

# Stream Fusion in HOL

Andreas Lochbihler      Alexandra Maximova

May 26, 2024

Stream Fusion is a system for removing intermediate list data structures from functional programs, in particular [Haskell](#). This entry adapts stream fusion to Isabelle/HOL and its code generator. We define stream types for finite and possibly infinite lists and stream versions for most of the fusible list functions in the theories *List* and *Coinductive-List*, and prove them correct with respect to the conversion functions between lists and streams. The Stream Fusion transformation itself is implemented as a simproc in the pre-processor of the code generator.

Brian Huffman's AFP entry [3] formalises stream fusion in HOLCF for the domain of lazy lists to prove the GHC compiler rewrite rules correct. In contrast, this work enables Isabelle's code generator to perform stream fusion itself. To that end, it covers both finite and coinductive lists from the HOL library and the Coinductive entry. The fusible list functions require specification and proof principles different from Huffman's.

# Contents

<b>1</b>	<b>Stream fusion implementation</b>	<b>4</b>
<b>2</b>	<b>Stream fusion for finite lists</b>	<b>4</b>
2.1	The type of generators for finite lists . . . . .	4
2.2	Conversion to <i>'a list</i> . . . . .	6
2.3	Producers . . . . .	7
2.3.1	Conversion to streams . . . . .	7
2.3.2	<i>replicate</i> . . . . .	7
2.3.3	<i>upt</i> . . . . .	8
2.3.4	<i>upto</i> . . . . .	8
2.3.5	<code>[]</code> . . . . .	8
2.4	Consumers . . . . .	8
2.4.1	<code>(!)</code> . . . . .	8
2.4.2	<i>length</i> . . . . .	9
2.4.3	<i>foldr</i> . . . . .	10
2.4.4	<i>foldl</i> . . . . .	10
2.4.5	<i>fold</i> . . . . .	10
2.4.6	<i>List.null</i> . . . . .	11
2.4.7	<i>hd</i> . . . . .	11
2.4.8	<i>last</i> . . . . .	12
2.4.9	<i>sum-list</i> . . . . .	12
2.4.10	<i>list-all2</i> . . . . .	12
2.4.11	<i>list-all</i> . . . . .	13
2.4.12	<i>ord.lexordp</i> . . . . .	13
2.5	Transformers . . . . .	15
2.5.1	<i>map</i> . . . . .	15
2.5.2	<i>drop</i> . . . . .	15
2.5.3	<i>dropWhile</i> . . . . .	16
2.5.4	<i>take</i> . . . . .	16
2.5.5	<i>takeWhile</i> . . . . .	17
2.5.6	<code>(@)</code> . . . . .	17
2.5.7	<i>filter</i> . . . . .	18
2.5.8	<i>zip</i> . . . . .	18
2.5.9	<i>tl</i> . . . . .	19
2.5.10	<i>butlast</i> . . . . .	19
2.5.11	<i>concat</i> . . . . .	20
2.5.12	<i>splice</i> . . . . .	21
2.5.13	<i>list-update</i> . . . . .	22
2.5.14	<i>removeAll</i> . . . . .	22
2.5.15	<i>remove1</i> . . . . .	23
2.5.16	<code>(#)</code> . . . . .	23
2.5.17	<i>List.maps</i> . . . . .	24

<b>3</b>	<b>Stream fusion for coinductive lists</b>	<b>26</b>
3.1	Conversions to <i>'a llist</i> . . . . .	27
3.1.1	Infinitely many consecutive <i>Skips</i> . . . . .	27
3.1.2	Finitely many consecutive <i>Skips</i> . . . . .	28
3.2	Producers . . . . .	29
3.2.1	Conversion to streams . . . . .	29
3.2.2	<i>iterates</i> . . . . .	29
3.2.3	<i>unfold-llist</i> . . . . .	30
3.2.4	<i>inf-llist</i> . . . . .	30
3.3	Consumers . . . . .	30
3.3.1	<i>lhd</i> . . . . .	30
3.3.2	<i>llength</i> . . . . .	31
3.3.3	<i>lnull</i> . . . . .	32
3.3.4	<i>llist-all2</i> . . . . .	32
3.3.5	<i>lnth</i> . . . . .	33
3.3.6	<i>lprefix</i> . . . . .	34
3.4	Transformers . . . . .	35
3.4.1	<i>lmap</i> . . . . .	35
3.4.2	<i>ltake</i> . . . . .	35
3.4.3	<i>ldropn</i> . . . . .	36
3.4.4	<i>ldrop</i> . . . . .	36
3.4.5	<i>ltakeWhile</i> . . . . .	36
3.4.6	<i>ldropWhile</i> . . . . .	37
3.4.7	<i>lzip</i> . . . . .	37
3.4.8	<i>lappend</i> . . . . .	38
3.4.9	<i>lfilter</i> . . . . .	39
3.4.10	<i>llist-of</i> . . . . .	39
<b>4</b>	<b>Examples and test cases for stream fusion</b>	<b>39</b>
4.1	Micro-benchmarks from Farmer et al. [2] . . . . .	40
4.2	Test stream fusion in the code generator . . . . .	40

# 1 Stream fusion implementation

**theory** *Stream-Fusion*

**imports**

*Main*

**begin**

$\langle ML \rangle$

**declare**  $[[\text{simproc del: stream-fusion}]]$

Install stream fusion as a simproc in the preprocessor for code equations

$\langle ML \rangle$

**end**

## 2 Stream fusion for finite lists

**theory** *Stream-Fusion-List*

**imports** *Stream-Fusion*

**begin**

**lemma** *map-option-mono* [*partial-function-mono*]:

$\text{mono-option } f \implies \text{mono-option } (\lambda x. \text{map-option } g (f x))$

$\langle \text{proof} \rangle$

### 2.1 The type of generators for finite lists

**datatype**  $('a, 's) \text{ step} = \text{Done} \mid \text{is-Skip: Skip } 's \mid \text{is-Yield: Yield } 'a 's$

**type-synonym**  $('a, 's) \text{ raw-generator} = 's \Rightarrow ('a, 's) \text{ step}$

Raw generators may not end in *Done*, but may lead to infinitely many *Yields* in a row. Such generators cannot be converted to finite lists, because it corresponds to an infinite list. Therefore, we introduce the type of generators that always end in *Done* after finitely many steps.

**inductive-set** *terminates-on*  $:: ('a, 's) \text{ raw-generator} \Rightarrow 's \text{ set}$

**for**  $g :: ('a, 's) \text{ raw-generator}$

**where**

$\text{stop: } g \text{ } s = \text{Done} \implies s \in \text{terminates-on } g$

$\mid \text{pause: } \llbracket g \text{ } s = \text{Skip } s'; s' \in \text{terminates-on } g \rrbracket \implies s \in \text{terminates-on } g$

$\mid \text{unfold: } \llbracket g \text{ } s = \text{Yield } a \text{ } s'; s' \in \text{terminates-on } g \rrbracket \implies s \in \text{terminates-on } g$

**definition** *terminates*  $:: ('a, 's) \text{ raw-generator} \Rightarrow \text{bool}$

**where**  $\text{terminates } g \iff (\text{terminates-on } g = \text{UNIV})$

**lemma** *terminatesI* [intro?]:  
 $(\bigwedge s. s \in \text{terminates-on } g) \implies \text{terminates } g$   
 ⟨proof⟩

**lemma** *terminatesD*:  
 $\text{terminates } g \implies s \in \text{terminates-on } g$   
 ⟨proof⟩

**lemma** *terminates-on-stop*:  
 $\text{terminates-on } (\lambda-. \text{Done}) = \text{UNIV}$   
 ⟨proof⟩

**lemma** *wf-terminates*:  
 assumes *wf R*  
 and *skip*:  $\bigwedge s s'. g s = \text{Skip } s' \implies (s', s) \in R$   
 and *yield*:  $\bigwedge s s' a. g s = \text{Yield } a s' \implies (s', s) \in R$   
 shows *terminates g*  
 ⟨proof⟩

**context** fixes  $g :: ('a, 's) \text{ raw-generator}$  **begin**

**partial-function** (*option*) *terminates-within* ::  $'s \Rightarrow \text{nat option}$  **where**  
 $\text{terminates-within } s = (\text{case } g s \text{ of}$   
   *Done*  $\Rightarrow \text{Some } 0$   
   | *Skip*  $s' \Rightarrow \text{map-option } (\lambda n. n + 1) (\text{terminates-within } s')$   
   | *Yield*  $a s' \Rightarrow \text{map-option } (\lambda n. n + 1) (\text{terminates-within } s'))$

**lemma** *terminates-on-conv-dom-terminates-within*:  
 $\text{terminates-on } g = \text{dom } \text{terminates-within}$   
 ⟨proof⟩

**end**

**lemma** *terminates-wfE*:  
 assumes *terminates g*  
 obtains *R*  
 where *wf R*  
 $\bigwedge s s'. (g s = \text{Skip } s') \implies (s', s) \in R$   
 $\bigwedge s a s'. (g s = \text{Yield } a s') \implies (s', s) \in R$   
 ⟨proof⟩

**typedef**  $('a, 's) \text{ generator} = \{g :: ('a, 's) \text{ raw-generator}. \text{terminates } g\}$   
**morphisms** *generator* *Generator*

*<proof>*

**setup-lifting** *type-definition-generator*

## 2.2 Conversion to 'a list

**context** fixes  $g :: ('a, 's)$  generator **begin**

**function** *unstream* :: 's  $\Rightarrow$  'a list

**where**

*unstream*  $s =$  (case generator  $g$   $s$  of  
  *Done*  $\Rightarrow$  []  
  | *Skip*  $s' \Rightarrow$  *unstream*  $s'$   
  | *Yield*  $x$   $s' \Rightarrow$   $x \#$  *unstream*  $s')$

*<proof>*

**termination**

*<proof>*

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

*generator*  $g$   $s =$  *Done*  $\Longrightarrow$  *unstream*  $s =$  []  
*generator*  $g$   $s =$  *Skip*  $s' \Longrightarrow$  *unstream*  $s =$  *unstream*  $s'$   
*generator*  $g$   $s =$  *Yield*  $x$   $s' \Longrightarrow$  *unstream*  $s =$   $x \#$  *unstream*  $s'$

*<proof>*

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

**function** *force* :: 's  $\Rightarrow$  ('a  $\times$  's) option

**where**

*force*  $s =$  (case generator  $g$   $s$  of *Done*  $\Rightarrow$  *None*  
  | *Skip*  $s' \Rightarrow$  *force*  $s'$   
  | *Yield*  $x$   $s' \Rightarrow$  *Some* ( $x, s')$ )

*<proof>*

**termination**

*<proof>*

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

*generator*  $g$   $s =$  *Done*  $\Longrightarrow$  *force*  $s =$  *None*  
*generator*  $g$   $s =$  *Skip*  $s' \Longrightarrow$  *force*  $s =$  *force*  $s'$   
*generator*  $g$   $s =$  *Yield*  $x$   $s' \Longrightarrow$  *force*  $s =$  *Some* ( $x, s')$

