Stream Fusion

Brian Huffman

April 12, 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 **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
  $mapL :: ('a \to 'b) \to 'a\ LList \to 'b\ LList$
where
  $mapL{\cdot}f{\cdot}LNil = LNil$
| $mapL{\cdot}f{\cdot}(LCons{\cdot}x{\cdot}xs) = LCons{\cdot}(f{\cdot}x){\cdot}(mapL{\cdot}f{\cdot}xs)$

lemma *mapL-strict* [*simp*]: $mapL{\cdot}f{\cdot}\bot = \bot$
by *fixrec-simp*

fixrec
  $filterL :: ('a \to tr) \to 'a\ LList \to 'a\ LList$
where
  $filterL{\cdot}p{\cdot}LNil = LNil$
| $filterL{\cdot}p{\cdot}(LCons{\cdot}x{\cdot}xs) =$
    $(If\ p{\cdot}x\ then\ LCons{\cdot}x{\cdot}(filterL{\cdot}p{\cdot}xs)\ else\ filterL{\cdot}p{\cdot}xs)$

lemma *filterL-strict* [*simp*]: $filterL{\cdot}p{\cdot}\bot = \bot$
by *fixrec-simp*

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

lemma *foldrL-strict* [*simp*]: $foldrL{\cdot}f{\cdot}z{\cdot}\bot = \bot$
by *fixrec-simp*

fixrec
  $enumFromToL :: int_\bot \to int_\bot \to (int_\bot)\ LList$
where
  $enumFromToL{\cdot}(up{\cdot}x){\cdot}(up{\cdot}y) =$
    $(if\ x \le y\ then\ LCons{\cdot}(up{\cdot}x){\cdot}(enumFromToL{\cdot}(up{\cdot}(x{+}1)){\cdot}(up{\cdot}y))\ else\ LNil)$

lemma *enumFromToL-simps′* [*simp*]:
  $x \le y \Longrightarrow$
   $enumFromToL{\cdot}(up{\cdot}x){\cdot}(up{\cdot}y) = LCons{\cdot}(up{\cdot}x){\cdot}(enumFromToL{\cdot}(up{\cdot}(x{+}1)){\cdot}(up{\cdot}y))$
  $\neg\ x \le y \Longrightarrow enumFromToL{\cdot}(up{\cdot}x){\cdot}(up{\cdot}y) = LNil$
  by *simp-all*

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

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

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*]: $appendL \cdot \bot \cdot ys = \bot$
by *fixrec-simp*

lemma *appendL-LNil-right*: $appendL \cdot xs \cdot LNil = xs$
by (*induct xs*) *simp-all*

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*]:
  $zipWithL \cdot f \cdot \bot \cdot ys = \bot$
  $zipWithL \cdot f \cdot (LCons \cdot x \cdot xs) \cdot \bot = \bot$
by *fixrec-simp*+

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*]: $concatMapL \cdot f \cdot \bot = \bot$
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 $('a,'s)$ *Step* = *Done* | *Skip* $'s$ | *Yield* (**lazy** $'a$) $'s$

