

# 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

<b>1</b>	<b>Lazy Lists</b>	<b>2</b>
<b>2</b>	<b>Stream Iterators</b>	<b>3</b>
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
<b>3</b>	<b>Stream Fusion</b>	<b>6</b>
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 . . . . .	8
3.6	Append function . . . . .	9
3.7	ZipWith function . . . . .	10
3.8	ConcatMap function . . . . .	12
3.9	Examples . . . . .	13

# 1 Lazy Lists

```
theory LazyList
imports HOLCF
begin
```

Discrete cpo instance for *int*.

```
instantiation int :: discrete-cpo
begin
```

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

```
instance <proof>
```

```
end
```

```
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)
```

```
fixpat mapL-strict [simp]: mapL.f.⊥
```

```
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 fi)
```

```
fixpat filterL-strict [simp]: filterL.p.⊥
```

```
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)
```

```
fixpat foldrL-strict [simp]: foldrL.f.z.⊥
```

```
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]:  
 $x \leq y \implies$   
 $enumFromToL \cdot (up \cdot x) \cdot (up \cdot y) = LCons \cdot (up \cdot x) \cdot (enumFromToL \cdot (up \cdot (x+1)) \cdot (up \cdot y))$   
 $\neg x \leq y \implies enumFromToL \cdot (up \cdot x) \cdot (up \cdot y) = LNil$   
 ⟨proof⟩

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

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

$enumFromToL \cdot \perp \cdot y = \perp$   
 $enumFromToL \cdot x \cdot \perp = \perp$   
 ⟨proof⟩

**fixrec**

$appendL :: 'a \text{ LList} \rightarrow 'a \text{ LList} \rightarrow 'a \text{ LList}$

**where**

$appendL \cdot LNil \cdot ys = ys$   
 $| appendL \cdot (LCons \cdot x \cdot xs) \cdot ys = LCons \cdot x \cdot (appendL \cdot xs \cdot ys)$

**fixpat** *appendL-strict* [simp]:  $appendL \cdot \perp \cdot ys$

**lemma** *appendL-LNil-right*:  $appendL \cdot xs \cdot LNil = xs$

⟨proof⟩

**fixrec**

$zipWithL :: ('a \rightarrow 'b \rightarrow 'c) \rightarrow 'a \text{ LList} \rightarrow 'b \text{ LList} \rightarrow 'c \text{ LList}$

**where**

$zipWithL \cdot f \cdot LNil \cdot ys = LNil$   
 $| zipWithL \cdot f \cdot (LCons \cdot x \cdot xs) \cdot LNil = LNil$   
 $| zipWithL \cdot f \cdot (LCons \cdot x \cdot xs) \cdot (LCons \cdot y \cdot ys) = LCons \cdot (f \cdot x \cdot y) \cdot (zipWithL \cdot f \cdot xs \cdot ys)$

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

$zipWithL \cdot f \cdot \perp \cdot ys$   
 $zipWithL \cdot f \cdot (LCons \cdot x \cdot xs) \cdot \perp$

**fixrec**

$concatMapL :: ('a \rightarrow 'b \text{ LList}) \rightarrow 'a \text{ LList} \rightarrow 'b \text{ LList}$

**where**

$concatMapL \cdot f \cdot LNil = LNil$   
 $| concatMapL \cdot f \cdot (LCons \cdot x \cdot xs) = appendL \cdot (f \cdot x) \cdot (concatMapL \cdot f \cdot xs)$

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

**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**  $(\prime a, \prime s)$  *Step* = *Done* | *Skip*  $\prime s$  | *Yield* (**lazy**  $\prime a$ )  $\prime s$

**domain**  $(\prime a, \prime s)$  *Stream* = *Stream* (**lazy**  $\prime s \rightarrow (\prime a, \prime s)$  *Step*)  $\prime s$

## 2.2 Converting from streams to lists

**fixrec**

$unfold2 :: (\prime s \rightarrow \prime a$  *LList*)  $\rightarrow (\prime a, \prime s)$  *Step*  $\rightarrow \prime a$  *LList*

**where**

$unfold2 \cdot u \cdot Done = LNil$

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

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

**fixpat** *unfold2-strict* [*simp*]:  $unfold2 \cdot u \cdot \perp$

**definition**

$unfold1 :: (\prime s \rightarrow (\prime a, \prime s)$  *Step*)  $\rightarrow (\prime s \rightarrow \prime a$  *LList*)  $\rightarrow (\prime s \rightarrow \prime a$  *LList*)

