xs*
using *assms* **by** (*induct ys*) (*auto simp add*: *llist-all-lappend-llist-of firstHolds-Suc′*)

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

definition *firstNC* **where**
firstNC xss ≡ *firstHolds* (λ*xs*. *xs* ≠ []) *xss*

lemma *firstNC-eq-0*:
assumes ∃ *xs*∈*lset xss*. *xs* ≠ []
shows *firstNC xss* = *0* ⟷ *lhd xss* ≠ []
using *assms* **unfolding** *firstNC-def*
by (*simp add*: *firstHolds-eq-0*)

lemma *firstNC-Suc*:
assumes ∃ *xs*∈*lset xss*. *xs* ≠ [] **and** *lhd xss* = []
shows *firstNC xss* = *Suc* (*firstNC* (*ltl xss*))
using *assms* **unfolding** *firstNC-def* **by** (*simp add*: *firstHolds-Suc*)

lemma *firstNC-LCons-notNil*: *xs* ≠ [] ⟹ *firstNC* (*xs* $ *xss*) = *0*
by (*simp add*: *firstNC-eq-0*)

lemma *firstNC-LCons-Nil*:
(∃ *ys*∈*lset xss*. *ys* ≠ []) ⟹ *xs* = [] ⟹ *firstNC* (*xs* $ *xss*) = *Suc* (*firstNC xss*)
by (*simp add*: *firstNC-Suc*)

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* ⇒ *bool*) ⇒ ('*trans* ⇒ '*a*) ⇒ '*trans llist* ⇒ '*a llist*
where
lfiltermap pred func tr ≡ *lmap func* (*lfilter pred tr*)

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

lemma *lfiltermap-lappend*: *lfinite tr* ⟹ *lfiltermap pred func* (*lappend tr tr1*) =
  lappend (*lfiltermap pred func tr*) (*lfiltermap pred func tr1*)
unfolding *lfiltermap-def* **by** (*simp add*: *lmap-lappend-distrib*)

lemma *lfiltermap-LNil-never*: *lfiltermap pred func tr* = [[]] ⟷ *lnever pred tr*
by (*simp add*: *lfilter-empty-conv lfiltermap-lmap-lfilter llist.pred-set*)

lemma *llength-lfiltermap*: *llength* (*lfiltermap pred func tr*) ≤ *llength tr*
by (*simp add*: *lfiltermap-lmap-lfilter llength-lfilter-ile*)

lemma *lfiltermap-llist-all*[*simp*]: *lfinite tr* ⟹ *lfiltermap pred func tr* = *lmap func tr* ⟷ *llist-all pred tr*
apply (*induction list-of tr arbitrary*: *tr*)
  subgoal using *llist-of-list-of*
   by (*metis lfiltermap-LNil-never llist.pred-inject*(*1*) *llist.simps*(*12*) *llist-of.simps*(*1*) *llist-of-list-of*)
  subgoal for *a x tr* **apply**(*cases tr*, *auto*)
    apply (*metis iless-Suc-eq length-list-of lfilter-LCons lfiltermap-lmap-lfilter lfinite-LConsI linorder-not-less llength-LCons llength-lfiltermap llength-lmap*)
   apply (*metis Suc-ile-eq eSuc-enat length-list-of lfilter-LCons lfiltermap-lmap-lfilter linorder-not-less*
      llength-LCons llength-lfiltermap llength-lmap llist.sel(*3*) *ltl-lmap*)
    by (*simp add*: *lfiltermap-lmap-lfilter*) **.**

lemma *lfilter-LNil-lnever*: [[]] = *lfilter pred xs* ⟹ *lnever pred xs*
by (*metis lfiltermap-LNil-never lfiltermap-lmap-lfilter llist.simps*(*12*))

lemma *lnever-LNil-lfilter*: *lnever pred xs* ⟷ [[]] = *lfilter pred xs*
by (*metis lfilter-empty-conv llist.pred-set*)

lemma *lfilter-LNil-lnever'*: *lfilter pred xs* = [[]] ⟹ *lnever pred xs*
by (*metis lfiltermap-LNil-never lfiltermap-lmap-lfilter llist.simps*(*12*))

lemma *lnever-LNil-lfilter'*: *lnever pred xs* ⟷ *lfilter pred xs* = [[]]
by (*metis lfilter-empty-conv llist.pred-set*)

lemma *lfiltermap-LCons2-eq*:
    lfiltermap pred func [[*x*, *x'*]] = *lfiltermap pred func* [[*y*, *y'*]]