type-synonym $('a,\ 's)$ *Stepper* = $'s \to ('a,\ 's)$ *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 (rule below-antisym)
  show unfold·h ⊑ fix·(unfoldF·h)
    apply (rule unfold.induct, simp, simp)
    apply (subst fix-eq)
    apply (rule cfun-belowI, rename-tac s)
    apply (case-tac s = ⊥, simp, simp)
    apply (intro monofun-cfun monofun-LAM below-refl, simp-all)
    done
  show fix·(unfoldF·h) ⊑ unfold·h
    apply (rule fix-ind, simp, simp)
    apply (subst unfold.unfold)
    apply (rule cfun-belowI, rename-tac s)
    apply (case-tac s = ⊥, simp, simp)
    apply (intro monofun-cfun monofun-LAM below-refl, simp-all)
    done
```

qed

lemma *unfold-ind*:
  fixes P :: $('s \to 'a\ LList) \Rightarrow bool$
  assumes $adm\ P$ **and** $P\ \bot$ **and** $\bigwedge u.\ P\ u \Longrightarrow P\ (unfoldF \cdot h \cdot u)$
  shows $P\ (unfold \cdot h)$
unfolding *unfold-eq-fix* **by** (*rule fix-ind* [*of P*, *OF assms*])

fixrec
  $unfold2 :: ('s \to 'a\ LList) \to ('a,\ 's)\ Step \to 'a\ LList$
where
  $unfold2 \cdot u \cdot Done = LNil$
$|\ s \neq \bot \Longrightarrow unfold2 \cdot u \cdot (Skip \cdot s) = u \cdot s$
$|\ s \neq \bot \Longrightarrow unfold2 \cdot u \cdot (Yield \cdot x \cdot s) = LCons \cdot x \cdot (u \cdot s)$

lemma *unfold2-strict* [*simp*]: $unfold2 \cdot u \cdot \bot = \bot$
by *fixrec-simp*

lemma *unfold*: $s \neq \bot \Longrightarrow unfold \cdot h \cdot s = unfold2 \cdot (unfold \cdot h) \cdot (h \cdot s)$
by (*case-tac* $h \cdot s$, *simp-all*)

lemma *unfoldF*: $s \neq \bot \Longrightarrow unfoldF \cdot h \cdot u \cdot s = unfold2 \cdot u \cdot (h \cdot s)$
by (*case-tac* $h \cdot s$, *simp-all*)

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

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

lemma *unstream-strict* [*simp*]: $unstream \cdot \bot = \bot$
by *fixrec-simp*

2.3 Converting from lists to streams

fixrec
  $streamStep :: ('a\ LList)_\bot \to ('a,\ ('a\ LList)_\bot)\ Step$
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*]: $streamStep \cdot (up \cdot \bot) = \bot$
by *fixrec-simp*

fixrec
  $stream :: 'a\ LList \to ('a,\ ('a\ LList)_\bot)\ Stream$

where
  $stream \cdot xs = Stream \cdot streamStep \cdot (up \cdot xs)$

lemma *stream-defined* [*simp*]: $stream \cdot xs \neq \bot$
  by *simp*

lemma *unstream-stream* [*simp*]:
  fixes $xs :: 'a\ LList$
  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** $\approx$ *50*)
where
  $a \approx b \longleftrightarrow unstream \cdot a = unstream \cdot b \wedge a \neq \bot \wedge b \neq \bot$

lemma *unstream-cong*:
  $a \approx b \Longrightarrow unstream \cdot a = unstream \cdot b$
    unfolding *bisimilar-def* **by** *simp*

lemma *stream-cong*:
  $xs = ys \Longrightarrow stream \cdot xs \approx stream \cdot ys$
    unfolding *bisimilar-def* **by** *simp*

lemma *stream-unstream-cong*:
  $a \approx b \Longrightarrow stream \cdot (unstream \cdot a) \approx b$
    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 $'a\ Maybe = Nothing \mid Just\ 'a$

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

domain $('a, 'b)\ Either = Left\ 'a \mid Right\ 'b$

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

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

3.2 Map function

fixrec
  $mapStep :: ('a \to 'b) \to ('s \to ('a,\ 's)\ Step) \to 's \to ('b,\ 's)\ Step$
where
  $mapStep{\cdot}f{\cdot}h{\cdot}\bot = \bot$
$|\ s \neq \bot \Longrightarrow mapStep{\cdot}f{\cdot}h{\cdot}s = (case\ h{\cdot}s\ of$
    $Done \Rightarrow Done$
  $|\ Skip{\cdot}s' \Rightarrow Skip{\cdot}s'$
  $|\ Yield{\cdot}x{\cdot}s' \Rightarrow Yield{\cdot}(f{\cdot}x){\cdot}s')$

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 (*rule below-antisym*)
  show $unfold{\cdot}(mapStep{\cdot}f{\cdot}h){\cdot}s \sqsubseteq mapL{\cdot}f{\cdot}(unfold{\cdot}h{\cdot}s)$
  using ‹$s \neq \bot$›
  apply (*induct arbitrary*: *s rule*: *unfold-ind* [**where** $h{=}mapStep{\cdot}f{\cdot}h$])
  apply (*simp*, *simp*)
  apply (*case-tac* $h{\cdot}s$, *simp-all add*: *unfold*)
  done
next
  show $mapL{\cdot}f{\cdot}(unfold{\cdot}h{\cdot}s) \sqsubseteq unfold{\cdot}(mapStep{\cdot}f{\cdot}h){\cdot}s$
  using ‹$s \neq \bot$›
  apply (*induct arbitrary*: *s rule*: *unfold-ind* [**where** $h{=}h$])
  apply (*simp*, *simp*)
  apply (*case-tac* $h{\cdot}s$, *simp-all add*: *unfold*)
  done
qed

lemma *unstream-mapS*:
  fixes $f :: 'a \to 'b$ **and** $a :: ('a,\ 's)\ Stream$
  shows $a \neq \bot \Longrightarrow unstream{\cdot}(mapS{\cdot}f{\cdot}a) = mapL{\cdot}f{\cdot}(unstream{\cdot}a)$
by (*induct a*, *simp*, *simp add*: *unfold-mapStep*)

lemma *mapS-defined*: $a \neq \bot \Longrightarrow mapS{\cdot}f{\cdot}a \neq \bot$
by (*induct a*, *simp-all*)

lemma *mapS-cong*:
  fixes $f :: 'a \to 'b$
  fixes $a :: ('a, 's)$ *Stream*
  fixes $b :: ('a, 't)$ *Stream*
  shows $f = g \Longrightarrow a \approx b \Longrightarrow mapS \cdot f \cdot a \approx mapS \cdot g \cdot b$
unfolding *bisimilar-def*
by (*simp add*: *unstream-mapS mapS-defined*)

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

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 (*rule below-antisym*)
  show $unfold \cdot (filterStep \cdot p \cdot h) \cdot s \sqsubseteq filterL \cdot p \cdot (unfold \cdot h \cdot s)$
  using ‹$s \neq \bot$›
  apply (*induct arbitrary*: *s rule*: *unfold-ind* [**where** h=$filterStep \cdot p \cdot h$])
  apply (*simp*, *simp*)
  apply (*case-tac* $h \cdot s$, *simp-all add*: *unfold*)
  apply (*case-tac* $p \cdot a$ *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 ‹$s \neq \bot$›
  apply (*induct arbitrary*: *s rule*: *unfold-ind* [**where** h=h])
  apply (*simp*, *simp*)
  apply (*case-tac* $h \cdot s$, *simp-all add*: *unfold*)
  apply (*case-tac* $p \cdot a$ *rule*: *trE*, *simp-all add*: *unfold*)
  done
qed

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

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

lemma *filterS-cong*:
  fixes $p :: 'a \to tr$
  fixes $a :: ('a, 's)\ Stream$
  fixes $b :: ('a, 't)\ Stream$
  shows $p = q \Longrightarrow a \approx b \Longrightarrow filterS\cdot p\cdot a \approx filterS\cdot q\cdot b$
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
  $foldrS :: ('a \to 'b \to 'b) \to 'b \to ('a, 's)\ Stream \to 'b$
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 (*rule below-antisym*)
  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 ‹$s \neq \bot$›
  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 ‹$s \neq \bot$›
  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*:
  $a \neq \bot \Longrightarrow foldrS\cdot f\cdot z\cdot a = foldrL\cdot f\cdot z\cdot(unstream\cdot a)$

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 $f = g \Longrightarrow z = w \Longrightarrow a \approx b \Longrightarrow foldrS{\cdot}f{\cdot}z{\cdot}a = foldrS{\cdot}g{\cdot}w{\cdot}b$
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

type-synonym $int' = int_\perp$

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

lemma *enumFromToStep-strict* [*simp*]:
  $enumFromToStep{\cdot}\perp{\cdot}x'' = \perp$
  $enumFromToStep{\cdot}(up{\cdot}y){\cdot}\perp = \perp$
  $enumFromToStep{\cdot}(up{\cdot}y){\cdot}(up{\cdot}\perp) = \perp$
by *fixrec-simp+*

lemma *enumFromToStep-simps'* [*simp*]:
  $x \le y \Longrightarrow enumFromToStep{\cdot}(up{\cdot}y){\cdot}(up{\cdot}(up{\cdot}x)) =$
    $Yield{\cdot}(up{\cdot}x){\cdot}(up{\cdot}(up{\cdot}(x{+}1)))$
  $\neg\ x \le y \Longrightarrow enumFromToStep{\cdot}(up{\cdot}y){\cdot}(up{\cdot}(up{\cdot}x)) = Done$
  by *simp-all*

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 (*rule below-antisym*)
  show $unfold{\cdot}(enumFromToStep{\cdot}(up{\cdot}y)){\cdot}(up{\cdot}n) \sqsubseteq enumFromToL{\cdot}n{\cdot}(up{\cdot}y)$
   apply (*induct arbitrary*: *n rule*: *unfold-ind* [**where** $h{=}enumFromToStep{\cdot}(up{\cdot}y)$])
    apply (*simp*, *simp*)

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

lemma *unstream-enumFromToS*:
$unstream \cdot (enumFromToS \cdot x \cdot y) = enumFromToL \cdot x \cdot y$
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*: $enumFromToS \cdot x \cdot y \neq \bot$
by (*simp add*: *enumFromToS.simps*)

lemma *enumFromToS-cong*:
$x = x' \Longrightarrow y = y' \Longrightarrow enumFromToS \cdot x \cdot y \approx enumFromToS \cdot x' \cdot y'$
unfolding *bisimilar-def* **by** (*simp add*: *enumFromToS-defined*)

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

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*]: $appendStep \cdot ha \cdot hb \cdot sb0 \cdot \bot = \bot$
by *fixrec-simp*

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 −
note *unfold* [*simp*]
let $?h = appendStep \cdot ha \cdot hb \cdot sb0$

have *1*:
$(\forall sa.\ sa \neq \bot \longrightarrow$
$unfold \cdot ?h \cdot (Left \cdot sa) \sqsubseteq$
$appendL \cdot (unfold \cdot ha \cdot sa) \cdot (unfold \cdot hb \cdot sb0))$
$\wedge$
$(\forall sb.\ sb \neq \bot \longrightarrow unfold \cdot ?h \cdot (Right \cdot sb) \sqsubseteq unfold \cdot hb \cdot sb)$
apply (*rule unfold-ind* [**where** $h = ?h$])
apply *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 $?P = \lambda ua\ ub.\ \forall sa.\ sa \neq \bot \longrightarrow$
$appendL \cdot (ua \cdot sa) \cdot (ub \cdot sb0) \sqsubseteq unfold \cdot ?h \cdot (Left \cdot sa)$

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

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

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

