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

David Trachtenherz

September 13, 2023

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

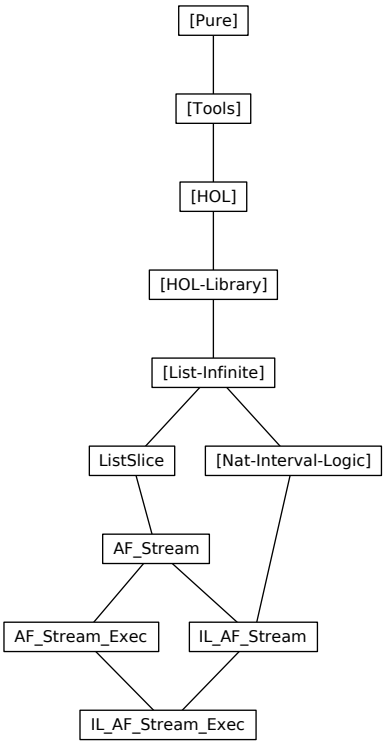
2 AutoFocus message streams **7**
2.1 Basic definitions 7
2.1.1 Time-synchronous streams 7
2.1.2 Time abstraction 10
2.2 Expanding and compressing lists and streams 11
2.2.1 Expanding message streams 11
2.2.2 Aggregating lists 16
2.2.3 Compressing message streams 19
2.2.4 Holding last messages in everly 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 *IL-Interval* 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 *f-Exec-Comp-Stream* and *f-Exec-Comp-Stream-Init* 99
5.2 *i-Exec-Comp-Stream-Acc-Output* and temporal operators with bounded intervals. 100
5.3 *i-Exec-Comp-Stream-Acc-Output* and temporal operators with unbounded intervals and start/finish events. 101
5.4 *i-Exec-Comp-Stream-Acc-Output* and temporal operators with idle states. 103


[Pure]
[Tools]
[HOL]
[HOL-Library]
[List-Infinite]
ListSlice
[Nat-Interval-Logic]
AF_Stream
AF_Stream_Exec
IL_AF_Stream
IL_AF_Stream_Exec

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 ⇒ nat ⇒ 'a list ilist*
  where *ilist-slice f k ≡ λx. map f [x * k..<Suc x * k]*

primrec *list-slice-aux* :: *'a list ⇒ nat ⇒ nat ⇒ 'a list list*
where
  list-slice-aux xs k 0 = []
| *list-slice-aux xs k (Suc n) = take k xs # list-slice-aux (xs ↑ k) k n*

definition *list-slice* :: *'a list ⇒ nat ⇒ 'a list list*
  where *list-slice xs k ≡ list-slice-aux xs k (length xs div k)*

definition *list-slice2* :: *'a list ⇒ nat ⇒ 'a list list*
  where *list-slice2 xs k ≡*
    *list-slice xs k @ (if length xs mod k = 0 then [] else [xs ↑ (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 ⇒ 'a ilist*
  where *ilist-unslice f ≡ λn. f (n div length (f 0)) ! (n mod length (f 0))*

lemma *list-slice-aux-length*: *⋀xs. length (list-slice-aux xs k n) = n*
⟨*proof*⟩

lemma *list-slice-aux-nth*:
*⋀m xs. m < n ⟹ (list-slice-aux xs k n) ! m = (xs ↑ (m * k) ↓ 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 (λx. [x]) xs*
⟨*proof*⟩

lemma *list-slice-less*: *length xs < k ⟹ list-slice xs k = []*
⟨*proof*⟩

lemma *list-slice-Nil*: *list-slice* [] k = []
⟨*proof*⟩

lemma *list-slice-nth*:
m < *length xs div* k ⟹ *list-slice xs* k ! m = xs ↑ (m * k) ↓ k
⟨*proof*⟩

lemma *list-slice-nth-length*:
m < *length xs div* k ⟹ *length* ((*list-slice xs* k) ! m) = k
⟨*proof*⟩

lemma *list-slice-nth-eq-sublist-list*:
m < *length xs div* k ⟹ *list-slice xs* k ! m = *sublist-list xs* [m * k..<m * k + k]
⟨*proof*⟩

lemma *list-slice-nth-nth*:
⟦ m < *length xs div* k; n < k ⟧ ⟹
(*list-slice xs* k) ! m ! n = xs ! (m * k + n)
⟨*proof*⟩

lemma *list-slice-nth-nth-rev*:
n < *length xs div* k * k ⟹
(*list-slice xs* k) ! (n *div* k) ! (n *mod* k) = xs ! n
⟨*proof*⟩

lemma *list-slice-eq-list-slice-take*:
list-slice (xs ↓ (*length xs div* k * k)) k = *list-slice xs* k
⟨*proof*⟩

lemma *list-slice-append-mult*:
⋀xs. *length xs* = m * k ⟹
list-slice (xs @ ys) k = *list-slice xs* k @ *list-slice ys* k
⟨*proof*⟩

lemma *list-slice-append-mod*:
length xs mod k = 0 ⟹
list-slice (xs @ ys) k = *list-slice xs* k @ *list-slice ys* k
⟨*proof*⟩

lemma *list-slice-div-eq-1*[*rule-format*]:
length xs div k = *Suc* 0 ⟹ *list-slice xs* k = [*take* k *xs*]
⟨*proof*⟩

lemma *list-slice-div-eq-Suc*[*rule-format*]:
length xs div k = *Suc* n ⟹
list-slice xs k = *list-slice* (xs ↓ (n * k)) k @ [xs ↑ (n * k) ↓ k]
⟨*proof*⟩

lemma *list-slice2-mod-0*:

length xs mod k = 0 ⟹ list-slice2 xs k = list-slice xs k
⟨*proof*⟩

lemma *list-slice2-mod-gr0*:
*0 < length xs mod k ⟹ list-slice2 xs k = list-slice xs k @ [xs ↑ (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 (λx. [x]) xs*
⟨*proof*⟩

lemma *list-slice2-le*:
length xs ≤ k ⟹ 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 ⟹ 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 ⟧ ⟹
*list-slice2 xs k ! m = xs ↑ (length xs div k * k)*
⟨*proof*⟩

lemma *list-slice2-nth*:
⟦ m < length xs div k ⟧ ⟹
*list-slice2 xs k ! m = xs ↑ (m * k) ↓ k*
⟨*proof*⟩

lemma *list-slice2-nth-length-eq1*:
m < length xs div k ⟹ length (list-slice2 xs k ! m) = k
⟨*proof*⟩

lemma *list-slice2-nth-length-eq2*:
⟦ length xs mod k > 0; m = length xs div k ⟧ ⟹
length (list-slice2 xs k ! m) = length xs mod k
⟨*proof*⟩

lemma *list-slice2-nth-nth-eq1*:
⟦ $m < length\ xs\ div\ k$; $n < k$ ⟧ ⟹
$(list\text{-}slice2\ xs\ k)\ !\ m\ !\ n = xs\ !\ (m * k + n)$
⟨*proof*⟩

lemma *list-slice2-nth-nth-eq2*:
⟦ $m = length\ xs\ div\ k$; $n < length\ xs\ mod\ k$ ⟧ ⟹
$(list\text{-}slice2\ xs\ k)\ !\ m\ !\ n = xs\ !\ (m * k + n)$
⟨*proof*⟩

lemma *list-slice2-nth-nth-rev*:
$n < length\ xs \Longrightarrow (list\text{-}slice2\ xs\ k)\ !\ (n\ div\ k)\ !\ (n\ mod\ k) = xs\ !\ n$
⟨*proof*⟩

lemma *list-slice2-append-mult*:
$length\ xs = m * k \Longrightarrow$
$list\text{-}slice2\ (xs\ @\ ys)\ k = list\text{-}slice2\ xs\ k\ @\ list\text{-}slice2\ ys\ k$
⟨*proof*⟩

lemma *list-slice2-append-mod*:
$length\ xs\ mod\ k = 0 \Longrightarrow$
$list\text{-}slice2\ (xs\ @\ ys)\ k = list\text{-}slice2\ xs\ k\ @\ list\text{-}slice2\ ys\ k$
⟨*proof*⟩

lemma *ilist-slice-nth*:
$(ilist\text{-}slice\ f\ k)\ m = map\ f\ [m * k..<Suc\ m * k]$
⟨*proof*⟩

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

lemma *ilist-slice-nth-nth*:
$n < k \Longrightarrow (ilist\text{-}slice\ f\ k)\ m\ !\ n = f\ (m * k + n)$
⟨*proof*⟩

lemma *ilist-slice-nth-nth-rev*:
$0 < k \Longrightarrow (ilist\text{-}slice\ f\ k)\ (n\ div\ k)\ !\ (n\ mod\ k) = f\ n$
⟨*proof*⟩

lemma *list-slice-concat*:
$concat\ (list\text{-}slice\ xs\ k) = xs \downarrow (length\ xs\ div\ k * k)$
(**is** *?P xs k*)
⟨*proof*⟩

lemma *list-slice-unslice-mult*:
$length\ xs = m * k \Longrightarrow concat\ (list\text{-}slice\ xs\ k) = xs$
⟨*proof*⟩

lemma *ilist-slice-unslice*: $0 < k \Longrightarrow$ *ilist-unslice* (*ilist-slice f k*) = *f*
⟨*proof*⟩

lemma *i-take-ilist-slice-eq-list-slice*:
$0 < k \Longrightarrow$ *ilist-slice f k* ⇓ *n* = *list-slice* (*f* ⇓ (*n* ∗ *k*)) *k*
⟨*proof*⟩

lemma *list-slice-i-take-eq-i-take-ilist-slice*:
list-slice (*f* ⇓ *n*) *k* = *ilist-slice f k* ⇓ (*n div k*)
⟨*proof*⟩

lemma *ilist-slice-i-append-mod*:
length xs mod k = 0 ⟹
ilist-slice (*xs* ⌢ *f*) *k* = *list-slice xs k* ⌢ *ilist-slice f k*
⟨*proof*⟩

corollary *ilist-slice-append-mult*:
length xs = *m* ∗ *k* ⟹
ilist-slice (*xs* ⌢ *f*) *k* = *list-slice xs k* ⌢ *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* ≠ *ε*) = (∃ *x*. *m* = *Msg x*)
⟨*proof*⟩

lemma *not-Msg-eq*: $(\forall\, x.\ m \neq \mathsf{Msg}\ x) = (m = \varepsilon)$
⟨*proof*⟩

primrec *the-af* :: *'a message-af* ⇒ *'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* ⇒ *bool*

overloading *is-NoMsg* ≡ *is-NoMsg* :: *'a message-af* ⇒ *bool*
begin

primrec *is-NoMsg* :: *'a message-af* ⇒ *bool*
where
  is-NoMsg ε = *True*
| *is-NoMsg* (**Msg** *x*) = *False*

end

overloading *is-NoMsg* ≡ *is-NoMsg* :: (*'a* × *'b*) ⇒ *bool*
begin

definition *is-NoMsg-tuple-def* :
  is-NoMsg (*p*::*'a* × *'b*) ≡ (*is-NoMsg* (*fst p*) ∧ *is-NoMsg* (*snd p*))

end

overloading *is-NoMsg* ≡ *is-NoMsg* :: *'a set* ⇒ *bool*
begin

definition *is-NoMsg-set-def* :
  is-NoMsg (*A*::*'a set*) ≡ (∀ *x*∈*A*. *is-NoMsg x*)

end

record *SomeRecordExample* =
  Field1 :: *nat message-af*
  Field2 :: *int message-af*
  Field3 :: *int message-af*
overloading *is-NoMsg* ≡ *is-NoMsg* :: *'a SomeRecordExample-scheme* ⇒ *bool*
begin

definition *is-NoMsg-SomeRecordExample-def* :
  is-NoMsg (*r*:: *'a SomeRecordExample-scheme*) ≡
    Field1 r = ε ∧ *Field2 r* = ε ∧ *Field3 r* = ε

end

definition *is-Msg* :: *'a* ⇒ *bool*
  where *is-Msg x* ≡ (¬ *is-NoMsg x*)

lemma *is-NoMsg-message-af-conv*: *is-NoMsg m* = (*case m of* ε ⇒ *True* | Msg *x* ⇒ *False*)
⟨*proof*⟩

lemma *is-NoMsg-message-af-conv2*: *is-NoMsg m* = (*m* = ε)
⟨*proof*⟩

lemma *is-Msg-message-af-conv*: *is-Msg m* = (*case m of* ε ⇒ *False* | Msg *x* ⇒ *True*)
⟨*proof*⟩

lemma *is-Msg-message-af-conv2*: *is-Msg m* = (*m* ≠ ε)
⟨*proof*⟩

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*: (¬ *is-NoMsg m*) = *is-Msg m*
⟨*proof*⟩

lemma *not-is-Msg*: (¬ *is-Msg m*) = *is-NoMsg m*
⟨*proof*⟩

lemma *is-NoMsg* (ε::(*nat message-af*))
⟨*proof*⟩

lemma *is-NoMsg* (ε::(*nat message-af*), ε::(*nat message-af*))
⟨*proof*⟩

lemma *is-NoMsg* (ε::(*nat message-af*), ε::(*nat message-af*), ε::(*nat message-af*))
⟨*proof*⟩

lemma *is-Msg* (ε::(*nat message-af*), Msg (*1*::*nat*), ε::(*nat message-af*))
⟨*proof*⟩

lemma *is-NoMsg* {ε::(*nat message-af*), ε}
⟨*proof*⟩

lemma *is-Msg* {*ε*::(*nat message-af*), *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* ⇒ *'a list*
where
untime [] = []
| *untime* (*x*#*xs*) =
(*if x* = *ε*
then (*untime xs*)
else (*the-af x*) # (*untime xs*))

lemma *untime-eq-filter*[*rule-format*]:
map (λ*x*. *Msg x*) (*untime s*) = *filter* (λ*x*. *x* ≠ *ε*) *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* (λ*x*. *the-af x*) (*filter* (λ*x*. *x* ≠ *ε*) *s*)
⟨*proof*⟩

definition *untime-length* :: *'a fstream-af* ⇒ *nat*
where *untime-length s* ≡ *length* (*untime s*)

primrec *untime-length-cnt* :: *'a fstream-af* ⇒ *nat*
where
untime-length-cnt [] = *0*
| *untime-length-cnt* (*x* # *xs*) =
(*if x* = *ε 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* ⇒ *nat*
where *untime-length-filter s* ≡ *length* (*filter* (λ*x*. *x* ≠ *ε*) *s*)

lemma *untime-length-filter-eq-untime-length*:
untime-length-filter s = *untime-length s*
⟨*proof*⟩

lemma *untime-empty-conv*: (*untime s* = []) = (∀ *n*<*length s*. *s* ! *n* = *ε*)
⟨*proof*⟩

lemma *untime-not-empty-conv*: (*untime s* ≠ []) = (∃ *n*<*length s*. *s* ! *n* ≠ *ε*)
⟨*proof*⟩

corollary *untime-empty-imp-NoMsg*[*rule-format*]:
⟦ *untime s* = []; *n* < *length s* ⟧ ⟹ *s* ! *n* = *ε*
⟨*proof*⟩

lemma *untime-nth-eq-filter*:
n < *untime-length s* ⟹
Msg (*untime s* ! *n*) = (*filter* (λ*x*. *x* ≠ *ε*) *s*) ! *n*
⟨*proof*⟩

corollary *untime-nth-eq-filter2*:
n < *untime-length s* ⟹
untime s ! *n* = *the-af* ((*filter* (λ*x*. *x* ≠ *ε*) *s*) ! *n*)
⟨*proof*⟩

lemma *untime-hd-eq-filter-hd*:
untime s ≠ [] ⟹
Msg (*hd* (*untime s*)) = *hd* (*filter* (λ*x*. *x* ≠ *ε*) *s*)
⟨*proof*⟩

corollary *untime-hd-eq-filter-hd2*:
untime s ≠ [] ⟹
hd (*untime s*) = *the-af* (*hd* (*filter* (λ*x*. *x* ≠ *ε*) *s*))
⟨*proof*⟩

lemma *untime-last-eq-filter-last*:
untime s ≠ [] ⟹
Msg (*last* (*untime s*)) = *last* (*filter* (λ*x*. *x* ≠ *ε*) *s*)
⟨*proof*⟩

corollary *untime-last-eq-filter-last2*:
untime s ≠ [] ⟹
last (*untime s*) = *the-af* (*last* (*filter* (λ*x*. *x* ≠ *ε*) *s*))
⟨*proof*⟩

2.2 Expanding and compressing lists and streams

2.2.1 Expanding message streams

primrec *f-expand* :: *'a fstream-af* ⇒ *nat* ⇒ *'a fstream-af* (**infixl** ⊙$_f$ *100*)
where

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

definition *i-expand* :: *'a istream-af* ⇒ *nat* ⇒ *'a istream-af* (**infixl** $\odot_i$ *100*)
where
i-expand ≡ λ*f k n*.
(*if k* = *0 then* ε *else*
if n mod k = *0 then f* (*n div k*) *else* ε)

primrec *f-expand-Suc* :: *'a fstream-af* ⇒ *nat* ⇒ *'a fstream-af* (**infixl** $\odot_{fSuc}$ *100*)
where
f-expand-Suc [] *k* = []
| *f-expand-Suc* (*x* # *xs*) *k* = *x* # ε^*k* @ (*f-expand-Suc xs k*)

definition *i-expand-Suc* :: *'a istream-af* ⇒ *nat* ⇒ *'a istream-af* (**infixl** $\odot_{iSuc}$ *100*)
where *i-expand-Suc* ≡ λ*f k n*. *if n mod* (*Suc k*) = *0 then f* (*n div* (*Suc k*)) *else* ε

notation
f-expand (**infixl** $\odot$ *100*) **and**
i-expand (**infixl** $\odot$ *100*)

lemma *length-f-expand-Suc*[*simp*]: *length* (*f-expand-Suc xs k*) = *length xs* ∗ *Suc k*
⟨*proof*⟩

lemma *i-expand-if*:
f $\odot_i$ *k* = (*if k* = *0 then* (λ*n*. ε) *else*
(λ*n*. *if n mod k* = *0 then f* (*n div k*) *else* ε))
⟨*proof*⟩

lemma *f-expand-one*: *0* < *k* ⟹ [*a*] $\odot_f$ *k* = *a* # ε^*k* − *Suc 0*
⟨*proof*⟩

lemma *f-expand-0*[*simp*]: *xs* $\odot_f$ *0* = []
⟨*proof*⟩
corollary *f-expand-0-is-zero-element*: *xs* $\odot_f$ *0* = *ys* $\odot_f$ *0*
⟨*proof*⟩
lemma *i-expand-0*[*simp*]: *f* $\odot_i$ *0* = (λ*n*. ε)
⟨*proof*⟩
corollary *i-expand-0-is-zero-element*: *f* $\odot_i$ *0* = *g* $\odot_i$ *0*
⟨*proof*⟩

lemma *f-expand-gr0-f-expand-Suc*: *0* < *k* ⟹ *xs* $\odot_f$ *k* = *f-expand-Suc xs* (*k* − *Suc 0*)
⟨*proof*⟩
lemma *i-expand-gr0-i-expand-Suc*: *0* < *k* ⟹ *f* $\odot_i$ *k* = *i-expand-Suc f* (*k* − *Suc 0*)
⟨*proof*⟩

lemma *i-expand-gr0*:
$0 < k \Longrightarrow f \odot_i k = (\lambda n.\ \mathit{if}\ n\ \mathit{mod}\ k = 0\ \mathit{then}\ f\ (n\ \mathit{div}\ k)\ \mathit{else}\ \varepsilon)$
⟨*proof*⟩
lemma *f-expand-1*[*simp*]: $xs \odot_f \mathit{Suc}\ 0 = xs$
⟨*proof*⟩
lemma *i-expand-1*[*simp*]: $f \odot_i \mathit{Suc}\ 0 = f$
⟨*proof*⟩

lemma *f-expand-length*[*simp*]: $\mathit{length}\ (xs \odot_f k) = \mathit{length}\ xs * k$
⟨*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*:
$0 < k \Longrightarrow (x \# xs) \odot_f k = x \# \varepsilon^{k - \mathit{Suc}\ 0} @ (xs \odot_f k)$
⟨*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*:
$0 < k \Longrightarrow (xs @ [x]) \odot_f k = xs \odot_f k @ x \# \mathit{replicate}\ (k - \mathit{Suc}\ 0)\ \varepsilon$
⟨*proof*⟩

lemma *f-expand-nth-mult*: $\bigwedge n.$
$[\![\ n < \mathit{length}\ xs;\ 0 < k\ ]\!] \Longrightarrow (xs \odot_f k)\ !\ (n * k) = xs\ !\ n$
⟨*proof*⟩

lemma *i-expand-nth-mult*: $0 < k \Longrightarrow (f \odot_i k)\ (n * k) = f\ n$
⟨*proof*⟩

lemma *f-expand-nth-if*: $\bigwedge n.$
$n < \mathit{length}\ xs * k \Longrightarrow$
$(xs \odot_f k)\ !\ n = (\mathit{if}\ n\ \mathit{mod}\ k = 0\ \mathit{then}\ xs\ !\ (n\ \mathit{div}\ k)\ \mathit{else}\ \varepsilon)$
⟨*proof*⟩

corollary *f-expand-nth-mod-eq-0*:
$[\![\ n < \mathit{length}\ xs * k;\ n\ \mathit{mod}\ k = 0\ ]\!] \Longrightarrow (xs \odot_f k)\ !\ n = xs\ !\ (n\ \mathit{div}\ k)$
⟨*proof*⟩

corollary *f-expand-nth-mod-neq-0*:
$[\![\ n < \mathit{length}\ xs * k;\ 0 < n\ \mathit{mod}\ k\ ]\!] \Longrightarrow (xs \odot_f k)\ !\ n = \varepsilon$
⟨*proof*⟩

lemma *f-expand-nth-0-upto-k-minus-1-if*:
$[\![\ t < \mathit{length}\ xs;\ n = t * k + i;\ i < k\ ]\!] \Longrightarrow$
$(xs \odot_f k)\ !\ n = (\mathit{if}\ i = 0\ \mathit{then}\ xs\ !\ t\ \mathit{else}\ \varepsilon)$

⟨*proof*⟩

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

lemma *f-expand-take-mod*:
$n \bmod k = 0 \Longrightarrow xs \odot_f k \downarrow n = xs \downarrow (n \operatorname{div} k) \odot_f k$
⟨*proof*⟩

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

lemma *f-expand-drop-mod*:
$n \bmod k = 0 \Longrightarrow xs \odot_f k \uparrow n = xs \uparrow (n \operatorname{div} k) \odot_f k$
⟨*proof*⟩

lemma *f-expand-take-mult-Suc*:
$[\![n < \mathit{length}\ xs;\ i < k]\!] \Longrightarrow$
$xs \odot_f k \downarrow (n * k + \mathit{Suc}\ i) = (xs \downarrow n) \odot_f k\ @\ (xs\ !\ n\ \#\ \varepsilon^{i})$
⟨*proof*⟩

lemma *f-expand-take-Suc*:
$n < \mathit{length}\ xs * k \Longrightarrow$
$xs \odot_f k \downarrow \mathit{Suc}\ n = (xs \downarrow (n \operatorname{div} k)) \odot_f k\ @\ (xs\ !\ (n \operatorname{div} k)\ \#\ \varepsilon^{n \bmod k})$
⟨*proof*⟩

lemma *i-expand-nth-if*:
$0 < k \Longrightarrow (f \odot_i k)\ n = (\mathit{if}\ n \bmod k = 0\ \mathit{then}\ f\ (n \operatorname{div} k)\ \mathit{else}\ \varepsilon)$
⟨*proof*⟩
corollary *i-expand-nth-mod-eq-0*:
$[\![0 < k;\ n \bmod k = 0]\!] \Longrightarrow (f \odot_i k)\ n = f\ (n \operatorname{div} k)$
⟨*proof*⟩
corollary *i-expand-nth-mod-neq-0*:
$0 < n \bmod k \Longrightarrow (f \odot_i k)\ n = \varepsilon$
⟨*proof*⟩

lemma *i-expand-nth-0-upto-k-minus-1-if*:
$[\![n = t * k + i;\ i < k]\!] \Longrightarrow$
$(f \odot_i k)\ n = (\mathit{if}\ i = 0\ \mathit{then}\ f\ t\ \mathit{else}\ \varepsilon)$
⟨*proof*⟩

lemma *i-expand-i-take-mult*: $f \odot_i k \Downarrow (n * k) = (f \Downarrow n) \odot_f k$
⟨*proof*⟩

lemma *i-expand-i-take-mod*:
$n \bmod k = 0 \Longrightarrow f \odot_i k \Downarrow n = f \Downarrow (n \operatorname{div} k) \odot_f k$
⟨*proof*⟩

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*:
$n \bmod k = 0 \Longrightarrow f \odot_i k \Uparrow n = f \Uparrow (n \operatorname{div} k) \odot_i k$
⟨*proof*⟩

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

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

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

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

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

lemma *i-expand-nth-interval-eq-replicate-NoMsg*:
$[\![\ 0 < k; t * k < t1; t1 \leq t2; t2 \leq t * k + k\ ]\!] \Longrightarrow$
$f \odot_i k \Downarrow t2 \uparrow t1 = \varepsilon^{t2 - t1}$
⟨*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*:
$0 < k \Longrightarrow (xs \odot_f k = ys \odot_f k) = (xs = ys)$
⟨*proof*⟩

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

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

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

2.2.2 Aggregating lists

definition *f-aggregate* :: *'a list* ⇒ *nat* ⇒ (*'a list* ⇒ *'a*) ⇒ *'a list*
where *f-aggregate s k ag* ≡ *map ag* (*list-slice s k*)

definition *i-aggregate* :: *'a ilist* ⇒ *nat* ⇒ (*'a list* ⇒ *'a*) ⇒ *'a ilist*
where *i-aggregate s k ag* ≡ λ*n*. *ag* (*s* ⇑ (*n* ∗ *k*) ⇓ *k*)

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

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

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

lemma *f-aggregate-length*[*simp*]: *length* (*f-aggregate xs k ag*) = *length xs div k*

⟨*proof*⟩

lemma *f-aggregate-empty-conv*:
$0 < k \Longrightarrow (f\text{-}aggregate\ xs\ k\ ag = []) = (length\ xs < k)$
⟨*proof*⟩

lemma *f-aggregate-one*:
$[\![\ 0 < k;\ length\ xs = k\ ]\!] \Longrightarrow f\text{-}aggregate\ xs\ k\ ag = [ag\ xs]$
⟨*proof*⟩

lemma *f-aggregate-Cons*:
$[\![\ 0 < k;\ length\ xs = k\ ]\!] \Longrightarrow$
$f\text{-}aggregate\ (xs\ @\ ys)\ k\ ag = ag\ xs\ \#\ (f\text{-}aggregate\ ys\ k\ ag)$
⟨*proof*⟩

