

Stream Fusion

Brian Huffman

October 13, 2025

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	2
2	Stream Iterators	3
2.1	Type definitions for streams	4
2.2	Converting from streams to lists	4
2.3	Converting from lists to streams	5
2.4	Bisimilarity relation on streams	6
3	Stream Fusion	6
3.1	Type constructors for state types	6
3.2	Map function	7
3.3	Filter function	8
3.4	Foldr function	9
3.5	EnumFromTo function	10
3.6	Append function	11
3.7	ZipWith function	14
3.8	ConcatMap function	17
3.9	Examples	20

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.⊥ = ⊥
by fixrec-simp

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.⊥ = ⊥
by fixrec-simp

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.⊥ = ⊥
by fixrec-simp

fixrec
  enumFromToL :: int⊥ → int⊥ → (int⊥) 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
by simp-all

declare enumFromToL.simps [simp del]

```

```

lemma enumFromToL-strict [simp]:
  enumFromToL. $\perp$ .y =  $\perp$ 
  enumFromToL.x. $\perp$  =  $\perp$ 
apply (subst enumFromToL.unfold, simp)
apply (induct x)
apply (subst enumFromToL.unfold, simp)+
done

fixrec
  appendL :: 'a LList  $\rightarrow$  'a LList  $\rightarrow$  'a LList
where
  appendL.LNil.ys = ys
  | appendL.(LCons.x.xs).ys = LCons.x.(appendL.xs.ys)

lemma appendL-strict [simp]: appendL. $\perp$ .ys =  $\perp$ 
by fixrec-simp

lemma appendL-LNil-right: appendL.xs.LNil = xs
by (induct xs) simp-all

fixrec
  zipWithL :: ('a  $\rightarrow$  'b  $\rightarrow$  'c)  $\rightarrow$  'a LList  $\rightarrow$  'b LList  $\rightarrow$  'c LList
where
  zipWithL.f.LNil.ys = LNil
  | zipWithL.f.(LCons.x.xs).LNil = LNil
  | zipWithL.f.(LCons.x.xs).(LCons.y.ys) = LCons.(f.x.y).(zipWithL.f.xs.ys)

lemma zipWithL-strict [simp]:
  zipWithL.f. $\perp$ .ys =  $\perp$ 
  zipWithL.f.(LCons.x.xs). $\perp$  =  $\perp$ 
by fixrec-simp+

fixrec
  concatMapL :: ('a  $\rightarrow$  'b LList)  $\rightarrow$  'a LList  $\rightarrow$  'b LList
where
  concatMapL.f.LNil = LNil
  | concatMapL.f.(LCons.x.xs) = appendL.(f.x).(concatMapL.f.xs)

lemma concatMapL-strict [simp]: concatMapL.f. $\perp$  =  $\perp$ 
by fixrec-simp

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 $(\text{'a}, \text{'s}) \text{ Step} = \text{Done} \mid \text{Skip } \text{'s} \mid \text{Yield } (\text{lazy } \text{'a}) \text{'s}$

type-synonym $(\text{'a}, \text{'s}) \text{ Stepper} = \text{'s} \rightarrow (\text{'a}, \text{'s}) \text{ Step}$

domain $(\text{'a}, \text{'s}) \text{ Stream} = \text{Stream } (\text{lazy } (\text{'a}, \text{'s}) \text{ Stepper}) \text{'s}$

2.2 Converting from streams to lists

fixrec

$\text{unfold} :: (\text{'a}, \text{'s}) \text{ Stepper} \rightarrow (\text{'s} \rightarrow \text{'a } LList)$

where

$\text{unfold} \cdot h \cdot \perp = \perp$
 $\mid s \neq \perp \implies$
 $\text{unfold} \cdot h \cdot s =$
 $(\text{case } h \cdot s \text{ of}$
 $\quad \text{Done} \Rightarrow LNil$
 $\mid \text{Skip} \cdot s' \Rightarrow \text{unfold} \cdot h \cdot s'$
 $\mid \text{Yield} \cdot x \cdot s' \Rightarrow LCons \cdot x \cdot (\text{unfold} \cdot h \cdot s'))$

fixrec

$\text{unfoldF} :: (\text{'a}, \text{'s}) \text{ Stepper} \rightarrow (\text{'s} \rightarrow \text{'a } LList) \rightarrow (\text{'s} \rightarrow \text{'a } LList)$

where

$\text{unfoldF} \cdot h \cdot u \cdot \perp = \perp$
 $\mid s \neq \perp \implies$
 $\text{unfoldF} \cdot h \cdot u \cdot s =$
 $(\text{case } h \cdot s \text{ of}$
 $\quad \text{Done} \Rightarrow LNil$
 $\mid \text{Skip} \cdot s' \Rightarrow u \cdot s'$
 $\mid \text{Yield} \cdot x \cdot s' \Rightarrow LCons \cdot x \cdot (u \cdot s'))$

lemma unfold-eq-fix : $\text{unfold} \cdot h = \text{fix} \cdot (\text{unfoldF} \cdot h)$

proof (rule below-antisym)

show $\text{unfold} \cdot h \sqsubseteq \text{fix} \cdot (\text{unfoldF} \cdot h)$

apply (rule unfold.induct, simp, simp)

apply (subst fix-eq)

apply (rule cfun-belowI, rename-tac s)

apply (case-tac s = $\perp$, simp, simp)

apply (intro monofun-cfun monofun-LAM below-refl, simp-all)

done

show $\text{fix} \cdot (\text{unfoldF} \cdot h) \sqsubseteq \text{unfold} \cdot h$

apply (rule fix-ind, simp, simp)

apply (subst unfold.unfold)

apply (rule cfun-belowI, rename-tac s)

apply (case-tac s = $\perp$, simp, simp)

apply (intro monofun-cfun monofun-LAM below-refl, simp-all)

done

qed

lemma *unfold-ind*:

fixes $P :: ('s \rightarrow 'a \text{ LList}) \Rightarrow \text{bool}$

assumes $\text{adm } P$ **and** $P \perp$ **and** $\bigwedge u. P\ u \implies P\ (\text{unfoldF} \cdot h \cdot u)$

shows $P\ (\text{unfold} \cdot h)$

unfolding *unfold-eq-fix* **by** (rule *fix-ind* [of P , $OF\ \text{assms}$])

fixrec

$\text{unfold2} :: ('s \rightarrow 'a \text{ LList}) \rightarrow ('a, 's) \text{ Step} \rightarrow 'a \text{ LList}$

where

$\text{unfold2} \cdot u \cdot \text{Done} = \text{LNil}$

| $s \neq \perp \implies \text{unfold2} \cdot u \cdot (\text{Skip} \cdot s) = u \cdot s$

| $s \neq \perp \implies \text{unfold2} \cdot u \cdot (\text{Yield} \cdot x \cdot s) = \text{LCons} \cdot x \cdot (u \cdot s)$

lemma *unfold2-strict* [*simp*]: $\text{unfold2} \cdot u \cdot \perp = \perp$

by *fixrec-simp*

lemma *unfold*: $s \neq \perp \implies \text{unfold} \cdot h \cdot s = \text{unfold2} \cdot (\text{unfold} \cdot h) \cdot (h \cdot s)$

by (*case-tac* $h \cdot s$, *simp-all*)

lemma *unfoldF*: $s \neq \perp \implies \text{unfoldF} \cdot h \cdot u \cdot s = \text{unfold2} \cdot u \cdot (h \cdot s)$

by (*case-tac* $h \cdot s$, *simp-all*)