$\Longrightarrow$ *lfiltermap pred func* $(x \ \$ \ x' \ \$ \ zs)$ = *lfiltermap pred func* $(y \ \$ \ y' \ \$ \ zs)$
by (*metis lappend-code*(*1*) *lappend-code*(*2*) *lfiltermap-lappend lfinite-LCons lfinite-LNil*)

lemma *lfiltermap-LCons-cong*:
lfiltermap pred func xs = *lfiltermap pred func ys*
$\Longrightarrow$ *lfiltermap pred func* $(x \ \$ \ xs)$ = *lfiltermap pred func* $(x \ \$ \ ys)$
by (*simp add*: *lfiltermap-lmap-lfilter*)

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* $(y \ \$ \ ys)$
by (*simp add*: *lfiltermap-lmap-lfilter*)

lemma *set-lfiltermap*:
lset (*lfiltermap pred func xs*) $\subseteq$ $\{$*func x* $\mid$ *x* . *x* $\in$ *lset xs* $\wedge$ *pred x*$\}$
unfolding *lfiltermap-def* **by** *auto*

lemma *lfinite-lfiltermap-filtermap*:
lfinite xs $\Longrightarrow$ *lfiltermap pred func xs* = *llist-of* (*filtermap pred func* (*list-of xs*))
by (*induct rule*: *lfinite.induct*, *auto simp*: *lfiltermap-lmap-lfilter*)

lemma *lfiltermap-llist-of-filtermap*:
lfiltermap pred func(*llist-of xs*) = *llist-of* (*filtermap pred func xs*)
by (*simp add*: *lfinite-lfiltermap-filtermap*)

lemma *filtermap-butlast*: *xs* $\neq$ [] $\Longrightarrow$
$\neg$ *pred* (*last xs*) $\Longrightarrow$
filtermap pred func xs = *filtermap pred func* (*butlast xs*)
by (*metis append-butlast-last-id not-holds-filtermap-RCons*)

lemma *filtermap-butlast'*:
xs $\neq$ [] $\Longrightarrow$ *pred* (*last xs*) $\Longrightarrow$
filtermap pred func xs = *filtermap pred func* (*butlast xs*) @ [*func* (*last xs*)]
by (*metis append-butlast-last-id holds-filtermap-RCons*)

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

lfiltermap pred func xs = *lfiltermap pred func* (*lbutlast xs*)
unfolding *lbutlast-def*
by (*metis* (*full-types*) *filtermap-butlast lfiltermap-llist-of-filtermap llast-last-llist-of*

llist-of.simps(*1*) *llist-of-list-of*)

lemma *last-filtermap*: *xs* $\neq$ [] $\Longrightarrow$ *pred* (*last xs*) $\Longrightarrow$
filtermap pred func xs $\neq$ [] $\wedge$ *last* (*filtermap pred func xs*) = *func* (*last xs*)
by (*metis holds-filtermap-RCons snoc-eq-iff-butlast*)

lemma *filtermap-ltakeUntil*[*simp*]:
$\exists x \in lset\ xs.\ pred\ x \Longrightarrow filtermap\ pred\ func\ (ltakeUntil\ pred\ xs) = [func\ (last\ (ltakeUntil\ pred\ xs))]$
unfolding *filtermap-def*
by (*smt* (*verit*, *del-insts*) *Cons-eq-map-conv append-butlast-last-id append-self-conv2 filter.simps*(*1*)
filter-eq-Cons-iff ltakeUntil-ex-butlast ltakeUntil-last ltakeUntil-not-Nil map-append)

lemma *last-ltakeUntil-filtermap*[*simp*]:
$\exists x \in lset\ xs.\ pred\ x \Longrightarrow func\ (last\ (ltakeUntil\ pred\ xs)) = lhd\ (lfiltermap\ pred\ func\ xs)$
unfolding *lfiltermap-lmap-lfilter*
by *simp* (*metis* (*no-types*, *lifting*) *ldropWhile-cong lfinite-LConsI lfinite-LNil lfinite-lappend*
lfinite-ltakeWhile llast-lappend-LCons llast-llist-of llast-singleton llist-of-list-of ltakeUntil-def)