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

lemma *f-aggregate-nth*:
$n < length\ xs\ div\ k \Longrightarrow$
$(f\text{-}aggregate\ xs\ k\ ag)\ !\ n = ag\ (xs \uparrow (n * k) \downarrow k)$
⟨*proof*⟩

lemma *f-aggregate-nth-eq-sublist-list*:
$n < length\ xs\ div\ k \Longrightarrow$
$(f\text{-}aggregate\ xs\ k\ ag)\ !\ n = ag\ (sublist\text{-}list\ xs\ [n * k..<n * k + k])$
⟨*proof*⟩

lemma *f-aggregate-take-nth*:
$\bigwedge xs\ m.\ [\![\ n < length\ xs\ div\ k;\ n < m\ div\ k\ ]\!] \Longrightarrow$
$f\text{-}aggregate\ (xs \downarrow m)\ k\ ag\ !\ n = f\text{-}aggregate\ xs\ k\ ag\ !\ n$
⟨*proof*⟩

lemma *f-aggregate-hd*:
$[\![\ 0 < k;\ k \leq length\ xs\ ]\!] \Longrightarrow$
$hd\ (f\text{-}aggregate\ xs\ k\ ag) = ag\ (xs \downarrow k)$
⟨*proof*⟩

lemma *f-aggregate-append-mod*:
$length\ xs\ mod\ k = 0 \Longrightarrow$
$f\text{-}aggregate\ (xs\ @\ ys)\ k\ ag =$
$f\text{-}aggregate\ xs\ k\ ag\ @\ f\text{-}aggregate\ ys\ k\ ag$
⟨*proof*⟩
lemma *f-aggregate-append-mult*:
$length\ xs = m * k \Longrightarrow$
$f\text{-}aggregate\ (xs\ @\ ys)\ k\ ag =$
$f\text{-}aggregate\ xs\ k\ ag\ @\ f\text{-}aggregate\ ys\ k\ ag$
⟨*proof*⟩

lemma *f-aggregate-snoc*:
⟦ *0 < k*; *length ys = k*; *length xs mod k = 0* ⟧ ⟹
f-aggregate (*xs* @ *ys*) *k ag* = *f-aggregate xs k ag* @ [*ag ys*]
⟨*proof*⟩

lemma *f-aggregate-take*:
f-aggregate (*xs* ↓ *n*) *k ag* = *f-aggregate xs k ag* ↓ (*n div k*)
⟨*proof*⟩

lemma *f-aggregate-take-mult*:
f-aggregate (*xs* ↓ (*n* ∗ *k*)) *k ag* = *f-aggregate xs k ag* ↓ *n*
⟨*proof*⟩

lemma *f-aggregate-drop-mult*:
f-aggregate (*xs* ↑ (*n* ∗ *k*)) *k ag* = *f-aggregate xs k ag* ↑ *n*
⟨*proof*⟩
lemma *f-aggregate-drop-mod*:
n mod k = 0 ⟹ *f-aggregate* (*xs* ↑ *n*) *k ag* = *f-aggregate xs k ag* ↑ (*n div k*)
⟨*proof*⟩

lemma *f-aggregate-assoc*:
(⋀*xs*. *length xs mod a = 0* ⟹ *ag* (*f-aggregate xs a ag*) = *ag xs*) ⟹
f-aggregate (*f-aggregate xs a ag*) *b ag* = *f-aggregate xs* (*a* ∗ *b*) *ag*
⟨*proof*⟩

lemma *f-aggregate-commute*:
⟦ ⋀*xs*. *length xs mod a = 0* ⟹ *ag* (*f-aggregate xs a ag*) = *ag xs*;
⋀*xs*. *length xs mod b = 0* ⟹ *ag* (*f-aggregate xs b ag*) = *ag xs* ⟧ ⟹
f-aggregate (*f-aggregate xs a ag*) *b ag* = *f-aggregate* (*f-aggregate xs b ag*) *a ag*
⟨*proof*⟩

lemma *i-aggregate-0*[*simp*]: *i-aggregate f 0 ag* = (λ*x*. *ag* [])
⟨*proof*⟩
lemma *i-aggregate-1*: (⋀*x*. *ag* [*x*] = *x*) ⟹ *i-aggregate f* (*Suc 0*) *ag* = *f*
⟨*proof*⟩
lemma *i-aggregate-nth*: *i-aggregate f k ag n* = *ag* (*f* ⇑ (*n* ∗ *k*) ⇓ *k*)
⟨*proof*⟩
lemma *i-aggregate-hd*: *i-aggregate f k ag 0* = *ag* (*f* ⇓ *k*)
⟨*proof*⟩

lemma *i-aggregate-nth-eq-map*: *i-aggregate f k ag n* = *ag* (*map f* [*n* ∗ *k*..<*n* ∗ *k* + *k*])
⟨*proof*⟩

lemma *i-aggregate-i-append-mod*:
length xs mod k = 0 ⟹
i-aggregate (*xs* ⌢ *f*) *k ag* = *f-aggregate xs k ag* ⌢ *i-aggregate f k ag*
⟨*proof*⟩

lemma *i-aggregate-i-append-mult*:
*length xs = m * k ⟹*
i-aggregate (xs ⌢ f) k ag = f-aggregate xs k ag ⌢ i-aggregate f k ag
⟨*proof*⟩

lemma *i-aggregate-Cons*:
⟦ 0 < k; length xs = k ⟧ ⟹
i-aggregate (xs ⌢ f) k ag = [ag xs] ⌢ (i-aggregate f k ag)
⟨*proof*⟩

lemma *i-aggregate-take-nth*:
n < m div k ⟹ f-aggregate (f ⇓ m) k ag ! n = i-aggregate f k ag n
⟨*proof*⟩

lemma *i-aggregate-i-take*:
f-aggregate (f ⇓ n) k ag = i-aggregate f k ag ⇓ (n div k)
⟨*proof*⟩

lemma *i-aggregate-i-take-mult*:
*0 < k ⟹ f-aggregate (f ⇓ (n * k)) k ag = i-aggregate f k ag ⇓ n*
⟨*proof*⟩

lemma *i-aggregate-i-drop-mult*:
*i-aggregate (f ⇑ (n * k)) k ag = i-aggregate f k ag ⇑ n*
⟨*proof*⟩

lemma *i-aggregate-i-drop-mod*:
n mod k = 0 ⟹
i-aggregate (f ⇑ n) k ag = i-aggregate f k ag ⇑ (n div k)
⟨*proof*⟩

lemma *i-aggregate-assoc*:
⟦ 0 < a; 0 < b;
⋀xs. length xs mod a = 0 ⟹ ag (f-aggregate xs a ag) = ag xs ⟧ ⟹
*i-aggregate (i-aggregate f a ag) b ag = i-aggregate f (a * b) ag*
⟨*proof*⟩

lemma *i-aggregate-commute*:
⟦ 0 < a; 0 < b;
⋀xs. length xs mod a = 0 ⟹ ag (f-aggregate xs a ag) = ag xs;
⋀xs. length xs mod b = 0 ⟹ ag (f-aggregate xs b ag) = ag xs ⟧ ⟹
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 ⇒ 'a message-af*

where
last-message [] = ε
| *last-message* (*x* # *xs*) = (*if last-message xs* = ε *then x else last-message xs*)

definition *f-shrink* :: *'a fstream-af* ⇒ *nat* ⇒ *'a fstream-af* (**infixl** $\div_f$ *100*)
where *f-shrink xs k* ≡ *f-aggregate xs k last-message*

definition *i-shrink* :: *'a istream-af* ⇒ *nat* ⇒ *'a istream-af* (**infixl** $\div_i$ *100*)
where *i-shrink f k* ≡ *i-aggregate f k last-message*

notation
f-shrink (**infixl** ÷ *100*) **and**
i-shrink (**infixl** ÷ *100*)

lemmas *f-shrink-defs* = *f-shrink-def f-aggregate-def*
lemmas *i-shrink-defs* = *i-shrink-def i-aggregate-def*

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

lemma *last-message-one*: *last-message* [*m*] = *m*
⟨*proof*⟩

lemma *last-message-replicate*: *0* < *n* ⟹ *last-message* (m^n) = *m*
⟨*proof*⟩

lemma *last-message-replicate-NoMsg*: *last-message* (ε^n) = ε
⟨*proof*⟩

lemma *last-message-Cons-NoMsg*: *last-message* (ε # *xs*) = *last-message xs*
⟨*proof*⟩

lemma *last-message-append-one*:
last-message (*xs* @ [*m*]) = (*if m* = ε *then last-message xs else m*)
⟨*proof*⟩

lemma *last-message-append*: ⋀*xs*.
last-message (*xs* @ *ys*) = (
if last-message ys = ε *then last-message xs else last-message ys*)
⟨*proof*⟩

corollary *last-message-append-replicate-NoMsg*:
last-message (*xs* @ ε^n) = *last-message xs*
⟨*proof*⟩

lemma *last-message-replicate-NoMsg-append*:
last-message (ε^n @ *xs*) = *last-message xs*

⟨*proof*⟩

lemma *last-message-NoMsg-conv*:
(*last-message xs* = ε) = (∀ *i*<*length xs*. *xs* ! *i* = ε)
⟨*proof*⟩

lemma *last-message-not-NoMsg-conv*:
(*last-message xs* ≠ ε) = (∃ *i*<*length xs*. *xs* ! *i* ≠ ε)
⟨*proof*⟩

lemma *not-NoMsg-imp-last-message*:
⟦ *i* < *length xs*; *xs* ! *i* ≠ ε ⟧ ⟹ *last-message xs* ≠ ε
⟨*proof*⟩

lemma *last-message-exists-nth*:
last-message xs ≠ ε ⟹
∃ *i*<*length xs*. *last-message xs* = *xs* ! *i* ∧ (∀ *j*<*length xs*. *i* < *j* ⟶ *xs* ! *j* = ε)
⟨*proof*⟩

lemma *last-message-exists-nth'*:
last-message xs ≠ ε ⟹ ∃ *i*<*length xs*. *last-message xs* = *xs* ! *i*
⟨*proof*⟩

lemma *last-messageI2-aux*: ⋀*i*.
⟦ *i* < *length xs*; *xs* ! *i* ≠ ε;
∀ *j*. *i* < *j* ∧ *j* < *length xs* ⟶ *xs* ! *j* = ε ⟧ ⟹
last-message xs = *xs* ! *i*
⟨*proof*⟩

lemma *last-messageI2*:
⟦ *i* < *length xs*; *xs* ! *i* ≠ ε;
⋀*j*. ⟦ *i* < *j*; *j* < *length xs* ⟧ ⟹ *xs* ! *j* = ε ⟧ ⟹
last-message xs = *xs* ! *i*
⟨*proof*⟩

lemma *last-messageI*:
⟦ *m* ≠ ε; *i* < *length xs*; *xs* ! *i* = *m*;
⋀*j*. ⟦ *i* < *j*; *j* < *length xs* ⟧ ⟹ *xs* ! *j* = ε ⟧ ⟹
last-message xs = *m*
⟨*proof*⟩

lemma *last-message-Msg-eq-last*:
⟦ *xs* ≠ []; *last xs* ≠ ε ⟧ ⟹ *last-message xs* = *last xs*
⟨*proof*⟩

lemma *last-message-conv*:
m ≠ ε ⟹
(*last-message xs* = *m*) =
(∃ *i*<*length xs*. *xs* ! *i* = *m* ∧ (∀ *j*<*length xs*. *i* < *j* ⟶ *xs* ! *j* = ε))

⟨*proof*⟩

lemma *last-message-conv-if*:
(*last-message xs* = *m*) =
(*if m* = ε *then* ∀ *i*<*length xs*. *xs* ! *i* = ε
else ∃ *i*<*length xs*. *xs* ! *i* = *m* ∧ (∀ *j*<*length xs*. *i* < *j* ⟶ *xs* ! *j* = ε))
⟨*proof*⟩

lemma *last-message-not-NoMsg-eq-conv*:
⟦ *last-message xs* ≠ ε; *last-message ys* ≠ ε ⟧ ⟹
(*last-message xs* = *last-message ys*) =
(∃ *i j*. *i* < *length xs* ∧ *j* < *length ys* ∧ *xs* ! *i* ≠ ε ∧
xs ! *i* = *ys* ! *j* ∧
(∀ *n*<*length xs*. *i* < *n* ⟶ *xs* ! *n* = ε) ∧
(∀ *n*<*length ys*. *j* < *n* ⟶ *ys* ! *n* = ε))
⟨*proof*⟩

lemma *f-shrink-0*[*simp*]: *xs* ÷$_f$ *0* = []
⟨*proof*⟩

lemma *f-shrink-1*[*simp*]: *xs* ÷$_f$ *Suc 0* = *xs*
⟨*proof*⟩

lemma *f-shrink-Nil*[*simp*]: [] ÷$_f$ *k* = []
⟨*proof*⟩

lemma *f-shrink-length*: *length* (*xs* ÷$_f$ *k*) = *length xs div k*
⟨*proof*⟩

lemma *f-shrink-empty-conv*: *0* < *k* ⟹ (*xs* ÷$_f$ *k* = []) = (*length xs* < *k*)
⟨*proof*⟩

lemma *f-shrink-Cons*:
⟦ *0* < *k*; *length xs* = *k* ⟧ ⟹ (*xs* @ *ys*) ÷$_f$ *k* = *last-message xs* # (*ys* ÷$_f$ *k*)
⟨*proof*⟩

lemma *f-shrink-one*:
⟦ *0* < *k*; *length xs* = *k* ⟧ ⟹ *xs* ÷$_f$ *k* = [*last-message xs*]
⟨*proof*⟩

lemma *f-shrink-eq-f-shrink-take*:
xs ↓ (*length xs div k* ∗ *k*) ÷$_f$ *k* = *xs* ÷$_f$ *k*
⟨*proof*⟩

lemma *f-shrink-nth*:
n < *length xs div k* ⟹
(*xs* ÷$_f$ *k*) ! *n* = *last-message* (*xs* ↑ (*n* ∗ *k*) ↓ *k*)
⟨*proof*⟩

lemma *f-shrink-nth-eq-sublist-list*:
$n < length\ xs\ div\ k \Longrightarrow$
$(xs \div_f k)\ !\ n = last\text{-}message\ (sublist\text{-}list\ xs\ [n * k..<n * k + k])$
⟨*proof*⟩

lemma *f-shrink-take-nth*:
$[\![\ n < length\ xs\ div\ k;\ n < m\ div\ k\ ]\!] \Longrightarrow (xs \downarrow m) \div_f k\ !\ n = xs \div_f k\ !\ n$
⟨*proof*⟩

lemma *f-shrink-hd*:
$[\![\ 0 < k;\ k \leq length\ xs\ ]\!] \Longrightarrow hd\ (xs \div_f k) = last\text{-}message\ (xs \downarrow k)$
⟨*proof*⟩

lemma *f-shrink-append-mod*:
$length\ xs\ mod\ k = 0 \Longrightarrow (xs\ @\ ys) \div_f k = xs \div_f k\ @\ (ys \div_f k)$
⟨*proof*⟩

lemma *f-shrink-append-mult*:
$length\ xs = m * k \Longrightarrow (xs\ @\ ys) \div_f k = xs \div_f k\ @\ (ys \div_f k)$
⟨*proof*⟩

lemma *f-shrink-snoc*:
$[\![\ 0 < k;\ length\ ys = k;\ length\ xs\ mod\ k = 0\ ]\!] \Longrightarrow$
$(xs\ @\ ys) \div_f k = xs \div_f k\ @\ [last\text{-}message\ ys]$
⟨*proof*⟩

lemma *f-shrink-last-message*[*rule-format*]:
$length\ xs\ mod\ k = 0 \longrightarrow last\text{-}message\ (xs \div_f k) = last\text{-}message\ xs$
⟨*proof*⟩

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

lemma *f-shrink-f-expand-id*: $0 < k \Longrightarrow xs \odot_f k \div_f k = xs$
⟨*proof*⟩

lemma *f-expand-f-shrink-id-take*[*rule-format*]:
$[\![\ \forall i{<}length\ xs.\ 0 < i\ mod\ k \longrightarrow xs\ !\ i = \varepsilon\ ]\!] \Longrightarrow$
$xs \div_f k \odot_f k = xs \downarrow (length\ xs\ div\ k * k)$
⟨*proof*⟩

corollary *f-expand-f-shrink-id-mod-0*:
$[\![\ length\ xs\ mod\ k = 0;$
$\bigwedge i.\ [\![\ i < length\ xs;\ 0 < i\ mod\ k\ ]\!] \Longrightarrow xs\ !\ i = \varepsilon\ ]\!] \Longrightarrow$
$xs \div_f k \odot_f k = xs$
⟨*proof*⟩

lemma *f-shrink-take*:

$xs \downarrow n \div_f k = xs \div_f k \downarrow (n \mathit{div} k)$
⟨*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*:
$n \mathit{mod} k = 0 \Longrightarrow xs \uparrow n \div_f k = xs \div_f k \uparrow (n \mathit{div} k)$
⟨*proof*⟩

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

lemma *f-shrink-eq-conv'*:
$(xs' \div_f k = xs) =$
$(\mathit{length}\ xs' \mathit{div} k = \mathit{length}\ xs \wedge$
$(\forall i{<}\mathit{length}\ xs.$
$\quad \mathit{if}\ xs\ !\ i = \varepsilon\ \mathit{then}\ (\forall j{<}k.\ xs'\ !\ (i * k + j) = \varepsilon)$
$\quad \mathit{else}\ (\exists n{<}k.\ xs'\ !\ (i * k + n) = xs\ !\ i \wedge$
$\qquad (\forall j{<}k.\ n < j \longrightarrow xs'\ !\ (i * k + j) = \varepsilon))))$
⟨*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*]: $f \div_i \mathit{Suc}\ 0 = f$
⟨*proof*⟩
lemma *i-shrink-nth*: $(f \div_i k)\ n = \mathit{last\text{-}message}\ (f \Uparrow (n * k) \Downarrow k)$
⟨*proof*⟩
lemma *i-shrink-nth-eq-map*: $(f \div_i k)\ n = \mathit{last\text{-}message}\ (\mathit{map}\ f\ [n * k..{<}n * k + k])$
⟨*proof*⟩
lemma *i-shrink-hd*: $(f \div_i k)\ 0 = \mathit{last\text{-}message}\ (f \Downarrow k)$
⟨*proof*⟩

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

⟨*proof*⟩

lemma *i-shrink-i-append-mult*:
  length xs = *m* ∗ *k* ⟹ (*xs* ⌢ *f*) ÷$_i$ *k* = *xs* ÷$_f$ *k* ⌢ (*f* ÷$_i$ *k*)
⟨*proof*⟩

lemma *i-shrink-Cons*:
  ⟦ *0* < *k*; *length xs* = *k* ⟧ ⟹ (*xs* ⌢ *f*) ÷$_i$ *k* = [*last-message xs*] ⌢ (*f* ÷$_i$ *k*)
⟨*proof*⟩

lemma *i-shrink-take-nth*:
  n < *m div k* ⟹ (*f* ⇓ *m*) ÷$_f$ *k* ! *n* = (*f* ÷$_i$ *k*) *n*
⟨*proof*⟩

lemma *i-shrink-const*[*simp*]: *0* < *k* ⟹ (λ*x*. *m*) ÷$_i$ *k* = (λ*x*. *m*)
⟨*proof*⟩

lemma *i-shrink-const-NoMsg*[*simp*]: (λ*x*. ε) ÷$_i$ *k* = (λ*x*. ε)
⟨*proof*⟩

lemma *i-shrink-i-expand-id*: *0* < *k* ⟹ *f* ⊙$_i$ *k* ÷$_i$ *k* = *f*
⟨*proof*⟩

lemma *i-shrink-i-take-mult*: *0* < *k* ⟹ *f* ⇓ (*n* ∗ *k*) ÷$_f$ *k* = *f* ÷$_i$ *k* ⇓ *n*
⟨*proof*⟩

lemma *i-shrink-i-take*:
  f ⇓ *n* ÷$_f$ *k* = *f* ÷$_i$ *k* ⇓ (*n div k*)
⟨*proof*⟩

lemma *i-shrink-i-drop-mult*: *f* ⇑ (*n* ∗ *k*) ÷$_i$ *k* = *f* ÷$_i$ *k* ⇑ *n*
⟨*proof*⟩
lemma *i-shrink-i-drop-mod*:
  n mod k = *0* ⟹ *f* ⇑ *n* ÷$_i$ *k* = *f* ÷$_i$ *k* ⇑ (*n div k*)
⟨*proof*⟩

lemma *i-shrink-eq-conv*:
  (*f* ÷$_i$ *k1* = *g* ÷$_i$ *k2*) =
  (∀ *i*. *last-message* (*f* ⇑ (*i* ∗ *k1*) ⇓ *k1*) =
      last-message (*g* ⇑ (*i* ∗ *k2*) ⇓ *k2*))
⟨*proof*⟩

lemma *i-shrink-eq-conv′*:
  (*f′* ÷$_i$ *k* = *f*) =
  (∀ *i*. *if f i* = ε *then* ∀ *j*<*k*. *f′* (*i* ∗ *k* + *j*) = ε
      else ∃ *n*<*k*. *f′* (*i* ∗ *k* + *n*) = *f i* ∧
                ∀ *j*<*k*. *n* < *j* ⟶ *f′* (*i* ∗ *k* + *j*) = ε))
⟨*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* ⇒ *'a message-af* ⇒ *'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* ⇒ *'a fstream-af*
  where *last-message-hold xs* ≡ *last-message-hold-init xs* ε

lemma *last-message-hold-init-length*[*simp*]:
  ⋀*m*. *length* (*last-message-hold-init xs m*) = *length xs*
⟨*proof*⟩

lemma *last-message-hold-init-nth*:
  ⋀*i m*. *i* < *length xs* ⟹
  (*last-message-hold-init xs m*) ! *i* = *last-message* (*m* # *xs* ↓ *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*]:
  ⋀*xs m*. *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*]: *length* (*last-message-hold xs*) = *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 < *length xs* ⟹ *last-message-hold xs* ! *i* = *last-message* (*xs* ↓ *Suc i*)

⟨*proof*⟩

lemma *last-message-hold-last*:
xs ≠ [] ⟹ *last* (*last-message-hold xs*) = *last-message xs*
⟨*proof*⟩

lemma *last-message-hold-take*:
last-message-hold xs ↓ *n* = *last-message-hold* (*xs* ↓ *n*)
⟨*proof*⟩

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

lemma *last-message-hold-append*:
last-message-hold (*xs* @ *ys*) =
last-message-hold xs @ *last-message-hold-init ys* (*last-message xs*)
⟨*proof*⟩

lemma *last-message-hold-append′*:
last-message-hold (*xs* @ *ys*) =
last-message-hold xs @ *tl* (*last-message-hold* (*last-message xs* # *ys*))
⟨*proof*⟩

lemma *last-message-last-message-hold*[*simp*]:
last-message (*last-message-hold xs*) = *last-message xs*
⟨*proof*⟩

lemma *last-message-hold-idem*[*simp*]:
last-message-hold (*last-message-hold xs*) = *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* ⇒ *nat* ⇒ *′a fstream-af* (**infixl** ⟼$_f$ *100*)
where *f-last-message-hold xs k* ≡ *concat* (*map last-message-hold* (*list-slice2 xs k*))

definition *i-last-message-hold* :: *′a istream-af* ⇒ *nat* ⇒ *′a istream-af* (**infixl** ⟼$_i$ *100*)
where *i-last-message-hold f k* ≡ λ*n*. *last-message* (*f* ⇑ (*n* − *n mod k*) ⇓ *Suc* (*n mod k*))

notation
f-last-message-hold (**infixl** ⟼ *100*) **and**
i-last-message-hold (**infixl** ⟼ *100*)

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

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

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

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

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

lemma *f-last-message-hold-append-mult*:
$length\ xs = m * k \Longrightarrow (xs\ @\ ys) \longmapsto_f k = xs \longmapsto_f k\ @\ (ys \longmapsto_f k)$
⟨*proof*⟩

lemma *f-last-message-hold-append-mod*:
$length\ xs\ mod\ k = 0 \Longrightarrow (xs\ @\ ys) \longmapsto_f k = xs \longmapsto_f k\ @\ (ys \longmapsto_f k)$
⟨*proof*⟩

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

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

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

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

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

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

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

lemma *last-message-f-last-message-hold*:
last-message $(xs \longmapsto_f k) = $ *last-message* xs
⟨*proof*⟩

lemma *i-last-message-hold-0*[*simp*]: $f \longmapsto_i 0 = (\lambda n.\ \textit{last-message}\ (f \Downarrow \textit{Suc}\ n))$
⟨*proof*⟩
lemma *i-last-message-hold-1*[*simp*]: $f \longmapsto_i \textit{Suc}\ 0 = f$
⟨*proof*⟩

lemma *i-last-message-hold-nth*:
$(f \longmapsto_i k)\ n = \textit{last-message}\ (f \Uparrow (n - n\ \textit{mod}\ k) \Downarrow \textit{Suc}\ (n\ \textit{mod}\ k))$
⟨*proof*⟩

lemma *i-last-message-hold-i-append-mult*:
$\textit{length}\ xs = m * k \Longrightarrow (xs \frown f) \longmapsto_i k = (xs \longmapsto_f k) \frown (f \longmapsto_i k)$
⟨*proof*⟩

lemma *i-last-message-hold-i-append-mod*:
$\textit{length}\ xs\ \textit{mod}\ k = 0 \Longrightarrow (xs \frown f) \longmapsto_i k = (xs \longmapsto_f k) \frown (f \longmapsto_i k)$
⟨*proof*⟩

lemma *i-last-message-hold-i-take*: $f \Downarrow n \longmapsto_f k = (f \longmapsto_i k) \Downarrow n$
⟨*proof*⟩

lemma *i-last-message-hold-i-drop-mult*:
$f \Uparrow (n * k) \longmapsto_i k = f \longmapsto_i k \Uparrow (n * k)$
⟨*proof*⟩

lemma *i-last-message-hold-i-drop-mod*:
$n\ \textit{mod}\ k = 0 \Longrightarrow f \Uparrow n \longmapsto_i k = f \longmapsto_i k \Uparrow n$
⟨*proof*⟩

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

lemma *i-shrink-nth-eq-i-last-message-hold-nth*:
$0 < k \Longrightarrow (f \div_i k)\ n = (f \longmapsto_i k)\ (n * k + k - \textit{Suc}\ 0)$
⟨*proof*⟩

lemma *i-shrink-nth-eq-i-last-message-hold-last*:
$0 < k \Longrightarrow (f \div_i k)\ n = \textit{last}\ (f \longmapsto_i k \Uparrow (n * k) \Downarrow k)$
⟨*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-shrink-last* :: *'a list ⇒ nat ⇒ 'a list* (**infixl** $\div_{fl}$ *100*)
where *f-shrink-last xs k ≡ f-aggregate xs k last*

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

notation
f-shrink-last (**infixl** $\div_{l}$ *100*) **and**
i-shrink-last (**infixl** $\div_{l}$ *100*)

lemma *f-shrink-last-0*[*simp*]: *xs* $\div_{fl}$ *0* = []
⟨*proof*⟩

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

lemma *f-shrink-last-Nil*[*simp*]: [] $\div_{fl}$ *k* = []
⟨*proof*⟩

lemma *f-shrink-last-length*: *length* (*xs* $\div_{fl}$ *k*) = *length xs div k*
⟨*proof*⟩

lemma *f-shrink-last-empty-conv*:
0 < k ⟹ (*xs* $\div_{fl}$ *k* = []) = (*length xs < k*)
⟨*proof*⟩

lemma *f-shrink-last-Cons*:
⟦ *0 < k*;
length xs = *k* ⟧ ⟹ (*xs* @ *ys*) $\div_{fl}$ *k* = *last xs* # (*ys* $\div_{fl}$ *k*)
⟨*proof*⟩

lemma *f-shrink-last-one*:
⟦ *0 < k*; *length xs* = *k* ⟧ ⟹ *xs* $\div_{fl}$ *k* = [*last xs*]
⟨*proof*⟩

lemma *f-shrink-last-eq-f-shrink-last-take*:
xs ↓ (*length xs div k* ∗ *k*) $\div_{fl}$ *k* = *xs* $\div_{fl}$ *k*
⟨*proof*⟩

lemma *f-shrink-last-nth*:
n < length xs div k ⟹ (*xs* $\div_{fl}$ *k*) ! *n* = *xs* ! (*n* ∗ *k* + *k* − *Suc 0*)
⟨*proof*⟩

corollary *f-shrink-last-nth'*:
n < length xs div k ⟹ (*xs* $\div_{fl}$ *k*) ! *n* = *xs* ! (*Suc n* ∗ *k* − *Suc 0*)
⟨*proof*⟩

lemma *f-shrink-last-hd*:
$[\![\ 0 < k;\ k \leq length\ xs\ ]\!] \Longrightarrow hd\ (xs \div_{fl} k) = xs\ !\ (k - Suc\ 0)$
⟨*proof*⟩

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

lemma *f-shrink-last-append-mod*:
$length\ xs\ mod\ k = 0 \Longrightarrow (xs\ @\ ys) \div_{fl} k = xs \div_{fl} k\ @\ (ys \div_{fl} k)$
⟨*proof*⟩

lemma *f-shrink-last-append-mult*:
$length\ xs = m * k \Longrightarrow (xs\ @\ ys) \div_{fl} k = xs \div_{fl} k\ @\ (ys \div_{fl} k)$
⟨*proof*⟩

lemma *f-shrink-last-snoc*:
$[\![\ 0 < k;\ length\ ys = k;\ length\ xs\ mod\ k = 0\ ]\!] \Longrightarrow$
$(xs\ @\ ys) \div_{fl} k = xs \div_{fl} k\ @\ [last\ ys]$
⟨*proof*⟩

lemma *f-shrink-last-last*:
$length\ xs\ mod\ k = 0 \Longrightarrow last\ (xs \div_{fl} k) = last\ xs$
⟨*proof*⟩

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

lemma *f-shrink-last-take*:
$xs \downarrow n \div_{fl} k = xs \div_{fl} k \downarrow (n\ div\ k)$
⟨*proof*⟩

