

Stream Fusion

Brian Huffman

December 12, 2009

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	4
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	5
3	Stream Fusion	6
3.1	Type constructors for state types	6
3.2	Map function	6
3.3	Filter function	7
3.4	Foldr function	8
3.5	EnumFromTo function	9
3.6	Append function	11
3.7	ZipWith function	13
3.8	ConcatMap function	17
3.9	Examples	19

1 Lazy Lists

```
theory LazyList
imports HOLCF
begin

  Discrete cpo instance for int.

instantiation int :: discrete-cpo
begin

definition below-int-def:
  (x::int)  $\sqsubseteq$  y  $\longleftrightarrow$  x = y

instance proof
qed (rule below-int-def)

end

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

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

fixpat mapL-strict [simp]: mapL.f. $\perp$ 

fixrec
  filterL :: ('a  $\rightarrow$  tr)  $\rightarrow$  'a LList  $\rightarrow$  '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 fi)

fixpat filterL-strict [simp]: filterL.p. $\perp$ 

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

fixpat foldrL-strict [simp]: foldrL.f.z. $\perp$ 

fixrec
  enumFromToL :: int $\perp$   $\rightarrow$  int $\perp$   $\rightarrow$  (int $\perp$ ) LList
where
  enumFromToL.(up.x).(up.y) =
    (if x  $\leq$  y then LCons.(up.x).(enumFromToL.(up.(x+1)).(up.y)) else LNil)
```

```

lemma enumFromToL-simps' [simp]:
   $x \leq y \implies$ 
    enumFromToL.(up.x).(up.y) = LCons.(up.x).(enumFromToL.(up.(x+1)).(up.y))
   $\neg x \leq y \implies$  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.y = y
  | appendL.(LCons.x.xs).y = LCons.x.(appendL.xs.y)

fixpat appendL-strict [simp]: appendL. $\perp$ .y

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.y = LNil
  | zipWithL.f.(LCons.x.xs).LNil = LNil
  | zipWithL.f.(LCons.x.xs).(LCons.y.y) = LCons.(f.x.y).(zipWithL.f.xs.y)

fixpat zipWithL-strict [simp]:
  zipWithL.f. $\perp$ .y
  zipWithL.f.(LCons.x.xs). $\perp$ 

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)

fixpat concatMapL-strict [simp]: concatMapL.f. $\perp$ 

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
```

```
domain ('a,'s) Stream = Stream (lazy 's → ('a,'s) Step) 's
```

2.2 Converting from streams to lists

```
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)
```

```
fixpat unfold2-strict [simp]: unfold2·u·⊥
```

```
definition
```

```
  unfold1 :: ('s → ('a, 's) Step) → ('s → 'a LList) → ('s → 'a LList)
```

```
where
```

```
  unfold1 = (λ h u. strictify·(λ s. unfold2·u·(h·s)))
```

```
lemma unfold1-simps [simp]:
```

```
  unfold1·h·u·⊥ = ⊥
```

```
  s ≠ ⊥ ⇒⇒ unfold1·h·u·s = unfold2·u·(h·s)
```

```
unfolding unfold1-def by simp-all
```

```
definition
```

```
  unfold :: ('s → ('a, 's) Step) → 's → 'a LList
```

```
where
```

```
  unfold = (λ h. fix·(unfold1·h))
```

```
lemma unfold-beta: unfold·h = fix·(unfold1·h)
```

```
unfolding unfold-def by simp
```

```
lemma unfold: unfold·h·s = unfold1·h·(unfold·h)·s
```

```
unfolding unfold-beta by (subst fix-eq, simp)
```

```
lemma unfold-ind:
```

```
  fixes P :: ('s → 'a LList) ⇒ bool
```

```
  assumes adm: adm P
```

```
  assumes base: P ⊥
```

assumes $step: \bigwedge u. P u \implies P (unfold1 \cdot h \cdot u)$
shows $P (unfold \cdot h)$
unfolding $unfold\text{-}beta$ **by** (*rule* $fix\text{-}ind$ [*of* P , *OF* *adm* *base* *step*])

fixrec
 $unstream :: ('a, 's) Stream \rightarrow 'a LList$
where
 $s \neq \perp \implies unstream \cdot (Stream \cdot h \cdot s) = unfold \cdot h \cdot s$

