Stream Fusion

Brian Huffman

May 26, 2024

Abstract

Stream Fusion [1] is a system for removing intermediate list structures from Haskell programs; it consists of a Haskell library along with several compiler rewrite rules. (The library is available online at http://www.cse.unsw.edu.au/~dons/streams.html.)

These theories contain a formalization of much of the Stream Fusion library in HOLCF. Lazy list and stream types are defined, along with coercions between the two types, as well as an equivalence relation for streams that generate the same list. List and stream versions of `map`, `filter`, `foldr`, `enumFromTo`, `append`, `zipWith`, and `concatMap` are defined, and the stream versions are shown to respect stream equivalence.

Contents

1 Lazy Lists

theory *LazyList*
imports *HOLCF HOLCF−Library.Int-Discrete*
begin

domain *'a LList* = *LNil* | *LCons* (**lazy** *'a*) (**lazy** *'a LList*)

fixrec
  mapL :: (*'a* → *'b*) → *'a LList* → *'b LList*
where
  mapL·*f*·*LNil* = *LNil*
| *mapL*·*f*·(*LCons*·*x*·*xs*) = *LCons*·(*f*·*x*)·(*mapL*·*f*·*xs*)

lemma *mapL-strict* [*simp*]: *mapL*·*f*·⊥ = ⊥
⟨*proof*⟩

fixrec
  filterL :: (*'a* → *tr*) → *'a LList* → *'a LList*
where
  filterL·*p*·*LNil* = *LNil*
| *filterL*·*p*·(*LCons*·*x*·*xs*) =
    (*If p*·*x then LCons*·*x*·(*filterL*·*p*·*xs*) *else filterL*·*p*·*xs*)

lemma *filterL-strict* [*simp*]: *filterL*·*p*·⊥ = ⊥
⟨*proof*⟩

fixrec
  foldrL :: (*'a* → *'b* → *'b*) → *'b* → *'a LList* → *'b*
where
  foldrL·*f*·*z*·*LNil* = *z*
| *foldrL*·*f*·*z*·(*LCons*·*x*·*xs*) = *f*·*x*·(*foldrL*·*f*·*z*·*xs*)

lemma *foldrL-strict* [*simp*]: *foldrL*·*f*·*z*·⊥ = ⊥
⟨*proof*⟩

fixrec
  enumFromToL :: $int_\perp$ → $int_\perp$ → ($int_\perp$) *LList*
where
  enumFromToL·(*up*·*x*)·(*up*·*y*) =
    (*if x* ≤ *y then LCons*·(*up*·*x*)·(*enumFromToL*·(*up*·(*x*+1))·(*up*·*y*)) *else LNil*)

lemma *enumFromToL-simps'* [*simp*]:
  x ≤ *y* ⟹
   enumFromToL·(*up*·*x*)·(*up*·*y*) = *LCons*·(*up*·*x*)·(*enumFromToL*·(*up*·(*x*+1))·(*up*·*y*))
  ¬ *x* ≤ *y* ⟹ *enumFromToL*·(*up*·*x*)·(*up*·*y*) = *LNil*
  ⟨*proof*⟩

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

lemma *enumFromToL-strict* [*simp*]:
  $enumFromToL{\cdot}\bot{\cdot}y = \bot$
  $enumFromToL{\cdot}x{\cdot}\bot = \bot$
⟨*proof*⟩

fixrec
  $appendL :: 'a\ LList \to 'a\ LList \to 'a\ LList$
where
  $appendL{\cdot}LNil{\cdot}ys = ys$
| $appendL{\cdot}(LCons{\cdot}x{\cdot}xs){\cdot}ys = LCons{\cdot}x{\cdot}(appendL{\cdot}xs{\cdot}ys)$

lemma *appendL-strict* [*simp*]: $appendL{\cdot}\bot{\cdot}ys = \bot$
⟨*proof*⟩

lemma *appendL-LNil-right*: $appendL{\cdot}xs{\cdot}LNil = xs$
⟨*proof*⟩

fixrec
  $zipWithL :: ('a \to 'b \to 'c) \to 'a\ LList \to 'b\ LList \to 'c\ LList$
where
  $zipWithL{\cdot}f{\cdot}LNil{\cdot}ys = LNil$
| $zipWithL{\cdot}f{\cdot}(LCons{\cdot}x{\cdot}xs){\cdot}LNil = LNil$
| $zipWithL{\cdot}f{\cdot}(LCons{\cdot}x{\cdot}xs){\cdot}(LCons{\cdot}y{\cdot}ys) = LCons{\cdot}(f{\cdot}x{\cdot}y){\cdot}(zipWithL{\cdot}f{\cdot}xs{\cdot}ys)$