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

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

lemma *f-shrink-last-drop-mod*:
$n\ mod\ k = 0 \Longrightarrow xs \uparrow n \div_{fl} k = xs \div_{fl} k \uparrow (n\ div\ k)$
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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

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

lemma *i-shrink-last-i-take*:
$f \Downarrow n \div_{fl} k = f \div_{il} k \Downarrow (n\ div\ k)$
⟨*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*:
$n\ mod\ k = 0 \Longrightarrow f \Uparrow n \div_{il} k = f \div_{il} k \Uparrow (n\ div\ k)$
⟨*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*:
$xs \div_f k = xs \longmapsto_f k \div_{fl} k$
⟨*proof*⟩

lemma *i-shrink-eq-i-last-message-hold-shrink-last*:
$0 < k \Longrightarrow f \div_i k = f \longmapsto_i k \div_{il} k$
⟨*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* ⇒ '*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* ⇒ '*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* ⇒ '*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* ⇒ '*comp* ⇒ '*comp*

— Execute a component for all inputs in the input stream '*input list*
primrec *f-Exec-Comp* :: ('*comp*, '*input*) *Comp-Trans-Fun* ⇒ '*input list* ⇒ '*comp* ⇒ '*comp*
where

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

— Execute the component for at most n steps
definition *f-Exec-Comp-N* :: (*'comp*, *'input*) *Comp-Trans-Fun* ⇒ *nat* ⇒ *'input list* ⇒ *'comp* ⇒ *'comp*
where *f-Exec-Comp-N trans-fun n xs c* ≡ *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* ⇒ *'input list* ⇒ *'comp* ⇒ *'comp list*
where
f-Exec-Stream-Nil: *f-Exec-Comp-Stream trans-fun* [] *c* = []
| *f-Exec-Stream-Cons*: *f-Exec-Comp-Stream trans-fun* (*x* # *xs*) *c* =
(*trans-fun x c*) # (*f-Exec-Comp-Stream trans-fun xs* (*trans-fun x c*))

primrec *f-Exec-Comp-Stream-Init* ::
(*'comp*, *'input*) *Comp-Trans-Fun* ⇒ *'input list* ⇒ *'comp* ⇒ *'comp list*
where
f-Exec-Stream-Init-Nil: *f-Exec-Comp-Stream-Init trans-fun* [] *c* = [*c*]
| *f-Exec-Stream-Init-Cons*: *f-Exec-Comp-Stream-Init trans-fun* (*x* # *xs*) *c* =
c # (*f-Exec-Comp-Stream-Init trans-fun xs* (*trans-fun x c*))

definition *i-Exec-Comp-Stream* ::
(*'comp*, *'input*) *Comp-Trans-Fun* ⇒ *'input ilist* ⇒ *'comp* ⇒ *'comp ilist*
where *i-Exec-Comp-Stream* ≡ λ*trans-fun input c n*. *f-Exec-Comp trans-fun* (*input* ⇓ *Suc n*) *c*

definition *i-Exec-Comp-Stream-Init* ::
(*'comp*, *'input*) *Comp-Trans-Fun* ⇒ *'input ilist* ⇒ *'comp* ⇒ *'comp ilist*
where *i-Exec-Comp-Stream-Init* ≡ λ*trans-fun input c n*. *f-Exec-Comp trans-fun* (*input* ⇓ *n*) *c*

3.1.2 Basic results

lemma *f-Exec-one*: *f-Exec-Comp trans-fun* [*m*] *c* = *trans-fun m c*
⟨*proof*⟩

lemma *f-Exec-Stream-length*[*rule-format*, *simp*]:
∀ *c*. *length* (*f-Exec-Comp-Stream trans-fun xs c*) = *length xs*
⟨*proof*⟩

lemma *f-Exec-Stream-empty-conv*:
(*f-Exec-Comp-Stream trans-fun xs c* = []) = (*xs* = [])
⟨*proof*⟩

lemma *f-Exec-Stream-not-empty-conv*:
(*f-Exec-Comp-Stream trans-fun xs c* ≠ []) = (*xs* ≠ [])

⟨*proof*⟩

lemma *f-Exec-eq-f-Exec-Stream-last*[*rule-format*]:
∀ *c*. *f-Exec-Comp trans-fun xs c* = *last* (*c* # (*f-Exec-Comp-Stream trans-fun xs c*))
⟨*proof*⟩

corollary *f-Exec-eq-f-Exec-Stream-last2*[*rule-format*]:
xs ≠ [] ⟹
f-Exec-Comp trans-fun xs c = *last* (*f-Exec-Comp-Stream trans-fun xs c*)
⟨*proof*⟩

corollary *f-Exec-eq-f-Exec-Stream-last-if*:
f-Exec-Comp trans-fun xs c = (*if xs* = [] *then c else last* (*f-Exec-Comp-Stream trans-fun xs c*))
⟨*proof*⟩

corollary *f-Exec-take-eq-last-f-Exec-Stream-take*:
⟦ *xs* ≠ []; *0* < *n* ⟧ ⟹
f-Exec-Comp trans-fun (*xs* ↓ *n*) *c* =
last (*f-Exec-Comp-Stream trans-fun* (*xs* ↓ *n*) *c*)
⟨*proof*⟩

corollary *f-Exec-N-eq-last-f-Exec-Stream-take*:
⟦ *xs* ≠ []; *0* < *n* ⟧ ⟹
f-Exec-Comp-N trans-fun n xs c =
last (*f-Exec-Comp-Stream trans-fun* (*xs* ↓ *n*) *c*)
⟨*proof*⟩

lemma *f-Exec-Stream-nth*:
⋀*n c*. *n* < *length xs* ⟹
f-Exec-Comp-Stream trans-fun xs c ! *n* = *f-Exec-Comp trans-fun* (*xs* ↓ *Suc n*) *c*
⟨*proof*⟩

lemma *f-Exec-Stream-nth2*:
n ≤ *length xs* ⟹
(*c* # *f-Exec-Comp-Stream trans-fun xs c*) ! *n* = *f-Exec-Comp trans-fun* (*xs* ↓ *n*) *c*
⟨*proof*⟩

lemma *f-Exec-N-all*:
length xs ≤ *n* ⟹
f-Exec-Comp-N trans-fun n xs c = *f-Exec-Comp trans-fun xs c*
⟨*proof*⟩

lemma *f-Exec-Stream-append*[*rule-format*]:∀ *c*.
f-Exec-Comp-Stream trans-fun (*xs* @ *ys*) *c* =
(*f-Exec-Comp-Stream trans-fun xs c*) @
(*f-Exec-Comp-Stream trans-fun ys* (*f-Exec-Comp trans-fun xs c*))

⟨*proof*⟩

corollary *f-Exec-Stream-append-last-Cons*[*rule-format*]:
f-Exec-Comp-Stream trans-fun (*xs* @ *ys*) *c* =
(*f-Exec-Comp-Stream trans-fun xs c*) @
(*f-Exec-Comp-Stream trans-fun ys* (*last* (*c* # (*f-Exec-Comp-Stream trans-fun xs c*))))
⟨*proof*⟩

corollary *f-Exec-Stream-append-last*[*rule-format*]:
xs ≠ [] ⟹
f-Exec-Comp-Stream trans-fun (*xs* @ *ys*) *c* =
(*f-Exec-Comp-Stream trans-fun xs c*) @
(*f-Exec-Comp-Stream trans-fun ys* (*last* (*f-Exec-Comp-Stream trans-fun xs c*)))
⟨*proof*⟩

corollary *f-Exec-Stream-append-if*:
f-Exec-Comp-Stream trans-fun (*xs* @ *ys*) *c* =
(*f-Exec-Comp-Stream trans-fun xs c*) @
(*f-Exec-Comp-Stream trans-fun ys* (
if xs = [] *then c else last* (*f-Exec-Comp-Stream trans-fun xs c*)))
⟨*proof*⟩
corollary *f-Exec-append*:
f-Exec-Comp trans-fun (*xs* @ *ys*) *c* =
f-Exec-Comp trans-fun ys (*f-Exec-Comp trans-fun xs c*)
⟨*proof*⟩

corollary *f-Exec-Stream-Cons-rev*:
xs ≠ [] ⟹
(*trans-fun* (*hd xs*) *c*) #
f-Exec-Comp-Stream trans-fun (*tl xs*) (*trans-fun* (*hd xs*) *c*) =
f-Exec-Comp-Stream trans-fun xs c
⟨*proof*⟩

lemma *f-Exec-Stream-snoc*:
f-Exec-Comp-Stream trans-fun (*xs* @ [*x*]) *c* =
f-Exec-Comp-Stream trans-fun xs c @
[*trans-fun x* (*f-Exec-Comp trans-fun xs c*)]
⟨*proof*⟩

lemma *f-Exec-snoc*:
f-Exec-Comp trans-fun (*xs* @ [*x*]) *c* =
trans-fun x (*f-Exec-Comp trans-fun xs c*)
⟨*proof*⟩

lemma *f-Exec-N-append*[*rule-format*]:
f-Exec-Comp-N trans-fun (*a* + *b*) *xs c* =
f-Exec-Comp-N trans-fun b (*xs* ↑ *a*) (*f-Exec-Comp-N trans-fun a xs c*)

⟨*proof*⟩

corollary *f-Exec-N-Suc*[*rule-format*]:
f-Exec-Comp-N trans-fun (*Suc n*) *xs c* =
f-Exec-Comp-N trans-fun (*Suc 0*) (*xs* ↑ *n*) (*f-Exec-Comp-N trans-fun n xs c*)
⟨*proof*⟩

corollary *f-Exec-N-Suc2*[*rule-format*]:
n < *length xs* ⟹
f-Exec-Comp-N trans-fun (*Suc n*) *xs c* =
trans-fun (*xs* ! *n*) (*f-Exec-Comp-N trans-fun n xs c*)
⟨*proof*⟩

theorem *f-Exec-Stream-take*:
(*f-Exec-Comp-Stream trans-fun xs c*) ↓ *n* =
f-Exec-Comp-Stream trans-fun (*xs* ↓ *n*) *c*
⟨*proof*⟩

theorem *f-Exec-Stream-drop*:
(*f-Exec-Comp-Stream trans-fun xs c*) ↑ *n* =
f-Exec-Comp-Stream trans-fun (*xs* ↑ *n*)
(*f-Exec-Comp trans-fun* (*xs* ↓ *n*) *c*)
⟨*proof*⟩

lemma *i-Exec-Stream-nth*:
i-Exec-Comp-Stream trans-fun input c n = *f-Exec-Comp trans-fun* (*input* ⇓ *Suc n*) *c*
⟨*proof*⟩

lemma *i-Exec-Stream-nth-Suc*:
i-Exec-Comp-Stream trans-fun input c (*Suc n*) =
trans-fun (*input* (*Suc n*)) (*i-Exec-Comp-Stream trans-fun input c n*)
⟨*proof*⟩

lemma *i-Exec-Stream-nth-Suc-first*:
i-Exec-Comp-Stream trans-fun input c (*Suc n*) =
(*i-Exec-Comp-Stream trans-fun* (*input* ⇑ *Suc 0*) (*trans-fun* (*input 0*) *c*) *n*)
⟨*proof*⟩

lemma *f-Exec-Stream-nth-eq-i-Exec-Stream-nth*:
n < *n*′ ⟹
f-Exec-Comp-Stream trans-fun (*input* ⇓ *n*′) *c* ! *n* =
i-Exec-Comp-Stream trans-fun input c n
⟨*proof*⟩

lemma *i-Exec-Stream-append*:
i-Exec-Comp-Stream trans-fun (*xs* ⌢ *input*) *c* =
f-Exec-Comp-Stream trans-fun xs c ⌢

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 ≠ [] ⟹
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*] ⌢
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*) ⇓ *n* =
f-Exec-Comp-Stream trans-fun (*input* ⇓ *n*) *c*
⟨*proof*⟩

theorem *i-Exec-Stream-drop*:
(*i-Exec-Comp-Stream trans-fun input c*) ⇑ *n* =
i-Exec-Comp-Stream trans-fun (*input* ⇑ *n*) (*f-Exec-Comp trans-fun* (*input* ⇓ *n*)
c)
⟨*proof*⟩

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

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

corollary *f-Exec-Stream-expand-aggregate-take*:
f-aggregate (*f-Exec-Comp-Stream trans-fun* (*xs* $\odot_f$ *k*) *c*) *k ag* $\downarrow$ *n* =
f-aggregate (*f-Exec-Comp-Stream trans-fun* ((*xs* $\downarrow$ *n*) $\odot_f$ *k*) *c*) *k ag*
⟨*proof*⟩

lemma *i-Exec-Stream-expand-aggregate-map-take*:
0 < *k* $\Longrightarrow$
i-aggregate (*f* $\circ$ (*i-Exec-Comp-Stream trans-fun* (*input* $\odot_i$ *k*) *c*)) *k ag* $\Downarrow$ *n* =
f-aggregate (*map f* (*f-Exec-Comp-Stream trans-fun* ((*input* $\Downarrow$ *n*) $\odot_f$ *k*) *c*)) *k ag*
⟨*proof*⟩

corollary *i-Exec-Stream-expand-aggregate-take*:
0 < *k* $\Longrightarrow$
i-aggregate (*i-Exec-Comp-Stream trans-fun* (*input* $\odot_i$ *k*) *c*) *k ag* $\Downarrow$ *n* =
f-aggregate (*f-Exec-Comp-Stream trans-fun* ((*input* $\Downarrow$ *n*) $\odot_f$ *k*) *c*) *k ag*
⟨*proof*⟩

lemma *f-Exec-Stream-expand-aggregate-map-drop*:
f-aggregate (*map f* (*f-Exec-Comp-Stream trans-fun* (*xs* $\odot_f$ *k*) *c*)) *k ag* $\uparrow$ *n* =
f-aggregate (*map f* (*f-Exec-Comp-Stream trans-fun* ((*xs* $\uparrow$ *n*) $\odot_f$ *k*) (
f-Exec-Comp trans-fun ((*xs* $\downarrow$ *n*) $\odot_f$ *k*) *c*))) *k ag*
⟨*proof*⟩

corollary *f-Exec-Stream-expand-aggregate-drop*:
f-aggregate (*f-Exec-Comp-Stream trans-fun* (*xs* $\odot_f$ *k*) *c*) *k ag* $\uparrow$ *n* =
f-aggregate (*f-Exec-Comp-Stream trans-fun* ((*xs* $\uparrow$ *n*) $\odot_f$ *k*) (
f-Exec-Comp trans-fun ((*xs* $\downarrow$ *n*) $\odot_f$ *k*) *c*)) *k ag*
⟨*proof*⟩

lemma *i-Exec-Stream-expand-aggregate-map-drop*:
0 < *k* $\Longrightarrow$
i-aggregate (*f* $\circ$ (*i-Exec-Comp-Stream trans-fun* (*input* $\odot_i$ *k*) *c*)) *k ag* $\Uparrow$ *n* =
i-aggregate (*f* $\circ$ (*i-Exec-Comp-Stream trans-fun* ((*input* $\Uparrow$ *n*) $\odot_i$ *k*) (
f-Exec-Comp trans-fun ((*input* $\Downarrow$ *n*) $\odot_f$ *k*) *c*))) *k ag*
⟨*proof*⟩

corollary *i-Exec-Stream-expand-aggregate-drop*:
0 < *k* $\Longrightarrow$
i-aggregate (*i-Exec-Comp-Stream trans-fun* (*input* $\odot_i$ *k*) *c*) *k ag* $\Uparrow$ *n* =
i-aggregate (*i-Exec-Comp-Stream trans-fun* ((*input* $\Uparrow$ *n*) $\odot_i$ *k*) (
f-Exec-Comp trans-fun ((*input* $\Downarrow$ *n*) $\odot_f$ *k*) *c*)) *k ag*
⟨*proof*⟩

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

$[\![\ 0 < k;\ n < n'\ ]\!] \Longrightarrow$
f-aggregate (*map f* (*f-Exec-Comp-Stream trans-fun* (*input* $\Downarrow$ n' $\odot_f$ *k*) *c*)) *k ag* ! n =
i-aggregate (*f* $\circ$ (*i-Exec-Comp-Stream trans-fun* (*input* $\odot_i$ *k*) *c*)) *k ag n*
⟨*proof*⟩

corollary *f-Exec-Stream-expand-aggregate-map-nth-eq-i-nth'*:
$0 < k \Longrightarrow$
f-aggregate (*map f* (*f-Exec-Comp-Stream trans-fun* (*input* $\Downarrow$ *Suc n* $\odot_f$ *k*) *c*)) *k ag* ! n =
i-aggregate (*f* $\circ$ (*i-Exec-Comp-Stream trans-fun* (*input* $\odot_i$ *k*) *c*)) *k ag n*
⟨*proof*⟩

corollary *f-Exec-Stream-expand-aggregate-nth-eq-i-nth*:
$[\![\ 0 < k;\ n < n'\ ]\!] \Longrightarrow$
f-aggregate (*f-Exec-Comp-Stream trans-fun* (*input* $\Downarrow$ n' $\odot_f$ *k*) *c*) *k ag* ! n =
i-aggregate (*i-Exec-Comp-Stream trans-fun* (*input* $\odot_i$ *k*) *c*) *k ag n*
⟨*proof*⟩

corollary *f-Exec-Stream-expand-aggregate-nth-eq-i-nth'*:
$0 < k \Longrightarrow$
f-aggregate (*f-Exec-Comp-Stream trans-fun* (*input* $\Downarrow$ *Suc n* $\odot_f$ *k*) *c*) *k ag* ! n =
i-aggregate (*i-Exec-Comp-Stream trans-fun* (*input* $\odot_i$ *k*) *c*) *k ag n*
⟨*proof*⟩

lemma *f-Exec-Stream-expand-shrink-last-map-nth-eq-f-Exec-Comp*:
$[\![\ 0 < k;\ n < \mathit{length}\ xs\ ]\!] \Longrightarrow$
map f (*f-Exec-Comp-Stream trans-fun* (*xs* $\odot_f$ *k*) *c*) $\div_{fl}$ *k* ! n =
f (*f-Exec-Comp trans-fun* ((*xs* $\downarrow$ *Suc n*) $\odot_f$ *k*) *c*)
⟨*proof*⟩

corollary *f-Exec-Stream-expand-shrink-last-nth-eq-f-Exec-Comp*:
$[\![\ 0 < k;\ n < \mathit{length}\ xs\ ]\!] \Longrightarrow$
f-Exec-Comp-Stream trans-fun (*xs* $\odot_f$ *k*) *c* $\div_{fl}$ *k* ! n =
f-Exec-Comp trans-fun ((*xs* $\downarrow$ *Suc n*) $\odot_f$ *k*) *c*
⟨*proof*⟩

lemma *f-Exec-Stream-expand-aggregate-map-nth*:
$[\![\ 0 < k;\ n < \mathit{length}\ xs\ ]\!] \Longrightarrow$
f-aggregate (*map f* (*f-Exec-Comp-Stream trans-fun* (*xs* $\odot_f$ *k*) *c*)) *k ag* ! n =
ag (*map f* (*f-Exec-Comp-Stream trans-fun* (*xs* ! n # $\varepsilon^{k\ -\ Suc\ 0}$)
(*f-Exec-Comp trans-fun* (*xs* $\downarrow$ n $\odot_f$ *k*) *c*)))
⟨*proof*⟩

corollary *f-Exec-Stream-expand-aggregate-nth*:
$[\![\ 0 < k;\ n < \mathit{length}\ xs\ ]\!] \Longrightarrow$
f-aggregate (*f-Exec-Comp-Stream trans-fun* (*xs* $\odot_f$ *k*) *c*) *k ag* ! n =
ag (*f-Exec-Comp-Stream trans-fun* (*xs* ! n # $\varepsilon^{k\ -\ Suc\ 0}$)

(*f-Exec-Comp trans-fun* (*xs* ↓ *n* ⊙$_f$ *k*) *c*))
⟨*proof*⟩

corollary *f-Exec-Stream-expand-shrink-map-nth*:
⟦ *0* < *k*; *n* < *length xs* ⟧ ⟹
(*map f* (*f-Exec-Comp-Stream trans-fun* (*xs* ⊙$_f$ *k*) *c*)) ÷$_f$ *k* ! *n* =
last-message (*map f* (*f-Exec-Comp-Stream trans-fun* (*xs* ! *n* # ε$^{k - Suc\ 0}$)
(*f-Exec-Comp trans-fun* (*xs* ↓ *n* ⊙$_f$ *k*) *c*)))
⟨*proof*⟩

lemma *i-Exec-Stream-expand-aggregate-map-nth*:
0 < *k* ⟹
i-aggregate (*f* ∘ (*i-Exec-Comp-Stream trans-fun* (*input* ⊙$_i$ *k*) *c*)) *k ag n* =
ag (*map f* (*f-Exec-Comp-Stream trans-fun* (*input n* # ε$^{k - Suc\ 0}$)
(*f-Exec-Comp trans-fun* ((*input* ⇓ *n*) ⊙$_f$ *k*) *c*)))
⟨*proof*⟩

corollary *i-Exec-Stream-expand-aggregate-nth*:
0 < *k* ⟹
i-aggregate (*i-Exec-Comp-Stream trans-fun* (*input* ⊙$_i$ *k*) *c*) *k ag n* =
ag (*f-Exec-Comp-Stream trans-fun* (*input n* # ε$^{k - Suc\ 0}$)
(*f-Exec-Comp trans-fun* ((*input* ⇓ *n*) ⊙$_f$ *k*) *c*))
⟨*proof*⟩

corollary *i-Exec-Stream-expand-shrink-map-nth*:
0 < *k* ⟹
((*f* ∘ (*i-Exec-Comp-Stream trans-fun* (*input* ⊙$_i$ *k*) *c*)) ÷$_i$ *k*) *n* =
last-message (*map f* (*f-Exec-Comp-Stream trans-fun* (*input n* # ε$^{k - Suc\ 0}$)
(*f-Exec-Comp trans-fun* (*input* ⇓ *n* ⊙$_f$ *k*) *c*)))
⟨*proof*⟩

lemma *f-Exec-Stream-expand-snoc*:
⟦ *0* < *k*; *n* < *length xs* ⟧ ⟹
f-Exec-Comp-Stream trans-fun (*xs* ⊙$_f$ *k*) *c* ↑ (*n* ∗ *k*) ↓ *k* =
f-Exec-Comp-Stream trans-fun (*xs* ! *n* # ε$^{k - Suc\ 0}$)
(*f-Exec-Comp trans-fun* (*xs* ↓ *n* ⊙$_f$ *k*) *c*)
⟨*proof*⟩

lemma *f-Exec-Stream-expand-map-aggregate-append*:
f-aggregate (*map f* (*f-Exec-Comp-Stream trans-fun* ((*xs* @ *ys*) ⊙$_f$ *k*) *c*)) *k ag* =
f-aggregate (*map f* (*f-Exec-Comp-Stream trans-fun* (*xs* ⊙$_f$ *k*) *c*)) *k ag* @
f-aggregate (*map f* (*f-Exec-Comp-Stream trans-fun* (*ys* ⊙$_f$ *k*) (
f-Exec-Comp trans-fun (*xs* ⊙$_f$ *k*) *c*))) *k ag*
⟨*proof*⟩

lemma *i-Exec-Stream-expand-map-aggregate-append*:
i-aggregate (*f* ∘ (*i-Exec-Comp-Stream trans-fun* ((*xs* ⌢ *input*) ⊙$_i$ *k*) *c*)) *k ag* =
f-aggregate (*map f* (*f-Exec-Comp-Stream trans-fun* (*xs* ⊙$_f$ *k*) *c*)) *k ag* ⌢
i-aggregate (*f* ∘ (*i-Exec-Comp-Stream trans-fun* (*input* ⊙$_i$ *k*) (

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

lemma *f-Exec-Stream-expand-map-aggregate-Cons*:
0 < *k* ⟹
f-aggregate (*map f* (*f-Exec-Comp-Stream trans-fun* ((*x* # *xs*) $\odot_f$ *k*) *c*)) *k ag* =
ag (*map f* (*f-Exec-Comp-Stream trans-fun* (*x* # $\varepsilon^{k - Suc\ 0}$) *c*)) #
f-aggregate (*map f* (*f-Exec-Comp-Stream trans-fun* (*xs* $\odot_f$ *k*) (
f-Exec-Comp trans-fun (*x* # $\varepsilon^{k - Suc\ 0}$) *c*))) *k ag*
⟨*proof*⟩

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

lemma *i-Exec-Stream-expand-map-aggregate-Cons*:
0 < *k* ⟹
i-aggregate (*f* ∘ (*i-Exec-Comp-Stream trans-fun* (([*x*] ⌢ *input*) $\odot_i$ *k*) *c*)) *k ag* =
[*ag* (*map f* (*f-Exec-Comp-Stream trans-fun* (*x* # $\varepsilon^{k - Suc\ 0}$) *c*))] ⌢
i-aggregate (*f* ∘ (*i-Exec-Comp-Stream trans-fun* (*input* $\odot_i$ *k*) (
f-Exec-Comp trans-fun (*x* # $\varepsilon^{k - Suc\ 0}$) *c*))) *k ag*
⟨*proof*⟩

lemma *f-Exec-N-eq-f-Exec-Stream-nth*:
n ≤ *length xs* ⟹
f-Exec-Comp-N trans-fun n xs c = (*c* # *f-Exec-Comp-Stream trans-fun xs c*) ! *n*
⟨*proof*⟩

theorem *f-Exec-Stream-causal*:
xs ↓ *n* = *ys* ↓ *n* ⟹
(*f-Exec-Comp-Stream trans-fun xs c*) ↓ *n* = (*f-Exec-Comp-Stream trans-fun ys c*)
↓ *n*
⟨*proof*⟩
theorem *i-Exec-Stream-causal*:
input1 ⇓ *n* = *input2* ⇓ *n* ⟹
(*i-Exec-Comp-Stream trans-fun input1 c*) ⇓ *n* = (*i-Exec-Comp-Stream trans-fun*
input2 c) ⇓ *n*
⟨*proof*⟩

Results for *f-Exec-Comp-Stream-Init*

f-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-Exec-Stream-Init-length*[*rule-format*, *simp*]:
∀ *c*. *length* (*f-Exec-Comp-Stream-Init trans-fun xs c*) = *Suc* (*length xs*)

⟨*proof*⟩

lemma *f-Exec-Stream-Init-not-empty*:
(*f-Exec-Comp-Stream-Init trans-fun xs c* ≠ [])
⟨*proof*⟩

lemma *f-Exec-eq-f-Exec-Stream-Init-last*[*rule-format*]:
∀ *c*. *f-Exec-Comp trans-fun xs c* = *last* (*f-Exec-Comp-Stream-Init trans-fun xs c*)
⟨*proof*⟩

lemma *f-Exec-Stream-Init-eq-f-Exec-Stream-Cons*[*rule-format*]:
∀ *c*. *f-Exec-Comp-Stream-Init trans-fun xs c* = *c* # *f-Exec-Comp-Stream trans-fun xs c*
⟨*proof*⟩

corollary *f-Exec-Stream-Init-eq-f-Exec-Stream-Cons-output*:
output-fun c = ε ⟹
map output-fun (*f-Exec-Comp-Stream-Init trans-fun xs c*) =
ε # *map output-fun* (*f-Exec-Comp-Stream trans-fun xs c*)
⟨*proof*⟩

corollary *f-Exec-Stream-Init-tl-eq-f-Exec-Stream*:
tl (*f-Exec-Comp-Stream-Init trans-fun xs c*) = *f-Exec-Comp-Stream trans-fun xs c*
⟨*proof*⟩

lemma *f-Exec-N-eq-last-f-Exec-Stream-Init-take*:
f-Exec-Comp-N trans-fun n xs c =
last (*f-Exec-Comp-Stream-Init trans-fun* (*xs* ↓ *n*) *c*)
⟨*proof*⟩

lemma *f-Exec-Stream-Init-nth*:
n ≤ *length xs* ⟹
f-Exec-Comp-Stream-Init trans-fun xs c ! *n* = *f-Exec-Comp trans-fun* (*xs* ↓ *n*) *c*
⟨*proof*⟩

lemma *f-Exec-Stream-Init-nth-0*: *f-Exec-Comp-Stream-Init trans-fun xs c* ! *0* = *c*
⟨*proof*⟩

lemma *f-Exec-Stream-Init-hd*: *hd* (*f-Exec-Comp-Stream-Init trans-fun xs c*) = *c*
⟨*proof*⟩

lemma *f-Exec-Stream-Init-nth-Suc-eq-f-Exec-Stream-nth*:
f-Exec-Comp-Stream-Init trans-fun xs c ! (*Suc n*) = *f-Exec-Comp-Stream trans-fun xs c* ! *n*
⟨*proof*⟩

lemma *f-Exec-Stream-Init-append*:
f-Exec-Comp-Stream-Init trans-fun (*xs* @ *ys*) *c* =