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

2.3 Converting from lists to streams

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

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

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

lemma $stream\text{-}defined$ [*simp*]: $stream \cdot xs \neq \perp$
by $simp$

lemma $unstream\text{-}stream$ [*simp*]:
fixes $xs :: 'a LList$
shows $unstream \cdot (stream \cdot xs) = xs$
by (*induct* xs , *simp*-*all* *add*: *unfold*)

declare $stream\text{-}simps$ [*simp del*]

2.4 Bisimilarity relation on streams

definition
 $bisimilar :: ('a, 's) Stream \Rightarrow ('a, 't) Stream \Rightarrow bool$ (**infix** ≈ 50)
where
 $a \approx b \iff unstream \cdot a = unstream \cdot b \wedge a \neq \perp \wedge b \neq \perp$

lemma $unstream\text{-}cong$:
 $a \approx b \implies unstream \cdot a = unstream \cdot b$
unfolding $bisimilar\text{-}def$ **by** $simp$

lemma $stream\text{-}cong$:
 $xs = ys \implies stream \cdot xs \approx stream \cdot ys$
unfolding $bisimilar\text{-}def$ **by** $simp$

```

lemma stream-unstream-cong:
   $a \approx b \implies \text{stream}.\text{unstream}.a \approx b$ 
  unfolding bisimilar-def by simp

end

```

3 Stream Fusion

```

theory StreamFusion
imports Stream
begin

```

3.1 Type constructors for state types

```

domain Switch = S1 | S2

domain 'a Maybe = Nothing | Just 'a

hide (open) const Left Right

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

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

syntax Both :: type  $\Rightarrow$  type  $\Rightarrow$  type (infixl !: 25)

domain 'a L = L (lazy 'a)

```

3.2 Map function

```

fixrec
  mapStep :: ('a  $\rightarrow$  'b)  $\rightarrow$  ('s  $\rightarrow$  ('a, 's) Step)  $\rightarrow$  's  $\rightarrow$  ('b, 's) Step
where
  mapStep.f.h.s = (case h.s of
    Done  $\Rightarrow$  Done
  | Skip.s'  $\Rightarrow$  Skip.s'
  | Yield.x.s'  $\Rightarrow$  Yield.(f.x).s')

```

```

fixrec
  mapS :: ('a  $\rightarrow$  'b)  $\rightarrow$  ('a, 's) Stream  $\rightarrow$  ('b, 's) Stream
where
   $s \neq \perp \implies \text{mapS}.f.(Stream.h.s) = Stream.(mapStep.f.h).s$ 

```

```

lemma unfold-mapStep:
  fixes  $f :: 'a \rightarrow 'b$  and  $h :: 's \rightarrow ('a, 's) Step$ 
  assumes  $s \neq \perp$ 
  shows  $\text{unfold}.(mapStep.f.h).s = \text{mapL}.f.(unfold.h.s)$ 

```

proof (*rule antisym-less*)
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 $f :: 'a \rightarrow 'b$
fixes $a :: ('a, 's) \text{Stream}$
fixes $b :: ('a, 't) \text{Stream}$
shows $f = g \implies a \approx b \implies \text{mapS} \cdot f \cdot a \approx \text{mapS} \cdot g \cdot b$
unfolding *bisimilar-def*
by (*simp add: unstream-mapS mapS-defined*)

lemma *mapL-eq*: $\text{mapL} \cdot f \cdot xs = \text{unstream} \cdot (\text{mapS} \cdot f \cdot (\text{stream} \cdot xs))$
by (*simp add: unstream-mapS*)