*<proof>*

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

**lemma** *unstream-force-None* [*simp*]: *force*  $s =$  *None*  $\Longrightarrow$  *unstream*  $s =$  []

*<proof>*

**lemma** *unstream-force-Some [simp]: force s = Some (x, s')  $\implies$  unstream s = x # unstream s'*

*<proof>*

**end**

*<ML>*

## 2.3 Producers

### 2.3.1 Conversion to streams

**fun** *stream-raw* :: 'a list  $\Rightarrow$  ('a, 'a list) step

**where**

*stream-raw* [] = Done

| *stream-raw* (x # xs) = Yield x xs

**lemma** *terminates-stream-raw: terminates stream-raw*

*<proof>*

**lift-definition** *stream* :: ('a, 'a list) generator **is** *stream-raw* *<proof>*

**lemma** *unstream-stream: unstream stream xs = xs*

*<proof>*

### 2.3.2 replicate

**fun** *replicate-raw* :: 'a  $\Rightarrow$  ('a, nat) raw-generator

**where**

*replicate-raw* a 0 = Done

| *replicate-raw* a (Suc n) = Yield a n

**lemma** *terminates-replicate-raw: terminates (replicate-raw a)*

*<proof>*

**lift-definition** *replicate-prod* :: 'a  $\Rightarrow$  ('a, nat) generator **is** *replicate-raw*

*<proof>*

**lemma** *unstream-replicate-prod [stream-fusion]: unstream (replicate-prod x) n = replicate n x*

*<proof>*

### 2.3.3 *upt*

**definition** *upt-raw* ::  $\text{nat} \Rightarrow (\text{nat}, \text{nat})$  *raw-generator*  
**where** *upt-raw*  $n\ m = (\text{if } m \geq n \text{ then Done else Yield } m\ (\text{Suc } m))$

**lemma** *terminates-upt-raw*: *terminates* (*upt-raw*  $n$ )  
*<proof>*

**lift-definition** *upt-prod* ::  $\text{nat} \Rightarrow (\text{nat}, \text{nat})$  *generator* **is** *upt-raw* *<proof>*

**lemma** *unstream-upt-prod* [*stream-fusion*]: *unstream* (*upt-prod*  $n$ )  $m = \text{upt } m\ n$   
*<proof>*

### 2.3.4 *upto*

**definition** *upto-raw* ::  $\text{int} \Rightarrow (\text{int}, \text{int})$  *raw-generator*  
**where** *upto-raw*  $n\ m = (\text{if } m \leq n \text{ then Yield } m\ (m + 1) \text{ else Done})$

**lemma** *terminates-upto-raw*: *terminates* (*upto-raw*  $n$ )  
*<proof>*

**lift-definition** *upto-prod* ::  $\text{int} \Rightarrow (\text{int}, \text{int})$  *generator* **is** *upto-raw* *<proof>*

**lemma** *unstream-upto-prod* [*stream-fusion*]: *unstream* (*upto-prod*  $n$ )  $m = \text{upto } m\ n$   
*<proof>*

### 2.3.5 []

**lift-definition** *Nil-prod* ::  $(\text{'a}, \text{unit})$  *generator* **is**  $\lambda\-. \text{Done}$   
*<proof>*

**lemma** *generator-Nil-prod*: *generator* *Nil-prod* =  $(\lambda\-. \text{Done})$   
*<proof>*

**lemma** *unstream-Nil-prod* [*stream-fusion*]: *unstream* *Nil-prod*  $() = []$   
*<proof>*

## 2.4 Consumers

### 2.4.1 (!)

**context** *fixes*  $g :: (\text{'a}, \text{'s})$  *generator* **begin**

**definition** *nth-cons* ::  $\text{'s} \Rightarrow \text{nat} \Rightarrow \text{'a}$   
**where** [*stream-fusion*]: *nth-cons*  $s\ n = \text{unstream } g\ s\ !\ n$



**lemma** *nth-cons-code* [code]:  
 $nth-cons\ s\ n =$   
 (case generator  $g\ s$  of *Done* => *undefined*  $n$   
 | *Skip*  $s' => nth-cons\ s'\ n$   
 | *Yield*  $x\ s' => (case\ n\ of\ 0 => x\ |\ Suc\ n' => nth-cons\ s'\ n')$ )  
 <proof>

**end**

### 2.4.2 length

**context** fixes  $g :: ('a, 's) generator$  **begin**

**definition** *length-cons* ::  $'s \Rightarrow nat$   
**where** *length-cons*  $s = length\ (unstream\ g\ s)$

**lemma** *length-cons-code* [code]:  
 $length-cons\ s =$   
 (case generator  $g\ s$  of  
   *Done*  $\Rightarrow 0$   
 | *Skip*  $s' \Rightarrow length-cons\ s'$   
 | *Yield*  $a\ s' \Rightarrow 1 + length-cons\ s'$ )  
 <proof>

**definition** *gen-length-cons* ::  $nat \Rightarrow 's \Rightarrow nat$   
**where** *gen-length-cons*  $n\ s = n + length\ (unstream\ g\ s)$

**lemma** *gen-length-cons-code* [code]:  
 $gen-length-cons\ n\ s = (case\ generator\ g\ s\ of$   
   *Done*  $\Rightarrow n\ |$  *Skip*  $s' \Rightarrow gen-length-cons\ n\ s'\ |$  *Yield*  $a\ s' \Rightarrow gen-length-cons\ (Suc\ n)$   
 $s')$   
 <proof>

**lemma** *unstream-gen-length* [stream-fusion]:  $gen-length-cons\ 0\ s = length\ (unstream\ g\ s)$   
 <proof>

**lemma** *unstream-gen-length2* [stream-fusion]:  $gen-length-cons\ n\ s = List.gen-length\ n\ (unstream\ g\ s)$   
 <proof>

**end**

### 2.4.3 *foldr*

**context**

**fixes**  $g :: ('a, 's)$  generator  
**and**  $f :: 'a \Rightarrow 'b \Rightarrow 'b$   
**and**  $z :: 'b$

**begin**

**definition**  $foldr-cons :: 's \Rightarrow 'b$

**where** [*stream-fusion*]:  $foldr-cons\ s = foldr\ f\ (unstream\ g\ s)\ z$

**lemma**  $foldr-cons-code$  [*code*]:

$foldr-cons\ s =$   
  (case generator  $g\ s$  of  
    Done  $\Rightarrow z$   
    | Skip  $s' \Rightarrow foldr-cons\ s'$   
    | Yield  $a\ s' \Rightarrow f\ a\ (foldr-cons\ s')$ )  
<proof>

**end**

### 2.4.4 *foldl*

**context**

**fixes**  $g :: ('b, 's)$  generator  
**and**  $f :: 'a \Rightarrow 'b \Rightarrow 'a$

**begin**

**definition**  $foldl-cons :: 'a \Rightarrow 's \Rightarrow 'a$

**where** [*stream-fusion*]:  $foldl-cons\ z\ s = foldl\ f\ z\ (unstream\ g\ s)$

**lemma**  $foldl-cons-code$  [*code*]:

$foldl-cons\ z\ s =$   
  (case generator  $g\ s$  of  
    Done  $\Rightarrow z$   
    | Skip  $s' \Rightarrow foldl-cons\ z\ s'$   
    | Yield  $a\ s' \Rightarrow foldl-cons\ (f\ z\ a)\ s')$   
<proof>

**end**

### 2.4.5 *fold*

**context**

**fixes**  $g :: ('a, 's)$  generator

**and**  $f :: 'a \Rightarrow 'b \Rightarrow 'b$   
**begin**

**definition**  $fold-cons :: 'b \Rightarrow 's \Rightarrow 'b$   
**where** [*stream-fusion*]:  $fold-cons\ z\ s = fold\ f\ (unstream\ g\ s)\ z$

**lemma**  $fold-cons-code$  [*code*]:  
 $fold-cons\ z\ s =$   
  (*case generator g s of*  
    *Done*  $\Rightarrow z$   
    | *Skip s'*  $\Rightarrow fold-cons\ z\ s'$   
    | *Yield a s'*  $\Rightarrow fold-cons\ (f\ a\ z)\ s'$ )  
<*proof*>

**end**

#### 2.4.6 *List.null*

**definition**  $null-cons :: ('a, 's)\ generator \Rightarrow 's \Rightarrow bool$   
**where** [*stream-fusion*]:  $null-cons\ g\ s = List.null\ (unstream\ g\ s)$

**lemma**  $null-cons-code$  [*code*]:  
 $null-cons\ g\ s = (case\ generator\ g\ s\ of\ Done\ \Rightarrow\ True\ |\ Skip\ s'\ \Rightarrow\ null-cons\ g\ s'\ |\ Yield$   
 $- -\ \Rightarrow\ False)$   
<*proof*>

#### 2.4.7 *hd*

**context fixes**  $g :: ('a, 's)\ generator$  **begin**

**definition**  $hd-cons :: 's \Rightarrow 'a$   
**where** [*stream-fusion*]:  $hd-cons\ s = hd\ (unstream\ g\ s)$

**lemma**  $hd-cons-code$  [*code*]:  
 $hd-cons\ s =$   
  (*case generator g s of*  
    *Done*  $\Rightarrow undefined$   
    | *Skip s'*  $\Rightarrow hd-cons\ s'$   
    | *Yield a s'*  $\Rightarrow a$ )  
<*proof*>

**end**

### 2.4.8 *last*

**context** fixes  $g :: ('a, 's)$  generator **begin**

**definition**  $last-cons :: 'a$  option  $\Rightarrow 's \Rightarrow 'a$

**where**  $last-cons\ x\ s =$  (if unstream  $g\ s = []$  then the  $x$  else  $last\ (unstream\ g\ s)$ )

**lemma**  $last-cons-code$  [code]:

$last-cons\ x\ s =$   
(case generator  $g\ s$  of  $Done \Rightarrow$  the  $x$   
|  $Skip\ s' \Rightarrow last-cons\ x\ s'$   
|  $Yield\ a\ s' \Rightarrow last-cons\ (Some\ a)\ s'$ )

$\langle proof \rangle$

**lemma**  $unstream-last-cons$  [stream-fusion]:  $last-cons\ None\ s = last\ (unstream\ g\ s)$

$\langle proof \rangle$

**end**

### 2.4.9 *sum-list*

**context** fixes  $g :: ('a :: monoid-add, 's)$  generator **begin**

**definition**  $sum-list-cons :: 's \Rightarrow 'a$

**where** [stream-fusion]:  $sum-list-cons\ s = sum-list\ (unstream\ g\ s)$

**lemma**  $sum-list-cons-code$  [code]:

$sum-list-cons\ s =$   
(case generator  $g\ s$  of  
   $Done \Rightarrow 0$   
  |  $Skip\ s' \Rightarrow sum-list-cons\ s'$   
  |  $Yield\ a\ s' \Rightarrow a + sum-list-cons\ s'$ )

$\langle proof \rangle$

**end**

### 2.4.10 *list-all2*

**context**

  fixes  $g :: ('a, 's1)$  generator

  and  $h :: ('b, 's2)$  generator

  and  $P :: 'a \Rightarrow 'b \Rightarrow bool$

**begin**

**definition**  $list-all2-cons :: 's1 \Rightarrow 's2 \Rightarrow bool$

**where** [*stream-fusion*]:  $list\text{-}all2\text{-}cons\ sg\ sh = list\text{-}all2\ P\ (unstream\ g\ sg)\ (unstream\ h\ sh)$

**definition**  $list\text{-}all2\text{-}cons1 :: 'a \Rightarrow 's1 \Rightarrow 's2 \Rightarrow bool$

**where**  $list\text{-}all2\text{-}cons1\ x\ sg'\ sh = list\text{-}all2\ P\ (x \# unstream\ g\ sg')\ (unstream\ h\ sh)$

**lemma**  $list\text{-}all2\text{-}cons\text{-}code$  [*code*]:

```
list-all2-cons sg sh =
  (case generator g sg of
    Done  $\Rightarrow$  null-cons h sh
  | Skip sg'  $\Rightarrow$  list-all2-cons sg' sh
  | Yield a sg'  $\Rightarrow$  list-all2-cons1 a sg' sh)
⟨proof⟩
```