(*f-Exec-Comp-Stream-Init trans-fun xs c*) @
tl (*f-Exec-Comp-Stream-Init trans-fun ys* (*f-Exec-Comp trans-fun xs c*))
⟨*proof*⟩

corollary *f-Exec-Stream-Init-append-last*:
f-Exec-Comp-Stream-Init trans-fun (*xs* @ *ys*) *c* =
(*f-Exec-Comp-Stream-Init trans-fun xs c*) @
tl (*f-Exec-Comp-Stream-Init trans-fun ys* (*last* (*f-Exec-Comp-Stream-Init trans-fun xs c*)))
⟨*proof*⟩

lemma *f-Exec-Stream-Init-f-Exec-Stream-append*:
f-Exec-Comp-Stream-Init trans-fun (*xs* @ *ys*) *c* =
(*f-Exec-Comp-Stream-Init trans-fun xs c*) @
(*f-Exec-Comp-Stream trans-fun ys* (*f-Exec-Comp trans-fun xs c*))
⟨*proof*⟩

lemma *f-Exec-Stream-Init-take*:
(*f-Exec-Comp-Stream-Init trans-fun xs c*) ↓ *Suc n* =
f-Exec-Comp-Stream-Init trans-fun (*xs* ↓ *n*) *c*
⟨*proof*⟩

lemma *f-Exec-Stream-Init-drop*:
n ≤ *length xs* ⟹
(*f-Exec-Comp-Stream-Init trans-fun xs c*) ↑ *n* =
f-Exec-Comp-Stream-Init trans-fun (*xs* ↑ *n*)
(*f-Exec-Comp trans-fun* (*xs* ↓ *n*) *c*)
⟨*proof*⟩

lemma *f-Exec-Stream-Init-drop-geq-not-valid*:
length xs ≤ *n* ⟹
(*f-Exec-Comp-Stream-Init trans-fun xs c*) ↑ *Suc n* ≠
f-Exec-Comp-Stream-Init trans-fun arbitrary-input arbitrary-comp
⟨*proof*⟩

lemma *i-Exec-Stream-Init-nth*:
i-Exec-Comp-Stream-Init trans-fun input c n = *f-Exec-Comp trans-fun* (*input* ⇓ *n*) *c*
⟨*proof*⟩

lemma *i-Exec-Stream-Init-nth-0*:
i-Exec-Comp-Stream-Init trans-fun input c 0 = *c*
⟨*proof*⟩

lemma *i-Exec-Stream-Init-nth-Suc-eq-i-Exec-Stream-nth*:
i-Exec-Comp-Stream-Init trans-fun input c (*Suc n*) = *i-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*] ⌢ *i-Exec-Comp-Stream trans-fun input c*
⟨*proof*⟩

corollary *i-Exec-Stream-Init-eq-i-Exec-Stream-Cons-output*:
output-fun c = ε ⟹
output-fun ∘ *i-Exec-Comp-Stream-Init trans-fun input c* =
[ε] ⌢ (*output-fun* ∘ *i-Exec-Comp-Stream trans-fun input c*)
⟨*proof*⟩

lemma *i-Exec-Stream-Init-append*:
i-Exec-Comp-Stream-Init trans-fun (*input1* ⌢ *input2*) *c* =
(*f-Exec-Comp-Stream-Init trans-fun input1 c*) ⌢
((*i-Exec-Comp-Stream-Init trans-fun input2* (*f-Exec-Comp trans-fun input1 c*)) ⇑ *Suc 0*)
⟨*proof*⟩

corollary *i-Exec-Stream-Init-append-last*:
i-Exec-Comp-Stream-Init trans-fun (*input1* ⌢ *input2*) *c* =
(*f-Exec-Comp-Stream-Init trans-fun input1 c*) ⌢
((*i-Exec-Comp-Stream-Init trans-fun input2* (*last* (*f-Exec-Comp-Stream-Init trans-fun input1 c*))) ⇑ *Suc 0*)
⟨*proof*⟩

lemma *i-Exec-Stream-Init-i-Exec-Stream-append*:
i-Exec-Comp-Stream-Init trans-fun (*input1* ⌢ *input2*) *c* =
(*f-Exec-Comp-Stream-Init trans-fun input1 c*) ⌢
(*i-Exec-Comp-Stream trans-fun input2* (*f-Exec-Comp trans-fun input1 c*))
⟨*proof*⟩

lemma *i-Exec-Stream-Init-take*:
(*i-Exec-Comp-Stream-Init trans-fun input c*) ⇓ *Suc n* =
f-Exec-Comp-Stream-Init trans-fun (*input* ⇓ *n*) *c*
⟨*proof*⟩
lemma *i-Exec-Stream-Init-drop*:
(*i-Exec-Comp-Stream-Init trans-fun input c*) ⇑ *n* =
i-Exec-Comp-Stream-Init trans-fun (*input* ⇑ *n*)
(*f-Exec-Comp trans-fun* (*input* ⇓ *n*) *c*)
⟨*proof*⟩

theorem *f-Exec-Stream-Init-strictly-causal*:
xs ↓ *n* = *ys* ↓ *n* ⟹
(*f-Exec-Comp-Stream-Init trans-fun xs c*) ↓ *Suc n* = (*f-Exec-Comp-Stream-Init trans-fun ys c*) ↓ *Suc n*
⟨*proof*⟩

theorem *i-Exec-Stream-Init-strictly-causal*:
input1 ⇓ *n* = *input2* ⇓ *n* ⟹

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

theorem *f-Exec-N-eq-f-Exec-Stream-Init-nth*:
n ≤ *length xs* ⟹
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* ⇒ *'value* ⇒ *nat* ⇒ *'value*
where *list-Previous xs init n* ≡
case n of
0 ⇒ *init*
| *Suc n'* ⇒ *xs* ! *n'*

definition *ilist-Previous* :: *'value ilist* ⇒ *'value* ⇒ *nat* ⇒ *'value*
where *ilist-Previous f init n* ≡
case n of
0 ⇒ *init*
| *Suc n'* ⇒ *f n'*

abbreviation *list-Previous'* :: *'value list* ⇒ *'value* ⇒ *nat* ⇒ *'value*
(-$^{\leftarrow''}$ - [*1000*, *10*, *100*] *100*)
where *xs*$^{\leftarrow' \ init}$ *n* ≡ *list-Previous xs init n*

abbreviation *ilist-Previous'* :: *'value ilist* ⇒ *'value* ⇒ *nat* ⇒ *'value*
(-$^{\leftarrow}$ - [*1000*, *10*, *100*] *100*)
where *f*$^{\leftarrow \ init}$ *n* ≡ *ilist-Previous f init n*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *f-Exec-Stream-nth-0*:
$0 < length\ xs \Longrightarrow (f\text{-}Exec\text{-}Comp\text{-}Stream\ trans\text{-}fun\ xs\ c)\ !\ 0 = trans\text{-}fun\ (xs\ !\ 0)\ c$
⟨*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*:
$[\![\ n < length\ xs;\ 0 < n\ ]\!] \Longrightarrow$
f-Exec-Comp-Stream trans-fun xs c ! n =
trans-fun (xs ! n) (f-Exec-Comp-Stream trans-fun xs c ! (n − 1))
⟨*proof*⟩

lemma *f-Exec-Stream-nth-calc-Previous*:
n < *length xs* ⟹
f-Exec-Comp-Stream trans-fun xs c ! *n* =
trans-fun (*xs* ! *n*) ((*f-Exec-Comp-Stream trans-fun xs c*)^←′ *c* *n*)
⟨*proof*⟩

lemma *i-Exec-Stream-nth-0*:
(*i-Exec-Comp-Stream trans-fun input c*) *0* = *trans-fun* (*input 0*) *c*
⟨*proof*⟩

lemma *i-Exec-Stream-nth-gr0-calc*:
0 < *n* ⟹
(*i-Exec-Comp-Stream trans-fun input c*) *n* =
trans-fun (*input n*) ((*i-Exec-Comp-Stream trans-fun input c*) (*n* − *1*))
⟨*proof*⟩

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*:
i-Exec-Comp-Stream trans-fun input c n =
trans-fun (*input n*) ((*i-Exec-Comp-Stream trans-fun input c*)^← *c* *n*)
⟨*proof*⟩

lemma *f-Exec-Stream-Init-nth-Suc-calc*:
n < *length xs* ⟹
f-Exec-Comp-Stream-Init trans-fun xs c ! *Suc n* =
trans-fun (*xs* ! *n*) (*f-Exec-Comp-Stream-Init trans-fun xs c* ! *n*)
⟨*proof*⟩

lemma *f-Exec-Stream-Init-nth-Plus1-calc*:
n < *length xs* ⟹
f-Exec-Comp-Stream-Init trans-fun xs c ! (*n* + *1*)=
trans-fun (*xs* ! *n*) (*f-Exec-Comp-Stream-Init trans-fun xs c* ! *n*)
⟨*proof*⟩

lemma *f-Exec-Stream-Init-nth-gr0-calc*:
⟦ *n* ≤ *length xs*; *0* < *n* ⟧ ⟹
f-Exec-Comp-Stream-Init trans-fun xs c ! *n* =
trans-fun (*xs* ! (*n* − *1*)) (*f-Exec-Comp-Stream-Init trans-fun xs c* ! (*n* − *1*))
⟨*proof*⟩

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*:
i-Exec-Comp-Stream-Init trans-fun input c (Suc n) =
trans-fun (input n) (i-Exec-Comp-Stream-Init trans-fun input c n)
⟨*proof*⟩

lemma *i-Exec-Stream-Init-nth-Plus1-calc*:
i-Exec-Comp-Stream-Init trans-fun input c (n + 1) =
trans-fun (input n) (i-Exec-Comp-Stream-Init trans-fun input c n)
⟨*proof*⟩

lemma *i-Exec-Stream-Init-nth-gr0-calc*:
0 < n ⟹
i-Exec-Comp-Stream-Init trans-fun input c n =
trans-fun (input (n − 1)) (i-Exec-Comp-Stream-Init trans-fun input c (n − 1))
⟨*proof*⟩

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

lemma *f-Exec-Stream-Pre-Post1*:
⟦ n < length xs;
c-n = (f-Exec-Comp-Stream trans-fun xs c)$^{\leftarrow' c}$ *n; x-n = xs ! n ⟧ ⟹*
(P1 x-n ∧ P2 c-n ⟶ Q (f-Exec-Comp-Stream trans-fun xs c ! n)) =
(P1 x-n ∧ P2 c-n ⟶ Q (trans-fun x-n c-n))
⟨*proof*⟩

Direct relation between input and result after transition

lemma *f-Exec-Stream-Pre-Post2*:
⟦ n < length xs;
c-n = (f-Exec-Comp-Stream trans-fun xs c)$^{\leftarrow' c}$ *n; x-n = xs ! n ⟧ ⟹*
(P c-n ⟶ Q (xs ! n) (f-Exec-Comp-Stream trans-fun xs c ! n)) =
(P c-n ⟶ Q x-n (trans-fun x-n c-n))
⟨*proof*⟩

lemma *f-Exec-Stream-Pre-Post2-Suc*:
⟦ Suc n < length xs;
c-n = f-Exec-Comp-Stream trans-fun xs c ! n; x-n1 = xs ! Suc n ⟧ ⟹
(P c-n ⟶ Q (xs ! Suc n) (f-Exec-Comp-Stream trans-fun xs c ! Suc n)) =
(P c-n ⟶ Q x-n1 (trans-fun x-n1 c-n))
⟨*proof*⟩

lemma *f-Exec-Stream-Init-Pre-Post1*:
⟦ n < length xs;
c-n = f-Exec-Comp-Stream-Init trans-fun xs c ! n; x-n = xs ! n ⟧ ⟹
(P1 x-n ∧ P2 c-n ⟶ Q (f-Exec-Comp-Stream-Init trans-fun xs c ! Suc n)) =
(P1 x-n ∧ P2 c-n ⟶ Q (trans-fun x-n c-n))

⟨*proof*⟩

Direct relation between input and state before transition

lemma *f-Exec-Stream-Init-Pre-Post2*:
⟦ *n* < *length xs*;
c-n = *f-Exec-Comp-Stream-Init trans-fun xs c* ! *n*; *x-n* = *xs* ! *n* ⟧ ⟹
(*P* (*xs* ! *n*) (*f-Exec-Comp-Stream-Init trans-fun xs c* ! *n*) ⟶
Q (*f-Exec-Comp-Stream-Init trans-fun xs c* ! *Suc n*)) =
(*P x-n c-n* ⟶ *Q* (*trans-fun x-n c-n*))
⟨*proof*⟩

lemma *i-Exec-Stream-Pre-Post1*:
⟦ *c-n* = (*i-Exec-Comp-Stream trans-fun input c*)^← *c* *n*; *x-n* = *input n* ⟧ ⟹
(*P1 x-n* ∧ *P2 c-n* ⟶ *Q* (*i-Exec-Comp-Stream trans-fun input c n*)) =
(*P1 x-n* ∧ *P2 c-n* ⟶ *Q* (*trans-fun x-n c-n*))
⟨*proof*⟩

Direct relation between input and result after transition

lemma *i-Exec-Stream-Pre-Post2*:
⟦ *c-n* = (*i-Exec-Comp-Stream trans-fun input c*)^← *c* *n*; *x-n* = *input n* ⟧ ⟹
(*P c-n* ⟶ *Q* (*input n*) (*i-Exec-Comp-Stream trans-fun input c n*)) =
(*P c-n* ⟶ *Q x-n* (*trans-fun x-n c-n*))
⟨*proof*⟩

lemma *i-Exec-Stream-Pre-Post2-Suc*:
⟦ *c-n* = *i-Exec-Comp-Stream trans-fun input c n*; *x-n1* = *input* (*Suc n*) ⟧ ⟹
(*P c-n* ⟶ *Q* (*input* (*Suc n*)) (*i-Exec-Comp-Stream trans-fun input c* (*Suc n*)))
=
(*P c-n* ⟶ *Q x-n1* (*trans-fun x-n1 c-n*))
⟨*proof*⟩

lemma *i-Exec-Stream-Init-Pre-Post1*:
⟦ *c-n* = *i-Exec-Comp-Stream-Init trans-fun input c n*; *x-n* = *input n* ⟧ ⟹
(*P1 x-n* ∧ *P2 c-n* ⟶ *Q* (*i-Exec-Comp-Stream-Init trans-fun input c* (*Suc n*))) =
(*P1 x-n* ∧ *P2 c-n* ⟶ *Q* (*trans-fun x-n c-n*))
⟨*proof*⟩

Direct relation between input and state before transition

lemma *i-Exec-Stream-Init-Pre-Post2*:
⟦ *c-n* = *i-Exec-Comp-Stream-Init trans-fun input c n*; *x-n* = *input n* ⟧ ⟹
(*P* (*input n*) (*i-Exec-Comp-Stream-Init trans-fun input c n*) ⟶
Q (*i-Exec-Comp-Stream-Init trans-fun input c* (*Suc n*))) =
(*P x-n c-n* ⟶ *Q* (*trans-fun x-n c-n*))
⟨*proof*⟩

Basic results for stream prefices

lemma *f-Exec-Stream-prefix*:

prefix xs ys $\Longrightarrow$
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* $\Longrightarrow$
f-Exec-Comp-Stream trans-fun xs c $\sqsubseteq$
i-Exec-Comp-Stream trans-fun input c
⟨*proof*⟩

lemma *f-Exec-N-prefix*:
⟦ *n* $\leq$ *length xs*; *prefix xs ys* ⟧ $\Longrightarrow$
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*) $\Longrightarrow$
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 $\Longrightarrow$
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* $\Longrightarrow$
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*) $\Longrightarrow$
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* ::
(*'comp*, *'input*) *Comp-Trans-Fun* $\Rightarrow$ (*'comp*, *'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*:

⟦ *Deterministic-Trans-Fun trans-fun localState*; *localState c1* = *localState c2*; *xs* ≠ [] ⟧ ⟹
f-Exec-Comp trans-fun xs c1 = *f-Exec-Comp trans-fun xs c2*
⟨*proof*⟩

lemma *Deterministic-f-Exec-Stream*:
⟦ *Deterministic-Trans-Fun trans-fun localState*; *localState c1* = *localState c2* ⟧ ⟹
f-Exec-Comp-Stream trans-fun xs c1 = *f-Exec-Comp-Stream trans-fun xs c2*
⟨*proof*⟩

lemma *Deterministic-i-Exec-Stream*:
⟦ *Deterministic-Trans-Fun trans-fun localState*; *localState c1* = *localState c2* ⟧ ⟹
i-Exec-Comp-Stream trans-fun input c1 = *i-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* ⇒ *'a fstream-af* ⇒ *bool*
where *f-Streams-Connected outS inS* ≡ *inS* = *ε* # *outS*

definition *i-Streams-Connected* :: *'a istream-af* ⇒ *'a istream-af* ⇒ *bool*
where *i-Streams-Connected outS inS* ≡ *inS* = [*ε*] ⌢ *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* ⟹ *inS* ≠ []
⟨*proof*⟩

lemma *f-Streams-Connected-nth-conv*:
f-Streams-Connected outS inS =
(*length inS* = *Suc* (*length outS*) ∧
(∀ *i*<*length inS*. *inS* ! *i* = (*case i of 0* ⇒ *ε* | *Suc k* ⇒ *outS* ! *k*)))
⟨*proof*⟩

lemma *f-Streams-Connected-nth-conv-if*:
f-Streams-Connected outS inS =
(*length inS* = *Suc* (*length outS*) ∧
(∀ *i*<*length inS*. *inS* ! *i* = (*if i* = *0 then ε else outS* ! (*i* − *Suc 0*))))
⟨*proof*⟩

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

i-Streams-Connected outS inS =
(∀ i. inS i = (case i of 0 ⇒ ε | Suc k ⇒ outS k))
⟨*proof*⟩

lemma *i-Streams-Connected-nth-conv-if*:
i-Streams-Connected outS inS =
(∀ i. inS i = (if i = 0 then ε else outS (i − Suc 0)))
⟨*proof*⟩

lemma *f-Exec-Stream-Init-eq-output-channel*:
⟦ output-fun c = ε;
f-Streams-Connected
(map output-fun (f-Exec-Comp-Stream trans-fun xs c))
channel ⟧ ⟹
map output-fun (f-Exec-Comp-Stream-Init trans-fun xs c) = channel
⟨*proof*⟩

lemma *i-Exec-Stream-Init-eq-output-channel*:
⟦ output-fun c = ε;
i-Streams-Connected
(output-fun ∘ (i-Exec-Comp-Stream trans-fun input c))
channel ⟧ ⟹
output-fun ∘ (i-Exec-Comp-Stream-Init trans-fun input c) = channel
⟨*proof*⟩

lemma *f-Exec-Stream-output-causal*:
⟦ xs ↓ n = ys ↓ n;
output1 = map output-fun (f-Exec-Comp-Stream trans-fun xs c);
output2 = map output-fun (f-Exec-Comp-Stream trans-fun ys c) ⟧ ⟹
output1 ↓ n = output2 ↓ n
⟨*proof*⟩

lemma *f-Exec-Stream-Init-output-strictly-causal*:
⟦ xs ↓ n = ys ↓ n;
output1 = map output-fun (f-Exec-Comp-Stream-Init trans-fun xs c);
output2 = map output-fun (f-Exec-Comp-Stream-Init trans-fun ys c) ⟧ ⟹
output1 ↓ Suc n = output2 ↓ Suc n
⟨*proof*⟩

lemma *i-Exec-Stream-output-causal*:
⟦ input1 ⇓ n = input2 ⇓ n;
output1 = output-fun ∘ i-Exec-Comp-Stream trans-fun input1 c;
output2 = output-fun ∘ i-Exec-Comp-Stream trans-fun input2 c ⟧ ⟹
output1 ⇓ n = output2 ⇓ n
⟨*proof*⟩

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

⟦ *input1* ⇓ *n* = *input2* ⇓ *n*;
output1 = *output-fun* ∘ *i-Exec-Comp-Stream-Init trans-fun input1 c*;
output2 = *output-fun* ∘ *i-Exec-Comp-Stream-Init trans-fun input2 c* ⟧ ⟹
output1 ⇓ *Suc n* = *output2* ⇓ *Suc n*
⟨*proof*⟩

lemma *f-Exec-Stream-Connected-strictly-causal*:
⟦ *xs* ↓ *n* = *ys* ↓ *n*;
f-Streams-Connected
(*map output-fun* (*f-Exec-Comp-Stream trans-fun xs c*))
channel1;
f-Streams-Connected
(*map output-fun* (*f-Exec-Comp-Stream trans-fun ys c*))
channel2 ⟧ ⟹
channel1 ↓ *Suc n* = *channel2* ↓ *Suc n*
⟨*proof*⟩

lemma *i-Exec-Stream-Connected-strictly-causal*:
⟦ *input1* ⇓ *n* = *input2* ⇓ *n*;
i-Streams-Connected
(*portOutput* ∘ (*i-Exec-Comp-Stream trans-fun input1 c*))
channel1;
i-Streams-Connected
(*portOutput* ∘ (*i-Exec-Comp-Stream trans-fun input2 c*))
channel2 ⟧ ⟹
channel1 ⇓ *Suc n* = *channel2* ⇓ *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* ⇒ *'a fstream-af* ⇒ *bool*
where *f-Streams-Connected-Init outS inS* ≡ *inS* = *outS*

definition *i-Streams-Connected-Init* :: *'a istream-af* ⇒ *'a istream-af* ⇒ *bool*
where *i-Streams-Connected-Init outS inS* ≡ *inS* = *outS*

lemmas *Streams-Connected-Init-defs* =
f-Streams-Connected-Init-def
i-Streams-Connected-Init-def

lemma *f-Streams-Connected-Init-nth-conv*:
f-Streams-Connected-Init outS inS =
(*length inS* = *length outS* ∧ (∀ *i*<*length inS*. *inS* ! *i* = *outS* ! *i*))
⟨*proof*⟩

lemma *i-Streams-Connected-Init-nth-conv*:
i-Streams-Connected-Init outS inS =
(∀ *i*. *inS i* = *outS i*)
⟨*proof*⟩

lemma *f-Exec-Stream-Init-eq-output-channel2*:
⟦ *output-fun c* = *ε*;
f-Streams-Connected-Init
(*map output-fun* (*f-Exec-Comp-Stream-Init trans-fun xs c*))
channel ⟧ ⟹
map output-fun (*f-Exec-Comp-Stream-Init trans-fun xs c*) = *channel*
⟨*proof*⟩
lemma *i-Exec-Stream-Init-eq-output-channel2*:
⟦ *output-fun c* = *ε*;
i-Streams-Connected-Init
(*output-fun* ∘ (*i-Exec-Comp-Stream-Init trans-fun input c*))
channel ⟧ ⟹
output-fun ∘ (*i-Exec-Comp-Stream-Init trans-fun input c*) = *channel*
⟨*proof*⟩

lemma *f-Exec-Stream-Connected-Init-strictly-causal*:
⟦ *xs* ↓ *n* = *ys* ↓ *n*;
f-Streams-Connected-Init
(*map output-fun* (*f-Exec-Comp-Stream-Init trans-fun xs c*))
channel1;
f-Streams-Connected-Init
(*map output-fun* (*f-Exec-Comp-Stream-Init trans-fun ys c*))
channel2 ⟧ ⟹
channel1 ↓ *Suc n* = *channel2* ↓ *Suc n*
⟨*proof*⟩

lemma *i-Exec-Stream-Connected-Init-strictly-causal*:
⟦ *input1* ⇓ *n* = *input2* ⇓ *n*;
i-Streams-Connected-Init
(*portOutput* ∘ (*i-Exec-Comp-Stream-Init trans-fun input1 c*))
channel1;
i-Streams-Connected-Init
(*portOutput* ∘ (*i-Exec-Comp-Stream-Init trans-fun input2 c*))
channel2 ⟧ ⟹
channel1 ⇓ *Suc n* = *channel2* ⇓ *Suc n*
⟨*proof*⟩

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*:
⟦ *n1* ≤ *n2*; *n1* ∈ *I*; *n2* ∈ *I*;
¬ *P* (*f-Exec-Comp trans-fun* (*input* ↓ *n1*) *c*);
P (*f-Exec-Comp trans-fun* (*input* ↓ *n2*) *c*) ⟧ ⟹
∃ *n*∈*I*. *n1* ≤ *n* ∧ *n* < *n2* ∧