3.3 Filter function

fixrec
 $\text{filterStep} :: ('a \rightarrow \text{tr}) \rightarrow ('s \rightarrow ('a, 's) \text{Step}) \rightarrow 's \rightarrow ('a, 's) \text{Step}$
where
 $\text{filterStep} \cdot p \cdot h \cdot s = (\text{case } h \cdot s \text{ of}$
 $\quad \text{Done} \Rightarrow \text{Done}$
 $\quad | \text{Skip} \cdot s' \Rightarrow \text{Skip} \cdot s'$
 $\quad | \text{Yield} \cdot x \cdot s' \Rightarrow (\text{If } p \cdot x \text{ then } \text{Yield} \cdot x \cdot s' \text{ else } \text{Skip} \cdot s' \text{ fi}))$

fixrec
 $\text{filterS} :: ('a \rightarrow \text{tr}) \rightarrow ('a, 's) \text{Stream} \rightarrow ('a, 's) \text{Stream}$
where
 $s \neq \perp \implies \text{filterS} \cdot p \cdot (\text{Stream} \cdot h \cdot s) = \text{Stream} \cdot (\text{filterStep} \cdot p \cdot h) \cdot s$

lemma *unfold-filterStep*:
fixes $p :: 'a \rightarrow tr$ **and** $h :: 's \rightarrow ('a, 's) Step$
assumes $s \neq \perp$
shows $unfold \cdot (filterStep \cdot p \cdot h) \cdot s = filterL \cdot p \cdot (unfold \cdot h \cdot s)$
proof (*rule antisym-less*)
show $unfold \cdot (filterStep \cdot p \cdot h) \cdot s \sqsubseteq filterL \cdot p \cdot (unfold \cdot h \cdot s)$
using $\langle s \neq \perp \rangle$
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 $filterL \cdot p \cdot (unfold \cdot h \cdot s) \sqsubseteq unfold \cdot (filterStep \cdot p \cdot h) \cdot s$
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*)
apply (*case-tac p.a rule: trE, simp-all add: unfold*)
done
qed

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

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

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

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

3.4 Foldr function

fixrec
 $foldrS :: ('a \rightarrow 'b \rightarrow 'b) \rightarrow 'b \rightarrow ('a, 's) Stream \rightarrow 'b$
where
 $foldrS \cdot Stream$:
 $s \neq \perp \implies foldrS \cdot f \cdot z \cdot (Stream \cdot h \cdot s) =$
(case h.s of Done \Rightarrow z

| $Skip \cdot s' \Rightarrow foldrS \cdot f \cdot z \cdot (Stream \cdot h \cdot s')$
| $Yield \cdot x \cdot s' \Rightarrow f \cdot x \cdot (foldrS \cdot f \cdot z \cdot (Stream \cdot h \cdot s'))$

lemma *unfold-foldrS*:

assumes $s \neq \perp$ **shows** $foldrS \cdot f \cdot z \cdot (Stream \cdot h \cdot s) = foldrL \cdot f \cdot z \cdot (unfold \cdot h \cdot s)$

proof (*rule antisym-less*)

show $foldrS \cdot f \cdot z \cdot (Stream \cdot h \cdot s) \sqsubseteq foldrL \cdot f \cdot z \cdot (unfold \cdot h \cdot s)$

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 $foldrL \cdot f \cdot z \cdot (unfold \cdot h \cdot s) \sqsubseteq foldrS \cdot f \cdot z \cdot (Stream \cdot h \cdot s)$

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*:

$a \neq \perp \implies foldrS \cdot f \cdot z \cdot a = foldrL \cdot f \cdot z \cdot (unstream \cdot a)$

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

lemma *foldrS-cong*:

fixes $a :: ('a, 's) Stream$

fixes $b :: ('a, 't) Stream$

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

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

lemma *foldrL-eq*:

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

by (*simp add: unstream-foldrS*)

3.5 EnumFromTo function

types $int' = int_{\perp}$

fixrec

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

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 $Done$)

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

$enumFromToStep \cdot \perp \cdot x''$

$enumFromToStep \cdot (up \cdot y) \cdot \perp$

$enumFromToStep \cdot (up \cdot y) \cdot (up \cdot \perp)$

```

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

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 antisym-less)
  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 ≤ 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: enumFromToL-induct)
  apply (simp, simp)
  apply (rename-tac e n)
  apply (case-tac n, simp)
  apply (case-tac x ≤ y, simp-all add: unfold)
  done
qed

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

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

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

```

lemma *enumFromToL-eq*: $enumFromToL.x.y = unstream.(enumFromToS.x.y)$
by (*simp add: unstream-enumFromToS*)

