

AutoFocus Stream Processing for Single-Clocking and Multi-Clocking Semantics

David Trachtenherz

May 26, 2024

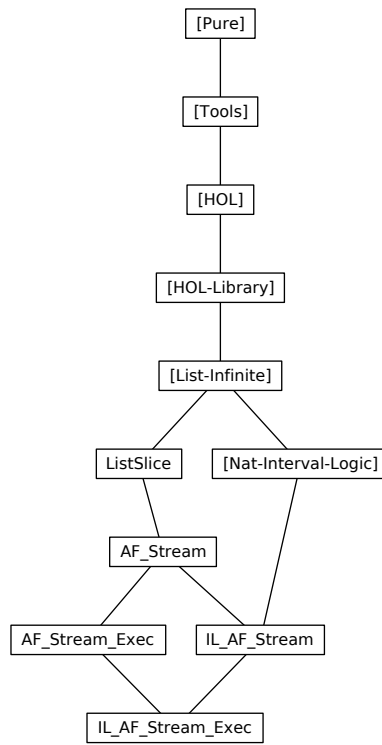
Abstract

We formalize the AutoFocus Semantics (a time-synchronous subset of the Focus formalism) as stream processing functions on finite and infinite message streams represented as finite/infinite lists. The formalization comprises both the conventional single-clocking semantics (uniform global clock for all components and communications channels) and its extension to multi-clocking semantics (internal execution clocking of a component may be a multiple of the external communication clocking). The semantics is defined by generic stream processing functions making it suitable for simulation/code generation in Isabelle/HOL. Furthermore, a number of AutoFocus semantics properties are formalized using definitions from the Nat-Interval-Logic theories.

Contents

1	Additional definitions and results for lists	3
1.1	Slicing lists into lists of lists	3
2	AutoFocus message streams	7
2.1	Basic definitions	7
2.1.1	Time-synchronous streams	7
2.1.2	Time abstraction	10
2.2	Expanding and compressing lists and streams	11
2.2.1	Expanding message streams	11
2.2.2	Aggregating lists	16
2.2.3	Compressing message streams	19
2.2.4	Holding last messages in everly cycle of a stream . . .	26
2.2.5	Compressing lists	29
3	Processing of message streams	33
3.1	Executing components with state transition functions	33
3.1.1	Basic definitions	33

3.1.2	Basic results	34
3.1.3	Connected streams	52
3.1.4	Additional auxiliary results	55
3.2	Components with accelerated execution	57
3.2.1	Equivalence relation for executions	57
3.2.2	Idle states	63
3.2.3	Basic definitions for accelerated execution	67
3.2.4	Basic results for accelerated execution	68
3.2.5	Basic results for accelerated execution with initial state in the resulting stream	76
3.2.6	Rules for proving execution equivalence	79
3.2.7	Idle states and accelerated execution	85
4	AutoFocus message streams and temporal logic on intervals	88
4.1	Stream views – joining streams and intervals	88
4.1.1	Basic definitions	88
4.1.2	Basic results	89
4.1.3	Results for intervals from <i>IL-Interval</i>	93
4.2	Streams and temporal operators	97
5	AutoFocus message stream processing and temporal logic on intervals	98
5.1	Correlation between Pre/Post-Conditions for <i>f-Exec-Comp-Stream</i> and <i>f-Exec-Comp-Stream-Init</i>	99
5.2	<i>i-Exec-Comp-Stream-Acc-Output</i> and temporal operators with bounded intervals.	100
5.3	<i>i-Exec-Comp-Stream-Acc-Output</i> and temporal operators with unbounded intervals and start/finish events.	101
5.4	<i>i-Exec-Comp-Stream-Acc-Output</i> and temporal operators with idle states.	103



1 Additional definitions and results for lists

```

theory ListSlice
imports List-Infinite.ListInf
begin

```

1.1 Slicing lists into lists of lists

```

definition ilist-slice :: 'a ilist  $\Rightarrow$  nat  $\Rightarrow$  'a list ilist
  where ilist-slice f k  $\equiv$   $\lambda x.$  map f [x * k..Suc x * k]

```

```

primrec list-slice-aux :: 'a list  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  'a list list
where
  list-slice-aux xs k 0 = []
| list-slice-aux xs k (Suc n) = take k xs # list-slice-aux (xs  $\uparrow$  k) k n

```

```

definition list-slice :: 'a list  $\Rightarrow$  nat  $\Rightarrow$  'a list list
  where list-slice xs k  $\equiv$  list-slice-aux xs k (length xs div k)

```

```

definition list-slice2 :: 'a list  $\Rightarrow$  nat  $\Rightarrow$  'a list list
  where list-slice2 xs k  $\equiv$ 
    list-slice xs k @ (if length xs mod k = 0 then [] else [xs  $\uparrow$  (length xs div k * k)])

```

No function *list-unslice* for finite lists is needed because the corresponding functionality is already provided by *concat*. Therefore, only a *ilist-unslice* function for infinite lists is defined.

```

definition ilist-unslice :: 'a list ilist  $\Rightarrow$  'a ilist
  where ilist-unslice f  $\equiv$   $\lambda n.$  f (n div length (f 0)) ! (n mod length (f 0))

```

```

lemma list-slice-aux-length:  $\bigwedge xs.$  length (list-slice-aux xs k n) = n
<proof>

```

```

lemma list-slice-aux-nth:
   $\bigwedge m xs.$  m < n  $\implies$  (list-slice-aux xs k n) ! m = (xs  $\uparrow$  (m * k)  $\downarrow$  k)
<proof>

```

```

lemma list-slice-length: length (list-slice xs k) = length xs div k
<proof>

```

```

lemma list-slice-0: list-slice xs 0 = []
<proof>

```

```

lemma list-slice-1: list-slice xs (Suc 0) = map ( $\lambda x.$  [x]) xs
<proof>

```

```

lemma list-slice-less: length xs < k  $\implies$  list-slice xs k = []
<proof>

```

lemma *list-slice-Nil*: $list\text{-}slice\ []\ k = []$
 ⟨proof⟩

lemma *list-slice-nth*:
 $m < length\ xs\ div\ k \implies list\text{-}slice\ xs\ k\ !\ m = xs\ \uparrow\ (m * k)\ \downarrow\ k$
 ⟨proof⟩

lemma *list-slice-nth-length*:
 $m < length\ xs\ div\ k \implies length\ ((list\text{-}slice\ xs\ k)\ !\ m) = k$
 ⟨proof⟩

lemma *list-slice-nth-eq-sublist-list*:
 $m < length\ xs\ div\ k \implies list\text{-}slice\ xs\ k\ !\ m = sublist\text{-}list\ xs\ [m * k..<m * k + k]$
 ⟨proof⟩

lemma *list-slice-nth-nth*:
 $\llbracket m < length\ xs\ div\ k; n < k \rrbracket \implies$
 $(list\text{-}slice\ xs\ k)\ !\ m\ !\ n = xs\ !\ (m * k + n)$
 ⟨proof⟩

lemma *list-slice-nth-nth-rev*:
 $n < length\ xs\ div\ k * k \implies$
 $(list\text{-}slice\ xs\ k)\ !\ (n\ div\ k)\ !\ (n\ mod\ k) = xs\ !\ n$
 ⟨proof⟩

lemma *list-slice-eq-list-slice-take*:
 $list\text{-}slice\ (xs\ \downarrow\ (length\ xs\ div\ k * k))\ k = list\text{-}slice\ xs\ k$
 ⟨proof⟩

lemma *list-slice-append-mult*:
 $\bigwedge xs. length\ xs = m * k \implies$
 $list\text{-}slice\ (xs\ @\ ys)\ k = list\text{-}slice\ xs\ k\ @\ list\text{-}slice\ ys\ k$
 ⟨proof⟩

lemma *list-slice-append-mod*:
 $length\ xs\ mod\ k = 0 \implies$
 $list\text{-}slice\ (xs\ @\ ys)\ k = list\text{-}slice\ xs\ k\ @\ list\text{-}slice\ ys\ k$
 ⟨proof⟩

lemma *list-slice-div-eq-1*[rule-format]:
 $length\ xs\ div\ k = Suc\ 0 \implies list\text{-}slice\ xs\ k = [take\ k\ xs]$
 ⟨proof⟩

lemma *list-slice-div-eq-Suc*[rule-format]:
 $length\ xs\ div\ k = Suc\ n \implies$
 $list\text{-}slice\ xs\ k = list\text{-}slice\ (xs\ \downarrow\ (n * k))\ k\ @\ [xs\ \uparrow\ (n * k)\ \downarrow\ k]$
 ⟨proof⟩

lemma *list-slice2-mod-0*:

$length\ xs\ mod\ k = 0 \implies list\ slice2\ xs\ k = list\ slice\ xs\ k$
 ⟨proof⟩

lemma *list-slice2-mod-gr0*:

$0 < length\ xs\ mod\ k \implies list\ slice2\ xs\ k = list\ slice\ xs\ k\ @\ [xs\ \uparrow\ (length\ xs\ div\ k\ * k)]$
 ⟨proof⟩

lemma *list-slice2-length*:

$length\ (list\ slice2\ xs\ k) =$
 $(if\ length\ xs\ mod\ k = 0\ then\ length\ xs\ div\ k\ else\ Suc\ (length\ xs\ div\ k))$
 ⟨proof⟩

lemma *list-slice2-0*:

$list\ slice2\ xs\ 0 = (if\ (length\ xs = 0)\ then\ []\ else\ [xs])$
 ⟨proof⟩

lemma *list-slice2-1*: $list\ slice2\ xs\ (Suc\ 0) = map\ (\lambda x. [x])\ xs$

⟨proof⟩

lemma *list-slice2-le*:

$length\ xs \leq k \implies list\ slice2\ xs\ k = (if\ length\ xs = 0\ then\ []\ else\ [xs])$
 ⟨proof⟩

lemma *list-slice2-Nil*: $list\ slice2\ []\ k = []$

⟨proof⟩

lemma *list-slice2-list-slice-nth*:

$m < length\ xs\ div\ k \implies list\ slice2\ xs\ k\ !\ m = list\ slice\ xs\ k\ !\ m$
 ⟨proof⟩

lemma *list-slice2-last*:

$[length\ xs\ mod\ k > 0; m = length\ xs\ div\ k] \implies$
 $list\ slice2\ xs\ k\ !\ m = xs\ \uparrow\ (length\ xs\ div\ k\ * k)$
 ⟨proof⟩

lemma *list-slice2-nth*:

$[m < length\ xs\ div\ k] \implies$
 $list\ slice2\ xs\ k\ !\ m = xs\ \uparrow\ (m\ * k)\ \downarrow\ k$
 ⟨proof⟩

lemma *list-slice2-nth-length-eq1*:

$m < length\ xs\ div\ k \implies length\ (list\ slice2\ xs\ k\ !\ m) = k$
 ⟨proof⟩

lemma *list-slice2-nth-length-eq2*:

$[length\ xs\ mod\ k > 0; m = length\ xs\ div\ k] \implies$
 $length\ (list\ slice2\ xs\ k\ !\ m) = length\ xs\ mod\ k$
 ⟨proof⟩

lemma *list-slice2-nth-nth-eq1*:

$\llbracket m < \text{length } xs \text{ div } k; n < k \rrbracket \implies$
 $(\text{list-slice2 } xs \ k) ! m ! n = xs ! (m * k + n)$
 <proof>

lemma *list-slice2-nth-nth-eq2*:

$\llbracket m = \text{length } xs \text{ div } k; n < \text{length } xs \text{ mod } k \rrbracket \implies$
 $(\text{list-slice2 } xs \ k) ! m ! n = xs ! (m * k + n)$
 <proof>

lemma *list-slice2-nth-nth-rev*:

$n < \text{length } xs \implies (\text{list-slice2 } xs \ k) ! (n \text{ div } k) ! (n \text{ mod } k) = xs ! n$
 <proof>

lemma *list-slice2-append-mult*:

$\text{length } xs = m * k \implies$
 $\text{list-slice2 } (xs @ ys) \ k = \text{list-slice2 } xs \ k @ \text{list-slice2 } ys \ k$
 <proof>

lemma *list-slice2-append-mod*:

$\text{length } xs \text{ mod } k = 0 \implies$
 $\text{list-slice2 } (xs @ ys) \ k = \text{list-slice2 } xs \ k @ \text{list-slice2 } ys \ k$
 <proof>

lemma *ilist-slice-nth*:

$(\text{ilist-slice } f \ k) \ m = \text{map } f \ [m * k .. < \text{Suc } m * k]$
 <proof>

lemma *ilist-slice-nth-length*: $\text{length } ((\text{ilist-slice } f \ k) \ m) = k$

<proof>

lemma *ilist-slice-nth-nth*:

$n < k \implies (\text{ilist-slice } f \ k) \ m ! n = f \ (m * k + n)$
 <proof>

lemma *ilist-slice-nth-nth-rev*:

$0 < k \implies (\text{ilist-slice } f \ k) \ (n \text{ div } k) ! (n \text{ mod } k) = f \ n$
 <proof>

lemma *list-slice-concat*:

$\text{concat } (\text{list-slice } xs \ k) = xs \downarrow (\text{length } xs \text{ div } k * k)$
 (is ?P xs k)
 <proof>

lemma *list-slice-unslice-mult*:

$\text{length } xs = m * k \implies \text{concat } (\text{list-slice } xs \ k) = xs$
 <proof>

lemma *ilist-slice-unslice*: $0 < k \implies \text{ilist-unslice } (\text{ilist-slice } f \ k) = f$
 ⟨proof⟩

lemma *i-take-ilist-slice-eq-list-slice*:
 $0 < k \implies \text{ilist-slice } f \ k \Downarrow n = \text{list-slice } (f \Downarrow (n * k)) \ k$
 ⟨proof⟩

lemma *list-slice-i-take-eq-i-take-ilist-slice*:
 $\text{list-slice } (f \Downarrow n) \ k = \text{ilist-slice } f \ k \Downarrow (n \text{ div } k)$
 ⟨proof⟩

lemma *ilist-slice-i-append-mod*:
 $\text{length } xs \text{ mod } k = 0 \implies$
 $\text{ilist-slice } (xs \frown f) \ k = \text{list-slice } xs \ k \frown \text{ilist-slice } f \ k$
 ⟨proof⟩

corollary *ilist-slice-append-mult*:
 $\text{length } xs = m * k \implies$
 $\text{ilist-slice } (xs \frown f) \ k = \text{list-slice } xs \ k \frown \text{ilist-slice } f \ k$
 ⟨proof⟩

end

2 AutoFocus message streams

theory *AF-Stream*
imports *ListSlice*
begin

2.1 Basic definitions

2.1.1 Time-synchronous streams

datatype *'a message-af* = *NoMsg* | *Msg 'a*

notation (*latex*)
NoMsg (ε) **and**
Msg (*Msg*)

Abbreviation for finite streams

type-synonym *'a fstream-af* = *'a message-af list*

Abbreviation for infinite streams

type-synonym *'a istream-af* = *'a message-af ilist*

lemma *not-NoMsg-eq*: $(m \neq \varepsilon) = (\exists x. m = \text{Msg } x)$
 ⟨proof⟩

lemma *not-Msg-eq*: $(\forall x. m \neq \text{Msg } x) = (m = \varepsilon)$
<proof>

primrec *the-af* :: $'a \text{ message-af} \Rightarrow 'a$
where *the-af* (*Msg* x) = x

By this definition one can determine, whether data elements of different data structures with messages, especially product types of arbitrary sizes and records, are pointwise equal to NoMsg, i.e., contain only NoMsg entries.

consts *is-NoMsg* :: $'a \Rightarrow \text{bool}$

overloading *is-NoMsg* \equiv *is-NoMsg* :: $'a \text{ message-af} \Rightarrow \text{bool}$
begin

primrec *is-NoMsg* :: $'a \text{ message-af} \Rightarrow \text{bool}$
where
is-NoMsg $\varepsilon = \text{True}$
| *is-NoMsg* (*Msg* x) = *False*

end

overloading *is-NoMsg* \equiv *is-NoMsg* :: $('a \times 'b) \Rightarrow \text{bool}$
begin

definition *is-NoMsg-tuple-def* :
is-NoMsg ($p :: 'a \times 'b$) \equiv (*is-NoMsg* (*fst* p) \wedge *is-NoMsg* (*snd* p))

end

overloading *is-NoMsg* \equiv *is-NoMsg* :: $'a \text{ set} \Rightarrow \text{bool}$
begin

definition *is-NoMsg-set-def* :
is-NoMsg ($A :: 'a \text{ set}$) \equiv $(\forall x \in A. \text{is-NoMsg } x)$

end

record *SomeRecordExample* =
Field1 :: *nat message-af*
Field2 :: *int message-af*
Field3 :: *int message-af*

overloading *is-NoMsg* \equiv *is-NoMsg* :: $'a \text{ SomeRecordExample-scheme} \Rightarrow \text{bool}$
begin

definition *is-NoMsg-SomeRecordExample-def* :
is-NoMsg ($r :: 'a \text{ SomeRecordExample-scheme}$) \equiv
Field1 $r = \varepsilon \wedge$ *Field2* $r = \varepsilon \wedge$ *Field3* $r = \varepsilon$

end

definition $is-Msg :: 'a \Rightarrow bool$
where $is-Msg\ x \equiv (\neg is-NoMsg\ x)$

lemma $is-NoMsg-message-af-conv$: $is-NoMsg\ m = (case\ m\ of\ \varepsilon \Rightarrow True \mid Msg\ x \Rightarrow False)$
 $\langle proof \rangle$

lemma $is-NoMsg-message-af-conv2$: $is-NoMsg\ m = (m = \varepsilon)$
 $\langle proof \rangle$

lemma $is-Msg-message-af-conv$: $is-Msg\ m = (case\ m\ of\ \varepsilon \Rightarrow False \mid Msg\ x \Rightarrow True)$
 $\langle proof \rangle$

lemma $is-Msg-message-af-conv2$: $is-Msg\ m = (m \neq \varepsilon)$
 $\langle proof \rangle$

Collection for definitions for $is-NoMsg$.

named-theorems $is-NoMsg-defs$

declare

$is-NoMsg-tuple-def[is-NoMsg-defs]$
 $is-NoMsg-set-def [is-NoMsg-defs]$
 $is-NoMsg-SomeRecordExample-def[is-NoMsg-defs]$
 $is-Msg-def[is-NoMsg-defs]$

lemma $not-is-NoMsg$: $(\neg is-NoMsg\ m) = is-Msg\ m$
 $\langle proof \rangle$

lemma $not-is-Msg$: $(\neg is-Msg\ m) = is-NoMsg\ m$
 $\langle proof \rangle$

lemma $is-NoMsg\ (\varepsilon :: (nat\ message-af))$
 $\langle proof \rangle$

lemma $is-NoMsg\ (\varepsilon :: (nat\ message-af), \varepsilon :: (nat\ message-af))$
 $\langle proof \rangle$

lemma $is-NoMsg\ (\varepsilon :: (nat\ message-af), \varepsilon :: (nat\ message-af), \varepsilon :: (nat\ message-af))$
 $\langle proof \rangle$

lemma $is-Msg\ (\varepsilon :: (nat\ message-af), Msg\ (1 :: nat), \varepsilon :: (nat\ message-af))$
 $\langle proof \rangle$

lemma $is-NoMsg\ \{\varepsilon :: (nat\ message-af), \varepsilon\}$
 $\langle proof \rangle$

lemma *is-Msg* { $\varepsilon :: (\text{nat message-af}), \text{Msg } 1$ }
 ⟨*proof*⟩

lemma *is-NoMsg* (| *Field1* = ε , *Field2* = ε , *Field3* = ε |)
 ⟨*proof*⟩

lemma *is-Msg* (| *Field1* = ε , *Field2* = *Msg 1*, *Field3* = ε |)
 ⟨*proof*⟩

2.1.2 Time abstraction

primrec *untime* :: 'a *fstream-af* \Rightarrow 'a *list*

where

untime [] = []
 | *untime* ($x \# xs$) =
 (if $x = \varepsilon$
 then (*untime* xs)
 else (*the-af* x) # (*untime* xs))

lemma *untime-eq-filter*[*rule-format*]:

map ($\lambda x. \text{Msg } x$) (*untime* s) = *filter* ($\lambda x. x \neq \varepsilon$) s
 ⟨*proof*⟩

The following lemma involves *the-af* function and thus is some more limited than the previous lemma

corollary *untime-eq-filter2*[*rule-format*]:

untime s = *map* ($\lambda x. \text{the-af } x$) (*filter* ($\lambda x. x \neq \varepsilon$) s)
 ⟨*proof*⟩

definition *untime-length* :: 'a *fstream-af* \Rightarrow *nat*

where *untime-length* $s \equiv \text{length} (\text{untime } s)$

primrec *untime-length-cnt* :: 'a *fstream-af* \Rightarrow *nat*

where

untime-length-cnt [] = 0
 | *untime-length-cnt* ($x \# xs$) =
 (if $x = \varepsilon$ then 0 else *Suc* 0) + *untime-length-cnt* xs

lemma *untime-length-eq-untime-length-cnt*:

untime-length s = *untime-length-cnt* s
 ⟨*proof*⟩

definition *untime-length-filter* :: 'a *fstream-af* \Rightarrow *nat*

where *untime-length-filter* $s \equiv \text{length} (\text{filter} (\lambda x. x \neq \varepsilon) s)$

lemma *untime-length-filter-eq-untime-length*:

untime-length-filter s = *untime-length* s
 ⟨*proof*⟩

lemma *untime-empty-conv*: $(\text{untime } s = []) = (\forall n < \text{length } s. s ! n = \varepsilon)$
 ⟨proof⟩

lemma *untime-not-empty-conv*: $(\text{untime } s \neq []) = (\exists n < \text{length } s. s ! n \neq \varepsilon)$
 ⟨proof⟩

corollary *untime-empty-imp-NoMsg*[rule-format]:
 $\llbracket \text{untime } s = [] ; n < \text{length } s \rrbracket \implies s ! n = \varepsilon$
 ⟨proof⟩

lemma *untime-nth-eq-filter*:
 $n < \text{untime-length } s \implies$
 $\text{Msg } (\text{untime } s ! n) = (\text{filter } (\lambda x. x \neq \varepsilon) s) ! n$
 ⟨proof⟩

corollary *untime-nth-eq-filter2*:
 $n < \text{untime-length } s \implies$
 $\text{untime } s ! n = \text{the-af } ((\text{filter } (\lambda x. x \neq \varepsilon) s) ! n)$
 ⟨proof⟩

lemma *untime-hd-eq-filter-hd*:
 $\text{untime } s \neq [] \implies$
 $\text{Msg } (\text{hd } (\text{untime } s)) = \text{hd } (\text{filter } (\lambda x. x \neq \varepsilon) s)$
 ⟨proof⟩

corollary *untime-hd-eq-filter-hd2*:
 $\text{untime } s \neq [] \implies$
 $\text{hd } (\text{untime } s) = \text{the-af } (\text{hd } (\text{filter } (\lambda x. x \neq \varepsilon) s))$
 ⟨proof⟩

lemma *untime-last-eq-filter-last*:
 $\text{untime } s \neq [] \implies$
 $\text{Msg } (\text{last } (\text{untime } s)) = \text{last } (\text{filter } (\lambda x. x \neq \varepsilon) s)$
 ⟨proof⟩

corollary *untime-last-eq-filter-last2*:
 $\text{untime } s \neq [] \implies$
 $\text{last } (\text{untime } s) = \text{the-af } (\text{last } (\text{filter } (\lambda x. x \neq \varepsilon) s))$
 ⟨proof⟩

2.2 Expanding and compressing lists and streams

2.2.1 Expanding message streams

primrec *f-expand* :: 'a fstream-af \Rightarrow nat \Rightarrow 'a fstream-af (**infixl** \odot_f 100)
 where

$f\text{-expand-Nil}: [] \odot_f k = []$
 $| f\text{-expand-Cons}: (x \# xs) \odot_f k =$
 $(\text{if } 0 < k \text{ then } x \# \varepsilon^k - \text{Suc } 0 @ (xs \odot_f k) \text{ else } [])$

definition $i\text{-expand} :: 'a \text{ istream-af} \Rightarrow \text{nat} \Rightarrow 'a \text{ istream-af}$ (**infixl** \odot_i 100)
where

$i\text{-expand} \equiv \lambda f k n.$
 $(\text{if } k = 0 \text{ then } \varepsilon \text{ else}$
 $\text{if } n \bmod k = 0 \text{ then } f (n \text{ div } k) \text{ else } \varepsilon)$

primrec $f\text{-expand-Suc} :: 'a \text{ fstream-af} \Rightarrow \text{nat} \Rightarrow 'a \text{ fstream-af}$ (**infixl** $\odot_{f\text{Suc}}$ 100)
where

$f\text{-expand-Suc} [] k = []$
 $| f\text{-expand-Suc} (x \# xs) k = x \# \varepsilon^k @ (f\text{-expand-Suc } xs k)$

definition $i\text{-expand-Suc} :: 'a \text{ istream-af} \Rightarrow \text{nat} \Rightarrow 'a \text{ istream-af}$ (**infixl** $\odot_{i\text{Suc}}$ 100)
where $i\text{-expand-Suc} \equiv \lambda f k n.$ $\text{if } n \bmod (\text{Suc } k) = 0 \text{ then } f (n \text{ div } (\text{Suc } k)) \text{ else } \varepsilon$

notation

$f\text{-expand}$ (**infixl** \odot 100) **and**
 $i\text{-expand}$ (**infixl** \odot 100)

lemma $\text{length-}f\text{-expand-Suc}[\text{simp}]$: $\text{length } (f\text{-expand-Suc } xs k) = \text{length } xs * \text{Suc } k$
 $\langle \text{proof} \rangle$

lemma $i\text{-expand-if}$:

$f \odot_i k = (\text{if } k = 0 \text{ then } (\lambda n. \varepsilon) \text{ else}$
 $(\lambda n. \text{if } n \bmod k = 0 \text{ then } f (n \text{ div } k) \text{ else } \varepsilon))$
 $\langle \text{proof} \rangle$

lemma $f\text{-expand-one}$: $0 < k \Longrightarrow [a] \odot_f k = a \# \varepsilon^k - \text{Suc } 0$
 $\langle \text{proof} \rangle$

lemma $f\text{-expand-0}[\text{simp}]$: $xs \odot_f 0 = []$
 $\langle \text{proof} \rangle$

corollary $f\text{-expand-0-is-zero-element}$: $xs \odot_f 0 = ys \odot_f 0$
 $\langle \text{proof} \rangle$

lemma $i\text{-expand-0}[\text{simp}]$: $f \odot_i 0 = (\lambda n. \varepsilon)$
 $\langle \text{proof} \rangle$

corollary $i\text{-expand-0-is-zero-element}$: $f \odot_i 0 = g \odot_i 0$
 $\langle \text{proof} \rangle$

lemma $f\text{-expand-gr0-}f\text{-expand-Suc}$: $0 < k \Longrightarrow xs \odot_f k = f\text{-expand-Suc } xs (k - \text{Suc } 0)$
 $\langle \text{proof} \rangle$

lemma $i\text{-expand-gr0-}i\text{-expand-Suc}$: $0 < k \Longrightarrow f \odot_i k = i\text{-expand-Suc } f (k - \text{Suc } 0)$
 $\langle \text{proof} \rangle$

lemma *i-expand-gr0*:

$$0 < k \implies f \odot_i k = (\lambda n. \text{if } n \bmod k = 0 \text{ then } f (n \operatorname{div} k) \text{ else } \varepsilon)$$

<proof>

lemma *f-expand-1[simp]*: $xs \odot_f \operatorname{Suc} 0 = xs$

<proof>

lemma *i-expand-1[simp]*: $f \odot_i \operatorname{Suc} 0 = f$

<proof>

lemma *f-expand-length[simp]*: $\text{length } (xs \odot_f k) = \text{length } xs * k$

<proof>

lemma *f-expand-empty-conv*: $(xs \odot_f k = []) = (xs = [] \vee k = 0)$

<proof>

lemma *f-expand-not-empty-conv*: $(xs \odot_f k \neq []) = (xs \neq [] \wedge 0 < k)$

<proof>

lemma *f-expand-Cons*:

$$0 < k \implies (x \# xs) \odot_f k = x \# \varepsilon^k - \operatorname{Suc} 0 @ (xs \odot_f k)$$

<proof>

lemma *f-expand-append[simp]*: $\bigwedge ys. (xs @ ys) \odot_f k = (xs \odot_f k) @ (ys \odot_f k)$

<proof>

lemma *f-expand-snoc*:

$$0 < k \implies (xs @ [x]) \odot_f k = xs \odot_f k @ x \# \operatorname{replicate} (k - \operatorname{Suc} 0) \varepsilon$$

<proof>

lemma *f-expand-nth-mult*: $\bigwedge n.$

$$\llbracket n < \text{length } xs; 0 < k \rrbracket \implies (xs \odot_f k) ! (n * k) = xs ! n$$

<proof>

lemma *i-expand-nth-mult*: $0 < k \implies (f \odot_i k) (n * k) = f n$

<proof>

lemma *f-expand-nth-if*: $\bigwedge n.$

$$n < \text{length } xs * k \implies$$

$$(xs \odot_f k) ! n = (\text{if } n \bmod k = 0 \text{ then } xs ! (n \operatorname{div} k) \text{ else } \varepsilon)$$

<proof>

corollary *f-expand-nth-mod-eq-0*:

$$\llbracket n < \text{length } xs * k; n \bmod k = 0 \rrbracket \implies (xs \odot_f k) ! n = xs ! (n \operatorname{div} k)$$

<proof>

corollary *f-expand-nth-mod-neq-0*:

$$\llbracket n < \text{length } xs * k; 0 < n \bmod k \rrbracket \implies (xs \odot_f k) ! n = \varepsilon$$

<proof>

lemma *f-expand-nth-0-upto-k-minus-1-if*:

$$\llbracket t < \text{length } xs; n = t * k + i; i < k \rrbracket \implies$$

$$(xs \odot_f k) ! n = (\text{if } i = 0 \text{ then } xs ! t \text{ else } \varepsilon)$$

$\langle proof \rangle$

lemma *f-expand-take-mult*: $xs \odot_f k \downarrow (n * k) = (xs \downarrow n) \odot_f k$
 $\langle proof \rangle$

lemma *f-expand-take-mod*:
 $n \bmod k = 0 \implies xs \odot_f k \downarrow n = xs \downarrow (n \text{ div } k) \odot_f k$
 $\langle proof \rangle$

lemma *f-expand-drop-mult*: $xs \odot_f k \uparrow (n * k) = (xs \uparrow n) \odot_f k$
 $\langle proof \rangle$

lemma *f-expand-drop-mod*:
 $n \bmod k = 0 \implies xs \odot_f k \uparrow n = xs \uparrow (n \text{ div } k) \odot_f k$
 $\langle proof \rangle$

lemma *f-expand-take-mult-Suc*:
 $\llbracket n < \text{length } xs; i < k \rrbracket \implies$
 $xs \odot_f k \downarrow (n * k + \text{Suc } i) = (xs \downarrow n) \odot_f k @ (xs ! n \# \varepsilon^i)$
 $\langle proof \rangle$

lemma *f-expand-take-Suc*:
 $n < \text{length } xs * k \implies$
 $xs \odot_f k \downarrow \text{Suc } n = (xs \downarrow (n \text{ div } k)) \odot_f k @ (xs ! (n \text{ div } k) \# \varepsilon^{n \bmod k})$
 $\langle proof \rangle$

lemma *i-expand-nth-if*:
 $0 < k \implies (f \odot_i k) n = (\text{if } n \bmod k = 0 \text{ then } f (n \text{ div } k) \text{ else } \varepsilon)$
 $\langle proof \rangle$

corollary *i-expand-nth-mod-eq-0*:
 $\llbracket 0 < k; n \bmod k = 0 \rrbracket \implies (f \odot_i k) n = f (n \text{ div } k)$
 $\langle proof \rangle$

corollary *i-expand-nth-mod-neq-0*:
 $0 < n \bmod k \implies (f \odot_i k) n = \varepsilon$
 $\langle proof \rangle$

lemma *i-expand-nth-0-upto-k-minus-1-if*:
 $\llbracket n = t * k + i; i < k \rrbracket \implies$
 $(f \odot_i k) n = (\text{if } i = 0 \text{ then } f t \text{ else } \varepsilon)$
 $\langle proof \rangle$

lemma *i-expand-i-take-mult*: $f \odot_i k \downarrow (n * k) = (f \downarrow n) \odot_f k$
 $\langle proof \rangle$

lemma *i-expand-i-take-mod*:
 $n \bmod k = 0 \implies f \odot_i k \downarrow n = f \downarrow (n \text{ div } k) \odot_f k$
 $\langle proof \rangle$

lemma *i-expand-i-drop-mult*: $(f \odot_i k) \uparrow (n * k) = (f \uparrow n) \odot_i k$
 ⟨proof⟩

lemma *i-expand-i-drop-mod*:
 $n \bmod k = 0 \implies f \odot_i k \uparrow n = f \uparrow (n \text{ div } k) \odot_i k$
 ⟨proof⟩

lemma *i-expand-i-take-mult-Suc*:
 $i < k \implies f \odot_i k \downarrow (n * k + \text{Suc } i) = (f \downarrow n) \odot_f k @ (f n \# \varepsilon^i)$
 ⟨proof⟩

lemma *i-expand-i-take-Suc*:
 $0 < k \implies f \odot_i k \downarrow \text{Suc } n = (f \downarrow (n \text{ div } k)) \odot_f k @ (f (n \text{ div } k) \# \varepsilon^{n \bmod k})$
 ⟨proof⟩

lemma *f-expand-nth-interval-eq-nth-append-replicate-NoMsg*[rule-format]:
 $\llbracket 0 < k; t < \text{length } xs; t * k \leq t1; t1 \leq t * k + k - \text{Suc } 0 \rrbracket \implies$
 $xs \odot_f k \downarrow \text{Suc } t1 \uparrow (t * k) = xs ! t \# \varepsilon^{t1 - t * k}$
 ⟨proof⟩

lemma *f-expand-nth-interval-eq-replicate-NoMsg*:
 $\llbracket 0 < k; t * k < t1; t1 \leq t2; t2 \leq t * k + k; t2 \leq \text{length } xs * k \rrbracket \implies$
 $xs \odot_f k \downarrow t2 \uparrow t1 = \varepsilon^{t2 - t1}$
 ⟨proof⟩

lemma *i-expand-nth-interval-eq-nth-append-replicate-NoMsg*[rule-format]:
 $\llbracket 0 < k; t * k \leq t1; t1 \leq t * k + k - \text{Suc } 0 \rrbracket \implies$
 $f \odot_i k \downarrow \text{Suc } t1 \uparrow (t * k) = f t \# \varepsilon^{t1 - t * k}$
 ⟨proof⟩

lemma *i-expand-nth-interval-eq-replicate-NoMsg*:
 $\llbracket 0 < k; t * k < t1; t1 \leq t2; t2 \leq t * k + k \rrbracket \implies$
 $f \odot_i k \downarrow t2 \uparrow t1 = \varepsilon^{t2 - t1}$
 ⟨proof⟩

lemma *f-expand-replicate-NoMsg*[simp]: $(\varepsilon^n) \odot_f k = \varepsilon^{n * k}$
 ⟨proof⟩

lemma *i-expand-const-NoMsg*[simp]: $(\lambda n. \varepsilon) \odot_i k = (\lambda n. \varepsilon)$
 ⟨proof⟩

lemma *f-expand-assoc*: $xs \odot_f a \odot_f b = xs \odot_f (a * b)$
 ⟨proof⟩

lemma *i-expand-assoc*: $f \odot_i a \odot_i b = f \odot_i (a * b)$
 ⟨proof⟩

lemma *f-expand-commute*: $xs \odot_f a \odot_f b = xs \odot_f b \odot_f a$
 ⟨proof⟩

lemma *i-expand-commute*: $f \odot_i a \odot_i b = f \odot_i b \odot_i a$
 ⟨proof⟩

lemma *i-expand-i-append*: $(xs \frown f) \odot_i k = xs \odot_f k \frown (f \odot_i k)$
 ⟨proof⟩

lemma *f-expand-eq-conv*:
 $0 < k \implies (xs \odot_f k = ys \odot_f k) = (xs = ys)$
 ⟨proof⟩

lemma *i-expand-eq-conv*:
 $0 < k \implies (f \odot_i k = g \odot_i k) = (f = g)$
 ⟨proof⟩

lemma *f-expand-eq-conv'*:
 $(xs' \odot_f k = xs) =$
 $(\text{length } xs' * k = \text{length } xs \wedge$
 $(\forall i < \text{length } xs. xs ! i = (\text{if } i \bmod k = 0 \text{ then } xs' ! (i \text{ div } k) \text{ else } \varepsilon)))$
 ⟨proof⟩

lemma *i-expand-eq-conv'*:
 $0 < k \implies (f' \odot_i k = f) =$
 $(\forall i. f i = (\text{if } i \bmod k = 0 \text{ then } f' (i \text{ div } k) \text{ else } \varepsilon))$
 ⟨proof⟩

2.2.2 Aggregating lists

definition *f-aggregate* :: 'a list \Rightarrow nat \Rightarrow ('a list \Rightarrow 'a) \Rightarrow 'a list
 where *f-aggregate* s k ag \equiv map ag (list-slice s k)

definition *i-aggregate* :: 'a ilist \Rightarrow nat \Rightarrow ('a list \Rightarrow 'a) \Rightarrow 'a ilist
 where *i-aggregate* s k ag \equiv $\lambda n. ag (s \uparrow (n * k) \downarrow k)$

lemma *f-aggregate-0[simp]*: *f-aggregate* xs 0 ag = []
 ⟨proof⟩

lemma *f-aggregate-1*:
 $(\bigwedge x. ag [x] = x) \implies$
 $f\text{-aggregate } xs (Suc 0) ag = xs$
 ⟨proof⟩

lemma *f-aggregate-Nil[simp]*: *f-aggregate* [] k ag = []
 ⟨proof⟩

lemma *f-aggregate-length[simp]*: $\text{length } (f\text{-aggregate } xs k ag) = \text{length } xs \text{ div } k$

$\langle \text{proof} \rangle$

lemma *f-aggregate-empty-conv*:

$$0 < k \implies (f\text{-aggregate } xs \ k \ ag = []) = (length \ xs < k)$$

$\langle \text{proof} \rangle$

lemma *f-aggregate-one*:

$$\llbracket 0 < k; length \ xs = k \rrbracket \implies f\text{-aggregate } xs \ k \ ag = [ag \ xs]$$

$\langle \text{proof} \rangle$

lemma *f-aggregate-Cons*:

$$\llbracket 0 < k; length \ xs = k \rrbracket \implies$$

$$f\text{-aggregate } (xs \ @ \ ys) \ k \ ag = ag \ xs \ \# \ (f\text{-aggregate } ys \ k \ ag)$$

$\langle \text{proof} \rangle$

lemma *f-aggregate-eq-f-aggregate-take*:

$$f\text{-aggregate } (xs \ \downarrow \ (length \ xs \ \text{div} \ k * k)) \ k \ ag = f\text{-aggregate } xs \ k \ ag$$

$\langle \text{proof} \rangle$

lemma *f-aggregate-nth*:

$$n < length \ xs \ \text{div} \ k \implies$$

$$(f\text{-aggregate } xs \ k \ ag) ! n = ag \ (xs \ \uparrow \ (n * k) \ \downarrow \ k)$$

$\langle \text{proof} \rangle$

lemma *f-aggregate-nth-eq-sublist-list*:

$$n < length \ xs \ \text{div} \ k \implies$$

$$(f\text{-aggregate } xs \ k \ ag) ! n = ag \ (sublist\text{-list } xs \ [n * k..<n * k + k])$$

$\langle \text{proof} \rangle$

lemma *f-aggregate-take-nth*:

$$\bigwedge xs \ m. \llbracket n < length \ xs \ \text{div} \ k; n < m \ \text{div} \ k \rrbracket \implies$$

$$f\text{-aggregate } (xs \ \downarrow \ m) \ k \ ag ! n = f\text{-aggregate } xs \ k \ ag ! n$$

$\langle \text{proof} \rangle$

lemma *f-aggregate-hd*:

$$\llbracket 0 < k; k \leq length \ xs \rrbracket \implies$$

$$hd \ (f\text{-aggregate } xs \ k \ ag) = ag \ (xs \ \downarrow \ k)$$

$\langle \text{proof} \rangle$

lemma *f-aggregate-append-mod*:

$$length \ xs \ \text{mod} \ k = 0 \implies$$

$$f\text{-aggregate } (xs \ @ \ ys) \ k \ ag =$$

$$f\text{-aggregate } xs \ k \ ag \ @ \ f\text{-aggregate } ys \ k \ ag$$

$\langle \text{proof} \rangle$

lemma *f-aggregate-append-mult*:

$$length \ xs = m * k \implies$$

$$f\text{-aggregate } (xs \ @ \ ys) \ k \ ag =$$

$$f\text{-aggregate } xs \ k \ ag \ @ \ f\text{-aggregate } ys \ k \ ag$$

$\langle \text{proof} \rangle$

lemma *f-aggregate-snoc*:

$\llbracket 0 < k; \text{length } ys = k; \text{length } xs \text{ mod } k = 0 \rrbracket \implies$
 $f\text{-aggregate } (xs @ ys) k ag = f\text{-aggregate } xs k ag @ [ag ys]$
 ⟨proof⟩

lemma *f-aggregate-take*:

$f\text{-aggregate } (xs \downarrow n) k ag = f\text{-aggregate } xs k ag \downarrow (n \text{ div } k)$
 ⟨proof⟩

lemma *f-aggregate-take-mult*:

$f\text{-aggregate } (xs \downarrow (n * k)) k ag = f\text{-aggregate } xs k ag \downarrow n$
 ⟨proof⟩

lemma *f-aggregate-drop-mult*:

$f\text{-aggregate } (xs \uparrow (n * k)) k ag = f\text{-aggregate } xs k ag \uparrow n$
 ⟨proof⟩

lemma *f-aggregate-drop-mod*:

$n \text{ mod } k = 0 \implies f\text{-aggregate } (xs \uparrow n) k ag = f\text{-aggregate } xs k ag \uparrow (n \text{ div } k)$
 ⟨proof⟩

lemma *f-aggregate-assoc*:

$(\bigwedge xs. \text{length } xs \text{ mod } a = 0 \implies ag (f\text{-aggregate } xs a ag) = ag xs) \implies$
 $f\text{-aggregate } (f\text{-aggregate } xs a ag) b ag = f\text{-aggregate } xs (a * b) ag$
 ⟨proof⟩

lemma *f-aggregate-commute*:

$\llbracket \bigwedge xs. \text{length } xs \text{ mod } a = 0 \implies ag (f\text{-aggregate } xs a ag) = ag xs;$
 $\bigwedge xs. \text{length } xs \text{ mod } b = 0 \implies ag (f\text{-aggregate } xs b ag) = ag xs \rrbracket \implies$
 $f\text{-aggregate } (f\text{-aggregate } xs a ag) b ag = f\text{-aggregate } (f\text{-aggregate } xs b ag) a ag$
 ⟨proof⟩

lemma *i-aggregate-0[simp]*: $i\text{-aggregate } f 0 ag = (\lambda x. ag [])$

⟨proof⟩

lemma *i-aggregate-1*: $(\bigwedge x. ag [x] = x) \implies i\text{-aggregate } f (Suc 0) ag = f$

⟨proof⟩

lemma *i-aggregate-nth*: $i\text{-aggregate } f k ag n = ag (f \uparrow (n * k) \downarrow k)$

⟨proof⟩

lemma *i-aggregate-hd*: $i\text{-aggregate } f k ag 0 = ag (f \downarrow k)$

⟨proof⟩

lemma *i-aggregate-nth-eq-map*: $i\text{-aggregate } f k ag n = ag (\text{map } f [n * k..<n * k + k])$

⟨proof⟩

lemma *i-aggregate-i-append-mod*:

$\text{length } xs \text{ mod } k = 0 \implies$
 $i\text{-aggregate } (xs \frown f) k ag = f\text{-aggregate } xs k ag \frown i\text{-aggregate } f k ag$
 ⟨proof⟩

lemma *i-aggregate-i-append-mult*:

$length\ xs = m * k \implies$
 $i-aggregate\ (xs \frown f)\ k\ ag = f-aggregate\ xs\ k\ ag \frown i-aggregate\ f\ k\ ag$
 ⟨proof⟩

lemma *i-aggregate-Cons*:

$\llbracket 0 < k; length\ xs = k \rrbracket \implies$
 $i-aggregate\ (xs \frown f)\ k\ ag = [ag\ xs] \frown (i-aggregate\ f\ k\ ag)$
 ⟨proof⟩

lemma *i-aggregate-take-nth*:

$n < m\ div\ k \implies f-aggregate\ (f\ \Downarrow\ m)\ k\ ag\ !\ n = i-aggregate\ f\ k\ ag\ n$
 ⟨proof⟩

lemma *i-aggregate-i-take*:

$f-aggregate\ (f\ \Downarrow\ n)\ k\ ag = i-aggregate\ f\ k\ ag\ \Downarrow\ (n\ div\ k)$
 ⟨proof⟩

lemma *i-aggregate-i-take-mult*:

$0 < k \implies f-aggregate\ (f\ \Downarrow\ (n * k))\ k\ ag = i-aggregate\ f\ k\ ag\ \Downarrow\ n$
 ⟨proof⟩

lemma *i-aggregate-i-drop-mult*:

$i-aggregate\ (f\ \Uparrow\ (n * k))\ k\ ag = i-aggregate\ f\ k\ ag\ \Uparrow\ n$
 ⟨proof⟩

lemma *i-aggregate-i-drop-mod*:

$n\ mod\ k = 0 \implies$
 $i-aggregate\ (f\ \Uparrow\ n)\ k\ ag = i-aggregate\ f\ k\ ag\ \Uparrow\ (n\ div\ k)$
 ⟨proof⟩

lemma *i-aggregate-assoc*:

$\llbracket 0 < a; 0 < b; \bigwedge xs. length\ xs\ mod\ a = 0 \implies ag\ (f-aggregate\ xs\ a\ ag) = ag\ xs \rrbracket \implies$
 $i-aggregate\ (i-aggregate\ f\ a\ ag)\ b\ ag = i-aggregate\ f\ (a * b)\ ag$
 ⟨proof⟩

lemma *i-aggregate-commute*:

$\llbracket 0 < a; 0 < b; \bigwedge xs. length\ xs\ mod\ a = 0 \implies ag\ (f-aggregate\ xs\ a\ ag) = ag\ xs; \bigwedge xs. length\ xs\ mod\ b = 0 \implies ag\ (f-aggregate\ xs\ b\ ag) = ag\ xs \rrbracket \implies$
 $i-aggregate\ (i-aggregate\ xs\ a\ ag)\ b\ ag = i-aggregate\ (i-aggregate\ xs\ b\ ag)\ a\ ag$
 ⟨proof⟩

2.2.3 Compressing message streams

Determines the last non-empty message.

primrec *last-message* :: 'a fstream-af \Rightarrow 'a message-af

where

$last_message [] = \varepsilon$
 $| last_message (x \# xs) = (if\ last_message\ xs = \varepsilon\ then\ x\ else\ last_message\ xs)$

definition $f_shrink :: 'a\ fstream\text{-}af \Rightarrow nat \Rightarrow 'a\ fstream\text{-}af$ (**infixl** \div_f 100)
where $f_shrink\ xs\ k \equiv f_aggregate\ xs\ k\ last_message$

definition $i_shrink :: 'a\ istream\text{-}af \Rightarrow nat \Rightarrow 'a\ istream\text{-}af$ (**infixl** \div_i 100)
where $i_shrink\ f\ k \equiv i_aggregate\ f\ k\ last_message$

notation

f_shrink (**infixl** \div 100) **and**
 i_shrink (**infixl** \div 100)

lemmas $f_shrink\text{-}defs = f_shrink\text{-}def\ f_aggregate\text{-}def$

lemmas $i_shrink\text{-}defs = i_shrink\text{-}def\ i_aggregate\text{-}def$

lemma $last_message\text{-}Nil$: $last_message [] = \varepsilon$
 $\langle proof \rangle$

lemma $last_message\text{-}one$: $last_message [m] = m$
 $\langle proof \rangle$

lemma $last_message\text{-}replicate$: $0 < n \implies last_message (m^n) = m$
 $\langle proof \rangle$

lemma $last_message\text{-}replicate\text{-}NoMsg$: $last_message (\varepsilon^n) = \varepsilon$
 $\langle proof \rangle$

lemma $last_message\text{-}Cons\text{-}NoMsg$: $last_message (\varepsilon \# xs) = last_message xs$
 $\langle proof \rangle$

lemma $last_message\text{-}append\text{-}one$:
 $last_message (xs @ [m]) = (if\ m = \varepsilon\ then\ last_message\ xs\ else\ m)$
 $\langle proof \rangle$

lemma $last_message\text{-}append$: $\bigwedge xs.$
 $last_message (xs @ ys) =$
 $if\ last_message\ ys = \varepsilon\ then\ last_message\ xs\ else\ last_message\ ys$
 $\langle proof \rangle$

corollary $last_message\text{-}append\text{-}replicate\text{-}NoMsg$:

$last_message (xs @ \varepsilon^n) = last_message xs$
 $\langle proof \rangle$

lemma $last_message\text{-}replicate\text{-}NoMsg\text{-}append$:

$last_message (\varepsilon^n @ xs) = last_message xs$

$\langle \text{proof} \rangle$

lemma *last-message-NoMsg-conv*:

$$(\text{last-message } xs = \varepsilon) = (\forall i < \text{length } xs. xs ! i = \varepsilon)$$

$\langle \text{proof} \rangle$

lemma *last-message-not-NoMsg-conv*:

$$(\text{last-message } xs \neq \varepsilon) = (\exists i < \text{length } xs. xs ! i \neq \varepsilon)$$

$\langle \text{proof} \rangle$

lemma *not-NoMsg-imp-last-message*:

$$\llbracket i < \text{length } xs; xs ! i \neq \varepsilon \rrbracket \implies \text{last-message } xs \neq \varepsilon$$

$\langle \text{proof} \rangle$

lemma *last-message-exists-nth*:

$$\text{last-message } xs \neq \varepsilon \implies$$

$$\exists i < \text{length } xs. \text{last-message } xs = xs ! i \wedge (\forall j < \text{length } xs. i < j \longrightarrow xs ! j = \varepsilon)$$

$\langle \text{proof} \rangle$

lemma *last-message-exists-nth'*:

$$\text{last-message } xs \neq \varepsilon \implies \exists i < \text{length } xs. \text{last-message } xs = xs ! i$$

$\langle \text{proof} \rangle$

lemma *last-messageI2-aux*: $\bigwedge i.$

$$\llbracket i < \text{length } xs; xs ! i \neq \varepsilon;$$

$$\forall j. i < j \wedge j < \text{length } xs \longrightarrow xs ! j = \varepsilon \rrbracket \implies$$

$$\text{last-message } xs = xs ! i$$

$\langle \text{proof} \rangle$

lemma *last-messageI2*:

$$\llbracket i < \text{length } xs; xs ! i \neq \varepsilon;$$

$$\bigwedge j. \llbracket i < j; j < \text{length } xs \rrbracket \implies xs ! j = \varepsilon \rrbracket \implies$$

$$\text{last-message } xs = xs ! i$$

$\langle \text{proof} \rangle$

lemma *last-messageI*:

$$\llbracket m \neq \varepsilon; i < \text{length } xs; xs ! i = m;$$

$$\bigwedge j. \llbracket i < j; j < \text{length } xs \rrbracket \implies xs ! j = \varepsilon \rrbracket \implies$$

$$\text{last-message } xs = m$$

$\langle \text{proof} \rangle$

lemma *last-message-Msg-eq-last*:

$$\llbracket xs \neq []; \text{last } xs \neq \varepsilon \rrbracket \implies \text{last-message } xs = \text{last } xs$$

$\langle \text{proof} \rangle$

lemma *last-message-conv*:

$$m \neq \varepsilon \implies$$

$$(\text{last-message } xs = m) =$$

$$(\exists i < \text{length } xs. xs ! i = m \wedge (\forall j < \text{length } xs. i < j \longrightarrow xs ! j = \varepsilon))$$

$\langle \text{proof} \rangle$

lemma *last-message-conv-if*:

$(\text{last-message } xs = m) =$
 $(\text{if } m = \varepsilon \text{ then } \forall i < \text{length } xs. xs ! i = \varepsilon$
 $\text{else } \exists i < \text{length } xs. xs ! i = m \wedge (\forall j < \text{length } xs. i < j \longrightarrow xs ! j = \varepsilon))$
 $\langle \text{proof} \rangle$

lemma *last-message-not-NoMsg-eq-conv*:

$\llbracket \text{last-message } xs \neq \varepsilon; \text{last-message } ys \neq \varepsilon \rrbracket \implies$
 $(\text{last-message } xs = \text{last-message } ys) =$
 $(\exists i j. i < \text{length } xs \wedge j < \text{length } ys \wedge xs ! i \neq \varepsilon \wedge$
 $ys ! i = ys ! j \wedge$
 $(\forall n < \text{length } xs. i < n \longrightarrow xs ! n = \varepsilon) \wedge$
 $(\forall n < \text{length } ys. j < n \longrightarrow ys ! n = \varepsilon))$
 $\langle \text{proof} \rangle$

lemma *f-shrink-0[simp]*: $xs \div_f 0 = []$

$\langle \text{proof} \rangle$

lemma *f-shrink-1[simp]*: $xs \div_f \text{Suc } 0 = xs$

$\langle \text{proof} \rangle$

lemma *f-shrink-Nil[simp]*: $[] \div_f k = []$

$\langle \text{proof} \rangle$

lemma *f-shrink-length*: $\text{length } (xs \div_f k) = \text{length } xs \text{ div } k$

$\langle \text{proof} \rangle$

lemma *f-shrink-empty-conv*: $0 < k \implies (xs \div_f k = []) = (\text{length } xs < k)$

$\langle \text{proof} \rangle$

lemma *f-shrink-Cons*:

$\llbracket 0 < k; \text{length } xs = k \rrbracket \implies (xs @ ys) \div_f k = \text{last-message } xs \# (ys \div_f k)$
 $\langle \text{proof} \rangle$

lemma *f-shrink-one*:

$\llbracket 0 < k; \text{length } xs = k \rrbracket \implies xs \div_f k = [\text{last-message } xs]$
 $\langle \text{proof} \rangle$

lemma *f-shrink-eq-f-shrink-take*:

$xs \downarrow (\text{length } xs \text{ div } k * k) \div_f k = xs \div_f k$
 $\langle \text{proof} \rangle$

lemma *f-shrink-nth*:

$n < \text{length } xs \text{ div } k \implies$
 $(xs \div_f k) ! n = \text{last-message } (xs \uparrow (n * k) \downarrow k)$
 $\langle \text{proof} \rangle$

lemma *f-shrink-nth-eq-sublist-list*:

$n < \text{length } xs \text{ div } k \implies$
 $(xs \div_f k) ! n = \text{last-message } (\text{sublist-list } xs [n * k .. < n * k + k])$
 ⟨proof⟩

lemma *f-shrink-take-nth*:

$\llbracket n < \text{length } xs \text{ div } k; n < m \text{ div } k \rrbracket \implies (xs \downarrow m) \div_f k ! n = xs \div_f k ! n$
 ⟨proof⟩

lemma *f-shrink-hd*:

$\llbracket 0 < k; k \leq \text{length } xs \rrbracket \implies \text{hd } (xs \div_f k) = \text{last-message } (xs \downarrow k)$
 ⟨proof⟩

lemma *f-shrink-append-mod*:

$\text{length } xs \text{ mod } k = 0 \implies (xs @ ys) \div_f k = xs \div_f k @ (ys \div_f k)$
 ⟨proof⟩

lemma *f-shrink-append-mult*:

$\text{length } xs = m * k \implies (xs @ ys) \div_f k = xs \div_f k @ (ys \div_f k)$
 ⟨proof⟩

lemma *f-shrink-snoc*:

$\llbracket 0 < k; \text{length } ys = k; \text{length } xs \text{ mod } k = 0 \rrbracket \implies$
 $(xs @ ys) \div_f k = xs \div_f k @ [\text{last-message } ys]$
 ⟨proof⟩

lemma *f-shrink-last-message[rule-format]*:

$\text{length } xs \text{ mod } k = 0 \longrightarrow \text{last-message } (xs \div_f k) = \text{last-message } xs$
 ⟨proof⟩

lemma *f-shrink-replicate*: $m^n \div_f k = m^n \text{ div } k$

⟨proof⟩

lemma *f-shrink-f-expand-id*: $0 < k \implies xs \odot_f k \div_f k = xs$

⟨proof⟩

lemma *f-expand-f-shrink-id-take[rule-format]*:

$\llbracket \forall i < \text{length } xs. 0 < i \text{ mod } k \longrightarrow xs ! i = \varepsilon \rrbracket \implies$
 $xs \div_f k \odot_f k = xs \downarrow (\text{length } xs \text{ div } k * k)$
 ⟨proof⟩

corollary *f-expand-f-shrink-id-mod-0*:

$\llbracket \text{length } xs \text{ mod } k = 0;$
 $\bigwedge i. \llbracket i < \text{length } xs; 0 < i \text{ mod } k \rrbracket \implies xs ! i = \varepsilon \rrbracket \implies$
 $xs \div_f k \odot_f k = xs$
 ⟨proof⟩

lemma *f-shrink-take*:

$xs \downarrow n \div_f k = xs \div_f k \downarrow (n \text{ div } k)$
 ⟨proof⟩

lemma *f-shrink-take-mult*: $xs \downarrow (n * k) \div_f k = xs \div_f k \downarrow n$
 ⟨proof⟩

lemma *f-shrink-drop-mult*: $xs \uparrow (n * k) \div_f k = xs \div_f k \uparrow n$
 ⟨proof⟩

lemma *f-shrink-drop-mod*:
 $n \text{ mod } k = 0 \implies xs \uparrow n \div_f k = xs \div_f k \uparrow (n \text{ div } k)$
 ⟨proof⟩

lemma *f-shrink-eq-conv*:
 $(xs \div_f k1 = ys \div_f k2) =$
 $(\text{length } xs \text{ div } k1 = \text{length } ys \text{ div } k2 \wedge$
 $(\forall i < \text{length } xs \text{ div } k1.$
 $\text{last-message } (xs \uparrow (i * k1) \downarrow k1) = \text{last-message } (ys \uparrow (i * k2) \downarrow k2)))$
 ⟨proof⟩

lemma *f-shrink-eq-conv'*:
 $(xs' \div_f k = xs) =$
 $(\text{length } xs' \text{ div } k = \text{length } xs \wedge$
 $(\forall i < \text{length } xs.$
 $\text{if } xs ! i = \varepsilon \text{ then } (\forall j < k. xs' ! (i * k + j) = \varepsilon)$
 $\text{else } (\exists n < k. xs' ! (i * k + n) = xs ! i \wedge$
 $(\forall j < k. n < j \implies xs' ! (i * k + j) = \varepsilon))))$
 ⟨proof⟩

lemma *f-shrink-assoc*: $xs \div_f a \div_f b = xs \div_f (a * b)$
 ⟨proof⟩

lemma *f-shrink-commute*: $xs \div_f a \div_f b = xs \div_f b \div_f a$
 ⟨proof⟩

lemma *i-shrink-0[simp]*: $f \div_i 0 = (\lambda n. \varepsilon)$
 ⟨proof⟩

lemma *i-shrink-1[simp]*: $f \div_i \text{Suc } 0 = f$
 ⟨proof⟩

lemma *i-shrink-nth*: $(f \div_i k) n = \text{last-message } (f \uparrow (n * k) \downarrow k)$
 ⟨proof⟩

lemma *i-shrink-nth-eq-map*: $(f \div_i k) n = \text{last-message } (\text{map } f [n * k..<n * k + k])$
 ⟨proof⟩

lemma *i-shrink-hd*: $(f \div_i k) 0 = \text{last-message } (f \downarrow k)$
 ⟨proof⟩

lemma *i-shrink-i-append-mod*:
 $\text{length } xs \text{ mod } k = 0 \implies (xs \frown f) \div_i k = xs \div_f k \frown (f \div_i k)$

<proof>

lemma *i-shrink-i-append-mult*:

$$\text{length } xs = m * k \implies (xs \frown f) \div_i k = xs \div_f k \frown (f \div_i k)$$

<proof>

lemma *i-shrink-Cons*:

$$\llbracket 0 < k; \text{length } xs = k \rrbracket \implies (xs \frown f) \div_i k = [\text{last-message } xs] \frown (f \div_i k)$$

<proof>

lemma *i-shrink-take-nth*:

$$n < m \text{ div } k \implies (f \Downarrow m) \div_f k ! n = (f \div_i k) n$$

<proof>

lemma *i-shrink-const[simp]*: $0 < k \implies (\lambda x. m) \div_i k = (\lambda x. m)$

<proof>

lemma *i-shrink-const-NoMsg[simp]*: $(\lambda x. \varepsilon) \div_i k = (\lambda x. \varepsilon)$

<proof>

lemma *i-shrink-i-expand-id*: $0 < k \implies f \odot_i k \div_i k = f$

<proof>

lemma *i-shrink-i-take-mult*: $0 < k \implies f \Downarrow (n * k) \div_f k = f \div_i k \Downarrow n$

<proof>

lemma *i-shrink-i-take*:

$$f \Downarrow n \div_f k = f \div_i k \Downarrow (n \text{ div } k)$$

<proof>

lemma *i-shrink-i-drop-mult*: $f \Uparrow (n * k) \div_i k = f \div_i k \Uparrow n$

<proof>

lemma *i-shrink-i-drop-mod*:

$$n \bmod k = 0 \implies f \Uparrow n \div_i k = f \div_i k \Uparrow (n \text{ div } k)$$

<proof>

lemma *i-shrink-eq-conv*:

$$\begin{aligned} (f \div_i k1 = g \div_i k2) = \\ (\forall i. \text{last-message } (f \Uparrow (i * k1) \Downarrow k1) = \\ \text{last-message } (g \Uparrow (i * k2) \Downarrow k2)) \end{aligned}$$

<proof>

lemma *i-shrink-eq-conv'*:

$$\begin{aligned} (f' \div_i k = f) = \\ (\forall i. \text{if } f i = \varepsilon \text{ then } \forall j < k. f' (i * k + j) = \varepsilon \\ \text{else } \exists n < k. f' (i * k + n) = f i \wedge \\ (\forall j < k. n < j \longrightarrow f' (i * k + j) = \varepsilon)) \end{aligned}$$

<proof>

lemma *i-shrink-assoc*: $f \div_i a \div_i b = f \div_i (a * b)$
 ⟨proof⟩

lemma *i-shrink-commute*: $f \div_i a \div_i b = f \div_i b \div_i a$
 ⟨proof⟩

2.2.4 Holding last messages in everly cycle of a stream

primrec *last-message-hold-init* :: 'a fstream-af \Rightarrow 'a message-af \Rightarrow 'a fstream-af
where

last-message-hold-init [] $m = []$
 | *last-message-hold-init* (x # xs) $m =$
 (if x = ε then m else x) #
 (*last-message-hold-init* xs (if x = ε then m else x))

definition *last-message-hold* :: 'a fstream-af \Rightarrow 'a fstream-af
where *last-message-hold* xs \equiv *last-message-hold-init* xs ε

lemma *last-message-hold-init-length[simp]*:
 $\bigwedge m. \text{length } (\text{last-message-hold-init } xs \ m) = \text{length } xs$
 ⟨proof⟩

lemma *last-message-hold-init-nth*:
 $\bigwedge i \ m. i < \text{length } xs \implies$
 (*last-message-hold-init* xs m) ! $i = \text{last-message } (m \ \# \ xs \ \downarrow \ \text{Suc } i)$
 ⟨proof⟩

lemma *last-message-hold-init-snoc*:
last-message-hold-init (xs @ [x]) $m =$
last-message-hold-init xs m @
 [if x = ε then *last-message* (m # xs) else x]
 ⟨proof⟩

lemma *last-message-hold-init-append[rule-format]*:
 $\bigwedge xs \ m. \text{last-message-hold-init } (xs \ @ \ ys) \ m =$
last-message-hold-init xs m @ *last-message-hold-init* ys (*last-message* (m # xs))
 ⟨proof⟩

lemma *last-message-hold-length[simp]*: $\text{length } (\text{last-message-hold } xs) = \text{length } xs$
 ⟨proof⟩

lemma *last-message-hold-Nil[simp]*: *last-message-hold* [] = []
 ⟨proof⟩

lemma *last-message-hold-one[simp]*: *last-message-hold* [x] = [x]
 ⟨proof⟩

lemma *last-message-hold-nth*:
 $i < \text{length } xs \implies \text{last-message-hold } xs \ ! \ i = \text{last-message } (xs \ \downarrow \ \text{Suc } i)$

<proof>

lemma *last-message-hold-last*:

$$xs \neq [] \implies \text{last } (\text{last-message-hold } xs) = \text{last-message } xs$$

<proof>

lemma *last-message-hold-take*:

$$\text{last-message-hold } xs \downarrow n = \text{last-message-hold } (xs \downarrow n)$$

<proof>

lemma *last-message-hold-snoc*:

$$\begin{aligned} \text{last-message-hold } (xs @ [x]) &= \\ \text{last-message-hold } xs @ [\text{if } x = \varepsilon \text{ then last-message } xs \text{ else } x] & \end{aligned}$$

<proof>

lemma *last-message-hold-append*:

$$\begin{aligned} \text{last-message-hold } (xs @ ys) &= \\ \text{last-message-hold } xs @ \text{last-message-hold-init } ys (\text{last-message } xs) & \end{aligned}$$

<proof>

lemma *last-message-hold-append'*:

$$\begin{aligned} \text{last-message-hold } (xs @ ys) &= \\ \text{last-message-hold } xs @ \text{tl } (\text{last-message-hold } (\text{last-message } xs \# ys)) & \end{aligned}$$

<proof>

lemma *last-message-last-message-hold[simp]*:

$$\text{last-message } (\text{last-message-hold } xs) = \text{last-message } xs$$

<proof>

lemma *last-message-hold-idem[simp]*:

$$\text{last-message-hold } (\text{last-message-hold } xs) = \text{last-message-hold } xs$$

<proof>

Returns for each point in time the currently last non-empty message of the current stream cycle of length k .

definition *f-last-message-hold* :: 'a fstream-af \Rightarrow nat \Rightarrow 'a fstream-af (**infixl** \mapsto_f 100)

where *f-last-message-hold* $xs\ k \equiv \text{concat } (\text{map } \text{last-message-hold } (\text{list-slice2 } xs\ k))$

definition *i-last-message-hold* :: 'a istream-af \Rightarrow nat \Rightarrow 'a istream-af (**infixl** \mapsto_i 100)

where *i-last-message-hold* $f\ k \equiv \lambda n. \text{last-message } (f \uparrow (n - n \bmod k) \downarrow \text{Suc } (n \bmod k))$

notation

f-last-message-hold (**infixl** \mapsto 100) **and**

i-last-message-hold (**infixl** \mapsto 100)

lemma *f-last-message-hold-0[simp]*: $xs \mapsto_f 0 = \text{last-message-hold } xs$
 ⟨proof⟩

lemma *f-last-message-hold-1[simp]*: $xs \mapsto_f (\text{Suc } 0) = xs$
 ⟨proof⟩

lemma *f-last-message-hold-Nil[simp]*: $[] \mapsto_f k = []$
 ⟨proof⟩

lemma *f-last-message-hold-length[simp]*: $\text{length } (xs \mapsto_f k) = \text{length } xs$
 ⟨proof⟩

lemma *f-last-message-hold-le*: $\text{length } xs \leq k \implies xs \mapsto_f k = \text{last-message-hold } xs$
 ⟨proof⟩

lemma *f-last-message-hold-append-mult*:
 $\text{length } xs = m * k \implies (xs @ ys) \mapsto_f k = xs \mapsto_f k @ (ys \mapsto_f k)$
 ⟨proof⟩

lemma *f-last-message-hold-append-mod*:
 $\text{length } xs \bmod k = 0 \implies (xs @ ys) \mapsto_f k = xs \mapsto_f k @ (ys \mapsto_f k)$
 ⟨proof⟩

lemma *f-last-message-hold-nth[rule-format]*:
 $\forall n. n < \text{length } xs \implies xs \mapsto_f k ! n = \text{last-message } (xs \uparrow (n \text{ div } k * k) \downarrow \text{Suc } (n \text{ mod } k))$
 ⟨proof⟩

lemma *f-last-message-hold-take*: $xs \downarrow n \mapsto_f k = xs \mapsto_f k \downarrow n$
 ⟨proof⟩

lemma *f-last-message-hold-drop-mult*:
 $xs \uparrow (n * k) \mapsto_f k = xs \mapsto_f k \uparrow (n * k)$
 ⟨proof⟩

lemma *f-last-message-hold-drop-mod*:
 $n \bmod k = 0 \implies xs \uparrow n \mapsto_f k = xs \mapsto_f k \uparrow n$
 ⟨proof⟩

lemma *f-last-message-hold-idem*: $xs \mapsto_f k \mapsto_f k = xs \mapsto_f k$
 ⟨proof⟩

lemma *f-shrink-nth-eq-f-last-message-hold-last*:
 $n < \text{length } xs \text{ div } k \implies xs \div_f k ! n = \text{last } (xs \mapsto_f k \uparrow (n * k) \downarrow k)$
 ⟨proof⟩

lemma *f-shrink-nth-eq-f-last-message-hold-nth*:
 $n < \text{length } xs \text{ div } k \implies xs \div_f k ! n = xs \mapsto_f k ! (n * k + k - \text{Suc } 0)$
 ⟨proof⟩

lemma *last-message-f-last-message-hold*:

$$\text{last-message } (xs \mapsto_f k) = \text{last-message } xs$$

<proof>

lemma *i-last-message-hold-0[simp]*: $f \mapsto_i 0 = (\lambda n. \text{last-message } (f \Downarrow \text{Suc } n))$

<proof>

lemma *i-last-message-hold-1[simp]*: $f \mapsto_i \text{Suc } 0 = f$

<proof>

lemma *i-last-message-hold-nth*:

$$(f \mapsto_i k) n = \text{last-message } (f \Uparrow (n - n \bmod k) \Downarrow \text{Suc } (n \bmod k))$$

<proof>

lemma *i-last-message-hold-i-append-mult*:

$$\text{length } xs = m * k \implies (xs \frown f) \mapsto_i k = (xs \mapsto_f k) \frown (f \mapsto_i k)$$

<proof>

lemma *i-last-message-hold-i-append-mod*:

$$\text{length } xs \bmod k = 0 \implies (xs \frown f) \mapsto_i k = (xs \mapsto_f k) \frown (f \mapsto_i k)$$

<proof>

lemma *i-last-message-hold-i-take*: $f \Downarrow n \mapsto_f k = (f \mapsto_i k) \Downarrow n$

<proof>

lemma *i-last-message-hold-i-drop-mult*:

$$f \Uparrow (n * k) \mapsto_i k = f \mapsto_i k \Uparrow (n * k)$$

<proof>

lemma *i-last-message-hold-i-drop-mod*:

$$n \bmod k = 0 \implies f \Uparrow n \mapsto_i k = f \mapsto_i k \Uparrow n$$

<proof>

lemma *i-last-message-hold-idem*: $f \mapsto_i k \mapsto_i k = f \mapsto_i k$

<proof>

lemma *i-shrink-nth-eq-i-last-message-hold-nth*:

$$0 < k \implies (f \div_i k) n = (f \mapsto_i k) (n * k + k - \text{Suc } 0)$$

<proof>

lemma *i-shrink-nth-eq-i-last-message-hold-last*:

$$0 < k \implies (f \div_i k) n = \text{last } (f \mapsto_i k \Uparrow (n * k) \Downarrow k)$$

<proof>

2.2.5 Compressing lists

Lists/Non-message streams do not have to permit the empty message ε to be element. Thus, they are compressed by factor k by just aggregating every sequence of length k to its last element.

definition $f\text{-shrink-last} :: 'a \text{ list} \Rightarrow \text{nat} \Rightarrow 'a \text{ list}$ (**infixl** \div_{fl} 100)
where $f\text{-shrink-last } xs \ k \equiv f\text{-aggregate } xs \ k \ \text{last}$

definition $i\text{-shrink-last} :: 'a \ \text{ilist} \Rightarrow \text{nat} \Rightarrow 'a \ \text{ilist}$ (**infixl** \div_{il} 100)
where $i\text{-shrink-last } f \ k \equiv i\text{-aggregate } f \ k \ \text{last}$

notation

$f\text{-shrink-last}$ (**infixl** \div_l 100) **and**
 $i\text{-shrink-last}$ (**infixl** \div_l 100)

lemma $f\text{-shrink-last-0}[simp]$: $xs \div_{fl} 0 = []$
 $\langle proof \rangle$

lemma $f\text{-shrink-last-1}[simp]$: $xs \div_{fl} \text{Suc } 0 = xs$
 $\langle proof \rangle$

lemma $f\text{-shrink-last-Nil}[simp]$: $[] \div_{fl} k = []$
 $\langle proof \rangle$

lemma $f\text{-shrink-last-length}$: $\text{length } (xs \div_{fl} k) = \text{length } xs \ \text{div } k$
 $\langle proof \rangle$

lemma $f\text{-shrink-last-empty-conv}$:
 $0 < k \implies (xs \div_{fl} k = []) = (\text{length } xs < k)$
 $\langle proof \rangle$

lemma $f\text{-shrink-last-Cons}$:
 $\llbracket 0 < k;$
 $\text{length } xs = k \rrbracket \implies (xs @ ys) \div_{fl} k = \text{last } xs \ \# \ (ys \div_{fl} k)$
 $\langle proof \rangle$

lemma $f\text{-shrink-last-one}$:
 $\llbracket 0 < k; \text{length } xs = k \rrbracket \implies xs \div_{fl} k = [\text{last } xs]$
 $\langle proof \rangle$

lemma $f\text{-shrink-last-eq-f-shrink-last-take}$:
 $xs \downarrow (\text{length } xs \ \text{div } k * k) \div_{fl} k = xs \div_{fl} k$
 $\langle proof \rangle$

lemma $f\text{-shrink-last-nth}$:
 $n < \text{length } xs \ \text{div } k \implies (xs \div_{fl} k) ! n = xs ! (n * k + k - \text{Suc } 0)$
 $\langle proof \rangle$

corollary $f\text{-shrink-last-nth}'$:
 $n < \text{length } xs \ \text{div } k \implies (xs \div_{fl} k) ! n = xs ! (\text{Suc } n * k - \text{Suc } 0)$
 $\langle proof \rangle$

lemma *f-shrink-last-hd*:

$$\llbracket 0 < k; k \leq \text{length } xs \rrbracket \implies \text{hd } (xs \div_{\text{fl}} k) = xs ! (k - \text{Suc } 0)$$

<proof>

lemma *f-shrink-last-map*: $(\text{map } f \text{ } xs) \div_{\text{fl}} k = \text{map } f (xs \div_{\text{fl}} k)$

<proof>

lemma *f-shrink-last-append-mod*:

$$\text{length } xs \bmod k = 0 \implies (xs @ ys) \div_{\text{fl}} k = xs \div_{\text{fl}} k @ (ys \div_{\text{fl}} k)$$

<proof>

lemma *f-shrink-last-append-mult*:

$$\text{length } xs = m * k \implies (xs @ ys) \div_{\text{fl}} k = xs \div_{\text{fl}} k @ (ys \div_{\text{fl}} k)$$

<proof>

lemma *f-shrink-last-snoc*:

$$\llbracket 0 < k; \text{length } ys = k; \text{length } xs \bmod k = 0 \rrbracket \implies$$

$$(xs @ ys) \div_{\text{fl}} k = xs \div_{\text{fl}} k @ [\text{last } ys]$$

<proof>

lemma *f-shrink-last-last*:

$$\text{length } xs \bmod k = 0 \implies \text{last } (xs \div_{\text{fl}} k) = \text{last } xs$$

<proof>

lemma *f-shrink-last-replicate*: $m^n \div_{\text{fl}} k = m^n \text{ div } k$

<proof>

lemma *f-shrink-last-take*:

$$xs \downarrow n \div_{\text{fl}} k = xs \div_{\text{fl}} k \downarrow (n \text{ div } k)$$

<proof>

lemma *f-shrink-last-take-mult*: $xs \downarrow (n * k) \div_{\text{fl}} k = xs \div_{\text{fl}} k \downarrow n$

<proof>

lemma *f-shrink-last-drop-mult*: $xs \uparrow (n * k) \div_{\text{fl}} k = xs \div_{\text{fl}} k \uparrow n$

<proof>

lemma *f-shrink-last-drop-mod*:

$$n \bmod k = 0 \implies xs \uparrow n \div_{\text{fl}} k = xs \div_{\text{fl}} k \uparrow (n \text{ div } k)$$

<proof>

lemma *f-shrink-last-assoc*: $xs \div_{\text{fl}} a \div_{\text{fl}} b = xs \div_{\text{fl}} (a * b)$

<proof>

lemma *f-shrink-last-commute*: $xs \div_{\text{fl}} a \div_{\text{fl}} b = xs \div_{\text{fl}} b \div_{\text{fl}} a$

<proof>

lemma *i-shrink-last-1*[simp]: $f \div_{il} \text{Suc } 0 = f$
 ⟨proof⟩

lemma *i-shrink-last-nth*: $0 < k \implies (f \div_{il} k) n = f (n * k + k - \text{Suc } 0)$
 ⟨proof⟩

lemma *i-shrink-last-nth'*: $0 < k \implies (f \div_{il} k) n = f (\text{Suc } n * k - \text{Suc } 0)$
 ⟨proof⟩

lemma *i-shrink-last-hd*: $(f \div_{il} k) 0 = \text{last } (f \Downarrow k)$
 ⟨proof⟩

lemma *i-shrink-last-o*: $0 < k \implies (f \circ g) \div_{il} k = f \circ (g \div_{il} k)$
 ⟨proof⟩

lemma *i-shrink-last-i-append-mod*:
 $\text{length } xs \bmod k = 0 \implies (xs \frown f) \div_{il} k = xs \div_{fl} k \frown (f \div_{il} k)$
 ⟨proof⟩

lemma *i-shrink-last-i-append-mult*:
 $\text{length } xs = m * k \implies (xs \frown f) \div_{il} k = xs \div_{fl} k \frown (f \div_{il} k)$
 ⟨proof⟩

lemma *i-shrink-last-Cons*:
 $\llbracket 0 < k; \text{length } xs = k \rrbracket \implies (xs \frown f) \div_{il} k = [\text{last } xs] \frown (f \div_{il} k)$
 ⟨proof⟩

lemma *i-shrink-last-const*: $0 < k \implies (\lambda x. m) \div_{il} k = (\lambda x. m)$
 ⟨proof⟩

lemma *i-shrink-last-i-take-mult*:
 $0 < k \implies f \Downarrow (n * k) \div_{fl} k = f \div_{il} k \Downarrow n$
 ⟨proof⟩

lemma *i-shrink-last-i-take*:
 $f \Downarrow n \div_{fl} k = f \div_{il} k \Downarrow (n \text{ div } k)$
 ⟨proof⟩

lemma *i-shrink-last-i-drop-mult*: $f \Uparrow (n * k) \div_{il} k = f \div_{il} k \Uparrow n$
 ⟨proof⟩

lemma *i-shrink-last-i-drop-mod*:
 $n \bmod k = 0 \implies f \Uparrow n \div_{il} k = f \div_{il} k \Uparrow (n \text{ div } k)$
 ⟨proof⟩

lemma *i-shrink-last-assoc*: $f \div_{il} a \div_{il} b = f \div_{il} (a * b)$
 ⟨proof⟩

lemma *i-shrink-last-commute*: $f \div_{il} a \div_{il} b = f \div_{il} b \div_{il} a$

<proof>

Shrinking a message stream with *last-message* as aggregation function corresponds to shrinking the stream holding last message in each cycle with *last* as aggregation function.

lemma *f-shrink-eq-f-last-message-hold-shrink-last:*

$$xs \div_f k = xs \mapsto_f k \div_{fl} k$$

<proof>

lemma *i-shrink-eq-i-last-message-hold-shrink-last:*

$$0 < k \implies f \div_i k = f \mapsto_i k \div_{il} k$$

<proof>

end

3 Processing of message streams

theory *AF-Stream-Exec*

imports *AF-Stream List-Infinite.ListInf-Prefix List-Infinite.SetIntervalStep*

begin

3.1 Executing components with state transition functions

3.1.1 Basic definitions

Function type for functions converting an input value to an input port message for a component

type-synonym (*'a, 'in*) *Port-Input-Value* = *'a* \Rightarrow *'in message-af*

Function type for functions extracting the output value of a single output port from a component value

type-synonym (*'comp, 'out*) *Port-Output-Value* = *'comp* \Rightarrow *'out message-af*

Function type for functions extracting the local state of a component from a component value

type-synonym (*'comp, 'state*) *Comp-Local-State* = *'comp* \Rightarrow *'state*

Function type for transition functions computing the component's value after processing an input for a single time unit

type-synonym (*'comp, 'input*) *Comp-Trans-Fun* = *'input* \Rightarrow *'comp* \Rightarrow *'comp*

— Execute a component for all inputs in the input stream *'input list*

primrec *f-Exec-Comp* :: (*'comp, 'input*) *Comp-Trans-Fun* \Rightarrow *'input list* \Rightarrow *'comp* \Rightarrow *'comp*

where

f-Exec-Nil: $f\text{-Exec-Comp trans-fun } [] c = c$
| *f-Exec-Cons*: $f\text{-Exec-Comp trans-fun } (x\#xs) c = f\text{-Exec-Comp trans-fun } xs (trans\text{-fun } x c)$