lemma *zipWithL-strict* [*simp*]:
  $zipWithL{\cdot}f{\cdot}\bot{\cdot}ys = \bot$
  $zipWithL{\cdot}f{\cdot}(LCons{\cdot}x{\cdot}xs){\cdot}\bot = \bot$
⟨*proof*⟩

fixrec
  $concatMapL :: ('a \to 'b\ LList) \to 'a\ LList \to 'b\ LList$
where
  $concatMapL{\cdot}f{\cdot}LNil = LNil$
| $concatMapL{\cdot}f{\cdot}(LCons{\cdot}x{\cdot}xs) = appendL{\cdot}(f{\cdot}x){\cdot}(concatMapL{\cdot}f{\cdot}xs)$

lemma *concatMapL-strict* [*simp*]: $concatMapL{\cdot}f{\cdot}\bot = \bot$
⟨*proof*⟩

end

2 Stream Iterators

theory *Stream*
imports *LazyList*
begin

2.1 Type definitions for streams

Note that everything is strict in the state type.

domain ('a,'s) *Step* = *Done* | *Skip* 's | *Yield* (**lazy** 'a) 's

type-synonym ('a, 's) *Stepper* = 's → ('a, 's) *Step*

domain ('a,'s) *Stream* = *Stream* (**lazy** ('a, 's) *Stepper*) 's

2.2 Converting from streams to lists

fixrec
  unfold :: ('a, 's) *Stepper* −> ('s −> 'a *LList*)
where
  unfold·*h*·⊥ = ⊥
| *s* ≠ ⊥ ⟹
    unfold·*h*·*s* =
      (*case h*·*s of*
        Done ⇒ *LNil*
      | *Skip*·*s'* ⇒ *unfold*·*h*·*s'*
      | *Yield*·*x*·*s'* ⇒ *LCons*·*x*·(*unfold*·*h*·*s'*))

fixrec
  unfoldF :: ('a, 's) *Stepper* → ('s → 'a *LList*) → ('s → 'a *LList*)
where
  unfoldF·*h*·*u*·⊥ = ⊥
| *s* ≠ ⊥ ⟹
    unfoldF·*h*·*u*·*s* =
      (*case h*·*s of*
        Done ⇒ *LNil*
      | *Skip*·*s'* ⇒ *u*·*s'*
      | *Yield*·*x*·*s'* ⇒ *LCons*·*x*·(*u*·*s'*))

lemma *unfold-eq-fix*: *unfold*·*h* = *fix*·(*unfoldF*·*h*)
⟨*proof*⟩

lemma *unfold-ind*:
  fixes *P* :: ('s → 'a *LList*) ⇒ *bool*
  assumes *adm P* **and** *P* ⊥ **and** ⋀*u*. *P u* ⟹ *P* (*unfoldF*·*h*·*u*)
  shows *P* (*unfold*·*h*)
⟨*proof*⟩

fixrec
  unfold2 :: ('s → 'a *LList*) → ('a, 's) *Step* → 'a *LList*
where
  unfold2·*u*·*Done* = *LNil*
| *s* ≠ ⊥ ⟹ *unfold2*·*u*·(*Skip*·*s*) = *u*·*s*
| *s* ≠ ⊥ ⟹ *unfold2*·*u*·(*Yield*·*x*·*s*) = *LCons*·*x*·(*u*·*s*)

lemma *unfold2-strict* [*simp*]: $unfold2{\cdot}u{\cdot}\bot = \bot$
⟨*proof*⟩

lemma *unfold*: $s \neq \bot \Longrightarrow unfold{\cdot}h{\cdot}s = unfold2{\cdot}(unfold{\cdot}h){\cdot}(h{\cdot}s)$
⟨*proof*⟩

lemma *unfoldF*: $s \neq \bot \Longrightarrow unfoldF{\cdot}h{\cdot}u{\cdot}s = unfold2{\cdot}u{\cdot}(h{\cdot}s)$
⟨*proof*⟩

declare *unfold.simps*(*2*) [*simp del*]
declare *unfoldF.simps*(*2*) [*simp del*]
declare *unfoldF* [*simp*]

fixrec
  unstream :: ('*a*, '*s*) *Stream* → '*a LList*
where
  $s \neq \bot \Longrightarrow unstream{\cdot}(Stream{\cdot}h{\cdot}s) = unfold{\cdot}h{\cdot}s$

lemma *unstream-strict* [*simp*]: $unstream{\cdot}\bot = \bot$
⟨*proof*⟩

2.3 Converting from lists to streams

fixrec
  streamStep :: ('*a LList*)$_\bot$ → ('*a*, ('*a LList*)$_\bot$) *Step*
where
  $streamStep{\cdot}(up{\cdot}LNil) = Done$
| $streamStep{\cdot}(up{\cdot}(LCons{\cdot}x{\cdot}xs)) = Yield{\cdot}x{\cdot}(up{\cdot}xs)$

lemma *streamStep-strict* [*simp*]: $streamStep{\cdot}(up{\cdot}\bot) = \bot$
⟨*proof*⟩

fixrec
  stream :: '*a LList* → ('*a*, ('*a LList*)$_\bot$) *Stream*
where
  $stream{\cdot}xs = Stream{\cdot}streamStep{\cdot}(up{\cdot}xs)$

lemma *stream-defined* [*simp*]: $stream{\cdot}xs \neq \bot$
  ⟨*proof*⟩

lemma *unstream-stream* [*simp*]:
  fixes *xs* :: '*a LList*
  shows $unstream{\cdot}(stream{\cdot}xs) = xs$
⟨*proof*⟩