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*)
using *assms* **apply**(*coinduction arbitrary*: *xs ys*)
by *simp* (*smt* (*verit*, *best*) *LCons-lfilter-ldropUntil lfiltermap-lmap-lfilter*
llist.map-disc-iff lnull-lfilter ltl-lmap ltl-simps(*2*) *not-lnull-conv*)

lemma *lfiltermap-lfinite-lsplit*:
assumes *lfiltermap pred func xs* = *lfiltermap pred func ys*
shows *lfinite* (*lsplit pred xs*) $\longleftrightarrow$ *lfinite* (*lsplit pred ys*)
by (*metis assms lfiltermap-lmap-filtermap-lsplit llength-eq-infty-conv-lfinite llength-lmap*)

lemma *lfiltermap-lsplitRemainder*[*simp*]: *lfiltermap pred func* (*lsplitRemainder pred xs*) = [[]]
by (*metis lfiltermap-LNil-never llist.pred-set lset-lsplitRemainder*)

lemma *lfiltermap-lconcat-lsplit*:
lfiltermap pred func xs =
lfiltermap pred func (*lconcat* (*lmap llist-of* (*lsplit pred xs*)))
apply(*subst lsplit-lsplitRemainder*[*of xs pred*])
apply(*cases lfinite* (*lconcat* (*lmap llist-of* (*lsplit pred xs*))))
subgoal apply(*subst lfiltermap-lappend*) **by** *auto*
subgoal apply(*subst lappend-inf*) **by** *auto* **.**

lemma *lfilter-lconcat-lfinite'*: $(\bigwedge i.\ i < llength\ yss \Longrightarrow lfinite\ (lnth\ yss\ i))$
$\Longrightarrow lfilter\ pred\ (lconcat\ yss) = lconcat\ (lmap\ (lfilter\ pred)\ yss)$
by (*metis in-lset-conv-lnth lfilter-lconcat-lfinite*)

lemma *lfilter-lconcat-llist-of*:

lfilter pred (*lconcat* (*lmap llist-of yss*)) = *lconcat* (*lmap* (*lfilter pred*) (*lmap llist-of yss*))
apply(*rule lfilter-lconcat-lfinite*) **by** *auto*

lemma *lfiltermap-lconcat-lmap-llist-of*:
lfiltermap pred func (*lconcat* (*lmap llist-of yss*)) =
lconcat (*lmap* (*llist-of o filtermap pred func*) *yss*)
unfolding *lfiltermap-def lfilter-lconcat-llist-of*
unfolding *lmap-lconcat filtermap-def*
by *simp* (*smt* (*verit*, *best*) *in-lset-conv-lnth lfilter-llist-of*
llength-lmap llist.map-comp llist.map-cong lmap-llist-of lnth-lmap)

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−
define *yss* **where** *yss*: *yss* ≡ *lsplit pred xs*
define *yss′* **where** *yss′*: *yss′* ≡ *lsplit pred xs′*
define *u* **where** *u*: *u* = *lmap* (*llist-of o filtermap pred func*) *yss*
define *u′* **where** *u′*: *u′* = *lmap* (*llist-of o filtermap pred func*) *yss′*
have *lfiltermap pred func* (*lconcat* (*lmap llist-of yss*)) ≠
lfiltermap pred func (*lconcat* (*lmap llist-of yss′*))
using *l*[*unfolded lfiltermap-lconcat-lsplit*[*of pred func xs*]
lfiltermap-lconcat-lsplit[*of pred func xs′*]]
unfolding *yss yss′* **.**
hence *lconcat u* ≠ *lconcat u′*
unfolding *lfiltermap-lconcat-lmap-llist-of u u′* **.**
hence *u* ≠ *u′* **by** *auto*
moreover have *llength u* = *llength u′*
by (*simp add*: *u u′ len yss yss′*)
ultimately obtain *i0* **where** *0*: *i0* < *llength u lnth u i0* ≠ *lnth u′ i0*
by (*metis llist.rel-eq llist-all2-all-lnthI*)