— Execute the component for at most n steps

definition *f-Exec-Comp-N* :: (*'comp*, *'input*) *Comp-Trans-Fun* \Rightarrow *nat* \Rightarrow *'input list* \Rightarrow *'comp* \Rightarrow *'comp*

where *f-Exec-Comp-N trans-fun n xs c* \equiv *f-Exec-Comp trans-fun (xs ↓ n) c*

— Produce the component stream for all inputs in the input stream

primrec *f-Exec-Comp-Stream* :: (*'comp*, *'input*) *Comp-Trans-Fun* \Rightarrow *'input list* \Rightarrow *'comp* \Rightarrow *'comp list*

where

f-Exec-Stream-Nil: $f\text{-Exec-Comp-Stream trans-fun } [] c = []$
| *f-Exec-Stream-Cons*: $f\text{-Exec-Comp-Stream trans-fun } (x \# xs) c =$
 $(trans\text{-fun } x c) \# (f\text{-Exec-Comp-Stream trans-fun } xs (trans\text{-fun } x c))$

primrec *f-Exec-Comp-Stream-Init* ::

(*'comp*, *'input*) *Comp-Trans-Fun* \Rightarrow *'input list* \Rightarrow *'comp* \Rightarrow *'comp list*

where

f-Exec-Stream-Init-Nil: $f\text{-Exec-Comp-Stream-Init trans-fun } [] c = [c]$
| *f-Exec-Stream-Init-Cons*: $f\text{-Exec-Comp-Stream-Init trans-fun } (x \# xs) c =$
 $c \# (f\text{-Exec-Comp-Stream-Init trans-fun } xs (trans\text{-fun } x c))$

definition *i-Exec-Comp-Stream* ::

(*'comp*, *'input*) *Comp-Trans-Fun* \Rightarrow *'input ilist* \Rightarrow *'comp* \Rightarrow *'comp ilist*

where *i-Exec-Comp-Stream* \equiv $\lambda trans\text{-fun input } c n. f\text{-Exec-Comp trans-fun } (input \Downarrow Suc n) c$

definition *i-Exec-Comp-Stream-Init* ::

(*'comp*, *'input*) *Comp-Trans-Fun* \Rightarrow *'input ilist* \Rightarrow *'comp* \Rightarrow *'comp ilist*

where *i-Exec-Comp-Stream-Init* \equiv $\lambda trans\text{-fun input } c n. f\text{-Exec-Comp trans-fun } (input \Downarrow n) c$

3.1.2 Basic results

lemma *f-Exec-one*: $f\text{-Exec-Comp trans-fun } [m] c = trans\text{-fun } m c$
⟨proof⟩

lemma *f-Exec-Stream-length*[*rule-format*, *simp*]:

$\forall c. length (f\text{-Exec-Comp-Stream trans-fun } xs c) = length xs$
⟨proof⟩

lemma *f-Exec-Stream-empty-conv*:

$(f\text{-Exec-Comp-Stream trans-fun } xs c = []) = (xs = [])$
⟨proof⟩

lemma *f-Exec-Stream-not-empty-conv*:

$(f\text{-Exec-Comp-Stream trans-fun } xs c \neq []) = (xs \neq [])$

$\langle \text{proof} \rangle$

lemma *f-Exec-eq-f-Exec-Stream-last*[*rule-format*]:

$\forall c. f\text{-Exec-Comp trans-fun } xs \ c = \text{last } (c \ \# \ (f\text{-Exec-Comp-Stream trans-fun } xs \ c))$

$\langle \text{proof} \rangle$

corollary *f-Exec-eq-f-Exec-Stream-last2*[*rule-format*]:

$xs \neq [] \implies$

$f\text{-Exec-Comp trans-fun } xs \ c = \text{last } (f\text{-Exec-Comp-Stream trans-fun } xs \ c)$

$\langle \text{proof} \rangle$

corollary *f-Exec-eq-f-Exec-Stream-last-if*:

$f\text{-Exec-Comp trans-fun } xs \ c = (\text{if } xs = [] \ \text{then } c \ \text{else } \text{last } (f\text{-Exec-Comp-Stream trans-fun } xs \ c))$

$\langle \text{proof} \rangle$

corollary *f-Exec-take-eq-last-f-Exec-Stream-take*:

$[xs \neq []; 0 < n] \implies$

$f\text{-Exec-Comp trans-fun } (xs \downarrow n) \ c =$

$\text{last } (f\text{-Exec-Comp-Stream trans-fun } (xs \downarrow n) \ c)$

$\langle \text{proof} \rangle$

corollary *f-Exec-N-eq-last-f-Exec-Stream-take*:

$[xs \neq []; 0 < n] \implies$

$f\text{-Exec-Comp-N trans-fun } n \ xs \ c =$

$\text{last } (f\text{-Exec-Comp-Stream trans-fun } (xs \downarrow n) \ c)$

$\langle \text{proof} \rangle$

lemma *f-Exec-Stream-nth*:

$\bigwedge n \ c. \ n < \text{length } xs \implies$

$f\text{-Exec-Comp-Stream trans-fun } xs \ c \ ! \ n = f\text{-Exec-Comp trans-fun } (xs \downarrow \text{Suc } n) \ c$

$\langle \text{proof} \rangle$

lemma *f-Exec-Stream-nth2*:

$n \leq \text{length } xs \implies$

$(c \ \# \ f\text{-Exec-Comp-Stream trans-fun } xs \ c) \ ! \ n = f\text{-Exec-Comp trans-fun } (xs \downarrow n) \ c$

$\langle \text{proof} \rangle$

lemma *f-Exec-N-all*:

$\text{length } xs \leq n \implies$

$f\text{-Exec-Comp-N trans-fun } n \ xs \ c = f\text{-Exec-Comp trans-fun } xs \ c$

$\langle \text{proof} \rangle$

lemma *f-Exec-Stream-append*[*rule-format*]: $\forall c.$

$f\text{-Exec-Comp-Stream trans-fun } (xs \ @ \ ys) \ c =$

$(f\text{-Exec-Comp-Stream trans-fun } xs \ c) \ @$

$(f\text{-Exec-Comp-Stream trans-fun } ys \ (f\text{-Exec-Comp trans-fun } xs \ c))$

$\langle \text{proof} \rangle$

corollary *f-Exec-Stream-append-last-Cons*[rule-format]:

$$\begin{aligned} & f\text{-Exec-Comp-Stream trans-fun } (xs @ ys) c = \\ & \quad (f\text{-Exec-Comp-Stream trans-fun } xs c) @ \\ & \quad (f\text{-Exec-Comp-Stream trans-fun } ys (\text{last } (c \# (f\text{-Exec-Comp-Stream trans-fun } \\ & \quad xs c)))) \end{aligned}$$

$\langle \text{proof} \rangle$

corollary *f-Exec-Stream-append-last*[rule-format]:

$$\begin{aligned} & xs \neq [] \implies \\ & f\text{-Exec-Comp-Stream trans-fun } (xs @ ys) c = \\ & \quad (f\text{-Exec-Comp-Stream trans-fun } xs c) @ \\ & \quad (f\text{-Exec-Comp-Stream trans-fun } ys (\text{last } (f\text{-Exec-Comp-Stream trans-fun } xs c))) \end{aligned}$$

$\langle \text{proof} \rangle$

corollary *f-Exec-Stream-append-if*:

$$\begin{aligned} & f\text{-Exec-Comp-Stream trans-fun } (xs @ ys) c = \\ & \quad (f\text{-Exec-Comp-Stream trans-fun } xs c) @ \\ & \quad (f\text{-Exec-Comp-Stream trans-fun } ys (\\ & \quad \text{if } xs = [] \text{ then } c \text{ else last } (f\text{-Exec-Comp-Stream trans-fun } xs c))) \end{aligned}$$

$\langle \text{proof} \rangle$

corollary *f-Exec-append*:

$$\begin{aligned} & f\text{-Exec-Comp trans-fun } (xs @ ys) c = \\ & \quad f\text{-Exec-Comp trans-fun } ys (f\text{-Exec-Comp trans-fun } xs c) \end{aligned}$$

$\langle \text{proof} \rangle$

corollary *f-Exec-Stream-Cons-rev*:

$$\begin{aligned} & xs \neq [] \implies \\ & \quad (\text{trans-fun } (\text{hd } xs) c) \# \\ & \quad f\text{-Exec-Comp-Stream trans-fun } (\text{tl } xs) (\text{trans-fun } (\text{hd } xs) c) = \\ & \quad f\text{-Exec-Comp-Stream trans-fun } xs c \end{aligned}$$

$\langle \text{proof} \rangle$

lemma *f-Exec-Stream-snoc*:

$$\begin{aligned} & f\text{-Exec-Comp-Stream trans-fun } (xs @ [x]) c = \\ & \quad f\text{-Exec-Comp-Stream trans-fun } xs c @ \\ & \quad [\text{trans-fun } x (f\text{-Exec-Comp trans-fun } xs c)] \end{aligned}$$

$\langle \text{proof} \rangle$

lemma *f-Exec-snoc*:

$$\begin{aligned} & f\text{-Exec-Comp trans-fun } (xs @ [x]) c = \\ & \quad \text{trans-fun } x (f\text{-Exec-Comp trans-fun } xs c) \end{aligned}$$

$\langle \text{proof} \rangle$

lemma *f-Exec-N-append*[rule-format]:

$$\begin{aligned} & f\text{-Exec-Comp-N trans-fun } (a + b) xs c = \\ & \quad f\text{-Exec-Comp-N trans-fun } b (xs \uparrow a) (f\text{-Exec-Comp-N trans-fun } a xs c) \end{aligned}$$

$\langle \text{proof} \rangle$

corollary *f-Exec-N-Suc*[rule-format]:

$f\text{-Exec-Comp-N trans-fun (Suc n) xs c} =$
 $f\text{-Exec-Comp-N trans-fun (Suc 0) (xs \uparrow n) (f\text{-Exec-Comp-N trans-fun n xs c})$
 $\langle \text{proof} \rangle$

corollary *f-Exec-N-Suc2*[rule-format]:

$n < \text{length xs} \implies$
 $f\text{-Exec-Comp-N trans-fun (Suc n) xs c} =$
 $\text{trans-fun (xs ! n) (f\text{-Exec-Comp-N trans-fun n xs c})$
 $\langle \text{proof} \rangle$

theorem *f-Exec-Stream-take*:

$(f\text{-Exec-Comp-Stream trans-fun xs c}) \downarrow n =$
 $f\text{-Exec-Comp-Stream trans-fun (xs \downarrow n) c}$
 $\langle \text{proof} \rangle$

theorem *f-Exec-Stream-drop*:

$(f\text{-Exec-Comp-Stream trans-fun xs c}) \uparrow n =$
 $f\text{-Exec-Comp-Stream trans-fun (xs \uparrow n)$
 $(f\text{-Exec-Comp trans-fun (xs \downarrow n) c})$
 $\langle \text{proof} \rangle$

lemma *i-Exec-Stream-nth*:

$i\text{-Exec-Comp-Stream trans-fun input c n} = f\text{-Exec-Comp trans-fun (input \Downarrow Suc$
 $n) c$
 $\langle \text{proof} \rangle$

lemma *i-Exec-Stream-nth-Suc*:

$i\text{-Exec-Comp-Stream trans-fun input c (Suc n)} =$
 $\text{trans-fun (input (Suc n)) (i\text{-Exec-Comp-Stream trans-fun input c n})$
 $\langle \text{proof} \rangle$

lemma *i-Exec-Stream-nth-Suc-first*:

$i\text{-Exec-Comp-Stream trans-fun input c (Suc n)} =$
 $(i\text{-Exec-Comp-Stream trans-fun (input \uparrow Suc 0) (trans-fun (input 0) c) n)$
 $\langle \text{proof} \rangle$

lemma *f-Exec-Stream-nth-eq-i-Exec-Stream-nth*:

$n < n' \implies$
 $f\text{-Exec-Comp-Stream trans-fun (input \Downarrow n') c ! n} =$
 $i\text{-Exec-Comp-Stream trans-fun input c n}$
 $\langle \text{proof} \rangle$

lemma *i-Exec-Stream-append*:

$i\text{-Exec-Comp-Stream trans-fun (xs \frown \text{input}) c} =$
 $f\text{-Exec-Comp-Stream trans-fun xs c} \frown$

i-Exec-Comp-Stream trans-fun input (f-Exec-Comp trans-fun xs c)
 ⟨proof⟩

lemma *i-Exec-Stream-append-last-Cons*:
i-Exec-Comp-Stream trans-fun (xs ◊ input) c =
f-Exec-Comp-Stream trans-fun xs c ◊
i-Exec-Comp-Stream trans-fun input (
last (c # f-Exec-Comp-Stream trans-fun xs c))
 ⟨proof⟩

lemma *i-Exec-Stream-append-last*:
 $xs \neq [] \implies$
i-Exec-Comp-Stream trans-fun (xs ◊ input) c =
f-Exec-Comp-Stream trans-fun xs c ◊
i-Exec-Comp-Stream trans-fun input (
last (f-Exec-Comp-Stream trans-fun xs c))
 ⟨proof⟩

lemma *i-Exec-Stream-append-if*:
i-Exec-Comp-Stream trans-fun (xs ◊ input) c =
f-Exec-Comp-Stream trans-fun xs c ◊
i-Exec-Comp-Stream trans-fun input (
if xs = [] then c
else last (f-Exec-Comp-Stream trans-fun xs c))
 ⟨proof⟩

corollary *i-Exec-Stream-Cons*:
i-Exec-Comp-Stream trans-fun ([x] ◊ input) c =
[trans-fun x c] ◊ i-Exec-Comp-Stream trans-fun input (trans-fun x c)
 ⟨proof⟩

corollary *i-Exec-Stream-Cons-rev*:
 $[trans-fun (input\ 0)\ c] \frown$
i-Exec-Comp-Stream trans-fun (input ↑ Suc 0) (trans-fun (input 0) c) =
i-Exec-Comp-Stream trans-fun input c
 ⟨proof⟩

theorem *i-Exec-Stream-take*:
 $(i-Exec-Comp-Stream trans-fun input c) \Downarrow n =$
 $f-Exec-Comp-Stream trans-fun (input \Downarrow n) c$
 ⟨proof⟩

theorem *i-Exec-Stream-drop*:
 $(i-Exec-Comp-Stream trans-fun input c) \Uparrow n =$
 $i-Exec-Comp-Stream trans-fun (input \Uparrow n) (f-Exec-Comp trans-fun (input \Downarrow n)$
 $c)$
 ⟨proof⟩

lemma *f-Exec-Stream-expand-aggregate-map-take*:

$f\text{-aggregate } (\text{map } f \text{ (} f\text{-Exec-Comp-Stream trans-fun } (xs \odot_f k) c)) k \text{ ag } \downarrow n =$
 $f\text{-aggregate } (\text{map } f \text{ (} f\text{-Exec-Comp-Stream trans-fun } ((xs \downarrow n) \odot_f k) c)) k \text{ ag}$
 <proof>

corollary *f-Exec-Stream-expand-aggregate-take:*

$f\text{-aggregate } (f\text{-Exec-Comp-Stream trans-fun } (xs \odot_f k) c) k \text{ ag } \downarrow n =$
 $f\text{-aggregate } (f\text{-Exec-Comp-Stream trans-fun } ((xs \downarrow n) \odot_f k) c) k \text{ ag}$
 <proof>

lemma *i-Exec-Stream-expand-aggregate-map-take:*

$0 < k \implies$
 $i\text{-aggregate } (f \circ (i\text{-Exec-Comp-Stream trans-fun } (\text{input} \odot_i k) c)) k \text{ ag } \downarrow n =$
 $f\text{-aggregate } (\text{map } f \text{ (} f\text{-Exec-Comp-Stream trans-fun } ((\text{input} \downarrow n) \odot_f k) c)) k \text{ ag}$
 <proof>

corollary *i-Exec-Stream-expand-aggregate-take:*

$0 < k \implies$
 $i\text{-aggregate } (i\text{-Exec-Comp-Stream trans-fun } (\text{input} \odot_i k) c) k \text{ ag } \downarrow n =$
 $f\text{-aggregate } (f\text{-Exec-Comp-Stream trans-fun } ((\text{input} \downarrow n) \odot_f k) c) k \text{ ag}$
 <proof>

lemma *f-Exec-Stream-expand-aggregate-map-drop:*

$f\text{-aggregate } (\text{map } f \text{ (} f\text{-Exec-Comp-Stream trans-fun } (xs \odot_f k) c)) k \text{ ag } \uparrow n =$
 $f\text{-aggregate } (\text{map } f \text{ (} f\text{-Exec-Comp-Stream trans-fun } ((xs \uparrow n) \odot_f k) ($
 $f\text{-Exec-Comp trans-fun } ((xs \downarrow n) \odot_f k) c)) k \text{ ag}$
 <proof>

corollary *f-Exec-Stream-expand-aggregate-drop:*

$f\text{-aggregate } (f\text{-Exec-Comp-Stream trans-fun } (xs \odot_f k) c) k \text{ ag } \uparrow n =$
 $f\text{-aggregate } (f\text{-Exec-Comp-Stream trans-fun } ((xs \uparrow n) \odot_f k) ($
 $f\text{-Exec-Comp trans-fun } ((xs \downarrow n) \odot_f k) c)) k \text{ ag}$
 <proof>

lemma *i-Exec-Stream-expand-aggregate-map-drop:*

$0 < k \implies$
 $i\text{-aggregate } (f \circ (i\text{-Exec-Comp-Stream trans-fun } (\text{input} \odot_i k) c)) k \text{ ag } \uparrow n =$
 $i\text{-aggregate } (f \circ (i\text{-Exec-Comp-Stream trans-fun } ((\text{input} \uparrow n) \odot_i k) ($
 $f\text{-Exec-Comp trans-fun } ((\text{input} \downarrow n) \odot_f k) c)) k \text{ ag}$
 <proof>

corollary *i-Exec-Stream-expand-aggregate-drop:*

$0 < k \implies$
 $i\text{-aggregate } (i\text{-Exec-Comp-Stream trans-fun } (\text{input} \odot_i k) c) k \text{ ag } \uparrow n =$
 $i\text{-aggregate } (i\text{-Exec-Comp-Stream trans-fun } ((\text{input} \uparrow n) \odot_i k) ($
 $f\text{-Exec-Comp trans-fun } ((\text{input} \downarrow n) \odot_f k) c)) k \text{ ag}$
 <proof>

lemma *f-Exec-Stream-expand-aggregate-map-nth-eq-i-nth:*

$\llbracket 0 < k; n < n' \rrbracket \implies$
 $f\text{-aggregate} (\text{map } f (\text{f-Exec-Comp-Stream trans-fun} (\text{input} \Downarrow n' \odot_f k) c)) k \text{ ag} ! n =$
 $i\text{-aggregate} (f \circ (\text{i-Exec-Comp-Stream trans-fun} (\text{input} \odot_i k) c)) k \text{ ag } n$
 <proof>

corollary *f-Exec-Stream-expand-aggregate-map-nth-eq-i-nth'*:

$0 < k \implies$
 $f\text{-aggregate} (\text{map } f (\text{f-Exec-Comp-Stream trans-fun} (\text{input} \Downarrow \text{Suc } n \odot_f k) c)) k \text{ ag} ! n =$
 $i\text{-aggregate} (f \circ (\text{i-Exec-Comp-Stream trans-fun} (\text{input} \odot_i k) c)) k \text{ ag } n$
 <proof>

corollary *f-Exec-Stream-expand-aggregate-nth-eq-i-nth*:

$\llbracket 0 < k; n < n' \rrbracket \implies$
 $f\text{-aggregate} (\text{f-Exec-Comp-Stream trans-fun} (\text{input} \Downarrow n' \odot_f k) c) k \text{ ag} ! n =$
 $i\text{-aggregate} (\text{i-Exec-Comp-Stream trans-fun} (\text{input} \odot_i k) c) k \text{ ag } n$
 <proof>

corollary *f-Exec-Stream-expand-aggregate-nth-eq-i-nth'*:

$0 < k \implies$
 $f\text{-aggregate} (\text{f-Exec-Comp-Stream trans-fun} (\text{input} \Downarrow \text{Suc } n \odot_f k) c) k \text{ ag} ! n =$
 $i\text{-aggregate} (\text{i-Exec-Comp-Stream trans-fun} (\text{input} \odot_i k) c) k \text{ ag } n$
 <proof>

lemma *f-Exec-Stream-expand-shrink-last-map-nth-eq-f-Exec-Comp*:

$\llbracket 0 < k; n < \text{length } xs \rrbracket \implies$
 $\text{map } f (\text{f-Exec-Comp-Stream trans-fun} (xs \odot_f k) c) \div_{\text{fl}} k ! n =$
 $f (\text{f-Exec-Comp trans-fun} ((xs \downarrow \text{Suc } n) \odot_f k) c)$
 <proof>

corollary *f-Exec-Stream-expand-shrink-last-nth-eq-f-Exec-Comp*:

$\llbracket 0 < k; n < \text{length } xs \rrbracket \implies$
 $\text{f-Exec-Comp-Stream trans-fun} (xs \odot_f k) c \div_{\text{fl}} k ! n =$
 $\text{f-Exec-Comp trans-fun} ((xs \downarrow \text{Suc } n) \odot_f k) c$
 <proof>

lemma *f-Exec-Stream-expand-aggregate-map-nth*:

$\llbracket 0 < k; n < \text{length } xs \rrbracket \implies$
 $f\text{-aggregate} (\text{map } f (\text{f-Exec-Comp-Stream trans-fun} (xs \odot_f k) c)) k \text{ ag} ! n =$
 $\text{ag} (\text{map } f (\text{f-Exec-Comp-Stream trans-fun} (xs ! n \# \varepsilon^k - \text{Suc } 0)$
 $(\text{f-Exec-Comp trans-fun} (xs \downarrow n \odot_f k) c)))$
 <proof>

corollary *f-Exec-Stream-expand-aggregate-nth*:

$\llbracket 0 < k; n < \text{length } xs \rrbracket \implies$
 $f\text{-aggregate} (\text{f-Exec-Comp-Stream trans-fun} (xs \odot_f k) c) k \text{ ag} ! n =$
 $\text{ag} (\text{f-Exec-Comp-Stream trans-fun} (xs ! n \# \varepsilon^k - \text{Suc } 0)$

(*f-Exec-Comp trans-fun* ($xs \downarrow n \odot_f k$) c)
 ⟨proof⟩

corollary *f-Exec-Stream-expand-shrink-map-nth*:

[[$0 < k$; $n < \text{length } xs$]] \implies
 (*map f* (*f-Exec-Comp-Stream trans-fun* ($xs \odot_f k$) c)) $\div_f k ! n =$
last-message (*map f* (*f-Exec-Comp-Stream trans-fun* ($xs ! n \# \varepsilon^k - \text{Suc } 0$)
 (*f-Exec-Comp trans-fun* ($xs \downarrow n \odot_f k$) c)))
 ⟨proof⟩

lemma *i-Exec-Stream-expand-aggregate-map-nth*:

$0 < k \implies$
i-aggregate ($f \circ$ (*i-Exec-Comp-Stream trans-fun* ($\text{input} \odot_i k$) c)) $k \text{ ag } n =$
ag (*map f* (*f-Exec-Comp-Stream trans-fun* ($\text{input } n \# \varepsilon^k - \text{Suc } 0$)
 (*f-Exec-Comp trans-fun* ($(\text{input} \downarrow n) \odot_f k$) c)))
 ⟨proof⟩

corollary *i-Exec-Stream-expand-aggregate-nth*:

$0 < k \implies$
i-aggregate (*i-Exec-Comp-Stream trans-fun* ($\text{input} \odot_i k$) c) $k \text{ ag } n =$
ag (*f-Exec-Comp-Stream trans-fun* ($\text{input } n \# \varepsilon^k - \text{Suc } 0$)
 (*f-Exec-Comp trans-fun* ($(\text{input} \downarrow n) \odot_f k$) c))
 ⟨proof⟩

corollary *i-Exec-Stream-expand-shrink-map-nth*:

$0 < k \implies$
 ($(f \circ$ (*i-Exec-Comp-Stream trans-fun* ($\text{input} \odot_i k$) c)) $\div_i k$) $n =$
last-message (*map f* (*f-Exec-Comp-Stream trans-fun* ($\text{input } n \# \varepsilon^k - \text{Suc } 0$)
 (*f-Exec-Comp trans-fun* ($\text{input} \downarrow n \odot_f k$) c)))
 ⟨proof⟩

lemma *f-Exec-Stream-expand-snoc*:

[[$0 < k$; $n < \text{length } xs$]] \implies
f-Exec-Comp-Stream trans-fun ($xs \odot_f k$) $c \uparrow (n * k) \downarrow k =$
f-Exec-Comp-Stream trans-fun ($xs ! n \# \varepsilon^k - \text{Suc } 0$)
 (*f-Exec-Comp trans-fun* ($xs \downarrow n \odot_f k$) c)
 ⟨proof⟩

lemma *f-Exec-Stream-expand-map-aggregate-append*:

f-aggregate (*map f* (*f-Exec-Comp-Stream trans-fun* ($(xs @ ys) \odot_f k$) c)) $k \text{ ag } =$
f-aggregate (*map f* (*f-Exec-Comp-Stream trans-fun* ($xs \odot_f k$) c)) $k \text{ ag } @$
f-aggregate (*map f* (*f-Exec-Comp-Stream trans-fun* ($ys \odot_f k$) (
f-Exec-Comp trans-fun ($xs \odot_f k$) c))) $k \text{ ag}$
 ⟨proof⟩

lemma *i-Exec-Stream-expand-map-aggregate-append*:

i-aggregate ($f \circ$ (*i-Exec-Comp-Stream trans-fun* ($(xs \frown \text{input}) \odot_i k$) c)) $k \text{ ag } =$
f-aggregate (*map f* (*f-Exec-Comp-Stream trans-fun* ($xs \odot_f k$) c)) $k \text{ ag } \frown$
i-aggregate ($f \circ$ (*i-Exec-Comp-Stream trans-fun* ($\text{input} \odot_i k$) (
f-Exec-Comp trans-fun ($xs \odot_f k$) c)))

$f\text{-Exec-Comp trans-fun } (xs \odot_f k) c))) k ag$
 ⟨proof⟩

lemma $f\text{-Exec-Stream-expand-map-aggregate-Cons}$:

$0 < k \implies$
 $f\text{-aggregate } (map f (f\text{-Exec-Comp-Stream trans-fun } ((x \# xs) \odot_f k) c)) k ag =$
 $ag (map f (f\text{-Exec-Comp-Stream trans-fun } (x \# \varepsilon^k - Suc 0) c)) \#$
 $f\text{-aggregate } (map f (f\text{-Exec-Comp-Stream trans-fun } (xs \odot_f k) ($
 $f\text{-Exec-Comp trans-fun } (x \# \varepsilon^k - Suc 0) c))) k ag$
 ⟨proof⟩

lemma $f\text{-Exec-Stream-expand-map-aggregate-snoc}$:

$0 < k \implies$
 $f\text{-aggregate } (map f (f\text{-Exec-Comp-Stream trans-fun } ((xs @ [x]) \odot_f k) c)) k ag =$
 $f\text{-aggregate } (map f (f\text{-Exec-Comp-Stream trans-fun } (xs \odot_f k) c)) k ag @$
 $[ag (map f (f\text{-Exec-Comp-Stream trans-fun } (x \# \varepsilon^k - Suc 0) ($
 $f\text{-Exec-Comp trans-fun } (xs \odot_f k) c)))]$
 ⟨proof⟩

lemma $i\text{-Exec-Stream-expand-map-aggregate-Cons}$:

$0 < k \implies$
 $i\text{-aggregate } (f \circ (i\text{-Exec-Comp-Stream trans-fun } (([x] \frown input) \odot_i k) c)) k ag =$
 $[ag (map f (f\text{-Exec-Comp-Stream trans-fun } (x \# \varepsilon^k - Suc 0) c))] \frown$
 $i\text{-aggregate } (f \circ (i\text{-Exec-Comp-Stream trans-fun } (input \odot_i k) ($
 $f\text{-Exec-Comp trans-fun } (x \# \varepsilon^k - Suc 0) c))) k ag$
 ⟨proof⟩

lemma $f\text{-Exec-N-eq-f-Exec-Stream-nth}$:

$n \leq \text{length } xs \implies$
 $f\text{-Exec-Comp-N trans-fun } n xs c = (c \# f\text{-Exec-Comp-Stream trans-fun } xs c) ! n$
 ⟨proof⟩

theorem $f\text{-Exec-Stream-causal}$:

$xs \downarrow n = ys \downarrow n \implies$
 $(f\text{-Exec-Comp-Stream trans-fun } xs c) \downarrow n = (f\text{-Exec-Comp-Stream trans-fun } ys c)$
 $\downarrow n$
 ⟨proof⟩

theorem $i\text{-Exec-Stream-causal}$:

$input1 \Downarrow n = input2 \Downarrow n \implies$
 $(i\text{-Exec-Comp-Stream trans-fun } input1 c) \Downarrow n = (i\text{-Exec-Comp-Stream trans-fun } input2 c) \Downarrow n$
 ⟨proof⟩

Results for $f\text{-Exec-Comp-Stream-Init}$

$f\text{-Exec-Comp-Stream-Init}$ computes the execution stream of a component with the initial value of the component at the beginning of the result stream.

lemma $f\text{-Exec-Stream-Init-length}[rule-format, simp]$:

$\forall c. \text{length } (f\text{-Exec-Comp-Stream-Init trans-fun } xs c) = Suc (\text{length } xs)$

$\langle \text{proof} \rangle$

lemma *f-Exec-Stream-Init-not-empty*:

$(f\text{-Exec-Comp-Stream-Init trans-fun } xs \ c \neq [])$

$\langle \text{proof} \rangle$

lemma *f-Exec-eq-f-Exec-Stream-Init-last*[rule-format]:

$\forall c. f\text{-Exec-Comp trans-fun } xs \ c = \text{last } (f\text{-Exec-Comp-Stream-Init trans-fun } xs \ c)$

$\langle \text{proof} \rangle$

lemma *f-Exec-Stream-Init-eq-f-Exec-Stream-Cons*[rule-format]:

$\forall c. f\text{-Exec-Comp-Stream-Init trans-fun } xs \ c = c \# f\text{-Exec-Comp-Stream trans-fun } xs \ c$

$\langle \text{proof} \rangle$

corollary *f-Exec-Stream-Init-eq-f-Exec-Stream-Cons-output*:

$\text{output-fun } c = \varepsilon \implies$

$\text{map output-fun } (f\text{-Exec-Comp-Stream-Init trans-fun } xs \ c) =$

$\varepsilon \# \text{map output-fun } (f\text{-Exec-Comp-Stream trans-fun } xs \ c)$

$\langle \text{proof} \rangle$

corollary *f-Exec-Stream-Init-tl-eq-f-Exec-Stream*:

$\text{tl } (f\text{-Exec-Comp-Stream-Init trans-fun } xs \ c) = f\text{-Exec-Comp-Stream trans-fun } xs \ c$

$\langle \text{proof} \rangle$

lemma *f-Exec-N-eq-last-f-Exec-Stream-Init-take*:

$f\text{-Exec-Comp-N trans-fun } n \ xs \ c =$

$\text{last } (f\text{-Exec-Comp-Stream-Init trans-fun } (xs \downarrow n) \ c)$

$\langle \text{proof} \rangle$

lemma *f-Exec-Stream-Init-nth*:

$n \leq \text{length } xs \implies$

$f\text{-Exec-Comp-Stream-Init trans-fun } xs \ c \ ! \ n = f\text{-Exec-Comp trans-fun } (xs \downarrow n) \ c$

$\langle \text{proof} \rangle$

lemma *f-Exec-Stream-Init-nth-0*: $f\text{-Exec-Comp-Stream-Init trans-fun } xs \ c \ ! \ 0 = c$

$\langle \text{proof} \rangle$

lemma *f-Exec-Stream-Init-hd*: $\text{hd } (f\text{-Exec-Comp-Stream-Init trans-fun } xs \ c) = c$

$\langle \text{proof} \rangle$

lemma *f-Exec-Stream-Init-nth-Suc-eq-f-Exec-Stream-nth*:

$f\text{-Exec-Comp-Stream-Init trans-fun } xs \ c \ ! \ (\text{Suc } n) = f\text{-Exec-Comp-Stream trans-fun } xs \ c \ ! \ n$

$\langle \text{proof} \rangle$

lemma *f-Exec-Stream-Init-append*:

$f\text{-Exec-Comp-Stream-Init trans-fun } (xs \ @ \ ys) \ c =$

$(f\text{-Exec-Comp-Stream-Init trans-fun } xs \ c) \ @$
 $tl (f\text{-Exec-Comp-Stream-Init trans-fun } ys \ (f\text{-Exec-Comp trans-fun } xs \ c))$
 ⟨proof⟩

corollary *f-Exec-Stream-Init-append-last:*

$f\text{-Exec-Comp-Stream-Init trans-fun } (xs \ @ \ ys) \ c =$
 $(f\text{-Exec-Comp-Stream-Init trans-fun } xs \ c) \ @$
 $tl (f\text{-Exec-Comp-Stream-Init trans-fun } ys \ (last (f\text{-Exec-Comp-Stream-Init trans-fun } xs \ c)))$
 ⟨proof⟩

lemma *f-Exec-Stream-Init-f-Exec-Stream-append:*

$f\text{-Exec-Comp-Stream-Init trans-fun } (xs \ @ \ ys) \ c =$
 $(f\text{-Exec-Comp-Stream-Init trans-fun } xs \ c) \ @$
 $(f\text{-Exec-Comp-Stream trans-fun } ys \ (f\text{-Exec-Comp trans-fun } xs \ c))$
 ⟨proof⟩

lemma *f-Exec-Stream-Init-take:*

$(f\text{-Exec-Comp-Stream-Init trans-fun } xs \ c) \ \downarrow \ Suc \ n =$
 $f\text{-Exec-Comp-Stream-Init trans-fun } (xs \ \downarrow \ n) \ c$
 ⟨proof⟩

lemma *f-Exec-Stream-Init-drop:*

$n \leq length \ xs \ \Longrightarrow$
 $(f\text{-Exec-Comp-Stream-Init trans-fun } xs \ c) \ \uparrow \ n =$
 $f\text{-Exec-Comp-Stream-Init trans-fun } (xs \ \uparrow \ n)$
 $(f\text{-Exec-Comp trans-fun } (xs \ \downarrow \ n) \ c)$
 ⟨proof⟩

lemma *f-Exec-Stream-Init-drop-geq-not-valid:*

$length \ xs \ \leq \ n \ \Longrightarrow$
 $(f\text{-Exec-Comp-Stream-Init trans-fun } xs \ c) \ \uparrow \ Suc \ n \neq$
 $f\text{-Exec-Comp-Stream-Init trans-fun arbitrary-input arbitrary-comp}$
 ⟨proof⟩

lemma *i-Exec-Stream-Init-nth:*

$i\text{-Exec-Comp-Stream-Init trans-fun input } c \ n = f\text{-Exec-Comp trans-fun } (input \ \Downarrow \ n) \ c$
 ⟨proof⟩

lemma *i-Exec-Stream-Init-nth-0:*

$i\text{-Exec-Comp-Stream-Init trans-fun input } c \ 0 = c$
 ⟨proof⟩

lemma *i-Exec-Stream-Init-nth-Suc-eq-i-Exec-Stream-nth:*

$i\text{-Exec-Comp-Stream-Init trans-fun input } c \ (Suc \ n) = i\text{-Exec-Comp-Stream trans-fun input } c \ n$
 ⟨proof⟩

lemma *i-Exec-Stream-Init-eq-i-Exec-Stream-Cons*:

i-Exec-Comp-Stream-Init trans-fun input c = [c] \frown i-Exec-Comp-Stream trans-fun input c
 $\langle \text{proof} \rangle$

corollary *i-Exec-Stream-Init-eq-i-Exec-Stream-Cons-output*:

output-fun c = $\varepsilon \implies$
output-fun \circ i-Exec-Comp-Stream-Init trans-fun input c =
[ε] \frown (output-fun \circ i-Exec-Comp-Stream trans-fun input c)
 $\langle \text{proof} \rangle$

lemma *i-Exec-Stream-Init-append*:

i-Exec-Comp-Stream-Init trans-fun (input1 \frown input2) c =
(f-Exec-Comp-Stream-Init trans-fun input1 c) \frown
((i-Exec-Comp-Stream-Init trans-fun input2 (f-Exec-Comp trans-fun input1 c))
 $\uparrow \text{Suc } 0$)
 $\langle \text{proof} \rangle$

corollary *i-Exec-Stream-Init-append-last*:

i-Exec-Comp-Stream-Init trans-fun (input1 \frown input2) c =
(f-Exec-Comp-Stream-Init trans-fun input1 c) \frown
((i-Exec-Comp-Stream-Init trans-fun input2 (last (f-Exec-Comp-Stream-Init
trans-fun input1 c))) $\uparrow \text{Suc } 0$)
 $\langle \text{proof} \rangle$

lemma *i-Exec-Stream-Init-i-Exec-Stream-append*:

i-Exec-Comp-Stream-Init trans-fun (input1 \frown input2) c =
(f-Exec-Comp-Stream-Init trans-fun input1 c) \frown
(i-Exec-Comp-Stream trans-fun input2 (f-Exec-Comp trans-fun input1 c))
 $\langle \text{proof} \rangle$

lemma *i-Exec-Stream-Init-take*:

(i-Exec-Comp-Stream-Init trans-fun input c) $\downarrow \text{Suc } n =$
f-Exec-Comp-Stream-Init trans-fun (input $\downarrow n$) c
 $\langle \text{proof} \rangle$

lemma *i-Exec-Stream-Init-drop*:

(i-Exec-Comp-Stream-Init trans-fun input c) $\uparrow n =$
i-Exec-Comp-Stream-Init trans-fun (input $\uparrow n$)
(f-Exec-Comp trans-fun (input $\downarrow n$) c)
 $\langle \text{proof} \rangle$

theorem *f-Exec-Stream-Init-strictly-causal*:

xs $\downarrow n =$ ys $\downarrow n \implies$
(f-Exec-Comp-Stream-Init trans-fun xs c) $\downarrow \text{Suc } n =$ (f-Exec-Comp-Stream-Init
trans-fun ys c) $\downarrow \text{Suc } n$
 $\langle \text{proof} \rangle$

theorem *i-Exec-Stream-Init-strictly-causal*:

input1 $\downarrow n =$ input2 $\downarrow n \implies$

(*i-Exec-Comp-Stream-Init trans-fun input1 c*) \Downarrow *Suc n* = (*i-Exec-Comp-Stream-Init trans-fun input2 c*) \Downarrow *Suc n*
 ⟨proof⟩

theorem *f-Exec-N-eq-f-Exec-Stream-Init-nth*:

$n \leq \text{length } xs \implies$

f-Exec-Comp-N trans-fun n xs c = *f-Exec-Comp-Stream-Init trans-fun xs c ! n*

⟨proof⟩

Basic results for previous element functions

The functions *list-Previous* and *ilist-Previous* return the previous element of the list relatively to the specified position *n* or the initial element if *n* is 0,

definition *list-Previous* :: 'value list \Rightarrow 'value \Rightarrow nat \Rightarrow 'value

where *list-Previous xs init n* \equiv

case *n* of

0 \Rightarrow *init*

| *Suc n'* \Rightarrow *xs ! n'*

definition *ilist-Previous* :: 'value ilist \Rightarrow 'value \Rightarrow nat \Rightarrow 'value

where *ilist-Previous f init n* \equiv

case *n* of

0 \Rightarrow *init*

| *Suc n'* \Rightarrow *f n'*

abbreviation *list-Previous'* :: 'value list \Rightarrow 'value \Rightarrow nat \Rightarrow 'value

($-\overset{\leftarrow}{\prime}$ - [1000, 10, 100] 100)

where *xs $^{\leftarrow \prime}$ init n* \equiv *list-Previous xs init n*

abbreviation *ilist-Previous'* :: 'value ilist \Rightarrow 'value \Rightarrow nat \Rightarrow 'value

($-\overset{\leftarrow}{\prime}$ - [1000, 10, 100] 100)

where *f $^{\leftarrow \prime}$ init n* \equiv *ilist-Previous f init n*

lemma *list-Previous-nth*: *xs $^{\leftarrow \prime}$ init n* = (case *n* of 0 \Rightarrow *init* | *Suc n'* \Rightarrow *xs ! n'*)
 ⟨proof⟩

lemma *ilist-Previous-nth*: *f $^{\leftarrow \prime}$ init n* = (case *n* of 0 \Rightarrow *init* | *Suc n'* \Rightarrow *f n'*)
 ⟨proof⟩

lemma *list-Previous-nth-if*: *xs $^{\leftarrow \prime}$ init n* = (if *n* = 0 then *init* else *xs ! (n - Suc 0)*)
 ⟨proof⟩

lemma *ilist-Previous-nth-if*: *f $^{\leftarrow \prime}$ init n* = (if *n* = 0 then *init* else *f (n - Suc 0)*)
 ⟨proof⟩

lemma *list-Previous-Cons*: *xs $^{\leftarrow \prime}$ init n* = (*init # xs*) ! *n*
 ⟨proof⟩

lemma *ilist-Previous-Cons*: $f^{\leftarrow} \text{init } n = ([\text{init}] \frown f) n$
 ⟨proof⟩

lemma *list-Previous-0*: $xs^{\leftarrow'} \text{init } 0 = \text{init}$
 ⟨proof⟩

lemma *ilist-Previous-0*: $f^{\leftarrow} \text{init } 0 = \text{init}$
 ⟨proof⟩

lemma *list-Previous-gr0*: $0 < n \implies xs^{\leftarrow'} \text{init } n = xs ! (n - \text{Suc } 0)$
 ⟨proof⟩

lemma *ilist-Previous-gr0*: $0 < n \implies f^{\leftarrow} \text{init } n = f (n - \text{Suc } 0)$
 ⟨proof⟩

lemma *list-Previous-Suc*: $xs^{\leftarrow'} \text{init } (\text{Suc } n) = xs ! n$
 ⟨proof⟩

lemma *ilist-Previous-Suc*: $f^{\leftarrow} \text{init } (\text{Suc } n) = f n$
 ⟨proof⟩

lemma *f-Exec-Stream-Previous-f-Exec-Stream-Init*:
 $f\text{-Exec-Comp-Stream-Init trans-fun } xs \ c ! n =$
 $(f\text{-Exec-Comp-Stream trans-fun } xs \ c)^{\leftarrow'} c \ n$
 ⟨proof⟩

lemma *i-Exec-Stream-Previous-i-Exec-Stream-Init*:
 $i\text{-Exec-Comp-Stream-Init trans-fun input } c \ n =$
 $(i\text{-Exec-Comp-Stream trans-fun input } c)^{\leftarrow} c \ n$
 ⟨proof⟩

lemma *f-Exec-Stream-hd*:
 $0 < \text{length } xs \implies \text{hd } (f\text{-Exec-Comp-Stream trans-fun } xs \ c) = \text{trans-fun } (\text{hd } xs) \ c$
 ⟨proof⟩

lemma *f-Exec-Stream-nth-0*:
 $0 < \text{length } xs \implies (f\text{-Exec-Comp-Stream trans-fun } xs \ c) ! 0 = \text{trans-fun } (xs ! 0) \ c$
 ⟨proof⟩

The calculation of the n-th result stream element from the previous result stream element and the current input stream element.

lemma *f-Exec-Stream-nth-gr0-calc*:
 $\llbracket n < \text{length } xs; 0 < n \rrbracket \implies$
 $f\text{-Exec-Comp-Stream trans-fun } xs \ c ! n =$
 $\text{trans-fun } (xs ! n) (f\text{-Exec-Comp-Stream trans-fun } xs \ c ! (n - 1))$
 ⟨proof⟩

lemma *f-Exec-Stream-nth-calc-Previous:*

$$\begin{aligned} n < \text{length } xs &\implies \\ f\text{-Exec-Comp-Stream } \text{trans-fun } xs \ c \ ! \ n &= \\ \text{trans-fun } (xs \ ! \ n) \ ((f\text{-Exec-Comp-Stream } \text{trans-fun } xs \ c)^{\leftarrow' \ c} \ n) & \\ \langle \text{proof} \rangle & \end{aligned}$$

lemma *i-Exec-Stream-nth-0:*

$$\begin{aligned} (i\text{-Exec-Comp-Stream } \text{trans-fun } \text{input } c) \ 0 &= \text{trans-fun } (\text{input } 0) \ c \\ \langle \text{proof} \rangle & \end{aligned}$$

lemma *i-Exec-Stream-nth-gr0-calc:*

$$\begin{aligned} 0 < n &\implies \\ (i\text{-Exec-Comp-Stream } \text{trans-fun } \text{input } c) \ n &= \\ \text{trans-fun } (\text{input } n) \ ((i\text{-Exec-Comp-Stream } \text{trans-fun } \text{input } c) \ (n - 1)) & \\ \langle \text{proof} \rangle & \end{aligned}$$

The component state (and thus its output) at time point n is computed from the previous state (the state at time $n - (1::'a)$ for $(0::'a) < n$ or the initial state for $n = (0::'a)$) and the input at time n .

lemma *i-Exec-Stream-nth-calc-Previous:*

$$\begin{aligned} i\text{-Exec-Comp-Stream } \text{trans-fun } \text{input } c \ n &= \\ \text{trans-fun } (\text{input } n) \ ((i\text{-Exec-Comp-Stream } \text{trans-fun } \text{input } c)^{\leftarrow \ c} \ n) & \\ \langle \text{proof} \rangle & \end{aligned}$$

lemma *f-Exec-Stream-Init-nth-Suc-calc:*

$$\begin{aligned} n < \text{length } xs &\implies \\ f\text{-Exec-Comp-Stream-Init } \text{trans-fun } xs \ c \ ! \ \text{Suc } n &= \\ \text{trans-fun } (xs \ ! \ n) \ (f\text{-Exec-Comp-Stream-Init } \text{trans-fun } xs \ c \ ! \ n) & \\ \langle \text{proof} \rangle & \end{aligned}$$

lemma *f-Exec-Stream-Init-nth-Plus1-calc:*

$$\begin{aligned} n < \text{length } xs &\implies \\ f\text{-Exec-Comp-Stream-Init } \text{trans-fun } xs \ c \ ! \ (n + 1) &= \\ \text{trans-fun } (xs \ ! \ n) \ (f\text{-Exec-Comp-Stream-Init } \text{trans-fun } xs \ c \ ! \ n) & \\ \langle \text{proof} \rangle & \end{aligned}$$

lemma *f-Exec-Stream-Init-nth-gr0-calc:*

$$\begin{aligned} \llbracket n \leq \text{length } xs; \ 0 < n \rrbracket &\implies \\ f\text{-Exec-Comp-Stream-Init } \text{trans-fun } xs \ c \ ! \ n &= \\ \text{trans-fun } (xs \ ! \ (n - 1)) \ (f\text{-Exec-Comp-Stream-Init } \text{trans-fun } xs \ c \ ! \ (n - 1)) & \\ \langle \text{proof} \rangle & \end{aligned}$$

At the beginning, the component state (and thus its output) for the execution stream with initial state is represented by the initial state, contrary to the *i-Exec-Comp-Stream* that does not contain the initial state.