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

2.4 Bisimilarity relation on streams

definition

bisimilar :: ('*a*, '*s*) *Stream* ⇒ ('*a*, '*t*) *Stream* ⇒ *bool* (**infix** ≈ *50*)
where
a ≈ *b* ⟷ *unstream*·*a* = *unstream*·*b* ∧ *a* ≠ ⊥ ∧ *b* ≠ ⊥

lemma *unstream-cong*:
a ≈ *b* ⟹ *unstream*·*a* = *unstream*·*b*
⟨*proof*⟩

lemma *stream-cong*:
xs = *ys* ⟹ *stream*·*xs* ≈ *stream*·*ys*
⟨*proof*⟩

lemma *stream-unstream-cong*:
a ≈ *b* ⟹ *stream*·(*unstream*·*a*) ≈ *b*
⟨*proof*⟩

end

3 Stream Fusion

theory *StreamFusion*
imports *Stream*
begin

3.1 Type constructors for state types

domain *Switch* = *S1* | *S2*

domain '*a Maybe* = *Nothing* | *Just* '*a*

hide-const (**open**) *Left Right*

domain ('*a*, '*b*) *Either* = *Left* '*a* | *Right* '*b*

domain ('*a*, '*b*) *Both* (**infixl** :!: *25*) = *Both* '*a* '*b* (**infixl** :!: *75*)

domain '*a L* = *L* (**lazy** '*a*)

3.2 Map function

fixrec
mapStep :: ('*a* → '*b*) → ('*s* → ('*a*, '*s*) *Step*) → '*s* → ('*b*, '*s*) *Step*
where
mapStep·*f*·*h*·⊥ = ⊥
| *s* ≠ ⊥ ⟹ *mapStep*·*f*·*h*·*s* = (*case h*·*s of*
Done ⇒ *Done*
| *Skip*·*s*′ ⇒ *Skip*·*s*′
| *Yield*·*x*·*s*′ ⇒ *Yield*·(*f*·*x*)·*s*′)

fixrec
  $mapS :: ('a \to 'b) \to ('a,\ 's)\ Stream \to ('b,\ 's)\ Stream$
where
  $s \neq \bot \Longrightarrow mapS{\cdot}f{\cdot}(Stream{\cdot}h{\cdot}s) = Stream{\cdot}(mapStep{\cdot}f{\cdot}h){\cdot}s$

lemma *unfold-mapStep*:
  fixes $f :: 'a \to 'b$ **and** $h :: 's \to ('a,\ 's)\ Step$
  assumes $s \neq \bot$
  shows $unfold{\cdot}(mapStep{\cdot}f{\cdot}h){\cdot}s = mapL{\cdot}f{\cdot}(unfold{\cdot}h{\cdot}s)$
⟨*proof*⟩

lemma *unstream-mapS*:
  fixes $f :: 'a \to 'b$ **and** $a :: ('a,\ 's)\ Stream$
  shows $a \neq \bot \Longrightarrow unstream{\cdot}(mapS{\cdot}f{\cdot}a) = mapL{\cdot}f{\cdot}(unstream{\cdot}a)$
⟨*proof*⟩

lemma *mapS-defined*: $a \neq \bot \Longrightarrow mapS{\cdot}f{\cdot}a \neq \bot$
⟨*proof*⟩

lemma *mapS-cong*:
  fixes $f :: 'a \to 'b$
  fixes $a :: ('a,\ 's)\ Stream$
  fixes $b :: ('a,\ 't)\ Stream$
  shows $f = g \Longrightarrow a \approx b \Longrightarrow mapS{\cdot}f{\cdot}a \approx mapS{\cdot}g{\cdot}b$
⟨*proof*⟩

lemma *mapL-eq*: $mapL{\cdot}f{\cdot}xs = unstream{\cdot}(mapS{\cdot}f{\cdot}(stream{\cdot}xs))$
⟨*proof*⟩