show *?thesis* **unfolding** *yss*[*symmetric*] *yss′*[*symmetric*] **apply**(*rule exI*[*of - i0*])
using *0 len* **unfolding** *u u′*
by *simp* (*metis lnth-lmap yss yss′*)
qed

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−
obtain *a ass′* **where** *1*: *lfilter pred as = LCons a ass′ b = func a bss = lmap func ass′*
using *assms* **by** (*metis lmap-eq-LCons-conv*)
obtain *us vs* **where** *2*: *as = lappend us (a $ vs) lfinite us*
∀ *u*∈*lset us.* ¬ *pred u pred a ass′ = lfilter pred vs*
using *lfilter-eq-LConsD*[*OF 1*(*1*)] **by** *auto*
show *?thesis* **apply**(*rule exI*[*of - list-of us*]) **apply**(*rule exI*[*of - a*]) **apply**(*rule exI*[*of - vs*])
using *1 2*
by *simp* (*metis Ball-set set-list-of*)
qed

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−
obtain *a ass′* **where** *1*: *lfilter pred′ as = LCons a ass′ b = func′ a bss = lmap func′ ass′*
using *assms* **by** (*metis lmap-eq-LCons-conv*)
obtain *us vs* **where** *2*: *as = lappend us (a $ vs) lfinite us*
∀ *u*∈*lset us.* ¬ *pred′ u pred′ a ass′ = lfilter pred′ vs*
using *lfilter-eq-LConsD*[*OF 1*(*1*)] **by** *auto*
show *?thesis* **apply**(*rule exI*[*of - list-of us*]) **apply**(*rule exI*[*of - a*]) **apply**(*rule exI*[*of - vs*])
using *1 2*
by *simp* (*metis Ball-set set-list-of*)
qed