**lemma**  $list\text{-}all2\text{-}cons1\text{-}code$  [*code*]:

```
list-all2-cons1 x sg' sh =
  (case generator h sh of
    Done  $\Rightarrow$  False
  | Skip sh'  $\Rightarrow$  list-all2-cons1 x sg' sh'
  | Yield y sh'  $\Rightarrow$  P x y  $\wedge$  list-all2-cons sg' sh')
⟨proof⟩
```

**end**

#### 2.4.11 *list-all*

**context**

**fixes**  $g :: ('a, 's)\ generator$

**and**  $P :: 'a \Rightarrow bool$

**begin**

**definition**  $list\text{-}all\text{-}cons :: 's \Rightarrow bool$

**where** [*stream-fusion*]:  $list\text{-}all\text{-}cons\ s = list\text{-}all\ P\ (unstream\ g\ s)$

**lemma**  $list\text{-}all\text{-}cons\text{-}code$  [*code*]:

```
list-all-cons s  $\longleftrightarrow$ 
  (case generator g s of
    Done  $\Rightarrow$  True | Skip s'  $\Rightarrow$  list-all-cons s' | Yield x s'  $\Rightarrow$  P x  $\wedge$  list-all-cons s')
⟨proof⟩
```

**end**

#### 2.4.12 *ord.lexordp*

**context** *ord* **begin**

**definition** *lexord-fusion* :: ('a, 's1) generator ⇒ ('a, 's2) generator ⇒ 's1 ⇒ 's2 ⇒ bool

**where** [code del]: *lexord-fusion* g1 g2 s1 s2 = ord-class.lexordp (unstream g1 s1) (unstream g2 s2)

**definition** *lexord-eq-fusion* :: ('a, 's1) generator ⇒ ('a, 's2) generator ⇒ 's1 ⇒ 's2 ⇒ bool

**where** [code del]: *lexord-eq-fusion* g1 g2 s1 s2 = lexordp-eq (unstream g1 s1) (unstream g2 s2)

**lemma** *lexord-fusion-code*:

*lexord-fusion* g1 g2 s1 s2 ⟷  
 (case generator g1 s1 of  
   Done ⇒ ¬ null-cons g2 s2  
   | Skip s1' ⇒ *lexord-fusion* g1 g2 s1' s2  
   | Yield x s1' ⇒  
     (case force g2 s2 of  
       None ⇒ False  
       | Some (y, s2') ⇒ x < y ∨ ¬ y < x ∧ *lexord-fusion* g1 g2 s1' s2'))  
 ⟨proof⟩

**lemma** *lexord-eq-fusion-code*:

*lexord-eq-fusion* g1 g2 s1 s2 ⟷  
 (case generator g1 s1 of  
   Done ⇒ True  
   | Skip s1' ⇒ *lexord-eq-fusion* g1 g2 s1' s2  
   | Yield x s1' ⇒  
     (case force g2 s2 of  
       None ⇒ False  
       | Some (y, s2') ⇒ x < y ∨ ¬ y < x ∧ *lexord-eq-fusion* g1 g2 s1' s2'))  
 ⟨proof⟩

**end**

**lemmas** [code] =

*lexord-fusion-code* ord.*lexord-fusion-code*  
*lexord-eq-fusion-code* ord.*lexord-eq-fusion-code*

**lemmas** [stream-fusion] =

*lexord-fusion-def* ord.*lexord-fusion-def*  
*lexord-eq-fusion-def* ord.*lexord-eq-fusion-def*

## 2.5 Transformers

### 2.5.1 map

**definition**  $map\text{-}raw :: ('a \Rightarrow 'b) \Rightarrow ('a, 's) \text{ raw-generator} \Rightarrow ('b, 's) \text{ raw-generator}$   
**where**

$map\text{-}raw\ f\ g\ s = (\text{case } g\ s\ \text{of}$   
     $Done \Rightarrow Done$   
     $| Skip\ s' \Rightarrow Skip\ s'$   
     $| Yield\ a\ s' \Rightarrow Yield\ (f\ a)\ s')$

**lemma**  $terminates\text{-}map\text{-}raw$ :

**assumes**  $terminates\ g$   
    **shows**  $terminates\ (map\text{-}raw\ f\ g)$

$\langle proof \rangle$

**lift-definition**  $map\text{-}trans :: ('a \Rightarrow 'b) \Rightarrow ('a, 's) \text{ generator} \Rightarrow ('b, 's) \text{ generator}$  **is**  $map\text{-}raw$

$\langle proof \rangle$

**lemma**  $unstream\text{-}map\text{-}trans$  [ $stream\text{-}fusion$ ]:  $unstream\ (map\text{-}trans\ f\ g)\ s = map\ f\ (unstream\ g\ s)$

$\langle proof \rangle$

### 2.5.2 drop

**fun**  $drop\text{-}raw :: ('a, 's) \text{ raw-generator} \Rightarrow ('a, (nat \times 's)) \text{ raw-generator}$

**where**

$drop\text{-}raw\ g\ (n, s) = (\text{case } g\ s\ \text{of}$   
     $Done \Rightarrow Done \mid Skip\ s' \Rightarrow Skip\ (n, s')$   
     $| Yield\ a\ s' \Rightarrow (\text{case } n\ \text{of } 0 \Rightarrow Yield\ a\ (0, s') \mid Suc\ n \Rightarrow Skip\ (n, s')))$

**lemma**  $terminates\text{-}drop\text{-}raw$ :

**assumes**  $terminates\ g$   
    **shows**  $terminates\ (drop\text{-}raw\ g)$

$\langle proof \rangle$

**lift-definition**  $drop\text{-}trans :: ('a, 's) \text{ generator} \Rightarrow ('a, nat \times 's) \text{ generator}$  **is**  $drop\text{-}raw$

$\langle proof \rangle$

**lemma**  $unstream\text{-}drop\text{-}trans$  [ $stream\text{-}fusion$ ]:  $unstream\ (drop\text{-}trans\ g)\ (n, s) = drop\ n\ (unstream\ g\ s)$

$\langle proof \rangle$

### 2.5.3 *dropWhile*

**fun** *dropWhile-raw* :: ('a ⇒ bool) ⇒ ('a, 's) raw-generator ⇒ ('a, bool × 's) raw-generator

— Boolean flag indicates whether we are still in dropping phase

**where**

*dropWhile-raw* *P g* (*True*, *s*) = (case *g s* of  
  *Done* ⇒ *Done* | *Skip s'* ⇒ *Skip (True, s')*  
  | *Yield a s'* ⇒ (if *P a* then *Skip (True, s')* else *Yield a (False, s')*))  
| *dropWhile-raw P g* (*False*, *s*) = (case *g s* of  
  *Done* ⇒ *Done* | *Skip s'* ⇒ *Skip (False, s')* | *Yield a s'* ⇒ *Yield a (False, s')*)

**lemma** *terminates-dropWhile-raw*:

**assumes** *terminates g*

**shows** *terminates (dropWhile-raw P g)*

⟨*proof*⟩

**lift-definition** *dropWhile-trans* :: ('a ⇒ bool) ⇒ ('a, 's) generator ⇒ ('a, bool × 's) generator

**is** *dropWhile-raw* ⟨*proof*⟩

**lemma** *unstream-dropWhile-trans-False*:

*unstream (dropWhile-trans P g) (False, s)* = *unstream g s*

⟨*proof*⟩

**lemma** *unstream-dropWhile-trans [stream-fusion]*:

*unstream (dropWhile-trans P g) (True, s)* = *dropWhile P (unstream g s)*

⟨*proof*⟩

### 2.5.4 *take*

**fun** *take-raw* :: ('a, 's) raw-generator ⇒ ('a, (nat × 's)) raw-generator

**where**

*take-raw g* (*0*, *s*) = *Done*  
| *take-raw g* (*Suc n*, *s*) = (case *g s* of  
  *Done* ⇒ *Done* | *Skip s'* ⇒ *Skip (Suc n, s')* | *Yield a s'* ⇒ *Yield a (n, s')*)

**lemma** *terminates-take-raw*:

**assumes** *terminates g*

**shows** *terminates (take-raw g)*

⟨*proof*⟩

**lift-definition** *take-trans* :: ('a, 's) generator ⇒ ('a, nat × 's) generator **is** *take-raw*

⟨*proof*⟩



**lemma** *unstream-take-trans* [*stream-fusion*]:  $\text{unstream } (\text{take-trans } g) (n, s) = \text{take } n (\text{unstream } g s)$   
 ⟨*proof*⟩

### 2.5.5 *take While*

**definition** *take While-raw* ::  $('a \Rightarrow \text{bool}) \Rightarrow ('a, 's) \text{ raw-generator} \Rightarrow ('a, 's) \text{ raw-generator}$   
**where**

*take While-raw*  $P g s = (\text{case } g s \text{ of}$   
 $\text{Done} \Rightarrow \text{Done} \mid \text{Skip } s' \Rightarrow \text{Skip } s' \mid \text{Yield } a s' \Rightarrow \text{if } P a \text{ then Yield } a s' \text{ else Done})$

**lemma** *terminates-take While-raw*:

**assumes** *terminates*  $g$

**shows** *terminates*  $(\text{take While-raw } P g)$

⟨*proof*⟩

**lift-definition** *take While-trans* ::  $('a \Rightarrow \text{bool}) \Rightarrow ('a, 's) \text{ generator} \Rightarrow ('a, 's) \text{ generator}$   
**is** *take While-raw* ⟨*proof*⟩

**lemma** *unstream-take While-trans* [*stream-fusion*]:

$\text{unstream } (\text{take While-trans } P g) s = \text{take While } P (\text{unstream } g s)$

⟨*proof*⟩

### 2.5.6 (@)

**fun** *append-raw* ::  $('a, 'sg) \text{ raw-generator} \Rightarrow ('a, 'sh) \text{ raw-generator} \Rightarrow 'sh \Rightarrow ('a, 'sg + 'sh) \text{ raw-generator}$

**where**

*append-raw*  $g h sh\text{-start} (\text{Inl } sg) = (\text{case } g sg \text{ of}$   
 $\text{Done} \Rightarrow \text{Skip } (\text{Inr } sh\text{-start}) \mid \text{Skip } sg' \Rightarrow \text{Skip } (\text{Inl } sg') \mid \text{Yield } a sg' \Rightarrow \text{Yield } a (\text{Inl } sg'))$

$\mid \text{append-raw } g h sh\text{-start} (\text{Inr } sh) = (\text{case } h sh \text{ of}$

$\text{Done} \Rightarrow \text{Done} \mid \text{Skip } sh' \Rightarrow \text{Skip } (\text{Inr } sh') \mid \text{Yield } a sh' \Rightarrow \text{Yield } a (\text{Inr } sh'))$

**lemma** *terminates-on-append-raw-Inr*:

**assumes** *terminates*  $h$

**shows**  $\text{Inr } sh \in \text{terminates-on } (\text{append-raw } g h sh\text{-start})$

⟨*proof*⟩

**lemma** *terminates-append-raw*:

**assumes** *terminates*  $g$  *terminates*  $h$

**shows** *terminates*  $(\text{append-raw } g h sh\text{-start})$

⟨*proof*⟩

**lift-definition** *append-trans* :: ('a, 'sg) generator ⇒ ('a, 'sh) generator ⇒ 'sh ⇒ ('a, 'sg + 'sh) generator  
**is** *append-raw* ⟨proof⟩

**lemma** *unstream-append-trans-Inr*: *unstream* (*append-trans* *g h sh*) (*Inr sh'*) = *unstream* *h sh'*  
 ⟨proof⟩

**lemma** *unstream-append-trans* [*stream-fusion*]:  
*unstream* (*append-trans* *g h sh*) (*Inl sg*) = *append* (*unstream* *g sg*) (*unstream* *h sh*)  
 ⟨proof⟩