The component state (and thus its output) at time point $n + (1::'a)$ for

the execution stream with initial state is computed from the previous state (the state at time n) and the previous input (input at time n), contrary to the *i-Exec-Comp-Stream*, where each state at time n represents the resulting state after processing the input at time n .

lemma *i-Exec-Stream-Init-nth-Suc-calc*:

$$\begin{aligned} & i\text{-Exec-Comp-Stream-Init trans-fun input } c \text{ (Suc } n) = \\ & \text{trans-fun (input } n) (i\text{-Exec-Comp-Stream-Init trans-fun input } c \text{ } n) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *i-Exec-Stream-Init-nth-Plus1-calc*:

$$\begin{aligned} & i\text{-Exec-Comp-Stream-Init trans-fun input } c \text{ (} n + 1) = \\ & \text{trans-fun (input } n) (i\text{-Exec-Comp-Stream-Init trans-fun input } c \text{ } n) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *i-Exec-Stream-Init-nth-gr0-calc*:

$$\begin{aligned} & 0 < n \implies \\ & i\text{-Exec-Comp-Stream-Init trans-fun input } c \text{ } n = \\ & \text{trans-fun (input (} n - 1)) (i\text{-Exec-Comp-Stream-Init trans-fun input } c \text{ (} n - 1)) \\ & \langle \text{proof} \rangle \end{aligned}$$

Correlation between Pre/Post-Conditions for *f-Exec-Comp-Stream* and *f-Exec-Comp-Stream-Init*

lemma *f-Exec-Stream-Pre-Post1*:

$$\begin{aligned} & \llbracket n < \text{length } xs; \\ & \quad c\text{-}n = (f\text{-Exec-Comp-Stream trans-fun } xs \text{ } c)^{\leftarrow' c} n; x\text{-}n = xs ! n \rrbracket \implies \\ & (P1 \ x\text{-}n \wedge P2 \ c\text{-}n \longrightarrow Q (f\text{-Exec-Comp-Stream trans-fun } xs \text{ } c ! n)) = \\ & (P1 \ x\text{-}n \wedge P2 \ c\text{-}n \longrightarrow Q (\text{trans-fun } x\text{-}n \ c\text{-}n)) \\ & \langle \text{proof} \rangle \end{aligned}$$

Direct relation between input and result after transition

lemma *f-Exec-Stream-Pre-Post2*:

$$\begin{aligned} & \llbracket n < \text{length } xs; \\ & \quad c\text{-}n = (f\text{-Exec-Comp-Stream trans-fun } xs \text{ } c)^{\leftarrow' c} n; x\text{-}n = xs ! n \rrbracket \implies \\ & (P \ c\text{-}n \longrightarrow Q (xs ! n) (f\text{-Exec-Comp-Stream trans-fun } xs \text{ } c ! n)) = \\ & (P \ c\text{-}n \longrightarrow Q \ x\text{-}n (\text{trans-fun } x\text{-}n \ c\text{-}n)) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *f-Exec-Stream-Pre-Post2-Suc*:

$$\begin{aligned} & \llbracket \text{Suc } n < \text{length } xs; \\ & \quad c\text{-}n = f\text{-Exec-Comp-Stream trans-fun } xs \text{ } c ! n; x\text{-}n1 = xs ! \text{Suc } n \rrbracket \implies \\ & (P \ c\text{-}n \longrightarrow Q (xs ! \text{Suc } n) (f\text{-Exec-Comp-Stream trans-fun } xs \text{ } c ! \text{Suc } n)) = \\ & (P \ c\text{-}n \longrightarrow Q \ x\text{-}n1 (\text{trans-fun } x\text{-}n1 \ c\text{-}n)) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *f-Exec-Stream-Init-Pre-Post1*:

$$\begin{aligned} & \llbracket n < \text{length } xs; \\ & \quad c\text{-}n = f\text{-Exec-Comp-Stream-Init trans-fun } xs \text{ } c ! n; x\text{-}n = xs ! n \rrbracket \implies \\ & (P1 \ x\text{-}n \wedge P2 \ c\text{-}n \longrightarrow Q (f\text{-Exec-Comp-Stream-Init trans-fun } xs \text{ } c ! \text{Suc } n)) = \\ & (P1 \ x\text{-}n \wedge P2 \ c\text{-}n \longrightarrow Q (\text{trans-fun } x\text{-}n \ c\text{-}n)) \end{aligned}$$

$\langle proof \rangle$

Direct relation between input and state before transition

lemma *f-Exec-Stream-Init-Pre-Post2*:

$$\begin{aligned} & \llbracket n < \text{length } xs; \\ & \quad c-n = f\text{-Exec-Comp-Stream-Init } \text{trans-fun } xs \ c \ ! \ n; \ x-n = xs \ ! \ n \rrbracket \implies \\ & (P \ (xs \ ! \ n) \ (f\text{-Exec-Comp-Stream-Init } \text{trans-fun } xs \ c \ ! \ n) \longrightarrow \\ & \quad Q \ (f\text{-Exec-Comp-Stream-Init } \text{trans-fun } xs \ c \ ! \ \text{Suc } n)) = \\ & (P \ x-n \ c-n \longrightarrow Q \ (\text{trans-fun } x-n \ c-n)) \end{aligned}$$

$\langle proof \rangle$

lemma *i-Exec-Stream-Pre-Post1*:

$$\begin{aligned} & \llbracket c-n = (i\text{-Exec-Comp-Stream } \text{trans-fun } \text{input } c)^{\leftarrow c} \ n; \ x-n = \text{input } n \rrbracket \implies \\ & (P1 \ x-n \wedge P2 \ c-n \longrightarrow Q \ (i\text{-Exec-Comp-Stream } \text{trans-fun } \text{input } c \ n)) = \\ & (P1 \ x-n \wedge P2 \ c-n \longrightarrow Q \ (\text{trans-fun } x-n \ c-n)) \end{aligned}$$

$\langle proof \rangle$

Direct relation between input and result after transition

lemma *i-Exec-Stream-Pre-Post2*:

$$\begin{aligned} & \llbracket c-n = (i\text{-Exec-Comp-Stream } \text{trans-fun } \text{input } c)^{\leftarrow c} \ n; \ x-n = \text{input } n \rrbracket \implies \\ & (P \ c-n \longrightarrow Q \ (\text{input } n) \ (i\text{-Exec-Comp-Stream } \text{trans-fun } \text{input } c \ n)) = \\ & (P \ c-n \longrightarrow Q \ x-n \ (\text{trans-fun } x-n \ c-n)) \end{aligned}$$

$\langle proof \rangle$

lemma *i-Exec-Stream-Pre-Post2-Suc*:

$$\begin{aligned} & \llbracket c-n = i\text{-Exec-Comp-Stream } \text{trans-fun } \text{input } c \ n; \ x-n1 = \text{input } (\text{Suc } n) \rrbracket \implies \\ & (P \ c-n \longrightarrow Q \ (\text{input } (\text{Suc } n)) \ (i\text{-Exec-Comp-Stream } \text{trans-fun } \text{input } c \ (\text{Suc } n))) \\ & = \\ & (P \ c-n \longrightarrow Q \ x-n1 \ (\text{trans-fun } x-n1 \ c-n)) \end{aligned}$$

$\langle proof \rangle$

lemma *i-Exec-Stream-Init-Pre-Post1*:

$$\begin{aligned} & \llbracket c-n = i\text{-Exec-Comp-Stream-Init } \text{trans-fun } \text{input } c \ n; \ x-n = \text{input } n \rrbracket \implies \\ & (P1 \ x-n \wedge P2 \ c-n \longrightarrow Q \ (i\text{-Exec-Comp-Stream-Init } \text{trans-fun } \text{input } c \ (\text{Suc } n))) = \\ & (P1 \ x-n \wedge P2 \ c-n \longrightarrow Q \ (\text{trans-fun } x-n \ c-n)) \end{aligned}$$

$\langle proof \rangle$

Direct relation between input and state before transition

lemma *i-Exec-Stream-Init-Pre-Post2*:

$$\begin{aligned} & \llbracket c-n = i\text{-Exec-Comp-Stream-Init } \text{trans-fun } \text{input } c \ n; \ x-n = \text{input } n \rrbracket \implies \\ & (P \ (\text{input } n) \ (i\text{-Exec-Comp-Stream-Init } \text{trans-fun } \text{input } c \ n) \longrightarrow \\ & \quad Q \ (i\text{-Exec-Comp-Stream-Init } \text{trans-fun } \text{input } c \ (\text{Suc } n))) = \\ & (P \ x-n \ c-n \longrightarrow Q \ (\text{trans-fun } x-n \ c-n)) \end{aligned}$$

$\langle proof \rangle$

Basic results for stream prefixes

lemma *f-Exec-Stream-prefix*:

$prefix\ xs\ ys \implies$
 $prefix\ (f-Exec-Comp-Stream\ trans-fun\ xs\ c)$
 $(f-Exec-Comp-Stream\ trans-fun\ ys\ c)$
 ⟨proof⟩

lemma *i-Exec-Stream-prefix*:

$xs \sqsubseteq input \implies$
 $f-Exec-Comp-Stream\ trans-fun\ xs\ c \sqsubseteq$
 $i-Exec-Comp-Stream\ trans-fun\ input\ c$
 ⟨proof⟩

lemma *f-Exec-N-prefix*:

$\llbracket n \leq length\ xs; prefix\ xs\ ys \rrbracket \implies$
 $f-Exec-Comp-N\ trans-fun\ n\ xs\ c =$
 $f-Exec-Comp-N\ trans-fun\ n\ ys\ c$
 ⟨proof⟩

theorem *f-Exec-Stream-prefix-causal*[rule-format]:

$n \leq length\ (xs \sqcap ys) \implies$
 $f-Exec-Comp-Stream\ trans-fun\ xs\ c \downarrow n =$
 $f-Exec-Comp-Stream\ trans-fun\ ys\ c \downarrow n$
 ⟨proof⟩

lemma *f-Exec-Stream-Init-prefix*:

$prefix\ xs\ ys \implies$
 $prefix\ (f-Exec-Comp-Stream-Init\ trans-fun\ xs\ c)$
 $(f-Exec-Comp-Stream-Init\ trans-fun\ ys\ c)$
 ⟨proof⟩

lemma *i-Exec-Stream-Init-prefix*:

$xs \sqsubseteq input \implies$
 $f-Exec-Comp-Stream-Init\ trans-fun\ xs\ c \sqsubseteq$
 $i-Exec-Comp-Stream-Init\ trans-fun\ input\ c$
 ⟨proof⟩

theorem *f-Exec-Stream-Init-prefix-strictly-causal*[rule-format]:

$n \leq length\ (xs \sqcap ys) \implies$
 $f-Exec-Comp-Stream-Init\ trans-fun\ xs\ c \downarrow Suc\ n =$
 $f-Exec-Comp-Stream-Init\ trans-fun\ ys\ c \downarrow Suc\ n$
 ⟨proof⟩

A predicate indicating whether a component is deterministically dependent on the local state extracted by the the given local state function.

definition *Deterministic-Trans-Fun* ::

$(\prime comp, \prime input)\ Comp-Trans-Fun \Rightarrow (\prime comp, \prime state)\ Comp-Local-State \Rightarrow bool$

where *Deterministic-Trans-Fun* $trans-fun\ localState \equiv$

$\forall c1\ c2\ x.\ localState\ c1 = localState\ c2 \longrightarrow trans-fun\ x\ c1 = trans-fun\ x\ c2$

lemma *Deterministic-f-Exec*:

$\llbracket \text{Deterministic-Trans-Fun trans-fun localState; localState } c1 = \text{localState } c2; xs \neq [] \rrbracket \implies$
 $f\text{-Exec-Comp trans-fun } xs \ c1 = f\text{-Exec-Comp trans-fun } xs \ c2$
 <proof>

lemma *Deterministic-f-Exec-Stream:*

$\llbracket \text{Deterministic-Trans-Fun trans-fun localState; localState } c1 = \text{localState } c2 \rrbracket \implies$
 $f\text{-Exec-Comp-Stream trans-fun } xs \ c1 = f\text{-Exec-Comp-Stream trans-fun } xs \ c2$
 <proof>

lemma *Deterministic-i-Exec-Stream:*

$\llbracket \text{Deterministic-Trans-Fun trans-fun localState; localState } c1 = \text{localState } c2 \rrbracket \implies$
 $i\text{-Exec-Comp-Stream trans-fun input } c1 = i\text{-Exec-Comp-Stream trans-fun input } c2$
 <proof>

3.1.3 Connected streams

A predicate indicating for two message streams, that the ports, they correspond to, are connected. The predicate implies strict causality.

definition *f-Streams-Connected* :: 'a fstream-af \Rightarrow 'a fstream-af \Rightarrow bool
where *f-Streams-Connected* outS inS \equiv inS = ε # outS

definition *i-Streams-Connected* :: 'a istream-af \Rightarrow 'a istream-af \Rightarrow bool
where *i-Streams-Connected* outS inS \equiv inS = [ε] \frown outS

lemmas *Streams-Connected-defs* =

f-Streams-Connected-def
i-Streams-Connected-def

lemma *f-Streams-Connected-imp-not-empty*: *f-Streams-Connected* outS inS \implies inS $\neq []$
 <proof>

lemma *f-Streams-Connected-nth-conv*:

f-Streams-Connected outS inS =
 (length inS = Suc (length outS) \wedge
 ($\forall i < \text{length inS}. \text{inS} ! i = (\text{case } i \text{ of } 0 \Rightarrow \varepsilon \mid \text{Suc } k \Rightarrow \text{outS} ! k))$)
 <proof>

lemma *f-Streams-Connected-nth-conv-if*:

f-Streams-Connected outS inS =
 (length inS = Suc (length outS) \wedge
 ($\forall i < \text{length inS}. \text{inS} ! i = (\text{if } i = 0 \text{ then } \varepsilon \text{ else } \text{outS} ! (i - \text{Suc } 0))$))
 <proof>

lemma *i-Streams-Connected-nth-conv*:

i-Streams-Connected outS inS =
 $(\forall i. \text{inS } i = (\text{case } i \text{ of } 0 \Rightarrow \varepsilon \mid \text{Suc } k \Rightarrow \text{outS } k))$
 ⟨proof⟩

lemma *i-Streams-Connected-nth-conv-if*:
i-Streams-Connected outS inS =
 $(\forall i. \text{inS } i = (\text{if } i = 0 \text{ then } \varepsilon \text{ else } \text{outS } (i - \text{Suc } 0)))$
 ⟨proof⟩

lemma *f-Exec-Stream-Init-eq-output-channel*:
 $\llbracket \text{output-fun } c = \varepsilon;$
f-Streams-Connected
 $(\text{map output-fun } (f\text{-Exec-Comp-Stream trans-fun } xs \ c))$
 $\text{channel } \rrbracket \Longrightarrow$
 $\text{map output-fun } (f\text{-Exec-Comp-Stream-Init trans-fun } xs \ c) = \text{channel}$
 ⟨proof⟩

lemma *i-Exec-Stream-Init-eq-output-channel*:
 $\llbracket \text{output-fun } c = \varepsilon;$
i-Streams-Connected
 $(\text{output-fun } \circ (i\text{-Exec-Comp-Stream trans-fun input } c))$
 $\text{channel } \rrbracket \Longrightarrow$
 $\text{output-fun } \circ (i\text{-Exec-Comp-Stream-Init trans-fun input } c) = \text{channel}$
 ⟨proof⟩

lemma *f-Exec-Stream-output-causal*:
 $\llbracket xs \downarrow n = ys \downarrow n;$
 $\text{output1} = \text{map output-fun } (f\text{-Exec-Comp-Stream trans-fun } xs \ c);$
 $\text{output2} = \text{map output-fun } (f\text{-Exec-Comp-Stream trans-fun } ys \ c) \rrbracket \Longrightarrow$
 $\text{output1 } \downarrow n = \text{output2 } \downarrow n$
 ⟨proof⟩

lemma *f-Exec-Stream-Init-output-strictly-causal*:
 $\llbracket xs \downarrow n = ys \downarrow n;$
 $\text{output1} = \text{map output-fun } (f\text{-Exec-Comp-Stream-Init trans-fun } xs \ c);$
 $\text{output2} = \text{map output-fun } (f\text{-Exec-Comp-Stream-Init trans-fun } ys \ c) \rrbracket \Longrightarrow$
 $\text{output1 } \downarrow \text{Suc } n = \text{output2 } \downarrow \text{Suc } n$
 ⟨proof⟩

lemma *i-Exec-Stream-output-causal*:
 $\llbracket \text{input1 } \Downarrow n = \text{input2 } \Downarrow n;$
 $\text{output1} = \text{output-fun } \circ i\text{-Exec-Comp-Stream trans-fun input1 } c;$
 $\text{output2} = \text{output-fun } \circ i\text{-Exec-Comp-Stream trans-fun input2 } c \rrbracket \Longrightarrow$
 $\text{output1 } \Downarrow n = \text{output2 } \Downarrow n$
 ⟨proof⟩

lemma *i-Exec-Stream-Init-output-strictly-causal*:

$$\llbracket \text{input1} \Downarrow n = \text{input2} \Downarrow n;$$

$$\text{output1} = \text{output-fun} \circ \text{i-Exec-Comp-Stream-Init trans-fun input1 c};$$

$$\text{output2} = \text{output-fun} \circ \text{i-Exec-Comp-Stream-Init trans-fun input2 c} \rrbracket \implies$$

$$\text{output1} \Downarrow \text{Suc } n = \text{output2} \Downarrow \text{Suc } n$$
 <proof>

lemma *f-Exec-Stream-Connected-strictly-causal*:

$$\llbracket \text{xs} \Downarrow n = \text{ys} \Downarrow n;$$

$$\text{f-Streams-Connected}$$

$$(\text{map output-fun (f-Exec-Comp-Stream trans-fun xs c)})$$

$$\text{channel1};$$

$$\text{f-Streams-Connected}$$

$$(\text{map output-fun (f-Exec-Comp-Stream trans-fun ys c)})$$

$$\text{channel2} \rrbracket \implies$$

$$\text{channel1} \Downarrow \text{Suc } n = \text{channel2} \Downarrow \text{Suc } n$$
 <proof>

lemma *i-Exec-Stream-Connected-strictly-causal*:

$$\llbracket \text{input1} \Downarrow n = \text{input2} \Downarrow n;$$

$$\text{i-Streams-Connected}$$

$$(\text{portOutput} \circ (\text{i-Exec-Comp-Stream trans-fun input1 c}))$$

$$\text{channel1};$$

$$\text{i-Streams-Connected}$$

$$(\text{portOutput} \circ (\text{i-Exec-Comp-Stream trans-fun input2 c}))$$

$$\text{channel2} \rrbracket \implies$$

$$\text{channel1} \Downarrow \text{Suc } n = \text{channel2} \Downarrow \text{Suc } n$$
 <proof>

A predicate for the semantics with initial state in result stream indicating for two message streams that the ports, they correspond to, are connected.

definition *f-Streams-Connected-Init* :: 'a fstream-af \Rightarrow 'a fstream-af \Rightarrow bool
where *f-Streams-Connected-Init* outS inS \equiv inS = outS

definition *i-Streams-Connected-Init* :: 'a istream-af \Rightarrow 'a istream-af \Rightarrow bool
where *i-Streams-Connected-Init* outS inS \equiv inS = outS

lemmas *Streams-Connected-Init-defs* =

f-Streams-Connected-Init-def
i-Streams-Connected-Init-def

lemma *f-Streams-Connected-Init-nth-conv*:

$$\text{f-Streams-Connected-Init outS inS} =$$

$$(\text{length inS} = \text{length outS} \wedge (\forall i < \text{length inS}. \text{inS} ! i = \text{outS} ! i))$$
 <proof>

lemma *i-Streams-Connected-Init-nth-conv*:

$$\text{i-Streams-Connected-Init outS inS} =$$

$$(\forall i. \text{inS} i = \text{outS} i)$$
 <proof>

lemma *f-Exec-Stream-Init-eq-output-channel2*:

$$\begin{aligned} & \llbracket \text{output-fun } c = \varepsilon; \\ & \quad \text{f-Streams-Connected-Init} \\ & \quad (\text{map output-fun } (f\text{-Exec-Comp-Stream-Init trans-fun } xs \ c)) \\ & \quad \text{channel} \rrbracket \implies \\ & \text{map output-fun } (f\text{-Exec-Comp-Stream-Init trans-fun } xs \ c) = \text{channel} \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *i-Exec-Stream-Init-eq-output-channel2*:

$$\begin{aligned} & \llbracket \text{output-fun } c = \varepsilon; \\ & \quad \text{i-Streams-Connected-Init} \\ & \quad (\text{output-fun} \circ (\text{i-Exec-Comp-Stream-Init trans-fun input } c)) \\ & \quad \text{channel} \rrbracket \implies \\ & \text{output-fun} \circ (\text{i-Exec-Comp-Stream-Init trans-fun input } c) = \text{channel} \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *f-Exec-Stream-Connected-Init-strictly-causal*:

$$\begin{aligned} & \llbracket xs \downarrow n = ys \downarrow n; \\ & \quad \text{f-Streams-Connected-Init} \\ & \quad (\text{map output-fun } (f\text{-Exec-Comp-Stream-Init trans-fun } xs \ c)) \\ & \quad \text{channel1}; \\ & \quad \text{f-Streams-Connected-Init} \\ & \quad (\text{map output-fun } (f\text{-Exec-Comp-Stream-Init trans-fun } ys \ c)) \\ & \quad \text{channel2} \rrbracket \implies \\ & \text{channel1} \downarrow \text{Suc } n = \text{channel2} \downarrow \text{Suc } n \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *i-Exec-Stream-Connected-Init-strictly-causal*:

$$\begin{aligned} & \llbracket \text{input1} \Downarrow n = \text{input2} \Downarrow n; \\ & \quad \text{i-Streams-Connected-Init} \\ & \quad (\text{portOutput} \circ (\text{i-Exec-Comp-Stream-Init trans-fun input1 } c)) \\ & \quad \text{channel1}; \\ & \quad \text{i-Streams-Connected-Init} \\ & \quad (\text{portOutput} \circ (\text{i-Exec-Comp-Stream-Init trans-fun input2 } c)) \\ & \quad \text{channel2} \rrbracket \implies \\ & \text{channel1} \Downarrow \text{Suc } n = \text{channel2} \Downarrow \text{Suc } n \\ & \langle \text{proof} \rangle \end{aligned}$$

3.1.4 Additional auxiliary results

The following lemma shows that, if the system state is different at some time points with respect to a certain predicate P , then there exists a defined time point between these two, where the state change has taken place

lemma *f-State-Change-exists-set*:

$$\begin{aligned} & \llbracket n1 \leq n2; n1 \in I; n2 \in I; \\ & \quad \neg P (f\text{-Exec-Comp trans-fun } (\text{input} \downarrow n1) \ c); \\ & \quad P (f\text{-Exec-Comp trans-fun } (\text{input} \downarrow n2) \ c) \rrbracket \implies \\ & \exists n \in I. n1 \leq n \wedge n < n2 \wedge \end{aligned}$$

$\neg P (f\text{-Exec-Comp trans-fun (input } \downarrow n) c) \wedge$
 $P (f\text{-Exec-Comp trans-fun (input } \downarrow (\text{inext } n I)) c)$
 ⟨proof⟩

lemma *f-State-Change-exists*:

$\llbracket n1 \leq n2;$
 $\neg P (f\text{-Exec-Comp trans-fun (input } \downarrow n1) c);$
 $P (f\text{-Exec-Comp trans-fun (input } \downarrow n2) c) \rrbracket \implies$
 $\exists n \geq n1. n < n2 \wedge$
 $\neg P (f\text{-Exec-Comp trans-fun (input } \downarrow n) c) \wedge$
 $P (f\text{-Exec-Comp trans-fun (input } \downarrow (\text{Suc } n)) c)$
 ⟨proof⟩

lemma *i-State-Change-exists-set*:

$\llbracket n1 \leq n2; n1 \in I; n2 \in I;$
 $\neg P (i\text{-Exec-Comp-Stream trans-fun input } c n1);$
 $P (i\text{-Exec-Comp-Stream trans-fun input } c n2) \rrbracket \implies$
 $\exists n \in I. n1 \leq n \wedge n < n2 \wedge$
 $\neg P (i\text{-Exec-Comp-Stream trans-fun input } c n) \wedge$
 $P (i\text{-Exec-Comp-Stream trans-fun input } c (\text{inext } n I))$
 ⟨proof⟩

lemma *i-State-Change-exists*:

$\llbracket n1 \leq n2;$
 $\neg P (i\text{-Exec-Comp-Stream trans-fun input } c n1);$
 $P (i\text{-Exec-Comp-Stream trans-fun input } c n2) \rrbracket \implies$
 $\exists n \geq n1. n < n2 \wedge$
 $\neg P (i\text{-Exec-Comp-Stream trans-fun input } c n) \wedge$
 $P (i\text{-Exec-Comp-Stream trans-fun input } c (\text{Suc } n))$
 ⟨proof⟩

lemma *i-State-Change-Init-exists-set*:

$\llbracket n1 \leq n2; n1 \in I; n2 \in I;$
 $\neg P (i\text{-Exec-Comp-Stream-Init trans-fun input } c n1);$
 $P (i\text{-Exec-Comp-Stream-Init trans-fun input } c n2) \rrbracket \implies$
 $\exists n \in I. n1 \leq n \wedge n < n2 \wedge$
 $\neg P (i\text{-Exec-Comp-Stream-Init trans-fun input } c n) \wedge$
 $P (i\text{-Exec-Comp-Stream-Init trans-fun input } c (\text{inext } n I))$
 ⟨proof⟩

lemma *i-State-Change-Init-exists*:

$\llbracket n1 \leq n2;$
 $\neg P (i\text{-Exec-Comp-Stream-Init trans-fun input } c n1);$
 $P (i\text{-Exec-Comp-Stream-Init trans-fun input } c n2) \rrbracket \implies$
 $\exists n \geq n1. n < n2 \wedge$
 $\neg P (i\text{-Exec-Comp-Stream-Init trans-fun input } c n) \wedge$
 $P (i\text{-Exec-Comp-Stream-Init trans-fun input } c (\text{Suc } n))$
 ⟨proof⟩

3.2 Components with accelerated execution

This section deals with variable execution speed components. A component accelerated by a (clocking) factor k processes streams expanded by factor k and its output streams are compressed by factor k .

3.2.1 Equivalence relation for executions

A predicate indicating for two components together with transition functions and a given equivalence predicate for their local states, that the components exhibit equivalent observable behaviour after expanding input streams and shrinking output streams by a constant factor, given that their local states are equivalent with respect to the specified equivalence relations.

definition

Equiv-Exec ::

'input \Rightarrow
 ('state1 \Rightarrow 'state2 \Rightarrow bool) \Rightarrow — Equivalence predicate for local states
 ('comp1, 'state1) *Comp-Local-State* \Rightarrow
 ('comp2, 'state2) *Comp-Local-State* \Rightarrow
 ('input, 'input1) *Port-Input-Value* \Rightarrow — Input adaptor for first component
 ('input, 'input2) *Port-Input-Value* \Rightarrow — Input adaptor for second component
 ('comp1, 'output) *Port-Output-Value* \Rightarrow
 ('comp2, 'output) *Port-Output-Value* \Rightarrow
 ('comp1, 'input1 message-af) *Comp-Trans-Fun* \Rightarrow
 ('comp2, 'input2 message-af) *Comp-Trans-Fun* \Rightarrow
 nat \Rightarrow nat \Rightarrow 'comp1 \Rightarrow 'comp2 \Rightarrow bool

where

Equiv-Exec

m *equiv-states*

localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2
 trans-fun1 trans-fun2 k1 k2 c1 c2 \equiv

equiv-states (localState1 c1) (localState2 c2) \longrightarrow (
 last-message (map output-fun1 (
 f-Exec-Comp-Stream trans-fun1 (input-fun1 m # $\varepsilon^{k1} - \text{Suc } 0$) c1)) =
 last-message (map output-fun2 (
 f-Exec-Comp-Stream trans-fun2 (input-fun2 m # $\varepsilon^{k2} - \text{Suc } 0$) c2)) \wedge
equiv-states
 (localState1 (f-Exec-Comp trans-fun1 (input-fun1 m # $\varepsilon^{k1} - \text{Suc } 0$) c1))
 (localState2 (f-Exec-Comp trans-fun2 (input-fun2 m # $\varepsilon^{k2} - \text{Suc } 0$) c2)))

Predicate indicating for two components together with transition functions and a given equivalence predicate for their local states, that the equivalence predicate is stable with respect to component execution, i.e., it determines the equivalence of components' local states both for the initial states and after the components have processed an arbitrary input. The restricting version *Equiv-Exec-stable-set* guarantees stability only for inputs from a given restriction set, the not-restricting version guarantees stability for all inputs.

definition

Equiv-Exec-stable-set ::

'input set \Rightarrow
('state1 \Rightarrow *'state2* \Rightarrow *bool)* \Rightarrow — Equivalence predicate for local states
('comp1, 'state1) Comp-Local-State \Rightarrow
('comp2, 'state2) Comp-Local-State \Rightarrow
('input, 'input1) Port-Input-Value \Rightarrow — Input adaptor for first component
('input, 'input2) Port-Input-Value \Rightarrow — Input adaptor for second component
('comp1, 'output) Port-Output-Value \Rightarrow
('comp2, 'output) Port-Output-Value \Rightarrow
('comp1, 'input1 message-af) Comp-Trans-Fun \Rightarrow
('comp2, 'input2 message-af) Comp-Trans-Fun \Rightarrow
nat \Rightarrow *nat* \Rightarrow *'comp1* \Rightarrow *'comp2* \Rightarrow *bool*

where

Equiv-Exec-stable-set A

equiv-states localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2
trans-fun1 trans-fun2 k1 k2 c1 c2 \equiv
 \forall *input m. set input* \subseteq *A* \wedge *m* \in *A* \longrightarrow
Equiv-Exec m
equiv-states localState1 localState2 input-fun1 input-fun2 output-fun1 out-
put-fun2
trans-fun1 trans-fun2 k1 k2
(f-Exec-Comp trans-fun1 (map input-fun1 input \odot_f k1) c1)
(f-Exec-Comp trans-fun2 (map input-fun2 input \odot_f k2) c2)

definition

Equiv-Exec-stable ::

('state1 \Rightarrow *'state2* \Rightarrow *bool)* \Rightarrow — Equivalence predicate for local states
('comp1, 'state1) Comp-Local-State \Rightarrow
('comp2, 'state2) Comp-Local-State \Rightarrow
('input, 'input1) Port-Input-Value \Rightarrow — Input adaptor for first component
('input, 'input2) Port-Input-Value \Rightarrow — Input adaptor for second component
('comp1, 'output) Port-Output-Value \Rightarrow
('comp2, 'output) Port-Output-Value \Rightarrow
('comp1, 'input1 message-af) Comp-Trans-Fun \Rightarrow
('comp2, 'input2 message-af) Comp-Trans-Fun \Rightarrow
nat \Rightarrow *nat* \Rightarrow *'comp1* \Rightarrow *'comp2* \Rightarrow *bool*

where

Equiv-Exec-stable

equiv-states localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2
trans-fun1 trans-fun2 k1 k2 c1 c2 \equiv
 \forall *input m.*
Equiv-Exec m
equiv-states localState1 localState2 input-fun1 input-fun2 output-fun1 out-
put-fun2
trans-fun1 trans-fun2 k1 k2
(f-Exec-Comp trans-fun1 (map input-fun1 input \odot_f k1) c1)
(f-Exec-Comp trans-fun2 (map input-fun2 input \odot_f k2) c2)

lemma *Equiv-Exec-equiv-statesI*:

\llbracket *equiv-states* (*localState1* *c1*) (*localState2* *c2*);
Equiv-Exec
m equiv-states
localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2
trans-fun1 trans-fun2 k1 k2 c1 c2 $\rrbracket \implies$
equiv-states
(*localState1* (*f-Exec-Comp trans-fun1* (*input-fun1 m* $\# \varepsilon^{k1} - \text{Suc } 0$) *c1*))
(*localState2* (*f-Exec-Comp trans-fun2* (*input-fun2 m* $\# \varepsilon^{k2} - \text{Suc } 0$) *c2*))
⟨*proof*⟩

lemma *Equiv-Exec-output-eqI*:

\llbracket *equiv-states* (*localState1* *c1*) (*localState2* *c2*);
Equiv-Exec
m equiv-states
localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2
trans-fun1 trans-fun2 k1 k2 c1 c2 $\rrbracket \implies$
last-message (*map output-fun1* (
f-Exec-Comp-Stream trans-fun1 (*input-fun1 m* $\# \varepsilon^{k1} - \text{Suc } 0$) *c1*)) =
last-message (*map output-fun2* (
f-Exec-Comp-Stream trans-fun2 (*input-fun2 m* $\# \varepsilon^{k2} - \text{Suc } 0$) *c2*))
⟨*proof*⟩

lemma *Equiv-Exec-equiv-statesI'*:

\llbracket *equiv-states* (*localState1* *c1*) (*localState2* *c2*);
Equiv-Exec
m equiv-states
localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2
trans-fun1 trans-fun2 k1 k2 c1 c2 $\rrbracket \implies$
equiv-states
(*localState1* (*f-Exec-Comp trans-fun1 NoMsg*^{*k1*} $- \text{Suc } 0$ (*trans-fun1* (*input-fun1*
m) *c1*)))
(*localState2* (*f-Exec-Comp trans-fun2 NoMsg*^{*k2*} $- \text{Suc } 0$ (*trans-fun2* (*input-fun2*
m) *c2*)))
⟨*proof*⟩

lemma *Equiv-Exec-le1*:

\llbracket *k1* \leq *Suc 0*; *k2* \leq *Suc 0*;
equiv-states (*localState1* *c1*) (*localState2* *c2*);
Equiv-Exec m
equiv-states localState1 localState2 input-fun1 input-fun2 output-fun1 out-
put-fun2
trans-fun1 trans-fun2 k1 k2 c1 c2 $\rrbracket \implies$
output-fun1 (*trans-fun1* (*input-fun1 m*) *c1*) =
output-fun2 (*trans-fun2* (*input-fun2 m*) *c2*) \wedge
equiv-states
(*localState1* (*trans-fun1* (*input-fun1 m*) *c1*))
(*localState2* (*trans-fun2* (*input-fun2 m*) *c2*))
⟨*proof*⟩

lemma *Equiv-Exec-stable-set-UNIV*:

Equiv-Exec-stable-set
UNIV equiv-states
localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2
trans-fun1 trans-fun2 k1 k2 c1 c2 =
Equiv-Exec-stable
equiv-states
localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2
trans-fun1 trans-fun2 k1 k2 c1 c2
 ⟨proof⟩

lemma *Equiv-Exec-stable-setI*:

[[*Equiv-Exec-stable-set A*
equiv-states localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2
trans-fun1 trans-fun2 k1 k2 c1 c2;
set input \subseteq *A*; *m* \in *A*]] \implies
Equiv-Exec
m equiv-states
localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2
trans-fun1 trans-fun2 k1 k2
 (*f-Exec-Comp trans-fun1 (map input-fun1 input* \odot_f *k1) c1*)
 (*f-Exec-Comp trans-fun2 (map input-fun2 input* \odot_f *k2) c2*)
 ⟨proof⟩

lemma *Equiv-Exec-stableI*:

Equiv-Exec-stable
equiv-states localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2
trans-fun1 trans-fun2 k1 k2 c1 c2 \implies
Equiv-Exec m
equiv-states localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2
trans-fun1 trans-fun2 k1 k2
 (*f-Exec-Comp trans-fun1 (map input-fun1 input* \odot_f *k1) c1*)
 (*f-Exec-Comp trans-fun2 (map input-fun2 input* \odot_f *k2) c2*)
 ⟨proof⟩

Reflexivity, symmetry and transitivity results for *Equiv-Exec*

lemma *Equiv-Exec-refl*:

[[$\bigwedge c.$ *equiv-states (localState c) (localState c)*]] \implies
Equiv-Exec
m equiv-states
localState localState input-fun input-fun output-fun output-fun
trans-fun trans-fun k k c c
 ⟨proof⟩

lemma *Equiv-Exec-sym[rule-format]*:

[[$\forall c1 c2.$
equiv-states (localState1 c1) (localState2 c2) =

$\text{equiv-states } (\text{localState2 } c2) (\text{localState1 } c1) \text{] } \implies$
Equiv-Exec
 $m \text{ equiv-states}$
 $\text{localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2}$
 $\text{trans-fun1 trans-fun2 k1 k2 c1 c2} =$
Equiv-Exec
 $m \text{ equiv-states}$
 $\text{localState2 localState1 input-fun2 input-fun1 output-fun2 output-fun1}$
 $\text{trans-fun2 trans-fun1 k2 k1 c2 c1}$
 <proof>

lemma *Equiv-Exec-sym2*:

$\llbracket \text{equiv-states-sym} = (\lambda s1 s2. \text{equiv-states } s2 s1) \rrbracket \implies$
Equiv-Exec
 $m \text{ equiv-states}$
 $\text{localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2}$
 $\text{trans-fun1 trans-fun2 k1 k2 c1 c2} =$
Equiv-Exec
 $m \text{ equiv-states-sym}$
 $\text{localState2 localState1 input-fun2 input-fun1 output-fun2 output-fun1}$
 $\text{trans-fun2 trans-fun1 k2 k1 c2 c1}$
 <proof>

lemma *Equiv-Exec-sym2-ex*:

$\exists \text{equiv-states-sym.}$
Equiv-Exec
 $m \text{ equiv-states}$
 $\text{localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2}$
 $\text{trans-fun1 trans-fun2 k1 k2 c1 c2} =$
Equiv-Exec
 $m \text{ equiv-states-sym}$
 $\text{localState2 localState1 input-fun2 input-fun1 output-fun2 output-fun1}$
 $\text{trans-fun2 trans-fun1 k2 k1 c2 c1}$
 <proof>

lemma *Equiv-Exec-trans*:

$\llbracket \text{Equiv-Exec}$
 $m \text{ equiv-states12}$
 $\text{localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2}$
 $\text{trans-fun1 trans-fun2 k1 k2 c1 c2};$
Equiv-Exec
 $m \text{ equiv-states23}$
 $\text{localState2 localState3 input-fun2 input-fun3 output-fun2 output-fun3}$
 $\text{trans-fun2 trans-fun3 k2 k3 c2 c3};$
 $\text{equiv-states13} = (\lambda s1 s3. ($
 $\text{if } s1 = \text{localState1 } c1 \wedge s3 = \text{localState3 } c3 \text{ then}$
 $\text{equiv-states12 } s1 (\text{localState2 } c2) \wedge$
 $\text{equiv-states23 } (\text{localState2 } c2) s3$
 else

$$\begin{aligned}
& \text{equiv-states12 } s1 \ (\\
& \quad \text{localState2 } (f\text{-Exec-Comp } \text{trans-fun2 } (\text{input-fun2 } m \# \varepsilon^{k2} - \text{Suc } 0) \ c2)) \\
\wedge & \\
& \text{equiv-states23 } (\\
& \quad \text{localState2 } (f\text{-Exec-Comp } \text{trans-fun2 } (\text{input-fun2 } m \# \varepsilon^{k2} - \text{Suc } 0) \ c2)) \\
s3) \] \implies & \\
& \text{Equiv-Exec} \\
& \quad m \ \text{equiv-states13} \\
& \quad \text{localState1 } \text{localState3 } \text{input-fun1 } \text{input-fun3 } \text{output-fun1 } \text{output-fun3} \\
& \quad \text{trans-fun1 } \text{trans-fun3 } k1 \ k3 \ c1 \ c3 \\
\langle \text{proof} \rangle &
\end{aligned}$$

lemma *Equiv-Exec-trans-ex*:

$$\begin{aligned}
& \llbracket \text{Equiv-Exec} \\
& \quad m \ \text{equiv-states12} \\
& \quad \text{localState1 } \text{localState2 } \text{input-fun1 } \text{input-fun2 } \text{output-fun1 } \text{output-fun2} \\
& \quad \text{trans-fun1 } \text{trans-fun2 } k1 \ k2 \ c1 \ c2; \\
& \text{Equiv-Exec} \\
& \quad m \ \text{equiv-states23} \\
& \quad \text{localState2 } \text{localState3 } \text{input-fun2 } \text{input-fun3 } \text{output-fun2 } \text{output-fun3} \\
& \quad \text{trans-fun2 } \text{trans-fun3 } k2 \ k3 \ c2 \ c3 \] \implies \\
& \exists \text{equiv-states13. } \text{Equiv-Exec} \\
& \quad m \ \text{equiv-states13} \\
& \quad \text{localState1 } \text{localState3 } \text{input-fun1 } \text{input-fun3 } \text{output-fun1 } \text{output-fun3} \\
& \quad \text{trans-fun1 } \text{trans-fun3 } k1 \ k3 \ c1 \ c3 \\
\langle \text{proof} \rangle &
\end{aligned}$$

A predicate indicating for a given local state extraction function and a given transition function, that components, whose states are equal with regard to the local state extraction function, are transformed into equal components, when the transition function is applied with the same input.

definition *Exec-Equal-State* ::

$$('comp, 'state) \text{Comp-Local-State} \Rightarrow ('comp, 'input \text{message-af}) \text{Comp-Trans-Fun} \Rightarrow \text{bool}$$

where *Exec-Equal-State* *localState* *trans-fun* \equiv

$$\forall c1 \ c2 \ m. \ \text{localState } c1 = \text{localState } c2 \longrightarrow \text{trans-fun } m \ c1 = \text{trans-fun } m \ c2$$

lemma *Exec-Equal-StateD*:

$$\begin{aligned}
& \llbracket \text{Exec-Equal-State } \text{localState } \text{trans-fun}; \\
& \quad \text{localState } c1 = \text{localState } c2 \] \implies \\
& \quad \text{trans-fun } m \ c1 = \text{trans-fun } m \ c2 \\
\langle \text{proof} \rangle &
\end{aligned}$$

lemma *Exec-Equal-StateD'*:

$$\begin{aligned}
& \text{Exec-Equal-State } \text{localState } \text{trans-fun} \implies \\
& \quad \forall c1 \ c2 \ m. \ \text{localState } c1 = \text{localState } c2 \longrightarrow \text{trans-fun } m \ c1 = \text{trans-fun } m \ c2 \\
\langle \text{proof} \rangle &
\end{aligned}$$

lemma *Exec-Equal-StateI*:

$(\bigwedge c1\ c2\ m.\ localState\ c1 = localState\ c2 \implies trans-fun\ m\ c1 = trans-fun\ m\ c2)$
 $\implies Exec-Equal-State\ localState\ trans-fun$
 <proof>

lemma *f-Exec-Equal-State*: $\bigwedge c1\ c2.$
 $\llbracket Exec-Equal-State\ localState\ trans-fun;$
 $localState\ c1 = localState\ c2; xs \neq [] \rrbracket \implies$
 $f-Exec-Comp\ trans-fun\ xs\ c1 = f-Exec-Comp\ trans-fun\ xs\ c2$
 <proof>

lemma *f-Exec-Stream-Equal-State*:
 $\llbracket Exec-Equal-State\ localState\ trans-fun;$
 $localState\ c1 = localState\ c2 \rrbracket \implies$
 $f-Exec-Comp-Stream\ trans-fun\ xs\ c1 =$
 $f-Exec-Comp-Stream\ trans-fun\ xs\ c2$
 <proof>

lemma *i-Exec-Stream-Equal-State*:
 $\llbracket Exec-Equal-State\ localState\ trans-fun;$
 $localState\ c1 = localState\ c2 \rrbracket \implies$
 $i-Exec-Comp-Stream\ trans-fun\ input\ c1 =$
 $i-Exec-Comp-Stream\ trans-fun\ input\ c2$
 <proof>

3.2.2 Idle states

definition *State-Idle* ::
 $('comp, 'state) Comp-Local-State \Rightarrow ('comp \Rightarrow 'output\ message-af) \Rightarrow$
 $('comp, 'input\ message-af) Comp-Trans-Fun \Rightarrow 'state \Rightarrow bool$
where *State-Idle* $localState\ output-fun\ trans-fun\ state \equiv$
 $\forall c.\ localState\ c = state \longrightarrow$
 $localState\ (trans-fun\ \varepsilon\ c) = state \wedge$
 $output-fun\ (trans-fun\ \varepsilon\ c) = \varepsilon$

lemma *State-IdleD*:
 $\llbracket State-Idle\ localState\ output-fun\ trans-fun\ state;$
 $localState\ c = state \rrbracket \implies$
 $localState\ (trans-fun\ \varepsilon\ c) = state \wedge$
 $output-fun\ (trans-fun\ \varepsilon\ c) = \varepsilon$
 <proof>

lemma *State-IdleD'*:
 $State-Idle\ localState\ output-fun\ trans-fun\ state \implies$
 $\forall c.\ localState\ c = state \longrightarrow$
 $localState\ (trans-fun\ \varepsilon\ c) = state \wedge$
 $output-fun\ (trans-fun\ \varepsilon\ c) = \varepsilon$
 <proof>

lemma *State-IdleI*:

$$\llbracket \bigwedge c. \text{localState } c = \text{state} \implies$$

$$\text{localState } (\text{trans-fun } \varepsilon \ c) = \text{state} \wedge$$

$$\text{output-fun } (\text{trans-fun } \varepsilon \ c) = \varepsilon \rrbracket \implies$$

$$\text{State-Idle localState output-fun trans-fun state}$$

<proof>

lemma *State-Idle-step*[rule-format]:

$$\llbracket \text{State-Idle localState output-fun trans-fun (localState } c) \rrbracket \implies$$

$$\text{State-Idle localState output-fun trans-fun (localState (trans-fun } \varepsilon \ c))$$

<proof>

lemma *f-Exec-State-Idle-replicate-NoMsg-state*[rule-format]:

$$\bigwedge c. \text{State-Idle localState output-fun trans-fun (localState } c) \implies$$

$$\text{localState (f-Exec-Comp trans-fun } \varepsilon^n \ c) = \text{localState } c$$

<proof>

lemma *f-Exec-State-Idle-replicate-NoMsg-gr0-output*[rule-format]: $\bigwedge c.$

$$\llbracket \text{State-Idle localState output-fun trans-fun (localState } c); 0 < n \rrbracket \implies$$

$$\text{output-fun (f-Exec-Comp trans-fun } \varepsilon^n \ c) = \varepsilon$$

<proof>

lemma *f-Exec-State-Idle-replicate-NoMsg-output*[rule-format]:

$$\llbracket \text{State-Idle localState output-fun trans-fun (localState } c);$$

$$\text{output-fun } c = \varepsilon \rrbracket \implies$$

$$\text{output-fun (f-Exec-Comp trans-fun } \varepsilon^n \ c) = \varepsilon$$

<proof>

lemma *f-Exec-Stream-State-Idle-replicate-NoMsg-output*[rule-format]:

$$\llbracket \text{State-Idle localState output-fun trans-fun (localState } c) \rrbracket \implies$$

$$\text{map output-fun (f-Exec-Comp-Stream trans-fun } \varepsilon^n \ c) = \varepsilon^n$$

<proof>

corollary *f-Exec-State-Idle-append-replicate-NoMsg-state*:

$$\llbracket \text{State-Idle localState output-fun trans-fun (}$$

$$\text{localState (f-Exec-Comp trans-fun } xs \ c)) \rrbracket \implies$$

$$\text{localState (f-Exec-Comp trans-fun (xs @ } \varepsilon^n) \ c) =$$

$$\text{localState (f-Exec-Comp trans-fun } xs \ c)$$

<proof>

corollary *f-Exec-State-Idle-append-replicate-NoMsg-ge-state*:

$$\llbracket \text{State-Idle localState output-fun trans-fun (}$$

$$\text{localState (f-Exec-Comp trans-fun (xs @ } \varepsilon^m) \ c));$$

$$m \leq n \rrbracket \implies$$

$$\text{localState (f-Exec-Comp trans-fun (xs @ } \varepsilon^n) \ c) =$$

$$\text{localState (f-Exec-Comp trans-fun (xs @ } \varepsilon^m) \ c)$$

<proof>

corollary *f-Exec-State-Idle-replicate-NoMsg-ge-state*:

$$\llbracket \text{State-Idle localState output-fun trans-fun (}$$

$$\text{localState (f-Exec-Comp trans-fun } \varepsilon^m \text{ c));}$$

$$m \leq n \rrbracket \implies$$

$$\text{localState (f-Exec-Comp trans-fun } \varepsilon^n \text{ c) =}$$

$$\text{localState (f-Exec-Comp trans-fun } \varepsilon^m \text{ c)}$$

<proof>

corollary *f-Exec-State-Idle-append-replicate-NoMsg-gr0-output:*

$$\llbracket \text{State-Idle localState output-fun trans-fun (}$$

$$\text{localState (f-Exec-Comp trans-fun } xs \text{ c));}$$

$$0 < n \rrbracket \implies$$

$$\text{output-fun (f-Exec-Comp trans-fun (} xs \text{ @ } \varepsilon^n \text{) c) = } \varepsilon$$

<proof>

corollary *f-Exec-Stream-State-Idle-append-replicate-NoMsg-gr0-output:*

$$\llbracket \text{State-Idle localState output-fun trans-fun (}$$

$$\text{localState (f-Exec-Comp trans-fun } xs \text{ c)) \rrbracket \implies$$

$$\text{map output-fun (f-Exec-Comp-Stream trans-fun (} xs \text{ @ } \varepsilon^n \text{) c) =}$$

$$\text{map output-fun (f-Exec-Comp-Stream trans-fun } xs \text{ c) @ } \varepsilon^n$$

<proof>

corollary *f-Exec-State-Idle-append-replicate-NoMsg-gr-output:*

$$\llbracket \text{State-Idle localState output-fun trans-fun (}$$

$$\text{localState (f-Exec-Comp trans-fun (} xs \text{ @ } \varepsilon^m \text{) c));}$$

$$m < n \rrbracket \implies$$

$$\text{output-fun (f-Exec-Comp trans-fun (} xs \text{ @ } \varepsilon^n \text{) c) = } \varepsilon$$

<proof>

corollary *f-Exec-State-Idle-append-replicate-NoMsg-ge-output:*

$$\llbracket \text{State-Idle localState output-fun trans-fun (}$$

$$\text{localState (f-Exec-Comp trans-fun (} xs \text{ @ } \varepsilon^m \text{) c));}$$

$$\text{output-fun (f-Exec-Comp trans-fun (} xs \text{ @ } \varepsilon^m \text{) c) = } \varepsilon; m \leq n \rrbracket \implies$$

$$\text{output-fun (f-Exec-Comp trans-fun (} xs \text{ @ } \varepsilon^n \text{) c) = } \varepsilon$$

<proof>

corollary *f-Exec-State-Idle-replicate-NoMsg-gr-output:*

$$\llbracket \text{State-Idle localState output-fun trans-fun (}$$

$$\text{localState (f-Exec-Comp trans-fun } \varepsilon^m \text{ c));}$$

$$m < n \rrbracket \implies$$

$$\text{output-fun (f-Exec-Comp trans-fun } \varepsilon^n \text{ c) = } \varepsilon$$

<proof>

corollary *f-Exec-State-Idle-replicate-NoMsg-ge-output:*

$$\llbracket \text{State-Idle localState output-fun trans-fun (}$$

$$\text{localState (f-Exec-Comp trans-fun } \varepsilon^m \text{ c));}$$

$$\text{output-fun (f-Exec-Comp trans-fun } \varepsilon^m \text{ c) = } \varepsilon; m \leq n \rrbracket \implies$$

$$\text{output-fun (f-Exec-Comp trans-fun } \varepsilon^n \text{ c) = } \varepsilon$$

<proof>

lemma *State-Idle-append-replicate-NoMsg-output-last-message:*

$$\begin{aligned} & \llbracket \text{State-Idle } \text{localState } \text{output-fun } \text{trans-fun } (\\ & \quad \text{localState } (f\text{-Exec-Comp } \text{trans-fun } xs \ c)) \rrbracket \implies \\ & \text{last-message } (\text{map } \text{output-fun } (f\text{-Exec-Comp-Stream } \text{trans-fun } (xs \ @ \ \varepsilon^n) \ c)) = \\ & \text{last-message } (\text{map } \text{output-fun } (f\text{-Exec-Comp-Stream } \text{trans-fun } xs \ c)) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *State-Idle-append-replicate-NoMsg-output-Msg-eq-last-message:*

$$\begin{aligned} & \llbracket \text{State-Idle } \text{localState } \text{output-fun } \text{trans-fun } (\\ & \quad \text{localState } (f\text{-Exec-Comp } \text{trans-fun } xs \ c)); \\ & \quad \text{output-fun } (f\text{-Exec-Comp } \text{trans-fun } xs \ c) \neq \varepsilon; \\ & \quad xs \neq [] \rrbracket \implies \\ & \text{last-message } (\text{map } \text{output-fun } (f\text{-Exec-Comp-Stream } \text{trans-fun } (xs \ @ \ \varepsilon^n) \ c)) = \\ & \text{output-fun } (f\text{-Exec-Comp } \text{trans-fun } xs \ c) \\ & \langle \text{proof} \rangle \end{aligned}$$

corollary *State-Idle-output-Msg-eq-last-message:*

$$\begin{aligned} & \llbracket \text{State-Idle } \text{localState } \text{output-fun } \text{trans-fun } (\\ & \quad \text{localState } (f\text{-Exec-Comp } \text{trans-fun } xs \ c)); \\ & \quad \text{output-fun } (f\text{-Exec-Comp } \text{trans-fun } xs \ c) \neq \varepsilon; \\ & \quad xs \neq [] \rrbracket \implies \\ & \text{last-message } (\text{map } \text{output-fun } (f\text{-Exec-Comp-Stream } \text{trans-fun } xs \ c)) = \\ & \text{output-fun } (f\text{-Exec-Comp } \text{trans-fun } xs \ c) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *State-Idle-imp-exists-state-change:*

$$\begin{aligned} & \llbracket \neg \text{State-Idle } \text{localState } \text{output-fun } \text{trans-fun } (\text{localState } c); \\ & \quad \text{State-Idle } \text{localState } \text{output-fun } \text{trans-fun } (\text{localState } (f\text{-Exec-Comp } \text{trans-fun } \varepsilon^n \\ & \quad c)) \rrbracket \implies \\ & \exists i < n. (\\ & \quad \neg \text{State-Idle } \text{localState } \text{output-fun } \text{trans-fun } (\text{localState } (f\text{-Exec-Comp } \text{trans-fun } \\ & \quad \varepsilon^i \ c)) \wedge (\\ & \quad \forall j \leq n. i < j \longrightarrow \text{State-Idle } \text{localState } \text{output-fun } \text{trans-fun } (\text{localState } (f\text{-Exec-Comp } \\ & \quad \text{trans-fun } \varepsilon^j \ c)))) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *State-Idle-imp-exists-state-change2:*

$$\begin{aligned} & \llbracket \neg \text{State-Idle } \text{localState } \text{output-fun } \text{trans-fun } (\text{localState } c); \\ & \quad \text{State-Idle } \text{localState } \text{output-fun } \text{trans-fun } (\text{localState } (f\text{-Exec-Comp } \text{trans-fun } \varepsilon^n \\ & \quad c)) \rrbracket \implies \\ & \exists i < n. (\\ & \quad (\forall j \leq i. \neg \text{State-Idle } \text{localState } \text{output-fun } \text{trans-fun } (\text{localState } (f\text{-Exec-Comp } \\ & \quad \text{trans-fun } \varepsilon^i \ c))) \wedge \\ & \quad (\forall j \leq n. i < j \longrightarrow \text{State-Idle } \text{localState } \text{output-fun } \text{trans-fun } (\text{localState } (f\text{-Exec-Comp } \\ & \quad \text{trans-fun } \varepsilon^j \ c)))) \\ & \langle \text{proof} \rangle \end{aligned}$$

3.2.3 Basic definitions for accelerated execution

Stream processing with accelerated components

definition *f-Exec-Comp-Stream-Acc-Output* ::

nat \Rightarrow — Acceleration factor
 (*'comp* \Rightarrow *'output message-af*) \Rightarrow — Output extraction function
 (*'comp*, *'input message-af*) *Comp-Trans-Fun* \Rightarrow
'input fstream-af \Rightarrow *'comp* \Rightarrow
'output fstream-af

where *f-Exec-Comp-Stream-Acc-Output* *k output-fun trans-fun xs c* \equiv
 (*map output-fun (f-Exec-Comp-Stream trans-fun (xs \odot_f k) c)*) \div_f *k*

definition *f-Exec-Comp-Stream-Acc-LocalState* ::

nat \Rightarrow — Acceleration factor
 (*'comp* \Rightarrow *'state*) \Rightarrow — Local state extraction function
 (*'comp*, *'input message-af*) *Comp-Trans-Fun* \Rightarrow
'input fstream-af \Rightarrow *'comp* \Rightarrow
'state list

where *f-Exec-Comp-Stream-Acc-LocalState* *k localState trans-fun xs c* \equiv
 (*map localState (f-Exec-Comp-Stream trans-fun (xs \odot_f k) c)*) \div_{fl} *k*

definition *i-Exec-Comp-Stream-Acc-Output* ::

nat \Rightarrow — Acceleration factor
 (*'comp* \Rightarrow *'output message-af*) \Rightarrow — Output extraction function
 (*'comp*, *'input message-af*) *Comp-Trans-Fun* \Rightarrow
'input istream-af \Rightarrow *'comp* \Rightarrow
'output istream-af

where *i-Exec-Comp-Stream-Acc-Output* *k output-fun trans-fun input c* \equiv
 (*output-fun \circ (i-Exec-Comp-Stream trans-fun (input \odot_i k) c)*) \div_i *k*

definition *i-Exec-Comp-Stream-Acc-LocalState* ::

nat \Rightarrow — Acceleration factor
 (*'comp* \Rightarrow *'state*) \Rightarrow — Local state extraction function
 (*'comp*, *'input message-af*) *Comp-Trans-Fun* \Rightarrow
'input istream-af \Rightarrow *'comp* \Rightarrow
'state ilist

where *i-Exec-Comp-Stream-Acc-LocalState* *k localState trans-fun input c* \equiv
 (*localState \circ (i-Exec-Comp-Stream trans-fun (input \odot_i k) c)*) \div_{il} *k*

definition *f-Exec-Comp-Stream-Acc-Output-Init* ::

nat \Rightarrow — Acceleration factor
 (*'comp* \Rightarrow *'output message-af*) \Rightarrow — Output extraction function
 (*'comp*, *'input message-af*) *Comp-Trans-Fun* \Rightarrow
'input fstream-af \Rightarrow *'comp* \Rightarrow
'output fstream-af

where *f-Exec-Comp-Stream-Acc-Output-Init* *k output-fun trans-fun xs c* \equiv
 (*output-fun c*) $\#$ *f-Exec-Comp-Stream-Acc-Output* *k output-fun trans-fun xs c*

definition *f-Exec-Comp-Stream-Acc-LocalState-Init* ::

$nat \Rightarrow$ — Acceleration factor
 $('comp \Rightarrow 'state) \Rightarrow$ — Local state extraction function
 $('comp, 'input\ message\text{-}af) \textit{Comp-Trans-Fun} \Rightarrow 'input\ fstream\text{-}af \Rightarrow 'comp \Rightarrow 'state\ list$
where $f\text{-Exec-Comp-Stream-Acc-LocalState-Init } k\ localState\ trans\text{-}fun\ xs\ c \equiv (localState\ c) \# f\text{-Exec-Comp-Stream-Acc-LocalState } k\ localState\ trans\text{-}fun\ xs\ c$

definition $i\text{-Exec-Comp-Stream-Acc-Output-Init} ::$

$nat \Rightarrow$ — Acceleration factor
 $('comp \Rightarrow 'output\ message\text{-}af) \Rightarrow$ — Output extraction function
 $('comp, 'input\ message\text{-}af) \textit{Comp-Trans-Fun} \Rightarrow 'input\ istream\text{-}af \Rightarrow 'comp \Rightarrow 'output\ istream\text{-}af$
where $i\text{-Exec-Comp-Stream-Acc-Output-Init } k\ output\text{-}fun\ trans\text{-}fun\ input\ c \equiv [output\text{-}fun\ c] \frown (i\text{-Exec-Comp-Stream-Acc-Output } k\ output\text{-}fun\ trans\text{-}fun\ input\ c)$

definition $i\text{-Exec-Comp-Stream-Acc-LocalState-Init} ::$

$nat \Rightarrow$ — Acceleration factor
 $('comp \Rightarrow 'state) \Rightarrow$ — Local state extraction function
 $('comp, 'input\ message\text{-}af) \textit{Comp-Trans-Fun} \Rightarrow 'input\ istream\text{-}af \Rightarrow 'comp \Rightarrow 'state\ ilist$
where $i\text{-Exec-Comp-Stream-Acc-LocalState-Init } k\ localState\ trans\text{-}fun\ input\ c \equiv [localState\ c] \frown (i\text{-Exec-Comp-Stream-Acc-LocalState } k\ localState\ trans\text{-}fun\ input\ c)$

lemma $f\text{-Exec-Stream-Acc-Output-length}[simp]:$

$0 < k \implies$
 $length\ (f\text{-Exec-Comp-Stream-Acc-Output } k\ output\text{-}fun\ trans\text{-}fun\ xs\ c) = length\ xs$
 $\langle proof \rangle$

lemma $f\text{-Exec-Stream-Acc-LocalState-length}[simp]:$

$0 < k \implies$
 $length\ (f\text{-Exec-Comp-Stream-Acc-LocalState } k\ localState\ trans\text{-}fun\ xs\ c) = length\ xs$
 $\langle proof \rangle$

lemmas $f\text{-Exec-Stream-Acc-length} =$
 $f\text{-Exec-Stream-Acc-LocalState-length}$
 $f\text{-Exec-Stream-Acc-Output-length}$

3.2.4 Basic results for accelerated execution

lemma $f\text{-Exec-Stream-Acc-Output-Nil}[simp]:$

$f\text{-Exec-Comp-Stream-Acc-Output } k\ output\text{-}fun\ trans\text{-}fun\ []\ c = []$
 $\langle proof \rangle$

lemma $f\text{-Exec-Stream-Acc-LocalState-Nil}[simp]:$

f-Exec-Comp-Stream-Acc-LocalState k *localState* *trans-fun* \square $c = \square$
 ⟨proof⟩

lemmas *f-Exec-Stream-Acc-Nil* =
f-Exec-Stream-Acc-LocalState-Nil
f-Exec-Stream-Acc-Output-Nil

lemma *f-Exec-Stream-Acc-Output-0*[simp]:
f-Exec-Comp-Stream-Acc-Output 0 *output-fun* *trans-fun* xs $c = \square$
 ⟨proof⟩

lemma *f-Exec-Stream-Acc-LocalState-0*[simp]:
f-Exec-Comp-Stream-Acc-LocalState 0 *localState* *trans-fun* xs $c = \square$
 ⟨proof⟩

lemmas *f-Exec-Stream-Acc-0* =
f-Exec-Stream-Acc-LocalState-0
f-Exec-Stream-Acc-Output-0

lemma *f-Exec-Stream-Acc-Output-1*[simp]:
f-Exec-Comp-Stream-Acc-Output (*Suc* 0) *output-fun* *trans-fun* xs $c =$
map *output-fun* (*f-Exec-Comp-Stream* *trans-fun* xs c)
 ⟨proof⟩

lemma *f-Exec-Stream-Acc-LocalState-1*[simp]:
f-Exec-Comp-Stream-Acc-LocalState (*Suc* 0) *localState* *trans-fun* xs $c =$
map *localState* (*f-Exec-Comp-Stream* *trans-fun* xs c)
 ⟨proof⟩

lemma *i-Exec-Stream-Acc-Output-1*[simp]:
i-Exec-Comp-Stream-Acc-Output (*Suc* 0) *output-fun* *trans-fun* *input* $c =$
output-fun \circ (*i-Exec-Comp-Stream* *trans-fun* *input* c)
 ⟨proof⟩

lemma *i-Exec-Stream-Acc-LocalState-1*[simp]:
i-Exec-Comp-Stream-Acc-LocalState (*Suc* 0) *localState* *trans-fun* *input* $c =$
localState \circ (*i-Exec-Comp-Stream* *trans-fun* *input* c)
 ⟨proof⟩

lemma *f-Exec-Stream-Acc-Output-eq-last-message-hold*:
f-Exec-Comp-Stream-Acc-Output k *output-fun* *trans-fun* xs $c =$
map *output-fun* (*f-Exec-Comp-Stream* *trans-fun* ($xs \odot_f k$) c) $\mapsto_f k \div_{fl} k$
 ⟨proof⟩

lemma *i-Exec-Stream-Acc-Output-eq-last-message-hold*: $0 < k \implies$
i-Exec-Comp-Stream-Acc-Output k *output-fun* *trans-fun* *input* $c =$
output-fun \circ (*i-Exec-Comp-Stream* *trans-fun* (*input* $\odot_i k$) c) $\mapsto_i k \div_{il} k$
 ⟨proof⟩

lemma *f-Exec-Stream-Acc-Output-take*:

f-Exec-Comp-Stream-Acc-Output *k* *output-fun* *trans-fun* *xs* $c \downarrow n =$
f-Exec-Comp-Stream-Acc-Output *k* *output-fun* *trans-fun* (*xs* $\downarrow n$) *c*
 ⟨*proof*⟩

lemma *f-Exec-Stream-Acc-Output-drop*:

f-Exec-Comp-Stream-Acc-Output *k* *output-fun* *trans-fun* *xs* $c \uparrow n =$
f-Exec-Comp-Stream-Acc-Output *k* *output-fun* *trans-fun* (*xs* $\uparrow n$) (
f-Exec-Comp *trans-fun* (*xs* $\downarrow n \odot_f k$) *c*)
 ⟨*proof*⟩

lemma *i-Exec-Stream-Acc-Output-take*:

$0 < k \implies$
i-Exec-Comp-Stream-Acc-Output *k* *output-fun* *trans-fun* *input* $c \downarrow n =$
f-Exec-Comp-Stream-Acc-Output *k* *output-fun* *trans-fun* (*input* $\downarrow n$) *c*
 ⟨*proof*⟩

lemma *i-Exec-Stream-Acc-Output-drop*:

$0 < k \implies$
i-Exec-Comp-Stream-Acc-Output *k* *output-fun* *trans-fun* *input* $c \uparrow n =$
i-Exec-Comp-Stream-Acc-Output *k* *output-fun* *trans-fun* (*input* $\uparrow n$) (
f-Exec-Comp *trans-fun* (*input* $\downarrow n \odot_f k$) *c*)
 ⟨*proof*⟩

lemma *i-Exec-Stream-Acc-LocalState-take*:

$0 < k \implies$
i-Exec-Comp-Stream-Acc-LocalState *k* *localState* *trans-fun* *input* $c \downarrow n =$
f-Exec-Comp-Stream-Acc-LocalState *k* *localState* *trans-fun* (*input* $\downarrow n$) *c*
 ⟨*proof*⟩

lemma *i-Exec-Stream-Acc-LocalState-drop*:

$0 < k \implies$
i-Exec-Comp-Stream-Acc-LocalState *k* *localState* *trans-fun* *input* $c \uparrow n =$
i-Exec-Comp-Stream-Acc-LocalState *k* *localState* *trans-fun* (*input* $\uparrow n$) (
f-Exec-Comp *trans-fun* (*input* $\downarrow n \odot_f k$) *c*)
 ⟨*proof*⟩

lemma *f-Exec-Stream-Acc-Output-append*:

f-Exec-Comp-Stream-Acc-Output *k* *output-fun* *trans-fun* (*xs* $@$ *ys*) *c* =
f-Exec-Comp-Stream-Acc-Output *k* *output-fun* *trans-fun* *xs* *c* $@$
f-Exec-Comp-Stream-Acc-Output *k* *output-fun* *trans-fun* *ys* (
f-Exec-Comp *trans-fun* (*xs* $\odot_f k$) *c*)
 ⟨*proof*⟩

lemma *f-Exec-Stream-Acc-Output-Cons*:

$0 < k \implies$
f-Exec-Comp-Stream-Acc-Output *k* *output-fun* *trans-fun* (*x* $\#$ *xs*) *c* =
last-message (*map* *output-fun* (*f-Exec-Comp-Stream* *trans-fun* (*x* $\#$ $\varepsilon^k - \text{Suc } 0$)
c)) $\#$

f-Exec-Comp-Stream-Acc-Output k *output-fun* *trans-fun* xs (
f-Exec-Comp *trans-fun* ($x \# \varepsilon^k - \text{Suc } 0$) c)
 ⟨proof⟩

lemma *f-Exec-Stream-Acc-Output-one*:

$0 < k \implies$
f-Exec-Comp-Stream-Acc-Output k *output-fun* *trans-fun* $[x]$ $c =$
 $[\text{last-message } (\text{map } \text{output-fun } (\text{f-Exec-Comp-Stream } \text{trans-fun } (x \# \varepsilon^k - \text{Suc } 0)$
 $c))]$
 ⟨proof⟩

lemma *f-Exec-Stream-Acc-Output-snoc*:

$0 < k \implies$
f-Exec-Comp-Stream-Acc-Output k *output-fun* *trans-fun* $(xs @ [x])$ $c =$
f-Exec-Comp-Stream-Acc-Output k *output-fun* *trans-fun* xs $c @$
 $[\text{last-message } (\text{map } \text{output-fun } (\text{f-Exec-Comp-Stream } \text{trans-fun } (x \# \varepsilon^k - \text{Suc } 0)$
 $($
 $\text{f-Exec-Comp } \text{trans-fun } (xs \odot_f k) c)))]$
 ⟨proof⟩

lemma *i-Exec-Stream-Acc-Output-append*:

i-Exec-Comp-Stream-Acc-Output k *output-fun* *trans-fun* $(xs \frown \text{input})$ $c =$
f-Exec-Comp-Stream-Acc-Output k *output-fun* *trans-fun* xs $c \frown$
i-Exec-Comp-Stream-Acc-Output k *output-fun* *trans-fun* input (
f-Exec-Comp *trans-fun* $(xs \odot_f k)$ c)
 ⟨proof⟩

lemma *i-Exec-Stream-Acc-Output-Cons*:

$0 < k \implies$
i-Exec-Comp-Stream-Acc-Output k *output-fun* *trans-fun* $([x] \frown \text{input})$ $c =$
 $[\text{last-message } (\text{map } \text{output-fun } (\text{f-Exec-Comp-Stream } \text{trans-fun } (x \# \varepsilon^k - \text{Suc } 0)$
 $c))]$ \frown
i-Exec-Comp-Stream-Acc-Output k *output-fun* *trans-fun* input (
f-Exec-Comp *trans-fun* $(x \# \varepsilon^k - \text{Suc } 0)$ c)
 ⟨proof⟩

lemma *f-Exec-Stream-Acc-LocalState-append*:

f-Exec-Comp-Stream-Acc-LocalState k *localState* *trans-fun* $(xs @ ys)$ $c =$
f-Exec-Comp-Stream-Acc-LocalState k *localState* *trans-fun* xs $c @$
f-Exec-Comp-Stream-Acc-LocalState k *localState* *trans-fun* ys (
f-Exec-Comp *trans-fun* $(xs \odot_f k)$ c)
 ⟨proof⟩

lemma *f-Exec-Stream-Acc-LocalState-Cons*:

$0 < k \implies$
f-Exec-Comp-Stream-Acc-LocalState k *localState* *trans-fun* $(x \# xs)$ $c =$
localState (*f-Exec-Comp* *trans-fun* $(x \# \varepsilon^k - \text{Suc } 0)$ $c)$ $\#$
f-Exec-Comp-Stream-Acc-LocalState k *localState* *trans-fun* xs (
f-Exec-Comp *trans-fun* $(x \# \varepsilon^k - \text{Suc } 0)$ c)

$\langle \text{proof} \rangle$

lemma *f-Exec-Stream-Acc-LocalState-one*:

$0 < k \implies$
 $f\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun } [x] \text{ } c =$
 $[localState (f\text{-Exec-Comp trans-fun } (x \# \varepsilon^k - Suc \ 0) \ c)]$
 $\langle \text{proof} \rangle$

lemma *f-Exec-Stream-Acc-LocalState-snoc*:

$0 < k \implies$
 $f\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun } (xs \ @ \ [x]) \ c =$
 $f\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun } xs \ c \ @$
 $[localState (f\text{-Exec-Comp trans-fun } ((xs \ @ \ [x]) \ \odot_f \ k) \ c)]$
 $\langle \text{proof} \rangle$

lemma *i-Exec-Stream-Acc-LocalState-append*:

$i\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun } (xs \ \frown \ input) \ c =$
 $f\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun } xs \ c \ \frown$
 $i\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun } input \ ($
 $f\text{-Exec-Comp trans-fun } (xs \ \odot_f \ k) \ c)$
 $\langle \text{proof} \rangle$

lemma *i-Exec-Stream-Acc-LocalState-Cons*:

$0 < k \implies$
 $i\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun } ([x] \ \frown \ input) \ c =$
 $[localState (f\text{-Exec-Comp trans-fun } (x \# \varepsilon^k - Suc \ 0) \ c)] \ \frown$
 $i\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun } input \ ($
 $f\text{-Exec-Comp trans-fun } (x \# \varepsilon^k - Suc \ 0) \ c)$
 $\langle \text{proof} \rangle$

lemma *f-Exec-Stream-Acc-Output-nth*:

$\llbracket 0 < k; n < \text{length } xs \rrbracket \implies$
 $f\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun } xs \ c \ ! \ n =$
 $\text{last-message } (\text{map } \text{output-fun } ($
 $f\text{-Exec-Comp-Stream trans-fun } (xs \ ! \ n \ \# \ \varepsilon^k - Suc \ 0) \ ($
 $f\text{-Exec-Comp trans-fun } (xs \ \downarrow \ n \ \odot_f \ k) \ c))$
 $\langle \text{proof} \rangle$

lemma *f-Exec-Stream-Acc-Output-nth-eq-i-nth*:

$\llbracket 0 < k; n < n' \rrbracket \implies$
 $f\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun } (input \ \Downarrow \ n') \ c \ ! \ n =$
 $i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun } input \ c \ n$
 $\langle \text{proof} \rangle$

lemma *i-Exec-Stream-Acc-Output-nth*:

$0 < k \implies$
 $i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun } input \ c \ n =$
 $\text{last-message } (\text{map } \text{output-fun } ($
 $f\text{-Exec-Comp-Stream trans-fun } (input \ n \ \# \ \varepsilon^k - Suc \ 0) \ ($

$f\text{-Exec-Comp trans-fun (input } \Downarrow n \odot_f k) c))$
 ⟨proof⟩

corollary *i-Exec-Stream-Acc-Output-nth-f-nth:*

$0 < k \implies$
 $i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c \ n =$
 $f\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun (input } \Downarrow \text{Suc } n) \ c \ ! \ n$
 ⟨proof⟩

corollary *i-Exec-Stream-Acc-Output-nth-f-last:*

$0 < k \implies$
 $i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c \ n =$
 $\text{last (} f\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun (input } \Downarrow \text{Suc } n) \ c)$
 ⟨proof⟩

lemma *f-Exec-Stream-Acc-LocalState-nth:*

$\llbracket 0 < k; n < \text{length } xs \rrbracket \implies$
 $f\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun } xs \ c \ ! \ n =$
 $\text{localState (} f\text{-Exec-Comp trans-fun (} xs \downarrow \text{Suc } n \odot_f k) \ c)$
 ⟨proof⟩

lemma *f-Exec-Stream-Acc-LocalState-nth-eq-i-nth:*

$\llbracket 0 < k; n < n' \rrbracket \implies$
 $f\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun (input } \Downarrow n') \ c \ ! \ n =$
 $i\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun input } c \ n$
 ⟨proof⟩

corollary *i-Exec-Stream-Acc-LocalState-nth-f-nth:*

$0 < k \implies$
 $i\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ output-fun trans-fun input } c \ n =$
 $f\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ output-fun trans-fun (input } \Downarrow \text{Suc } n) \ c \ ! \ n$
 ⟨proof⟩

corollary *i-Exec-Stream-Acc-LocalState-nth-f-last:*

$0 < k \implies$
 $i\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun input } c \ n =$
 $\text{last (} f\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun (input } \Downarrow \text{Suc } n)$
 $c)$
 ⟨proof⟩

lemma *i-Exec-Stream-Acc-LocalState-nth:*

$0 < k \implies$
 $i\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun input } c \ n =$
 $\text{localState (} f\text{-Exec-Comp trans-fun (input } \Downarrow \text{Suc } n \odot_f k) \ c)$
 ⟨proof⟩

lemma *f-Exec-Stream-Acc-Output-causal:*

$xs \downarrow n = ys \downarrow n \implies$
 $f\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun } xs \ c \downarrow n =$

f-Exec-Comp-Stream-Acc-Output k *output-fun* *trans-fun* ys $c \downarrow n$
 ⟨proof⟩

lemma *i-Exec-Stream-Acc-Output-causal*:

$input1 \Downarrow n = input2 \Downarrow n \implies$
i-Exec-Comp-Stream-Acc-Output k *output-fun* *trans-fun* $input1$ $c \Downarrow n =$
i-Exec-Comp-Stream-Acc-Output k *output-fun* *trans-fun* $input2$ $c \Downarrow n$
 ⟨proof⟩

lemma *f-Exec-Stream-Acc-Output-Connected-strictly-causal*:

$\llbracket xs \downarrow n = ys \downarrow n;$
f-Streams-Connected
(f-Exec-Comp-Stream-Acc-Output k *output-fun* *trans-fun* xs $c)$
channel1;
f-Streams-Connected
(f-Exec-Comp-Stream-Acc-Output k *output-fun* *trans-fun* ys $c)$
channel2 $\rrbracket \implies$
channel1 $\downarrow Suc\ n = channel2 \downarrow Suc\ n$
 ⟨proof⟩

lemma *i-Exec-Stream-Acc-Output-Connected-strictly-causal*:

$\llbracket input1 \downarrow n = input2 \downarrow n;$
i-Streams-Connected
(i-Exec-Comp-Stream-Acc-Output k *output-fun* *trans-fun* $input1$ $c)$
channel1;
i-Streams-Connected
(i-Exec-Comp-Stream-Acc-Output k *output-fun* *trans-fun* $input2$ $c)$
channel2 $\rrbracket \implies$
channel1 $\Downarrow Suc\ n = channel2 \Downarrow Suc\ n$
 ⟨proof⟩

Complete execution cycles/steps of accelerated execution

definition *Acc-Trans-Fun-Step* ::

$nat \Rightarrow \text{--- Acceleration factor}$
 $('comp, 'input\ message\text{-}af) \text{ Comp-Trans-Fun} \Rightarrow$
 $('comp\ list \Rightarrow 'comp) \Rightarrow \text{--- Pointwise output shrink function}$
 $'input\ message\text{-}af \Rightarrow 'comp \Rightarrow$
 $'comp$

where *Acc-Trans-Fun-Step* k *trans-fun* *pointwise-shrink* x $c \equiv$
 $pointwise\text{-}shrink\ (f\text{-Exec-Comp-Stream}\ trans\text{-}fun\ (x \# \varepsilon^k - Suc\ 0)\ c)$

definition *is-Pointwise-Output-Shrink* ::

$('comp\ list \Rightarrow 'comp) \Rightarrow \text{--- Pointwise output shrink function}$
 $('comp \Rightarrow 'output\ message\text{-}af) \Rightarrow \text{--- Output extraction function for consideration}$
 $bool$

where *is-Pointwise-Output-Shrink* *pointwise-shrink* *output-fun* \equiv
 $\forall cs. output\text{-}fun\ (pointwise\text{-}shrink\ cs) = last\text{-}message\ (map\ output\text{-}fun\ cs)$

primrec *is-Pointwise-Output-Shrink-list* ::

$(\text{'comp list} \Rightarrow \text{'comp}) \Rightarrow \text{---}$ Pointwise output shrink function
 $(\text{'comp} \Rightarrow \text{'output message-af}) \text{ list} \Rightarrow \text{---}$ List of output extraction functions for consideration
bool

where

$\text{is-Pointwise-Output-Shrink-list pointwise-shrink []} = \text{True}$
 $\text{is-Pointwise-Output-Shrink-list pointwise-shrink (f \# fs)} =$
 $(\text{is-Pointwise-Output-Shrink pointwise-shrink f} \wedge$
 $\text{is-Pointwise-Output-Shrink-list pointwise-shrink fs})$

definition *is-correct-localState-Pointwise-Output-Shrink* ::

$(\text{'comp list} \Rightarrow \text{'comp}) \Rightarrow \text{---}$ Pointwise output shrink function
 $(\text{'comp} \Rightarrow \text{'state}) \Rightarrow \text{---}$ Local state extraction function
bool

where *is-correct-localState-Pointwise-Output-Shrink pointwise-shrink localState*

\equiv

$\forall \text{cs. cs} \neq [] \longrightarrow \text{localState (pointwise-shrink cs)} = \text{localState (last cs)}$

lemma *Deterministic-trans-fun-imp-acc-trans-fun*:

Deterministic-Trans-Fun trans-fun localState \implies

Deterministic-Trans-Fun (Acc-Trans-Fun-Step k trans-fun pointwise-shrink) localState
 $\langle \text{proof} \rangle$

lemma *is-Pointwise-Output-Shrink-list-imp-is-Pointwise-Output-Shrink*:

$\llbracket \text{is-Pointwise-Output-Shrink-list pointwise-shrink fs; output-fun} \in \text{set fs} \rrbracket \implies$
 $\text{is-Pointwise-Output-Shrink pointwise-shrink output-fun}$

$\langle \text{proof} \rangle$

lemma *is-Pointwise-Output-Shrink-list-eq-is-Pointwise-Output-Shrink-all*:

$(\text{is-Pointwise-Output-Shrink-list pointwise-shrink fs}) =$
 $(\forall \text{output-fun} \in \text{set fs. is-Pointwise-Output-Shrink pointwise-shrink output-fun})$

$\langle \text{proof} \rangle$

lemma *is-Pointwise-Output-Shrink-subset*:

$\llbracket \text{is-Pointwise-Output-Shrink-list pointwise-shrink fs; set fs}' \subseteq \text{set fs} \rrbracket \implies$
 $\text{is-Pointwise-Output-Shrink-list pointwise-shrink fs}'$

$\langle \text{proof} \rangle$

lemma *f-Exec-Stream-Acc-LocalState-eq-Acc-Trans-Fun-Step-LocalState*: $\bigwedge c.$

$\llbracket 0 < k;$

Deterministic-Trans-Fun trans-fun localState;

$\text{is-correct-localState-Pointwise-Output-Shrink pointwise-shrink localState} \rrbracket \implies$

f-Exec-Comp-Stream-Acc-LocalState k localState trans-fun xs c =

map localState (f-Exec-Comp-Stream (Acc-Trans-Fun-Step k trans-fun pointwise-shrink) xs c)

$\langle \text{proof} \rangle$

lemma *f-Exec-Stream-Acc-Output-eq-Acc-Trans-Fun-Step-Output*: $\bigwedge c.$

$\llbracket 0 < k;$
Deterministic-Trans-Fun *trans-fun localState*;
is-correct-localState-Pointwise-Output-Shrink *pointwise-shrink localState*;
is-Pointwise-Output-Shrink *pointwise-shrink output-fun* $\rrbracket \implies$
f-Exec-Comp-Stream-Acc-Output *k output-fun trans-fun xs c* =
map output-fun (f-Exec-Comp-Stream (Acc-Trans-Fun-Step k trans-fun point-
wise-shrink) xs c)
 ⟨*proof*⟩

lemma *i-Exec-Stream-Acc-LocalState-eq-Acc-Trans-Fun-Step-LocalState*: $\wedge c.$
 $\llbracket 0 < k;$
Deterministic-Trans-Fun *trans-fun localState*;
is-correct-localState-Pointwise-Output-Shrink *pointwise-shrink localState* $\rrbracket \implies$
i-Exec-Comp-Stream-Acc-LocalState *k localState trans-fun input c* =
localState \circ (*i-Exec-Comp-Stream (Acc-Trans-Fun-Step k trans-fun pointwise-shrink)*
input c)
 ⟨*proof*⟩

lemma *i-Exec-Stream-Acc-Output-eq-Acc-Trans-Fun-Step-Output*: $\wedge c.$
 $\llbracket 0 < k;$
Deterministic-Trans-Fun *trans-fun localState*;
is-correct-localState-Pointwise-Output-Shrink *pointwise-shrink localState*;
is-Pointwise-Output-Shrink *pointwise-shrink output-fun* $\rrbracket \implies$
i-Exec-Comp-Stream-Acc-Output *k output-fun trans-fun input c* =
output-fun \circ (*i-Exec-Comp-Stream (Acc-Trans-Fun-Step k trans-fun pointwise-shrink)*
input c)
 ⟨*proof*⟩

3.2.5 Basic results for accelerated execution with initial state in the resulting stream

lemma *f-Exec-Stream-Acc-Output-Init-length*:
 $0 < k \implies$
length (f-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun xs c) = *Suc*
(length xs)
 ⟨*proof*⟩

lemma *f-Exec-Stream-Acc-LocalState-Init-length*:
 $0 < k \implies$
length (f-Exec-Comp-Stream-Acc-LocalState-Init k localState trans-fun xs c) = *Suc*
(length xs)
 ⟨*proof*⟩

lemma *f-Exec-Stream-Acc-Output-Init-Nil*:
f-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun [] c = [*output-fun c*]
 ⟨*proof*⟩

lemma *f-Exec-Stream-Acc-LocalState-Init-Nil*:
f-Exec-Comp-Stream-Acc-LocalState-Init k localState trans-fun [] c = [*localState*

c]
 ⟨proof⟩

lemma *f-Exec-Stream-Acc-Output-Init-1*:
 $f\text{-Exec-Comp-Stream-Acc-Output-Init } (Suc\ 0)\ output\text{-fun } trans\text{-fun } xs\ c =$
 $map\ output\text{-fun } (f\text{-Exec-Comp-Stream-Init } trans\text{-fun } xs\ c)$
 ⟨proof⟩

lemma *f-Exec-Stream-Acc-LocalState-Init-1*:
 $f\text{-Exec-Comp-Stream-Acc-LocalState-Init } (Suc\ 0)\ localState\ trans\text{-fun } xs\ c =$
 $map\ localState\ (f\text{-Exec-Comp-Stream-Init } trans\text{-fun } xs\ c)$
 ⟨proof⟩

lemma *i-Exec-Stream-Acc-Output-Init-1*:
 $i\text{-Exec-Comp-Stream-Acc-Output-Init } (Suc\ 0)\ output\text{-fun } trans\text{-fun } input\ c =$
 $output\text{-fun } \circ (i\text{-Exec-Comp-Stream-Init } trans\text{-fun } input\ c)$
 ⟨proof⟩

lemma *i-Exec-Stream-Acc-LocalState-Init-1*:
 $i\text{-Exec-Comp-Stream-Acc-LocalState-Init } (Suc\ 0)\ localState\ trans\text{-fun } input\ c =$
 $localState \circ (i\text{-Exec-Comp-Stream-Init } trans\text{-fun } input\ c)$
 ⟨proof⟩

lemma *f-Exec-Stream-Acc-Output-Init-take*:
 $f\text{-Exec-Comp-Stream-Acc-Output-Init } k\ output\text{-fun } trans\text{-fun } xs\ c \downarrow (Suc\ n) =$
 $f\text{-Exec-Comp-Stream-Acc-Output-Init } k\ output\text{-fun } trans\text{-fun } (xs \downarrow n)\ c$
 ⟨proof⟩

lemma *f-Exec-Stream-Acc-Output-Init-drop'*:
 $\llbracket 0 < k; n < length\ xs \rrbracket \implies$
 $f\text{-Exec-Comp-Stream-Acc-Output-Init } k\ output\text{-fun } trans\text{-fun } xs\ c \uparrow Suc\ n =$
 $f\text{-Exec-Comp-Stream-Acc-Output } k\ output\text{-fun } trans\text{-fun } xs\ c \uparrow n$
 ⟨proof⟩

lemma *i-Exec-Stream-Acc-Output-Init-take*:
 $0 < k \implies$
 $i\text{-Exec-Comp-Stream-Acc-Output-Init } k\ output\text{-fun } trans\text{-fun } input\ c \downarrow (Suc\ n) =$
 $f\text{-Exec-Comp-Stream-Acc-Output-Init } k\ output\text{-fun } trans\text{-fun } (input \downarrow n)\ c$
 ⟨proof⟩

lemma *i-Exec-Stream-Acc-Output-Init-drop'*:
 $0 < k \implies$
 $i\text{-Exec-Comp-Stream-Acc-Output-Init } k\ output\text{-fun } trans\text{-fun } xs\ c \uparrow Suc\ n =$
 $i\text{-Exec-Comp-Stream-Acc-Output } k\ output\text{-fun } trans\text{-fun } xs\ c \uparrow n$
 ⟨proof⟩

lemma *f-Exec-Stream-Acc-Output-Init-strictly-causal*:
 $xs \downarrow n = ys \downarrow n \implies$

$f\text{-Exec-Comp-Stream-Acc-Output-Init } k \text{ output-fun trans-fun } xs \ c \ \downarrow \ Suc \ n =$
 $f\text{-Exec-Comp-Stream-Acc-Output-Init } k \text{ output-fun trans-fun } ys \ c \ \downarrow \ Suc \ n$
 ⟨proof⟩

lemma *i-Exec-Stream-Acc-Output-Init-strictly-causal:*

$input1 \ \downarrow \ n = input2 \ \downarrow \ n \implies$
 $i\text{-Exec-Comp-Stream-Acc-Output-Init } k \text{ output-fun trans-fun } input1 \ c \ \downarrow \ Suc \ n =$
 $i\text{-Exec-Comp-Stream-Acc-Output-Init } k \text{ output-fun trans-fun } input2 \ c \ \downarrow \ Suc \ n$
 ⟨proof⟩

lemma *f-Exec-Stream-Acc-Output-Init-eq-f-Exec-Stream-Acc-Output-Cons:*

$f\text{-Exec-Comp-Stream-Acc-Output-Init } k \text{ output-fun trans-fun } xs \ c =$
 $output\text{-fun } c \ \# \ f\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun } xs \ c$
 ⟨proof⟩

lemma *f-Exec-Stream-Acc-Output-Init-eq-f-Exec-Stream-Acc-Output-Cons-output:*

$output\text{-fun } c = \varepsilon \implies$
 $f\text{-Exec-Comp-Stream-Acc-Output-Init } k \text{ output-fun trans-fun } xs \ c =$
 $\varepsilon \ \# \ f\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun } xs \ c$
 ⟨proof⟩

lemma *f-Exec-Stream--Acc-OutputInit-tl-eq-f-Exec-Stream-Acc-Output:*

$tl \ (f\text{-Exec-Comp-Stream-Acc-Output-Init } k \text{ output-fun trans-fun } xs \ c) =$
 $f\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun } xs \ c$
 ⟨proof⟩

lemma *f-Exec-Stream-Previous-f-Exec-Stream-Acc-Output-Init:*

$f\text{-Exec-Comp-Stream-Acc-Output-Init } k \text{ output-fun trans-fun } xs \ c \ ! \ n =$
 $(f\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun } xs \ c)^{\leftarrow n} \text{ output-fun } c \ n$
 ⟨proof⟩

lemma *f-Exec-Stream-Acc-Output-Init-eq-output-channel:*

$\llbracket \text{ output-fun } c = \varepsilon;$
 $f\text{-Streams-Connected}$
 $(f\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun } xs \ c)$
 $\text{channel} \rrbracket \implies$
 $f\text{-Exec-Comp-Stream-Acc-Output-Init } k \text{ output-fun trans-fun } xs \ c = \text{channel}$
 ⟨proof⟩

lemma *i-Exec-Stream-Acc-Output-Init-eq-i-Exec-Stream-Acc-Output-Cons:*

$i\text{-Exec-Comp-Stream-Acc-Output-Init } k \text{ output-fun trans-fun } input \ c =$
 $[\text{output-fun } c] \ \frown \ i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun } input \ c$
 ⟨proof⟩

lemma *i-Exec-Stream-Acc-Output-Init-eq-i-Exec-Stream-Acc-Output-Cons-output:*

$output\text{-fun } c = \varepsilon \implies$

i-Exec-Comp-Stream-Acc-Output-Init k *output-fun* *trans-fun* *input* $c =$
 $[\varepsilon] \frown i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c$
 $\langle \text{proof} \rangle$

lemma *i-Exec-Stream-Previous-i-Exec-Stream-Acc-Output-Init*:

i-Exec-Comp-Stream-Acc-Output-Init k *output-fun* *trans-fun* *input* c $n =$
 $(i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c)^{\leftarrow \text{output-fun } c} n$
 $\langle \text{proof} \rangle$

lemma *i-Exec-Stream-Acc-Output-Init-eq-output-channel*:

$\llbracket \text{output-fun } c = \varepsilon;$
i-Streams-Connected
 $(i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c)$
 $\text{channel} \rrbracket \implies$
i-Exec-Comp-Stream-Acc-Output-Init k *output-fun* *trans-fun* *input* $c = \text{channel}$
 $\langle \text{proof} \rangle$

3.2.6 Rules for proving execution equivalence

A required precondition is that the *equiv-states* relation, which indicates whether the local states of $c1$ and $c2$ are equivalent with respect to observable behaviour, is preserved also after executing an input stream, because the *equiv-states* relation should deliver valid results not only at the time point $0::'a$ but at every time point.

lemma *f-Equiv-Exec-Stream-expand-shrink-equiv-state-set*[*rule-format*]:

$\bigwedge c1 \ c2 \ i. \llbracket$
 $0 < k1; 0 < k2;$
 $\text{equiv-states } (\text{localState1 } c1) (\text{localState2 } c2);$
 $\forall \text{input0. set input0} \subseteq A \longrightarrow (\forall m \in A.$
Equiv-Exec $m \text{ equiv-states}$
 $\text{localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2}$
 $\text{trans-fun1 trans-fun2 } k1 \ k2$
 $(f\text{-Exec-Comp trans-fun1 } (\text{map input-fun1 input0} \odot_f k1) \ c1)$
 $(f\text{-Exec-Comp trans-fun2 } (\text{map input-fun2 input0} \odot_f k2) \ c2));$
— *equiv-states* relation implies equivalent executions
— not only at the beginning but also after processing an input
 $\text{set input} \subseteq A; i < \text{length input} \rrbracket \implies$
equiv-states
 $(\text{localState1 } ((f\text{-Exec-Comp-Stream trans-fun1 } (\text{map input-fun1 input} \odot_f k1)$
 $c1) \div_f k1 \ ! \ i))$
 $(\text{localState2 } ((f\text{-Exec-Comp-Stream trans-fun2 } (\text{map input-fun2 input} \odot_f k2)$
 $c2) \div_f k2 \ ! \ i))$
 $\langle \text{proof} \rangle$

corollary *f-Equiv-Exec-Stream-expand-shrink-equiv-state*:

$\llbracket 0 < k1; 0 < k2;$
 $\text{equiv-states } (\text{localState1 } c1) (\text{localState2 } c2);$
 $\bigwedge \text{input0 } m. \text{Equiv-Exec } m$

$equiv\text{-}states\ localState1\ localState2\ input\text{-}fun1\ input\text{-}fun2\ output\text{-}fun1\ output\text{-}fun2$
 $trans\text{-}fun1\ trans\text{-}fun2\ k1\ k2$
 $(f\text{-}Exec\text{-}Comp\ trans\text{-}fun1\ (map\ input\text{-}fun1\ input0\ \odot_f\ k1)\ c1)$
 $(f\text{-}Exec\text{-}Comp\ trans\text{-}fun2\ (map\ input\text{-}fun2\ input0\ \odot_f\ k2)\ c2);$
 $i < length\ input\] \implies$
 $equiv\text{-}states$
 $(localState1\ ((f\text{-}Exec\text{-}Comp\text{-}Stream\ trans\text{-}fun1\ (map\ input\text{-}fun1\ input\ \odot_f\ k1)$
 $c1) \div_{\#} k1\ !\ i))$
 $(localState2\ ((f\text{-}Exec\text{-}Comp\text{-}Stream\ trans\text{-}fun2\ (map\ input\text{-}fun2\ input\ \odot_f\ k2)$
 $c2) \div_{\#} k2\ !\ i))$
 $\langle proof \rangle$

lemma $f\text{-}Equiv\text{-}Exec\text{-}expand\text{-}shrink\text{-}equiv\text{-}state\text{-}set$:

$\llbracket 0 < k1; 0 < k2; equiv\text{-}states\ (localState1\ c1)\ (localState2\ c2);$
 $\bigwedge input0\ m. \llbracket set\ input0 \subseteq A; m \in A \rrbracket \implies$
 $Equiv\text{-}Exec$
 $m\ equiv\text{-}states\ localState1\ localState2$
 $input\text{-}fun1\ input\text{-}fun2\ output\text{-}fun1\ output\text{-}fun2\ trans\text{-}fun1\ trans\text{-}fun2\ k1\ k2$
 $(f\text{-}Exec\text{-}Comp\ trans\text{-}fun1\ (map\ input\text{-}fun1\ input0\ \odot_f\ k1)\ c1)$
 $(f\text{-}Exec\text{-}Comp\ trans\text{-}fun2\ (map\ input\text{-}fun2\ input0\ \odot_f\ k2)\ c2);$
 $set\ input \subseteq A\] \implies$
 $equiv\text{-}states$
 $(localState1\ (f\text{-}Exec\text{-}Comp\ trans\text{-}fun1\ (map\ input\text{-}fun1\ input\ \odot_f\ k1)\ c1))$
 $(localState2\ (f\text{-}Exec\text{-}Comp\ trans\text{-}fun2\ (map\ input\text{-}fun2\ input\ \odot_f\ k2)\ c2))$
 $\langle proof \rangle$

lemma $f\text{-}Equiv\text{-}Exec\text{-}expand\text{-}shrink\text{-}equiv\text{-}state$:

$\llbracket 0 < k1; 0 < k2; equiv\text{-}states\ (localState1\ c1)\ (localState2\ c2);$
 $\bigwedge input0\ m.$
 $Equiv\text{-}Exec$
 $m\ equiv\text{-}states\ localState1\ localState2$
 $input\text{-}fun1\ input\text{-}fun2\ output\text{-}fun1\ output\text{-}fun2\ trans\text{-}fun1\ trans\text{-}fun2\ k1\ k2$
 $(f\text{-}Exec\text{-}Comp\ trans\text{-}fun1\ (map\ input\text{-}fun1\ input0\ \odot_f\ k1)\ c1)$
 $(f\text{-}Exec\text{-}Comp\ trans\text{-}fun2\ (map\ input\text{-}fun2\ input0\ \odot_f\ k2)\ c2)\] \implies$
 $equiv\text{-}states$
 $(localState1\ (f\text{-}Exec\text{-}Comp\ trans\text{-}fun1\ (map\ input\text{-}fun1\ input\ \odot_f\ k1)\ c1))$
 $(localState2\ (f\text{-}Exec\text{-}Comp\ trans\text{-}fun2\ (map\ input\text{-}fun2\ input\ \odot_f\ k2)\ c2))$
 $\langle proof \rangle$

lemma $i\text{-}Equiv\text{-}Exec\text{-}Stream\text{-}expand\text{-}shrink\text{-}equiv\text{-}state\text{-}set[rule\text{-}format]$:

$\llbracket 0 < k1; 0 < k2; equiv\text{-}states\ (localState1\ c1)\ (localState2\ c2);$
 $\bigwedge input0\ m. \llbracket set\ input0 \subseteq A; m \in A \rrbracket \implies$
 $Equiv\text{-}Exec$
 $m\ equiv\text{-}states\ localState1\ localState2$
 $input\text{-}fun1\ input\text{-}fun2\ output\text{-}fun1\ output\text{-}fun2\ trans\text{-}fun1\ trans\text{-}fun2\ k1\ k2$
 $(f\text{-}Exec\text{-}Comp\ trans\text{-}fun1\ (map\ input\text{-}fun1\ input0\ \odot_f\ k1)\ c1)$
 $(f\text{-}Exec\text{-}Comp\ trans\text{-}fun2\ (map\ input\text{-}fun2\ input0\ \odot_f\ k2)\ c2);$
 $range\ input \subseteq A\] \implies$

equiv-states
 (localState1 ((i-Exec-Comp-Stream trans-fun1 ((input-fun1 \circ input) \odot_i k1) c1
 \div_{il} k1) i))
 (localState2 ((i-Exec-Comp-Stream trans-fun2 ((input-fun2 \circ input) \odot_i k2) c2
 \div_{il} k2) i))
 <proof>

lemma *i-Equiv-Exec-Stream-expand-shrink-equiv-state:*

$\llbracket 0 < k1; 0 < k2; \text{equiv-states (localState1 c1) (localState2 c2);$
 $\wedge \text{input0 m.}$
Equiv-Exec
 $m \text{ equiv-states localState1 localState2}$
 $\text{input-fun1 input-fun2 output-fun1 output-fun2 trans-fun1 trans-fun2 k1 k2}$
 $(f\text{-Exec-Comp trans-fun1 (map input-fun1 input0 } \odot_f \text{ k1) c1)}$
 $(f\text{-Exec-Comp trans-fun2 (map input-fun2 input0 } \odot_f \text{ k2) c2}) \rrbracket \implies$
equiv-states
 (localState1 ((i-Exec-Comp-Stream trans-fun1 ((input-fun1 \circ input) \odot_i k1) c1
 \div_{il} k1) i))
 (localState2 ((i-Exec-Comp-Stream trans-fun2 ((input-fun2 \circ input) \odot_i k2) c2
 \div_{il} k2) i))
 <proof>

lemma *f-Equiv-Exec-Stream-expand-shrink-output-set-eq:*

$\llbracket 0 < k1; 0 < k2;$
 $\text{equiv-states (localState1 c1) (localState2 c2);}$
 $\wedge \text{input0 m. } \llbracket \text{set input0 } \subseteq A; m \in A \rrbracket \implies$
Equiv-Exec
 $m \text{ equiv-states localState1 localState2}$
 $\text{input-fun1 input-fun2 output-fun1 output-fun2 trans-fun1 trans-fun2 k1 k2}$
 $(f\text{-Exec-Comp trans-fun1 (map input-fun1 input0 } \odot_f \text{ k1) c1)}$
 $(f\text{-Exec-Comp trans-fun2 (map input-fun2 input0 } \odot_f \text{ k2) c2});$
 $\text{set input } \subseteq A \rrbracket \implies$
 $(\text{map output-fun1 (}$
 $f\text{-Exec-Comp-Stream trans-fun1 (map input-fun1 input } \odot_f \text{ k1) c1})) \div_f \text{ k1} =$
 $(\text{map output-fun2 (}$
 $f\text{-Exec-Comp-Stream trans-fun2 (map input-fun2 input } \odot_f \text{ k2) c2})) \div_f \text{ k2}$
 <proof>

lemma *f-Equiv-Exec-Stream-expand-shrink-output-eq:*

$\llbracket 0 < k1; 0 < k2;$
 $\text{equiv-states (localState1 c1) (localState2 c2);}$
 $\wedge \text{input0 m.}$
Equiv-Exec
 $m \text{ equiv-states localState1 localState2}$
 $\text{input-fun1 input-fun2 output-fun1 output-fun2}$
 $\text{trans-fun1 trans-fun2 k1 k2}$
 $(f\text{-Exec-Comp trans-fun1 (map input-fun1 input0 } \odot_f \text{ k1) c1)}$
 $(f\text{-Exec-Comp trans-fun2 (map input-fun2 input0 } \odot_f \text{ k2) c2}) \rrbracket \implies$
 $(\text{map output-fun1 (}$

$f\text{-Exec-Comp-Stream trans-fun1 (map input-fun1 input } \odot_f k1) c1) \div_f k1 =$
 $(\text{map output-fun2 (}$
 $f\text{-Exec-Comp-Stream trans-fun2 (map input-fun2 input } \odot_f k2) c2) \div_f k2$
 \langle proof \rangle

lemma *i-Equiv-Exec-Stream-expand-shrink-output-set-eq:*

$\llbracket 0 < k1; 0 < k2;$
 $\text{equiv-states (localState1 c1) (localState2 c2);}$
 $\wedge \text{input0 m. } \llbracket \text{set input0 } \subseteq A; m \in A \rrbracket \implies$
 Equiv-Exec
 $m \text{ equiv-states localState1 localState2}$
 $\text{input-fun1 input-fun2 output-fun1 output-fun2}$
 $\text{trans-fun1 trans-fun2 k1 k2}$
 $(f\text{-Exec-Comp trans-fun1 (map input-fun1 input0 } \odot_f k1) c1)$
 $(f\text{-Exec-Comp trans-fun2 (map input-fun2 input0 } \odot_f k2) c2);$
 $\text{range input } \subseteq A \rrbracket \implies$
 $(\text{output-fun1 } \circ$
 $i\text{-Exec-Comp-Stream trans-fun1 ((input-fun1 } \circ \text{input) } \odot_i k1) c1) \div_i k1 =$
 $(\text{output-fun2 } \circ$
 $i\text{-Exec-Comp-Stream trans-fun2 ((input-fun2 } \circ \text{input) } \odot_i k2) c2) \div_i k2$
 \langle proof \rangle

lemma *i-Equiv-Exec-Stream-expand-shrink-output-eq:*

$\llbracket 0 < k1; 0 < k2;$
 $\text{equiv-states (localState1 c1) (localState2 c2);}$
 $\wedge \text{input0 m.}$
 Equiv-Exec
 $m \text{ equiv-states localState1 localState2}$
 $\text{input-fun1 input-fun2 output-fun1 output-fun2 trans-fun1 trans-fun2 k1 k2}$
 $(f\text{-Exec-Comp trans-fun1 (map input-fun1 input0 } \odot_f k1) c1)$
 $(f\text{-Exec-Comp trans-fun2 (map input-fun2 input0 } \odot_f k2) c2) \rrbracket \implies$
 $(\text{output-fun1 } \circ$
 $i\text{-Exec-Comp-Stream trans-fun1 ((input-fun1 } \circ \text{input) } \odot_i k1) c1) \div_i k1 =$
 $(\text{output-fun2 } \circ$
 $i\text{-Exec-Comp-Stream trans-fun2 ((input-fun2 } \circ \text{input) } \odot_i k2) c2) \div_i k2$
 \langle proof \rangle

lemma *f-Equiv-Exec-Stream-Acc-LocalState-set:*

$\llbracket 0 < k1; 0 < k2;$
 $\text{equiv-states (localState1 c1) (localState2 c2);}$
 $\text{Equiv-Exec-stable-set A}$
 $\text{equiv-states localState1 localState2}$
 $\text{input-fun1 input-fun2 output-fun1 output-fun2}$
 $\text{trans-fun1 trans-fun2 k1 k2 c1 c2;}$
 — *equiv-states* relation implies equivalent executions
 — not only at the beginning but also after processing an input
 $\text{set input } \subseteq A;$
 $i < \text{length input} \rrbracket \implies$

equiv-states
 (*f-Exec-Comp-Stream-Acc-LocalState* $k1$ *localState1* *trans-fun1* (*map input-fun1*
input) $c1 ! i$)
 (*f-Exec-Comp-Stream-Acc-LocalState* $k2$ *localState2* *trans-fun2* (*map input-fun2*
input) $c2 ! i$)
 ⟨*proof*⟩

lemma *f-Equiv-Exec-Stream-Acc-LocalState*:

[$0 < k1; 0 < k2;$
equiv-states (*localState1* $c1$) (*localState2* $c2$);
Equiv-Exec-stable
equiv-states *localState1* *localState2*
input-fun1 *input-fun2* *output-fun1* *output-fun2*
trans-fun1 *trans-fun2* $k1$ $k2$ $c1$ $c2$;
 — *equiv-states* relation implies equivalent executions
 — not only at the beginning but also after processing an input
 $i < \text{length } \textit{input}$] \implies
equiv-states
 (*f-Exec-Comp-Stream-Acc-LocalState* $k1$ *localState1* *trans-fun1* (*map input-fun1*
input) $c1 ! i$)
 (*f-Exec-Comp-Stream-Acc-LocalState* $k2$ *localState2* *trans-fun2* (*map input-fun2*
input) $c2 ! i$)
 ⟨*proof*⟩

lemma *f-Equiv-Exec-Stream-Acc-Output-set-eq*:

[$0 < k1; 0 < k2;$
equiv-states (*localState1* $c1$) (*localState2* $c2$);
Equiv-Exec-stable-set A
equiv-states *localState1* *localState2*
input-fun1 *input-fun2* *output-fun1* *output-fun2* *trans-fun1* *trans-fun2* $k1$ $k2$ $c1$
 $c2$;
 $\textit{set } \textit{input} \subseteq A$] \implies
f-Exec-Comp-Stream-Acc-Output $k1$ *output-fun1* *trans-fun1* (*map input-fun1* *in-*
put) $c1 =$
f-Exec-Comp-Stream-Acc-Output $k2$ *output-fun2* *trans-fun2* (*map input-fun2* *in-*
put) $c2$
 ⟨*proof*⟩

lemma *f-Equiv-Exec-Stream-Acc-Output-eq*:

[$0 < k1; 0 < k2;$
equiv-states (*localState1* $c1$) (*localState2* $c2$);
Equiv-Exec-stable
equiv-states *localState1* *localState2*
input-fun1 *input-fun2* *output-fun1* *output-fun2* *trans-fun1* *trans-fun2* $k1$ $k2$ $c1$
 $c2$] \implies
f-Exec-Comp-Stream-Acc-Output $k1$ *output-fun1* *trans-fun1* (*map input-fun1* *in-*
put) $c1 =$
f-Exec-Comp-Stream-Acc-Output $k2$ *output-fun2* *trans-fun2* (*map input-fun2* *in-*
put) $c2$

<proof>

lemma *i-Equiv-Exec-Stream-Acc-LocalState-set:*

$\llbracket 0 < k1; 0 < k2;$
 $\text{equiv-states } (\text{localState1 } c1) (\text{localState2 } c2);$
 $\text{Equiv-Exec-stable-set } A$
 $\text{equiv-states } \text{localState1 } \text{localState2 } \text{input-fun1 } \text{input-fun2 } \text{output-fun1 } \text{output-fun2}$
 $\text{trans-fun1 } \text{trans-fun2 } k1 k2 c1 c2;$
 $\text{range input } \subseteq A \rrbracket \implies$
 equiv-states
 $(\text{i-Exec-Comp-Stream-Acc-LocalState } k1 \text{ localState1 } \text{trans-fun1 } (\text{input-fun1 } \circ \text{input}) c1 i)$
 $(\text{i-Exec-Comp-Stream-Acc-LocalState } k2 \text{ localState2 } \text{trans-fun2 } (\text{input-fun2 } \circ \text{input}) c2 i)$
<proof>

lemma *i-Equiv-Exec-Stream-Acc-LocalState:*

$\llbracket 0 < k1; 0 < k2;$
 $\text{equiv-states } (\text{localState1 } c1) (\text{localState2 } c2);$
 Equiv-Exec-stable
 $\text{equiv-states } \text{localState1 } \text{localState2}$
 $\text{input-fun1 } \text{input-fun2 } \text{output-fun1 } \text{output-fun2}$
 $\text{trans-fun1 } \text{trans-fun2 } k1 k2 c1 c2 \rrbracket \implies$
 equiv-states
 $(\text{i-Exec-Comp-Stream-Acc-LocalState } k1 \text{ localState1 } \text{trans-fun1 } (\text{input-fun1 } \circ \text{input}) c1 i)$
 $(\text{i-Exec-Comp-Stream-Acc-LocalState } k2 \text{ localState2 } \text{trans-fun2 } (\text{input-fun2 } \circ \text{input}) c2 i)$
<proof>

lemma *i-Equiv-Exec-Stream-Acc-Output-set-eq:*

$\llbracket 0 < k1; 0 < k2;$
 $\text{equiv-states } (\text{localState1 } c1) (\text{localState2 } c2);$
 $\text{Equiv-Exec-stable-set } A$
 $\text{equiv-states } \text{localState1 } \text{localState2}$
 $\text{input-fun1 } \text{input-fun2 } \text{output-fun1 } \text{output-fun2 } \text{trans-fun1 } \text{trans-fun2 } k1 k2 c1$
 $c2;$
 $\text{range input } \subseteq A \rrbracket \implies$
 $\text{i-Exec-Comp-Stream-Acc-Output } k1 \text{ output-fun1 } \text{trans-fun1 } (\text{input-fun1 } \circ \text{input})$
 $c1 =$
 $\text{i-Exec-Comp-Stream-Acc-Output } k2 \text{ output-fun2 } \text{trans-fun2 } (\text{input-fun2 } \circ \text{input})$
 $c2$
<proof>

lemma *i-Equiv-Exec-Stream-Acc-Output-eq:*

$\llbracket 0 < k1; 0 < k2;$
 $\text{equiv-states } (\text{localState1 } c1) (\text{localState2 } c2);$

Equiv-Exec-stable
equiv-states localState1 localState2
input-fun1 input-fun2 output-fun1 output-fun2 trans-fun1 trans-fun2 k1 k2 c1
c2 $\mathbb{J} \implies$
i-Exec-Comp-Stream-Acc-Output k1 output-fun1 trans-fun1 (input-fun1 \circ input)
c1 =
i-Exec-Comp-Stream-Acc-Output k2 output-fun2 trans-fun2 (input-fun2 \circ input)
c2
 ⟨proof⟩

3.2.7 Idle states and accelerated execution

lemma *f-Exec-Stream-Acc-LocalState--State-Idle-nth*[rule-format]:

$\bigwedge c i.$
 $\mathbb{J} 0 < l; l \leq k; \text{Exec-Equal-State } localState \text{ trans-fun};$
 $\forall n \leq i. \text{State-Idle } localState \text{ output-fun trans-fun (}$
 $\quad f\text{-Exec-Comp-Stream-Acc-LocalState } l \text{ localState trans-fun } xs \text{ c ! } n);$
 $i < \text{length } xs \mathbb{J} \implies$
 $f\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun } xs \text{ c ! } i =$
 $f\text{-Exec-Comp-Stream-Acc-LocalState } l \text{ localState trans-fun } xs \text{ c ! } i$
 ⟨proof⟩

corollary *f-Exec-Stream-Acc-LocalState--State-Idle-eq*[rule-format]:

$\mathbb{J} 0 < l; l \leq k; \text{Exec-Equal-State } localState \text{ trans-fun};$
 $\forall n < \text{length } xs. \text{State-Idle } localState \text{ output-fun trans-fun (}$
 $\quad f\text{-Exec-Comp-Stream-Acc-LocalState } l \text{ localState trans-fun } xs \text{ c ! } n) \mathbb{J} \implies$
 $f\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun } xs \text{ c} =$
 $f\text{-Exec-Comp-Stream-Acc-LocalState } l \text{ localState trans-fun } xs \text{ c}$
 ⟨proof⟩

lemma *i-Exec-Stream-Acc-LocalState--State-Idle-nth*[rule-format]:

$\mathbb{J} 0 < l; l \leq k; \text{Exec-Equal-State } localState \text{ trans-fun};$
 $\forall n \leq i. \text{State-Idle } localState \text{ output-fun trans-fun (}$
 $\quad i\text{-Exec-Comp-Stream-Acc-LocalState } l \text{ localState trans-fun input } c \text{ n) } \mathbb{J} \implies$
 $i\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun input } c \text{ } i =$
 $i\text{-Exec-Comp-Stream-Acc-LocalState } l \text{ localState trans-fun input } c \text{ } i$
 ⟨proof⟩

corollary *i-Exec-Stream-Acc-LocalState--State-Idle-eq*[rule-format]:

$\mathbb{J} 0 < l; l \leq k; \text{Exec-Equal-State } localState \text{ trans-fun};$
 $\forall n. \text{State-Idle } localState \text{ output-fun trans-fun (}$
 $\quad i\text{-Exec-Comp-Stream-Acc-LocalState } l \text{ localState trans-fun input } c \text{ n) } \mathbb{J} \implies$
 $i\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun input } c =$
 $i\text{-Exec-Comp-Stream-Acc-LocalState } l \text{ localState trans-fun input } c$
 ⟨proof⟩

lemma *f-Exec-Stream-Acc-Output--State-Idle-nth*[rule-format]:

$\mathbb{J} 0 < l; l \leq k; \text{Exec-Equal-State } localState \text{ trans-fun};$
 $\forall n \leq i. \text{State-Idle } localState \text{ output-fun trans-fun (}$