3.6 Append function

fixrec

appendStep ::
 ('s → ('a, 's) Step) →
 ('t → ('a, 't) Step) →
 't → ('s, 't) Either → ('a, ('s, 't) Either) Step

where

$sa \neq \perp \implies appendStep.ha.hb.sb0.(Left.sa) =$
 (case *ha.sa* of
 Done ⇒ *Skip*·(*Right.sb0*)
 | *Skip.sa'* ⇒ *Skip*·(*Left.sa'*)
 | *Yield.x.sa'* ⇒ *Yield.x*·(*Left.sa'*)
 | $sb \neq \perp \implies appendStep.ha.hb.sb0.(Right.sb) =$
 (case *hb.sb* of
 Done ⇒ *Done*
 | *Skip.sb'* ⇒ *Skip*·(*Right.sb'*)
 | *Yield.x.sb'* ⇒ *Yield.x*·(*Right.sb'*))

fixpat *appendStep-strict* [*simp*]: $appendStep.ha.hb.sb0.\perp$

fixrec

appendS ::
 ('a, 's) Stream → ('a, 't) Stream → ('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 → ('a, 's) Step

fixes *hb* :: 't → ('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))
  ^
  (∀ sb. sb ≠ ⊥ → unfold·?h·(Right·sb) ⊆ 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 = λua ub. ∀ sa. sa ≠ ⊥ →
  appendL·(ua·sa)·(ub·sb0) ⊆ unfold·?h·(Left·sa)

let ?Q = λub. ∀ sb. sb ≠ ⊥ → ub·sb ⊆ unfold·?h·(Right·sb)

have P-base: ∧ub. ?P ⊥ ub
  by simp

have Q-base: ?Q ⊥
  by simp

have P-step: ∧ua ub. ?P ua ub ⇒ ?Q ub ⇒ ?P (unfold1·ha·ua) ub
  apply (intro allI impI)
  apply (case-tac ha·sa, simp, simp, simp, simp)
  done

have Q-step: ∧ua ub. ?Q ub ⇒ ?Q (unfold1·hb·ub)
  apply (intro allI impI)
  apply (case-tac hb·sb, simp, simp, simp, simp)
  done

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

have P: ?P (unfold·ha) (unfold·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·ha) (unfold·hb) ∧ ?Q (unfold·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*: $xs \neq \perp \implies ys \neq \perp \implies \text{appendS} \cdot xs \cdot ys \neq \perp$
by (*cases xs, simp, cases ys, simp, simp*)

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

lemma *appendS-cong*:
fixes $f :: 'a \rightarrow 'b$
fixes $a :: ('a, 's) \text{Stream}$
fixes $b :: ('a, 't) \text{Stream}$
shows $a \approx a' \implies b \approx b' \implies \text{appendS} \cdot a \cdot b \approx \text{appendS} \cdot a' \cdot b'$
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 ::
 $('a \rightarrow 'b \rightarrow 'c) \rightarrow$
 $('s \rightarrow ('a, 's) \text{Step}) \rightarrow$
 $('t \rightarrow ('b, 't) \text{Step}) \rightarrow$
 $'s \text{!}:: 't \text{!}:: 'a \text{L Maybe} \rightarrow ('c, 's \text{!}:: 't \text{!}:: 'a \text{L Maybe}) \text{Step}$

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

fixpat *zipWithStep-strict* [*simp*]: $\text{zipWithStep} \cdot f \cdot ha \cdot hb \cdot \perp$

fixrec

$zipWithS :: ('a \rightarrow 'b \rightarrow 'c) \rightarrow$
 $('a, 's) Stream \rightarrow ('b, 't) Stream \rightarrow ('c, 's !: 't !: 'a L Maybe) Stream$
where
 $sa0 \neq \perp \implies sb0 \neq \perp \implies zipWithS.f \cdot (Stream.ha \cdot sa0) \cdot (Stream.hb \cdot sb0) =$
 $Stream \cdot (zipWithStep.f \cdot ha \cdot hb) \cdot (sa0 !: sb0 !: Nothing)$