### 2.5.7 filter

**definition** *filter-raw* :: ('a ⇒ bool) ⇒ ('a, 's) raw-generator ⇒ ('a, 's) raw-generator  
**where**

*filter-raw* *P g s* = (case *g s* of  
 Done ⇒ Done | Skip *s'* ⇒ Skip *s'* | Yield *a s'* ⇒ if *P a* then Yield *a s'* else Skip *s'*)

**lemma** *terminates-filter-raw*:  
**assumes** *terminates* *g*  
**shows** *terminates* (*filter-raw* *P g*)  
 ⟨proof⟩

**lift-definition** *filter-trans* :: ('a ⇒ bool) ⇒ ('a, 's) generator ⇒ ('a, 's) generator  
**is** *filter-raw* ⟨proof⟩

**lemma** *unstream-filter-trans* [*stream-fusion*]: *unstream* (*filter-trans* *P g*) *s* = *filter* *P* (*unstream* *g s*)  
 ⟨proof⟩

### 2.5.8 zip

**fun** *zip-raw* :: ('a, 'sg) raw-generator ⇒ ('b, 'sh) raw-generator ⇒ ('a × 'b, 'sg × 'sh × 'a option) raw-generator

— We search first the left list for the next element and cache it in the 'a option part of the state once we found one

**where**

*zip-raw* *g h* (*sg, sh, None*) = (case *g sg* of  
 Done ⇒ Done | Skip *sg'* ⇒ Skip (*sg', sh, None*) | Yield *a sg'* ⇒ Skip (*sg', sh, Some a*))  
 | *zip-raw* *g h* (*sg, sh, Some a*) = (case *h sh* of  
 Done ⇒ Done | Skip *sh'* ⇒ Skip (*sg, sh', Some a*) | Yield *b sh'* ⇒ Yield (*a, b*) (*sg, sh', None*))

**lemma** *terminates-zip-raw*:

**assumes** *terminates g terminates h*

**shows** *terminates (zip-raw g h)*

*<proof>*

**lift-definition** *zip-trans* ::  $('a, 'sg)$  generator  $\Rightarrow$   $('b, 'sh)$  generator  $\Rightarrow$   $('a \times 'b, 'sg \times 'sh \times 'a \text{ option})$  generator

**is** *zip-raw* *<proof>*

**lemma** *unstream-zip-trans* [*stream-fusion*]:

*unstream (zip-trans g h) (sg, sh, None) = zip (unstream g sg) (unstream h sh)*

*<proof>*

### 2.5.9 *tl*

**fun** *tl-raw* ::  $('a, 'sg)$  raw-generator  $\Rightarrow$   $('a, \text{bool} \times 'sg)$  raw-generator

— The Boolean flag stores whether we have already skipped the first element

**where**

*tl-raw g (False, sg) = (case g sg of*

*Done  $\Rightarrow$  Done | Skip sg'  $\Rightarrow$  Skip (False, sg') | Yield a sg'  $\Rightarrow$  Skip (True, sg')*

*| tl-raw g (True, sg) = (case g sg of*

*Done  $\Rightarrow$  Done | Skip sg'  $\Rightarrow$  Skip (True, sg') | Yield a sg'  $\Rightarrow$  Yield a (True, sg')*)

**lemma** *terminates-tl-raw*:

**assumes** *terminates g*

**shows** *terminates (tl-raw g)*

*<proof>*

**lift-definition** *tl-trans* ::  $('a, 'sg)$  generator  $\Rightarrow$   $('a, \text{bool} \times 'sg)$  generator

**is** *tl-raw* *<proof>*

**lemma** *unstream-tl-trans-True*: *unstream (tl-trans g) (True, s) = unstream g s*

*<proof>*

**lemma** *unstream-tl-trans* [*stream-fusion*]: *unstream (tl-trans g) (False, s) = tl (unstream g s)*

*<proof>*

### 2.5.10 *butlast*

**fun** *butlast-raw* ::  $('a, 's)$  raw-generator  $\Rightarrow$   $('a, 'a \text{ option} \times 's)$  raw-generator

— The *'a option* caches the previous element we have seen

**where**

$butlast\text{-}raw\ g\ (None, s) = (case\ g\ s\ of$   
 $\quad Done \Rightarrow Done \mid Skip\ s' \Rightarrow Skip\ (None, s') \mid Yield\ a\ s' \Rightarrow Skip\ (Some\ a, s'))$   
 $|\ butlast\text{-}raw\ g\ (Some\ b, s) = (case\ g\ s\ of$   
 $\quad Done \Rightarrow Done \mid Skip\ s' \Rightarrow Skip\ (Some\ b, s') \mid Yield\ a\ s' \Rightarrow Yield\ b\ (Some\ a, s'))$

**lemma** *terminates-butlast-raw*:

**assumes** *terminates g*  
**shows** *terminates (butlast-raw g)*

*<proof>*

**lift-definition** *butlast-trans* :: (*'a, 's*) generator  $\Rightarrow$  (*'a, 'a option  $\times$  's*) generator  
**is** *butlast-raw* *<proof>*

**lemma** *unstream-butlast-trans-Some*:

*unstream (butlast-trans g) (Some b, s) = butlast (b # (unstream g s))*

*<proof>*

**lemma** *unstream-butlast-trans [stream-fusion]*:

*unstream (butlast-trans g) (None, s) = butlast (unstream g s)*

*<proof>*

### 2.5.11 *concat*

We only do the easy version here where the generator has type (*'a list, 's*) generator, not (*'a, 's*) generator, *'s*) generator

**fun** *concat-raw* :: (*'a list, 's*) raw-generator  $\Rightarrow$  (*'a, 'a list  $\times$  's*) raw-generator

**where**

*concat-raw g ([], s) = (case g s of*  
 $\quad Done \Rightarrow Done \mid Skip\ s' \Rightarrow Skip\ ([], s') \mid Yield\ xs\ s' \Rightarrow Skip\ (xs, s'))$

$|\ concat\text{-}raw\ g\ (x\ \#\ xs, s) = Yield\ x\ (xs, s)$

**lemma** *terminates-concat-raw*:

**assumes** *terminates g*  
**shows** *terminates (concat-raw g)*

*<proof>*

**lift-definition** *concat-trans* :: (*'a list, 's*) generator  $\Rightarrow$  (*'a, 'a list  $\times$  's*) generator

**is** *concat-raw* *<proof>*

**lemma** *unstream-concat-trans-gen*: *unstream (concat-trans g) (xs, s) = xs @ (concat (unstream g s))*

*<proof>*

**lemma** *unstream-concat-trans* [*stream-fusion*]:  
*unstream* (*concat-trans* *g*) ( $\square$ , *s*) = *concat* (*unstream* *g* *s*)  
 ⟨*proof*⟩

### 2.5.12 *splice*

**datatype** (*'a*, *'b*) *splice-state* = *Left* *'a* *'b* | *Right* *'a* *'b* | *Left-only* *'a* | *Right-only* *'b*

**fun** *splice-raw* :: (*'a*, *'sg*) *raw-generator*  $\Rightarrow$  (*'a*, *'sh*) *raw-generator*  $\Rightarrow$  (*'a*, (*'sg*, *'sh*) *splice-state*) *raw-generator*

**where**

*splice-raw* *g* *h* (*Left-only* *sg*) = (case *g* *sg* of  
   *Done*  $\Rightarrow$  *Done* | *Skip* *sg'*  $\Rightarrow$  *Skip* (*Left-only* *sg'*) | *Yield* *a* *sg'*  $\Rightarrow$  *Yield* *a* (*Left-only* *sg'*))  
 | *splice-raw* *g* *h* (*Left* *sg* *sh*) = (case *g* *sg* of  
   *Done*  $\Rightarrow$  *Skip* (*Right-only* *sh*) | *Skip* *sg'*  $\Rightarrow$  *Skip* (*Left* *sg'* *sh*) | *Yield* *a* *sg'*  $\Rightarrow$  *Yield* *a* (*Right* *sg'* *sh*))  
 | *splice-raw* *g* *h* (*Right-only* *sh*) = (case *h* *sh* of  
   *Done*  $\Rightarrow$  *Done* | *Skip* *sh'*  $\Rightarrow$  *Skip* (*Right-only* *sh'*) | *Yield* *a* *sh'*  $\Rightarrow$  *Yield* *a* (*Right-only* *sh'*))  
 | *splice-raw* *g* *h* (*Right* *sg* *sh*) = (case *h* *sh* of  
   *Done*  $\Rightarrow$  *Skip* (*Left-only* *sg*) | *Skip* *sh'*  $\Rightarrow$  *Skip* (*Right* *sg* *sh'*) | *Yield* *a* *sh'*  $\Rightarrow$  *Yield* *a* (*Left* *sg* *sh'*))

**lemma** *terminates-splice-raw*:

**assumes** *g*: *terminates* *g* **and** *h*: *terminates* *h*  
**shows** *terminates* (*splice-raw* *g* *h*)  
 ⟨*proof*⟩

**lift-definition** *splice-trans* :: (*'a*, *'sg*) *generator*  $\Rightarrow$  (*'a*, *'sh*) *generator*  $\Rightarrow$  (*'a*, (*'sg*, *'sh*) *splice-state*) *generator*  
**is** *splice-raw* ⟨*proof*⟩

**lemma** *unstream-splice-trans-Right-only*: *unstream* (*splice-trans* *g* *h*) (*Right-only* *sh*) = *unstream* *h* *sh*  
 ⟨*proof*⟩

**lemma** *unstream-splice-trans-Left-only*: *unstream* (*splice-trans* *g* *h*) (*Left-only* *sg*) = *unstream* *g* *sg*  
 ⟨*proof*⟩

**lemma** *unstream-splice-trans* [*stream-fusion*]:

*unstream* (*splice-trans* *g* *h*) (*Left* *sg* *sh*) = *splice* (*unstream* *g* *sg*) (*unstream* *h* *sh*)  
 ⟨*proof*⟩

### 2.5.13 list-update

**fun** *list-update-raw* :: ('a,'s) raw-generator  $\Rightarrow$  'a  $\Rightarrow$  ('a, nat  $\times$  's) raw-generator

**where**

*list-update-raw* *g b* (*n, s*) = (case *g s* of  
  *Done*  $\Rightarrow$  *Done* | *Skip s'*  $\Rightarrow$  *Skip* (*n, s'*)  
  | *Yield a s'*  $\Rightarrow$  if *n* = 0 then *Yield a* (*0, s'*)  
                  else if *n* = 1 then *Yield b* (*0, s'*)  
                  else *Yield a* (*n - 1, s'*)

**lemma** *terminates-list-update-raw*:

**assumes** *terminates g*

**shows** *terminates* (*list-update-raw g b*)

*<proof>*

**lift-definition** *list-update-trans* :: ('a,'s) generator  $\Rightarrow$  'a  $\Rightarrow$  ('a, nat  $\times$  's) generator

**is** *list-update-raw* *<proof>*

**lemma** *unstream-lift-update-trans-None*: *unstream* (*list-update-trans g b*) (*0, s*) = *unstream g s*

*<proof>*

**lemma** *unstream-list-update-trans [stream-fusion]*:

*unstream* (*list-update-trans g b*) (*Suc n, s*) = *list-update* (*unstream g s*) *n b*