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 safe
  assume ln: lnever pred lxs
  show lnever pred′ lxs′
  unfolding llist-all-lnth using P ln apply (intro allI impI)
  subgoal for i proof(induct i arbitrary: lxs lxs′ rule: less-induct)
    case (less i lxs lxs′)
    show ?case using lappend[OF less(2)] proof(elim disjE exE conjE)
      fix ys llxs ys′ llxs′
       assume yss′: ys ≠ [] ys′ ≠ [] map func (filter pred ys) = map func′ (filter
pred′ ys′)
      and lxs: lxs = lappend (llist-of ys) llxs and lxs′: lxs′ = lappend (llist-of ys′)
llxs′
      and P: P llxs llxs′
      have lnys: never pred ys and lnllxs: lnever pred llxs using ‹lnever pred lxs›
unfolding lxs
        by (auto simp add: list-all-iff llist.pred-set)
      hence lnys′: never pred′ ys′ using yss′(2)
        by (metis Nil-is-map-conv never-Nil-filter yss′(3))
      show ¬ pred′ (lnth lxs′ i)
      proof(cases i < length ys′)
        case True note i = True
        hence 0: lnth lxs′ i = ys′ ! i unfolding lxs lxs′
          by (auto simp: lnth-lappend-llist-of)
        show ?thesis using lnys′ i unfolding 0
          by (simp add: list-all-length)
      next
        case False note i = False
        then obtain j where i: i = length ys′ + j
        using le-Suc-ex not-le-imp-less by blast
        hence j: j < llength llxs′ using ‹i < llength lxs′› unfolding lxs′
        using enat-add-mono by fastforce
        hence 0: lnth lxs′ i = lnth llxs′ j unfolding lxs lxs′
        unfolding i  by (auto simp: lnth-lappend-llist-of)
        show ?thesis unfolding 0 apply(rule less(1)[rule-format, OF - P])
        using j P yss′ less.prems lnllxs unfolding i by auto
      qed
   qed(metis less.prems(2) less.prems(3) llist-all-lnth lmap-eq-LNil lnever-LNil-lfilter′)
  qed .
next
  assume ln′: lnever pred′ lxs′
  show lnever pred lxs
  unfolding llist-all-lnth using P ln′ apply (intro allI impI)
  subgoal for i proof(induct i arbitrary: lxs lxs′ rule: less-induct)
    case (less i lxs lxs′)
    show ?case using lappend[OF less(2)] proof(elim disjE exE conjE)
      fix ys llxs ys′ llxs′
       assume yss′: ys ≠ [] ys′ ≠ [] map func (filter pred ys) = map func′ (filter
```

pred′ ys′)
  and *lxs*: *lxs* = *lappend* (*llist-of ys*) *llxs* **and** *lxs′*: *lxs′* = *lappend* (*llist-of ys′*) *llxs′*
  and *P*: *P llxs llxs′*
  have *lnys′*: *never pred′ ys′* **and** *lnllxs*: *lnever pred′ llxs′* **using** ‹*lnever pred′ lxs′*› **unfolding** *lxs′*
    by (*auto simp add*: *list-all-iff llist.pred-set*)
  hence *lnys*: *never pred ys* **using** *yss′*(*2*)
    by (*metis Nil-is-map-conv never-Nil-filter yss′*(*3*))
  show ¬ *pred* (*lnth lxs i*)
  proof(*cases i* < *length ys*)
    case *True* **note** *i* = *True*
    hence *0*: *lnth lxs i* = *ys* ! *i* **unfolding** *lxs lxs′*
      by (*auto simp*: *lnth-lappend-llist-of*)
    show *?thesis* **using** *lnys i* **unfolding** *0*
      by (*simp add*: *list-all-length*)
  next
    case *False* **note** *i* = *False*
    then obtain *j* **where** *i*: *i* = *length ys* + *j*
    using *le-Suc-ex not-le-imp-less* **by** *blast*
    hence *j*: *j* < *llength llxs* **using** ‹*i* < *llength lxs*› **unfolding** *lxs*
    using *enat-add-mono* **by** *fastforce*
    hence *0*: *lnth lxs i* = *lnth llxs j* **unfolding** *lxs lxs′*
    unfolding *i* **by** (*auto simp*: *lnth-lappend-llist-of*)
    show *?thesis* **unfolding** *0* **apply**(*rule less*(*1*)[*rule-format*, *OF* - *P*])
    using *j P yss′ less.prems lnllxs* **unfolding** *i* **by** *auto*
  qed
 qed(*metis less.prems*(*2*) *less.prems*(*3*) *llist-all-lnth lmap-eq-LNil lnever-LNil-lfilter′*)
 qed .
qed

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′)
using *P* **proof**(*induct firstHolds pred lxs arbitrary*: *lxs lxs′ rule*: *less-induct*)

```
  case (less lxs lxs')
  show ?case using lappend[OF less(2)] proof(elim disjE allE conjE exE)
    fix ys llxs ys' llxs'
    assume yss': ys ≠ [] ys' ≠ [] map func (filter pred ys) = map func' (filter pred'
ys')
     and lxs: lxs = lappend (llist-of ys) llxs and lxs': lxs' = lappend (llist-of ys')
llxs'
    and P: P llxs llxs'
    show ?thesis
    proof(cases lnever pred lxs)
      case True note ln = True
      hence ln': lnever pred' lxs'
      using lappend less.prems lmap-lfilter-lappend-lnever by blast
      show ?thesis apply(rule disjI1)
      using ln ln' by (simp add: lnever-LNil-lfilter')
    next
      case False note ln = False
      hence ln': ¬ lnever pred' lxs'
      using lappend less.prems lmap-lfilter-lappend-lnever by blast
     have ¬ never pred ys ∨ (never pred ys ∧ ¬ lnever pred llxs) using ln unfolding
lxs
      unfolding llist-all-lappend-llist-of by simp
      thus ?thesis proof(elim disjE conjE)
        assume ys: ¬ never pred ys
        show ?thesis apply(rule disjI2)
         apply(rule exI[of - ys]) apply(rule exI[of - llxs]) apply(rule exI[of - ys'])
apply(rule exI[of - llxs'])
        using yss' lxs lxs' P ys by (auto simp:  never-Nil-filter)
      next
        assume ys: never pred ys and llxs: ¬ lnever pred llxs
        hence ys': never pred' ys' and llxs': ¬ lnever pred' llxs'
        apply (metis Nil-is-map-conv never-Nil-filter yss'(3))
        using P lappend llxs lmap-lfilter-lappend-lnever by blast