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

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

declare *unfoldF* [*simp*]

fixrec

$\text{unstream} :: ('a, 's) \text{ Stream} \rightarrow 'a \text{ LList}$

where

$s \neq \perp \implies \text{unstream} \cdot (\text{Stream} \cdot h \cdot s) = \text{unfold} \cdot h \cdot s$

lemma *unstream-strict* [*simp*]: $\text{unstream} \cdot \perp = \perp$

by *fixrec-simp*

2.3 Converting from lists to streams

fixrec

$\text{streamStep} :: ('a \text{ LList})_{\perp} \rightarrow ('a, ('a \text{ LList})_{\perp}) \text{ Step}$

where

$\text{streamStep} \cdot (\text{up} \cdot \text{LNil}) = \text{Done}$

| $\text{streamStep} \cdot (\text{up} \cdot (\text{LCons} \cdot x \cdot xs)) = \text{Yield} \cdot x \cdot (\text{up} \cdot xs)$

lemma *streamStep-strict* [*simp*]: $\text{streamStep} \cdot (\text{up} \cdot \perp) = \perp$

by *fixrec-simp*

fixrec

$\text{stream} :: 'a \text{ LList} \rightarrow ('a, ('a \text{ LList})_{\perp}) \text{ Stream}$

where
 $stream \cdot xs = Stream \cdot streamStep \cdot (up \cdot xs)$
lemma *stream-defined* [*simp*]: $stream \cdot xs \neq \perp$
by *simp*
lemma *unstream-stream* [*simp*]:
fixes $xs :: 'a \text{ LList}$
shows $unstream \cdot (stream \cdot xs) = xs$
by (*induct xs, simp-all add: unfold*)
declare *stream.simps* [*simp del*]

2.4 Bisimilarity relation on streams

definition
 $bisimilar :: ('a, 's) Stream \Rightarrow ('a, 't) Stream \Rightarrow bool$ (**infix** $\langle \approx \rangle$ 50)
where
 $a \approx b \iff unstream \cdot a = unstream \cdot b \wedge a \neq \perp \wedge b \neq \perp$
lemma *unstream-cong*:
 $a \approx b \implies unstream \cdot a = unstream \cdot b$
unfolding *bisimilar-def* **by** *simp*
lemma *stream-cong*:
 $xs = ys \implies stream \cdot xs \approx stream \cdot ys$
unfolding *bisimilar-def* **by** *simp*
lemma *stream-unstream-cong*:
 $a \approx b \implies stream \cdot (unstream \cdot a) \approx b$
unfolding *bisimilar-def* **by** *simp*
end

3 Stream Fusion

theory *StreamFusion*
imports *Stream*
begin

3.1 Type constructors for state types

domain $Switch = S1 \mid S2$
domain $'a \text{ Maybe} = Nothing \mid Just \ 'a$
hide-const (**open**) *Left Right*
domain $('a, 'b) \text{ Either} = Left \ 'a \mid Right \ 'b$

domain ('a, 'b) Both (infixl <!!> 25) = Both 'a 'b (infixl <!!> 75)

domain 'a L = L (lazy 'a)

3.2 Map function

fixrec

$\text{mapStep} :: ('a \rightarrow 'b) \rightarrow ('s \rightarrow ('a, 's) \text{Step}) \rightarrow 's \rightarrow ('b, 's) \text{Step}$

where

$\text{mapStep} \cdot f \cdot h \cdot \perp = \perp$

| $s \neq \perp \implies \text{mapStep} \cdot f \cdot h \cdot s = (\text{case } h \cdot s \text{ of}$

$\text{Done} \Rightarrow \text{Done}$

| $\text{Skip} \cdot s' \Rightarrow \text{Skip} \cdot s'$

| $\text{Yield} \cdot x \cdot s' \Rightarrow \text{Yield} \cdot (f \cdot x) \cdot s'$)

fixrec

$\text{mapS} :: ('a \rightarrow 'b) \rightarrow ('a, 's) \text{Stream} \rightarrow ('b, 's) \text{Stream}$

where

$s \neq \perp \implies \text{mapS} \cdot f \cdot (\text{Stream} \cdot h \cdot s) = \text{Stream} \cdot (\text{mapStep} \cdot f \cdot h) \cdot s$

lemma *unfold-mapStep*:

fixes $f :: 'a \rightarrow 'b$ **and** $h :: 's \rightarrow ('a, 's) \text{Step}$

assumes $s \neq \perp$

shows $\text{unfold} \cdot (\text{mapStep} \cdot f \cdot h) \cdot s = \text{mapL} \cdot f \cdot (\text{unfold} \cdot h \cdot s)$

proof (*rule below-antisym*)

show $\text{unfold} \cdot (\text{mapStep} \cdot f \cdot h) \cdot s \sqsubseteq \text{mapL} \cdot f \cdot (\text{unfold} \cdot h \cdot s)$

using $\langle s \neq \perp \rangle$

apply (*induct arbitrary: s rule: unfold-ind [where h=mapStep.f.h]*)

apply (*simp, simp*)

apply (*case-tac h.s, simp-all add: unfold*)

done

next

show $\text{mapL} \cdot f \cdot (\text{unfold} \cdot h \cdot s) \sqsubseteq \text{unfold} \cdot (\text{mapStep} \cdot f \cdot h) \cdot s$

using $\langle s \neq \perp \rangle$

apply (*induct arbitrary: s rule: unfold-ind [where h=h]*)

apply (*simp, simp*)

apply (*case-tac h.s, simp-all add: unfold*)

done

qed

lemma *unstream-mapS*:

fixes $f :: 'a \rightarrow 'b$ **and** $a :: ('a, 's) \text{Stream}$

shows $a \neq \perp \implies \text{unstream} \cdot (\text{mapS} \cdot f \cdot a) = \text{mapL} \cdot f \cdot (\text{unstream} \cdot a)$

by (*induct a, simp, simp add: unfold-mapStep*)

lemma *mapS-defined*: $a \neq \perp \implies \text{mapS} \cdot f \cdot a \neq \perp$

by (*induct a, simp-all*)

```

lemma mapS-cong:
  fixes f :: 'a → 'b
  fixes a :: ('a, 's) Stream
  fixes b :: ('a, 't) Stream
  shows f = g ⇒ a ≈ b ⇒ mapS.f.a ≈ mapS.g.b
unfolding bisimilar-def
by (simp add: unstream-mapS mapS-defined)

lemma mapL-eq: mapL.f.xs = unstream.(mapS.f.(stream.xs))
by (simp add: unstream-mapS)