3.3 Filter function

fixrec
  $filterStep :: ('a \to tr) \to ('s \to ('a,\ 's)\ Step) \to 's \to ('a,\ 's)\ Step$
where
  $filterStep{\cdot}p{\cdot}h{\cdot}\bot = \bot$
| $s \neq \bot \Longrightarrow filterStep{\cdot}p{\cdot}h{\cdot}s = (case\ h{\cdot}s\ of$
    $Done \Rightarrow Done$
  | $Skip{\cdot}s' \Rightarrow Skip{\cdot}s'$
  | $Yield{\cdot}x{\cdot}s' \Rightarrow (If\ p{\cdot}x\ then\ Yield{\cdot}x{\cdot}s'\ else\ Skip{\cdot}s'))$

fixrec
  $filterS :: ('a \to tr) \to ('a,\ 's)\ Stream \to ('a,\ 's)\ Stream$
where
  $s \neq \bot \Longrightarrow filterS{\cdot}p{\cdot}(Stream{\cdot}h{\cdot}s) = Stream{\cdot}(filterStep{\cdot}p{\cdot}h){\cdot}s$

lemma *unfold-filterStep*:
  fixes $p :: 'a \to tr$ **and** $h :: 's \to ('a,\ 's)\ Step$
  assumes $s \neq \bot$
  shows $unfold{\cdot}(filterStep{\cdot}p{\cdot}h){\cdot}s = filterL{\cdot}p{\cdot}(unfold{\cdot}h{\cdot}s)$

⟨*proof*⟩

lemma *unstream-filterS*:
  $a \neq \bot \Longrightarrow unstream{\cdot}(filterS{\cdot}p{\cdot}a) = filterL{\cdot}p{\cdot}(unstream{\cdot}a)$
⟨*proof*⟩

lemma *filterS-defined*: $a \neq \bot \Longrightarrow filterS{\cdot}p{\cdot}a \neq \bot$
⟨*proof*⟩

lemma *filterS-cong*:
  fixes $p :: 'a \to tr$
  fixes $a :: ('a, 's)\ Stream$
  fixes $b :: ('a, 't)\ Stream$
  shows $p = q \Longrightarrow a \approx b \Longrightarrow filterS{\cdot}p{\cdot}a \approx filterS{\cdot}q{\cdot}b$
⟨*proof*⟩

lemma *filterL-eq*: $filterL{\cdot}p{\cdot}xs = unstream{\cdot}(filterS{\cdot}p{\cdot}(stream{\cdot}xs))$
⟨*proof*⟩

3.4 Foldr function

fixrec
  $foldrS :: ('a \to 'b \to 'b) \to 'b \to ('a, 's)\ Stream \to 'b$
where
  foldrS-Stream:
  $s \neq \bot \Longrightarrow foldrS{\cdot}f{\cdot}z{\cdot}(Stream{\cdot}h{\cdot}s) =$
    $(case\ h{\cdot}s\ of\ Done \Rightarrow z$
      $\mid Skip{\cdot}s' \Rightarrow foldrS{\cdot}f{\cdot}z{\cdot}(Stream{\cdot}h{\cdot}s')$
      $\mid Yield{\cdot}x{\cdot}s' \Rightarrow f{\cdot}x{\cdot}(foldrS{\cdot}f{\cdot}z{\cdot}(Stream{\cdot}h{\cdot}s')))$

lemma *unfold-foldrS*:
  assumes $s \neq \bot$ **shows** $foldrS{\cdot}f{\cdot}z{\cdot}(Stream{\cdot}h{\cdot}s) = foldrL{\cdot}f{\cdot}z{\cdot}(unfold{\cdot}h{\cdot}s)$
⟨*proof*⟩

lemma *unstream-foldrS*:
  $a \neq \bot \Longrightarrow foldrS{\cdot}f{\cdot}z{\cdot}a = foldrL{\cdot}f{\cdot}z{\cdot}(unstream{\cdot}a)$
⟨*proof*⟩

lemma *foldrS-cong*:
  fixes $a :: ('a, 's)\ Stream$
  fixes $b :: ('a, 't)\ Stream$
  shows $f = g \Longrightarrow z = w \Longrightarrow a \approx b \Longrightarrow foldrS{\cdot}f{\cdot}z{\cdot}a = foldrS{\cdot}g{\cdot}w{\cdot}b$
⟨*proof*⟩

lemma *foldrL-eq*:
  $foldrL{\cdot}f{\cdot}z{\cdot}xs = foldrS{\cdot}f{\cdot}z{\cdot}(stream{\cdot}xs)$
⟨*proof*⟩