        have firstHolds: firstHolds pred llxs < firstHolds pred lxs
        unfolding lxs firstHolds-append[OF llxs ys] using yss' by simp

         show ?thesis proof(cases lmap func (lfilter pred llxs) = lmap func' (lfilter
pred' llxs'))
          case True
          hence lmap func (lfilter pred lxs) = lmap func' (lfilter pred' lxs')
          unfolding lxs lxs' using ys ys'
          by (simp add: lmap-lappend-distrib yss'(3))
          thus ?thesis by simp
        next
          case False
           then obtain yys lllxs yys' lllxs' where yyss': map func (filter pred yys)
≠ []
          map func (filter pred yys) = map func' (filter pred' yys')
```

and *llxs*: *llxs* = *lappend* (*llist-of yys*) *lllxs* **and** *llxs'*: *llxs'* = *lappend* (*llist-of yys'*) *lllxs'*
and *P lllxs lllxs'* **using** *less*(*1*)[*OF firstHolds P*] **by** *blast*

show *?thesis* **apply**(*rule disjI2*)
apply(*rule exI*[*of - ys @ yys*]) **apply**(*rule exI*[*of - lllxs*])
apply(*rule exI*[*of - ys' @ yys'*]) **apply**(*rule exI*[*of - lllxs'*])
apply(*intro conjI*)
subgoal using *yyss'*(*1*) **by** *simp*
subgoal apply *simp* **unfolding** *yss'*(*3*) *yyss'*(*2*) **..**
subgoal unfolding *lxs llxs lappend-assoc lappend-llist-of-llist-of*[*symmetric*] **..**
subgoal unfolding *lxs' llxs' lappend-assoc lappend-llist-of-llist-of*[*symmetric*] **..**
subgoal by *fact* **.**
qed
qed
qed
qed *simp*
qed

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−
have *lappend*:
$\bigwedge$*lxs lxs'*. *P lxs lxs'* $\Longrightarrow$
lmap func (*lfilter pred lxs*) = *lmap func'* (*lfilter pred' lxs'*) $\vee$
($\exists$ *ys llxs ys' llxs'*.
map func (*filter pred 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')
using *lmap-lfilter-lappend-makeStronger*[*OF lappend*] **by** *auto*
show *?thesis* **apply**(*rule llist-lappend-coind*[**where** *P* = λ*as as'*.
$\exists$ *lxs lxs'*. *as* = *lmap func* (*lfilter pred lxs*) $\wedge$
as' = *lmap func'* (*lfilter pred' lxs'*) $\wedge$
P lxs lxs'])

subgoal using *P* **by** *auto*
subgoal for *lxs lxs'* **using** *lappend*
by (*smt* (*verit*, *ccfv-SIG*) *lfilter-lappend-llist-of list.map-disc-iff lmap-lappend-distrib lmap-llist-of*) **.**
qed

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'*) ∨
(∃ *v ys llxs ys' llxs'*.
(*ys* ≠ [] ∧ *ys'* ≠ [] ∨ (*v*,*w*) ∈ *W*) ∧
map func (*filter pred ys*) = *map func'* (*filter pred' ys'*) ∧
lxs = *lappend* (*llist-of ys*) *llxs* ∧ *lxs'* = *lappend* (*llist-of ys'*) *llxs'* ∧
P v llxs llxs')
shows *lmap func* (*lfilter pred lxs*) = *lmap func'* (*lfilter pred' lxs'*)
proof−
define *Q* **where** *Q* ≡ λ *llxs llxs'*. ∃ *w*. *P w llxs llxs'*
have *Q*: *Q lxs lxs'* **using** *P* **unfolding** *Q-def* **by** *auto*
{fix *lxs lxs'* **assume** *Q lxs lxs'*
then obtain *w* **where** *P w lxs lxs'* **using** *Q-def* **by** *auto*
hence *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'* ∧
Q llxs llxs')
proof(*induct w arbitrary*: *lxs lxs' rule*: *wf-induct-rule*[*OF W*])
case (*1 w lxs lxs'*)
show *?case* **using** *lappend*[*OF 1*(*2*)] **apply**(*elim disjE*)
subgoal by *simp*
subgoal proof(*elim exE disjE conjE*)
fix *v ys llxs ys' llxs'* **assume** *vw*: (*v*, *w*) ∈ *W*
and *yss'*: *map func* (*filter pred ys*) = *map func'* (*filter pred' ys'*)
and *lxs*: *lxs* = *lappend* (*llist-of ys*) *llxs*
and *lxs'*: *lxs'* = *lappend* (*llist-of ys'*) *llxs'*
and *P*: *P v llxs llxs'*
show *?thesis*
proof(*cases lmap func* (*lfilter pred lxs*) = *lmap func'* (*lfilter pred' lxs'*))
case *True*
thus *?thesis* **by** *simp*
next
case *False*
hence *lmap func* (*lfilter pred llxs*) ≠ *lmap func'* (*lfilter pred' llxs'*)
using *yss'* **unfolding** *lxs lxs'* **by** (*auto simp*: *lmap-lappend-distrib*)