```
  have P-step: ⋀ua ub. ?P ua ub ⟹ ?Q ub ⟹ ?P (unfoldF·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 (unfoldF·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 ≠ ⊥ ⟹ ys ≠ ⊥ ⟹ appendS·xs·ys ≠ ⊥
by (cases xs, simp, cases ys, simp, simp)

lemma unstream-appendS:
  a ≠ ⊥ ⟹ b ≠ ⊥ ⟹
    unstream·(appendS·a·b) = appendL·(unstream·a)·(unstream·b)
apply (cases a, simp, cases b, simp)
apply (simp add: unfold-appendStep)
done

lemma appendS-cong:
  fixes f :: 'a → 'b
  fixes a :: ('a, 's) Stream
  fixes b :: ('a, 't) Stream
  shows a ≈ a' ⟹ b ≈ b' ⟹ appendS·a·b ≈ appendS·a'·b'
unfolding bisimilar-def
by (simp add: unstream-appendS appendS-defined)
```

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

3.7 ZipWith function

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

lemma *zipWithStep-strict* [*simp*]: $zipWithStep \cdot f \cdot ha \cdot hb \cdot \bot = \bot$
by *fixrec-simp*

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 $P\ i\ j \wedge Q\ i\ j$
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: $x = fix \cdot f$ **and** y: $y = fix \cdot g$