3.5 EnumFromTo function

type-synonym $int' = int_\perp$

fixrec
  $enumFromToStep :: int' \to (int')_\perp \to (int', (int')_\perp)\ Step$
where
  $enumFromToStep{\cdot}(up{\cdot}y){\cdot}(up{\cdot}(up{\cdot}x)) =$
    $(if\ x \leq y\ then\ Yield{\cdot}(up{\cdot}x){\cdot}(up{\cdot}(up{\cdot}(x{+}1)))\ else\ Done)$

lemma *enumFromToStep-strict* [*simp*]:
  $enumFromToStep{\cdot}\perp{\cdot}x'' = \perp$
  $enumFromToStep{\cdot}(up{\cdot}y){\cdot}\perp = \perp$
  $enumFromToStep{\cdot}(up{\cdot}y){\cdot}(up{\cdot}\perp) = \perp$
⟨*proof*⟩

lemma *enumFromToStep-simps'* [*simp*]:
  $x \leq y \Longrightarrow enumFromToStep{\cdot}(up{\cdot}y){\cdot}(up{\cdot}(up{\cdot}x)) =$
    $Yield{\cdot}(up{\cdot}x){\cdot}(up{\cdot}(up{\cdot}(x{+}1)))$
  $\neg\ x \leq y \Longrightarrow enumFromToStep{\cdot}(up{\cdot}y){\cdot}(up{\cdot}(up{\cdot}x)) = Done$
  ⟨*proof*⟩

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

fixrec
  $enumFromToS :: int' \to int' \to (int', (int')_\perp)\ Stream$
where
  $enumFromToS{\cdot}x{\cdot}y = Stream{\cdot}(enumFromToStep{\cdot}y){\cdot}(up{\cdot}x)$

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

lemma *unfold-enumFromToStep*:
  $unfold{\cdot}(enumFromToStep{\cdot}(up{\cdot}y)){\cdot}(up{\cdot}n) = enumFromToL{\cdot}n{\cdot}(up{\cdot}y)$
⟨*proof*⟩

lemma *unstream-enumFromToS*:
  $unstream{\cdot}(enumFromToS{\cdot}x{\cdot}y) = enumFromToL{\cdot}x{\cdot}y$
⟨*proof*⟩

lemma *enumFromToS-defined*: $enumFromToS{\cdot}x{\cdot}y \neq \perp$
  ⟨*proof*⟩

lemma *enumFromToS-cong*:
  $x = x' \Longrightarrow y = y' \Longrightarrow enumFromToS{\cdot}x{\cdot}y \approx enumFromToS{\cdot}x'{\cdot}y'$
⟨*proof*⟩

lemma *enumFromToL-eq*: $enumFromToL{\cdot}x{\cdot}y = unstream{\cdot}(enumFromToS{\cdot}x{\cdot}y)$
⟨*proof*⟩

3.6 Append function

fixrec
  appendStep ::
    $('s \to ('a, 's)\ Step) \to$
     $('t \to ('a, 't)\ Step) \to$
     $'t \to ('s, 't)\ Either \to ('a, ('s, 't)\ Either)\ Step$
where
  $sa \neq \bot \Longrightarrow appendStep{\cdot}ha{\cdot}hb{\cdot}sb0{\cdot}(Left{\cdot}sa) =$
    $(case\ ha{\cdot}sa\ of$
      $Done \Rightarrow Skip{\cdot}(Right{\cdot}sb0)$
    $\mid Skip{\cdot}sa' \Rightarrow Skip{\cdot}(Left{\cdot}sa')$
    $\mid Yield{\cdot}x{\cdot}sa' \Rightarrow Yield{\cdot}x{\cdot}(Left{\cdot}sa'))$
$\mid sb \neq \bot \Longrightarrow appendStep{\cdot}ha{\cdot}hb{\cdot}sb0{\cdot}(Right{\cdot}sb) =$
    $(case\ hb{\cdot}sb\ of$
      $Done \Rightarrow Done$
    $\mid Skip{\cdot}sb' \Rightarrow Skip{\cdot}(Right{\cdot}sb')$
    $\mid Yield{\cdot}x{\cdot}sb' \Rightarrow Yield{\cdot}x{\cdot}(Right{\cdot}sb'))$

lemma *appendStep-strict* [*simp*]: $appendStep{\cdot}ha{\cdot}hb{\cdot}sb0{\cdot}\bot = \bot$
⟨*proof*⟩

fixrec
  appendS ::
    $('a, 's)\ Stream \to ('a, 't)\ Stream \to ('a, ('s, 't)\ Either)\ Stream$
where
  $sa0 \neq \bot \Longrightarrow sb0 \neq \bot \Longrightarrow$
    $appendS{\cdot}(Stream{\cdot}ha{\cdot}sa0){\cdot}(Stream{\cdot}hb{\cdot}sb0) =$
      $Stream{\cdot}(appendStep{\cdot}ha{\cdot}hb{\cdot}sb0){\cdot}(Left{\cdot}sa0)$