$\neg\ P\ (\textit{f-Exec-Comp trans-fun}\ (\textit{input} \downarrow n)\ c) \wedge$
$P\ (\textit{f-Exec-Comp trans-fun}\ (\textit{input} \downarrow (\textit{inext}\ n\ I))\ c)$
⟨*proof*⟩

lemma *f-State-Change-exists*:
$[\![\ \textit{n1} \leq \textit{n2};$
$\neg\ P\ (\textit{f-Exec-Comp trans-fun}\ (\textit{input} \downarrow \textit{n1})\ c);$
$P\ (\textit{f-Exec-Comp trans-fun}\ (\textit{input} \downarrow \textit{n2})\ c)\ ]\!] \Longrightarrow$
$\exists\, n{\geq}\textit{n1}.\ n < \textit{n2} \wedge$
$\neg\ P\ (\textit{f-Exec-Comp trans-fun}\ (\textit{input} \downarrow n)\ c) \wedge$
$P\ (\textit{f-Exec-Comp trans-fun}\ (\textit{input} \downarrow (\textit{Suc}\ n))\ c)$
⟨*proof*⟩

lemma *i-State-Change-exists-set*:
$[\![\ \textit{n1} \leq \textit{n2};\ \textit{n1} \in I;\ \textit{n2} \in I;$
$\neg\ P\ (\textit{i-Exec-Comp-Stream trans-fun input c n1});$
$P\ (\textit{i-Exec-Comp-Stream trans-fun input c n2})\ ]\!] \Longrightarrow$
$\exists\, n{\in}I.\ \textit{n1} \leq n \wedge n < \textit{n2} \wedge$
$\neg\ P\ (\textit{i-Exec-Comp-Stream trans-fun input c}\ n) \wedge$
$P\ (\textit{i-Exec-Comp-Stream trans-fun input c}\ (\textit{inext}\ n\ I))$
⟨*proof*⟩

lemma *i-State-Change-exists*:
$[\![\ \textit{n1} \leq \textit{n2};$
$\neg\ P\ (\textit{i-Exec-Comp-Stream trans-fun input c n1});$
$P\ (\textit{i-Exec-Comp-Stream trans-fun input c n2})\ ]\!] \Longrightarrow$
$\exists\, n{\geq}\textit{n1}.\ n < \textit{n2} \wedge$
$\neg\ P\ (\textit{i-Exec-Comp-Stream trans-fun input c}\ n) \wedge$
$P\ (\textit{i-Exec-Comp-Stream trans-fun input c}\ (\textit{Suc}\ n))$
⟨*proof*⟩

lemma *i-State-Change-Init-exists-set*:
$[\![\ \textit{n1} \leq \textit{n2};\ \textit{n1} \in I;\ \textit{n2} \in I;$
$\neg\ P\ (\textit{i-Exec-Comp-Stream-Init trans-fun input c n1});$
$P\ (\textit{i-Exec-Comp-Stream-Init trans-fun input c n2})\ ]\!] \Longrightarrow$
$\exists\, n{\in}I.\ \textit{n1} \leq n \wedge n < \textit{n2} \wedge$
$\neg\ P\ (\textit{i-Exec-Comp-Stream-Init trans-fun input c}\ n) \wedge$
$P\ (\textit{i-Exec-Comp-Stream-Init trans-fun input c}\ (\textit{inext}\ n\ I))$
⟨*proof*⟩

lemma *i-State-Change-Init-exists*:
$[\![\ \textit{n1} \leq \textit{n2};$
$\neg\ P\ (\textit{i-Exec-Comp-Stream-Init trans-fun input c n1});$
$P\ (\textit{i-Exec-Comp-Stream-Init trans-fun input c n2})\ ]\!] \Longrightarrow$
$\exists\, n{\geq}\textit{n1}.\ n < \textit{n2} \wedge$
$\neg\ P\ (\textit{i-Exec-Comp-Stream-Init trans-fun input c}\ n) \wedge$
$P\ (\textit{i-Exec-Comp-Stream-Init trans-fun input c}\ (\textit{Suc}\ n))$
⟨*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 ⇒
(*'state1* ⇒ *'state2* ⇒ *bool*) ⇒ — Equivalence predicate for local states
(*'comp1*, *'state1*) *Comp-Local-State* ⇒
(*'comp2*, *'state2*) *Comp-Local-State* ⇒
(*'input*, *'input1*) *Port-Input-Value* ⇒ — Input adaptor for first component
(*'input*, *'input2*) *Port-Input-Value* ⇒ — Input adaptor for second component
(*'comp1*, *'output*) *Port-Output-Value* ⇒
(*'comp2*, *'output*) *Port-Output-Value* ⇒
(*'comp1*, *'input1 message-af*) *Comp-Trans-Fun* ⇒
(*'comp2*, *'input2 message-af*) *Comp-Trans-Fun* ⇒
nat ⇒ *nat* ⇒ *'comp1* ⇒ *'comp2* ⇒ *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-states (*localState1 c1*) (*localState2 c2*) ⟶ (
last-message (*map output-fun1* (
f-Exec-Comp-Stream trans-fun1 (*input-fun1 m* # $\varepsilon^{k1 - Suc\ 0}$) *c1*)) =
last-message (*map output-fun2* (
f-Exec-Comp-Stream trans-fun2 (*input-fun2 m* # $\varepsilon^{k2 - Suc\ 0}$) *c2*)) ∧
equiv-states
(*localState1* (*f-Exec-Comp trans-fun1* (*input-fun1 m* # $\varepsilon^{k1 - Suc\ 0}$) *c1*))
(*localState2* (*f-Exec-Comp trans-fun2* (*input-fun2 m* # $\varepsilon^{k2 - Suc\ 0}$) *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 ⇒
(*'state1* ⇒ *'state2* ⇒ *bool*) ⇒ — Equivalence predicate for local states
(*'comp1*, *'state1*) *Comp-Local-State* ⇒
(*'comp2*, *'state2*) *Comp-Local-State* ⇒
(*'input*, *'input1*) *Port-Input-Value* ⇒ — Input adaptor for first component
(*'input*, *'input2*) *Port-Input-Value* ⇒ — Input adaptor for second component
(*'comp1*, *'output*) *Port-Output-Value* ⇒
(*'comp2*, *'output*) *Port-Output-Value* ⇒
(*'comp1*, *'input1 message-af*) *Comp-Trans-Fun* ⇒
(*'comp2*, *'input2 message-af*) *Comp-Trans-Fun* ⇒
nat ⇒ *nat* ⇒ *'comp1* ⇒ *'comp2* ⇒ *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 ≡
∀ *input m*. *set input* ⊆ *A* ∧ *m* ∈ *A* ⟶
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* ⊙$_f$ *k1*) *c1*)
(*f-Exec-Comp trans-fun2* (*map input-fun2 input* ⊙$_f$ *k2*) *c2*)

definition
Equiv-Exec-stable ::
(*'state1* ⇒ *'state2* ⇒ *bool*) ⇒ — Equivalence predicate for local states
(*'comp1*, *'state1*) *Comp-Local-State* ⇒
(*'comp2*, *'state2*) *Comp-Local-State* ⇒
(*'input*, *'input1*) *Port-Input-Value* ⇒ — Input adaptor for first component
(*'input*, *'input2*) *Port-Input-Value* ⇒ — Input adaptor for second component
(*'comp1*, *'output*) *Port-Output-Value* ⇒
(*'comp2*, *'output*) *Port-Output-Value* ⇒
(*'comp1*, *'input1 message-af*) *Comp-Trans-Fun* ⇒
(*'comp2*, *'input2 message-af*) *Comp-Trans-Fun* ⇒
nat ⇒ *nat* ⇒ *'comp1* ⇒ *'comp2* ⇒ *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 ≡
∀ *input m*.
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* ⊙$_f$ *k1*) *c1*)
(*f-Exec-Comp trans-fun2* (*map input-fun2 input* ⊙$_f$ *k2*) *c2*)

lemma *Equiv-Exec-equiv-statesI*:
⟦ *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 ⟧ ⟹
equiv-states
(*localState1* (*f-Exec-Comp trans-fun1* (*input-fun1 m* # $\varepsilon^{k1 - Suc\ 0}$) *c1*))
(*localState2* (*f-Exec-Comp trans-fun2* (*input-fun2 m* # $\varepsilon^{k2 - Suc\ 0}$) *c2*))
⟨*proof*⟩

lemma *Equiv-Exec-output-eqI*:
⟦ *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 ⟧ ⟹
last-message (*map output-fun1* (
f-Exec-Comp-Stream trans-fun1 (*input-fun1 m* # $\varepsilon^{k1 - Suc\ 0}$) *c1*)) =
last-message (*map output-fun2* (
f-Exec-Comp-Stream trans-fun2 (*input-fun2 m* # $\varepsilon^{k2 - Suc\ 0}$) *c2*))
⟨*proof*⟩

lemma *Equiv-Exec-equiv-statesI′*:
⟦ *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 ⟧ ⟹
equiv-states
(*localState1* (*f-Exec-Comp trans-fun1 NoMsg*$^{k1 - Suc\ 0}$ (*trans-fun1* (*input-fun1 m*) *c1*)))
(*localState2* (*f-Exec-Comp trans-fun2 NoMsg*$^{k2 - Suc\ 0}$ (*trans-fun2* (*input-fun2 m*) *c2*)))
⟨*proof*⟩

lemma *Equiv-Exec-le1*:
⟦ *k1* ≤ *Suc 0*; *k2* ≤ *Suc 0*;
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 ⟧ ⟹
output-fun1 (*trans-fun1* (*input-fun1 m*) *c1*) =
output-fun2 (*trans-fun2* (*input-fun2 m*) *c2*) ∧
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 ⊆ *A*; *m* ∈ *A* ⟧ ⟹
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* ⊙$_f$ *k1*) *c1*)
(*f-Exec-Comp trans-fun2* (*map input-fun2 input* ⊙$_f$ *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 ⟹
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* ⊙$_f$ *k1*) *c1*)
(*f-Exec-Comp trans-fun2* (*map input-fun2 input* ⊙$_f$ *k2*) *c2*)
⟨*proof*⟩

Reflexitity, symmetry and transitivity results for *Equiv-Exec*

lemma *Equiv-Exec-refl*:
⟦ ⋀*c*. *equiv-states* (*localState c*) (*localState c*) ⟧ ⟹
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*]:
⟦ ∀ *c1 c2*.
equiv-states (*localState1 c1*) (*localState2 c2*) =

equiv-states (*localState2 c2*) (*localState1 c1*) ⟧ ⟹
Equiv-Exec
m equiv-states
localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2
trans-fun1 trans-fun2 k1 k2 c1 c2 =
Equiv-Exec
m equiv-states
localState2 localState1 input-fun2 input-fun1 output-fun2 output-fun1
trans-fun2 trans-fun1 k2 k1 c2 c1
⟨*proof*⟩

lemma *Equiv-Exec-sym2*:
⟦ *equiv-states-sym* = (λ*s1 s2*. *equiv-states s2 s1*) ⟧ ⟹
Equiv-Exec
m equiv-states
localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2
trans-fun1 trans-fun2 k1 k2 c1 c2 =
Equiv-Exec
m equiv-states-sym
localState2 localState1 input-fun2 input-fun1 output-fun2 output-fun1
trans-fun2 trans-fun1 k2 k1 c2 c1
⟨*proof*⟩

lemma *Equiv-Exec-sym2-ex*:
∃ *equiv-states-sym*.
Equiv-Exec
m equiv-states
localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2
trans-fun1 trans-fun2 k1 k2 c1 c2 =
Equiv-Exec
m equiv-states-sym
localState2 localState1 input-fun2 input-fun1 output-fun2 output-fun1
trans-fun2 trans-fun1 k2 k1 c2 c1
⟨*proof*⟩

lemma *Equiv-Exec-trans*:
⟦ *Equiv-Exec*
m equiv-states12
localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2
trans-fun1 trans-fun2 k1 k2 c1 c2;
Equiv-Exec
m equiv-states23
localState2 localState3 input-fun2 input-fun3 output-fun2 output-fun3
trans-fun2 trans-fun3 k2 k3 c2 c3;
equiv-states13 = (λ*s1 s3*. (
if s1 = *localState1 c1* ∧ *s3* = *localState3 c3 then*
equiv-states12 s1 (*localState2 c2*) ∧
equiv-states23 (*localState2 c2*) *s3*
else

equiv-states12 s1 (
localState2 (*f-Exec-Comp trans-fun2* (*input-fun2 m* # $\varepsilon^{k2 - Suc\ 0}$) *c2*)))
∧
equiv-states23 (
localState2 (*f-Exec-Comp trans-fun2* (*input-fun2 m* # $\varepsilon^{k2 - Suc\ 0}$) *c2*))
s3) ⟧ ⟹
Equiv-Exec
m equiv-states13
localState1 localState3 input-fun1 input-fun3 output-fun1 output-fun3
trans-fun1 trans-fun3 k1 k3 c1 c3
⟨*proof*⟩

lemma *Equiv-Exec-trans-ex*:
⟦ *Equiv-Exec*
m equiv-states12
localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2
trans-fun1 trans-fun2 k1 k2 c1 c2;
Equiv-Exec
m equiv-states23
localState2 localState3 input-fun2 input-fun3 output-fun2 output-fun3
trans-fun2 trans-fun3 k2 k3 c2 c3 ⟧ ⟹
∃ *equiv-states13*. *Equiv-Exec*
m equiv-states13
localState1 localState3 input-fun1 input-fun3 output-fun1 output-fun3
trans-fun1 trans-fun3 k1 k3 c1 c3
⟨*proof*⟩

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 componenents, when the transition function is applied with the same input.

definition *Exec-Equal-State* ::
('*comp*, '*state*) *Comp-Local-State* ⇒ ('*comp*, '*input message-af*) *Comp-Trans-Fun*
⇒ *bool*
where *Exec-Equal-State localState trans-fun* ≡
∀ *c1 c2 m*. *localState c1* = *localState c2* ⟶ *trans-fun m c1* = *trans-fun m c2*

lemma *Exec-Equal-StateD*:
⟦ *Exec-Equal-State localState trans-fun*;
localState c1 = *localState c2* ⟧ ⟹
trans-fun m c1 = *trans-fun m c2*
⟨*proof*⟩

lemma *Exec-Equal-StateD'*:
Exec-Equal-State localState trans-fun ⟹
∀ *c1 c2 m*. *localState c1* = *localState c2* ⟶ *trans-fun m c1* = *trans-fun m c2*
⟨*proof*⟩

lemma *Exec-Equal-StateI*:

$(\bigwedge c1\ c2\ m.\ localState\ c1 = localState\ c2 \Longrightarrow trans\text{-}fun\ m\ c1 = trans\text{-}fun\ m\ c2)$
$\Longrightarrow Exec\text{-}Equal\text{-}State\ localState\ trans\text{-}fun$
⟨*proof*⟩

lemma *f-Exec-Equal-State*: $\bigwedge c1\ c2.$
$[\![\ Exec\text{-}Equal\text{-}State\ localState\ trans\text{-}fun;$
$localState\ c1 = localState\ c2;\ xs \neq []\ ]\!] \Longrightarrow$
$f\text{-}Exec\text{-}Comp\ trans\text{-}fun\ xs\ c1 = f\text{-}Exec\text{-}Comp\ trans\text{-}fun\ xs\ c2$
⟨*proof*⟩

lemma *f-Exec-Stream-Equal-State*:
$[\![\ Exec\text{-}Equal\text{-}State\ localState\ trans\text{-}fun;$
$localState\ c1 = localState\ c2\ ]\!] \Longrightarrow$
$f\text{-}Exec\text{-}Comp\text{-}Stream\ trans\text{-}fun\ xs\ c1 =$
$f\text{-}Exec\text{-}Comp\text{-}Stream\ trans\text{-}fun\ xs\ c2$
⟨*proof*⟩

lemma *i-Exec-Stream-Equal-State*:
$[\![\ Exec\text{-}Equal\text{-}State\ localState\ trans\text{-}fun;$
$localState\ c1 = localState\ c2\ ]\!] \Longrightarrow$
$i\text{-}Exec\text{-}Comp\text{-}Stream\ trans\text{-}fun\ input\ c1 =$
$i\text{-}Exec\text{-}Comp\text{-}Stream\ trans\text{-}fun\ input\ c2$
⟨*proof*⟩

3.2.2 Idle states

definition *State-Idle* ::
$('comp, 'state)\ Comp\text{-}Local\text{-}State \Rightarrow ('comp \Rightarrow 'output\ message\text{-}af) \Rightarrow$
$('comp, 'input\ message\text{-}af)\ Comp\text{-}Trans\text{-}Fun \Rightarrow 'state \Rightarrow bool$
where $State\text{-}Idle\ localState\ output\text{-}fun\ trans\text{-}fun\ state \equiv$
$\forall c.\ localState\ c = state \longrightarrow$
$localState\ (trans\text{-}fun\ \varepsilon\ c) = state \wedge$
$output\text{-}fun\ (trans\text{-}fun\ \varepsilon\ c) = \varepsilon$

lemma *State-IdleD*:
$[\![\ State\text{-}Idle\ localState\ output\text{-}fun\ trans\text{-}fun\ state;$
$localState\ c = state\ ]\!] \Longrightarrow$
$localState\ (trans\text{-}fun\ \varepsilon\ c) = state \wedge$
$output\text{-}fun\ (trans\text{-}fun\ \varepsilon\ c) = \varepsilon$
⟨*proof*⟩

lemma *State-IdleD'*:
$State\text{-}Idle\ localState\ output\text{-}fun\ trans\text{-}fun\ state \Longrightarrow$
$\forall c.\ localState\ c = state \longrightarrow$
$localState\ (trans\text{-}fun\ \varepsilon\ c) = state \wedge$
$output\text{-}fun\ (trans\text{-}fun\ \varepsilon\ c) = \varepsilon$
⟨*proof*⟩

lemma *State-IdleI*:

⟦ ⋀c. *localState* c = *state* ⟹
localState (*trans-fun* ε c) = *state* ∧
output-fun (*trans-fun* ε c) = ε ⟧ ⟹
State-Idle localState output-fun trans-fun state
⟨*proof*⟩

lemma *State-Idle-step*[*rule-format*]:
⟦ *State-Idle localState output-fun trans-fun* (*localState* c) ⟧ ⟹
State-Idle localState output-fun trans-fun (*localState* (*trans-fun* ε c))
⟨*proof*⟩

lemma *f-Exec-State-Idle-replicate-NoMsg-state*[*rule-format*]:
⋀c. *State-Idle localState output-fun trans-fun* (*localState* c) ⟹
localState (*f-Exec-Comp trans-fun* ε^n c) = *localState* c
⟨*proof*⟩

lemma *f-Exec-State-Idle-replicate-NoMsg-gr0-output*[*rule-format*]: ⋀c.
⟦ *State-Idle localState output-fun trans-fun* (*localState* c); 0 < n ⟧ ⟹
output-fun (*f-Exec-Comp trans-fun* ε^n c) = ε
⟨*proof*⟩

lemma *f-Exec-State-Idle-replicate-NoMsg-output*[*rule-format*]:
⟦ *State-Idle localState output-fun trans-fun* (*localState* c);
output-fun c = ε ⟧ ⟹
output-fun (*f-Exec-Comp trans-fun* ε^n c) = ε
⟨*proof*⟩

lemma *f-Exec-Stream-State-Idle-replicate-NoMsg-output*[*rule-format*]:
⟦ *State-Idle localState output-fun trans-fun* (*localState* c) ⟧ ⟹
map output-fun (*f-Exec-Comp-Stream trans-fun* ε^n c) = ε^n
⟨*proof*⟩

corollary *f-Exec-State-Idle-append-replicate-NoMsg-state*:
⟦ *State-Idle localState output-fun trans-fun* (
localState (*f-Exec-Comp trans-fun xs* c)) ⟧ ⟹
localState (*f-Exec-Comp trans-fun* (*xs* @ ε^n) c) =
localState (*f-Exec-Comp trans-fun xs* c)
⟨*proof*⟩

corollary *f-Exec-State-Idle-append-replicate-NoMsg-ge-state*:
⟦ *State-Idle localState output-fun trans-fun* (
localState (*f-Exec-Comp trans-fun* (*xs* @ ε^m) c));
m ≤ n ⟧ ⟹
localState (*f-Exec-Comp trans-fun* (*xs* @ ε^n) c) =
localState (*f-Exec-Comp trans-fun* (*xs* @ ε^m) c)
⟨*proof*⟩

corollary *f-Exec-State-Idle-replicate-NoMsg-ge-state*:

⟦ *State-Idle localState output-fun trans-fun* (
  localState (*f-Exec-Comp trans-fun* ε^m *c*));
 $m \leq n$ ⟧ ⟹
localState (*f-Exec-Comp trans-fun* ε^n *c*) =
localState (*f-Exec-Comp trans-fun* ε^m *c*)
⟨*proof*⟩

corollary *f-Exec-State-Idle-append-replicate-NoMsg-gr0-output*:
⟦ *State-Idle localState output-fun trans-fun* (
  localState (*f-Exec-Comp trans-fun xs c*));
 $0 < n$ ⟧ ⟹
output-fun (*f-Exec-Comp trans-fun* (*xs* @ ε^n) *c*) = ε
⟨*proof*⟩

corollary *f-Exec-Stream-State-Idle-append-replicate-NoMsg-gr0-output*:
⟦ *State-Idle localState output-fun trans-fun* (
  localState (*f-Exec-Comp trans-fun xs c*)) ⟧ ⟹
map output-fun (*f-Exec-Comp-Stream trans-fun* (*xs* @ ε^n) *c*) =
map output-fun (*f-Exec-Comp-Stream trans-fun xs c*) @ ε^n
⟨*proof*⟩

corollary *f-Exec-State-Idle-append-replicate-NoMsg-gr-output*:
⟦ *State-Idle localState output-fun trans-fun* (
  localState (*f-Exec-Comp trans-fun* (*xs* @ ε^m) *c*));
 $m < n$ ⟧ ⟹
output-fun (*f-Exec-Comp trans-fun* (*xs* @ ε^n) *c*) = ε
⟨*proof*⟩

corollary *f-Exec-State-Idle-append-replicate-NoMsg-ge-output*:
⟦ *State-Idle localState output-fun trans-fun* (
  localState (*f-Exec-Comp trans-fun* (*xs* @ ε^m) *c*));
 output-fun (*f-Exec-Comp trans-fun* (*xs* @ ε^m) *c*) = ε; $m \leq n$ ⟧ ⟹
output-fun (*f-Exec-Comp trans-fun* (*xs* @ ε^n) *c*) = ε
⟨*proof*⟩

corollary *f-Exec-State-Idle-replicate-NoMsg-gr-output*:
⟦ *State-Idle localState output-fun trans-fun* (
  localState (*f-Exec-Comp trans-fun* ε^m *c*));
 $m < n$ ⟧ ⟹
output-fun (*f-Exec-Comp trans-fun* ε^n *c*) = ε
⟨*proof*⟩

corollary *f-Exec-State-Idle-replicate-NoMsg-ge-output*:
⟦ *State-Idle localState output-fun trans-fun* (
  localState (*f-Exec-Comp trans-fun* ε^m *c*));
 output-fun (*f-Exec-Comp trans-fun* ε^m *c*) = ε; $m \leq n$ ⟧ ⟹
output-fun (*f-Exec-Comp trans-fun* ε^n *c*) = ε
⟨*proof*⟩

lemma *State-Idle-append-replicate-NoMsg-output-last-message*:
⟦ *State-Idle localState output-fun trans-fun* (
localState (*f-Exec-Comp trans-fun xs c*)) ⟧ ⟹
last-message (*map output-fun* (*f-Exec-Comp-Stream trans-fun* (*xs* @ ε^n) *c*)) =
last-message (*map output-fun* (*f-Exec-Comp-Stream trans-fun xs c*))
⟨*proof*⟩

lemma *State-Idle-append-replicate-NoMsg-output-Msg-eq-last-message*:
⟦ *State-Idle localState output-fun trans-fun* (
localState (*f-Exec-Comp trans-fun xs c*));
output-fun (*f-Exec-Comp trans-fun xs c*) ≠ ε;
xs ≠ [] ⟧ ⟹
last-message (*map output-fun* (*f-Exec-Comp-Stream trans-fun* (*xs* @ ε^n) *c*)) =
output-fun (*f-Exec-Comp trans-fun xs c*)
⟨*proof*⟩

corollary *State-Idle-output-Msg-eq-last-message*:
⟦ *State-Idle localState output-fun trans-fun* (
localState (*f-Exec-Comp trans-fun xs c*));
output-fun (*f-Exec-Comp trans-fun xs c*) ≠ ε;
xs ≠ [] ⟧ ⟹
last-message (*map output-fun* (*f-Exec-Comp-Stream trans-fun xs c*)) =
output-fun (*f-Exec-Comp trans-fun xs c*)
⟨*proof*⟩

lemma *State-Idle-imp-exists-state-change*:
⟦ ¬ *State-Idle localState output-fun trans-fun* (*localState c*);
State-Idle localState output-fun trans-fun (*localState* (*f-Exec-Comp trans-fun* ε^n *c*)) ⟧ ⟹
∃ *i*<*n*. (
¬ *State-Idle localState output-fun trans-fun* (*localState* (*f-Exec-Comp trans-fun* ε^i *c*)) ∧ (
∀ *j*≤*n*. *i* < *j* ⟶ *State-Idle localState output-fun trans-fun* (*localState* (*f-Exec-Comp trans-fun* ε^j *c*))))
⟨*proof*⟩

lemma *State-Idle-imp-exists-state-change2*:
⟦ ¬ *State-Idle localState output-fun trans-fun* (*localState c*);
State-Idle localState output-fun trans-fun (*localState* (*f-Exec-Comp trans-fun* ε^n *c*)) ⟧ ⟹
∃ *i*<*n*. (
(∀ *j*≤*i*. ¬ *State-Idle localState output-fun trans-fun* (*localState* (*f-Exec-Comp trans-fun* ε^i *c*))) ∧
(∀ *j*≤*n*. *i* < *j* ⟶ *State-Idle localState output-fun trans-fun* (*localState* (*f-Exec-Comp trans-fun* ε^j *c*))))
⟨*proof*⟩

3.2.3 Basic definitions for accelerated execution

Stream processing with accelerated components

definition *f-Exec-Comp-Stream-Acc-Output* ::
  nat ⇒ — Acceleration factor
    (*'comp* ⇒ *'output message-af*) ⇒ — Output extraction function
    (*'comp*, *'input message-af*) *Comp-Trans-Fun* ⇒
    'input fstream-af ⇒ *'comp* ⇒
    'output fstream-af
  where *f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun xs c* ≡
    (*map output-fun* (*f-Exec-Comp-Stream trans-fun* (*xs* ⊙$_f$ *k*) *c*)) ÷$_f$ *k*

definition *f-Exec-Comp-Stream-Acc-LocalState* ::
  nat ⇒ — Acceleration factor
    (*'comp* ⇒ *'state*) ⇒ — Local state extraction function
    (*'comp*, *'input message-af*) *Comp-Trans-Fun* ⇒
    'input fstream-af ⇒ *'comp* ⇒
    'state list
  where *f-Exec-Comp-Stream-Acc-LocalState k localState trans-fun xs c* ≡
    (*map localState* (*f-Exec-Comp-Stream trans-fun* (*xs* ⊙$_f$ *k*) *c*)) ÷$_{fl}$ *k*

definition *i-Exec-Comp-Stream-Acc-Output* ::
  nat ⇒ — Acceleration factor
    (*'comp* ⇒ *'output message-af*) ⇒ — Output extraction function
    (*'comp*, *'input message-af*) *Comp-Trans-Fun* ⇒
    'input istream-af ⇒ *'comp* ⇒
    'output istream-af
  where *i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c* ≡
    (*output-fun* ∘ (*i-Exec-Comp-Stream trans-fun* (*input* ⊙$_i$ *k*) *c*)) ÷$_i$ *k*

definition *i-Exec-Comp-Stream-Acc-LocalState* ::
  nat ⇒ — Acceleration factor
    (*'comp* ⇒ *'state*) ⇒ — Local state extraction function
    (*'comp*, *'input message-af*) *Comp-Trans-Fun* ⇒
    'input istream-af ⇒ *'comp* ⇒
    'state ilist
  where *i-Exec-Comp-Stream-Acc-LocalState k localState trans-fun input c* ≡
    (*localState* ∘ (*i-Exec-Comp-Stream trans-fun* (*input* ⊙$_i$ *k*) *c*)) ÷$_{il}$ *k*

definition *f-Exec-Comp-Stream-Acc-Output-Init* ::
  nat ⇒ — Acceleration factor
    (*'comp* ⇒ *'output message-af*) ⇒ — Output extraction function
    (*'comp*, *'input message-af*) *Comp-Trans-Fun* ⇒
    'input fstream-af ⇒ *'comp* ⇒
    'output fstream-af
  where *f-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun xs c* ≡
    (*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 ⇒ — Acceleration factor
('*comp* ⇒ '*state*) ⇒ — Local state extraction function
('*comp*, '*input message-af*) *Comp-Trans-Fun* ⇒ '*input fstream-af* ⇒ '*comp* ⇒
'*state list*
where *f-Exec-Comp-Stream-Acc-LocalState-Init k localState trans-fun xs c* ≡
(*localState c*) # *f-Exec-Comp-Stream-Acc-LocalState k localState trans-fun xs c*

definition *i-Exec-Comp-Stream-Acc-Output-Init* ::
nat ⇒ — Acceleration factor
('*comp* ⇒ '*output message-af*) ⇒ — Output extraction function
('*comp*, '*input message-af*) *Comp-Trans-Fun* ⇒
'*input istream-af* ⇒ '*comp* ⇒
'*output istream-af*
where *i-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun input c* ≡
[*output-fun c*] ⌢ (*i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c*)

definition *i-Exec-Comp-Stream-Acc-LocalState-Init* ::
nat ⇒ — Acceleration factor
('*comp* ⇒ '*state*) ⇒ — Local state extraction function
('*comp*, '*input message-af*) *Comp-Trans-Fun* ⇒
'*input istream-af* ⇒ '*comp* ⇒
'*state ilist*
where *i-Exec-Comp-Stream-Acc-LocalState-Init k localState trans-fun input c* ≡
[*localState c*] ⌢ (*i-Exec-Comp-Stream-Acc-LocalState k localState trans-fun input c*)

lemma *f-Exec-Stream-Acc-Output-length*[*simp*]:
0 < *k* ⟹
length (*f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun xs c*) = *length xs*
⟨*proof*⟩

lemma *f-Exec-Stream-Acc-LocalState-length*[*simp*]:
0 < *k* ⟹
length (*f-Exec-Comp-Stream-Acc-LocalState k localState trans-fun xs c*) = *length xs*
⟨*proof*⟩

lemmas *f-Exec-Stream-Acc-length* =
f-Exec-Stream-Acc-LocalState-length
f-Exec-Stream-Acc-Output-length