**where**

$unfold1 = (\Lambda h u. strictify \cdot (\Lambda s. unfold2 \cdot u \cdot (h \cdot s)))$

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

$unfold1 \cdot h \cdot u \cdot \perp = \perp$

$s \neq \perp \implies unfold1 \cdot h \cdot u \cdot s = unfold2 \cdot u \cdot (h \cdot s)$

*<proof>*

**definition**

$unfold :: (\prime s \rightarrow (\prime a, \prime s)$  *Step*)  $\rightarrow \prime s \rightarrow \prime a$  *LList*

**where**

$unfold = (\Lambda h. fix \cdot (unfold1 \cdot h))$

**lemma** *unfold-beta*:  $unfold \cdot h = fix \cdot (unfold1 \cdot h)$

*<proof>*

**lemma** *unfold*:  $unfold \cdot h \cdot s = unfold1 \cdot h \cdot (unfold \cdot h) \cdot s$

*<proof>*

**lemma** *unfold-ind*:

**fixes**  $P :: (\prime s \rightarrow \prime a$  *LList*)  $\Rightarrow bool$

**assumes** *adm*: *adm*  $P$

**assumes** *base*:  $P \perp$

**assumes** *step*:  $\bigwedge u. P u \implies P (unfold1 \cdot h \cdot u)$

**shows**  $P (unfold \cdot h)$

*<proof>*

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

**fixpat** *unstream-strict* [simp]:  $unstream.\perp$

## 2.3 Converting from lists to streams

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

**fixpat** *streamStep-strict* [simp]:  $streamStep.(up.\perp)$

**fixrec**  
 $stream :: 'a LList \rightarrow ('a, ('a LList)\perp) Stream$   
**where**  
 $stream.xs = Stream.streamStep.(up.xs)$

**lemma** *stream-defined* [simp]:  $stream.xs \neq \perp$   
 $\langle proof \rangle$

**lemma** *unstream-stream* [simp]:  
**fixes**  $xs :: 'a LList$   
**shows**  $unstream.(stream.xs) = xs$   
 $\langle proof \rangle$

**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**  
 $a \approx b \iff unstream.a = unstream.b \wedge a \neq \perp \wedge b \neq \perp$

**lemma** *unstream-cong*:  
 $a \approx b \implies unstream.a = unstream.b$   
 $\langle proof \rangle$

**lemma** *stream-cong*:  
 $xs = ys \implies stream.xs \approx stream.ys$   
 $\langle proof \rangle$

**lemma** *stream-unstream-cong*:  
 $a \approx b \implies stream.(unstream.a) \approx b$

*<proof>*

**end**

### 3 Stream Fusion

**theory** *StreamFusion*  
**imports** *Stream*  
**begin**

#### 3.1 Type constructors for state types

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

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

**hide (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*  $\cdot$  *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*  $\rightarrow$  *'b*)  $\rightarrow$  (*'a, 's*) *Stream*  $\rightarrow$  (*'b, 's*) *Stream*

**where**

*s*  $\neq \perp \implies$  *mapS*  $\cdot$  *f*  $\cdot$  (*Stream*  $\cdot$  *h*  $\cdot$  *s*) = *Stream*  $\cdot$  (*mapStep*  $\cdot$  *f*  $\cdot$  *h*)  $\cdot$  *s*

**lemma** *unfold-mapStep*:

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

**assumes** *s*  $\neq \perp$

**shows** *unfold*  $\cdot$  (*mapStep*  $\cdot$  *f*  $\cdot$  *h*)  $\cdot$  *s* = *mapL*  $\cdot$  *f*  $\cdot$  (*unfold*  $\cdot$  *h*  $\cdot$  *s*)

*<proof>*

**lemma** *unstream-mapS*:

**fixes**  $f :: 'a \rightarrow 'b$  **and**  $a :: ('a, 's) Stream$   
**shows**  $a \neq \perp \implies unstream \cdot (mapS \cdot f \cdot a) = mapL \cdot f \cdot (unstream \cdot a)$   
 $\langle proof \rangle$

**lemma** *mapS-defined*:  $a \neq \perp \implies mapS \cdot f \cdot a \neq \perp$   
 $\langle proof \rangle$

**lemma** *mapS-cong*:  
**fixes**  $f :: 'a \rightarrow 'b$   
**fixes**  $a :: ('a, 's) Stream$   
**fixes**  $b :: ('a, 't) Stream$   
**shows**  $f = g \implies a \approx b \implies mapS \cdot f \cdot a \approx mapS \cdot g \cdot b$   
 $\langle proof \rangle$