*<proof>*

### 2.5.14 removeAll

**definition** *removeAll-raw* :: 'a  $\Rightarrow$  ('a, 's) raw-generator  $\Rightarrow$  ('a, 's) raw-generator

**where**

*removeAll-raw b g s* = (case *g s* of  
  *Done*  $\Rightarrow$  *Done* | *Skip s'*  $\Rightarrow$  *Skip s'* | *Yield a s'*  $\Rightarrow$  if *a* = *b* then *Skip s'* else *Yield a s'*)

**lemma** *terminates-removeAll-raw*:

**assumes** *terminates g*

**shows** *terminates* (*removeAll-raw b g*)

*<proof>*

**lift-definition** *removeAll-trans* :: 'a  $\Rightarrow$  ('a, 's) generator  $\Rightarrow$  ('a, 's) generator

**is** *removeAll-raw* *<proof>*

**lemma** *unstream-removeAll-trans [stream-fusion]*:

*unstream* (*removeAll-trans b g*) *s* = *removeAll b* (*unstream g s*)

*<proof>*

### 2.5.15 *remove1*

**fun** *remove1-raw* :: 'a ⇒ ('a, 's) raw-generator ⇒ ('a, bool × 's) raw-generator

**where**

*remove1-raw* x g (b, s) = (case g s of  
  Done ⇒ Done | Skip s' ⇒ Skip (b, s')  
  | Yield y s' ⇒ if b ∧ x = y then Skip (False, s') else Yield y (b, s'))

**lemma** *terminates-remove1-raw*:

**assumes** *terminates* g

**shows** *terminates* (*remove1-raw* b g)

⟨*proof*⟩

**lift-definition** *remove1-trans* :: 'a ⇒ ('a, 's) generator ⇒ ('a, bool × 's) generator

**is** *remove1-raw* ⟨*proof*⟩

**lemma** *unstream-remove1-trans-False*: *unstream* (*remove1-trans* b g) (False, s) = *unstream* g s

⟨*proof*⟩

**lemma** *unstream-remove1-trans [stream-fusion]*:

*unstream* (*remove1-trans* b g) (True, s) = *remove1* b (*unstream* g s)

⟨*proof*⟩

### 2.5.16 (#)

**fun** *Cons-raw* :: 'a ⇒ ('a, 's) raw-generator ⇒ ('a, bool × 's) raw-generator

**where**

*Cons-raw* x g (b, s) = (if b then Yield x (False, s) else case g s of  
  Done ⇒ Done | Skip s' ⇒ Skip (False, s') | Yield y s' ⇒ Yield y (False, s'))

**lemma** *terminates-Cons-raw*:

**assumes** *terminates* g

**shows** *terminates* (*Cons-raw* x g)

⟨*proof*⟩

**lift-definition** *Cons-trans* :: 'a ⇒ ('a, 's) generator ⇒ ('a, bool × 's) generator

**is** *Cons-raw* ⟨*proof*⟩

**lemma** *unstream-Cons-trans-False*: *unstream* (*Cons-trans* x g) (False, s) = *unstream* g s

⟨*proof*⟩

We do not declare *Cons-trans* as a transformer. Otherwise, literal lists would be transformed into streams which adds a significant overhead to the stream state.

**lemma** *unstream-Cons-trans*:  $unstream (Cons-trans\ x\ g) (True, s) = x \# unstream\ g\ s$   
 ⟨proof⟩

### 2.5.17 *List.maps*

Stream version based on Coutts [1].

We restrict the function for generating the inner lists to terminating generators because the code generator does not directly supported nesting abstract datatypes in other types.

**fun** *maps-raw*

$:: ('a \Rightarrow ('b, 'sg)\ generator \times 'sg) \Rightarrow ('a, 's)\ raw-generator$   
 $\Rightarrow ('b, 's \times (('b, 'sg)\ generator \times 'sg)\ option)\ raw-generator$

**where**

$maps-raw\ f\ g\ (s, None) = (case\ g\ s\ of$   
 $\quad Done \Rightarrow Done \mid Skip\ s' \Rightarrow Skip\ (s', None) \mid Yield\ x\ s' \Rightarrow Skip\ (s', Some\ (f\ x)))$   
 $\mid maps-raw\ f\ g\ (s, Some\ (g'', s'')) = (case\ generator\ g''\ s''\ of$   
 $\quad Done \Rightarrow Skip\ (s, None) \mid Skip\ s' \Rightarrow Skip\ (s, Some\ (g'', s')) \mid Yield\ x\ s' \Rightarrow Yield\ x$   
 $(s, Some\ (g'', s')))$

**lemma** *terminates-on-maps-raw-Some*:

**assumes**  $(s, None) \in terminates-on\ (maps-raw\ f\ g)$   
**shows**  $(s, Some\ (g'', s'')) \in terminates-on\ (maps-raw\ f\ g)$

⟨proof⟩

**lemma** *terminates-maps-raw*:

**assumes** *terminates*  $g$   
**shows** *terminates*  $(maps-raw\ f\ g)$

⟨proof⟩

**lift-definition** *maps-trans*  $:: ('a \Rightarrow ('b, 'sg)\ generator \times 'sg) \Rightarrow ('a, 's)\ generator$   
 $\Rightarrow ('b, 's \times (('b, 'sg)\ generator \times 'sg)\ option)\ generator$

**is** *maps-raw* ⟨proof⟩

**lemma** *unstream-maps-trans-Some*:

$unstream\ (maps-trans\ f\ g)\ (s, Some\ (g'', s'')) = unstream\ g''\ s'' @ unstream\ (maps-trans$   
 $f\ g)\ (s, None)$

⟨proof⟩

**lemma** *unstream-maps-trans*:

$unstream\ (maps-trans\ f\ g)\ (s, None) = List.maps\ (case-prod\ unstream\ o\ f)\ (unstream$   
 $g\ s)$

⟨proof⟩

The rule *unstream-map-trans* is too complicated for fusion because of *split*, which does not arise naturally from stream fusion rules. Moreover, according to Farmer et al. [2],



this fusion is too general for further optimisations because the generators of the inner list are generated by the outer generator and therefore compilers may think that is was not known statically.

Instead, they propose a weaker version using *flatten* below. (More precisely, Coutts already mentions this approach in his PhD thesis [1], but dismisses it because it requires a stronger rewriting engine than GHC has. But Isabelle’s simplifier language is sufficiently powerful.

```
fun fix-step :: 'a ⇒ ('b, 's) step ⇒ ('b, 'a × 's) step
```

```
where
```

```
  fix-step a Done = Done
```

```
| fix-step a (Skip s) = Skip (a, s)
```

```
| fix-step a (Yield x s) = Yield x (a, s)
```

```
fun fix-gen-raw :: ('a ⇒ ('b, 's) raw-generator) ⇒ ('b, 'a × 's) raw-generator
```

```
where fix-gen-raw g (a, s) = fix-step a (g a s)
```

```
lemma terminates-fix-gen-raw:
```

```
  assumes  $\bigwedge x. \text{terminates } (g\ x)$ 
```

```
  shows terminates (fix-gen-raw g)
```

```
⟨proof⟩
```

```
lift-definition fix-gen :: ('a ⇒ ('b, 's) generator) ⇒ ('b, 'a × 's) generator
```

```
is fix-gen-raw ⟨proof⟩
```

```
lemma unstream-fix-gen: unstream (fix-gen g) (a, s) = unstream (g a) s
```

```
⟨proof⟩
```

```
context
```

```
  fixes f :: ('a ⇒ 's')
```

```
  and g'' :: ('b, 's') raw-generator
```

```
  and g :: ('a, 's) raw-generator
```

```
begin
```

```
fun flatten-raw :: ('b, 's × 's' option) raw-generator
```

```
where
```

```
  flatten-raw (s, None) = (case g s of
```

```
    Done ⇒ Done | Skip s' ⇒ Skip (s', None) | Yield x s' ⇒ Skip (s', Some (f x)))
```

```
| flatten-raw (s, Some s') = (case g'' s'' of
```

```
    Done ⇒ Skip (s, None) | Skip s' ⇒ Skip (s, Some s') | Yield x s' ⇒ Yield x (s, Some s'))
```

```
lemma terminates-flatten-raw:
```

```
  assumes terminates g'' terminates g
```

**shows** *terminates flatten-raw*  
 ⟨*proof*⟩

**end**

**lift-definition** *flatten* :: ('a ⇒ 's') ⇒ ('b, 's) generator ⇒ ('a, 's) generator ⇒ ('b, 's × 's' option) generator  
**is** *flatten-raw* ⟨*proof*⟩

**lemma** *unstream-flatten-Some*:

*unstream* (*flatten* *f* *g''* *g*) (*s*, *Some* *s'*) = *unstream* *g''* *s'* @ *unstream* (*flatten* *f* *g''* *g*) (*s*, *None*)  
 ⟨*proof*⟩

HO rewrite equations can express the variable capture in the generator unlike GHC rules

**lemma** *unstream-flatten-fix-gen* [*stream-fusion*]:

*unstream* (*flatten* (λ*s*. (*s*, *f* *s*)) (*fix-gen* *g''*) *g*) (*s*, *None*) =  
*List.maps* (λ*s'*. *unstream* (*g''* *s'*) (*f* *s'*)) (*unstream* *g* *s*)  
 ⟨*proof*⟩

Separate fusion rule when the inner generator does not depend on the elements of the outer stream.

**lemma** *unstream-flatten* [*stream-fusion*]:

*unstream* (*flatten* *f* *g''* *g*) (*s*, *None*) = *List.maps* (λ*s'*. *unstream* *g''* (*f* *s'*)) (*unstream* *g* *s*)  
 ⟨*proof*⟩

**end**

### 3 Stream fusion for coinductive lists

**theory** *Stream-Fusion-LList* **imports**

*Stream-Fusion-List*

*Coinductive.Coinductive-List*

**begin**

There are two choices of how many *Skips* may occur consecutively.

- A generator for '*a llist* may return only finitely many *Skips* before it has to decide on a *Done* or *Yield*. Then, we can define stream versions for all functions that can be defined by corecursion up-to. This in particular excludes *lfilter*. Moreover, we have to prove that every generator satisfies this restriction.
- A generator for '*a llist* may return infinitely many *Skips* in a row. Then, the *lunstream* function suffers from the same difficulties as *lfilter* with definitions, but we can define it using the least fixpoint approach described in [4]. Consequently,

we can only fuse transformers that are monotone and continuous with respect to the *ccpo* ordering. This in particular excludes *lappend*.

Here, we take the both approaches where we consider the first preferable to the second. Consequently, we define producers such that they produce generators of the first kind, if possible. There will be multiple equations for transformers and consumers that deal with all the different combinations for their parameter generators. Transformers should yield generators of the first kind whenever possible. Consumers can be defined using *lunstream* and refined with custom code equations, i.e., they can operate with infinitely many *Skips* in a row. We just have to lift the fusion equation to the first kind, too.

**type-synonym**  $(\prime a, \prime s)$  *lgenerator* =  $\prime s \Rightarrow (\prime a, \prime s)$  *step*

**inductive-set** *productive-on* ::  $(\prime a, \prime s)$  *lgenerator*  $\Rightarrow \prime s$  *set*

**for**  $g$  ::  $(\prime a, \prime s)$  *lgenerator*

**where**

*Done*:  $g\ s = \text{Done} \Longrightarrow s \in \text{productive-on } g$   
| *Skip*:  $\llbracket g\ s = \text{Skip } s'; s' \in \text{productive-on } g \rrbracket \Longrightarrow s \in \text{productive-on } g$   
| *Yield*:  $g\ s = \text{Yield } x\ s' \Longrightarrow s \in \text{productive-on } g$

**definition** *productive* ::  $(\prime a, \prime s)$  *lgenerator*  $\Rightarrow$  *bool*

**where** *productive*  $g \longleftrightarrow \text{productive-on } g = \text{UNIV}$

**lemma** *productiveI* [*intro?*]:

$(\bigwedge s. s \in \text{productive-on } g) \Longrightarrow \text{productive } g$   
 $\langle \text{proof} \rangle$

**lemma** *productive-onI* [*dest?*]: *productive*  $g \Longrightarrow s \in \text{productive-on } g$

$\langle \text{proof} \rangle$

A type of generators that eventually will yield something else than a skip.