```
  assumes adm-P: adm (λx. P (fst x) (snd x))
  assumes adm-Q: adm (λx. Q (fst x) (snd x))
  assumes P-0: ⋀b. P ⊥ b and P-Suc: ⋀a b. P a b ⟹ Q a b ⟹ P (f·a) b
  assumes Q-0: ⋀a. Q a ⊥ and Q-Suc: ⋀a b. P a b ⟹ Q a b ⟹ Q a (g·b)
  shows P x y ∧ Q x y
proof −
  have 1: ⋀i j. P (iterate i·f·⊥) (iterate j·g·⊥) ∧ Q (iterate i·f·⊥) (iterate j·g·⊥)
    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 (⨆i. (iterate i·f·⊥, iterate i·g·⊥))
    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 (⨆i. (iterate i·f·⊥, iterate i·g·⊥))
    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 → 'b → 'c
  fixes ha :: 's → ('a, 's) Step
  fixes hb :: 't → ('b, 't) Step
  defines h-def: h ≡ zipWithStep·f·ha·hb
  shows
  (∀ sa sb. sa ≠ ⊥ ⟶ sb ≠ ⊥ ⟶
    unfold·h·(sa :!: sb :!: Nothing) =
      zipWithL·f·(unfold·ha·sa)·(unfold·hb·sb)) ∧
   (∀ sa sb a. sa ≠ ⊥ ⟶ sb ≠ ⊥ ⟶
    unfold·h·(sa :!: sb :!: Just·(L·a)) =
      zipWithL·f·(LCons·a·(unfold·ha·sa))·(unfold·hb·sb))
proof −
  note unfold [simp]
  have h-simps [simp]:
```

```
  ⋀sa sb. sa ≠ ⊥ ⟹ sb ≠ ⊥ ⟹ h·(sa :!: sb :!: Nothing) =
    (case ha·sa of
      Done ⇒ Done
    | Skip·sa' ⇒ Skip·(sa' :!: sb :!: Nothing)
    | Yield·a·sa' ⇒ Skip·(sa' :!: sb :!: Just·(L·a)))
  ⋀sa sb a. sa ≠ ⊥ ⟹ sb ≠ ⊥ ⟹ h·(sa :!: sb :!: Just·(L·a)) =
    (case hb·sb of
      Done ⇒ Done
    | Skip·sb' ⇒ Skip·(sa :!: sb' :!: Just·(L·a))
    | Yield·b·sb' ⇒ Yield·(f·a·b)·(sa :!: sb' :!: Nothing))
  h·⊥ = ⊥
  unfolding h-def by simp-all

have 1:
(∀ sa sb. sa ≠ ⊥ ⟶ sb ≠ ⊥ ⟶
  unfold·h·(sa :!: sb :!: Nothing) ⊑
    zipWithL·f·(unfold·ha·sa)·(unfold·hb·sb))
 ∧
 (∀ sa sb a. sa ≠ ⊥ ⟶ sb ≠ ⊥ ⟶
  unfold·h·(sa :!: sb :!: Just·(L·a)) ⊑
    zipWithL·f·(LCons·a·(unfold·ha·sa))·(unfold·hb·sb))
  apply (rule unfold-ind [where h=h], 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 sb. sa ≠ ⊥ ⟶ sb ≠ ⊥ ⟶
      zipWithL·f·(ua·sa)·(ub·sb) ⊑ unfold·h·(sa :!: sb :!: Nothing)

let ?Q = λua ub. ∀ sa sb a. sa ≠ ⊥ ⟶ sb ≠ ⊥ ⟶
      zipWithL·f·(LCons·a·(ua·sa))·(ub·sb) ⊑
        unfold·h·(sa :!: sb :!: Just·(L·a))

have P-base: ⋀ub. ?P ⊥ ub
  by simp

have Q-base: ⋀ua. ?Q ua ⊥
  by simp

have P-step: ⋀ua ub. ?P ua ub ⟹ ?Q ua ub ⟹ ?P (unfoldF·ha·ua) ub
  by (clarsimp, case-tac ha·sa, simp-all)

have Q-step: ⋀ua ub. ?P ua ub ⟹ ?Q ua ub ⟹ ?Q ua (unfoldF·hb·ub)
  by (clarsimp, case-tac hb·sb, simp-all)