$f\text{-Exec-Comp-Stream-Acc-LocalState } l \text{ localState trans-fun } xs \ c \ ! \ n);$
 $i < \text{length } xs \] \implies$
 $f\text{-Exec-Comp-Stream-Acc-Output } k \ \text{output-fun trans-fun } xs \ c \ ! \ i =$
 $f\text{-Exec-Comp-Stream-Acc-Output } l \ \text{output-fun trans-fun } xs \ c \ ! \ i$
 <proof>

lemma $f\text{-Exec-Stream-Acc-Output--State-Idle-eq}$ [rule-format]:
 $\llbracket 0 < l; l \leq k; \text{Exec-Equal-State localState trans-fun};$
 $\forall n < \text{length } xs. \text{State-Idle localState output-fun trans-fun } ($
 $f\text{-Exec-Comp-Stream-Acc-LocalState } l \ \text{localState trans-fun } xs \ c \ ! \ n) \rrbracket \implies$
 $f\text{-Exec-Comp-Stream-Acc-Output } k \ \text{output-fun trans-fun } xs \ c =$
 $f\text{-Exec-Comp-Stream-Acc-Output } l \ \text{output-fun trans-fun } xs \ c$
 <proof>

lemma $i\text{-Exec-Stream-Acc-Output--State-Idle-nth}$ [rule-format]:
 $\llbracket 0 < l; l \leq k; \text{Exec-Equal-State localState trans-fun};$
 $\forall n \leq i. \text{State-Idle localState output-fun trans-fun } ($
 $i\text{-Exec-Comp-Stream-Acc-LocalState } l \ \text{localState trans-fun input } c \ n) \rrbracket \implies$
 $i\text{-Exec-Comp-Stream-Acc-Output } k \ \text{output-fun trans-fun input } c \ i =$
 $i\text{-Exec-Comp-Stream-Acc-Output } l \ \text{output-fun trans-fun input } c \ i$
 <proof>

lemma $i\text{-Exec-Stream-Acc-Output--State-Idle-eq}$ [rule-format]:
 $\llbracket 0 < l; l \leq k; \text{Exec-Equal-State localState trans-fun};$
 $\forall n. \text{State-Idle localState output-fun trans-fun } ($
 $i\text{-Exec-Comp-Stream-Acc-LocalState } l \ \text{localState trans-fun input } c \ n) \rrbracket \implies$
 $i\text{-Exec-Comp-Stream-Acc-Output } k \ \text{output-fun trans-fun input } c =$
 $i\text{-Exec-Comp-Stream-Acc-Output } l \ \text{output-fun trans-fun input } c$
 <proof>

When a certain number l of steps suffices to reach an idle state from any other idle state, than for any acceleration factor $l \leq k$ the accelerated processing of every input message will be finished in an idle state.

lemma $f\text{-Exec-Stream-Acc-LocalState--State-Idle-all}$ [rule-format]:
 $\bigwedge c \ xs. \llbracket 0 < l; l \leq k;$
 $\text{State-Idle localState output-fun trans-fun } (\text{localState } c);$
 $\forall c \ m. \text{State-Idle localState output-fun trans-fun } (\text{localState } c) \longrightarrow$
 $\text{State-Idle localState output-fun trans-fun } ($
 $\text{localState } (f\text{-Exec-Comp trans-fun } (m \ \# \ \varepsilon^l - \text{Suc } 0) \ c));$
 $i < \text{length } xs \] \implies$
 $\text{State-Idle localState output-fun trans-fun } ($
 $f\text{-Exec-Comp-Stream-Acc-LocalState } k \ \text{localState trans-fun } xs \ c \ ! \ i)$
 <proof>

lemma $i\text{-Exec-Stream-Acc-LocalState--State-Idle-all}$ [rule-format]:
 $\llbracket 0 < l; l \leq k;$
 $\text{State-Idle localState output-fun trans-fun } (\text{localState } c);$
 $\forall c \ m. \text{State-Idle localState output-fun trans-fun } (\text{localState } c) \longrightarrow$
 $\text{State-Idle localState output-fun trans-fun } ($

$localState (f-Exec-Comp trans-fun (m \# \varepsilon^l - Suc\ 0) c)) \]] \implies$
 $State-Idle localState output-fun trans-fun ($
 $i-Exec-Comp-Stream-Acc-LocalState k localState trans-fun xs c i)$
 ⟨proof⟩

lemma *f-Exec-Stream-Acc-Output--State-Idle-all-imp-eq*[rule-format]:
 $\llbracket 0 < l; l \leq k; Exec-Equal-State localState trans-fun;$
 $State-Idle localState output-fun trans-fun (localState c);$
 $\forall c m. State-Idle localState output-fun trans-fun (localState c) \longrightarrow$
 $State-Idle localState output-fun trans-fun ($
 $localState (f-Exec-Comp trans-fun (m \# \varepsilon^l - Suc\ 0) c)) \]] \implies$
 $f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun xs c =$
 $f-Exec-Comp-Stream-Acc-Output l output-fun trans-fun xs c$
 ⟨proof⟩

lemma *i-Exec-Stream-Acc-Output--State-Idle-all-imp-eq*[rule-format]:
 $\llbracket 0 < l; l \leq k; Exec-Equal-State localState trans-fun;$
 $State-Idle localState output-fun trans-fun (localState c);$
 $\forall c m. State-Idle localState output-fun trans-fun (localState c) \longrightarrow$
 $State-Idle localState output-fun trans-fun ($
 $localState (f-Exec-Comp trans-fun (m \# \varepsilon^l - Suc\ 0) c)) \]] \implies$
 $i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c =$
 $i-Exec-Comp-Stream-Acc-Output l output-fun trans-fun input c$
 ⟨proof⟩

lemma *f-Exec-Stream-Acc-LocalState--State-Idle-all-imp-eq*[rule-format]:
 $\llbracket 0 < l; l \leq k; Exec-Equal-State localState trans-fun;$
 $State-Idle localState output-fun trans-fun (localState c);$
 $\forall c m. State-Idle localState output-fun trans-fun (localState c) \longrightarrow$
 $State-Idle localState output-fun trans-fun ($
 $localState (f-Exec-Comp trans-fun (m \# \varepsilon^l - Suc\ 0) c)) \]] \implies$
 $f-Exec-Comp-Stream-Acc-LocalState k localState trans-fun xs c =$
 $f-Exec-Comp-Stream-Acc-LocalState l localState trans-fun xs c$
 ⟨proof⟩

lemma *i-Exec-Stream-Acc-LocalState--State-Idle-all-imp-eq*[rule-format]:
 $\llbracket 0 < l; l \leq k; Exec-Equal-State localState trans-fun;$
 $State-Idle localState output-fun trans-fun (localState c);$
 $\forall c m. State-Idle localState output-fun trans-fun (localState c) \longrightarrow$
 $State-Idle localState output-fun trans-fun ($
 $localState (f-Exec-Comp trans-fun (m \# \varepsilon^l - Suc\ 0) c)) \]] \implies$
 $i-Exec-Comp-Stream-Acc-LocalState k localState trans-fun xs c =$
 $i-Exec-Comp-Stream-Acc-LocalState l localState trans-fun xs c$
 ⟨proof⟩

Converting inputs

lemma *f-Exec-input-map*: $\bigwedge c.$
 $f-Exec-Comp trans-fun (map f xs) c = f-Exec-Comp (trans-fun \circ f) xs c$
 ⟨proof⟩

lemma *f-Exec-Stream-input-map*:
 $f\text{-Exec-Comp-Stream trans-fun (map f xs) c} =$
 $f\text{-Exec-Comp-Stream (trans-fun } \circ f) xs c$
 ⟨proof⟩

lemma *i-Exec-Stream-input-map*:
 $i\text{-Exec-Comp-Stream trans-fun (f } \circ \text{input) c} =$
 $i\text{-Exec-Comp-Stream (trans-fun } \circ f) \text{input c}$
 ⟨proof⟩

end

4 AutoFocus message streams and temporal logic on intervals

theory *IL-AF-Stream*

imports *Main Nat-Interval-Logic.IL-TemporalOperators AF-Stream*
begin

4.1 Stream views – joining streams and intervals

4.1.1 Basic definitions

primrec *f-join-aux* :: 'a list \Rightarrow nat \Rightarrow iT \Rightarrow 'a list

where

$f\text{-join-aux } [] n I = []$
 $| f\text{-join-aux } (x \# xs) n I =$
 $(\text{if } n \in I \text{ then } [x] \text{ else } []) @ f\text{-join-aux } xs (\text{Suc } n) I$

The functions *f-join* and *i-join* deliver views of finite and infinite streams through intervals (more exactly: arbitrary natural sets). A stream view contains only the elements of the original stream at positions, which are contained in the interval. For instance, $f\text{-join } [0,10,20,30,40] \{1,4\} = [10,40]$

definition *f-join* :: 'a list \Rightarrow iT \Rightarrow 'a list (**infixl** \bowtie_f 100)
where $xs \bowtie_f I \equiv f\text{-join-aux } xs 0 I$

definition *i-join* :: 'a ilist \Rightarrow iT \Rightarrow 'a ilist (**infixl** \bowtie_i 100)
where $f \bowtie_i I \equiv \lambda n. (f (I \rightarrow n))$

notation

f-join (**infixl** \bowtie 100) **and**
i-join (**infixl** \bowtie 100)

The function *i-f-join* can be used for the case, when an infinite stream is joined with a finite interval. The function *i-join* would then deliver an infinite stream, whose elements after position *card I* are equal to initial stream's element at position *Max I*. The function *i-f-join* in contrast cuts the resulting stream at this position and returns a finite stream.

definition $i\text{-}f\text{-}join :: 'a\ list \Rightarrow iT \Rightarrow 'a\ list$ (**infixl** \bowtie_{i-f} 100)

where $f \bowtie_{i-f} I \equiv f \Downarrow Suc (Max I) \bowtie_f I$

notation

$i\text{-}f\text{-}join$ (**infixl** \bowtie 100)

The function $i\text{-}f\text{-}join$ should be used only for finite sets in order to deliver well-defined results. The function $i\text{-}join$ should be used for infinite sets, because joining an infinite stream s and a finite set I using $i\text{-}join$ would deliver an infinite stream, ending with an infinite sequence of elements equal to $s (Max I)$.

4.1.2 Basic results

lemma $f\text{-}join\text{-}aux\text{-}length$:

$\bigwedge n. length (f\text{-}join\text{-}aux\ xs\ n\ I) = card (I \cap \{n..<n + length\ xs\})$
 $\langle proof \rangle$

lemma $f\text{-}join\text{-}aux\text{-}nth$ [*rule-format*]:

$\forall n\ i. i < card (I \cap \{n..<n + length\ xs\}) \longrightarrow$
 $(f\text{-}join\text{-}aux\ xs\ n\ I) ! i = xs ! ((I \cap \{n..<n + length\ xs\}) \rightarrow i) - n$
 $\langle proof \rangle$

Joining finite streams and intervals

lemma $f\text{-}join\text{-}length$: $length (xs \bowtie_f I) = card (I \Downarrow length\ xs)$

$\langle proof \rangle$

lemma $f\text{-}join\text{-}nth$: $n < length (xs \bowtie_f I) \implies (xs \bowtie_f I) ! n = xs ! (I \rightarrow n)$

$\langle proof \rangle$

lemma $f\text{-}join\text{-}nth2$: $n < card (I \Downarrow length\ xs) \implies (xs \bowtie_f I) ! n = xs ! (I \rightarrow n)$

$\langle proof \rangle$

lemma $f\text{-}join\text{-}empty$: $xs \bowtie_f \{\} = []$

$\langle proof \rangle$

lemma $f\text{-}join\text{-}Nil$: $[] \bowtie_f I = []$

$\langle proof \rangle$

lemma $f\text{-}join\text{-}Nil\text{-}conv$: $(xs \bowtie_f I = []) = (I \Downarrow length\ xs = \{\})$

$\langle proof \rangle$

lemma $f\text{-}join\text{-}Nil\text{-}conv'$: $(xs \bowtie_f I = []) = (\forall i < length\ xs. i \notin I)$

$\langle proof \rangle$

lemma $f\text{-}join\text{-}all\text{-}conv$: $(xs \bowtie_f I = xs) = (\{..<length\ xs\} \subseteq I)$

$\langle proof \rangle$

lemma $f\text{-}join\text{-}all$: $\{..<length\ xs\} \subseteq I \implies xs \bowtie_f I = xs$

<proof>

corollary *f-join-UNIV*: $xs \bowtie_f UNIV = xs$

<proof>

lemma *f-join-union*:

$$\llbracket \text{finite } A; \text{Max } A < \text{iMin } B \rrbracket \implies xs \bowtie_f (A \cup B) = xs \bowtie_f A @ (xs \bowtie_f B)$$

<proof>

lemma *f-join-singleton-if*:

$$xs \bowtie_f \{n\} = (\text{if } n < \text{length } xs \text{ then } [xs ! n] \text{ else } [])$$

<proof>

lemma *f-join-insert*:

$$n < \text{length } xs \implies$$

$$xs \bowtie_f \text{insert } n \ I = xs \bowtie_f (I \downarrow < n) @ (xs ! n) \# (xs \bowtie_f (I \downarrow > n))$$

<proof>

lemma *f-join-snoc*:

$$(xs @ [x]) \bowtie_f I =$$

$$xs \bowtie_f I @ (\text{if } \text{length } xs \in I \text{ then } [x] \text{ else } [])$$

<proof>

lemma *f-join-append*:

$$(xs @ ys) \bowtie_f I = xs \bowtie_f I @ ys \bowtie_f (I \oplus - (\text{length } xs))$$

<proof>

lemma *take-f-join-eq1*:

$$n < \text{card } (I \downarrow < \text{length } xs) \implies$$

$$(xs \bowtie_f I) \downarrow n = xs \bowtie_f (I \downarrow < (I \rightarrow n))$$

<proof>

lemma *take-f-join-eq2*:

$$\text{card } (I \downarrow < \text{length } xs) \leq n \implies (xs \bowtie_f I) \downarrow n = xs \bowtie_f I$$

<proof>

lemma *take-f-join-if*:

$$(xs \bowtie_f I) \downarrow n =$$

$$(\text{if } n < \text{card } (I \downarrow < \text{length } xs) \text{ then } xs \bowtie_f (I \downarrow < (I \rightarrow n)) \text{ else } xs \bowtie_f I)$$

<proof>

lemma *drop-f-join-eq1*:

$$n < \text{card } (I \downarrow < \text{length } xs) \implies$$

$$(xs \bowtie_f I) \uparrow n = xs \bowtie_f (I \downarrow \geq (I \rightarrow n))$$

<proof>

lemma *drop-f-join-eq2*:

$$\text{card } (I \downarrow < \text{length } xs) \leq n \implies (xs \bowtie_f I) \uparrow n = []$$

<proof>

lemma *drop-f-join-if*:

$(xs \bowtie_f I) \uparrow n =$
(if $n < \text{card } (I \downarrow < \text{length } xs)$ *then* $xs \bowtie_f (I \downarrow \geq (I \rightarrow n))$ *else* \square)
 ⟨proof⟩

lemma *f-join-take*: $xs \downarrow n \bowtie_f I = xs \bowtie_f (I \downarrow < n)$

⟨proof⟩

lemma *f-join-drop*: $xs \uparrow n \bowtie_f I = xs \bowtie_f (I \oplus n)$

⟨proof⟩

lemma *cut-less-eq-imp-f-join-eq*:

$A \downarrow < \text{length } xs = B \downarrow < \text{length } xs \implies xs \bowtie_f A = xs \bowtie_f B$
 ⟨proof⟩

corollary *f-join-cut-less-eq*:

$\text{length } xs \leq t \implies xs \bowtie_f (I \downarrow < t) = xs \bowtie_f I$
 ⟨proof⟩

lemma *take-Suc-Max-eq-imp-f-join-eq*:

$\llbracket \text{finite } I; xs \downarrow \text{Suc } (\text{Max } I) = ys \downarrow \text{Suc } (\text{Max } I) \rrbracket \implies$
 $xs \bowtie_f I = ys \bowtie_f I$
 ⟨proof⟩

corollary *f-join-take-Suc-Max-eq*:

$\text{finite } I \implies xs \downarrow \text{Suc } (\text{Max } I) \bowtie_f I = xs \bowtie_f I$
 ⟨proof⟩

Joining infinite streams and infinite intervals

lemma *i-join-nth*: $(f \bowtie_i I) n = f (I \rightarrow n)$

⟨proof⟩

lemma *i-join-UNIV*: $f \bowtie_i \text{UNIV} = f$

⟨proof⟩

lemma *i-join-union*:

$\llbracket \text{finite } A; \text{Max } A < \text{iMin } B; B \neq \{\} \rrbracket \implies$
 $f \bowtie_i (A \cup B) = (f \downarrow \text{Suc } (\text{Max } A) \bowtie_f A) \frown (f \bowtie_i B)$
 ⟨proof⟩

lemma *i-join-singleton*: $f \bowtie_i \{a\} = (\lambda n. f a)$

⟨proof⟩

lemma *i-join-insert*:

$f \bowtie_i (\text{insert } n I) =$
 $(f \downarrow n) \bowtie_f (I \downarrow < n) \frown [f n] \frown ($
 $\text{if } I \downarrow > n = \{\} \text{ then } (\lambda x. f n) \text{ else } f \bowtie_i (I \downarrow > n))$
 ⟨proof⟩

lemma *i-join-i-append*:

$$\text{infinite } I \implies (xs \frown f) \bowtie_i I = (xs \bowtie_f I) \frown (f \bowtie_i (I \oplus - \text{length } xs))$$

<proof>

lemma *i-take-i-join*: $\text{infinite } I \implies f \bowtie_i I \Downarrow n = f \Downarrow (I \rightarrow n) \bowtie_f I$

<proof>

lemma *i-drop-i-join*: $I \neq \{\}$ $\implies f \bowtie_i I \Uparrow n = f \bowtie_i (I \Downarrow \geq (I \rightarrow n))$

<proof>

lemma *i-join-i-take*: $f \Downarrow n \bowtie_f I = f \bowtie_i I \Downarrow \text{card } (I \Downarrow < n)$

<proof>

lemma *i-join-i-drop*: $I \neq \{\}$ $\implies f \Uparrow n \bowtie_i I = f \bowtie_i (I \oplus n)$

<proof>

lemma *i-join-finite-nth-ge-card-eq-nth-Max*:

$$\llbracket \text{finite } I; I \neq \{\}; \text{card } I \leq \text{Suc } n \rrbracket \implies (f \bowtie_i I) n = f (\text{Max } I)$$

<proof>

lemma *i-join-finite-i-drop-card-eq-const-nth-Max*:

$$\llbracket \text{finite } I; I \neq \{\} \rrbracket \implies (f \bowtie_i I) \Uparrow (\text{card } I) = (\lambda n. f (\text{Max } I))$$

<proof>

lemma *i-join-finite-i-append-nth-Max-conv*:

$$\llbracket \text{finite } I; I \neq \{\} \rrbracket \implies (f \bowtie_i I) = f \Downarrow \text{Suc } (\text{Max } I) \bowtie_f I \frown (\lambda n. f (\text{Max } I))$$

<proof>

Joining infinite streams and finite intervals

lemma *i-f-join-length*: $\text{finite } I \implies \text{length } (f \bowtie_{i-f} I) = \text{card } I$

<proof>

lemma *i-f-join-nth*: $n < \text{card } I \implies f \bowtie_{i-f} I ! n = f (I \rightarrow n)$

<proof>

lemma *i-f-join-empty*: $f \bowtie_{i-f} \{\} = []$

<proof>

lemma *i-f-join-eq-i-join-i-take*:

$$\text{finite } I \implies f \bowtie_{i-f} I = f \bowtie_i I \Downarrow (\text{card } I)$$

<proof>

lemma *i-f-join-union*:

$$\llbracket \text{finite } A; \text{finite } B; \text{Max } A < \text{iMin } B \rrbracket \implies f \bowtie_{i-f} (A \cup B) = f \bowtie_{i-f} A @ f \bowtie_{i-f} B$$

<proof>

lemma *i-f-join-singleton*: $f \bowtie_{i-f} \{n\} = [f n]$

<proof>

lemma *i-f-join-insert:*

finite I \implies
 $f \bowtie_{i-f} \text{insert } n \ I = f \bowtie_{i-f} (I \downarrow < n) @ f \ n \ \# \ f \ \bowtie_{i-f} (I \downarrow > n)$
<proof>

lemma *take-i-f-join-eq1:*

$n < \text{card } I \implies f \ \bowtie_{i-f} \ I \ \downarrow \ n = f \ \bowtie_{i-f} \ (I \ \downarrow < \ (I \ \rightarrow \ n))$
<proof>

lemma *take-i-f-join-eq2:*

$\llbracket \text{finite } I; \text{card } I \leq n \rrbracket \implies f \ \bowtie_{i-f} \ I \ \downarrow \ n = f \ \bowtie_{i-f} \ I$
<proof>

lemma *take-i-f-join-if:*

finite I \implies
 $f \ \bowtie_{i-f} \ I \ \downarrow \ n = (\text{if } n < \text{card } I \text{ then } f \ \bowtie_{i-f} \ (I \ \downarrow < \ (I \ \rightarrow \ n)) \ \text{else } f \ \bowtie_{i-f} \ I)$
<proof>

lemma *drop-i-f-join-eq1:*

$n < \text{card } I \implies f \ \bowtie_{i-f} \ I \ \uparrow \ n = f \ \bowtie_{i-f} \ (I \ \downarrow \geq \ (I \ \rightarrow \ n))$
<proof>

lemma *drop-i-f-join-eq2:*

$\llbracket \text{finite } I; \text{card } I \leq n \rrbracket \implies f \ \bowtie_{i-f} \ I \ \uparrow \ n = \llbracket \rrbracket$
<proof>

lemma *drop-i-f-join-if:*

finite I \implies
 $f \ \bowtie_{i-f} \ I \ \uparrow \ n = (\text{if } n < \text{card } I \text{ then } f \ \bowtie_{i-f} \ (I \ \downarrow \geq \ (I \ \rightarrow \ n)) \ \text{else } \llbracket \rrbracket)$
<proof>

lemma *i-f-join-i-drop:*

finite I $\implies f \ \uparrow \ n \ \bowtie_{i-f} \ I = f \ \bowtie_{i-f} \ (I \ \oplus \ n)$
<proof>

lemma *i-take-Suc-Max-eq-imp-i-f-join-eq:*

$f \ \downarrow \ \text{Suc} \ (\text{Max } I) = g \ \downarrow \ \text{Suc} \ (\text{Max } I) \implies f \ \bowtie_{i-f} \ I = g \ \bowtie_{i-f} \ I$
<proof>

lemma *i-take-i-join-eq-i-f-join:*

infinite I $\implies f \ \bowtie_i \ I \ \downarrow \ n = f \ \bowtie_{i-f} \ (I \ \downarrow < \ (I \ \rightarrow \ n))$
<proof>

4.1.3 Results for intervals from *IL-Interval*

lemma *f-join-iFROM:* $xs \ \bowtie_f \ [n..] = xs \ \uparrow \ n$

<proof>

lemma *i-join-iFROM*: $f \bowtie_i [n..] = f \uparrow n$
 ⟨proof⟩

lemma *f-join-iIN*: $xs \bowtie_f [n..,d] = xs \uparrow n \downarrow \text{Suc } d$
 ⟨proof⟩

lemma *i-f-join-iIN*: $f \bowtie_{i-f} [n..,d] = f \uparrow n \downarrow \text{Suc } d$
 ⟨proof⟩

lemma *f-join-iTILL*: $xs \bowtie_f [..n] = xs \downarrow (\text{Suc } n)$
 ⟨proof⟩

lemma *i-f-join-iTILL*: $f \bowtie_{i-f} [..n] = f \downarrow \text{Suc } n$
 ⟨proof⟩

lemma *f-join-f-expand-iT-Mult*:
 $0 < k \implies xs \odot_f k \bowtie_f (I \otimes k) = xs \bowtie_f I$
 ⟨proof⟩

lemma *i-join-i-expand-iT-Mult*:
 $\llbracket 0 < k; I \neq \{\} \rrbracket \implies f \odot_i k \bowtie_i (I \otimes k) = f \bowtie_i I$
 ⟨proof⟩

lemma *i-f-join-i-expand-iT-Mult*:
 $\llbracket 0 < k; \text{finite } I \rrbracket \implies f \odot_i k \bowtie_{i-f} (I \otimes k) = f \bowtie_{i-f} I$
 ⟨proof⟩

lemma *f-join-f-shrink-iT-Plus-iT-Div-mod*:
 $\llbracket 0 < k; \forall x \in I. x \bmod k = 0 \rrbracket \implies$
 $(xs \mapsto_f k) \bowtie_f (I \oplus (k - 1)) = xs \div_f k \bowtie_f (I \oslash k)$
 ⟨proof⟩

lemma *i-join-i-shrink-iT-Plus-iT-Div-mod*:
 $\llbracket 0 < k; I \neq \{\}; \forall x \in I. x \bmod k = 0 \rrbracket \implies$
 $(f \mapsto_i k) \bowtie_i (I \oplus (k - 1)) = f \div_i k \bowtie_i (I \oslash k)$
 ⟨proof⟩

lemma *i-f-join-i-shrink-iT-Plus-iT-Div-mod*:
 $\llbracket 0 < k; \text{finite } I; \forall x \in I. x \bmod k = 0 \rrbracket \implies$
 $(f \mapsto_{i-f} k) \bowtie_{i-f} (I \oplus (k - 1)) = f \div_{i-f} k \bowtie_{i-f} (I \oslash k)$
 ⟨proof⟩

corollary *f-join-f-shrink-iT-Plus-iT-Div-mod-subst*:
 $\llbracket 0 < k; \forall x \in I. x \bmod k = 0;$
 $A = I \oplus (k - 1); B = I \oslash k \rrbracket \implies$
 $(xs \mapsto_f k) \bowtie_f A = xs \div_f k \bowtie_f B$
 ⟨proof⟩

corollary *i-join-i-shrink-iT-Plus-iT-Div-mod-subst:*

$$\begin{aligned} & \llbracket 0 < k; I \neq \{\}; \forall x \in I. x \bmod k = 0; \\ & A = I \oplus (k - 1); B = I \odot k \rrbracket \implies \\ & (f \mapsto_i k) \bowtie_i A = f \div_i k \bowtie_i B \end{aligned}$$

<proof>

corollary *i-f-join-i-shrink-iT-Plus-iT-Div-mod-subst:*

$$\begin{aligned} & \llbracket 0 < k; \text{finite } I; \forall x \in I. x \bmod k = 0; \\ & A = I \oplus (k - 1); B = I \odot k \rrbracket \implies \\ & (f \mapsto_i k) \bowtie_{i-f} A = f \div_i k \bowtie_{i-f} B \end{aligned}$$

<proof>

lemma *f-join-f-shrink-iT-Div-mod:*

$$\begin{aligned} & \llbracket 0 < k; \forall x \in I. x \bmod k = k - 1 \rrbracket \implies \\ & (xs \mapsto_f k) \bowtie_f I = xs \div_f k \bowtie_f (I \odot k) \end{aligned}$$

<proof>

lemma *i-join-i-shrink-iT-Div-mod:*

$$\begin{aligned} & \llbracket 0 < k; I \neq \{\}; \forall x \in I. x \bmod k = k - 1 \rrbracket \implies \\ & (f \mapsto_i k) \bowtie_i I = f \div_i k \bowtie_i (I \odot k) \end{aligned}$$

<proof>

lemma *i-f-join-i-shrink-iT-Div-mod:*

$$\begin{aligned} & \llbracket 0 < k; \text{finite } I; \forall x \in I. x \bmod k = k - 1 \rrbracket \implies \\ & (f \mapsto_i k) \bowtie_{i-f} I = f \div_i k \bowtie_{i-f} (I \odot k) \end{aligned}$$

<proof>

lemma *f-join-f-expand-iMOD:*

$$0 < k \implies xs \odot_f k \bowtie_f [n * k, \text{mod } k] = xs \bowtie_f [n..]$$

<proof>

corollary *f-join-f-expand-iMOD-0:*

$$0 < k \implies xs \odot_f k \bowtie_f [0, \text{mod } k] = xs$$

<proof>

lemma *f-join-f-expand-iMODb:*

$$0 < k \implies xs \odot_f k \bowtie_f [n * k, \text{mod } k, d] = xs \bowtie_f [n.., d]$$

<proof>

corollary *f-join-f-expand-iMODb-0:*

$$0 < k \implies xs \odot_f k \bowtie_f [0, \text{mod } k, n] = xs \bowtie_f [..n]$$

<proof>

lemma *i-join-i-expand-iMOD:*

$$0 < k \implies f \odot_i k \bowtie_i [n * k, \text{mod } k] = f \bowtie_i [n..]$$

<proof>

corollary *i-join-i-expand-iMOD-0:*

$$0 < k \implies f \odot_i k \bowtie_i [0, \text{mod } k] = f$$

<proof>

lemma *i-join-i-expand-iMODb*:

$$0 < k \implies f \odot_i k \bowtie_i [n * k, \text{mod } k, d] = f \bowtie_i [n \dots, d]$$

<proof>

corollary *i-join-i-expand-iMODb-0*:

$$0 < k \implies f \odot_i k \bowtie_i [0, \text{mod } k, n] = f \bowtie_i [\dots n]$$

<proof>

lemma *i-f-join-i-expand-iMODb*:

$$0 < k \implies f \odot_i k \bowtie_{i-f} [n * k, \text{mod } k, d] = f \bowtie_{i-f} [n \dots, d]$$

<proof>

corollary *i-f-join-i-expand-iMODb-0*:

$$0 < k \implies f \odot_i k \bowtie_{i-f} [0, \text{mod } k, n] = f \bowtie_{i-f} [\dots n]$$

<proof>

lemma *f-join-f-shrink-iMOD*:

$$0 < k \implies (xs \mapsto_f k) \bowtie_f [n * k + (k - 1), \text{mod } k] = xs \div_f k \bowtie_f [n \dots]$$

<proof>

corollary *f-join-f-shrink-iMOD-0*:

$$0 < k \implies (xs \mapsto_f k) \bowtie_f [k - 1, \text{mod } k] = xs \div_f k$$

<proof>

lemma *f-join-f-shrink-iMODb*:

$$0 < k \implies (xs \mapsto_f k) \bowtie_f [n * k + (k - 1), \text{mod } k, d] = xs \div_f k \bowtie_f [n \dots, d]$$

<proof>

corollary *f-join-f-shrink-iMODb-0*:

$$0 < k \implies (xs \mapsto_f k) \bowtie_f [k - 1, \text{mod } k, n] = xs \div_f k \bowtie_f [\dots n]$$

<proof>

lemma *i-join-i-shrink-iMOD*:

$$0 < k \implies (f \mapsto_i k) \bowtie_i [n * k + (k - 1), \text{mod } k] = f \div_i k \bowtie_i [n \dots]$$

<proof>

corollary *i-join-i-shrink-iMOD-0*:

$$0 < k \implies (f \mapsto_i k) \bowtie_i [k - 1, \text{mod } k] = f \div_i k$$

<proof>

lemma *i-f-join-i-shrink-iMODb*:

$$0 < k \implies (f \mapsto_i k) \bowtie_{i-f} [n * k + (k - 1), \text{mod } k, d] = f \div_i k \bowtie_{i-f} [n \dots, d]$$

<proof>

corollary *i-f-join-i-shrink-iMODb-0*:

$$0 < k \implies (f \mapsto_i k) \bowtie_{i-f} [k - 1, \text{mod } k, n] = f \div_i k \bowtie_{i-f} [\dots n]$$

$\langle \text{proof} \rangle$

4.2 Streams and temporal operators

lemma *i-shrink-eq-NoMsg-iAll-conv*:

$$0 < k \implies ((s \dot{\div}_i k) t = \varepsilon) = (\Box t1 [t * k \dots, k - \text{Suc } 0]. s t1 = \varepsilon)$$

$\langle \text{proof} \rangle$

lemma *i-shrink-eq-NoMsg-iAll-conv2*:

$$0 < k \implies ((s \dot{\div}_i k) t = \varepsilon) = (\Box t1 [\dots k - 1] \oplus (t * k). s t1 = \varepsilon)$$

$\langle \text{proof} \rangle$

lemma *i-shrink-eq-Msg-iEx-iAll-conv*:

$$\begin{aligned} & \llbracket 0 < k; m \neq \varepsilon \rrbracket \implies \\ & ((s \dot{\div}_i k) t = m) = \\ & (\Diamond t1 [t * k \dots, k - \text{Suc } 0]. s t1 = m \wedge \\ & \quad (\Box t2 [\text{Suc } t1 \dots]. t2 \leq t * k + k - \text{Suc } 0 \longrightarrow s t2 = \varepsilon)) \end{aligned}$$

$\langle \text{proof} \rangle$

lemma *i-shrink-eq-Msg-iEx-iAll-conv2*:

$$\begin{aligned} & \llbracket 0 < k; m \neq \varepsilon \rrbracket \implies \\ & ((s \dot{\div}_i k) t = m) = \\ & (\Diamond t1 [\dots k - 1] \oplus (t * k). s t1 = m \wedge \\ & \quad (\Box t2 [1 \dots] \oplus t1 . t2 \leq t * k + k - 1 \longrightarrow s t2 = \varepsilon)) \end{aligned}$$

$\langle \text{proof} \rangle$

lemma *i-shrink-eq-Msg-iEx-iAll-cut-greater-conv*:

$$\begin{aligned} & \llbracket 0 < k; m \neq \varepsilon \rrbracket \implies \\ & ((s \dot{\div}_i k) t = m) = \\ & (\Diamond t1 [t * k \dots, k - \text{Suc } 0]. s t1 = m \wedge \\ & \quad (\Box t2 [t * k \dots, k - \text{Suc } 0] \downarrow > t1. s t2 = \varepsilon)) \end{aligned}$$

$\langle \text{proof} \rangle$

lemma *i-shrink-eq-Msg-iEx-iAll-cut-greater-conv2*:

$$\begin{aligned} & \llbracket 0 < k; m \neq \varepsilon \rrbracket \implies \\ & ((s \dot{\div}_i k) t = m) = \\ & (\Diamond t1 [\dots k - 1] \oplus (t * k). s t1 = m \wedge \\ & \quad (\Box t2 ([\dots k - 1] \oplus (t * k)) \downarrow > t1. s t2 = \varepsilon)) \end{aligned}$$

$\langle \text{proof} \rangle$

lemma *i-shrink-eq-Msg-iSince-conv*:

$$\begin{aligned} & \llbracket 0 < k; m \neq \varepsilon \rrbracket \implies \\ & ((s \dot{\div}_i k) t = m) = \\ & (s t2 = \varepsilon. t2 \mathcal{S} t1 [t * k \dots, k - \text{Suc } 0]. s t1 = m) \end{aligned}$$

$\langle \text{proof} \rangle$

lemma *i-shrink-eq-Msg-iSince-conv2*:

$$\begin{aligned} & \llbracket 0 < k; m \neq \varepsilon \rrbracket \implies \\ & ((s \dot{\div}_i k) t = m) = \\ & (s t2 = \varepsilon. t2 \mathcal{S} t1 [\dots k - 1] \oplus (t * k). s t1 = m) \end{aligned}$$

<proof>

lemma *iT-Mult-iAll-i-expand-nth-iff:*

$0 < k \implies (\Box t (I \otimes k). P ((f \odot_i k) t)) = (\Box t I. P (f t))$
<proof>

Streams and temporal operators cycle start/finish events

lemma *i-shrink-eq-NoMsg-iAll-start-event-conv:*

$\llbracket 0 < k; \bigwedge t. \text{event } t = (t \bmod k = 0); t0 = t * k \rrbracket \implies$
 $((s \div_i k) t = \varepsilon) =$
 $(s t0 = \varepsilon \wedge (\bigcirc t' t0 [0\dots]. (s t1 = \varepsilon. t1 \mathcal{U} t2 ([0\dots] \oplus t'). \text{event } t2)))$
<proof>

lemma *i-shrink-eq-Msg-iUntil-start-event-conv:*

$\llbracket 0 < k; m \neq \varepsilon; \bigwedge t. \text{event } t = (t \bmod k = 0); t0 = t * k \rrbracket \implies$
 $((s \div_i k) t = m) = ($
 $(s t0 = m \wedge (\bigcirc t' t0 [0\dots]. (s t1 = \varepsilon. t1 \mathcal{U} t2 ([0\dots] \oplus t'). \text{event } t2))) \vee$
 $(\bigcirc t' t0 [0\dots]. (\neg \text{event } t1. t1 \mathcal{U} t2 ([0\dots] \oplus t'). ($
 $s t2 = m \wedge \neg \text{event } t2 \wedge (\bigcirc t'' t2 [0\dots].$
 $(s t3 = \varepsilon. t3 \mathcal{U} t4 ([0\dots] \oplus t''). \text{event } t4))))))$
<proof>

lemma *i-shrink-eq-NoMsg-iAll-finish-event-conv:*

$\llbracket 1 < k; \bigwedge t. \text{event } t = (t \bmod k = k - 1); t0 = t * k \rrbracket \implies$
 $((s \div_i k) t = \varepsilon) =$
 $(s t0 = \varepsilon \wedge (\bigcirc t' t0 [0\dots]. (s t1 = \varepsilon. t1 \mathcal{U} t2 ([0\dots] \oplus t'). (\text{event } t2 \wedge s t2 =$
 $\varepsilon))))$
<proof>

lemma *i-shrink-eq-Msg-iUntil-finish-event-conv:*

$\llbracket 1 < k; m \neq \varepsilon; \bigwedge t. \text{event } t = (t \bmod k = k - 1); t0 = t * k \rrbracket \implies$
 $((s \div_i k) t = m) = ($
 $(\neg \text{event } t1. t1 \mathcal{U} t2 ([0\dots] \oplus t0). \text{event } t2 \wedge s t2 = m) \vee$
 $(\neg \text{event } t1. t1 \mathcal{U} t2 ([0\dots] \oplus t0). (\neg \text{event } t2 \wedge s t2 = m \wedge ($
 $\bigcirc t' t2 [0\dots]. (s t3 = \varepsilon. t3 \mathcal{U} t4 ([0\dots] \oplus t'). \text{event } t4 \wedge s t4 = \varepsilon))))$
<proof>

end

5 AutoFocus message stream processing and temporal logic on intervals

theory *IL-AF-Stream-Exec*

imports *Main IL-AF-Stream AF-Stream-Exec*

begin

5.1 Correlation between Pre/Post-Conditions for f -Exec-Comp-Stream and f -Exec-Comp-Stream-Init

lemma i -Exec-Stream-Pre-Post1-iAll:

$$\begin{aligned} & \llbracket \text{result} = i\text{-Exec-Comp-Stream trans-fun input } c; \\ & \quad \forall x\text{-}n \text{ } c\text{-}n. P1 \ x\text{-}n \wedge P2 \ c\text{-}n \longrightarrow Q \ (\text{trans-fun } x\text{-}n \ c\text{-}n) \rrbracket \Longrightarrow \\ & \square t \ I. (P1 \ (\text{input } t) \wedge P2 \ (\text{result}^{\leftarrow c} \ t) \longrightarrow Q \ (\text{result } t)) \\ & \langle \text{proof} \rangle \end{aligned}$$

Direct relation between input and result after transition

lemma i -Exec-Stream-Pre-Post2-iAll:

$$\begin{aligned} & \llbracket \text{result} = i\text{-Exec-Comp-Stream trans-fun input } c; \\ & \quad \forall x\text{-}n \ c\text{-}n. P \ c\text{-}n \longrightarrow Q \ x\text{-}n \ (\text{trans-fun } x\text{-}n \ c\text{-}n) \rrbracket \Longrightarrow \\ & \square t \ I. P \ (\text{result}^{\leftarrow c} \ t) \longrightarrow Q \ (\text{input } t) \ (\text{result } t) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma i -Exec-Stream-Pre-Post3-iAll-iNext:

$$\begin{aligned} & \llbracket \text{result} = i\text{-Exec-Comp-Stream trans-fun input } c; \\ & \quad \forall x\text{-}n \ c\text{-}n. P \ c\text{-}n \longrightarrow Q \ x\text{-}n \ (\text{trans-fun } x\text{-}n \ c\text{-}n); \\ & \quad \forall t \in I. \text{inext } t \ I' = \text{Suc } t \rrbracket \Longrightarrow \\ & \square t \ I. P \ (\text{result } t) \longrightarrow (\circ \ t1 \ t \ I'. Q \ (\text{input } t1) \ (\text{result } t1)) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma i -Exec-Stream-Init-Pre-Post1-iAll-iNext:

$$\begin{aligned} & \llbracket \text{result} = i\text{-Exec-Comp-Stream-Init trans-fun input } c; \\ & \quad \forall x\text{-}n \ c\text{-}n. P1 \ x\text{-}n \wedge P2 \ c\text{-}n \longrightarrow Q \ (\text{trans-fun } x\text{-}n \ c\text{-}n); \\ & \quad \forall t \in I. \text{inext } t \ I' = \text{Suc } t \rrbracket \Longrightarrow \\ & \square t \ I. (P1 \ (\text{input } t) \wedge P2 \ (\text{result } t) \longrightarrow (\circ \ t1 \ t \ I'. Q \ (\text{result } t1))) \\ & \langle \text{proof} \rangle \end{aligned}$$

Direct relation between input and state before transition

lemma i -Exec-Stream-Init-Pre-Post2-iAll-iNext:

$$\begin{aligned} & \llbracket \text{result} = i\text{-Exec-Comp-Stream-Init trans-fun input } c; \\ & \quad \forall x\text{-}n \ c\text{-}n. P \ x\text{-}n \ c\text{-}n \longrightarrow Q \ (\text{trans-fun } x\text{-}n \ c\text{-}n); \\ & \quad \forall t \in I. \text{inext } t \ I' = \text{Suc } t \rrbracket \Longrightarrow \\ & \square t \ I. (P \ (\text{input } t) \ (\text{result } t) \longrightarrow (\circ \ t1 \ t \ I'. Q \ (\text{result } t1))) \\ & \langle \text{proof} \rangle \end{aligned}$$

Relation between input and output

lemma i -Exec-Stream-Init-Pre-Post3-iAll-iNext:

$$\begin{aligned} & \llbracket \text{result} = i\text{-Exec-Comp-Stream-Init trans-fun input } c; \\ & \quad \forall x\text{-}n \ c\text{-}n. P \ c\text{-}n \longrightarrow Q \ x\text{-}n \ (\text{trans-fun } x\text{-}n \ c\text{-}n); \\ & \quad \forall t \in I. \text{inext } t \ I' = \text{Suc } t \rrbracket \Longrightarrow \\ & \square t \ I. (P \ (\text{result } t) \longrightarrow (\circ \ t1 \ t \ I'. Q \ (\text{input}^{\leftarrow \varepsilon} \ t1) \ (\text{result } t1))) \\ & \langle \text{proof} \rangle \end{aligned}$$

5.2 *i-Exec-Comp-Stream-Acc-Output* and temporal operators with bounded intervals.