**typedef**  $(\prime a, \prime s)$  *lgenerator'* =  $\{g :: (\prime a, \prime s)$  *lgenerator*. *productive*  $g\}$

**morphisms** *lgenerator* *Abs-lgenerator'*

$\langle \text{proof} \rangle$

**setup-lifting** *type-definition-lgenerator'*

### 3.1 Conversions to $\prime a$ *llist*

#### 3.1.1 Infinitely many consecutive *Skips*

**context fixes**  $g$  ::  $(\prime a, \prime s)$  *lgenerator*

**notes**  $\llbracket \text{function-internals} \rrbracket$

**begin**

**partial-function** (*llist*) *lunstream* ::  $\prime s \Rightarrow \prime a$  *llist*

**where**

$lunstream\ s = (case\ g\ s\ of$   
 $Done \Rightarrow LNil \mid Skip\ s' \Rightarrow lunstream\ s' \mid Yield\ x\ s' \Rightarrow LCons\ x\ (lunstream\ s'))$

**declare**  $lunstream.simps[code]$

**lemma**  $lunstream-simps:$

$g\ s = Done \implies lunstream\ s = LNil$   
 $g\ s = Skip\ s' \implies lunstream\ s = lunstream\ s'$   
 $g\ s = Yield\ x\ s' \implies lunstream\ s = LCons\ x\ (lunstream\ s')$   
 $\langle proof \rangle$

**lemma**  $lunstream-sels:$

**shows**  $lnull-lunstream: lnull\ (lunstream\ s) \longleftrightarrow$   
 $(case\ g\ s\ of\ Done \Rightarrow True \mid Skip\ s' \Rightarrow lnull\ (lunstream\ s') \mid Yield\ - \Rightarrow False)$   
**and**  $lhd-lunstream: lhd\ (lunstream\ s) =$   
 $(case\ g\ s\ of\ Skip\ s' \Rightarrow lhd\ (lunstream\ s') \mid Yield\ x\ - \Rightarrow x)$   
**and**  $ltl-lunstream: ltl\ (lunstream\ s) =$   
 $(case\ g\ s\ of\ Done \Rightarrow LNil \mid Skip\ s' \Rightarrow ltl\ (lunstream\ s') \mid Yield\ -\ s' \Rightarrow lunstream\ s')$   
 $\langle proof \rangle$

**end**

### 3.1.2 Finitely many consecutive *Skips*

**lift-definition**  $lunstream' :: ('a, 's)\ lgenerator' \Rightarrow 's \Rightarrow 'a\ llist$   
**is**  $lunstream\ \langle proof \rangle$

**lemma**  $lunstream'-simps:$

$lgenerator\ g\ s = Done \implies lunstream'\ g\ s = LNil$   
 $lgenerator\ g\ s = Skip\ s' \implies lunstream'\ g\ s = lunstream'\ g\ s'$   
 $lgenerator\ g\ s = Yield\ x\ s' \implies lunstream'\ g\ s = LCons\ x\ (lunstream'\ g\ s')$   
 $\langle proof \rangle$

**lemma**  $lunstream'-sels:$

**shows**  $lnull-lunstream': lnull\ (lunstream'\ g\ s) \longleftrightarrow$   
 $(case\ lgenerator\ g\ s\ of\ Done \Rightarrow True \mid Skip\ s' \Rightarrow lnull\ (lunstream'\ g\ s') \mid Yield\ - \Rightarrow$   
 $False)$   
**and**  $lhd-lunstream': lhd\ (lunstream'\ g\ s) =$   
 $(case\ lgenerator\ g\ s\ of\ Skip\ s' \Rightarrow lhd\ (lunstream'\ g\ s') \mid Yield\ x\ - \Rightarrow x)$   
**and**  $ltl-lunstream': ltl\ (lunstream'\ g\ s) =$   
 $(case\ lgenerator\ g\ s\ of\ Done \Rightarrow LNil \mid Skip\ s' \Rightarrow ltl\ (lunstream'\ g\ s') \mid Yield\ -\ s' \Rightarrow$   
 $lunstream'\ g\ s')$   
 $\langle proof \rangle$

$\langle ML \rangle$

## 3.2 Producers

### 3.2.1 Conversion to streams

**fun** *lstream* :: ('a, 'a llist) lgenerator  
**where**  
  *lstream* LNil = Done  
| *lstream* (LCons x xs) = Yield x xs

**lemma** *case-lstream-conv-case-llist*:

(*case lstream xs of Done*  $\Rightarrow$  *done* | *Skip xs'*  $\Rightarrow$  *skip xs'* | *Yield x xs'*  $\Rightarrow$  *yield x xs'*) =  
(*case xs of LNil*  $\Rightarrow$  *done* | *LCons x xs'*  $\Rightarrow$  *yield x xs'*)  
 $\langle proof \rangle$

**lemma** *mcont2mcont-lunstream*[*THEN llist.mcont2mcont, simp, cont-intro*]:

**shows** *mcont-lunstream*: *mcont* lSup lprefix lSup lprefix (*lunstream lstream*)  
 $\langle proof \rangle$

**lemma** *lunstream-lstream*: *lunstream lstream xs* = *xs*

$\langle proof \rangle$

**lift-definition** *lstream'* :: ('a, 'a llist) lgenerator'

**is** *lstream*

$\langle proof \rangle$

**lemma** *lunstream'-lstream*: *lunstream' lstream' xs* = *xs*

$\langle proof \rangle$

### 3.2.2 iterates

**definition** *iterates-raw* :: ('a  $\Rightarrow$  'a)  $\Rightarrow$  ('a, 'a) lgenerator

**where** *iterates-raw* f s = Yield s (f s)

**lemma** *lunstream-iterates-raw*: *lunstream (iterates-raw f) x* = *iterates f x*

$\langle proof \rangle$

**lift-definition** *iterates-prod* :: ('a  $\Rightarrow$  'a)  $\Rightarrow$  ('a, 'a) lgenerator' **is** *iterates-raw*

$\langle proof \rangle$

**lemma** *lunstream'-iterates-prod* [*stream-fusion*]: *lunstream' (iterates-prod f) x* = *iterates*

*f x*

$\langle proof \rangle$

### 3.2.3 *unfold-llist*

**definition** *unfold-llist-raw* :: ('a ⇒ bool) ⇒ ('a ⇒ 'b) ⇒ ('a ⇒ 'a) ⇒ ('b, 'a) lgenerator  
**where**

*unfold-llist-raw stop head tail s* = (if stop s then Done else Yield (head s) (tail s))

**lemma** *lunstream-unfold-llist-raw*:

*lunstream (unfold-llist-raw stop head tail) s* = *unfold-llist stop head tail s*  
<proof>

**lift-definition** *unfold-llist-prod* :: ('a ⇒ bool) ⇒ ('a ⇒ 'b) ⇒ ('a ⇒ 'a) ⇒ ('b, 'a) lgenerator'

**is** *unfold-llist-raw*

<proof>

**lemma** *lunstream'-unfold-llist-prod [stream-fusion]*:

*lunstream' (unfold-llist-prod stop head tail) s* = *unfold-llist stop head tail s*  
<proof>

### 3.2.4 *inf-llist*

**definition** *inf-llist-raw* :: (nat ⇒ 'a) ⇒ ('a, nat) lgenerator

**where** *inf-llist-raw f n* = Yield (f n) (Suc n)

**lemma** *lunstream-inf-llist-raw*: *lunstream (inf-llist-raw f) n* = *ldropn n (inf-llist f)*

<proof>

**lift-definition** *inf-llist-prod* :: (nat ⇒ 'a) ⇒ ('a, nat) lgenerator' **is** *inf-llist-raw*

<proof>

**lemma** *inf-llist-prod-fusion [stream-fusion]*:

*lunstream' (inf-llist-prod f) 0* = *inf-llist f*  
<proof>

## 3.3 Consumers

### 3.3.1 *lhd*

**context fixes** *g* :: ('a, 's) lgenerator **begin**

**definition** *lhd-cons* :: 's ⇒ 'a

**where** [*stream-fusion*]: *lhd-cons s* = *lhd (lunstream g s)*

**lemma** *lhd-cons-code[code]*:

*lhd-cons s* = (case *g s* of Done ⇒ undefined | Skip *s'* ⇒ *lhd-cons s'* | Yield *x* - ⇒ *x*)

*<proof>*

**end**

**lemma** *lhd-cons-fusion2* [*stream-fusion*]:

$lhd\text{-}cons (lgenerator\ g)\ s = lhd (lunstream'\ g\ s)$

*<proof>*

### 3.3.2 *llength*

**context fixes**  $g :: ('a, 's)\ lgenerator$  **begin**

**definition** *gen-llength-cons* ::  $enat \Rightarrow 's \Rightarrow enat$

**where**  $gen\text{-}llength\text{-}cons\ n\ s = n + llength (lunstream\ g\ s)$

**lemma** *gen-llength-cons-code* [*code*]:

$gen\text{-}llength\text{-}cons\ n\ s = (case\ g\ s\ of$

$Done \Rightarrow n \mid Skip\ s' \Rightarrow gen\text{-}llength\text{-}cons\ n\ s' \mid Yield\ -\ s' \Rightarrow gen\text{-}llength\text{-}cons\ (eSuc\ n)$   
 $s')$

*<proof>*

**lemma** *gen-llength-cons-fusion* [*stream-fusion*]:

$gen\text{-}llength\text{-}cons\ 0\ s = llength (lunstream\ g\ s)$

*<proof>*

**end**

**context fixes**  $g :: ('a, 's)\ lgenerator'$  **begin**

**definition** *gen-llength-cons'* ::  $enat \Rightarrow 's \Rightarrow enat$

**where**  $gen\text{-}llength\text{-}cons' = gen\text{-}llength\text{-}cons (lgenerator\ g)$

**lemma** *gen-llength-cons'-code* [*code*]:

$gen\text{-}llength\text{-}cons'\ n\ s = (case\ lgenerator\ g\ s\ of$

$Done \Rightarrow n \mid Skip\ s' \Rightarrow gen\text{-}llength\text{-}cons'\ n\ s' \mid Yield\ -\ s' \Rightarrow gen\text{-}llength\text{-}cons'\ (eSuc$   
 $n)\ s')$

*<proof>*

**lemma** *gen-llength-cons'-fusion* [*stream-fusion*]:

$gen\text{-}llength\text{-}cons'\ 0\ s = llength (lunstream'\ g\ s)$

*<proof>*

**end**

### 3.3.3 *lnull*

**context fixes**  $g :: ('a, 's) \text{ lgenerator}$  **begin**

**definition**  $lnull\text{-cons} :: 's \Rightarrow \text{bool}$

**where** [*stream-fusion*]:  $lnull\text{-cons } s \longleftrightarrow lnull (lunstream g s)$

**lemma**  $lnull\text{-cons-code}$  [*code*]:

$lnull\text{-cons } s \longleftrightarrow (\text{case } g \text{ s of}$

$\text{Done} \Rightarrow \text{True} \mid \text{Skip } s' \Rightarrow lnull\text{-cons } s' \mid \text{Yield } - \Rightarrow \text{False})$