lemma *unfold-appendStep*:
  fixes $ha :: 's \to ('a, 's)\ Step$
  fixes $hb :: 't \to ('a, 't)\ Step$
  assumes *sb0* [*simp*]: $sb0 \neq \bot$
  shows
  $(\forall sa.\ sa \neq \bot \longrightarrow unfold{\cdot}(appendStep{\cdot}ha{\cdot}hb{\cdot}sb0){\cdot}(Left{\cdot}sa) =$
        $appendL{\cdot}(unfold{\cdot}ha{\cdot}sa){\cdot}(unfold{\cdot}hb{\cdot}sb0)) \wedge$
   $(\forall sb.\ sb \neq \bot \longrightarrow unfold{\cdot}(appendStep{\cdot}ha{\cdot}hb{\cdot}sb0){\cdot}(Right{\cdot}sb) =$
        $unfold{\cdot}hb{\cdot}sb)$
⟨*proof*⟩

lemma *appendS-defined*: $xs \neq \bot \Longrightarrow ys \neq \bot \Longrightarrow appendS{\cdot}xs{\cdot}ys \neq \bot$
⟨*proof*⟩

lemma *unstream-appendS*:
  $a \neq \bot \Longrightarrow b \neq \bot \Longrightarrow$
    $unstream{\cdot}(appendS{\cdot}a{\cdot}b) = appendL{\cdot}(unstream{\cdot}a){\cdot}(unstream{\cdot}b)$
⟨*proof*⟩

lemma *appendS-cong*:

fixes $f :: 'a \to 'b$
fixes $a :: ('a, 's)\ Stream$
fixes $b :: ('a, 't)\ Stream$
shows $a \approx a' \Longrightarrow b \approx b' \Longrightarrow appendS{\cdot}a{\cdot}b \approx appendS{\cdot}a'{\cdot}b'$
⟨*proof*⟩

lemma *appendL-eq*: $appendL{\cdot}xs{\cdot}ys = unstream{\cdot}(appendS{\cdot}(stream{\cdot}xs){\cdot}(stream{\cdot}ys))$
⟨*proof*⟩

3.7 ZipWith function

fixrec
zipWithStep ::
($'a \to 'b \to 'c$) →
($'s \to ('a, 's)\ Step$) →
($'t \to ('b, 't)\ Step$) →
$'s$:!: $'t$:!: $'a\ L\ Maybe \to ('c, 's$:!: $'t$:!: $'a\ L\ Maybe)\ Step$
where
$sa \neq \bot \Longrightarrow sb \neq \bot \Longrightarrow$
$zipWithStep{\cdot}f{\cdot}ha{\cdot}hb{\cdot}(sa$:!: sb :!: $Nothing) =$
(*case* $ha{\cdot}sa$ *of*
Done ⇒ *Done*
| $Skip{\cdot}sa' \Rightarrow Skip{\cdot}(sa'$:!: sb :!: $Nothing)$
| $Yield{\cdot}a{\cdot}sa' \Rightarrow Skip{\cdot}(sa'$:!: sb :!: $Just{\cdot}(L{\cdot}a)))$
| $sa \neq \bot \Longrightarrow sb \neq \bot \Longrightarrow$
$zipWithStep{\cdot}f{\cdot}ha{\cdot}hb{\cdot}(sa$:!: sb :!: $Just{\cdot}(L{\cdot}a)) =$
(*case* $hb{\cdot}sb$ *of*
Done ⇒ *Done*
| $Skip{\cdot}sb' \Rightarrow Skip{\cdot}(sa$:!: sb' :!: $Just{\cdot}(L{\cdot}a))$
| $Yield{\cdot}b{\cdot}sb' \Rightarrow Yield{\cdot}(f{\cdot}a{\cdot}b){\cdot}(sa$:!: sb' :!: $Nothing))$

lemma *zipWithStep-strict* [*simp*]: $zipWithStep{\cdot}f{\cdot}ha{\cdot}hb{\cdot}\bot = \bot$
⟨*proof*⟩

fixrec
$zipWithS :: ('a \to 'b \to 'c) \to$
$('a, 's)\ Stream \to ('b, 't)\ Stream \to ('c, 's$:!: $'t$:!: $'a\ L\ Maybe)\ Stream$
where
$sa0 \neq \bot \Longrightarrow sb0 \neq \bot \Longrightarrow zipWithS{\cdot}f{\cdot}(Stream{\cdot}ha{\cdot}sa0){\cdot}(Stream{\cdot}hb{\cdot}sb0) =$
$Stream{\cdot}(zipWithStep{\cdot}f{\cdot}ha{\cdot}hb){\cdot}(sa0$:!: $sb0$:!: $Nothing)$

lemma *zipWithS-fix-ind-lemma*:
fixes $P\ Q :: nat \Rightarrow nat \Rightarrow bool$
assumes *P-0*: $\bigwedge j.\ P\ 0\ j$ **and** *P-Suc*: $\bigwedge i\ j.\ P\ i\ j \Longrightarrow Q\ i\ j \Longrightarrow P\ (Suc\ i)\ j$
assumes *Q-0*: $\bigwedge i.\ Q\ i\ 0$ **and** *Q-Suc*: $\bigwedge i\ j.\ P\ i\ j \Longrightarrow Q\ i\ j \Longrightarrow Q\ i\ (Suc\ j)$
shows $P\ i\ j \wedge Q\ i\ j$
⟨*proof*⟩