3.2.4 Basic results for accelerated execution

lemma *f-Exec-Stream-Acc-Output-Nil*[*simp*]:
f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun [] *c* = []
⟨*proof*⟩

lemma *f-Exec-Stream-Acc-LocalState-Nil*[*simp*]:

f-Exec-Comp-Stream-Acc-LocalState k localState trans-fun [] c = []
⟨*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 = []
⟨*proof*⟩

lemma *f-Exec-Stream-Acc-LocalState-0*[*simp*]:
f-Exec-Comp-Stream-Acc-LocalState 0 localState trans-fun xs c = []
⟨*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 ∘ (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 ∘ (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))* $\longmapsto_f$ *k* $\div_{fl}$ *k*
⟨*proof*⟩

lemma *i-Exec-Stream-Acc-Output-eq-last-message-hold*: *0 < k* $\Longrightarrow$
i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c =
(output-fun ∘ (i-Exec-Comp-Stream trans-fun (input $\odot_i$ *k) c))* $\longmapsto_i$ *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* $\Longrightarrow$
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* $\Longrightarrow$
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* $\Longrightarrow$
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* $\Longrightarrow$
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* $\Longrightarrow$
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* # $\varepsilon^{k - Suc\ 0}$)
c)) #

f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun xs (
f-Exec-Comp trans-fun (x # ε^k − Suc 0) c)
⟨*proof*⟩

lemma *f-Exec-Stream-Acc-Output-one*:
0 < k ⟹
f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun [x] c =
[last-message (map output-fun (f-Exec-Comp-Stream trans-fun (x # ε^k − Suc 0) c))]
⟨*proof*⟩

lemma *f-Exec-Stream-Acc-Output-snoc*:
0 < k ⟹
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 @
[last-message (map output-fun (f-Exec-Comp-Stream trans-fun (x # ε^k − Suc 0) (
f-Exec-Comp trans-fun (xs ⊙_f k) c)))]
⟨*proof*⟩

lemma *i-Exec-Stream-Acc-Output-append*:
i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun (xs ⌢ input) c =
f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun xs c ⌢
i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input (
f-Exec-Comp trans-fun (xs ⊙_f k) c)
⟨*proof*⟩

lemma *i-Exec-Stream-Acc-Output-Cons*:
0 < k ⟹
i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun ([x] ⌢ input) c =
[last-message (map output-fun (f-Exec-Comp-Stream trans-fun (x # ε^k − Suc 0) c))] ⌢
i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input (
f-Exec-Comp trans-fun (x # ε^k − 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 ⊙_f k) c)
⟨*proof*⟩

lemma *f-Exec-Stream-Acc-LocalState-Cons*:
0 < k ⟹
f-Exec-Comp-Stream-Acc-LocalState k localState trans-fun (x # xs) c =
localState (f-Exec-Comp trans-fun (x # ε^k − Suc 0) c) #
f-Exec-Comp-Stream-Acc-LocalState k localState trans-fun xs (
f-Exec-Comp trans-fun (x # ε^k − Suc 0) c)

⟨*proof*⟩

lemma *f-Exec-Stream-Acc-LocalState-one*:
0 < k ⟹
f-Exec-Comp-Stream-Acc-LocalState k localState trans-fun [*x*] *c* =
[*localState* (*f-Exec-Comp trans-fun* (*x* # $\varepsilon^{k - Suc\ 0}$) *c*)]
⟨*proof*⟩

lemma *f-Exec-Stream-Acc-LocalState-snoc*:
0 < k ⟹
f-Exec-Comp-Stream-Acc-LocalState k localState trans-fun (*xs* @ [*x*]) *c* =
f-Exec-Comp-Stream-Acc-LocalState k localState trans-fun xs c @
[*localState* (*f-Exec-Comp trans-fun* ((*xs* @ [*x*]) $\odot_f$ *k*) *c*)]
⟨*proof*⟩

lemma *i-Exec-Stream-Acc-LocalState-append*:
i-Exec-Comp-Stream-Acc-LocalState k localState trans-fun (*xs* ⌢ *input*) *c* =
f-Exec-Comp-Stream-Acc-LocalState k localState trans-fun xs c ⌢
i-Exec-Comp-Stream-Acc-LocalState k localState trans-fun input (
f-Exec-Comp trans-fun (*xs* $\odot_f$ *k*) *c*)
⟨*proof*⟩

lemma *i-Exec-Stream-Acc-LocalState-Cons*:
0 < k ⟹
i-Exec-Comp-Stream-Acc-LocalState k localState trans-fun ([*x*] ⌢ *input*) *c* =
[*localState* (*f-Exec-Comp trans-fun* (*x* # $\varepsilon^{k - Suc\ 0}$) *c*)] ⌢
i-Exec-Comp-Stream-Acc-LocalState k localState trans-fun input (
f-Exec-Comp trans-fun (*x* # $\varepsilon^{k - Suc\ 0}$) *c*)
⟨*proof*⟩

lemma *f-Exec-Stream-Acc-Output-nth*:
⟦ *0 < k*; *n < length xs* ⟧ ⟹
f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun xs c ! *n* =
last-message (*map output-fun* (
f-Exec-Comp-Stream trans-fun (*xs* ! *n* # $\varepsilon^{k - Suc\ 0}$) (
f-Exec-Comp trans-fun (*xs* ↓ *n* $\odot_f$ *k*) *c*)))
⟨*proof*⟩

lemma *f-Exec-Stream-Acc-Output-nth-eq-i-nth*:
⟦ *0 < k*; *n < n'* ⟧ ⟹
f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun (*input* ⇓ *n'*) *c* ! *n* =
i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c n
⟨*proof*⟩

lemma *i-Exec-Stream-Acc-Output-nth*:
0 < k ⟹
i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c n =
last-message (*map output-fun* (
f-Exec-Comp-Stream trans-fun (*input n* # $\varepsilon^{k - Suc\ 0}$) (

f-Exec-Comp trans-fun (input ⇓ n ⊙$_f$ k) c)))
⟨*proof*⟩

corollary *i-Exec-Stream-Acc-Output-nth-f-nth*:
0 < k ⟹
i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c n =
f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun (input ⇓ Suc n) c ! n
⟨*proof*⟩

corollary *i-Exec-Stream-Acc-Output-nth-f-last*:
0 < k ⟹
i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c n =
last (f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun (input ⇓ Suc n) c)
⟨*proof*⟩

lemma *f-Exec-Stream-Acc-LocalState-nth*:
⟦ 0 < k; n < length xs ⟧ ⟹
f-Exec-Comp-Stream-Acc-LocalState k localState trans-fun xs c ! n =
localState (f-Exec-Comp trans-fun (xs ↓ Suc n ⊙$_f$ k) c)
⟨*proof*⟩

lemma *f-Exec-Stream-Acc-LocalState-nth-eq-i-nth*:
⟦ 0 < k; n < n' ⟧ ⟹
f-Exec-Comp-Stream-Acc-LocalState k localState trans-fun (input ⇓ n') c ! n =
i-Exec-Comp-Stream-Acc-LocalState k localState trans-fun input c n
⟨*proof*⟩

corollary *i-Exec-Stream-Acc-LocalState-nth-f-nth*:
0 < k ⟹
i-Exec-Comp-Stream-Acc-LocalState k output-fun trans-fun input c n =
f-Exec-Comp-Stream-Acc-LocalState k output-fun trans-fun (input ⇓ Suc n) c ! n
⟨*proof*⟩

corollary *i-Exec-Stream-Acc-LocalState-nth-f-last*:
0 < k ⟹
i-Exec-Comp-Stream-Acc-LocalState k localState trans-fun input c n =
last (f-Exec-Comp-Stream-Acc-LocalState k localState trans-fun (input ⇓ Suc n) c)
⟨*proof*⟩

lemma *i-Exec-Stream-Acc-LocalState-nth*:
0 < k ⟹
i-Exec-Comp-Stream-Acc-LocalState k localState trans-fun input c n =
localState (f-Exec-Comp trans-fun (input ⇓ Suc n ⊙$_f$ k) c)
⟨*proof*⟩

lemma *f-Exec-Stream-Acc-Output-causal*:
xs ↓ n = ys ↓ n ⟹
f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun xs c ↓ n =

f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun ys c ↓ n
⟨*proof*⟩

lemma *i-Exec-Stream-Acc-Output-causal*:
input1 ⇓ n = input2 ⇓ n ⟹
i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input1 c ⇓ n =
i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input2 c ⇓ n
⟨*proof*⟩

lemma *f-Exec-Stream-Acc-Output-Connected-strictly-causal*:
⟦ *xs ↓ n = ys ↓ 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 ⟧ *⟹*
channel1 ↓ Suc n = channel2 ↓ Suc n
⟨*proof*⟩

lemma *i-Exec-Stream-Acc-Output-Connected-strictly-causal*:
⟦ *input1 ⇓ n = input2 ⇓ 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 ⟧ *⟹*
channel1 ⇓ Suc n = channel2 ⇓ Suc n
⟨*proof*⟩

Complete execution cycles/steps of accelrated execution

definition *Acc-Trans-Fun-Step* ::
nat ⇒ — Acceleration factor
(*'comp*, *'input message-af*) *Comp-Trans-Fun ⇒*
(*'comp list ⇒ 'comp*) *⇒* — Pointwise output shrink function
'input message-af ⇒ 'comp ⇒
'comp
where *Acc-Trans-Fun-Step k trans-fun pointwise-shrink x c ≡*
pointwise-shrink (*f-Exec-Comp-Stream trans-fun* (*x # ε*$^{k - Suc\ 0}$) *c*)

definition *is-Pointwise-Output-Shrink* ::
(*'comp list ⇒ 'comp*) *⇒* — Pointwise output shrink function
(*'comp ⇒ 'output message-af*) *⇒* — Output extraction function for consideration
bool
where *is-Pointwise-Output-Shrink pointwise-shrink output-fun ≡*
∀ cs. output-fun (*pointwise-shrink cs*) *= last-message* (*map output-fun cs*)

primrec *is-Pointwise-Output-Shrink-list* ::

(*'comp list* ⇒ *'comp*) ⇒ — Pointwise output shrink function
(*'comp* ⇒ *'output message-af*) *list* ⇒ — List of output extraction functions for consideration
bool
where
is-Pointwise-Output-Shrink-list pointwise-shrink [] = *True*
| *is-Pointwise-Output-Shrink-list pointwise-shrink* (*f* # *fs*) =
(*is-Pointwise-Output-Shrink pointwise-shrink f* ∧
is-Pointwise-Output-Shrink-list pointwise-shrink fs)

definition *is-correct-localState-Pointwise-Output-Shrink* ::
(*'comp list* ⇒ *'comp*) ⇒ — Pointwise output shrink function
(*'comp* ⇒ *'state*) ⇒ — Local state extraction function
bool
where *is-correct-localState-Pointwise-Output-Shrink pointwise-shrink localState* ≡
∀ *cs*. *cs* ≠ [] ⟶ *localState* (*pointwise-shrink cs*) = *localState* (*last cs*)

lemma *Deterministic-trans-fun-imp-acc-trans-fun*:
Deterministic-Trans-Fun trans-fun localState ⟹
Deterministic-Trans-Fun (*Acc-Trans-Fun-Step k trans-fun pointwise-shrink*) *localState*
⟨*proof*⟩

lemma *is-Pointwise-Output-Shrink-list-imp-is-Pointwise-Output-Shrink*:
⟦ *is-Pointwise-Output-Shrink-list pointwise-shrink fs*; *output-fun* ∈ *set fs* ⟧ ⟹
is-Pointwise-Output-Shrink pointwise-shrink output-fun
⟨*proof*⟩

lemma *is-Pointwise-Output-Shrink-list-eq-is-Pointwise-Output-Shrink-all*:
(*is-Pointwise-Output-Shrink-list pointwise-shrink fs*) =
(∀ *output-fun* ∈ *set fs*. *is-Pointwise-Output-Shrink pointwise-shrink output-fun*)
⟨*proof*⟩

lemma *is-Pointwise-Output-Shrink-subset*:
⟦ *is-Pointwise-Output-Shrink-list pointwise-shrink fs*; *set fs'* ⊆ *set fs* ⟧ ⟹
is-Pointwise-Output-Shrink-list pointwise-shrink fs'
⟨*proof*⟩

lemma *f-Exec-Stream-Acc-LocalState-eq-Acc-Trans-Fun-Step-LocalState*: ⋀*c*.
⟦ *0* < *k*;
Deterministic-Trans-Fun trans-fun localState;
is-correct-localState-Pointwise-Output-Shrink pointwise-shrink localState ⟧ ⟹
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*)
⟨*proof*⟩

lemma *f-Exec-Stream-Acc-Output-eq-Acc-Trans-Fun-Step-Output*: ⋀*c*.

⟦ *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 ⟧ ⟹
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 pointwise-shrink*) *xs c*)
⟨*proof*⟩

lemma *i-Exec-Stream-Acc-LocalState-eq-Acc-Trans-Fun-Step-LocalState*: ⋀*c*.
⟦ *0 < k*;
Deterministic-Trans-Fun trans-fun localState;
is-correct-localState-Pointwise-Output-Shrink pointwise-shrink localState ⟧ ⟹
i-Exec-Comp-Stream-Acc-LocalState k localState trans-fun input c =
localState ∘ (*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*: ⋀*c*.
⟦ *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 ⟧ ⟹
i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c =
output-fun ∘ (*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 ⟹
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 ⟹
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-Exec-Comp-Stream-Acc-Output-Init (*Suc 0*) *output-fun trans-fun xs c* =
map output-fun (*f-Exec-Comp-Stream-Init trans-fun xs c*)
⟨*proof*⟩

lemma *f-Exec-Stream-Acc-LocalState-Init-1*:
f-Exec-Comp-Stream-Acc-LocalState-Init (*Suc 0*) *localState trans-fun xs c* =
map localState (*f-Exec-Comp-Stream-Init trans-fun xs c*)
⟨*proof*⟩

lemma *i-Exec-Stream-Acc-Output-Init-1*:
i-Exec-Comp-Stream-Acc-Output-Init (*Suc 0*) *output-fun trans-fun input c* =
output-fun ∘ (*i-Exec-Comp-Stream-Init trans-fun input c*)
⟨*proof*⟩

lemma *i-Exec-Stream-Acc-LocalState-Init-1*:
i-Exec-Comp-Stream-Acc-LocalState-Init (*Suc 0*) *localState trans-fun input c* =
localState ∘ (*i-Exec-Comp-Stream-Init trans-fun input c*)
⟨*proof*⟩

lemma *f-Exec-Stream-Acc-Output-Init-take*:
f-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun xs c ↓ (*Suc n*) =
f-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun (*xs* ↓ *n*) *c*
⟨*proof*⟩

lemma *f-Exec-Stream-Acc-Output-Init-drop′*:
⟦ *0* < *k*; *n* < *length xs* ⟧ ⟹
f-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun xs c ↑ *Suc n* =
f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun xs c ↑ *n*
⟨*proof*⟩

lemma *i-Exec-Stream-Acc-Output-Init-take*:
0 < *k* ⟹
i-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun input c ⇓ (*Suc n*) =
f-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun (*input* ⇓ *n*) *c*
⟨*proof*⟩

lemma *i-Exec-Stream-Acc-Output-Init-drop′*:
0 < *k* ⟹
i-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun xs c ⇑ *Suc n* =
i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun xs c ⇑ *n*
⟨*proof*⟩

lemma *f-Exec-Stream-Acc-Output-Init-strictly-causal*:
xs ↓ *n* = *ys* ↓ *n* ⟹

f-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun xs c ↓ *Suc n* =
f-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun ys c ↓ *Suc n*
⟨*proof*⟩

lemma *i-Exec-Stream-Acc-Output-Init-strictly-causal*:
input1 ⇓ *n* = *input2* ⇓ *n* ⟹
i-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun input1 c ⇓ *Suc n* =
i-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun input2 c ⇓ *Suc n*
⟨*proof*⟩

lemma *f-Exec-Stream-Acc-Output-Init-eq-f-Exec-Stream-Acc-Output-Cons*:
f-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun xs c =
output-fun c # *f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun xs c*
⟨*proof*⟩

lemma *f-Exec-Stream-Acc-Output-Init-eq-f-Exec-Stream-Acc-Output-Cons-output*:

output-fun c = *ε* ⟹
f-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun xs c =
ε # *f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun xs c*
⟨*proof*⟩

lemma *f-Exec-Stream--Acc-OutputInit-tl-eq-f-Exec-Stream-Acc-Output*:
tl (*f-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun xs c*) =
f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun xs c
⟨*proof*⟩

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

lemma *f-Exec-Stream-Acc-Output-Init-eq-output-channel*:
⟦ *output-fun c* = *ε*;
f-Streams-Connected
(*f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun xs c*)
channel ⟧ ⟹
f-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun xs c = *channel*
⟨*proof*⟩

lemma *i-Exec-Stream-Acc-Output-Init-eq-i-Exec-Stream-Acc-Output-Cons*:
i-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun input c =
[*output-fun c*] ⌢ *i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c*
⟨*proof*⟩

lemma *i-Exec-Stream-Acc-Output-Init-eq-i-Exec-Stream-Acc-Output-Cons-output*:

output-fun c = *ε* ⟹

i-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun input c =
[ε] ⌢ i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c
⟨*proof*⟩

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-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c)$^{\leftarrow\ output\text{-}fun\ c\ n}$
⟨*proof*⟩

lemma *i-Exec-Stream-Acc-Output-Init-eq-output-channel*:
⟦ *output-fun c = ε*;
i-Streams-Connected
(i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c)
channel ⟧ ⟹
i-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun input c = channel
⟨*proof*⟩

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*]:
⋀*c1 c2 i.* ⟦
0 < k1; *0 < k2*;
equiv-states (*localState1 c1*) (*localState2 c2*);
∀ *input0. set input0 ⊆ A* ⟶ (∀ *m*∈*A.*
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 input0* ⊙$_f$ *k1*) *c1*)
(*f-Exec-Comp trans-fun2* (*map input-fun2 input0* ⊙$_f$ *k2*) *c2*));
— *equiv-states* relation implies equivalent executions
— not only at the beginning but also after processing an input
set input ⊆ A; *i < length input* ⟧ ⟹
equiv-states
(*localState1* ((*f-Exec-Comp-Stream trans-fun1* (*map input-fun1 input* ⊙$_f$ *k1*) *c1*) ÷$_{fl}$ *k1* ! *i*))
(*localState2* ((*f-Exec-Comp-Stream trans-fun2* (*map input-fun2 input* ⊙$_f$ *k2*) *c2*) ÷$_{fl}$ *k2* ! *i*))
⟨*proof*⟩

corollary *f-Equiv-Exec-Stream-expand-shrink-equiv-state*:
⟦ *0 < k1*; *0 < k2*;
equiv-states (*localState1 c1*) (*localState2 c2*);
⋀*input0 m. 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 input0* $\odot_f$ *k1*) *c1*)
(*f-Exec-Comp trans-fun2* (*map input-fun2 input0* $\odot_f$ *k2*) *c2*);
i<*length input* ⟧ ⟹
equiv-states
(*localState1* ((*f-Exec-Comp-Stream trans-fun1* (*map input-fun1 input* $\odot_f$ *k1*) *c1*) $\div_{fl}$ *k1* ! *i*))
(*localState2* ((*f-Exec-Comp-Stream trans-fun2* (*map input-fun2 input* $\odot_f$ *k2*) *c2*) $\div_{fl}$ *k2* ! *i*))
⟨*proof*⟩

lemma *f-Equiv-Exec-expand-shrink-equiv-state-set*:
⟦ *0* < *k1*; *0* < *k2*; *equiv-states* (*localState1 c1*) (*localState2 c2*);
⋀*input0 m*. ⟦*set input0* ⊆ *A*; *m* ∈ *A*⟧ ⟹
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 input0* $\odot_f$ *k1*) *c1*)
(*f-Exec-Comp trans-fun2* (*map input-fun2 input0* $\odot_f$ *k2*) *c2*);
set input ⊆ *A* ⟧ ⟹
equiv-states
(*localState1* (*f-Exec-Comp trans-fun1* (*map input-fun1 input* $\odot_f$ *k1*) *c1*))
(*localState2* (*f-Exec-Comp trans-fun2* (*map input-fun2 input* $\odot_f$ *k2*) *c2*))
⟨*proof*⟩

lemma *f-Equiv-Exec-expand-shrink-equiv-state*:
⟦ *0* < *k1*; *0* < *k2*; *equiv-states* (*localState1 c1*) (*localState2 c2*);
⋀*input0 m*.
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 input0* $\odot_f$ *k1*) *c1*)
(*f-Exec-Comp trans-fun2* (*map input-fun2 input0* $\odot_f$ *k2*) *c2*) ⟧ ⟹
equiv-states
(*localState1* (*f-Exec-Comp trans-fun1* (*map input-fun1 input* $\odot_f$ *k1*) *c1*))
(*localState2* (*f-Exec-Comp trans-fun2* (*map input-fun2 input* $\odot_f$ *k2*) *c2*))
⟨*proof*⟩

lemma *i-Equiv-Exec-Stream-expand-shrink-equiv-state-set*[*rule-format*]:
⟦ *0* < *k1*; *0* < *k2*; *equiv-states* (*localState1 c1*) (*localState2 c2*);
⋀*input0 m*. ⟦*set input0* ⊆ *A*; *m* ∈ *A*⟧ ⟹
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 input0* $\odot_f$ *k1*) *c1*)
(*f-Exec-Comp trans-fun2* (*map input-fun2 input0* $\odot_f$ *k2*) *c2*);
range input ⊆ *A* ⟧ ⟹

equiv-states
(*localState1* ((*i-Exec-Comp-Stream trans-fun1* ((*input-fun1* ∘ *input*) ⊙$_i$ *k1*) *c1* ÷$_{il}$ *k1*) *i*))
(*localState2* ((*i-Exec-Comp-Stream trans-fun2* ((*input-fun2* ∘ *input*) ⊙$_i$ *k2*) *c2* ÷$_{il}$ *k2*) *i*))
⟨*proof*⟩

lemma *i-Equiv-Exec-Stream-expand-shrink-equiv-state*:
⟦ *0* < *k1*; *0* < *k2*; *equiv-states* (*localState1 c1*) (*localState2 c2*);
⋀*input0 m*.
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 input0* ⊙$_f$ *k1*) *c1*)
(*f-Exec-Comp trans-fun2* (*map input-fun2 input0* ⊙$_f$ *k2*) *c2*) ⟧ ⟹
equiv-states
(*localState1* ((*i-Exec-Comp-Stream trans-fun1* ((*input-fun1* ∘ *input*) ⊙$_i$ *k1*) *c1* ÷$_{il}$ *k1*) *i*))
(*localState2* ((*i-Exec-Comp-Stream trans-fun2* ((*input-fun2* ∘ *input*) ⊙$_i$ *k2*) *c2* ÷$_{il}$ *k2*) *i*))
⟨*proof*⟩

lemma *f-Equiv-Exec-Stream-expand-shrink-output-set-eq*:
⟦ *0* < *k1*; *0* < *k2*;
equiv-states (*localState1 c1*) (*localState2 c2*);
⋀*input0 m*. ⟦ *set input0* ⊆ *A*; *m* ∈ *A* ⟧ ⟹
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 input0* ⊙$_f$ *k1*) *c1*)
(*f-Exec-Comp trans-fun2* (*map input-fun2 input0* ⊙$_f$ *k2*) *c2*);
set input ⊆ *A* ⟧ ⟹
(*map output-fun1* (
f-Exec-Comp-Stream trans-fun1 (*map input-fun1 input* ⊙$_f$ *k1*) *c1*)) ÷$_f$ *k1* =
(*map output-fun2* (
f-Exec-Comp-Stream trans-fun2 (*map input-fun2 input* ⊙$_f$ *k2*) *c2*)) ÷$_f$ *k2*
⟨*proof*⟩

lemma *f-Equiv-Exec-Stream-expand-shrink-output-eq*:
⟦ *0* < *k1*; *0* < *k2*;
equiv-states (*localState1 c1*) (*localState2 c2*);
⋀*input0 m*.
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 input0* ⊙$_f$ *k1*) *c1*)
(*f-Exec-Comp trans-fun2* (*map input-fun2 input0* ⊙$_f$ *k2*) *c2*) ⟧ ⟹
(*map output-fun1* (

f-Exec-Comp-Stream trans-fun1 (map input-fun1 input $\odot_f$ *k1) c1)) ÷*$_f$ *k1 =*
(map output-fun2 (
f-Exec-Comp-Stream trans-fun2 (map input-fun2 input $\odot_f$ *k2) c2)) ÷*$_f$ *k2*
⟨*proof*⟩

lemma *i-Equiv-Exec-Stream-expand-shrink-output-set-eq*:
⟦ *0 < k1*; *0 < k2*;
equiv-states (localState1 c1) (localState2 c2);
⋀*input0 m.* ⟦ *set input0 ⊆ A*; *m ∈ A* ⟧ ⟹
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 input0 $\odot_f$ *k1) c1)*
(f-Exec-Comp trans-fun2 (map input-fun2 input0 $\odot_f$ *k2) c2)*;
range input ⊆ A ⟧ ⟹
(output-fun1 ∘
i-Exec-Comp-Stream trans-fun1 ((input-fun1 ∘ input) $\odot_i$ *k1) c1) ÷*$_i$ *k1 =*
(output-fun2 ∘
i-Exec-Comp-Stream trans-fun2 ((input-fun2 ∘ input) $\odot_i$ *k2) c2) ÷*$_i$ *k2*
⟨*proof*⟩

lemma *i-Equiv-Exec-Stream-expand-shrink-output-eq*:
⟦ *0 < k1*; *0 < k2*;
equiv-states (localState1 c1) (localState2 c2);
⋀*input0 m.*
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 input0 $\odot_f$ *k1) c1)*
(f-Exec-Comp trans-fun2 (map input-fun2 input0 $\odot_f$ *k2) c2)* ⟧ ⟹
(output-fun1 ∘
i-Exec-Comp-Stream trans-fun1 ((input-fun1 ∘ input) $\odot_i$ *k1) c1) ÷*$_i$ *k1 =*
(output-fun2 ∘
i-Exec-Comp-Stream trans-fun2 ((input-fun2 ∘ input) $\odot_i$ *k2) c2) ÷*$_i$ *k2*
⟨*proof*⟩

lemma *f-Equiv-Exec-Stream-Acc-LocalState-set*:
⟦ *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;
— *equiv-states* relation implies equivalent executions
— not only at the beginning but also after processing an input
set input ⊆ A;
i < length input ⟧ ⟹

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 < *length input* ⟧ ⟹
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;
set input ⊆ *A* ⟧ ⟹
f-Exec-Comp-Stream-Acc-Output k1 output-fun1 trans-fun1 (*map input-fun1 input*) *c1* =
f-Exec-Comp-Stream-Acc-Output k2 output-fun2 trans-fun2 (*map input-fun2 input*) *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 ⟧ ⟹
f-Exec-Comp-Stream-Acc-Output k1 output-fun1 trans-fun1 (*map input-fun1 input*) *c1* =
f-Exec-Comp-Stream-Acc-Output k2 output-fun2 trans-fun2 (*map input-fun2 input*) *c2*

⟨*proof*⟩

lemma *i-Equiv-Exec-Stream-Acc-LocalState-set*:
⟦ *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;
range input ⊆ *A* ⟧ ⟹
equiv-states
(*i-Exec-Comp-Stream-Acc-LocalState k1 localState1 trans-fun1* (*input-fun1* ∘ *input*) *c1 i*)
(*i-Exec-Comp-Stream-Acc-LocalState k2 localState2 trans-fun2* (*input-fun2* ∘ *input*) *c2 i*)
⟨*proof*⟩

lemma *i-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
(*i-Exec-Comp-Stream-Acc-LocalState k1 localState1 trans-fun1* (*input-fun1* ∘ *input*) *c1 i*)
(*i-Exec-Comp-Stream-Acc-LocalState k2 localState2 trans-fun2* (*input-fun2* ∘ *input*) *c2 i*)
⟨*proof*⟩

lemma *i-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;
range input ⊆ *A* ⟧ ⟹
i-Exec-Comp-Stream-Acc-Output k1 output-fun1 trans-fun1 (*input-fun1* ∘ *input*) *c1* =
i-Exec-Comp-Stream-Acc-Output k2 output-fun2 trans-fun2 (*input-fun2* ∘ *input*) *c2*
⟨*proof*⟩

lemma *i-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 ⟧ ⟹
i-Exec-Comp-Stream-Acc-Output k1 output-fun1 trans-fun1 (*input-fun1* ∘ *input*) *c1* =
i-Exec-Comp-Stream-Acc-Output k2 output-fun2 trans-fun2 (*input-fun2* ∘ *input*) *c2*
⟨*proof*⟩