have 2: ?P (unfold·ha) (unfold·hb) ∧ ?Q (unfold·ha) (unfold·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 ≠ ⊥ ⟹ b ≠ ⊥ ⟹ zipWithS·f·a·b ≠ ⊥
by (cases a, simp, cases b, simp, simp)

lemma unstream-zipWithS:
  a ≠ ⊥ ⟹ b ≠ ⊥ ⟹
    unstream·(zipWithS·f·a·b) = zipWithL·f·(unstream·a)·(unstream·b)
apply (cases a, simp, cases b, simp)
apply (simp add: unfold-zipWithStep)
done

lemma zipWithS-cong:
  f = f' ⟹ a ≈ a' ⟹ b ≈ b' ⟹
    zipWithS·f·a·b ≈ zipWithS·f·a'·b'
unfolding bisimilar-def
by (simp add: unstream-zipWithS zipWithS-defined)

lemma zipWithL-eq:
  zipWithL·f·xs·ys = unstream·(zipWithS·f·(stream·xs)·(stream·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)
```

$\mid$ *Skip*·*sb'* ⇒ *Skip*·(*sa* :!: *Just*·(*Stream*·*hb*·*sb'*))
$\mid$ *Yield*·*b*·*sb'* ⇒ *Yield*·*b*·(*sa* :!: *Just*·(*Stream*·*hb*·*sb'*)))

lemma *concatMapStep-strict* [*simp*]: *concatMapStep*·*f*·*ha*·⊥ = ⊥
by *fixrec-simp*

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*·⊥ = ⊥
by *fixrec-simp*

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*]:
⋀*sa*. *sa* ≠ ⊥ ⟹ *h*·(*sa* :!: *Nothing*) =
(*case ha*·*sa of Done* ⇒ *Done*
$\mid$ *Skip*·*sa'* ⇒ *Skip*·(*sa'* :!: *Nothing*)
$\mid$ *Yield*·*a*·*sa'* ⇒ *Skip*·(*sa'* :!: *Just*·(*f*·*a*)))
⋀*sa hb sb*. *sa* ≠ ⊥ ⟹ *sb* ≠ ⊥ ⟹ *h*·(*sa* :!: *Just*·(*Stream*·*hb*·*sb*)) =
(*case hb*·*sb of Done* ⇒ *Skip*·(*sa* :!: *Nothing*)
$\mid$ *Skip*·*sb'* ⇒ *Skip*·(*sa* :!: *Just*·(*Stream*·*hb*·*sb'*))
$\mid$ *Yield*·*b*·*sb'* ⇒ *Yield*·*b*·(*sa* :!: *Just*·(*Stream*·*hb*·*sb'*)))
h·⊥ = ⊥
unfolding *h-def* **by** *simp-all*

have *f'-beta* [*simp*]: ⋀*a*. *f'*·*a* = *unstream*·(*f*·*a*)
unfolding *f'-def* **by** *simp*

have *1*:
(∀ *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)))
  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*:
$unstream \cdot (concatMapS \cdot f \cdot a) = concatMapL \cdot (unstream\ oo\ f) \cdot (unstream \cdot a)$
by (*cases a*, *simp*, *simp add*: *unfold-concatMapStep*)

lemma *concatMapS-defined*: $a \neq \bot \Longrightarrow concatMapS \cdot f \cdot a \neq \bot$
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 $(\bigwedge x.\ f\ x \approx g\ x) \Longrightarrow a \approx b \Longrightarrow cont\ f \Longrightarrow cont\ g \Longrightarrow$
$concatMapS \cdot (\Lambda\ x.\ f\ x) \cdot a \approx concatMapS \cdot (\Lambda\ x.\ g\ x) \cdot b$
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 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* $\Longrightarrow$ *f oo* (Λ *x*. *g x*) = (Λ *x*. *f*·(*g x*))
unfolding *oo-def* **by** *simp*

lemma
concatMapL·(Λ *k*.
mapL·(Λ *m*. *f·k·m)·(enumFromToL·one·k))·(enumFromToL·one·n) =*
unstream·(concatMapS·(Λ *k*.
mapS·(Λ *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.