lemma *zipWithS-fix-ind*:

assumes x: $x = fix{\cdot}f$ **and** y: $y = fix{\cdot}g$
assumes *adm-P*: $adm\ (\lambda x.\ P\ (fst\ x)\ (snd\ x))$
assumes *adm-Q*: $adm\ (\lambda x.\ Q\ (fst\ x)\ (snd\ x))$
assumes *P-0*: $\bigwedge b.\ P\ \bot\ b$ **and** *P-Suc*: $\bigwedge a\ b.\ P\ a\ b \Longrightarrow Q\ a\ b \Longrightarrow P\ (f{\cdot}a)\ b$
assumes *Q-0*: $\bigwedge a.\ Q\ a\ \bot$ **and** *Q-Suc*: $\bigwedge a\ b.\ P\ a\ b \Longrightarrow Q\ a\ b \Longrightarrow Q\ a\ (g{\cdot}b)$
shows $P\ x\ y \wedge Q\ x\ y$
⟨*proof*⟩

lemma *unfold-zipWithStep*:
fixes $f :: 'a \to 'b \to 'c$
fixes $ha :: 's \to ('a, 's)\ Step$
fixes $hb :: 't \to ('b, 't)\ Step$
defines *h-def*: $h \equiv zipWithStep{\cdot}f{\cdot}ha{\cdot}hb$
shows
$(\forall\, sa\ sb.\ sa \neq \bot \longrightarrow sb \neq \bot \longrightarrow$
$unfold{\cdot}h{\cdot}(sa :!: sb :!: Nothing) =$
$zipWithL{\cdot}f{\cdot}(unfold{\cdot}ha{\cdot}sa){\cdot}(unfold{\cdot}hb{\cdot}sb)) \wedge$
$(\forall\, sa\ sb\ a.\ sa \neq \bot \longrightarrow sb \neq \bot \longrightarrow$
$unfold{\cdot}h{\cdot}(sa :!: sb :!: Just{\cdot}(L{\cdot}a)) =$
$zipWithL{\cdot}f{\cdot}(LCons{\cdot}a{\cdot}(unfold{\cdot}ha{\cdot}sa)){\cdot}(unfold{\cdot}hb{\cdot}sb))$
⟨*proof*⟩

lemma *zipWithS-defined*: $a \neq \bot \Longrightarrow b \neq \bot \Longrightarrow zipWithS{\cdot}f{\cdot}a{\cdot}b \neq \bot$
⟨*proof*⟩

lemma *unstream-zipWithS*:
$a \neq \bot \Longrightarrow b \neq \bot \Longrightarrow$
$unstream{\cdot}(zipWithS{\cdot}f{\cdot}a{\cdot}b) = zipWithL{\cdot}f{\cdot}(unstream{\cdot}a){\cdot}(unstream{\cdot}b)$
⟨*proof*⟩

lemma *zipWithS-cong*:
$f = f' \Longrightarrow a \approx a' \Longrightarrow b \approx b' \Longrightarrow$
$zipWithS{\cdot}f{\cdot}a{\cdot}b \approx zipWithS{\cdot}f{\cdot}a'{\cdot}b'$
⟨*proof*⟩

lemma *zipWithL-eq*:
$zipWithL{\cdot}f{\cdot}xs{\cdot}ys = unstream{\cdot}(zipWithS{\cdot}f{\cdot}(stream{\cdot}xs){\cdot}(stream{\cdot}ys))$
⟨*proof*⟩