```

3.3 Filter function

```

fixrec
  filterStep :: ('a → tr) → ('s → ('a, 's) Step) → 's → ('a, 's) Step
where
  filterStep.p.h.⊥ = ⊥
| s ≠ ⊥ ⇒ filterStep.p.h.s = (case h.s of
  Done ⇒ Done
| Skip.s' ⇒ Skip.s'
| Yield.x.s' ⇒ (If p.x then Yield.x.s' else Skip.s'))

```

```

fixrec
  filterS :: ('a → tr) → ('a, 's) Stream → ('a, 's) Stream
where
  s ≠ ⊥ ⇒ filterS.p.(Stream.h.s) = Stream.(filterStep.p.h).s

```

```

lemma unfold-filterStep:
  fixes p :: 'a → tr and h :: 's → ('a, 's) Step
  assumes s ≠ ⊥
  shows unfold.(filterStep.p.h).s = filterL.p.(unfold.h.s)
proof (rule below-antisym)
  show unfold.(filterStep.p.h).s ⊆ filterL.p.(unfold.h.s)
  using ⟨s ≠ ⊥⟩
  apply (induct arbitrary: s rule: unfold-ind [where h=filterStep.p.h])
  apply (simp, simp)
  apply (case-tac h.s, simp-all add: unfold)
  apply (case-tac p.a rule: trE, simp-all)
  done
next
  show filterL.p.(unfold.h.s) ⊆ unfold.(filterStep.p.h).s
  using ⟨s ≠ ⊥⟩
  apply (induct arbitrary: s rule: unfold-ind [where h=h])
  apply (simp, simp)
  apply (case-tac h.s, simp-all add: unfold)
  apply (case-tac p.a rule: trE, simp-all add: unfold)
  done
qed

```

lemma *unstream-filterS*:
 $a \neq \perp \implies \text{unstream} \cdot (\text{filterS} \cdot p \cdot a) = \text{filterL} \cdot p \cdot (\text{unstream} \cdot a)$
by (*induct* *a*, *simp*, *simp* *add*: *unfold-filterStep*)

lemma *filterS-defined*: $a \neq \perp \implies \text{filterS} \cdot p \cdot a \neq \perp$
by (*induct* *a*, *simp*-*all*)

lemma *filterS-cong*:
fixes $p :: 'a \rightarrow tr$
fixes $a :: ('a, 's) \text{Stream}$
fixes $b :: ('a, 't) \text{Stream}$
shows $p = q \implies a \approx b \implies \text{filterS} \cdot p \cdot a \approx \text{filterS} \cdot q \cdot b$
unfolding *bisimilar-def*
by (*simp* *add*: *unstream-filterS filterS-defined*)

lemma *filterL-eq*: $\text{filterL} \cdot p \cdot xs = \text{unstream} \cdot (\text{filterS} \cdot p \cdot (\text{stream} \cdot xs))$
by (*simp* *add*: *unstream-filterS*)

3.4 Foldr function

fixrec
 $\text{foldrS} :: ('a \rightarrow 'b \rightarrow 'b) \rightarrow 'b \rightarrow ('a, 's) \text{Stream} \rightarrow 'b$
where
foldrS-Stream:
 $s \neq \perp \implies \text{foldrS} \cdot f \cdot z \cdot (\text{Stream} \cdot h \cdot s) =$
 $(\text{case } h \cdot s \text{ of } \text{Done} \Rightarrow z$
 $\quad | \text{Skip} \cdot s' \Rightarrow \text{foldrS} \cdot f \cdot z \cdot (\text{Stream} \cdot h \cdot s')$
 $\quad | \text{Yield} \cdot x \cdot s' \Rightarrow f \cdot x \cdot (\text{foldrS} \cdot f \cdot z \cdot (\text{Stream} \cdot h \cdot s')))$

lemma *unfold-foldrS*:
assumes $s \neq \perp$ **shows** $\text{foldrS} \cdot f \cdot z \cdot (\text{Stream} \cdot h \cdot s) = \text{foldrL} \cdot f \cdot z \cdot (\text{unfold} \cdot h \cdot s)$
proof (*rule* *below-antisym*)
show $\text{foldrS} \cdot f \cdot z \cdot (\text{Stream} \cdot h \cdot s) \sqsubseteq \text{foldrL} \cdot f \cdot z \cdot (\text{unfold} \cdot h \cdot s)$
using $\langle s \neq \perp \rangle$
apply (*induct* *arbitrary*: *s* *rule*: *foldrS.induct*)
apply (*simp*, *simp*, *simp*)
apply (*case-tac* *h*·*s*, *simp*-*all* *add*: *monofun-cfun* *unfold*)
done
next
show $\text{foldrL} \cdot f \cdot z \cdot (\text{unfold} \cdot h \cdot s) \sqsubseteq \text{foldrS} \cdot f \cdot z \cdot (\text{Stream} \cdot h \cdot s)$
using $\langle s \neq \perp \rangle$
apply (*induct* *arbitrary*: *s* *rule*: *unfold-ind*)
apply (*simp*, *simp*)
apply (*case-tac* *h*·*s*, *simp*-*all* *add*: *monofun-cfun* *unfold*)
done
qed

lemma *unstream-foldrS*:
 $a \neq \perp \implies \text{foldrS} \cdot f \cdot z \cdot a = \text{foldrL} \cdot f \cdot z \cdot (\text{unstream} \cdot a)$

by (*induct a, simp, simp del: foldrS-Stream add: unfold-foldrS*)

lemma *foldrS-cong*:

fixes $a :: ('a, 's) \text{Stream}$

fixes $b :: ('a, 't) \text{Stream}$

shows $f = g \implies z = w \implies a \approx b \implies \text{foldrS} \cdot f \cdot z \cdot a = \text{foldrS} \cdot g \cdot w \cdot b$

by (*simp add: bisimilar-def unstream-foldrS*)

lemma *foldrL-eq*:

$\text{foldrL} \cdot f \cdot z \cdot xs = \text{foldrS} \cdot f \cdot z \cdot (\text{stream} \cdot xs)$

by (*simp add: unstream-foldrS*)

3.5 EnumFromTo function

type-synonym $\text{int}' = \text{int}_{\perp}$

fixrec

$\text{enumFromToStep} :: \text{int}' \rightarrow (\text{int}')_{\perp} \rightarrow (\text{int}', (\text{int}')_{\perp}) \text{Step}$

where

$\text{enumFromToStep} \cdot (\text{up} \cdot y) \cdot (\text{up} \cdot (\text{up} \cdot x)) =$