**lemma** *mapL-eq*:  $mapL \cdot f \cdot xs = unstream \cdot (mapS \cdot f \cdot (stream \cdot xs))$   
 $\langle proof \rangle$

### 3.3 Filter function

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

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

**lemma** *unfold-filterStep*:  
**fixes**  $p :: 'a \rightarrow tr$  **and**  $h :: 's \rightarrow ('a, 's) Step$   
**assumes**  $s \neq \perp$   
**shows**  $unfold \cdot (filterStep \cdot p \cdot h) \cdot s = filterL \cdot p \cdot (unfold \cdot h \cdot s)$   
 $\langle proof \rangle$

**lemma** *unstream-filterS*:  
 $a \neq \perp \implies unstream \cdot (filterS \cdot p \cdot a) = filterL \cdot p \cdot (unstream \cdot a)$   
 $\langle proof \rangle$

**lemma** *filterS-defined*:  $a \neq \perp \implies filterS \cdot p \cdot a \neq \perp$   
 $\langle proof \rangle$

**lemma** *filterS-cong*:  
**fixes**  $p :: 'a \rightarrow tr$   
**fixes**  $a :: ('a, 's) Stream$   
**fixes**  $b :: ('a, 't) Stream$

**shows**  $p = q \implies a \approx b \implies \text{filterS} \cdot p \cdot a \approx \text{filterS} \cdot q \cdot b$   
 ⟨proof⟩

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

### 3.4 Foldr function

**fixrec**

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

**where**

*foldrS-Stream*:

$s \neq \perp \implies \text{foldrS} \cdot f \cdot z \cdot (\text{Stream} \cdot h \cdot s) =$   
 (case  $h \cdot s$  of *Done*  $\Rightarrow z$   
 | *Skip*  $\cdot s' \Rightarrow \text{foldrS} \cdot f \cdot z \cdot (\text{Stream} \cdot h \cdot s')$   
 | *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⟩

**lemma** *unstream-foldrS*:

$a \neq \perp \implies \text{foldrS} \cdot f \cdot z \cdot a = \text{foldrL} \cdot f \cdot z \cdot (\text{unstream} \cdot a)$   
 ⟨proof⟩

**lemma** *foldrS-cong*:

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

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

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

**lemma** *foldrL-eq*:

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

### 3.5 EnumFromTo function

**types**  $\text{int}' = \text{int}_{\perp}$

**fixrec**

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

**where**

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

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

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

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

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

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

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

**fixrec**

*enumFromToS* :: *int'* → *int'* → (*int'*, (*int'*)<sub>⊥</sub>) *Stream*

**where**

*enumFromToS* · *x* · *y* = *Stream* · (*enumFromToStep* · *y*) · (*up* · *x*)

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

**lemma** *unfold-enumFromToStep*:

*unfold* · (*enumFromToStep* · (*up* · *y*)) · (*up* · *n*) = *enumFromToL* · *n* · (*up* · *y*)  
*<proof>*

**lemma** *unstream-enumFromToS*:

*unstream* · (*enumFromToS* · *x* · *y*) = *enumFromToL* · *x* · *y*  
*<proof>*

**lemma** *enumFromToS-defined*: *enumFromToS* · *x* · *y* ≠ ⊥  
*<proof>*

**lemma** *enumFromToS-cong*:

$x = x' \implies y = y' \implies \text{enumFromToS} \cdot x \cdot y \approx \text{enumFromToS} \cdot x' \cdot y'$   
*<proof>*

**lemma** *enumFromToL-eq*: *enumFromToL* · *x* · *y* = *unstream* · (*enumFromToS* · *x* · *y*)  
*<proof>*

### 3.6 Append function

**fixrec**

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

**where**

$sa \neq \perp \implies \text{appendStep} \cdot ha \cdot hb \cdot sb0 \cdot (\text{Left} \cdot sa) =$   
(case *ha* · *sa* of  
*Done* ⇒ *Skip* · (*Right* · *sb0*)  
| *Skip* · *sa'* ⇒ *Skip* · (*Left* · *sa'*)  