$\langle \text{proof} \rangle$

**end**

**context fixes**  $g :: ('a, 's) \text{ lgenerator}'$  **begin**

**definition**  $lnull\text{-cons}' :: 's \Rightarrow \text{bool}$

**where**  $lnull\text{-cons}' = lnull\text{-cons} (lgenerator g)$

**lemma**  $lnull\text{-cons}'\text{-code}$  [*code*]:

$lnull\text{-cons}' s \longleftrightarrow (\text{case } lgenerator g \text{ s of}$

$\text{Done} \Rightarrow \text{True} \mid \text{Skip } s' \Rightarrow lnull\text{-cons}' s' \mid \text{Yield } - \Rightarrow \text{False})$

$\langle \text{proof} \rangle$

**lemma**  $lnull\text{-cons}'\text{-fusion}$  [*stream-fusion*]:

$lnull\text{-cons}' s \longleftrightarrow lnull (lunstream' g s)$

$\langle \text{proof} \rangle$

**end**

### 3.3.4 *llist-all2*

**context**

**fixes**  $g :: ('a, 'sg) \text{ lgenerator}$

**and**  $h :: ('b, 'sh) \text{ lgenerator}$

**and**  $P :: 'a \Rightarrow 'b \Rightarrow \text{bool}$

**begin**

**definition**  $l\text{list-all2-cons} :: 'sg \Rightarrow 'sh \Rightarrow \text{bool}$

**where** [*stream-fusion*]:  $l\text{list-all2-cons } sg \ sh \longleftrightarrow l\text{list-all2 } P (lunstream g sg) (lunstream h sh)$

**definition**  $l\text{list-all2-cons1} :: 'a \Rightarrow 'sg \Rightarrow 'sh \Rightarrow \text{bool}$

**where**  $l\text{list-all2-cons1 } x \ sg' \ sh = l\text{list-all2 } P (LCons x (lunstream g sg')) (lunstream h$



*sh*)

**lemma** *llist-all2-cons-code* [*code*]:

*llist-all2-cons sg sh =*  
  (*case g sg of*  
    *Done*  $\Rightarrow$  *lnull-cons h sh*  
    | *Skip sg'*  $\Rightarrow$  *llist-all2-cons sg' sh*  
    | *Yield a sg'*  $\Rightarrow$  *llist-all2-cons1 a sg' sh*)  
*<proof>*

**lemma** *llist-all2-cons1-code* [*code*]:

*llist-all2-cons1 x sg' sh =*  
  (*case h sh of*  
    *Done*  $\Rightarrow$  *False*  
    | *Skip sh'*  $\Rightarrow$  *llist-all2-cons1 x sg' sh'*  
    | *Yield y sh'*  $\Rightarrow$  *P x y  $\wedge$  llist-all2-cons sg' sh'*)  
*<proof>*

**end**

**lemma** *llist-all2-cons-fusion2* [*stream-fusion*]:

*llist-all2-cons (lgenerator g) (lgenerator h) P sg sh  $\longleftrightarrow$  llist-all2 P (lunstream' g sg)*  
*(lunstream' h sh)*  
*<proof>*

**lemma** *llist-all2-cons-fusion3* [*stream-fusion*]:

*llist-all2-cons g (lgenerator h) P sg sh  $\longleftrightarrow$  llist-all2 P (lunstream g sg) (lunstream' h*  
*sh)*  
*<proof>*

**lemma** *llist-all2-cons-fusion4* [*stream-fusion*]:

*llist-all2-cons (lgenerator g) h P sg sh  $\longleftrightarrow$  llist-all2 P (lunstream' g sg) (lunstream h*  
*sh)*  
*<proof>*

### 3.3.5 *lnth*

**context fixes** *g* :: (*'a*, *'s*) *lgenerator* **begin**

**definition** *lnth-cons* :: *nat*  $\Rightarrow$  *'s*  $\Rightarrow$  *'a*

**where** [*stream-fusion*]: *lnth-cons n s = lnth (lunstream g s) n*

**lemma** *lnth-cons-code* [*code*]:

*lnth-cons n s = (case g s of*

$Done \Rightarrow undefined\ n$   
 $| Skip\ s' \Rightarrow lnth-cons\ n\ s'$   
 $| Yield\ x\ s' \Rightarrow (if\ n = 0\ then\ x\ else\ lnth-cons\ (n - 1)\ s')$   
 <proof>

end

**lemma** *lnth-cons-fusion2* [*stream-fusion*]:  
 $lnth-cons\ (lgenerator\ g)\ n\ s = lnth\ (lunstream'\ g\ s)\ n$   
 <proof>

### 3.3.6 *lprefix*

**context**

**fixes**  $g :: ('a, 'sg)\ lgenerator$   
**and**  $h :: ('a, 'sh)\ lgenerator$

**begin**

**definition** *lprefix-cons* ::  $'sg \Rightarrow 'sh \Rightarrow bool$   
**where** [*stream-fusion*]:  $lprefix-cons\ sg\ sh \longleftrightarrow lprefix\ (lunstream\ g\ sg)\ (lunstream\ h\ sh)$

**definition** *lprefix-cons1* ::  $'a \Rightarrow 'sg \Rightarrow 'sh \Rightarrow bool$   
**where**  $lprefix-cons1\ x\ sg'\ sh \longleftrightarrow lprefix\ (LCons\ x\ (lunstream\ g\ sg'))\ (lunstream\ h\ sh)$

**lemma** *lprefix-cons-code* [*code*]:  
 $lprefix-cons\ sg\ sh \longleftrightarrow (case\ g\ sg\ of$   
 $Done \Rightarrow True\ | Skip\ sg' \Rightarrow lprefix-cons\ sg'\ sh\ | Yield\ x\ sg' \Rightarrow lprefix-cons1\ x\ sg'\ sh)$   
 <proof>

**lemma** *lprefix-cons1-code* [*code*]:  
 $lprefix-cons1\ x\ sg'\ sh \longleftrightarrow (case\ h\ sh\ of$   
 $Done \Rightarrow False\ | Skip\ sh' \Rightarrow lprefix-cons1\ x\ sg'\ sh'$   
 $| Yield\ y\ sh' \Rightarrow x = y \wedge lprefix-cons\ sg'\ sh')$   
 <proof>

end

**lemma** *lprefix-cons-fusion2* [*stream-fusion*]:  
 $lprefix-cons\ (lgenerator\ g)\ (lgenerator\ h)\ sg\ sh \longleftrightarrow lprefix\ (lunstream'\ g\ sg)\ (lunstream'\ h\ sh)$   
 <proof>

**lemma** *lprefix-cons-fusion3* [*stream-fusion*]:  
 $lprefix-cons\ g\ (lgenerator\ h)\ sg\ sh \longleftrightarrow lprefix\ (lunstream\ g\ sg)\ (lunstream'\ h\ sh)$

*<proof>*

**lemma** *lprefix-cons-fusion4* [*stream-fusion*]:

*lprefix-cons (lgenerator g) h sg sh*  $\longleftrightarrow$  *lprefix (lunstream' g sg) (lunstream h sh)*

*<proof>*

### 3.4 Transformers

#### 3.4.1 *lmap*

**definition** *lmap-trans* :: (*'a*  $\Rightarrow$  *'b*)  $\Rightarrow$  (*'a*, *'s*) *lgenerator*  $\Rightarrow$  (*'b*, *'s*) *lgenerator*

**where** *lmap-trans* = *map-raw*

**lemma** *lunstream-lmap-trans* [*stream-fusion*]: **fixes** *f g s*

**defines** [*simp*]: *g'*  $\equiv$  *lmap-trans f g*

**shows** *lunstream g' s* = *lmap f (lunstream g s)* (**is** *?lhs* = *?rhs*)

*<proof>*

**lift-definition** *lmap-trans'* :: (*'a*  $\Rightarrow$  *'b*)  $\Rightarrow$  (*'a*, *'s*) *lgenerator'*  $\Rightarrow$  (*'b*, *'s*) *lgenerator'*

**is** *lmap-trans*

*<proof>*

**lemma** *lunstream'-lmap-trans'* [*stream-fusion*]:

*lunstream' (lmap-trans' f g) s* = *lmap f (lunstream' g s)*

*<proof>*

#### 3.4.2 *ltake*

**fun** *ltake-trans* :: (*'a*, *'s*) *lgenerator*  $\Rightarrow$  (*'a*, (*enat*  $\times$  *'s*)) *lgenerator*

**where**

*ltake-trans g (n, s)* =

(*if n = 0 then Done else case g s of*

*Done*  $\Rightarrow$  *Done* | *Skip s'*  $\Rightarrow$  *Skip (n, s')* | *Yield a s'*  $\Rightarrow$  *Yield a (epred n, s')*)

**lemma** *ltake-trans-fusion* [*stream-fusion*]:

**fixes** *g' g*

**defines** [*simp*]: *g'*  $\equiv$  *ltake-trans g*

**shows** *lunstream g' (n, s)* = *ltake n (lunstream g s)* (**is** *?lhs* = *?rhs*)

*<proof>*

**lift-definition** *ltake-trans'* :: (*'a*, *'s*) *lgenerator'*  $\Rightarrow$  (*'a*, (*enat*  $\times$  *'s*)) *lgenerator'*

**is** *ltake-trans*

*<proof>*

**lemma** *ltake-trans'-fusion* [*stream-fusion*]:

$\text{lunstream}' (\text{ltake-trans}' g) (n, s) = \text{ltake } n (\text{lunstream}' g s)$   
 ⟨proof⟩

### 3.4.3 *ldropn*

**abbreviation** (*input*)  $\text{ldropn-trans} :: ('b, 'a) \text{lgenerator} \Rightarrow ('b, \text{nat} \times 'a) \text{lgenerator}$   
**where**  $\text{ldropn-trans} \equiv \text{drop-raw}$

**lemma**  $\text{ldropn-trans-fusion}$  [*stream-fusion*]:  
**fixes**  $g$  **defines** [*simp*]:  $g' \equiv \text{ldropn-trans } g$   
**shows**  $\text{lunstream } g' (n, s) = \text{ldropn } n (\text{lunstream } g s)$  (**is**  $?lhs = ?rhs$ )  
 ⟨proof⟩

**lift-definition**  $\text{ldropn-trans}' :: ('a, 's) \text{lgenerator}' \Rightarrow ('a, \text{nat} \times 's) \text{lgenerator}'$   
**is**  $\text{ldropn-trans}$   
 ⟨proof⟩

**lemma**  $\text{ldropn-trans}'\text{-fusion}$  [*stream-fusion*]:  
 $\text{lunstream}' (\text{ldropn-trans}' g) (n, s) = \text{ldropn } n (\text{lunstream}' g s)$   
 ⟨proof⟩

### 3.4.4 *ldrop*

**fun**  $\text{ldrop-trans} :: ('a, 's) \text{lgenerator} \Rightarrow ('a, \text{enat} \times 's) \text{lgenerator}$   
**where**

$\text{ldrop-trans } g (n, s) = (\text{case } g \text{ s of}$   
 $\text{Done} \Rightarrow \text{Done} \mid \text{Skip } s' \Rightarrow \text{Skip } (n, s')$   
 $\mid \text{Yield } x \text{ s}' \Rightarrow (\text{if } n = 0 \text{ then Yield } x (n, s') \text{ else Skip } (\text{epred } n, s')))$

**lemma**  $\text{ldrop-trans-fusion}$  [*stream-fusion*]:  
**fixes**  $g$   $g'$  **defines** [*simp*]:  $g' \equiv \text{ldrop-trans } g$   
**shows**  $\text{lunstream } g' (n, s) = \text{ldrop } n (\text{lunstream } g s)$  (**is**  $?lhs = ?rhs$ )  
 ⟨proof⟩