$(\text{if } x \leq y \text{ then } \text{Yield} \cdot (\text{up} \cdot x) \cdot (\text{up} \cdot (\text{up} \cdot (x+1))) \text{ else } \text{Done})$

lemma *enumFromToStep-strict* [*simp*]:

$\text{enumFromToStep} \cdot \perp \cdot x'' = \perp$

$\text{enumFromToStep} \cdot (\text{up} \cdot y) \cdot \perp = \perp$

$\text{enumFromToStep} \cdot (\text{up} \cdot y) \cdot (\text{up} \cdot \perp) = \perp$

by *fixrec-simp+*

lemma *enumFromToStep-simps'* [*simp*]:

$x \leq y \implies \text{enumFromToStep} \cdot (\text{up} \cdot y) \cdot (\text{up} \cdot (\text{up} \cdot x)) =$

$\text{Yield} \cdot (\text{up} \cdot x) \cdot (\text{up} \cdot (\text{up} \cdot (x+1)))$

$\neg x \leq y \implies \text{enumFromToStep} \cdot (\text{up} \cdot y) \cdot (\text{up} \cdot (\text{up} \cdot x)) = \text{Done}$

by *simp-all*

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

fixrec

$\text{enumFromToS} :: \text{int}' \rightarrow \text{int}' \rightarrow (\text{int}', (\text{int}')_{\perp}) \text{Stream}$

where

$\text{enumFromToS} \cdot x \cdot y = \text{Stream} \cdot (\text{enumFromToStep} \cdot y) \cdot (\text{up} \cdot x)$

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

lemma *unfold-enumFromToStep*:

$\text{unfold} \cdot (\text{enumFromToStep} \cdot (\text{up} \cdot y)) \cdot (\text{up} \cdot n) = \text{enumFromToL} \cdot n \cdot (\text{up} \cdot y)$

proof (*rule below-antisym*)

show $\text{unfold} \cdot (\text{enumFromToStep} \cdot (\text{up} \cdot y)) \cdot (\text{up} \cdot n) \sqsubseteq \text{enumFromToL} \cdot n \cdot (\text{up} \cdot y)$

apply (*induct arbitrary: n rule: unfold-ind* [**where** $h = \text{enumFromToStep} \cdot (\text{up} \cdot y)$])

apply (*simp, simp*)

```

    apply (case-tac n, simp, simp)
    apply (case-tac  $x \leq y$ , simp-all)
    done
next
show  $\text{enumFromToL} \cdot n \cdot (\text{up} \cdot y) \sqsubseteq \text{unfold} \cdot (\text{enumFromToStep} \cdot (\text{up} \cdot y)) \cdot (\text{up} \cdot n)$ 
  apply (induct arbitrary: n rule:  $\text{enumFromToL.induct}$ )
  apply (simp, simp)
  apply (rename-tac e n)
  apply (case-tac n, simp)
  apply (case-tac  $x \leq y$ , simp-all add: unfold)
  done
qed

```

```

lemma  $\text{unstream-enumFromToS}$ :
   $\text{unstream} \cdot (\text{enumFromToS} \cdot x \cdot y) = \text{enumFromToL} \cdot x \cdot y$ 
  apply (simp add:  $\text{enumFromToS.simps}$ )
  apply (induct y, simp add: unfold)
  apply (induct x, simp add: unfold)
  apply (simp add:  $\text{unfold-enumFromToStep}$ )
  done

```

```

lemma  $\text{enumFromToS-defined}$ :  $\text{enumFromToS} \cdot x \cdot y \neq \perp$ 
  by (simp add:  $\text{enumFromToS.simps}$ )

```

```

lemma  $\text{enumFromToS-cong}$ :
   $x = x' \implies y = y' \implies \text{enumFromToS} \cdot x \cdot y \approx \text{enumFromToS} \cdot x' \cdot y'$ 
  unfolding  $\text{bisimilar-def}$  by (simp add:  $\text{enumFromToS-defined}$ )

```

```

lemma  $\text{enumFromToL-eq}$ :  $\text{enumFromToL} \cdot x \cdot y = \text{unstream} \cdot (\text{enumFromToS} \cdot x \cdot y)$ 
  by (simp add:  $\text{unstream-enumFromToS}$ )

```

3.6 Append function

```

fixrec
  appendStep ::
    ( $'s \rightarrow ('a, 's) \text{Step}$ )  $\rightarrow$ 
    ( $'t \rightarrow ('a, 't) \text{Step}$ )  $\rightarrow$ 
     $'t \rightarrow ('s, 't) \text{Either} \rightarrow ('a, ('s, 't) \text{Either}) \text{Step}$ 
  where
     $sa \neq \perp \implies \text{appendStep} \cdot ha \cdot hb \cdot sb0 \cdot (\text{Left} \cdot sa) =$ 
      (case  $ha \cdot sa$  of
        Done  $\Rightarrow \text{Skip} \cdot (\text{Right} \cdot sb0)$ 
      | Skip  $\cdot sa' \Rightarrow \text{Skip} \cdot (\text{Left} \cdot sa')$ 
      | Yield  $\cdot x \cdot sa' \Rightarrow \text{Yield} \cdot x \cdot (\text{Left} \cdot sa')$ )
    |  $sb \neq \perp \implies \text{appendStep} \cdot ha \cdot hb \cdot sb0 \cdot (\text{Right} \cdot sb) =$ 
      (case  $hb \cdot sb$  of
        Done  $\Rightarrow \text{Done}$ 
      | Skip  $\cdot sb' \Rightarrow \text{Skip} \cdot (\text{Right} \cdot sb')$ 
      | Yield  $\cdot x \cdot sb' \Rightarrow \text{Yield} \cdot x \cdot (\text{Right} \cdot sb')$ )

```

lemma *appendStep-strict* [*simp*]: *appendStep*·*ha*·*hb*·*sb0*· $\perp$ = $\perp$
by *fixrec-simp*

fixrec

appendS ::
 ('a, 's) *Stream* $\rightarrow$ ('a, 't) *Stream* $\rightarrow$ ('a, ('s, 't) *Either*) *Stream*

where

sa0 $\neq \perp \implies$ *sb0* $\neq \perp \implies$
appendS·(*Stream*·*ha*·*sa0*)·(*Stream*·*hb*·*sb0*) =
Stream·(*appendStep*·*ha*·*hb*·*sb0*)·(*Left*·*sa0*)

lemma *unfold-appendStep*:

fixes *ha* :: 's $\rightarrow$ ('a, 's) *Step*
fixes *hb* :: 't $\rightarrow$ ('a, 't) *Step*
assumes *sb0* [*simp*]: *sb0* $\neq \perp$

shows

($\forall$ *sa*. *sa* $\neq \perp \longrightarrow$ *unfold*·(*appendStep*·*ha*·*hb*·*sb0*)·(*Left*·*sa*) =
appendL·(*unfold*·*ha*·*sa*)·(*unfold*·*hb*·*sb0*)) $\wedge$
($\forall$ *sb*. *sb* $\neq \perp \longrightarrow$ *unfold*·(*appendStep*·*ha*·*hb*·*sb0*)·(*Right*·*sb*) =
unfold·*hb*·*sb*)

proof –

note *unfold* [*simp*]
let ?*h* = *appendStep*·*ha*·*hb*·*sb0*

have 1:

($\forall$ *sa*. *sa* $\neq \perp \longrightarrow$
unfold·?*h*·(*Left*·*sa*) $\sqsubseteq$
appendL·(*unfold*·*ha*·*sa*)·(*unfold*·*hb*·*sb0*))
 $\wedge$
($\forall$ *sb*. *sb* $\neq \perp \longrightarrow$ *unfold*·?*h*·(*Right*·*sb*) $\sqsubseteq$ *unfold*·*hb*·*sb*)
apply (rule *unfold-ind* [**where** *h*=?*h*])
apply *simp*
apply *simp*
apply (*intro conjI allI impI*)
apply (*case-tac ha*·*sa*, *simp*, *simp*, *simp*, *simp*)
apply (*case-tac hb*·*sb*, *simp*, *simp*, *simp*, *simp*)
done

let ?*P* = $\lambda ua ub. \forall sa. sa \neq \perp \longrightarrow$
appendL·(*ua*·*sa*)·(*ub*·*sb0*) $\sqsubseteq$ *unfold*·?*h*·(*Left*·*sa*)

let ?*Q* = $\lambda ub. \forall sb. sb \neq \perp \longrightarrow ub \cdot sb \sqsubseteq$ *unfold*·?*h*·(*Right*·*sb*)

have *P-base*: $\bigwedge ub. ?P \perp ub$
by *simp*

have *Q-base*: ?*Q* $\perp$
by *simp*

```

have P-step:  $\bigwedge ua\ ub. ?P\ ua\ ub \implies ?Q\ ub \implies ?P\ (unfoldF \cdot ha \cdot ua)\ ub$ 
  apply (intro allI impI)
  apply (case-tac ha·sa, simp, simp, simp, simp)
done

have Q-step:  $\bigwedge ua\ ub. ?Q\ ub \implies ?Q\ (unfoldF \cdot hb \cdot ub)$ 
  apply (intro allI impI)
  apply (case-tac hb·sb, simp, simp, simp, simp)
done

have Q:  $?Q\ (unfold \cdot hb)$ 
  apply (rule unfold-ind [where h=hb], simp)
  apply (rule Q-base)
  apply (erule Q-step)
done

have P:  $?P\ (unfold \cdot ha)\ (unfold \cdot hb)$ 
  apply (rule unfold-ind [where h=ha], simp)
  apply (rule P-base)
  apply (erule P-step)
  apply (rule Q)
done

have 2:  $?P\ (unfold \cdot ha)\ (unfold \cdot hb) \wedge ?Q\ (unfold \cdot hb)$ 
  using P Q by (rule conjI)

from 1 2 show ?thesis
  by (simp add: po-eq-conv [where 'a='a LList])
qed

lemma appendS-defined:  $xs \neq \perp \implies ys \neq \perp \implies appendS \cdot xs \cdot ys \neq \perp$ 
by (cases xs, simp, cases ys, simp, simp)

lemma unstream-appendS:
   $a \neq \perp \implies b \neq \perp \implies$ 
   $unstream \cdot (appendS \cdot a \cdot b) = appendL \cdot (unstream \cdot a) \cdot (unstream \cdot b)$ 
  apply (cases a, simp, cases b, simp)
  apply (simp add: unfold-appendStep)
done

lemma appendS-cong:
  fixes f :: 'a  $\rightarrow$  'b
  fixes a :: ('a, 's) Stream
  fixes b :: ('a, 't) Stream
  shows  $a \approx a' \implies b \approx b' \implies appendS \cdot a \cdot b \approx appendS \cdot a' \cdot b'$ 
  unfolding bisimilar-def
  by (simp add: unstream-appendS appendS-defined)

```

lemma *appendL-eq*: $\text{appendL} \cdot xs \cdot ys = \text{unstream} \cdot (\text{appendS} \cdot (\text{stream} \cdot xs) \cdot (\text{stream} \cdot ys))$
by (*simp add: unstream-appendS*)

3.7 ZipWith function

fixrec

zipWithStep ::
 $(a \rightarrow b \rightarrow c) \rightarrow$
 $(s \rightarrow (a, s) \text{ Step}) \rightarrow$
 $(t \rightarrow (b, t) \text{ Step}) \rightarrow$
 $s :: t :: a \text{ L Maybe} \rightarrow (c, s :: t :: a \text{ L Maybe}) \text{ Step}$

where

$sa \neq \perp \implies sb \neq \perp \implies$
 $\text{zipWithStep} \cdot f \cdot ha \cdot hb \cdot (sa :: sb :: \text{Nothing}) =$
 $(\text{case } ha \cdot sa \text{ of}$
 $\quad \text{Done} \Rightarrow \text{Done}$
 $\quad | \text{Skip} \cdot sa' \Rightarrow \text{Skip} \cdot (sa' :: sb :: \text{Nothing})$
 $\quad | \text{Yield} \cdot a \cdot sa' \Rightarrow \text{Skip} \cdot (sa' :: sb :: \text{Just} \cdot (L \cdot a)))$
 $| sa \neq \perp \implies sb \neq \perp \implies$
 $\text{zipWithStep} \cdot f \cdot ha \cdot hb \cdot (sa :: sb :: \text{Just} \cdot (L \cdot a)) =$
 $(\text{case } hb \cdot sb \text{ of}$
 $\quad \text{Done} \Rightarrow \text{Done}$
 $\quad | \text{Skip} \cdot sb' \Rightarrow \text{Skip} \cdot (sa :: sb' :: \text{Just} \cdot (L \cdot a))$
 $\quad | \text{Yield} \cdot b \cdot sb' \Rightarrow \text{Yield} \cdot (f \cdot a \cdot b) \cdot (sa :: sb' :: \text{Nothing}))$

lemma *zipWithStep-strict* [*simp*]: $\text{zipWithStep} \cdot f \cdot ha \cdot hb \cdot \perp = \perp$
by *fixrec-simp*

fixrec

zipWithS :: $(a \rightarrow b \rightarrow c) \rightarrow$
 $(a, s) \text{ Stream} \rightarrow (b, t) \text{ Stream} \rightarrow (c, s :: t :: a \text{ L Maybe}) \text{ Stream}$