lemma *zipWithS-fix-ind-lemma*:
fixes $P Q :: nat \Rightarrow nat \Rightarrow bool$
assumes $P-0: \bigwedge j. P\ 0\ j$ **and** $P-Suc: \bigwedge i\ j. P\ i\ j \implies Q\ i\ j \implies P\ (Suc\ i)\ j$
assumes $Q-0: \bigwedge i. Q\ i\ 0$ **and** $Q-Suc: \bigwedge i\ j. P\ i\ j \implies Q\ i\ j \implies Q\ i\ (Suc\ j)$
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-Suc*)
apply (*case-tac j, simp add: Q-0, simp add: Q-Suc*)
done

lemma *zipWithS-fix-ind*:
assumes $x: x = fix.f$ **and** $y: y = fix.g$
assumes $adm-P: adm\ (\lambda x. P\ (fst\ x)\ (snd\ x))$
assumes $adm-Q: adm\ (\lambda x. Q\ (fst\ x)\ (snd\ x))$
assumes $P-0: \bigwedge b. P\ \perp\ b$ **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 $1: \bigwedge i\ j. P\ (iterate\ i.f.\ \perp)\ (iterate\ j.g.\ \perp) \wedge Q\ (iterate\ i.f.\ \perp)\ (iterate\ j.g.\ \perp)$
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 *split* $P\ (\bigsqcup i. \langle iterate\ i.f.\ \perp, iterate\ i.g.\ \perp \rangle)$
apply (*rule admD*)
apply (*simp add: split-def adm-P*)
apply *simp*
apply (*simp add: cpair-eq-pair 1*)
done
then have $P: P\ x\ y$
unfolding $x\ y\ fix-def2$
by (*simp add: thelub-cprod, simp add: cpair-eq-pair*)
have *split* $Q\ (\bigsqcup i. \langle iterate\ i.f.\ \perp, iterate\ i.g.\ \perp \rangle)$
apply (*rule admD*)
apply (*simp add: split-def adm-Q*)
apply *simp*
apply (*simp add: cpair-eq-pair 1*)
done
then have $Q: Q\ x\ y$

unfolding $x\ y\ \text{fix-def2}$
 by (*simp add: thelub-cprod, simp add: cpair-eq-pair*)
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
 $(\forall sa\ sb.\ sa \neq \perp \longrightarrow sb \neq \perp \longrightarrow$
 $\text{unfold}\cdot h\cdot (sa\ !:\ sb\ !:\ \text{Nothing}) =$
 $\text{zipWithL}\cdot f\cdot (\text{unfold}\cdot ha\cdot sa)\cdot (\text{unfold}\cdot hb\cdot sb)) \wedge$
 $(\forall sa\ sb\ a.\ sa \neq \perp \longrightarrow sb \neq \perp \longrightarrow$
 $\text{unfold}\cdot h\cdot (sa\ !:\ sb\ !:\ \text{Just}\cdot (L\cdot a)) =$
 $\text{zipWithL}\cdot f\cdot (L\text{Cons}\cdot a\cdot (\text{unfold}\cdot ha\cdot sa))\cdot (\text{unfold}\cdot hb\cdot sb))$

proof –

note *unfold* [*simp*]
have *h-simps* [*simp*]:
 $\wedge sa\ sb.\ sa \neq \perp \Longrightarrow sb \neq \perp \Longrightarrow h\cdot (sa\ !:\ sb\ !:\ \text{Nothing}) =$
 $(\text{case } ha\cdot sa\ \text{of}$
 $\quad \text{Done} \Rightarrow \text{Done}$
 $\quad | \text{Skip}\cdot sa' \Rightarrow \text{Skip}\cdot (sa' !:\ sb !:\ \text{Nothing})$
 $\quad | \text{Yield}\cdot a\cdot sa' \Rightarrow \text{Skip}\cdot (sa' !:\ sb !:\ \text{Just}\cdot (L\cdot a)))$
 $\wedge sa\ sb\ a.\ sa \neq \perp \Longrightarrow sb \neq \perp \Longrightarrow h\cdot (sa\ !:\ sb !:\ \text{Just}\cdot (L\cdot a)) =$
 $(\text{case } hb\cdot sb\ \text{of}$
 $\quad \text{Done} \Rightarrow \text{Done}$
 $\quad | \text{Skip}\cdot sb' \Rightarrow \text{Skip}\cdot (sa !:\ sb' !:\ \text{Just}\cdot (L\cdot a))$
 $\quad | \text{Yield}\cdot b\cdot sb' \Rightarrow \text{Yield}\cdot (f\cdot a\cdot b)\cdot (sa !:\ sb' !:\ \text{Nothing}))$
 $h\cdot \perp = \perp$
unfolding *h-def* **by** *simp-all*