**lemma**  $\text{ldrop-trans-fusion2}$  [*stream-fusion*]:  
 $\text{lunstream } (\text{ldrop-trans } (\text{lgenerator } g)) (n, s) = \text{ldrop } n (\text{lunstream}' g s)$   
 ⟨proof⟩

### 3.4.5 *ltakeWhile*

**abbreviation** (*input*)  $\text{ltakeWhile-trans} :: ('a \Rightarrow \text{bool}) \Rightarrow ('a, 's) \text{lgenerator} \Rightarrow ('a, 's) \text{lgenerator}$   
**where**  $\text{ltakeWhile-trans} \equiv \text{takeWhile-raw}$

**lemma**  $\text{ltakeWhile-trans-fusion}$  [*stream-fusion*]:

**fixes**  $P g g'$  **defines**  $[simp]: g' \equiv ltakeWhile\text{-}trans P g$   
**shows**  $lunstream g' s = ltakeWhile P (lunstream g s)$  (**is**  $?lhs = ?rhs$ )  
 $\langle proof \rangle$

**lift-definition**  $ltakeWhile\text{-}trans' :: ('a \Rightarrow bool) \Rightarrow ('a, 's) lgenerator' \Rightarrow ('a, 's) lgenerator'$   
**is**  $ltakeWhile\text{-}trans$   
 $\langle proof \rangle$

**lemma**  $ltakeWhile\text{-}trans'\text{-}fusion [stream\text{-}fusion]:$   
 $lunstream' (ltakeWhile\text{-}trans' P g) s = ltakeWhile P (lunstream' g s)$   
 $\langle proof \rangle$

### 3.4.6 $ldropWhile$

**abbreviation**  $(input) ldropWhile\text{-}trans :: ('a \Rightarrow bool) \Rightarrow ('a, 'b) lgenerator \Rightarrow ('a, bool \times 'b) lgenerator$   
**where**  $ldropWhile\text{-}trans \equiv dropWhile\text{-}raw$

**lemma**  $ldropWhile\text{-}trans\text{-}fusion [stream\text{-}fusion]:$   
**fixes**  $P g g'$  **defines**  $[simp]: g' \equiv ldropWhile\text{-}trans P g$   
**shows**  $lunstream g' (True, s) = ldropWhile P (lunstream g s)$  (**is**  $?lhs = ?rhs$ )  
 $\langle proof \rangle$

**lemma**  $ldropWhile\text{-}trans\text{-}fusion2 [stream\text{-}fusion]:$   
 $lunstream (ldropWhile\text{-}trans P (lgenerator g)) (True, s) = ldropWhile P (lunstream' g s)$   
 $\langle proof \rangle$

### 3.4.7 $lzip$

**abbreviation**  $(input) lzip\text{-}trans :: ('a, 's1) lgenerator \Rightarrow ('b, 's2) lgenerator \Rightarrow ('a \times 'b, 's1 \times 's2 \times 'a option) lgenerator$   
**where**  $lzip\text{-}trans \equiv zip\text{-}raw$

**lemma**  $lzip\text{-}trans\text{-}fusion [stream\text{-}fusion]:$   
**fixes**  $g h gh$  **defines**  $[simp]: gh \equiv lzip\text{-}trans g h$   
**shows**  $lunstream gh (sg, sh, None) = lzip (lunstream g sg) (lunstream h sh)$   
(**is**  $?lhs = ?rhs$ )  
 $\langle proof \rangle$

**lemma**  $lzip\text{-}trans\text{-}fusion2 [stream\text{-}fusion]:$   
 $lunstream (lzip\text{-}trans (lgenerator g) h) (sg, sh, None) = lzip (lunstream' g sg) (lunstream h sh)$

*<proof>*

**lemma** *lzip-trans-fusion3* [*stream-fusion*]:

$\text{lunstream} (\text{lzip-trans } g (\text{lgenerator } h)) (sg, sh, \text{None}) = \text{lzip} (\text{lunstream } g \text{ sg}) (\text{lunstream}' h \text{ sh})$

*<proof>*

**lift-definition** *lzip-trans'* ::  $('a, 's1) \text{lgenerator}' \Rightarrow ('b, 's2) \text{lgenerator}' \Rightarrow ('a \times 'b, 's1 \times 's2 \times 'a \text{ option}) \text{lgenerator}'$

**is** *lzip-trans*

*<proof>*

**lemma** *lzip-trans'-fusion* [*stream-fusion*]:

$\text{lunstream}' (\text{lzip-trans}' g h) (sg, sh, \text{None}) = \text{lzip} (\text{lunstream}' g \text{ sg}) (\text{lunstream}' h \text{ sh})$

*<proof>*

### 3.4.8 *lappend*

**lift-definition** *lappend-trans* ::  $('a, 'sg) \text{lgenerator}' \Rightarrow ('a, 'sh) \text{lgenerator}' \Rightarrow 'sh \Rightarrow ('a, 'sg + 'sh) \text{lgenerator}'$

**is** *append-raw* *<proof>*

**lemma** *lunstream-append-raw*:

**fixes** *g h sh* **defines** [*simp*]:  $gh \equiv \text{append-raw } g \text{ h sh}$

**assumes** *productive g*

**shows**  $\text{lunstream } gh (\text{Inl } sg) = \text{lappend} (\text{lunstream } g \text{ sg}) (\text{lunstream } h \text{ sh})$

*<proof>*

**lemma** *lappend-trans-fusion* [*stream-fusion*]:

$\text{lunstream} (\text{lappend-trans } g \text{ h sh}) (\text{Inl } sg) = \text{lappend} (\text{lunstream}' g \text{ sg}) (\text{lunstream } h \text{ sh})$

*<proof>*

**lift-definition** *lappend-trans'* ::  $('a, 'sg) \text{lgenerator}' \Rightarrow ('a, 'sh) \text{lgenerator}' \Rightarrow 'sh \Rightarrow ('a, 'sg + 'sh) \text{lgenerator}'$

**is** *append-raw*

*<proof>*

**lemma** *lappend-trans'-fusion* [*stream-fusion*]:

$\text{lunstream}' (\text{lappend-trans}' g \text{ h sh}) (\text{Inl } sg) = \text{lappend} (\text{lunstream}' g \text{ sg}) (\text{lunstream}' h \text{ sh})$

*<proof>*

### 3.4.9 *lfilter*

**definition** *lfilter-trans* :: ('a ⇒ bool) ⇒ ('a, 's) lgenerator ⇒ ('a, 's) lgenerator  
**where** *lfilter-trans* = *filter-raw*

**lemma** *lunstream-lfilter-trans* [*stream-fusion*]:  
  **fixes** *P g g'* **defines** [*simp*]: *g' ≡ lfilter-trans P g*  
  **shows** *lunstream g' s = lfilter P (lunstream g s)* (**is** ?lhs = ?rhs)  
  ⟨*proof*⟩

**lemma** *lunstream-lfilter-trans2* [*stream-fusion*]:  
  *lunstream (lfilter-trans P (lgenerator g)) s = lfilter P (lunstream' g s)*  
  ⟨*proof*⟩

### 3.4.10 *llist-of*

**lift-definition** *llist-of-trans* :: ('a, 's) generator ⇒ ('a, 's) lgenerator'  
**is**  $\lambda x. x$   
  ⟨*proof*⟩

**lemma** *lunstream-llist-of-trans* [*stream-fusion*]:  
  *lunstream' (llist-of-trans g) s = llist-of (unstream g s)*  
  ⟨*proof*⟩

We cannot define a stream version of *list-of* because we would have to test for finiteness first and therefore traverse the list twice.

**end**

## 4 Examples and test cases for stream fusion

**theory** *Stream-Fusion-Examples* **imports** *Stream-Fusion-LList* **begin**

**lemma** **fixes** *rhs z*  
  **defines** *rhs ≡ nth-cons (flatten (λs'. s') (upto-prod 17) (upto-prod z)) (2, None) 8*  
  **shows** *nth (List.maps (λx. upto x 17) (upto 2 z)) 8 = rhs*  
  ⟨*proof*⟩

**lemma** **fixes** *rhs z*  
  **defines** *rhs ≡ nth-cons (flatten (λs. (s, 1)) (fix-gen (λx. upto-prod (id x))) (upto-prod z)) (2, None) 8*  
  **shows** *nth (List.maps (λx. upto 1 (id x)) (upto 2 z)) 8 = rhs*  
  ⟨*proof*⟩

**lemma** **fixes** *rhs n*

**defines**  $rhs \equiv List.maps (\lambda x. [Suc\ 0..<sum-list-cons\ (replicate-prod\ x)\ x])\ [2..<n]$   
**shows**  $(concat\ (map\ (\lambda x. [1..<sum-list\ (replicate\ x\ x)])\ [2..<n])) = rhs$   
 $\langle proof \rangle$

#### 4.1 Micro-benchmarks from Farmer et al. [2]

**definition**  $test-enum :: nat \Rightarrow nat$  — *id* required to avoid eta contraction  
**where**  $test-enum\ n = foldl\ (+)\ 0\ (List.maps\ (\lambda x. upt\ 1\ (id\ x))\ (upt\ 1\ n))$

**definition**  $test-nested :: nat \Rightarrow nat$   
**where**  $test-nested\ n = foldl\ (+)\ 0\ (List.maps\ (\lambda x. List.maps\ (\lambda y. upt\ y\ x)\ (upt\ 1\ x))\ (upt\ 1\ n))$

**definition**  $test-merge :: integer \Rightarrow nat$   
**where**  $test-merge\ n = foldl\ (+)\ 0\ (List.maps\ (\lambda x. if\ 2\ dvd\ x\ then\ upt\ 1\ x\ else\ upt\ 2\ x)\ (upt\ 1\ (nat-of-integer\ n)))$

This rule performs the merge operation from [2, §5.2] for *if*. In general, we would also need it for all case operators.

**lemma** *unstream-if* [*stream-fusion*]:  
 $unstream\ (if\ b\ then\ g\ else\ g')\ (if\ b\ then\ s\ else\ s') =$   
 $(if\ b\ then\ unstream\ g\ s\ else\ unstream\ g'\ s')$   
 $\langle proof \rangle$

**lemma** *if-same* [*code-unfold*]:  $(if\ b\ then\ x\ else\ x) = x$   
 $\langle proof \rangle$

**code-thms** *test-enum*

**code-thms** *test-nested*

**code-thms** *test-merge*

#### 4.2 Test stream fusion in the code generator

**definition**  $fuse-test :: integer$   
**where**  $fuse-test =$   
 $integer-of-int\ (lhd\ (lfilter\ (\lambda x. x < 1)\ (lappend\ (lmap\ (\lambda x. x + 1)\ (llist-of\ (map\ (\lambda x. if\ x = 0\ then\ undefined\ else\ x)\ [-3..5])))\ (repeat\ 3))))$

$\langle ML \rangle$

**end**



## References

- [1] D. Coutts. *Stream Fusion: Practical shortcut fusion for coinductive sequence types*. PhD thesis, University of Oxford, 2010.
- [2] A. Farmer, C. Höner zu Siederdissen, and A. Gill. The HERMIT in the stream. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation (PEPM 2014)*, pages 97–108. ACM, 2014.
- [3] B. Huffman. Stream fusion. *Archive of Formal Proofs*, 2009. <http://isa-afp.org/entries/Stream-Fusion.shtml>, Formal proof development.
- [4] A. Lochbihler and J. Hölzl. Recursive functions on lazy lists via domains and topologies. volume 8558 of *LNCS (LNAI)*, pages 341–357. Springer, 2014.