where

$sa0 \neq \perp \implies sb0 \neq \perp \implies \text{zipWithS} \cdot f \cdot (\text{Stream} \cdot ha \cdot sa0) \cdot (\text{Stream} \cdot hb \cdot sb0) =$
 $\text{Stream} \cdot (\text{zipWithStep} \cdot f \cdot ha \cdot hb) \cdot (sa0 :: sb0 :: \text{Nothing})$

lemma *zipWithS-fix-ind-lemma*:

fixes $P \ Q :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{bool}$

assumes $P-0$: $\bigwedge j. P \ 0 \ j$ **and** $P\text{-Suc}$: $\bigwedge i \ j. P \ i \ j \implies Q \ i \ j \implies P \ (\text{Suc } i) \ j$

assumes $Q-0$: $\bigwedge i. Q \ i \ 0$ **and** $Q\text{-Suc}$: $\bigwedge i \ j. P \ i \ j \implies Q \ i \ j \implies Q \ i \ (\text{Suc } j)$

shows $P \ i \ j \wedge Q \ i \ j$

apply (*induct* $n \equiv i + j$ *arbitrary*: $i \ j$)

apply (*simp add*: $P-0 \ Q-0$)

apply (*rule conjI*)

apply (*case-tac* i , *simp add*: $P-0$, *simp add*: $P\text{-Suc}$)

apply (*case-tac* j , *simp add*: $Q-0$, *simp add*: $Q\text{-Suc}$)

done

lemma *zipWithS-fix-ind*:

assumes x : $x = \text{fix} \cdot f$ **and** y : $y = \text{fix} \cdot g$

```

assumes adm-P: adm ( $\lambda x. P \text{ (fst } x) \text{ (snd } x)$ )
assumes adm-Q: adm ( $\lambda x. Q \text{ (fst } x) \text{ (snd } x)$ )
assumes P-0:  $\bigwedge b. P \perp b$  and P-Suc:  $\bigwedge a b. P a b \implies Q a b \implies P (f \cdot a) b$ 
assumes Q-0:  $\bigwedge a. Q a \perp$  and Q-Suc:  $\bigwedge a b. P a b \implies Q a b \implies Q a (g \cdot b)$ 
shows  $P x y \wedge Q x y$ 
proof –
  have 1:  $\bigwedge i j. P \text{ (iterate } i \cdot f \cdot \perp) \text{ (iterate } j \cdot g \cdot \perp) \wedge Q \text{ (iterate } i \cdot f \cdot \perp) \text{ (iterate } j \cdot g \cdot \perp)$ 
    apply (rule-tac i=i and j=j in zipWithS-fix-ind-lemma)
    apply (simp add: P-0)
    apply (simp add: P-Suc)
    apply (simp add: Q-0)
    apply (simp add: Q-Suc)
    done
  have case-prod P ( $\bigsqcup i. \text{ (iterate } i \cdot f \cdot \perp, \text{ iterate } i \cdot g \cdot \perp)$ )
    apply (rule admD)
    apply (simp add: split-def adm-P)
    apply simp
    apply (simp add: 1)
    done
  then have P:  $P x y$ 
    unfolding  $x y$  fix-def2
    by (simp add: lub-prod)
  have case-prod Q ( $\bigsqcup i. \text{ (iterate } i \cdot f \cdot \perp, \text{ iterate } i \cdot g \cdot \perp)$ )
    apply (rule admD)
    apply (simp add: split-def adm-Q)
    apply simp
    apply (simp add: 1)
    done
  then have Q:  $Q x y$ 
    unfolding  $x y$  fix-def2
    by (simp add: lub-prod)
  from P Q show ?thesis by simp
qed

```

```

lemma unfold-zipWithStep:
  fixes  $f :: 'a \rightarrow 'b \rightarrow 'c$ 
  fixes  $ha :: 's \rightarrow ('a, 's) \text{ Step}$ 
  fixes  $hb :: 't \rightarrow ('b, 't) \text{ Step}$ 
  defines  $h\text{-def}: h \equiv \text{zipWithStep} \cdot f \cdot ha \cdot hb$ 
  shows
    ( $\forall sa sb. sa \neq \perp \longrightarrow sb \neq \perp \longrightarrow$ 
       $\text{unfold} \cdot h \cdot (sa \text{ !: } sb \text{ !: } \text{Nothing}) =$ 
       $\text{zipWithL} \cdot f \cdot (\text{unfold} \cdot ha \cdot sa) \cdot (\text{unfold} \cdot hb \cdot sb)) \wedge$ 
    ( $\forall sa sb a. sa \neq \perp \longrightarrow sb \neq \perp \longrightarrow$ 
       $\text{unfold} \cdot h \cdot (sa \text{ !: } sb \text{ !: } \text{Just} \cdot (L \cdot a)) =$ 
       $\text{zipWithL} \cdot f \cdot (L \text{Cons} \cdot a \cdot (\text{unfold} \cdot ha \cdot sa)) \cdot (\text{unfold} \cdot hb \cdot sb))$ )
proof –
  note unfold [simp]
  have h-simps [simp]:

```

$\bigwedge sa\ sb. sa \neq \perp \implies sb \neq \perp \implies h.(sa\ !:\ sb\ !:\ Nothing) =$
 $(case\ ha\cdot sa\ of$
 $\quad Done \Rightarrow Done$
 $\quad | Skip\cdot sa' \Rightarrow Skip.(sa' !:\ sb !:\ Nothing)$
 $\quad | Yield\cdot a\cdot sa' \Rightarrow Skip.(sa' !:\ sb !:\ Just.(L\cdot a)))$
 $\bigwedge sa\ sb\ a. sa \neq \perp \implies sb \neq \perp \implies h.(sa\ !:\ sb\ !:\ Just.(L\cdot a)) =$
 $(case\ hb\cdot sb\ of$
 $\quad Done \Rightarrow Done$
 $\quad | Skip\cdot sb' \Rightarrow Skip.(sa !:\ sb' !:\ Just.(L\cdot a))$
 $\quad | Yield\cdot b\cdot sb' \Rightarrow Yield.(f\cdot a\cdot b).(sa !:\ sb' !:\ Nothing))$
 $h.\perp = \perp$
unfolding h -def by *simp-all*

have 1:

$(\forall sa\ sb. sa \neq \perp \longrightarrow sb \neq \perp \longrightarrow$
 $\quad unfold\cdot h.(sa\ !:\ sb\ !:\ Nothing) \sqsubseteq$
 $\quad zipWithL\cdot f.(unfold\cdot ha\cdot sa).(unfold\cdot hb\cdot sb))$
 $\wedge$
 $(\forall sa\ sb\ a. sa \neq \perp \longrightarrow sb \neq \perp \longrightarrow$
 $\quad unfold\cdot h.(sa\ !:\ sb\ !:\ Just.(L\cdot a)) \sqsubseteq$
 $\quad zipWithL\cdot f.(LCons\cdot a.(unfold\cdot ha\cdot sa).(unfold\cdot hb\cdot sb))$
apply (rule *unfold-ind* [where $h=h$], *simp*)
apply *simp*
apply (intro *conjI* *allI* *impI*)
apply (case-tac $ha\cdot sa$, *simp*, *simp*, *simp*, *simp*)
apply (case-tac $hb\cdot sb$, *simp*, *simp*, *simp*, *simp*)
done

let $?P = \lambda ua\ ub. \forall sa\ sb. sa \neq \perp \longrightarrow sb \neq \perp \longrightarrow$
 $\quad zipWithL\cdot f.(ua\cdot sa).(ub\cdot sb) \sqsubseteq unfold\cdot h.(sa\ !:\ sb\ !:\ Nothing)$

let $?Q = \lambda ua\ ub. \forall sa\ sb\ a. sa \neq \perp \longrightarrow sb \neq \perp \longrightarrow$
 $\quad zipWithL\cdot f.(LCons\cdot a.(ua\cdot sa).(ub\cdot sb) \sqsubseteq$
 $\quad unfold\cdot h.(sa\ !:\ sb\ !:\ Just.(L\cdot a))$

have P -base: $\bigwedge ub. ?P\ \perp\ ub$
by *simp*

have Q -base: $\bigwedge ua. ?Q\ ua\ \perp$
by *simp*

have P -step: $\bigwedge ua\ ub. ?P\ ua\ ub \implies ?Q\ ua\ ub \implies ?P\ (unfoldF\cdot ha\cdot ua)\ ub$
by (*clarsimp*, case-tac $ha\cdot sa$, *simp-all*)

have Q -step: $\bigwedge ua\ ub. ?P\ ua\ ub \implies ?Q\ ua\ ub \implies ?Q\ ua\ (unfoldF\cdot hb\cdot ub)$
by (*clarsimp*, case-tac $hb\cdot sb$, *simp-all*)

have 2: $?P\ (unfold\cdot ha)\ (unfold\cdot hb) \wedge ?Q\ (unfold\cdot ha)\ (unfold\cdot hb)$
apply (rule *zipWithS-fix-ind* [OF *unfold-eq-fix* [of ha] *unfold-eq-fix* [of hb]])

```

    apply (simp, simp)
    apply (rule P-base)
    apply (erule (1) P-step)
    apply (rule Q-base)
    apply (erule (1) Q-step)
  done

  from 1 2 show ?thesis
  by (simp-all add: po-eq-conv [where 'a='c LList])
qed

lemma zipWithS-defined:  $a \neq \perp \implies b \neq \perp \implies \text{zipWithS}.f.a.b \neq \perp$ 
by (cases a, simp, cases b, simp, simp)

lemma unstream-zipWithS:
   $a \neq \perp \implies b \neq \perp \implies$ 
   $\text{unstream}.\text{zipWithS}.f.a.b = \text{zipWithL}.f.(\text{unstream}.a).(\text{unstream}.b)$ 
  apply (cases a, simp, cases b, simp)
  apply (simp add: unfold-zipWithStep)
  done

lemma zipWithS-cong:
   $f = f' \implies a \approx a' \implies b \approx b' \implies$ 
   $\text{zipWithS}.f.a.b \approx \text{zipWithS}.f'.a'.b'$ 
  unfolding bisimilar-def
  by (simp add: unstream-zipWithS zipWithS-defined)

lemma zipWithL-eq:
   $\text{zipWithL}.f.xs.ys = \text{unstream}.\text{zipWithS}.f.(\text{stream}.xs).(\text{stream}.ys)$ 
  by (simp add: unstream-zipWithS)

```

3.8 ConcatMap function

```

fixrec
  concatMapStep ::
    ( $'a \rightarrow ('b, 't) \text{Stream}$ )  $\rightarrow$ 
    ( $'s \rightarrow ('a, 's) \text{Step}$ )  $\rightarrow$ 
    ( $'s :: ('b, 't) \text{Stream Maybe} \rightarrow$ 
     ( $'b, 's :: ('b, 't) \text{Stream Maybe}$ )  $\text{Step}$ )
  where
     $sa \neq \perp \implies \text{concatMapStep}.f.ha.(sa :: \text{Nothing}) =$ 
      (case  $ha.sa$  of
        Done  $\Rightarrow$  Done
      | Skip. $sa'$   $\Rightarrow$  Skip. $(sa' :: \text{Nothing})$ 
      | Yield. $a.sa'$   $\Rightarrow$  Skip. $(sa' :: \text{Just}(f.a))$ )
    |  $sa \neq \perp \implies sb \neq \perp \implies$ 
       $\text{concatMapStep}.f.ha.(sa :: \text{Just}(\text{Stream}.hb.sb)) =$ 
      (case  $hb.sb$  of
        Done  $\Rightarrow$  Skip. $(sa :: \text{Nothing})$ 

```

| $Skip \cdot sb' \Rightarrow Skip \cdot (sa \text{ !: } Just \cdot (Stream \cdot hb \cdot sb'))$
| $Yield \cdot b \cdot sb' \Rightarrow Yield \cdot b \cdot (sa \text{ !: } Just \cdot (Stream \cdot hb \cdot sb'))$

lemma *concatMapStep-strict* [simp]: *concatMapStep*·*f*·*ha*· $\perp = \perp$
by *fixrec-simp*

fixrec

concatMapS ::
 $('a \rightarrow ('b, 't) \text{ Stream}) \rightarrow ('a, 's) \text{ Stream} \rightarrow$
 $('b, 's \text{ !: } ('b, 't) \text{ Stream Maybe}) \text{ Stream}$

where

$s \neq \perp \implies \text{concatMapS} \cdot f \cdot (Stream \cdot h \cdot s) = Stream \cdot (\text{concatMapStep} \cdot f \cdot h) \cdot (s \text{ !: } Nothing)$

lemma *concatMapS-strict* [simp]: *concatMapS*·*f*· $\perp = \perp$
by *fixrec-simp*

lemma *unfold-concatMapStep*:

fixes *ha* :: $'s \rightarrow ('a, 's) \text{ Step}$
fixes *f* :: $'a \rightarrow ('b, 't) \text{ Stream}$
defines *h-def*: $h \equiv \text{concatMapStep} \cdot f \cdot ha$
defines *f'-def*: $f' \equiv \text{unstream} \circ f$

shows

$(\forall sa. sa \neq \perp \longrightarrow$
 $\text{unfold} \cdot h \cdot (sa \text{ !: } Nothing) = \text{concatMapL} \cdot f' \cdot (\text{unfold} \cdot ha \cdot sa)) \wedge$
 $(\forall sa \ hb \ sb. sa \neq \perp \longrightarrow sb \neq \perp \longrightarrow$
 $\text{unfold} \cdot h \cdot (sa \text{ !: } Just \cdot (Stream \cdot hb \cdot sb)) =$
 $\text{appendL} \cdot (\text{unfold} \cdot hb \cdot sb) \cdot (\text{concatMapL} \cdot f' \cdot (\text{unfold} \cdot ha \cdot sa)))$

proof –

note *unfold* [simp]

have *h-simps* [simp]:

$\bigwedge sa. sa \neq \perp \implies h \cdot (sa \text{ !: } Nothing) =$
 $(\text{case } ha \cdot sa \text{ of } Done \Rightarrow Done$
| $Skip \cdot sa' \Rightarrow Skip \cdot (sa' \text{ !: } Nothing)$
| $Yield \cdot a \cdot sa' \Rightarrow Skip \cdot (sa' \text{ !: } Just \cdot (f \cdot a)))$
 $\bigwedge sa \ hb \ sb. sa \neq \perp \implies sb \neq \perp \implies h \cdot (sa \text{ !: } Just \cdot (Stream \cdot hb \cdot sb)) =$
 $(\text{case } hb \cdot sb \text{ of } Done \Rightarrow Skip \cdot (sa \text{ !: } Nothing)$
| $Skip \cdot sb' \Rightarrow Skip \cdot (sa \text{ !: } Just \cdot (Stream \cdot hb \cdot sb'))$
| $Yield \cdot b \cdot sb' \Rightarrow Yield \cdot b \cdot (sa \text{ !: } Just \cdot (Stream \cdot hb \cdot sb')))$
 $h \cdot \perp = \perp$

unfolding *h-def* **by** *simp-all*

have *f'-beta* [simp]: $\bigwedge a. f' \cdot a = \text{unstream} \cdot (f \cdot a)$

unfolding *f'-def* **by** *simp*

have *1*:

$(\forall sa. sa \neq \perp \longrightarrow$
 $\text{unfold} \cdot h \cdot (sa \text{ !: } Nothing) \sqsubseteq \text{concatMapL} \cdot f' \cdot (\text{unfold} \cdot ha \cdot sa))$
 $\wedge$

```

(∀ sa hb sb. sa ≠ ⊥ ⟶ sb ≠ ⊥ ⟶
  unfold·h·(sa !: Just·(Stream·hb·sb)) ⊆
  appendL·(unfold·hb·sb)·(concatMapL·f'·(unfold·ha·sa)))
apply (rule unfold-ind [where h=h], simp)
apply simp
apply (intro conjI allI impI)
apply (case-tac ha·sa, simp, simp, simp)
apply (rename-tac a sa')
apply (case-tac f·a, simp, simp)
apply (case-tac hb·sb, simp, simp, simp, simp)
done

let ?P = λua. ∀ sa. sa ≠ ⊥ ⟶
  concatMapL·f'·(ua·sa) ⊆ unfold·h·(sa !: Nothing)

let ?Q = λhb ua ub. ∀ sa sb. sa ≠ ⊥ ⟶ sb ≠ ⊥ ⟶
  appendL·(ub·sb)·(concatMapL·f'·(ua·sa)) ⊆
  unfold·h·(sa !: Just·(Stream·hb·sb))

have P-base: ?P ⊥
  by simp

have P-step: ∧ua. ?P ua ⟹ ∀ hb. ?Q hb ua (unfold·hb) ⟹ ?P (unfoldF·ha·ua)
  apply (intro allI impI)
  apply (case-tac ha·sa, simp, simp, simp)
  apply (rename-tac a sa')
  apply (case-tac f·a, simp, simp)
  done

have Q-base: ∧ua hb. ?Q hb ua ⊥
  by simp

have Q-step: ∧hb ua ub. ?P ua ⟹ ?Q hb ua ub ⟹ ?Q hb ua (unfoldF·hb·ub)
  apply (intro allI impI)
  apply (case-tac hb·sb, simp, simp, simp, simp)
  done

have 2: ?P (unfold·ha) ∧ (∀ hb. ?Q hb (unfold·ha) (unfold·hb))
  apply (rule unfold-ind [where h=ha], simp)
  apply (rule conjI)
  apply (rule P-base)
  apply (rule allI, rule-tac h=hb in unfold-ind, simp)
  apply (rule Q-base)
  apply (erule Q-step [OF P-base])
  apply (erule conjE)
  apply (rule conjI)
  apply (erule (1) P-step)
  apply (rule allI, rule-tac h=hb in unfold-ind, simp)
  apply (rule Q-base)

```

```

    apply (erule (2) Q-step [OF P-step])
  done

from 1 2 show ?thesis
  by (simp-all add: po-eq-conv [where 'a='b LList])
qed

lemma unstream-concatMapS:
  unstream.(concatMapS.f.a) = concatMapL.(unstream oo f).(unstream.a)
by (cases a, simp, simp add: unfold-concatMapStep)

lemma concatMapS-defined: a ≠ ⊥ ⇒ concatMapS.f.a ≠ ⊥
by (induct a, simp-all)

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 (⋀x. f x ≈ g x) ⇒ a ≈ b ⇒ cont f ⇒ cont g ⇒
    concatMapS.(⋀ x. f x).a ≈ concatMapS.(⋀ x. g x).b
unfolding bisimilar-def
by (simp add: unstream-concatMapS oo-def concatMapS-defined)

lemma concatMapL-eq:
  concatMapL.f.xs = unstream.(concatMapS.(stream oo f).(stream.xs))
by (simp add: unstream-concatMapS oo-def eta-cfun)

```

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.f oo filterL.p oo mapL.g =
unstream oo mapS.f oo filterS.p oo mapS.g oo stream
apply (*rule cfun-eqI, simp*)
apply (*unfold stream-eqs*)
apply (*intro stream-congs refl*)
done

lemma

foldrL.f.z.(mapL.g.(filterL.p.(enumFromToL.x.y))) =
foldrS.f.z.(mapS.g.(filterS.p.(enumFromToS.x.y)))
apply (*unfold stream-eqs*)
apply (*intro stream-congs refl*)
done

lemma *oo-LAM [simp]: cont g $\implies$ f oo ($\Lambda x. g x$) = ($\Lambda x. f.(g x)$)*
unfolding *oo-def by simp*

lemma

concatMapL.($\Lambda k.$
*mapL.($\Lambda m. f.k.m$).(*enumFromToL.one.k*)).(*enumFromToL.one.n*) =*
unstream.(concatMapS.($\Lambda k.$
*mapS.($\Lambda m. f.k.m$).(*enumFromToS.one.k*)).(*enumFromToS.one.n*))*
unfolding *stream-eqs*
apply *simp*
apply (*simp add: stream-congs*)
done

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.