have *1*:

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

let $?P = \lambda ua\ ub.\ \forall sa\ sb.\ sa \neq \perp \longrightarrow sb \neq \perp \longrightarrow$

$zipWithL.f \cdot (ua \cdot sa) \cdot (ub \cdot sb) \sqsubseteq unfold \cdot h \cdot (sa \text{ !: } sb \text{ !: } Nothing)$

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

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

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

have $P\text{-step}: \bigwedge ua \ ub. ?P \ ua \ ub \Longrightarrow ?Q \ ua \ ub \Longrightarrow ?P \ (unfold1 \cdot ha \cdot ua) \ ub$
by (*clarsimp*, *case-tac* $ha \cdot sa$, *simp-all*)

have $Q\text{-step}: \bigwedge ua \ ub. ?P \ ua \ ub \Longrightarrow ?Q \ ua \ ub \Longrightarrow ?Q \ ua \ (unfold1 \cdot hb \cdot ub)$
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* $zipWithS\text{-fix-ind}$ [*OF* $unfold\text{-beta}$ [*of* ha] $unfold\text{-beta}$ [*of* hb]])
apply (*simp*, *simp*)
apply (*rule* $P\text{-base}$)
apply (*erule* (1) $P\text{-step}$)
apply (*rule* $Q\text{-base}$)
apply (*erule* (1) $Q\text{-step}$)
done

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

qed

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

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

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

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

by (*simp add: unstream-zipWithS*)

3.8 ConcatMap function

fixrec

concatMapStep ::
 ('a → ('b, 't) Stream) →
 ('s → ('a, 's) Step) →
 's !: ('b, 't) Stream Maybe →
 ('b, 's !: ('b, 't) Stream Maybe) Step

where

sa ≠ ⊥ ⇒ *concatMapStep*·*f*·*ha*·(*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'*)))

fixpat *concatMapStep-strict* [*simp*]: *concatMapStep*·*f*·*ha*·⊥

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*)

fixpat *concatMapS-strict* [*simp*]: *concatMapS*·*f*·⊥

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 –

note *unfold* [*simp*]

have *h-simps* [*simp*]:

$\bigwedge sa. sa \neq \perp \implies h.(sa \text{ !: } \text{Nothing}) =$
 $(\text{case } ha \cdot sa \text{ of } \text{Done} \Rightarrow \text{Done}$
 $| \text{Skip} \cdot sa' \Rightarrow \text{Skip} \cdot (sa' \text{ !: } \text{Nothing})$
 $| \text{Yield} \cdot a \cdot sa' \Rightarrow \text{Skip} \cdot (sa' \text{ !: } \text{Just} \cdot (f \cdot a)))$
 $\bigwedge sa \ hb \ sb. sa \neq \perp \implies sb \neq \perp \implies h.(sa \text{ !: } \text{Just} \cdot (\text{Stream} \cdot hb \cdot sb)) =$
 $(\text{case } hb \cdot sb \text{ of } \text{Done} \Rightarrow \text{Skip} \cdot (sa \text{ !: } \text{Nothing})$
 $| \text{Skip} \cdot sb' \Rightarrow \text{Skip} \cdot (sa \text{ !: } \text{Just} \cdot (\text{Stream} \cdot hb \cdot sb'))$
 $| \text{Yield} \cdot b \cdot sb' \Rightarrow \text{Yield} \cdot b \cdot (sa \text{ !: } \text{Just} \cdot (\text{Stream} \cdot hb \cdot sb')))$
 $h.\perp = \perp$
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:
 $(\forall sa. sa \neq \perp \longrightarrow$
 $\text{unfold} \cdot h \cdot (sa \text{ !: } \text{Nothing}) \sqsubseteq \text{concatMapL} \cdot f' \cdot (\text{unfold} \cdot ha \cdot sa))$
 \wedge
 $(\forall sa \ hb \ sb. sa \neq \perp \longrightarrow sb \neq \perp \longrightarrow$
 $\text{unfold} \cdot h \cdot (sa \text{ !: } \text{Just} \cdot (\text{Stream} \cdot hb \cdot sb)) \sqsubseteq$
 $\text{appendL} \cdot (\text{unfold} \cdot hb \cdot sb) \cdot (\text{concatMapL} \cdot f' \cdot (\text{unfold} \cdot ha \cdot sa)))$
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 (*rename-tac* $a \ sa'$)
apply (*case-tac* $f \cdot a$, *simp*, *simp*)
apply (*case-tac* $hb \cdot sb$, *simp*, *simp*, *simp*, *simp*, *simp*)
done