3.2.7 Idle states and accelerated execution

lemma *f-Exec-Stream-Acc-LocalState--State-Idle-nth*[*rule-format*]:
⋀*c i.*
⟦ *0* < *l*; *l* ≤ *k*; *Exec-Equal-State localState trans-fun*;
∀ *n*≤*i. State-Idle localState output-fun trans-fun* (
f-Exec-Comp-Stream-Acc-LocalState l localState trans-fun xs c ! *n*);
i < *length xs* ⟧ ⟹
f-Exec-Comp-Stream-Acc-LocalState k localState trans-fun xs c ! *i* =
f-Exec-Comp-Stream-Acc-LocalState l localState trans-fun xs c ! *i*
⟨*proof*⟩

corollary *f-Exec-Stream-Acc-LocalState--State-Idle-eq*[*rule-format*]:
⟦ *0* < *l*; *l* ≤ *k*; *Exec-Equal-State localState trans-fun*;
∀ *n*<*length xs. State-Idle localState output-fun trans-fun* (
f-Exec-Comp-Stream-Acc-LocalState l localState trans-fun xs c ! *n*) ⟧ ⟹
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-nth*[*rule-format*]:
⟦ *0* < *l*; *l* ≤ *k*; *Exec-Equal-State localState trans-fun*;
∀ *n*≤*i. State-Idle localState output-fun trans-fun* (
i-Exec-Comp-Stream-Acc-LocalState l localState trans-fun input c n) ⟧ ⟹
i-Exec-Comp-Stream-Acc-LocalState k localState trans-fun input c i =
i-Exec-Comp-Stream-Acc-LocalState l localState trans-fun input c i
⟨*proof*⟩

corollary *i-Exec-Stream-Acc-LocalState--State-Idle-eq*[*rule-format*]:
⟦ *0* < *l*; *l* ≤ *k*; *Exec-Equal-State localState trans-fun*;
∀ *n. State-Idle localState output-fun trans-fun* (
i-Exec-Comp-Stream-Acc-LocalState l localState trans-fun input c n) ⟧ ⟹
i-Exec-Comp-Stream-Acc-LocalState k localState trans-fun input c =
i-Exec-Comp-Stream-Acc-LocalState l localState trans-fun input c
⟨*proof*⟩

lemma *f-Exec-Stream-Acc-Output--State-Idle-nth*[*rule-format*]:
⟦ *0* < *l*; *l* ≤ *k*; *Exec-Equal-State localState trans-fun*;
∀ *n*≤*i. State-Idle localState output-fun trans-fun* (

f-Exec-Comp-Stream-Acc-LocalState l localState trans-fun xs c ! n);
i < length xs ⟧ ⟹
f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun xs c ! i =
f-Exec-Comp-Stream-Acc-Output l output-fun trans-fun xs c ! i
⟨*proof*⟩

lemma *f-Exec-Stream-Acc-Output--State-Idle-eq*[*rule-format*]:
⟦ *0 < l*; *l ≤ k*; *Exec-Equal-State localState trans-fun*;
∀ *n*<*length xs. State-Idle localState output-fun trans-fun* (
f-Exec-Comp-Stream-Acc-LocalState l localState trans-fun xs c ! n) ⟧ ⟹
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-nth*[*rule-format*]:
⟦ *0 < l*; *l ≤ k*; *Exec-Equal-State localState trans-fun*;
∀ *n*≤*i. State-Idle localState output-fun trans-fun* (
i-Exec-Comp-Stream-Acc-LocalState l localState trans-fun input c n) ⟧ ⟹
i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c i =
i-Exec-Comp-Stream-Acc-Output l output-fun trans-fun input c i
⟨*proof*⟩

lemma *i-Exec-Stream-Acc-Output--State-Idle-eq*[*rule-format*]:
⟦ *0 < l*; *l ≤ k*; *Exec-Equal-State localState trans-fun*;
∀ *n. State-Idle localState output-fun trans-fun* (
i-Exec-Comp-Stream-Acc-LocalState l localState trans-fun input c n) ⟧ ⟹
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*⟩

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-Exec-Stream-Acc-LocalState--State-Idle-all*[*rule-format*]:
⋀*c xs.* ⟦ *0 < l*; *l ≤ k*;
State-Idle localState output-fun trans-fun (*localState c*);
∀ *c m. State-Idle localState output-fun trans-fun* (*localState c*) ⟶
State-Idle localState output-fun trans-fun (
localState (*f-Exec-Comp trans-fun* (*m # ε*$^{l - Suc\ 0}$) *c*));
i < length xs ⟧ ⟹
State-Idle localState output-fun trans-fun (
f-Exec-Comp-Stream-Acc-LocalState k localState trans-fun xs c ! i)
⟨*proof*⟩

lemma *i-Exec-Stream-Acc-LocalState--State-Idle-all*[*rule-format*]:
⟦ *0 < l*; *l ≤ k*;
State-Idle localState output-fun trans-fun (*localState c*);
∀ *c m. State-Idle localState output-fun trans-fun* (*localState c*) ⟶
State-Idle localState output-fun trans-fun (

localState (*f-Exec-Comp trans-fun* (*m* # $\varepsilon^{l\ -\ Suc\ 0}$) *c*)) ⟧ ⟹
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*]:
⟦ *0* < *l*; *l* ≤ *k*; *Exec-Equal-State localState trans-fun*;
State-Idle localState output-fun trans-fun (*localState c*);
∀ *c m*. *State-Idle localState output-fun trans-fun* (*localState c*) ⟶
State-Idle localState output-fun trans-fun (
localState (*f-Exec-Comp trans-fun* (*m* # $\varepsilon^{l\ -\ Suc\ 0}$) *c*)) ⟧ ⟹
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*]:
⟦ *0* < *l*; *l* ≤ *k*; *Exec-Equal-State localState trans-fun*;
State-Idle localState output-fun trans-fun (*localState c*);
∀ *c m*. *State-Idle localState output-fun trans-fun* (*localState c*) ⟶
State-Idle localState output-fun trans-fun (
localState (*f-Exec-Comp trans-fun* (*m* # $\varepsilon^{l\ -\ Suc\ 0}$) *c*)) ⟧ ⟹
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*]:
⟦ *0* < *l*; *l* ≤ *k*; *Exec-Equal-State localState trans-fun*;
State-Idle localState output-fun trans-fun (*localState c*);
∀ *c m*. *State-Idle localState output-fun trans-fun* (*localState c*) ⟶
State-Idle localState output-fun trans-fun (
localState (*f-Exec-Comp trans-fun* (*m* # $\varepsilon^{l\ -\ Suc\ 0}$) *c*)) ⟧ ⟹
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*]:
⟦ *0* < *l*; *l* ≤ *k*; *Exec-Equal-State localState trans-fun*;
State-Idle localState output-fun trans-fun (*localState c*);
∀ *c m*. *State-Idle localState output-fun trans-fun* (*localState c*) ⟶
State-Idle localState output-fun trans-fun (
localState (*f-Exec-Comp trans-fun* (*m* # $\varepsilon^{l\ -\ Suc\ 0}$) *c*)) ⟧ ⟹
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*: ⋀*c*.
f-Exec-Comp trans-fun (*map f xs*) *c* = *f-Exec-Comp* (*trans-fun* ∘ *f*) *xs c*
⟨*proof*⟩

lemma *f-Exec-Stream-input-map*:
f-Exec-Comp-Stream trans-fun (*map f xs*) *c* =
f-Exec-Comp-Stream (*trans-fun* ∘ *f*) *xs c*
⟨*proof*⟩
lemma *i-Exec-Stream-input-map*:
i-Exec-Comp-Stream trans-fun (*f* ∘ *input*) *c* =
i-Exec-Comp-Stream (*trans-fun* ∘ *f*) *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* ⇒ *nat* ⇒ *iT* ⇒ *'a list*
where
f-join-aux [] *n I* = []
| *f-join-aux* (*x* # *xs*) *n I* =
(*if n* ∈ *I then* [*x*] *else* []) @ *f-join-aux xs* (*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-join* [*0*,*10*,*20*,*30*,*40*] {*1*,*4*} = [*10*,*40*]

definition *f-join* :: *'a list* ⇒ *iT* ⇒ *'a list* (**infixl** ⋈$_f$ *100*)
where *xs* ⋈$_f$ *I* ≡ *f-join-aux xs 0 I*

definition *i-join* :: *'a ilist* ⇒ *iT* ⇒ *'a ilist* (**infixl** ⋈$_i$ *100*)
where *f* ⋈$_i$ *I* ≡ λ*n*. (*f* (*I* → *n*))

notation
f-join (**infixl** ⋈ *100*) **and**
i-join (**infixl** ⋈ *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-f-join* :: *'a ilist* ⇒ *iT* ⇒ *'a list* (**infixl** $\bowtie_{i-f}$ *100*)
where $f \bowtie_{i-f} I \equiv f \Downarrow Suc\ (Max\ I) \bowtie_f I$
notation
i-f-join (**infixl** ⋈ *100*)

The function *i-f-join* should be used only for finite sets in order to deliver well-defined results. The function *i-join* should be used for infinite sets, because joining an infinite stream s and a finite set I using *i-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-join-aux-length*:
$\bigwedge n.\ length\ (f\text{-}join\text{-}aux\ xs\ n\ I) = card\ (I \cap \{n..<n + length\ xs\})$
⟨*proof*⟩

lemma *f-join-aux-nth*[*rule-format*]:
$\forall\, n\ i.\ i < card\ (I \cap \{n..<n + length\ xs\}) \longrightarrow$
$(f\text{-}join\text{-}aux\ xs\ n\ I)\ !\ i = xs\ !\ (((I \cap \{n..<n + length\ xs\}) \to i) - n)$
⟨*proof*⟩

Joining finite streams and intervals

lemma *f-join-length*: $length\ (xs \bowtie_f I) = card\ (I \downarrow< length\ xs)$
⟨*proof*⟩

lemma *f-join-nth*: $n < length\ (xs \bowtie_f I) \Longrightarrow (xs \bowtie_f I)\ !\ n = xs\ !\ (I \to n)$
⟨*proof*⟩

lemma *f-join-nth2*: $n < card\ (I \downarrow< length\ xs) \Longrightarrow (xs \bowtie_f I)\ !\ n = xs\ !\ (I \to n)$
⟨*proof*⟩

lemma *f-join-empty*: $xs \bowtie_f \{\} = []$
⟨*proof*⟩

lemma *f-join-Nil*: $[] \bowtie_f I = []$
⟨*proof*⟩

lemma *f-join-Nil-conv*: $(xs \bowtie_f I = []) = (I \downarrow< length\ xs = \{\})$
⟨*proof*⟩

lemma *f-join-Nil-conv'*: $(xs \bowtie_f I = []) = (\forall\, i{<}length\ xs.\ i \notin I)$
⟨*proof*⟩

lemma *f-join-all-conv*: $(xs \bowtie_f I = xs) = (\{..<length\ xs\} \subseteq I)$
⟨*proof*⟩

lemma *f-join-all*: $\{..<length\ xs\} \subseteq I \Longrightarrow xs \bowtie_f I = xs$

⟨*proof*⟩

corollary *f-join-UNIV*: $xs \bowtie_f UNIV = xs$
⟨*proof*⟩

lemma *f-join-union*:
$[\![finite\ A;\ Max\ A < iMin\ B]\!] \Longrightarrow xs \bowtie_f (A \cup B) = xs \bowtie_f A\ @\ (xs \bowtie_f B)$
⟨*proof*⟩

lemma *f-join-singleton-if*:
$xs \bowtie_f \{n\} = (if\ n < length\ xs\ then\ [xs\ !\ n]\ else\ [])$
⟨*proof*⟩

lemma *f-join-insert*:
$n < length\ xs \Longrightarrow$
$xs \bowtie_f insert\ n\ I = xs \bowtie_f (I \downarrow< n)\ @\ (xs\ !\ n)\ \#\ (xs \bowtie_f (I \downarrow> n))$
⟨*proof*⟩

lemma *f-join-snoc*:
$(xs\ @\ [x]) \bowtie_f I =$
$xs \bowtie_f I\ @\ (if\ length\ xs \in I\ then\ [x]\ else\ [])$
⟨*proof*⟩

lemma *f-join-append*:
$(xs\ @\ ys) \bowtie_f I = xs \bowtie_f I\ @\ ys \bowtie_f (I \oplus- (length\ xs))$
⟨*proof*⟩

lemma *take-f-join-eq1*:
$n < card\ (I \downarrow< length\ xs) \Longrightarrow$
$(xs \bowtie_f I) \downarrow n = xs \bowtie_f (I \downarrow< (I \rightarrow n))$
⟨*proof*⟩

lemma *take-f-join-eq2*:
$card\ (I \downarrow< length\ xs) \leq n \Longrightarrow (xs \bowtie_f I) \downarrow n = xs \bowtie_f I$
⟨*proof*⟩
lemma *take-f-join-if*:
$(xs \bowtie_f I) \downarrow n =$
$(if\ n < card\ (I \downarrow< length\ xs)\ then\ xs \bowtie_f (I \downarrow< (I \rightarrow n))\ else\ xs \bowtie_f I)$
⟨*proof*⟩

lemma *drop-f-join-eq1*:
$n < card\ (I \downarrow< length\ xs) \Longrightarrow$
$(xs \bowtie_f I) \uparrow n = xs \bowtie_f (I \downarrow\geq (I \rightarrow n))$
⟨*proof*⟩

lemma *drop-f-join-eq2*:
$card\ (I \downarrow< length\ xs) \leq n \Longrightarrow (xs \bowtie_f I) \uparrow n = []$
⟨*proof*⟩

lemma *drop-f-join-if*:
$(xs \bowtie_f I) \uparrow n =$
$(\mathit{if}\ n < \mathit{card}\ (I \downarrow< \mathit{length}\ xs)\ \mathit{then}\ xs \bowtie_f (I \downarrow\geq (I \rightarrow n))\ \mathit{else}\ [])$
⟨*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*:
$A \downarrow< \mathit{length}\ xs = B \downarrow< \mathit{length}\ xs \Longrightarrow xs \bowtie_f A = xs \bowtie_f B$
⟨*proof*⟩

corollary *f-join-cut-less-eq*:
$\mathit{length}\ xs \leq t \Longrightarrow xs \bowtie_f (I \downarrow< t) = xs \bowtie_f I$
⟨*proof*⟩

lemma *take-Suc-Max-eq-imp-f-join-eq*:
$[\![\ \mathit{finite}\ I;\ xs \downarrow \mathit{Suc}\ (\mathit{Max}\ I) = ys \downarrow \mathit{Suc}\ (\mathit{Max}\ I)\ ]\!] \Longrightarrow$
$xs \bowtie_f I = ys \bowtie_f I$
⟨*proof*⟩

corollary *f-join-take-Suc-Max-eq*:
$\mathit{finite}\ I \Longrightarrow xs \downarrow \mathit{Suc}\ (\mathit{Max}\ I) \bowtie_f I = xs \bowtie_f I$
⟨*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*: $f \bowtie_i \mathit{UNIV} = f$
⟨*proof*⟩

lemma *i-join-union*:
$[\![\ \mathit{finite}\ A;\ \mathit{Max}\ A < \mathit{iMin}\ B;\ B \neq \{\}\ ]\!] \Longrightarrow$
$f \bowtie_i (A \cup B) = (f \Downarrow \mathit{Suc}\ (\mathit{Max}\ A) \bowtie_f A) \frown (f \bowtie_i B)$
⟨*proof*⟩

lemma *i-join-singleton*: $f \bowtie_i \{a\} = (\lambda n.\ f\ a)$
⟨*proof*⟩

lemma *i-join-insert*:
$f \bowtie_i (\mathit{insert}\ n\ I) =$
$(f \Downarrow n) \bowtie_f (I \downarrow< n) \frown [f\ n] \frown ($
$\mathit{if}\ I \downarrow> n = \{\}\ \mathit{then}\ (\lambda x.\ f\ n)\ \mathit{else}\ f \bowtie_i (I \downarrow> n))$
⟨*proof*⟩

lemma *i-join-i-append*:
infinite $I \Longrightarrow (xs \frown f) \bowtie_i I = (xs \bowtie_f I) \frown (f \bowtie_i (I \oplus- \mathit{length}\ xs))$
⟨*proof*⟩

lemma *i-take-i-join*: *infinite* $I \Longrightarrow f \bowtie_i I \Downarrow n = f \Downarrow (I \rightarrow n) \bowtie_f I$
⟨*proof*⟩

lemma *i-drop-i-join*: $I \neq \{\} \Longrightarrow f \bowtie_i I \Uparrow n = f \bowtie_i (I \downarrow\geq (I \rightarrow n))$
⟨*proof*⟩

lemma *i-join-i-take*: $f \Downarrow n \bowtie_f I = f \bowtie_i I \Downarrow \mathit{card}\ (I \downarrow< n)$
⟨*proof*⟩

lemma *i-join-i-drop*: $I \neq \{\} \Longrightarrow f \Uparrow n \bowtie_i I = f \bowtie_i (I \oplus n)$
⟨*proof*⟩

lemma *i-join-finite-nth-ge-card-eq-nth-Max*:
$[\![$ *finite* I; $I \neq \{\}$; *card* $I \leq \mathit{Suc}\ n$ $]\!] \Longrightarrow (f \bowtie_i I)\ n = f\ (\mathit{Max}\ I)$
⟨*proof*⟩

lemma *i-join-finite-i-drop-card-eq-const-nth-Max*:
$[\![$ *finite* I; $I \neq \{\}$ $]\!] \Longrightarrow (f \bowtie_i I) \Uparrow (\mathit{card}\ I) = (\lambda n.\ f\ (\mathit{Max}\ I))$
⟨*proof*⟩

lemma *i-join-finite-i-append-nth-Max-conv*:
$[\![$ *finite* I; $I \neq \{\}$ $]\!] \Longrightarrow (f \bowtie_i I) = f \Downarrow \mathit{Suc}\ (\mathit{Max}\ I) \bowtie_f I \frown (\lambda n.\ f\ (\mathit{Max}\ I))$
⟨*proof*⟩

Joining infinite streams and finite intervals

lemma *i-f-join-length*: *finite* $I \Longrightarrow \mathit{length}\ (f \bowtie_{i-f} I) = \mathit{card}\ I$
⟨*proof*⟩

lemma *i-f-join-nth*: $n < \mathit{card}\ I \Longrightarrow f \bowtie_{i-f} I\ !\ n = f\ (I \rightarrow n)$
⟨*proof*⟩

lemma *i-f-join-empty*: $f \bowtie_{i-f} \{\} = []$
⟨*proof*⟩

lemma *i-f-join-eq-i-join-i-take*:
finite $I \Longrightarrow f \bowtie_{i-f} I = f \bowtie_i I \Downarrow (\mathit{card}\ I)$
⟨*proof*⟩

lemma *i-f-join-union*:
$[\![$ *finite* A; *finite* B; $\mathit{Max}\ A < \mathit{iMin}\ B$ $]\!] \Longrightarrow$
$f \bowtie_{i-f} (A \cup B) = f \bowtie_{i-f} A\ @\ f \bowtie_{i-f} B$
⟨*proof*⟩

lemma *i-f-join-singleton*: $f \bowtie_{i-f} \{n\} = [f\ n]$

⟨*proof*⟩

lemma *i-f-join-insert*:
finite I ⟹
$f \bowtie_{i-f} \mathit{insert}\ n\ I = f \bowtie_{i-f} (I \downarrow< n)\ @\ f\ n\ \#\ f \bowtie_{i-f} (I \downarrow> n)$
⟨*proof*⟩

lemma *take-i-f-join-eq1*:
$n < \mathit{card}\ I \Longrightarrow f \bowtie_{i-f} I \downarrow n = f \bowtie_{i-f} (I \downarrow< (I \rightarrow n))$
⟨*proof*⟩

lemma *take-i-f-join-eq2*:
⟦ *finite I*; *card I* ≤ *n* ⟧ ⟹ $f \bowtie_{i-f} I \downarrow n = f \bowtie_{i-f} I$
⟨*proof*⟩

lemma *take-i-f-join-if*:
finite I ⟹
$f \bowtie_{i-f} I \downarrow n = (\mathit{if}\ n < \mathit{card}\ I\ \mathit{then}\ f \bowtie_{i-f} (I \downarrow< (I \rightarrow n))\ \mathit{else}\ f \bowtie_{i-f} I)$
⟨*proof*⟩

lemma *drop-i-f-join-eq1*:
$n < \mathit{card}\ I \Longrightarrow f \bowtie_{i-f} I \uparrow n = f \bowtie_{i-f} (I \downarrow\geq (I \rightarrow n))$
⟨*proof*⟩

lemma *drop-i-f-join-eq2*:
⟦ *finite I*; *card I* ≤ *n* ⟧ ⟹ $f \bowtie_{i-f} I \uparrow n = []$
⟨*proof*⟩

lemma *drop-i-f-join-if*:
finite I ⟹
$f \bowtie_{i-f} I \uparrow n = (\mathit{if}\ n < \mathit{card}\ I\ \mathit{then}\ f \bowtie_{i-f} (I \downarrow\geq (I \rightarrow n))\ \mathit{else}\ [])$
⟨*proof*⟩

lemma *i-f-join-i-drop*:
finite I ⟹ $f \Uparrow n \bowtie_{i-f} I = f \bowtie_{i-f} (I \oplus n)$
⟨*proof*⟩

lemma *i-take-Suc-Max-eq-imp-i-f-join-eq*:
$f \Downarrow \mathit{Suc}\ (\mathit{Max}\ I) = g \Downarrow \mathit{Suc}\ (\mathit{Max}\ I) \Longrightarrow f \bowtie_{i-f} I = g \bowtie_{i-f} I$
⟨*proof*⟩

lemma *i-take-i-join-eq-i-f-join*:
infinite I ⟹ $f \bowtie_i I \Downarrow n = f \bowtie_{i-f} (I \downarrow< (I \rightarrow n))$
⟨*proof*⟩

4.1.3 Results for intervals from *IL-Interval*

lemma *f-join-iFROM*: $xs \bowtie_f [n\ldots] = xs \uparrow n$
⟨*proof*⟩

lemma *i-join-iFROM*: $f \bowtie_i [n...] = f \Uparrow n$
⟨*proof*⟩

lemma *f-join-iIN*: $xs \bowtie_f [n...,d] = xs \uparrow n \downarrow Suc\ d$
⟨*proof*⟩

lemma *i-f-join-iIN*: $f \bowtie_{i-f} [n...,d] = f \Uparrow n \Downarrow Suc\ d$
⟨*proof*⟩

lemma *f-join-iTILL*: $xs \bowtie_f [...n] = xs \downarrow (Suc\ n)$
⟨*proof*⟩

lemma *i-f-join-iTILL*: $f \bowtie_{i-f} [...n] = f \Downarrow Suc\ n$
⟨*proof*⟩

lemma *f-join-f-expand-iT-Mult*:
$0 < k \Longrightarrow xs \odot_f k \bowtie_f (I \otimes k) = xs \bowtie_f I$
⟨*proof*⟩

lemma *i-join-i-expand-iT-Mult*:
$[\![\ 0 < k;\ I \neq \{\}\ ]\!] \Longrightarrow f \odot_i k \bowtie_i (I \otimes k) = f \bowtie_i I$
⟨*proof*⟩

lemma *i-f-join-i-expand-iT-Mult*:
$[\![\ 0 < k;\ finite\ I\ ]\!] \Longrightarrow f \odot_i k \bowtie_{i-f} (I \otimes k) = f \bowtie_{i-f} I$
⟨*proof*⟩

lemma *f-join-f-shrink-iT-Plus-iT-Div-mod*:
$[\![\ 0 < k;\ \forall x{\in}I.\ x\ mod\ k = 0\ ]\!] \Longrightarrow$
$(xs \longmapsto_f k) \bowtie_f (I \oplus (k - 1)) = xs \div_f k \bowtie_f (I \oslash k)$
⟨*proof*⟩

lemma *i-join-i-shrink-iT-Plus-iT-Div-mod*:
$[\![\ 0 < k;\ I \neq \{\};\ \forall x{\in}I.\ x\ mod\ k = 0\ ]\!] \Longrightarrow$
$(f \longmapsto_i k) \bowtie_i (I \oplus (k - 1)) = f \div_i k \bowtie_i (I \oslash k)$
⟨*proof*⟩

lemma *i-f-join-i-shrink-iT-Plus-iT-Div-mod*:
$[\![\ 0 < k;\ finite\ I;\ \forall x{\in}I.\ x\ mod\ k = 0\ ]\!] \Longrightarrow$
$(f \longmapsto_i k) \bowtie_{i-f} (I \oplus (k - 1)) = f \div_i k \bowtie_{i-f} (I \oslash k)$
⟨*proof*⟩

corollary *f-join-f-shrink-iT-Plus-iT-Div-mod-subst*:
$[\![\ 0 < k;\ \forall x{\in}I.\ x\ mod\ k = 0;$
$A = I \oplus (k - 1);\ B = I \oslash k\ ]\!] \Longrightarrow$
$(xs \longmapsto_f k) \bowtie_f A = xs \div_f k \bowtie_f B$
⟨*proof*⟩

corollary *i-join-i-shrink-iT-Plus-iT-Div-mod-subst*:
$[\![\ 0 < k;\ I \neq \{\};\ \forall\, x{\in}I.\ x\ mod\ k = 0;$
$A = I \oplus (k - 1);\ B = I \oslash k\ ]\!] \Longrightarrow$
$(f \longmapsto_i k) \bowtie_i A = f \div_i k \bowtie_i B$
⟨*proof*⟩
corollary *i-f-join-i-shrink-iT-Plus-iT-Div-mod-subst*:
$[\![\ 0 < k;\ finite\ I;\ \forall\, x{\in}I.\ x\ mod\ k = 0;$
$A = I \oplus (k - 1);\ B = I \oslash k\ ]\!] \Longrightarrow$
$(f \longmapsto_i k) \bowtie_{i-f} A{=} f \div_i k \bowtie_{i-f} B$
⟨*proof*⟩

lemma *f-join-f-shrink-iT-Div-mod*:
$[\![\ 0 < k;\ \forall\, x{\in}I.\ x\ mod\ k = k - 1\ ]\!] \Longrightarrow$
$(xs \longmapsto_f k) \bowtie_f I = xs \div_f k \bowtie_f (I \oslash k)$
⟨*proof*⟩

lemma *i-join-i-shrink-iT-Div-mod*:
$[\![\ 0 < k;\ I \neq \{\};\ \forall\, x{\in}I.\ x\ mod\ k = k - 1\ ]\!] \Longrightarrow$
$(f \longmapsto_i k) \bowtie_i I{=} f \div_i k \bowtie_i (I \oslash k)$
⟨*proof*⟩

lemma *i-f-join-i-shrink-iT-Div-mod*:
$[\![\ 0 < k;\ finite\ I;\ \forall\, x{\in}I.\ x\ mod\ k = k - 1\ ]\!] \Longrightarrow$
$(f \longmapsto_i k) \bowtie_{i-f} I = f \div_i k \bowtie_{i-f} (I \oslash k)$
⟨*proof*⟩

lemma *f-join-f-expand-iMOD*:
$0 < k \Longrightarrow xs \odot_f k \bowtie_f [n * k,\ mod\ k] = xs \bowtie_f [n...]$
⟨*proof*⟩
corollary *f-join-f-expand-iMOD-0*:
$0 < k \Longrightarrow xs \odot_f k \bowtie_f [0,\ mod\ k] = xs$
⟨*proof*⟩

lemma *f-join-f-expand-iMODb*:
$0 < k \Longrightarrow xs \odot_f k \bowtie_f [n * k,\ mod\ k,\ d] = xs \bowtie_f [n...,d]$
⟨*proof*⟩

corollary *f-join-f-expand-iMODb-0*:
$0 < k \Longrightarrow xs \odot_f k \bowtie_f [0,\ mod\ k,\ n] = xs \bowtie_f [...n]$
⟨*proof*⟩

lemma *i-join-i-expand-iMOD*:
$0 < k \Longrightarrow f \odot_i k \bowtie_i [n * k,\ mod\ k] = f \bowtie_i [n...]$
⟨*proof*⟩

corollary *i-join-i-expand-iMOD-0*:
$0 < k \Longrightarrow f \odot_i k \bowtie_i [0,\ mod\ k] = f$