```
        then obtain yys lllxs yys' lllxs' where yyss': yys ≠ [] yys' ≠ []
        map func (filter pred yys) = map func' (filter pred' yys')
        and llxs: llxs = lappend (llist-of yys) lllxs
        and llxs': llxs' = lappend (llist-of yys') lllxs'
        and Q lllxs lllxs'using 1(1)[OF vw P] by auto
        show ?thesis apply(rule disjI2)
        apply(rule exI[of - ys @ yys]) apply(rule exI[of - lllxs])
        apply(rule exI[of - ys' @ yys']) apply(rule exI[of - lllxs'])
        apply(intro conjI)
         subgoal using yyss'(1) by simp
         subgoal using yyss'(2) by simp
         subgoal apply simp unfolding yss' yyss' ..
      subgoal unfolding lxs llxs lappend-assoc lappend-llist-of-llist-of[symmetric]
..
      subgoal unfolding lxs' llxs' lappend-assoc lappend-llist-of-llist-of[symmetric]
..
         subgoal by fact .
    qed
   qed (unfold Q-def,blast) .
  qed
 } note lappend = this
 show ?thesis apply(rule lmap-lfilter-lappend-coind[of Q, OF Q lappend]) by auto
qed
```

```
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:
⋀w1 w2 lxs lxs'. P w1 w2 lxs lxs' ⟹
  lmap func (lfilter pred lxs) = lmap func' (lfilter pred' lxs') ∨
  (∃ v1 v2 ys llxs ys' llxs'.
      ((v1,w1) ∈ W1 ∨ ys ≠ []) ∧ ((v2,w2) ∈ W2 ∨ ys' ≠ []) ∧
      map func (filter pred ys) = map func' (filter pred' ys') ∧
      lxs = lappend (llist-of ys) llxs ∧ lxs' = lappend (llist-of ys') llxs' ∧
      P v1 v2 llxs llxs')
shows lmap func (lfilter pred lxs) = lmap func' (lfilter pred' lxs')
proof−
 {fix w1 w2  lxs lxs' assume P w1 w2 lxs lxs'
  hence lmap func (lfilter pred lxs) = lmap func' (lfilter pred' lxs') ∨
    (∃ v1 v2 ys llxs ys' llxs'.
      ys ≠ [] ∧ (ys' ≠ [] ∨ (v2,w2) ∈ trancl W2) ∧
      map func (filter pred ys) = map func' (filter pred' ys') ∧
      lxs = lappend (llist-of ys) llxs ∧ lxs' = lappend (llist-of ys') llxs' ∧
      P v1 v2 llxs llxs')
  proof(induct w1 arbitrary: w2 lxs lxs' rule: wf-induct-rule[OF W1])
   case (1 w1 w2 lxs lxs')
```