Temporal relation between uncompressed and compressed output of accelerated components.

lemma *i-Exec-Comp-Stream-Acc-Output--eq-NoMsg-iAll-conv:*

$$\begin{aligned}
& 0 < k \implies \\
& ((i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c) \ t = \varepsilon) = \\
& (\Box t1 [t * k \dots, k - \text{Suc } 0]. (\text{output-fun} \circ i\text{-Exec-Comp-Stream trans-fun } (\text{input} \\
& \odot_i k) \ c) \ t1 = \varepsilon) \\
& \langle \text{proof} \rangle
\end{aligned}$$

lemma *i-Exec-Comp-Stream-Acc-Output--eq-NoMsg-iAll-conv2:*

$$\begin{aligned}
& 0 < k \implies \\
& ((i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c) \ t = \varepsilon) = \\
& (\Box t1 [\dots k - \text{Suc } 0] \oplus (t * k). (\text{output-fun} \circ i\text{-Exec-Comp-Stream trans-fun} \\
& (\text{input} \odot_i k) \ c) \ t1 = \varepsilon) \\
& \langle \text{proof} \rangle
\end{aligned}$$

lemma *i-Exec-Comp-Stream-Acc-Output--Init--eq-NoMsg-iAll-conv:*

$$\begin{aligned}
& 0 < k \implies \\
& ((i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c) \ t = \varepsilon) = \\
& (\Box t1 [\text{Suc } (t * k) \dots, k - \text{Suc } 0]. (\text{output-fun} \circ i\text{-Exec-Comp-Stream-Init trans-fun} \\
& (\text{input} \odot_i k) \ c) \ t1 = \varepsilon) \\
& \langle \text{proof} \rangle
\end{aligned}$$

lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-iEx-iAll-cut-greater-conv:*

$$\begin{aligned}
& \llbracket 0 < k; m \neq \varepsilon; s = (\text{output-fun} \circ i\text{-Exec-Comp-Stream trans-fun } (\text{input} \odot_i k) \\
& c) \rrbracket \implies \\
& ((i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c) \ t = m) = \\
& (\Diamond t1 [t * k \dots, k - \text{Suc } 0]. (s \ t1 = m \wedge \\
& (\Box t2 [t * k \dots, k - \text{Suc } 0] \downarrow > t1 \ . \ s \ t2 = \varepsilon))) \\
& \langle \text{proof} \rangle
\end{aligned}$$

lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-iEx-iAll-cut-greater-conv2:*

$$\begin{aligned}
& \llbracket 0 < k; m \neq \varepsilon; s = (\text{output-fun} \circ i\text{-Exec-Comp-Stream trans-fun } (\text{input} \odot_i k) \\
& c) \rrbracket \implies \\
& ((i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c) \ t = m) = \\
& (\Diamond t1 [\dots k - \text{Suc } 0] \oplus (t * k). (s \ t1 = m \wedge \\
& (\Box t2 ([\dots k - \text{Suc } 0] \oplus (t * k)) \downarrow > t1 \ . \ s \ t2 = \varepsilon))) \\
& \langle \text{proof} \rangle
\end{aligned}$$

lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-iSince-conv:*

$$\begin{aligned}
& \llbracket 0 < k; m \neq \varepsilon; s = (\text{output-fun} \circ i\text{-Exec-Comp-Stream trans-fun } (\text{input} \odot_i k) \\
& c) \rrbracket \implies \\
& ((i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c) \ t = m) = \\
& (s \ t2 = \varepsilon. \ t2 \ \mathcal{S} \ t1 [t * k \dots, k - \text{Suc } 0]. \ s \ t1 = m) \\
& \langle \text{proof} \rangle
\end{aligned}$$

lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-iSince-conv2*:

$\llbracket 0 < k; m \neq \varepsilon; s = (\text{output-fun} \circ \text{i-Exec-Comp-Stream trans-fun} (\text{input} \odot_i k) c) \rrbracket \implies$
 $((\text{i-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c) t = m) =$
 $(s \text{ t2} = \varepsilon. \text{t2 } \mathcal{S} \text{ t1 } [\dots k - \text{Suc } 0] \oplus (t * k). s \text{ t1} = m)$
 <proof>

lemma *i-Exec-Comp-Stream-Acc-Output--Init--eq-Msg-iSince-conv*:

$\llbracket 0 < k; m \neq \varepsilon; s = (\text{output-fun} \circ \text{i-Exec-Comp-Stream-Init trans-fun} (\text{input} \odot_i k) c) \rrbracket \implies$
 $((\text{i-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c) t = m) =$
 $(s \text{ t2} = \varepsilon. \text{t2 } \mathcal{S} \text{ t1 } [\text{Suc } (t * k) \dots, k - \text{Suc } 0]. s \text{ t1} = m)$
 <proof>

lemma *i-Exec-Comp-Stream-Acc-Output--eq-iAll-iSince-conv*:

$\llbracket 0 < k; s = (\text{output-fun} \circ \text{i-Exec-Comp-Stream trans-fun} (\text{input} \odot_i k) c) \rrbracket \implies$
 $((\text{i-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c) t = m) =$
 $((m = \varepsilon \longrightarrow (\Box \text{t1 } [t * k \dots, k - \text{Suc } 0]. s \text{ t1} = \varepsilon)) \wedge$
 $((m \neq \varepsilon \longrightarrow (s \text{ t2} = \varepsilon. \text{t2 } \mathcal{S} \text{ t1 } [t * k \dots, k - \text{Suc } 0]. s \text{ t1} = m))))$
 <proof>

lemma *i-Exec-Comp-Stream-Acc-Output--eq-iAll-iSince-conv2*:

$\llbracket 0 < k; s = (\text{output-fun} \circ \text{i-Exec-Comp-Stream trans-fun} (\text{input} \odot_i k) c) \rrbracket \implies$
 $((\text{i-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c) t = m) =$
 $((m = \varepsilon \longrightarrow (\Box \text{t1 } [\dots k - \text{Suc } 0] \oplus (t * k). s \text{ t1} = \varepsilon)) \wedge$
 $((m \neq \varepsilon \longrightarrow (s \text{ t2} = \varepsilon. \text{t2 } \mathcal{S} \text{ t1 } [\dots k - \text{Suc } 0] \oplus (t * k). s \text{ t1} = m))))$
 <proof>

5.3 *i-Exec-Comp-Stream-Acc-Output* and temporal operators with unbounded intervals and start/finish events.

lemma *i-Exec-Comp-Stream-Acc-Output--eq-NoMsg-iAll-start-event-conv*:

$\llbracket 0 < k; \bigwedge t. \text{event } t = (t \bmod k = 0); t0 = t * k;$
 $s = (\text{output-fun} \circ \text{i-Exec-Comp-Stream trans-fun} (\text{input} \odot_i k) c) \rrbracket \implies$
 $((\text{i-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c) t = \varepsilon) =$
 $(s \text{ t0} = \varepsilon \wedge (\bigcirc \text{t}' \text{t0 } [0 \dots]. (s \text{ t1} = \varepsilon. \text{t1 } \mathcal{U} \text{t2 } [0 \dots] \oplus \text{t}'. \text{event } \text{t2})))$
 <proof>

lemma *i-Exec-Comp-Stream-Acc-Output--Init--eq-NoMsg-iAll-start-event-conv*:

$\llbracket 0 < k; \bigwedge t. \text{event } t = ((t + k - \text{Suc } 0) \bmod k = 0); t0 = \text{Suc } (t * k);$
 $s = (\text{output-fun} \circ \text{i-Exec-Comp-Stream-Init trans-fun} (\text{input} \odot_i k) c) \rrbracket \implies$
 $((\text{i-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c) t = \varepsilon) =$
 $(s \text{ t0} = \varepsilon \wedge (\bigcirc \text{t}' \text{t0 } [0 \dots]. (s \text{ t1} = \varepsilon. \text{t1 } \mathcal{U} \text{t2 } [0 \dots] \oplus \text{t}'. \text{event } \text{t2})))$
 <proof>

lemma *i-Exec-Comp-Stream-Acc-Output--Init--eq-NoMsg-iAll-start-event2-conv*:

$\llbracket \text{Suc } 0 < k; \bigwedge t. \text{event } t = (t \bmod k = \text{Suc } 0); t0 = \text{Suc } (t * k);$
 $s = (\text{output-fun} \circ \text{i-Exec-Comp-Stream-Init trans-fun} (\text{input} \odot_i k) c) \rrbracket \implies$
 $((\text{i-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c) t = \varepsilon) =$

$(s \ t0 = \varepsilon \wedge (\bigcirc \ t' \ t0 \ [0\dots]). (s \ t1 = \varepsilon. \ t1 \ \mathcal{U} \ t2 \ [0\dots] \oplus \ t'. \ \text{event } t2)))$
 <proof>

lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-iUntil-start-event-conv:*

$\llbracket 0 < k; m \neq \varepsilon; \bigwedge t. \text{event } t = (t \bmod k = 0); t0 = t * k;$
 $s = (\text{output-fun} \circ \text{i-Exec-Comp-Stream trans-fun } (\text{input} \odot_i k) \ c) \rrbracket \implies$
 $((\text{i-Exec-Comp-Stream-Acc-Output } k \ \text{output-fun trans-fun input } c) \ t = m) = ($
 $(s \ t0 = m \wedge (\bigcirc \ t' \ t0 \ [0\dots]). (s \ t1 = \varepsilon. \ t1 \ \mathcal{U} \ t2 \ ([0\dots] \oplus \ t'). \ \text{event } t2))) \vee$
 $(\bigcirc \ t' \ t0 \ [0\dots]). (\neg \text{event } t1. \ t1 \ \mathcal{U} \ t2 \ ([0\dots] \oplus \ t'). ($
 $s \ t2 = m \wedge \neg \text{event } t2 \wedge (\bigcirc \ t'' \ t2 \ [0\dots].$
 $(s \ t3 = \varepsilon. \ t3 \ \mathcal{U} \ t4 \ ([0\dots] \oplus \ t''). \ \text{event } t4))))))$
 <proof>

lemma *i-Exec-Comp-Stream-Acc-Output--Init--eq-Msg-iUntil-start-event-conv:*

$\llbracket 0 < k; m \neq \varepsilon; \bigwedge t. \text{event } t = ((t + k - \text{Suc } 0) \bmod k = 0); t0 = \text{Suc } (t * k);$
 $s = (\text{output-fun} \circ \text{i-Exec-Comp-Stream-Init trans-fun } (\text{input} \odot_i k) \ c) \rrbracket \implies$
 $((\text{i-Exec-Comp-Stream-Acc-Output } k \ \text{output-fun trans-fun input } c) \ t = m) = ($
 $(s \ t0 = m \wedge (\bigcirc \ t' \ t0 \ [0\dots]). (s \ t1 = \varepsilon. \ t1 \ \mathcal{U} \ t2 \ ([0\dots] \oplus \ t'). \ \text{event } t2))) \vee$
 $(\bigcirc \ t' \ t0 \ [0\dots]). (\neg \text{event } t1. \ t1 \ \mathcal{U} \ t2 \ ([0\dots] \oplus \ t'). ($
 $s \ t2 = m \wedge \neg \text{event } t2 \wedge (\bigcirc \ t'' \ t2 \ [0\dots].$
 $(s \ t3 = \varepsilon. \ t3 \ \mathcal{U} \ t4 \ ([0\dots] \oplus \ t''). \ \text{event } t4))))))$
 <proof>

lemma *i-Exec-Comp-Stream-Acc-Output--Init--eq-Msg-iUntil-start-event2-conv:*

$\llbracket \text{Suc } 0 < k; m \neq \varepsilon; \bigwedge t. \text{event } t = (t \bmod k = \text{Suc } 0); t0 = \text{Suc } (t * k);$
 $s = (\text{output-fun} \circ \text{i-Exec-Comp-Stream-Init trans-fun } (\text{input} \odot_i k) \ c) \rrbracket \implies$
 $((\text{i-Exec-Comp-Stream-Acc-Output } k \ \text{output-fun trans-fun input } c) \ t = m) = ($
 $(s \ t0 = m \wedge (\bigcirc \ t' \ t0 \ [0\dots]). (s \ t1 = \varepsilon. \ t1 \ \mathcal{U} \ t2 \ ([0\dots] \oplus \ t'). \ \text{event } t2))) \vee$
 $(\bigcirc \ t' \ t0 \ [0\dots]). (\neg \text{event } t1. \ t1 \ \mathcal{U} \ t2 \ ([0\dots] \oplus \ t'). ($
 $s \ t2 = m \wedge \neg \text{event } t2 \wedge (\bigcirc \ t'' \ t2 \ [0\dots].$
 $(s \ t3 = \varepsilon. \ t3 \ \mathcal{U} \ t4 \ ([0\dots] \oplus \ t''). \ \text{event } t4))))))$
 <proof>

lemma *i-Exec-Comp-Stream-Acc-Output--eq-NoMsg-iAll-finish-event-conv:*

$\llbracket \text{Suc } 0 < k; \bigwedge t. \text{event } t = (t \bmod k = k - \text{Suc } 0); t0 = t * k;$
 $s = (\text{output-fun} \circ \text{i-Exec-Comp-Stream trans-fun } (\text{input} \odot_i k) \ c) \rrbracket \implies$
 $((\text{i-Exec-Comp-Stream-Acc-Output } k \ \text{output-fun trans-fun input } c) \ t = \varepsilon) =$
 $(s \ t0 = \varepsilon \wedge (\bigcirc \ t' \ t0 \ [0\dots]). (s \ t1 = \varepsilon. \ t1 \ \mathcal{U} \ t2 \ [0\dots] \oplus \ t'. \ \text{event } t2 \wedge s \ t2 = \varepsilon))$
 <proof>

lemma *i-Exec-Comp-Stream-Acc-Output--Init--eq-NoMsg-iAll-finish-event-conv:*

$\llbracket \text{Suc } 0 < k; \bigwedge t. \text{event } t = (t \bmod k = 0); t0 = \text{Suc } (t * k);$
 $s = (\text{output-fun} \circ \text{i-Exec-Comp-Stream-Init trans-fun } (\text{input} \odot_i k) \ c) \rrbracket \implies$
 $((\text{i-Exec-Comp-Stream-Acc-Output } k \ \text{output-fun trans-fun input } c) \ t = \varepsilon) =$
 $(s \ t0 = \varepsilon \wedge (\bigcirc \ t' \ t0 \ [0\dots]). (s \ t1 = \varepsilon. \ t1 \ \mathcal{U} \ t2 \ [0\dots] \oplus \ t'. \ \text{event } t2 \wedge s \ t2 = \varepsilon))$
 <proof>

lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-iUntil-finish-event-conv:*

$\llbracket 0 < k; m \neq \varepsilon; \bigwedge t. \text{event } t = (t \bmod k = k - \text{Suc } 0); t0 = t * k;$

$$\begin{aligned}
& s = (\text{output-fun} \circ i\text{-Exec-Comp-Stream trans-fun} (\text{input} \odot_i k) c) \Longrightarrow \\
& ((i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c) t = m) = \\
& ((\neg \text{event } t1. t1 \mathcal{U} t2 ([0\dots] \oplus t0). \text{event } t2 \wedge s t2 = m) \vee \\
& (\neg \text{event } t1. t1 \mathcal{U} t2 ([0\dots] \oplus t0). (\neg \text{event } t2 \wedge s t2 = m \wedge (\\
& \quad \circ t' t2 [0\dots]. (s t3 = \varepsilon. t3 \mathcal{U} t4 ([0\dots] \oplus t'). \text{event } t4 \wedge s t4 = \varepsilon)))))) \\
& \langle \text{proof} \rangle
\end{aligned}$$

lemma *i-Exec-Comp-Stream-Acc-Output--Init--eq-Msg-iUntil-finish-event-conv:*

$$\begin{aligned}
& \llbracket \text{Suc } 0 < k; m \neq \varepsilon; \bigwedge t. \text{event } t = (t \bmod k = 0); t0 = \text{Suc } (t * k); \\
& s = (\text{output-fun} \circ i\text{-Exec-Comp-Stream-Init trans-fun} (\text{input} \odot_i k) c) \Longrightarrow \\
& ((i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c) t = m) = \\
& ((\neg \text{event } t1. t1 \mathcal{U} t2 ([0\dots] \oplus t0). \text{event } t2 \wedge s t2 = m) \vee \\
& (\neg \text{event } t1. t1 \mathcal{U} t2 ([0\dots] \oplus t0). (\neg \text{event } t2 \wedge s t2 = m \wedge (\\
& \quad \circ t' t2 [0\dots]. (s t3 = \varepsilon. t3 \mathcal{U} t4 ([0\dots] \oplus t'). \text{event } t4 \wedge s t4 = \varepsilon)))))) \\
& \langle \text{proof} \rangle
\end{aligned}$$

5.4 *i-Exec-Comp-Stream-Acc-Output* and temporal operators with idle states.

lemma *i-Exec-Comp-Stream-Acc-Output--eq-NoMsg-State-Idle-conv:*

$$\begin{aligned}
& \llbracket 0 < k; \\
& \quad \text{State-Idle localState output-fun trans-fun} (\\
& \quad \quad i\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun input } c t); \\
& \quad t0 = t * k; \\
& s = i\text{-Exec-Comp-Stream trans-fun} (\text{input} \odot_i k) c \Longrightarrow \\
& (i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c t = \varepsilon) = \\
& (\text{output-fun } (s t1) = \varepsilon. t1 \mathcal{U} t2 ([0\dots] \oplus t0). (\\
& \quad \text{output-fun } (s t2) = \varepsilon \wedge \text{State-Idle localState output-fun trans-fun} (\text{localState } (s \\
& \quad t2)))) \\
& \langle \text{proof} \rangle
\end{aligned}$$

lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-with-State-Idle-imp:*

$$\begin{aligned}
& \llbracket 0 < k; \\
& \quad s = i\text{-Exec-Comp-Stream trans-fun} (\text{input} \odot_i k) c; \\
& \quad t0 = t * k; \\
& \quad t1 \in [0\dots, k - \text{Suc } 0] \oplus t0; \\
& \quad \text{State-Idle localState output-fun trans-fun} (\text{localState } (s t1)); \\
& \quad \text{output-fun } (s t1) \neq \varepsilon \rrbracket \Longrightarrow \\
& i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c t = \text{output-fun } (s \\
& t1) \\
& \langle \text{proof} \rangle
\end{aligned}$$

lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-with-State-Idle-conv2:*

$$\begin{aligned}
& \llbracket 0 < k; \\
& \quad \text{State-Idle localState output-fun trans-fun} (\\
& \quad \quad i\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun input } c t); \\
& \quad m \neq \varepsilon; \\
& \quad t0 = t * k; \\
& s = i\text{-Exec-Comp-Stream trans-fun} (\text{input} \odot_i k) c;
\end{aligned}$$

$t1 \in [0.., k - \text{Suc } 0] \oplus t0;$
 $\text{State-Idle localState output-fun trans-fun (localState (s t1));}$
 $\text{output-fun (s t1) } \neq \varepsilon \text{] } \implies$
 $(i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c \text{ } t = m) =$
 $(\diamond t1 [0.., k - \text{Suc } 0] \oplus t0. ($
 $\text{output-fun (s t1) = } m \wedge \text{State-Idle localState output-fun trans-fun (localState}$
 $\text{(s t1))))))$
 $\langle \text{proof} \rangle$

Here the property to be checked uses only unbounded intervals suitable for LTL.

lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-with-State-Idle-conv:*

$\llbracket 0 < k;$
 $\text{State-Idle localState output-fun trans-fun (}$
 $\text{ } i\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun input } c \text{ } t);$
 $m \neq \varepsilon;$
 $t0 = t * k;$
 $s = i\text{-Exec-Comp-Stream trans-fun (input } \odot_i k) c;$
 $t1 \in [0.., k - \text{Suc } 0] \oplus t0;$
 $\text{State-Idle localState output-fun trans-fun (localState (s t1));}$
 $\text{output-fun (s t1) } \neq \varepsilon \text{] } \implies$
 $(i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c \text{ } t = m) =$
 $((\neg \text{State-Idle localState output-fun trans-fun (localState (s t2))). t2 \mathcal{U} t1 [0..])$
 $\oplus t0. ($
 $\text{output-fun (s t1) = } m \wedge \text{State-Idle localState output-fun trans-fun (localState}$
 $\text{(s t1))))))$
 $\langle \text{proof} \rangle$

lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-before-State-Idle-imp2:*

$\llbracket \text{Suc } 0 < k;$
 $\text{State-Idle localState output-fun trans-fun (}$
 $\text{ } i\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun input } c \text{ } t);$
 $m \neq \varepsilon;$
 $t0 = t * k;$
 $s = i\text{-Exec-Comp-Stream trans-fun (input } \odot_i k) c;$
 $t1 \in [0.., k - \text{Suc } 0] \oplus t0;$
 $\text{output-fun (s t1) = } m;$
 $\circ t2 t1 [0..].$
 $((\text{output-fun (s t3) = } \varepsilon. t3 \mathcal{U} t4 ([0..] \oplus t2).$
 $\text{output-fun (s t4) = } \varepsilon \wedge \text{State-Idle localState output-fun trans-fun (localState}$
 $\text{(s t4)))))) \llbracket \implies$
 $i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c \text{ } t = m$
 $\langle \text{proof} \rangle$

lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-before-State-Idle-conv2:*

$\llbracket \text{Suc } 0 < k;$
 $\text{State-Idle localState output-fun trans-fun (}$
 $\text{ } i\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun input } c \text{ } t);$
 $m \neq \varepsilon;$

$$\begin{aligned}
& t0 = t * k; \\
& s = i\text{-Exec-Comp-Stream trans-fun (input } \odot_i k) c; \\
& \square t1 [0\dots, k - \text{Suc } 0] \oplus t0. \neg (\\
& \quad \text{State-Idle localState output-fun trans-fun (localState (s t1)) } \wedge \\
& \quad \text{output-fun (s t1) } \neq \varepsilon) \mathbb{J} \implies \\
& (i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c \text{ } t = m) = \\
& (\diamond t1 [0\dots, k - \text{Suc } 0] \oplus t0. (\\
& \quad (\text{output-fun (s t1) } = m) \wedge \\
& \quad (\bigcirc t2 t1 [0\dots]. \\
& \quad \quad ((\text{output-fun (s t3) } = \varepsilon. t3 \mathcal{U} t4 ([0\dots] \oplus t2). \\
& \quad \quad (\text{output-fun (s t4) } = \varepsilon \wedge \text{State-Idle localState output-fun trans-fun (localState} \\
& \quad \quad (s t4))))))))) \\
& \langle \text{proof} \rangle
\end{aligned}$$

Here the property to be checked uses only unbounded intervals suitable for LTL.

lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-before-State-Idle-imp:*

$$\begin{aligned}
& \mathbb{J} \text{Suc } 0 < k; \\
& \quad \text{State-Idle localState output-fun trans-fun (} \\
& \quad \quad i\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun input } c \text{ } t); \\
& \quad m \neq \varepsilon; \\
& \quad t0 = t * k; \\
& \quad s = i\text{-Exec-Comp-Stream trans-fun (input } \odot_i k) c; \\
& \quad (\neg \text{State-Idle localState output-fun trans-fun (localState (s t1))). t1 } \mathcal{U} t2 [0\dots] \\
& \oplus t0. (\\
& \quad (\text{output-fun (s t2) } = m) \wedge \\
& \quad (\bigcirc t3 t2 [0\dots]. \\
& \quad \quad ((\text{output-fun (s t4) } = \varepsilon. t4 \mathcal{U} t5 ([0\dots] \oplus t3). \\
& \quad \quad (\text{output-fun (s t5) } = \varepsilon \wedge \text{State-Idle localState output-fun trans-fun (localState} \\
& \quad \quad (s t5)))))) \mathbb{J} \implies \\
& \quad i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c \text{ } t = m \\
& \langle \text{proof} \rangle
\end{aligned}$$

lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-before-State-Idle-conv:*

$$\begin{aligned}
& \mathbb{J} \text{Suc } 0 < k; \\
& \quad \text{State-Idle localState output-fun trans-fun (} \\
& \quad \quad i\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun input } c \text{ } t); \\
& \quad m \neq \varepsilon; \\
& \quad t0 = t * k; \\
& \quad s = i\text{-Exec-Comp-Stream trans-fun (input } \odot_i k) c; \\
& \quad \square t1 [0\dots, k - \text{Suc } 0] \oplus t0. \neg (\\
& \quad \quad \text{State-Idle localState output-fun trans-fun (localState (s t1)) } \wedge \\
& \quad \quad \text{output-fun (s t1) } \neq \varepsilon) \mathbb{J} \implies \\
& (i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c \text{ } t = m) = \\
& ((\neg \text{State-Idle localState output-fun trans-fun (localState (s t1))). t1 } \mathcal{U} t2 [0\dots] \\
& \oplus t0. (\\
& \quad (\text{output-fun (s t2) } = m) \wedge \\
& \quad (\bigcirc t3 t2 [0\dots]. \\
& \quad \quad ((\text{output-fun (s t4) } = \varepsilon. t4 \mathcal{U} t5 ([0\dots] \oplus t3). \\
& \quad \quad (\text{output-fun (s t5) } = \varepsilon \wedge \text{State-Idle localState output-fun trans-fun (localState} \\
& \quad \quad (s t5))))))
\end{aligned}$$

(*s t5*)))))))))
 ⟨*proof*⟩

lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-conv2*:

[[*Suc 0 < k*;
 State-Idle localState output-fun trans-fun (
 i-Exec-Comp-Stream-Acc-LocalState *k* localState trans-fun input *c t*);
m ≠ ε;
t0 = *t* * *k*;
s = i-Exec-Comp-Stream trans-fun (input ⊙_{*i*} *k*) *c*]] ⇒
 (i-Exec-Comp-Stream-Acc-Output *k* output-fun trans-fun input *c t* = *m*) =
 (◇ *t1* [0... , *k* - *Suc 0*] ⊕ *t0*. (
 output-fun (*s t1*) = *m* ∧
 (State-Idle localState output-fun trans-fun (localState (*s t1*)) ∨
 (○ *t2 t1* [0...].
 ((output-fun (*s t3*) = ε. *t3* U *t4* ([0...] ⊕ *t2*).
 (output-fun (*s t4*) = ε ∧ State-Idle localState output-fun trans-fun (localState
 (*s t4*)))))))))
 ⟨*proof*⟩

lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-conv2'*:

[[*Suc 0 < k*;
 State-Idle localState output-fun trans-fun (
 i-Exec-Comp-Stream-Acc-LocalState *k* localState trans-fun input *c t*);
m ≠ ε;
t0 = *t* * *k*;
s = i-Exec-Comp-Stream trans-fun (input ⊙_{*i*} *k*) *c*]] ⇒
 (i-Exec-Comp-Stream-Acc-Output *k* output-fun trans-fun input *c t* = *m*) =
 ((◇ *t1* [0... , *k* - *Suc 0*] ⊕ *t0*. (
 output-fun (*s t1*) = *m* ∧ State-Idle localState output-fun trans-fun (localState
 (*s t1*))) ∨
 (◇ *t1* [0... , *k* - *Suc 0*] ⊕ *t0*. (
 (output-fun (*s t1*) = *m*) ∧
 (○ *t2 t1* [0...].
 ((output-fun (*s t3*) = ε. *t3* U *t4* ([0...] ⊕ *t2*).
 (output-fun (*s t4*) = ε ∧ State-Idle localState output-fun trans-fun (localState
 (*s t4*)))))))))
 ⟨*proof*⟩

lemma *i-Exec-Comp-Stream-Acc-Output--eq-iAll-iUntil-State-Idle-conv2*:

[[*Suc 0 < k*;
 State-Idle localState output-fun trans-fun (
 i-Exec-Comp-Stream-Acc-LocalState *k* localState trans-fun input *c t*);
t0 = *t* * *k*;
s = i-Exec-Comp-Stream trans-fun (input ⊙_{*i*} *k*) *c*]] ⇒
 (i-Exec-Comp-Stream-Acc-Output *k* output-fun trans-fun input *c t* = *m*) = (
 (*m* = ε →
 (output-fun (*s t1*) = ε. *t1* U *t2* ([0...] ⊕ *t0*). (
 output-fun (*s t2*) = ε ∧ State-Idle localState output-fun trans-fun (localState

$(s\ t2)))))) \wedge$
 $(m \neq \varepsilon \longrightarrow$
 $(\diamond t1\ [0\dots, k - \text{Suc } 0] \oplus t0. ($
 $\text{output-fun } (s\ t1) = m \wedge$
 $(\text{State-Idle } \text{localState } \text{output-fun } \text{trans-fun } (\text{localState } (s\ t1))) \vee$
 $(\circ t2\ t1\ [0\dots].$
 $((\text{output-fun } (s\ t3) = \varepsilon. t3\ \mathcal{U}\ t4\ ([0\dots] \oplus t2).$
 $(\text{output-fun } (s\ t4) = \varepsilon \wedge \text{State-Idle } \text{localState } \text{output-fun } \text{trans-fun } (\text{localState}$
 $(s\ t4))))))))))$
 $\langle \text{proof} \rangle$

lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-conv'*:

$\llbracket \text{Suc } 0 < k;$
 $\text{State-Idle } \text{localState } \text{output-fun } \text{trans-fun } ($
 $\text{i-Exec-Comp-Stream-Acc-LocalState } k\ \text{localState } \text{trans-fun } \text{input } c\ t);$
 $m \neq \varepsilon;$
 $t0 = t * k;$
 $s = \text{i-Exec-Comp-Stream } \text{trans-fun } (\text{input } \odot_i\ k)\ c \rrbracket \implies$
 $(\text{i-Exec-Comp-Stream-Acc-Output } k\ \text{output-fun } \text{trans-fun } \text{input } c\ t = m) =$
 $((\neg \text{State-Idle } \text{localState } \text{output-fun } \text{trans-fun } (\text{localState } (s\ t2))). t2\ \mathcal{U}\ t1\ [0\dots]$
 $\oplus t0.$
 $(\text{output-fun } (s\ t1) = m \wedge \text{State-Idle } \text{localState } \text{output-fun } \text{trans-fun } (\text{localState}$
 $(s\ t1)))) \vee$
 $((\neg \text{State-Idle } \text{localState } \text{output-fun } \text{trans-fun } (\text{localState } (s\ t2))). t2\ \mathcal{U}\ t1\ [0\dots]$
 $\oplus t0.$
 $(\text{output-fun } (s\ t1) = m \wedge$
 $(\circ t3\ t1\ [0\dots].$
 $((\text{output-fun } (s\ t4) = \varepsilon. t4\ \mathcal{U}\ t5\ ([0\dots] \oplus t3).$
 $(\text{output-fun } (s\ t5) = \varepsilon \wedge \text{State-Idle } \text{localState } \text{output-fun } \text{trans-fun } (\text{localState}$
 $(s\ t5))))))))))$
 $\langle \text{proof} \rangle$

lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-conv*:

$\llbracket \text{Suc } 0 < k;$
 $\text{State-Idle } \text{localState } \text{output-fun } \text{trans-fun } ($
 $\text{i-Exec-Comp-Stream-Acc-LocalState } k\ \text{localState } \text{trans-fun } \text{input } c\ t);$
 $m \neq \varepsilon;$
 $t0 = t * k;$
 $s = \text{i-Exec-Comp-Stream } \text{trans-fun } (\text{input } \odot_i\ k)\ c \rrbracket \implies$
 $(\text{i-Exec-Comp-Stream-Acc-Output } k\ \text{output-fun } \text{trans-fun } \text{input } c\ t = m) =$
 $((\neg \text{State-Idle } \text{localState } \text{output-fun } \text{trans-fun } (\text{localState } (s\ t2))). t2\ \mathcal{U}\ t1\ [0\dots]$
 $\oplus t0.$
 $(\text{output-fun } (s\ t1) = m \wedge$
 $(\text{State-Idle } \text{localState } \text{output-fun } \text{trans-fun } (\text{localState } (s\ t1))) \vee$
 $(\circ t3\ t1\ [0\dots].$
 $((\text{output-fun } (s\ t4) = \varepsilon. t4\ \mathcal{U}\ t5\ ([0\dots] \oplus t3).$
 $(\text{output-fun } (s\ t5) = \varepsilon \wedge \text{State-Idle } \text{localState } \text{output-fun } \text{trans-fun } (\text{localState}$
 $(s\ t5))))))))))$
 $\langle \text{proof} \rangle$

lemma *i-Exec-Comp-Stream-Acc-Output--eq-iUntil-State-Idle-conv*:

$$\begin{aligned}
& \llbracket \text{Suc } 0 < k; \\
& \text{State-Idle localState output-fun trans-fun (} \\
& \quad \text{i-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun input } c \text{ t);} \\
& \text{t0} = t * k; \\
& s = \text{i-Exec-Comp-Stream trans-fun (input } \odot_i k \text{) } c \rrbracket \implies \\
& (\text{i-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c \text{ t} = m) = (\\
& (m = \varepsilon \longrightarrow \\
& \quad (\text{output-fun (s t1)} = \varepsilon. \text{t1 } \mathcal{U} \text{ t2 ([0..]} \oplus \text{t0}). (\\
& \quad \quad \text{output-fun (s t2)} = \varepsilon \wedge \text{State-Idle localState output-fun trans-fun (localState} \\
& (s \text{ t2)))))) \wedge \\
& (m \neq \varepsilon \longrightarrow \\
& \quad (((\neg \text{State-Idle localState output-fun trans-fun (localState (s t2))). t2 } \mathcal{U} \text{ t1 [0..]} \\
& \oplus \text{t0}. \\
& \quad (\text{output-fun (s t1)} = m \wedge \\
& \quad \quad (\text{State-Idle localState output-fun trans-fun (localState (s t1)) } \vee \\
& \quad \quad (\odot \text{t3 t1 [0..]}. \\
& \quad \quad \quad ((\text{output-fun (s t4)} = \varepsilon. \text{t4 } \mathcal{U} \text{ t5 ([0..]} \oplus \text{t3}). \\
& \quad \quad \quad (\text{output-fun (s t5)} = \varepsilon \wedge \text{State-Idle localState output-fun trans-fun} \\
& (\text{localState (s t5)))))))))) \\
& \langle \text{proof} \rangle
\end{aligned}$$

Sufficient conditions for output messages.

corollary *i-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-iEx-imp1*:

$$\begin{aligned}
& \llbracket \text{Suc } 0 < k; \\
& \text{State-Idle localState output-fun trans-fun (} \\
& \quad \text{i-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun input } c \text{ t);} \\
& m \neq \varepsilon; \\
& \text{t0} = t * k; \\
& s = \text{i-Exec-Comp-Stream trans-fun (input } \odot_i k \text{) } c; \\
& (\diamond \text{t1 [0.., } k - \text{Suc } 0] \oplus \text{t0}. (\\
& \quad \text{output-fun (s t1)} = m \wedge \text{State-Idle localState output-fun trans-fun (localState} \\
& (s \text{ t1)))) \rrbracket \implies \\
& \text{i-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c \text{ t} = m \\
& \langle \text{proof} \rangle
\end{aligned}$$

corollary *i-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-iEx-imp2*:

$$\begin{aligned}
& \llbracket \text{Suc } 0 < k; \\
& \text{State-Idle localState output-fun trans-fun (} \\
& \quad \text{i-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun input } c \text{ t);} \\
& m \neq \varepsilon; \\
& \text{t0} = t * k; \\
& s = \text{i-Exec-Comp-Stream trans-fun (input } \odot_i k \text{) } c; \\
& \diamond \text{t1 [0.., } k - \text{Suc } 0] \oplus \text{t0}. (\\
& \quad ((\text{output-fun (s t1)} = m) \wedge \\
& \quad (\odot \text{t2 t1 [0..]}. \\
& \quad \quad ((\text{output-fun (s t3)} = \varepsilon. \text{t3 } \mathcal{U} \text{ t4 ([0..]} \oplus \text{t2}). \\
& \quad \quad (\text{output-fun (s t4)} = \varepsilon \wedge \text{State-Idle localState output-fun trans-fun} \\
& \quad \quad \quad \text{output-fun trans-fun} \\
& \quad \quad \quad \text{output-fun trans-fun}
\end{aligned}$$

$(localState (s t_4)))))) \]] \implies$
i-Exec-Comp-Stream-Acc-Output k *output-fun* *trans-fun* *input* c $t = m$
 ⟨proof⟩

lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-iUntil-imp1*:

\llbracket *Suc* $0 < k$;
State-Idle *localState* *output-fun* *trans-fun* (
 i-Exec-Comp-Stream-Acc-LocalState k *localState* *trans-fun* *input* c t);
 $m \neq \varepsilon$;
 $t0 = t * k$;
 $s = i-Exec-Comp-Stream$ *trans-fun* (*input* $\odot_i k$) c ;
 $(\neg$ *State-Idle* *localState* *output-fun* *trans-fun* (*localState* (s $t2$))). $t2 \mathcal{U} t1$ $[0..]$
 \oplus $t0$.
 (*output-fun* (s $t1$) = $m \wedge$ *State-Idle* *localState* *output-fun* *trans-fun* (*localState*
 (s $t1$))) \llbracket \implies
i-Exec-Comp-Stream-Acc-Output k *output-fun* *trans-fun* *input* c $t = m$
 ⟨proof⟩

lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-iUntil-imp2*:

\llbracket *Suc* $0 < k$;
State-Idle *localState* *output-fun* *trans-fun* (
 i-Exec-Comp-Stream-Acc-LocalState k *localState* *trans-fun* *input* c t);
 $m \neq \varepsilon$;
 $t0 = t * k$;
 $s = i-Exec-Comp-Stream$ *trans-fun* (*input* $\odot_i k$) c ;
 $(\neg$ *State-Idle* *localState* *output-fun* *trans-fun* (*localState* (s $t2$))). $t2 \mathcal{U} t1$ $[0..]$
 \oplus $t0$.
 (*output-fun* (s $t1$) = $m \wedge$
 $(\circ$ $t3$ $t1$ $[0..]$).
 (*output-fun* (s $t4$) = ε . $t4 \mathcal{U} t5$ ($[0..] \oplus t3$).
 (*output-fun* (s $t5$) = $\varepsilon \wedge$ *State-Idle* *localState* *output-fun* *trans-fun* (*localState*
 (s $t5$)))))) \llbracket \implies
i-Exec-Comp-Stream-Acc-Output k *output-fun* *trans-fun* *input* c $t = m$
 ⟨proof⟩

List of selected lemmas about output of accelerated components.

thm *i-Exec-Comp-Stream-Acc-Output--eq-NoMsg-iAll-conv*

thm *i-Exec-Comp-Stream-Acc-Output--eq-Msg-iEx-iAll-cut-greater-conv*

thm *i-Exec-Comp-Stream-Acc-Output--eq-Msg-iSince-conv*

thm *i-Exec-Comp-Stream-Acc-Output--eq-iAll-iSince-conv*

thm *i-Exec-Comp-Stream-Acc-Output--eq-NoMsg-State-Idle-conv*

thm *i-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-conv2*

thm *i-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-conv*

thm *i-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-conv2'*

thm *i-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-conv'*

thm *i-Exec-Comp-Stream-Acc-Output--eq-iAll-iUntil-State-Idle-conv2*

thm *i-Exec-Comp-Stream-Acc-Output--eq-iUntil-State-Idle-conv*

```
thm i-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-iEx-imp1  
thm i-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-iEx-imp2  
thm i-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-iUntil-imp1  
thm i-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-iUntil-imp2  
  
end
```