| *Yield* · *x* · *sa'* ⇒ *Yield* · *x* · (*Left* · *sa'*)  
| *sb* ≠ ⊥ ⇒ *appendStep* · *ha* · *hb* · *sb0* · (*Right* · *sb*) =  
(case *hb* · *sb* of  
*Done* ⇒ *Done*

|  $Skip \cdot sb' \Rightarrow Skip \cdot (Right \cdot sb')$   
|  $Yield \cdot x \cdot sb' \Rightarrow Yield \cdot x \cdot (Right \cdot sb')$

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

**fixrec**

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

**where**

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

**lemma** *unfold-appendStep*:

**fixes**  $ha :: 's \rightarrow ('a, 's) Step$

**fixes**  $hb :: 't \rightarrow ('a, 't) Step$

**assumes**  $sb0$  [*simp*]:  $sb0 \neq \perp$

**shows**

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

*<proof>*

**lemma** *appendS-defined*:  $xs \neq \perp \implies ys \neq \perp \implies appendS \cdot xs \cdot ys \neq \perp$

*<proof>*

**lemma** *unstream-appendS*:

$a \neq \perp \implies b \neq \perp \implies$

$unstream \cdot (appendS \cdot a \cdot b) = appendL \cdot (unstream \cdot a) \cdot (unstream \cdot b)$

*<proof>*

**lemma** *appendS-cong*:

**fixes**  $f :: 'a \rightarrow 'b$

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

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

**shows**  $a \approx a' \implies b \approx b' \implies appendS \cdot a \cdot b \approx appendS \cdot a' \cdot b'$

*<proof>*

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

*<proof>*

### 3.7 ZipWith function

**fixrec**

*zipWithStep* ::  
 $('a \rightarrow 'b \rightarrow 'c) \rightarrow$   
 $('s \rightarrow ('a, 's) Step) \rightarrow$   
 $('t \rightarrow ('b, 't) Step) \rightarrow$

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

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

**fixrec**

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

**where**

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

**lemma**  $zipWithS\text{-fix-ind-lemma}$ :

**fixes**  $P Q :: nat \Rightarrow nat \Rightarrow bool$

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

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

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

$\langle proof \rangle$

**lemma**  $zipWithS\text{-fix-ind}$ :

**assumes**  $x: x = fix.f$  **and**  $y: y = fix.g$

**assumes**  $adm\text{-}P: adm\ (\lambda x. P\ (fst\ x)\ (snd\ x))$

**assumes**  $adm\text{-}Q: adm\ (\lambda x. Q\ (fst\ x)\ (snd\ x))$

**assumes**  $P\text{-0}: \bigwedge b. P\ \perp\ b$  **and**  $P\text{-Suc}: \bigwedge a\ b. P\ a\ b \implies Q\ a\ b \implies P\ (f.a)\ b$

**assumes**  $Q\text{-0}: \bigwedge a. Q\ a\ \perp$  **and**  $Q\text{-Suc}: \bigwedge a\ b. P\ a\ b \implies Q\ a\ b \implies Q\ a\ (g.b)$

**shows**  $P\ x\ y \wedge Q\ x\ y$

$\langle proof \rangle$

**lemma**  $unfold\text{-}zipWithStep$ :

**fixes**  $f :: 'a \rightarrow 'b \rightarrow 'c$

**fixes**  $ha :: 's \rightarrow ('a, 's) \text{ Step}$

**fixes**  $hb :: 't \rightarrow ('b, 't) \text{ Step}$

**defines**  $h\text{-def}: h \equiv zipWithStep.f.ha.hb$

**shows**

$(\forall sa\ sb. sa \neq \perp \longrightarrow sb \neq \perp \longrightarrow$

$unfold.h.(sa \text{ !: } sb \text{ !: } \text{Nothing}) =$

$zipWithL.f.(unfold.ha.sa).(unfold.hb.sb)) \wedge$

$$\begin{aligned}
& (\forall sa\ sb\ a.\ sa \neq \perp \implies sb \neq \perp \implies \\
& \quad unfold \cdot h \cdot (sa \text{ !: } sb \text{ !: } Just \cdot (L \cdot a)) = \\
& \quad zipWithL \cdot f \cdot (LCons \cdot a \cdot (unfold \cdot ha \cdot sa)) \cdot (unfold \cdot hb \cdot sb)) \\
\langle proof \rangle
\end{aligned}$$

**lemma** *zipWithS-defined*:  $a \neq \perp \implies b \neq \perp \implies zipWithS \cdot f \cdot a \cdot b \neq \perp$   
 $\langle proof \rangle$