```
    show ?case using lappend[OF 1(2)] apply(elim disjE)
      subgoal by simp
      subgoal proof(elim exE conjE)
        fix v1 v2 ys llxs ys' llxs' assume vw1: (v1, w1) ∈ W1 ∨ ys ≠ []
        and vw2: (v2, w2) ∈ W2 ∨ ys' ≠ []
        and yss': map func (filter pred ys) = map func' (filter pred' ys')
        and lxs: lxs = lappend (llist-of ys) llxs
        and lxs': lxs' = lappend (llist-of ys') llxs'
        and P: P v1 v2 llxs llxs'
        show ?thesis
        proof(cases ys ≠ [])
          case True
          thus ?thesis using vw2 yss' lxs lxs' P by blast
        next
          case False note ys = False
          hence vw1: (v1, w1) ∈ W1 using vw1 by auto
          show ?thesis
          proof(cases lmap func (lfilter pred lxs) = lmap func' (lfilter pred' lxs'))
            case True
            thus ?thesis by simp
          next
            case False
            hence lmap func (lfilter pred llxs) ≠ lmap func' (lfilter pred' llxs')
            using yss' unfolding lxs lxs' by (auto simp: lmap-lappend-distrib)
            then obtain v1 vv2 yys lllxs yys' lllxs' where yyss': yys ≠ [] yys' ≠ []
∨ (vv2,v2) ∈ trancl W2
            map func (filter pred yys) = map func' (filter pred' yys')
            and llxs: llxs = lappend (llist-of yys) lllxs
            and llxs': llxs' = lappend (llist-of yys') lllxs'
            and P v1 vv2 lllxs lllxs'using 1(1)[OF vw1 P] by blast
            show ?thesis apply(rule disjI2)
            apply(rule exI[of - v1]) apply(rule exI[of - vv2])
            apply(rule exI[of - ys @ yys]) apply(rule exI[of - lllxs])
            apply(rule exI[of - ys' @ yys']) apply(rule exI[of - lllxs'])
            apply(intro conjI)
              subgoal using yyss'(1) by simp
              subgoal using yyss'(2) vw2 by auto
              subgoal apply simp unfolding yss' yyss' ..
          subgoal unfolding lxs llxs lappend-assoc lappend-llist-of-llist-of[symmetric]
..
          subgoal unfolding lxs' llxs' lappend-assoc lappend-llist-of-llist-of[symmetric]
..
              subgoal by fact .
        qed
      qed
    qed .
  qed
 } note lappend = this
```

define *Q* **where** $Q \equiv \lambda\ w2\ llxs\ llxs'.\ \exists\, w1.\ P\ w1\ w2\ llxs\ llxs'$
have *W2p*: *wf* $(W2^{+})$
using *W2 wf-trancl* **by** *blast*
have *Q*: *Q w2 lxs lxs'* **using** *P* **unfolding** *Q-def* **by** *auto*
show *?thesis* **apply**(*rule lmap-lfilter-lappend-coind-wf*[*OF W2p*, *of Q*, *OF Q*])
using *lappend* **unfolding** *Q-def* **by** *meson*
qed

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'*)
apply(*rule lmap-lfilter-lappend-coind-wf2*[*OF wf wf*, *of sameFM wL wR*])
subgoal using *assms* **by** *simp*
subgoal apply (*erule sameFM.cases*)
by *simp-all blast* **.**

end

end