let $?P = \lambda ua. \forall sa. sa \neq \perp \longrightarrow$
 $\text{concatMapL} \cdot f' \cdot (ua \cdot sa) \sqsubseteq \text{unfold} \cdot h \cdot (sa \text{ !: } \text{Nothing})$

let $?Q = \lambda hb \ ua \ ub. \forall sa \ sb. sa \neq \perp \longrightarrow sb \neq \perp \longrightarrow$
 $\text{appendL} \cdot (ub \cdot sb) \cdot (\text{concatMapL} \cdot f' \cdot (ua \cdot sa)) \sqsubseteq$
 $\text{unfold} \cdot h \cdot (sa \text{ !: } \text{Just} \cdot (\text{Stream} \cdot hb \cdot sb))$

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

have P -step: $\bigwedge ua. ?P \ ua \implies \forall hb. ?Q \ hb \ ua \ (\text{unfold} \cdot hb) \implies ?P \ (\text{unfold1} \cdot ha \cdot ua)$
apply (*intro allI impI*)
apply (*case-tac* $ha \cdot sa$, *simp*, *simp*, *simp*, *simp*)
apply (*rename-tac* $a \ sa'$)
apply (*case-tac* $f \cdot a$, *simp*, *simp*)
done

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

have *Q-step*: $\bigwedge hb\ ua\ ub. ?P\ ua \implies ?Q\ hb\ ua\ ub \implies ?Q\ hb\ ua\ (unfold1\cdot hb\cdot ub)$
apply (*intro allI impI*)
apply (*case-tac hb.sb, simp, simp, simp, simp*)
done

have *2*: $?P\ (unfold\cdot ha) \wedge (\forall hb. ?Q\ hb\ (unfold\cdot ha)\ (unfold\cdot hb))$
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*:
 $unstream\cdot(concatMapS\cdot f\cdot a) = concatMapL\cdot(unstream\ oo f)\cdot(unstream\cdot a)$
by (*cases a, simp, simp add: unfold-concatMapStep*)

lemma *concatMapS-defined*: $a \neq \perp \implies concatMapS\cdot f\cdot a \neq \perp$
by (*induct a, simp-all*)

lemma *concatMapS-cong*:
fixes $f :: 'a \Rightarrow ('b, 's)\ Stream$
fixes $g :: 'a \Rightarrow ('b, 't)\ Stream$
fixes $a :: ('a, 'u)\ Stream$
fixes $b :: ('a, 'v)\ Stream$
shows $(\bigwedge x. f\ x \approx g\ x) \implies a \approx b \implies cont\ f \implies cont\ g \implies$
 $concatMapS\cdot(\bigwedge x. f\ x)\cdot a \approx concatMapS\cdot(\bigwedge x. g\ x)\cdot b$
unfolding *bisimilar-def*
by (*simp add: unstream-concatMapS oo-def concatMapS-defined*)

lemma *concatMapL-eq*:
 $concatMapL\cdot f\cdot xs = unstream\cdot(concatMapS\cdot(stream\ oo f)\cdot(stream\cdot xs))$
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 ext-cfun, 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.