3.8 ConcatMap function

fixrec
concatMapStep ::
$('a \to ('b, 't)\ Stream) \to$
$('s \to ('a, 's)\ Step) \to$
$'s :!: ('b, 't)\ Stream\ Maybe \to$
$('b, 's :!: ('b, 't)\ Stream\ Maybe)\ Step$
where
$sa \neq \bot \Longrightarrow concatMapStep{\cdot}f{\cdot}ha{\cdot}(sa :!: Nothing) =$

```
    (case ha·sa of
      Done ⇒ Done
    | Skip·sa' ⇒ Skip·(sa' :!: Nothing)
    | Yield·a·sa' ⇒ Skip·(sa' :!: Just·(f·a)))
| sa ≠ ⊥ ⟹ sb ≠ ⊥ ⟹
    concatMapStep·f·ha·(sa :!: Just·(Stream·hb·sb)) =
    (case hb·sb of
      Done ⇒ Skip·(sa :!: Nothing)
    | Skip·sb' ⇒ Skip·(sa :!: Just·(Stream·hb·sb'))
    | Yield·b·sb' ⇒ Yield·b·(sa :!: Just·(Stream·hb·sb')))

lemma concatMapStep-strict [simp]: concatMapStep·f·ha·⊥ = ⊥
⟨proof⟩

fixrec
  concatMapS ::
    ('a → ('b, 't) Stream) → ('a, 's) Stream →
     ('b, 's :!: ('b, 't) Stream Maybe) Stream
where
  s ≠ ⊥ ⟹ concatMapS·f·(Stream·h·s) = Stream·(concatMapStep·f·h)·(s :!: Nothing)

lemma concatMapS-strict [simp]: concatMapS·f·⊥ = ⊥
⟨proof⟩

lemma unfold-concatMapStep:
  fixes ha :: 's → ('a, 's) Step
  fixes f :: 'a → ('b, 't) Stream
  defines h-def: h ≡ concatMapStep·f·ha
  defines f'-def: f' ≡ unstream oo f
  shows
  (∀ sa. sa ≠ ⊥ ⟶
      unfold·h·(sa :!: Nothing) = concatMapL·f'·(unfold·ha·sa)) ∧
   (∀ sa hb sb. sa ≠ ⊥ ⟶ sb ≠ ⊥ ⟶
      unfold·h·(sa :!: Just·(Stream·hb·sb)) =
        appendL·(unfold·hb·sb)·(concatMapL·f'·(unfold·ha·sa)))
⟨proof⟩

lemma unstream-concatMapS:
  unstream·(concatMapS·f·a) = concatMapL·(unstream oo f)·(unstream·a)
⟨proof⟩

lemma concatMapS-defined: a ≠ ⊥ ⟹ concatMapS·f·a ≠ ⊥
⟨proof⟩

lemma concatMapS-cong:
  fixes f :: 'a ⇒ ('b, 's) Stream
  fixes g :: 'a ⇒ ('b, 't) Stream
  fixes a :: ('a, 'u) Stream
```

fixes $b :: ('a, 'v)\ Stream$
shows $(\bigwedge x.\ f\ x \approx g\ x) \Longrightarrow a \approx b \Longrightarrow cont\ f \Longrightarrow cont\ g \Longrightarrow$
$concatMapS{\cdot}(\Lambda\ x.\ f\ x){\cdot}a \approx concatMapS{\cdot}(\Lambda\ x.\ g\ x){\cdot}b$
⟨*proof*⟩

lemma *concatMapL-eq*:
$concatMapL{\cdot}f{\cdot}xs = unstream{\cdot}(concatMapS{\cdot}(stream\ oo\ f){\cdot}(stream{\cdot}xs))$
⟨*proof*⟩

3.9 Examples

lemmas *stream-eqs* =
mapL-eq
filterL-eq
foldrL-eq
enumFromToL-eq
appendL-eq
zipWithL-eq
concatMapL-eq

lemmas *stream-congs* =
unstream-cong
stream-cong
stream-unstream-cong
mapS-cong
filterS-cong
foldrS-cong
enumFromToS-cong
appendS-cong
zipWithS-cong
concatMapS-cong

lemma
$mapL{\cdot}f\ oo\ filterL{\cdot}p\ oo\ mapL{\cdot}g =$
$unstream\ oo\ mapS{\cdot}f\ oo\ filterS{\cdot}p\ oo\ mapS{\cdot}g\ oo\ stream$
⟨*proof*⟩

lemma
$foldrL{\cdot}f{\cdot}z{\cdot}(mapL{\cdot}g{\cdot}(filterL{\cdot}p{\cdot}(enumFromToL{\cdot}x{\cdot}y))) =$
$foldrS{\cdot}f{\cdot}z{\cdot}(mapS{\cdot}g{\cdot}(filterS{\cdot}p{\cdot}(enumFromToS{\cdot}x{\cdot}y)))$
⟨*proof*⟩

lemma *oo-LAM* [*simp*]: $cont\ g \Longrightarrow f\ oo\ (\Lambda\ x.\ g\ x) = (\Lambda\ x.\ f{\cdot}(g\ x))$
⟨*proof*⟩

lemma
$concatMapL{\cdot}(\Lambda\ k.$
$mapL{\cdot}(\Lambda\ m.\ f{\cdot}k{\cdot}m){\cdot}(enumFromToL{\cdot}one{\cdot}k)){\cdot}(enumFromToL{\cdot}one{\cdot}n) =$
$unstream{\cdot}(concatMapS{\cdot}(\Lambda\ k.$

$mapS \cdot (\Lambda\ m.\ f \cdot k \cdot m) \cdot (enumFromToS \cdot one \cdot k)) \cdot (enumFromToS \cdot one \cdot n))$
$\langle proof \rangle$

end

References

[1] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: From lists to streams to nothing at all. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2007*, Apr. 2007.