**lemma** *unstream-zipWithS*:  
 $a \neq \perp \implies b \neq \perp \implies$   
 $unstream \cdot (zipWithS \cdot f \cdot a \cdot b) = zipWithL \cdot f \cdot (unstream \cdot a) \cdot (unstream \cdot b)$   
 $\langle proof \rangle$

**lemma** *zipWithS-cong*:  
 $f = f' \implies a \approx a' \implies b \approx b' \implies$   
 $zipWithS \cdot f \cdot a \cdot b \approx zipWithS \cdot f \cdot a' \cdot b'$   
 $\langle proof \rangle$

**lemma** *zipWithL-eq*:  
 $zipWithL \cdot f \cdot xs \cdot ys = unstream \cdot (zipWithS \cdot f \cdot (stream \cdot xs) \cdot (stream \cdot ys))$   
 $\langle proof \rangle$

### 3.8 ConcatMap function

**fixrec**  
 $concatMapStep ::$   
 $(a \rightarrow (b, t) \text{ Stream}) \rightarrow$   
 $(s \rightarrow (a, s) \text{ Step}) \rightarrow$   
 $s \text{ !: } (b, t) \text{ Stream Maybe} \rightarrow$   
 $(b, s \text{ !: } (b, t) \text{ Stream Maybe}) \text{ Step}$

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

**fixpat** *concatMapStep-strict* [simp]:  $concatMapStep \cdot f \cdot ha \cdot \perp$

**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**

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

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

**lemma** *unfold-concatMapStep*:

**fixes**  $ha :: 's \rightarrow ('a, 's) \text{ Step}$

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

**defines**  $h\text{-def}: h \equiv \text{concatMapStep} \cdot f \cdot ha$

**defines**  $f'\text{-def}: f' \equiv \text{unstream} \text{ oo } f$

**shows**

$(\forall sa. sa \neq \perp \longrightarrow$

$\text{unfold} \cdot h \cdot (sa \text{ !: Nothing}) = \text{concatMapL} \cdot f' \cdot (\text{unfold} \cdot ha \cdot sa)) \wedge$

$(\forall sa \ hb \ sb. sa \neq \perp \longrightarrow sb \neq \perp \longrightarrow$

$\text{unfold} \cdot h \cdot (sa \text{ !: Just} \cdot (\text{Stream} \cdot hb \cdot sb)) =$

$\text{appendL} \cdot (\text{unfold} \cdot hb \cdot sb) \cdot (\text{concatMapL} \cdot f' \cdot (\text{unfold} \cdot ha \cdot sa)))$

$\langle \text{proof} \rangle$

**lemma** *unstream-concatMapS*:

$\text{unstream} \cdot (\text{concatMapS} \cdot f \cdot a) = \text{concatMapL} \cdot (\text{unstream} \text{ oo } f) \cdot (\text{unstream} \cdot a)$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

**lemma** *concatMapS-cong*:

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

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

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

**fixes**  $b :: ('a, 'v) \text{ Stream}$

**shows**  $(\bigwedge x. f \ x \approx g \ x) \implies a \approx b \implies \text{cont } f \implies \text{cont } g \implies$

$\text{concatMapS} \cdot (\bigwedge x. f \ x) \cdot a \approx \text{concatMapS} \cdot (\bigwedge x. g \ x) \cdot b$

$\langle \text{proof} \rangle$

**lemma** *concatMapL-eq*:

$\text{concatMapL} \cdot f \cdot xs = \text{unstream} \cdot (\text{concatMapS} \cdot (\text{stream} \text{ oo } f) \cdot (\text{stream} \cdot xs))$

$\langle \text{proof} \rangle$

### 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**

$mapL.f \circ filterL.p \circ mapL.g =$   
 $unstream \circ mapS.f \circ filterS.p \circ mapS.g \circ stream$   
{proof}

**lemma**

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

**lemma** *oo-LAM* [*simp*]:  $cont\ g \implies f \circ (\lambda x. g\ x) = (\lambda x. f \cdot (g\ x))$

{proof}

**lemma**

$concatMapL.(\lambda k.$   
 $mapL.(\lambda m. f \cdot k \cdot m) \cdot (enumFromToL.one \cdot k)) \cdot (enumFromToL.one \cdot n) =$   
 $unstream \cdot (concatMapS.(\lambda k.$   
 $mapS.(\lambda m. f \cdot k \cdot m) \cdot (enumFromToS.one \cdot k)) \cdot (enumFromToS.one \cdot n))$   
{proof}

**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.