⟨*proof*⟩

lemma *i-join-i-expand-iMODb*:
$0 < k \Longrightarrow f \odot_i k \bowtie_i [n * k, mod\ k, d] = f \bowtie_i [n..,d]$
⟨*proof*⟩

corollary *i-join-i-expand-iMODb-0*:
$0 < k \Longrightarrow f \odot_i k \bowtie_i [0, mod\ k, n] = f \bowtie_i [..n]$
⟨*proof*⟩

lemma *i-f-join-i-expand-iMODb*:
$0 < k \Longrightarrow f \odot_i k \bowtie_{i-f} [n * k, mod\ k, d] = f \bowtie_{i-f} [n..,d]$
⟨*proof*⟩
corollary *i-f-join-i-expand-iMODb-0*:
$0 < k \Longrightarrow f \odot_i k \bowtie_{i-f} [0, mod\ k, n] = f \bowtie_{i-f} [..n]$
⟨*proof*⟩

lemma *f-join-f-shrink-iMOD*:
$0 < k \Longrightarrow (xs \longmapsto_f k) \bowtie_f [n * k + (k - 1), mod\ k] = xs \div_f k \bowtie_f [n..]$
⟨*proof*⟩

corollary *f-join-f-shrink-iMOD-0*:
$0 < k \Longrightarrow (xs \longmapsto_f k) \bowtie_f [k - 1, mod\ k] = xs \div_f k$
⟨*proof*⟩

lemma *f-join-f-shrink-iMODb*:
$0 < k \Longrightarrow (xs \longmapsto_f k) \bowtie_f [n * k + (k - 1), mod\ k, d] = xs \div_f k \bowtie_f [n..,d]$
⟨*proof*⟩

corollary *f-join-f-shrink-iMODb-0*:
$0 < k \Longrightarrow (xs \longmapsto_f k) \bowtie_f [k - 1, mod\ k, n] = xs \div_f k \bowtie_f [..n]$
⟨*proof*⟩

lemma *i-join-i-shrink-iMOD*:
$0 < k \Longrightarrow (f \longmapsto_i k) \bowtie_i [n * k + (k - 1), mod\ k] = f \div_i k \bowtie_i [n..]$
⟨*proof*⟩

corollary *i-join-i-shrink-iMOD-0*:
$0 < k \Longrightarrow (f \longmapsto_i k) \bowtie_i [k - 1, mod\ k] = f \div_i k$
⟨*proof*⟩

lemma *i-f-join-i-shrink-iMODb*:
$0 < k \Longrightarrow (f \longmapsto_i k) \bowtie_{i-f} [n * k + (k - 1), mod\ k, d] = f \div_i k \bowtie_{i-f} [n..,d]$
⟨*proof*⟩

corollary *i-f-join-i-shrink-iMODb-0*:
$0 < k \Longrightarrow (f \longmapsto_i k) \bowtie_{i-f} [k - 1, mod\ k, n] = f \div_i k \bowtie_{i-f} [..n]$

⟨*proof*⟩

4.2 Streams and temporal operators

lemma *i-shrink-eq-NoMsg-iAll-conv*:
$0 < k \Longrightarrow ((s \div_i k)\ t = \varepsilon) = (\Box\ t1\ [t * k \ldots, k - Suc\ 0].\ s\ t1 = \varepsilon)$
⟨*proof*⟩

lemma *i-shrink-eq-NoMsg-iAll-conv2*:
$0 < k \Longrightarrow ((s \div_i k)\ t = \varepsilon) = (\Box\ t1\ [\ldots k - 1] \oplus (t * k).\ s\ t1 = \varepsilon)$
⟨*proof*⟩

lemma *i-shrink-eq-Msg-iEx-iAll-conv*:
$[\![\ 0 < k;\ m \neq \varepsilon\ ]\!] \Longrightarrow$
$((s \div_i k)\ t = m) =$
$(\Diamond\ t1\ [t * k \ldots, k - Suc\ 0].\ s\ t1 = m \wedge$
$(\Box\ t2\ [Suc\ t1 \ldots].\ t2 \leq t * k + k - Suc\ 0 \longrightarrow s\ t2 = \varepsilon))$
⟨*proof*⟩

lemma *i-shrink-eq-Msg-iEx-iAll-conv2*:
$[\![\ 0 < k;\ m \neq \varepsilon\ ]\!] \Longrightarrow$
$((s \div_i k)\ t = m) =$
$(\Diamond\ t1\ [\ldots k - 1] \oplus (t * k).\ s\ t1 = m \wedge$
$(\Box\ t2\ [1 \ldots] \oplus t1\ .\ t2 \leq t * k + k - 1 \longrightarrow s\ t2 = \varepsilon))$
⟨*proof*⟩

lemma *i-shrink-eq-Msg-iEx-iAll-cut-greater-conv*:
$[\![\ 0 < k;\ m \neq \varepsilon\ ]\!] \Longrightarrow$
$((s \div_i k)\ t = m) =$
$(\Diamond\ t1\ [t * k \ldots, k - Suc\ 0].\ s\ t1 = m \wedge$
$(\Box\ t2\ [t * k \ldots, k - Suc\ 0] \downarrow> t1.\ s\ t2 = \varepsilon))$
⟨*proof*⟩

lemma *i-shrink-eq-Msg-iEx-iAll-cut-greater-conv2*:
$[\![\ 0 < k;\ m \neq \varepsilon\ ]\!] \Longrightarrow$
$((s \div_i k)\ t = m) =$
$(\Diamond\ t1\ [\ldots k - 1] \oplus (t * k).\ s\ t1 = m \wedge$
$(\Box\ t2\ ([\ldots k - 1] \oplus (t * k)) \downarrow> t1.\ s\ t2 = \varepsilon))$
⟨*proof*⟩

lemma *i-shrink-eq-Msg-iSince-conv*:
$[\![\ 0 < k;\ m \neq \varepsilon\ ]\!] \Longrightarrow$
$((s \div_i k)\ t = m) =$
$(s\ t2 = \varepsilon.\ t2\ \mathcal{S}\ t1\ [t * k \ldots, k - Suc\ 0].\ s\ t1 = m)$
⟨*proof*⟩
lemma *i-shrink-eq-Msg-iSince-conv2*:
$[\![\ 0 < k;\ m \neq \varepsilon\ ]\!] \Longrightarrow$
$((s \div_i k)\ t = m) =$
$(s\ t2 = \varepsilon.\ t2\ \mathcal{S}\ t1\ [\ldots k - 1] \oplus (t * k).\ s\ t1 = m)$

⟨*proof*⟩

lemma *iT-Mult-iAll-i-expand-nth-iff*:
$0 < k \Longrightarrow (\Box\ t\ (I \otimes k).\ P\ ((f \odot_i k)\ t)) = (\Box\ t\ I.\ P\ (f\ t))$
⟨*proof*⟩

Streams and temporal operators cycle start/finish events

lemma *i-shrink-eq-NoMsg-iAll-start-event-conv*:
$[\![\ 0 < k;\ \bigwedge t.\ event\ t = (t\ mod\ k = 0);\ t0 = t * k\ ]\!] \Longrightarrow$
$((s \div_i k)\ t = \varepsilon) =$
$(s\ t0 = \varepsilon \wedge (\bigcirc\ t'\ t0\ [0\ldots].\ (s\ t1 = \varepsilon.\ t1\ \mathcal{U}\ t2\ ([0\ldots] \oplus t').\ event\ t2)))$
⟨*proof*⟩

lemma *i-shrink-eq-Msg-iUntil-start-event-conv*:
$[\![\ 0 < k;\ m \neq \varepsilon;\ \bigwedge t.\ event\ t = (t\ mod\ k = 0);\ t0 = t * k\ ]\!] \Longrightarrow$
$((s \div_i k)\ t = m) = ($
$(s\ t0 = m \wedge (\bigcirc\ t'\ t0\ [0\ldots].\ (s\ t1 = \varepsilon.\ t1\ \mathcal{U}\ t2\ ([0\ldots] \oplus t').\ event\ t2))) \vee$
$(\bigcirc\ t'\ t0\ [0\ldots].\ (\neg\ event\ t1.\ t1\ \mathcal{U}\ t2\ ([0\ldots] \oplus t').\ ($
$s\ t2 = m \wedge \neg\ event\ t2 \wedge (\bigcirc\ t''\ t2\ [0\ldots].$
$(s\ t3 = \varepsilon.\ t3\ \mathcal{U}\ t4\ ([0\ldots] \oplus t'').\ event\ t4))))))$
⟨*proof*⟩

lemma *i-shrink-eq-NoMsg-iAll-finish-event-conv*:
$[\![\ 1 < k;\ \bigwedge t.\ event\ t = (t\ mod\ k = k - 1);\ t0 = t * k\ ]\!] \Longrightarrow$
$((s \div_i k)\ t = \varepsilon) =$
$(s\ t0 = \varepsilon \wedge (\bigcirc\ t'\ t0\ [0\ldots].\ (s\ t1 = \varepsilon.\ t1\ \mathcal{U}\ t2\ ([0\ldots] \oplus t').\ (event\ t2 \wedge s\ t2 = \varepsilon))))$
⟨*proof*⟩

lemma *i-shrink-eq-Msg-iUntil-finish-event-conv*:
$[\![\ 1 < k;\ m \neq \varepsilon;\ \bigwedge t.\ event\ t = (t\ mod\ k = k - 1);\ t0 = t * k\ ]\!] \Longrightarrow$
$((s \div_i k)\ t = m) = ($
$(\neg\ event\ t1.\ t1\ \mathcal{U}\ t2\ ([0\ldots] \oplus t0).\ event\ t2 \wedge s\ t2 = m) \vee$
$(\neg\ event\ t1.\ t1\ \mathcal{U}\ t2\ ([0\ldots] \oplus t0).\ (\neg\ event\ t2 \wedge s\ t2 = m \wedge ($
$\bigcirc\ t'\ t2\ [0\ldots].\ (s\ t3 = \varepsilon.\ t3\ \mathcal{U}\ t4\ ([0\ldots] \oplus t').\ event\ t4 \wedge s\ t4 = \varepsilon)))))$
⟨*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*:
⟦ *result* = *i-Exec-Comp-Stream trans-fun input c*;
∀ *x-n c-n*. *P1 x-n* ∧ *P2 c-n* ⟶ *Q* (*trans-fun x-n c-n*) ⟧ ⟹
□ *t I*. (*P1* (*input t*) ∧ *P2* (*result*$^{\leftarrow\ c}$ *t*) ⟶ *Q* (*result t*))
⟨*proof*⟩

Direct relation between input and result after transition

lemma *i-Exec-Stream-Pre-Post2-iAll*:
⟦ *result* = *i-Exec-Comp-Stream trans-fun input c*;
∀ *x-n c-n*. *P c-n* ⟶ *Q x-n* (*trans-fun x-n c-n*) ⟧ ⟹
□ *t I*. *P* (*result*$^{\leftarrow\ c}$ *t*) ⟶ *Q* (*input t*) (*result t*)
⟨*proof*⟩

lemma *i-Exec-Stream-Pre-Post3-iAll-iNext*:
⟦ *result* = *i-Exec-Comp-Stream trans-fun input c*;
∀ *x-n c-n*. *P c-n* ⟶ *Q x-n* (*trans-fun x-n c-n*);
∀ *t*∈*I*. *inext t I'* = *Suc t* ⟧ ⟹
□ *t I*. *P* (*result t*) ⟶ (○ *t1 t I'*. *Q* (*input t1*) (*result t1*))
⟨*proof*⟩

lemma *i-Exec-Stream-Init-Pre-Post1-iAll-iNext*:
⟦ *result* = *i-Exec-Comp-Stream-Init trans-fun input c*;
∀ *x-n c-n*. *P1 x-n* ∧ *P2 c-n* ⟶ *Q* (*trans-fun x-n c-n*);
∀ *t*∈*I*. *inext t I'* = *Suc t* ⟧ ⟹
□ *t I*. (*P1* (*input t*) ∧ *P2* (*result t*) ⟶ (○ *t1 t I'*. *Q* (*result t1*)))
⟨*proof*⟩

Direct relation between input and state before transition

lemma *i-Exec-Stream-Init-Pre-Post2-iAll-iNext*:
⟦ *result* = *i-Exec-Comp-Stream-Init trans-fun input c*;
∀ *x-n c-n*. *P x-n c-n* ⟶ *Q* (*trans-fun x-n c-n*);
∀ *t*∈*I*. *inext t I'* = *Suc t* ⟧ ⟹
□ *t I*. (*P* (*input t*) (*result t*) ⟶ (○ *t1 t I'*. *Q* (*result t1*)))
⟨*proof*⟩

Relation between input and output

lemma *i-Exec-Stream-Init-Pre-Post3-iAll-iNext*:
⟦ *result* = *i-Exec-Comp-Stream-Init trans-fun input c*;
∀ *x-n c-n*. *P c-n* ⟶ *Q x-n* (*trans-fun x-n c-n*);
∀ *t*∈*I*. *inext t I'* = *Suc t* ⟧ ⟹
□ *t I*. (*P* (*result t*) ⟶ (○ *t1 t I'*. *Q* (*input*$^{\leftarrow\ \varepsilon}$ *t1*) (*result t1*)))
⟨*proof*⟩

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*:
$0 < k \Longrightarrow$
$((i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Acc\text{-}Output\ k\ output\text{-}fun\ trans\text{-}fun\ input\ c)\ t = \varepsilon) =$
$(\Box\ t1\ [t * k \ldots, k - Suc\ 0].\ (output\text{-}fun \circ i\text{-}Exec\text{-}Comp\text{-}Stream\ trans\text{-}fun\ (input \odot_i k)\ c)\ t1 = \varepsilon)$
⟨*proof*⟩

lemma *i-Exec-Comp-Stream-Acc-Output--eq-NoMsg-iAll-conv2*:
$0 < k \Longrightarrow$
$((i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Acc\text{-}Output\ k\ output\text{-}fun\ trans\text{-}fun\ input\ c)\ t = \varepsilon) =$
$(\Box\ t1\ [\ldots k - Suc\ 0] \oplus (t * k).\ (output\text{-}fun \circ i\text{-}Exec\text{-}Comp\text{-}Stream\ trans\text{-}fun\ (input \odot_i k)\ c)\ t1 = \varepsilon)$
⟨*proof*⟩

lemma *i-Exec-Comp-Stream-Acc-Output--Init--eq-NoMsg-iAll-conv*:
$0 < k \Longrightarrow$
$((i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Acc\text{-}Output\ k\ output\text{-}fun\ trans\text{-}fun\ input\ c)\ t = \varepsilon) =$
$(\Box\ t1\ [Suc\ (t * k) \ldots, k - Suc\ 0].\ (output\text{-}fun \circ i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Init\ trans\text{-}fun\ (input \odot_i k)\ c)\ t1 = \varepsilon)$
⟨*proof*⟩

lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-iEx-iAll-cut-greater-conv*:
$[\![\ 0 < k;\ m \neq \varepsilon;\ s = (output\text{-}fun \circ i\text{-}Exec\text{-}Comp\text{-}Stream\ trans\text{-}fun\ (input \odot_i k)\ c)\ ]\!] \Longrightarrow$
$((i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Acc\text{-}Output\ k\ output\text{-}fun\ trans\text{-}fun\ input\ c)\ t = m) =$
$(\Diamond\ t1\ [t * k \ldots, k - Suc\ 0].\ (s\ t1 = m \wedge$
$(\Box\ t2\ [t * k \ldots, k - Suc\ 0] \downarrow> t1\ .\ s\ t2 = \varepsilon)))$
⟨*proof*⟩

lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-iEx-iAll-cut-greater-conv2*:
$[\![\ 0 < k;\ m \neq \varepsilon;\ s = (output\text{-}fun \circ i\text{-}Exec\text{-}Comp\text{-}Stream\ trans\text{-}fun\ (input \odot_i k)\ c)\ ]\!] \Longrightarrow$
$((i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Acc\text{-}Output\ k\ output\text{-}fun\ trans\text{-}fun\ input\ c)\ t = m) =$
$(\Diamond\ t1\ [\ldots k - Suc\ 0] \oplus (t * k).\ (s\ t1 = m \wedge$
$(\Box\ t2\ ([\ldots k - Suc\ 0] \oplus (t * k)) \downarrow> t1\ .\ s\ t2 = \varepsilon)))$
⟨*proof*⟩

lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-iSince-conv*:
$[\![\ 0 < k;\ m \neq \varepsilon;\ s = (output\text{-}fun \circ i\text{-}Exec\text{-}Comp\text{-}Stream\ trans\text{-}fun\ (input \odot_i k)\ c)\ ]\!] \Longrightarrow$
$((i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Acc\text{-}Output\ k\ output\text{-}fun\ trans\text{-}fun\ input\ c)\ t = m) =$
$(s\ t2 = \varepsilon.\ t2\ \mathcal{S}\ t1\ [t * k \ldots, k - Suc\ 0].\ s\ t1 = m)$
⟨*proof*⟩

lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-iSince-conv2*:
⟦ *0 < k*; *m ≠ ε*; *s = (output-fun ∘ 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) =
(s t2 = ε. t2 𝒮 t1 [...k − Suc 0] ⊕ (t ∗ k). s t1 = m)
⟨*proof*⟩

lemma *i-Exec-Comp-Stream-Acc-Output--Init--eq-Msg-iSince-conv*:
⟦ *0 < k*; *m ≠ ε*; *s = (output-fun ∘ i-Exec-Comp-Stream-Init trans-fun (input ⊙ᵢ k) c)* ⟧ ⟹
((i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c) t = m) =
(s t2 = ε. t2 𝒮 t1 [Suc (t ∗ k)...,k − Suc 0]. s t1 = m)
⟨*proof*⟩

lemma *i-Exec-Comp-Stream-Acc-Output--eq-iAll-iSince-conv*:
⟦ *0 < k*; *s = (output-fun ∘ 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 ⟶ (□ t1 [t ∗ k...,k − Suc 0]. s t1 = ε)) ∧
((m ≠ ε ⟶ (s t2 = ε. t2 𝒮 t1 [t ∗ k...,k − Suc 0]. s t1 = m))))
⟨*proof*⟩

lemma *i-Exec-Comp-Stream-Acc-Output--eq-iAll-iSince-conv2*:
⟦ *0 < k*; *s = (output-fun ∘ 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 ⟶ (□ t1 [...k − Suc 0] ⊕ (t ∗ k). s t1 = ε)) ∧
((m ≠ ε ⟶ (s t2 = ε. t2 𝒮 t1 [...k − Suc 0] ⊕ (t ∗ k). s 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*:
⟦ *0 < k*; ⋀ *t. event t = (t mod k = 0)*; *t0 = t ∗ k*;
s = (output-fun ∘ i-Exec-Comp-Stream trans-fun (input ⊙ᵢ k) c) ⟧⟹
((i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c) t = ε) =
(s t0 = ε ∧ (○ t′ t0 [0...]. (s t1 = ε. t1 𝒰 t2 [0...] ⊕ t′. event t2)))
⟨*proof*⟩

lemma *i-Exec-Comp-Stream-Acc-Output--Init--eq-NoMsg-iAll-start-event-conv*:
⟦ *0 < k*; ⋀ *t. event t = ((t + k − Suc 0) mod k = 0)*; *t0 = Suc (t ∗ k)*;
s = (output-fun ∘ i-Exec-Comp-Stream-Init trans-fun (input ⊙ᵢ k) c) ⟧⟹
((i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c) t = ε) =
(s t0 = ε ∧ (○ t′ t0 [0...]. (s t1 = ε. t1 𝒰 t2 [0...] ⊕ t′. event t2)))
⟨*proof*⟩

lemma *i-Exec-Comp-Stream-Acc-Output--Init--eq-NoMsg-iAll-start-event2-conv*:
⟦ *Suc 0 < k*; ⋀ *t. event t = (t mod k = Suc 0)*; *t0 = Suc (t ∗ k)*;
s = (output-fun ∘ i-Exec-Comp-Stream-Init trans-fun (input ⊙ᵢ k) c) ⟧⟹
((i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c) t = ε) =

$(s\ t0 = \varepsilon \wedge (\bigcirc\ t'\ t0\ [0\ldots].\ (s\ t1 = \varepsilon.\ t1\ \mathcal{U}\ t2\ [0\ldots] \oplus t'.\ event\ t2)))$
⟨*proof*⟩

lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-iUntil-start-event-conv*:
$[\![\ 0 < k;\ m \neq \varepsilon;\ \bigwedge t.\ event\ t = (t\ mod\ k = 0);\ t0 = t * k;$
$s = (output\text{-}fun \circ i\text{-}Exec\text{-}Comp\text{-}Stream\ trans\text{-}fun\ (input \odot_i k)\ c)\ ]\!] \Longrightarrow$
$((i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Acc\text{-}Output\ k\ output\text{-}fun\ trans\text{-}fun\ input\ c)\ t = m) = ($
$(s\ t0 = m \wedge (\bigcirc\ t'\ t0\ [0\ldots].\ (s\ t1 = \varepsilon.\ t1\ \mathcal{U}\ t2\ ([0\ldots] \oplus t').\ event\ t2))) \vee$
$(\bigcirc\ t'\ t0\ [0\ldots].\ (\neg\ event\ t1.\ t1\ \mathcal{U}\ t2\ ([0\ldots] \oplus t').\ ($
$s\ t2 = m \wedge \neg\ event\ t2 \wedge (\bigcirc\ t''\ t2\ [0\ldots].$
$(s\ t3 = \varepsilon.\ t3\ \mathcal{U}\ t4\ ([0\ldots] \oplus t'').\ event\ t4))))))$
⟨*proof*⟩

lemma *i-Exec-Comp-Stream-Acc-Output--Init--eq-Msg-iUntil-start-event-conv*:
$[\![\ 0 < k;\ m \neq \varepsilon;\ \bigwedge t.\ event\ t = ((t + k - Suc\ 0)\ mod\ k = 0);\ t0 = Suc\ (t * k);$
$s = (output\text{-}fun \circ i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Init\ trans\text{-}fun\ (input \odot_i k)\ c)\ ]\!] \Longrightarrow$
$((i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Acc\text{-}Output\ k\ output\text{-}fun\ trans\text{-}fun\ input\ c)\ t = m) = ($
$(s\ t0 = m \wedge (\bigcirc\ t'\ t0\ [0\ldots].\ (s\ t1 = \varepsilon.\ t1\ \mathcal{U}\ t2\ ([0\ldots] \oplus t').\ event\ t2))) \vee$
$(\bigcirc\ t'\ t0\ [0\ldots].\ (\neg\ event\ t1.\ t1\ \mathcal{U}\ t2\ ([0\ldots] \oplus t').\ ($
$s\ t2 = m \wedge \neg\ event\ t2 \wedge (\bigcirc\ t''\ t2\ [0\ldots].$
$(s\ t3 = \varepsilon.\ t3\ \mathcal{U}\ t4\ ([0\ldots] \oplus t'').\ event\ t4))))))$
⟨*proof*⟩

lemma *i-Exec-Comp-Stream-Acc-Output--Init--eq-Msg-iUntil-start-event2-conv*:
$[\![\ Suc\ 0 < k;\ m \neq \varepsilon;\ \bigwedge t.\ event\ t = (t\ mod\ k = Suc\ 0);\ t0 = Suc\ (t * k);$
$s = (output\text{-}fun \circ i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Init\ trans\text{-}fun\ (input \odot_i k)\ c)\ ]\!] \Longrightarrow$
$((i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Acc\text{-}Output\ k\ output\text{-}fun\ trans\text{-}fun\ input\ c)\ t = m) = ($
$(s\ t0 = m \wedge (\bigcirc\ t'\ t0\ [0\ldots].\ (s\ t1 = \varepsilon.\ t1\ \mathcal{U}\ t2\ ([0\ldots] \oplus t').\ event\ t2))) \vee$
$(\bigcirc\ t'\ t0\ [0\ldots].\ (\neg\ event\ t1.\ t1\ \mathcal{U}\ t2\ ([0\ldots] \oplus t').\ ($
$s\ t2 = m \wedge \neg\ event\ t2 \wedge (\bigcirc\ t''\ t2\ [0\ldots].$
$(s\ t3 = \varepsilon.\ t3\ \mathcal{U}\ t4\ ([0\ldots] \oplus t'').\ event\ t4))))))$
⟨*proof*⟩

lemma *i-Exec-Comp-Stream-Acc-Output--eq-NoMsg-iAll-finish-event-conv*:
$[\![\ Suc\ 0 < k;\ \bigwedge\ t.\ event\ t = (t\ mod\ k = k - Suc\ 0);\ t0 = t * k;$
$s = (output\text{-}fun \circ i\text{-}Exec\text{-}Comp\text{-}Stream\ trans\text{-}fun\ (input \odot_i k)\ c)\ ]\!]\Longrightarrow$
$((i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Acc\text{-}Output\ k\ output\text{-}fun\ trans\text{-}fun\ input\ c)\ t = \varepsilon) =$
$(s\ t0 = \varepsilon \wedge (\bigcirc\ t'\ t0\ [0\ldots].\ (s\ t1 = \varepsilon.\ t1\ \mathcal{U}\ t2\ [0\ldots] \oplus t'.\ event\ t2 \wedge s\ t2 = \varepsilon)))$
⟨*proof*⟩

lemma *i-Exec-Comp-Stream-Acc-Output--Init--eq-NoMsg-iAll-finish-event-conv*:
$[\![\ Suc\ 0 < k;\ \bigwedge\ t.\ event\ t = (t\ mod\ k = 0);\ t0 = Suc\ (t * k);$
$s = (output\text{-}fun \circ i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Init\ trans\text{-}fun\ (input \odot_i k)\ c)\ ]\!]\Longrightarrow$
$((i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Acc\text{-}Output\ k\ output\text{-}fun\ trans\text{-}fun\ input\ c)\ t = \varepsilon) =$
$(s\ t0 = \varepsilon \wedge (\bigcirc\ t'\ t0\ [0\ldots].\ (s\ t1 = \varepsilon.\ t1\ \mathcal{U}\ t2\ [0\ldots] \oplus t'.\ event\ t2 \wedge s\ t2 = \varepsilon)))$
⟨*proof*⟩

lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-iUntil-finish-event-conv*:
$[\![\ 0 < k;\ m \neq \varepsilon;\ \bigwedge\ t.\ event\ t = (t\ mod\ k = k - Suc\ 0);\ t0 = t * k;$

$s = (output\text{-}fun \circ i\text{-}Exec\text{-}Comp\text{-}Stream\ trans\text{-}fun\ (input \odot_i k)\ c)\ ]\!]\Longrightarrow$
$((i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Acc\text{-}Output\ k\ output\text{-}fun\ trans\text{-}fun\ input\ c)\ t = m) =$
$((\neg\ event\ t1.\ t1\ \mathcal{U}\ t2\ ([0\ldots] \oplus t0).\ event\ t2 \wedge s\ t2 = m) \vee$
$(\neg\ event\ t1.\ t1\ \mathcal{U}\ t2\ ([0\ldots] \oplus t0).\ (\neg\ event\ t2 \wedge s\ t2 = m \wedge ($
$\bigcirc\ t'\ t2\ [0\ldots].\ (s\ t3 = \varepsilon.\ t3\ \mathcal{U}\ t4\ ([0\ldots] \oplus t').\ event\ t4 \wedge s\ t4 = \varepsilon)))))$
⟨*proof*⟩

lemma *i-Exec-Comp-Stream-Acc-Output--Init--eq-Msg-iUntil-finish-event-conv*:
$[\![\ Suc\ 0 < k;\ m \neq \varepsilon;\ \bigwedge t.\ event\ t = (t\ mod\ k = 0);\ t0 = Suc\ (t * k);$
$s = (output\text{-}fun \circ i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Init\ trans\text{-}fun\ (input \odot_i k)\ c)\ ]\!]\Longrightarrow$
$((i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Acc\text{-}Output\ k\ output\text{-}fun\ trans\text{-}fun\ input\ c)\ t = m) =$
$((\neg\ event\ t1.\ t1\ \mathcal{U}\ t2\ ([0\ldots] \oplus t0).\ event\ t2 \wedge s\ t2 = m) \vee$
$(\neg\ event\ t1.\ t1\ \mathcal{U}\ t2\ ([0\ldots] \oplus t0).\ (\neg\ event\ t2 \wedge s\ t2 = m \wedge ($
$\bigcirc\ t'\ t2\ [0\ldots].\ (s\ t3 = \varepsilon.\ t3\ \mathcal{U}\ t4\ ([0\ldots] \oplus t').\ event\ t4 \wedge s\ t4 = \varepsilon)))))$
⟨*proof*⟩

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*:
$[\![\ 0 < k;$
$State\text{-}Idle\ localState\ output\text{-}fun\ trans\text{-}fun\ ($
$i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Acc\text{-}LocalState\ k\ localState\ trans\text{-}fun\ input\ c\ t);$
$t0 = t * k;$
$s = i\text{-}Exec\text{-}Comp\text{-}Stream\ trans\text{-}fun\ (input \odot_i k)\ c\ ]\!] \Longrightarrow$
$(i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Acc\text{-}Output\ k\ output\text{-}fun\ trans\text{-}fun\ input\ c\ t = \varepsilon) =$
$(output\text{-}fun\ (s\ t1) = \varepsilon.\ t1\ \mathcal{U}\ t2\ ([0\ldots] \oplus t0).\ ($
$output\text{-}fun\ (s\ t2) = \varepsilon \wedge State\text{-}Idle\ localState\ output\text{-}fun\ trans\text{-}fun\ (localState\ (s\ t2))))$
⟨*proof*⟩

lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-with-State-Idle-imp*:
$[\![\ 0 < k;$
$s = i\text{-}Exec\text{-}Comp\text{-}Stream\ trans\text{-}fun\ (input \odot_i k)\ c;$
$t0 = t * k;$
$t1 \in [0\ldots, k - Suc\ 0] \oplus t0;$
$State\text{-}Idle\ localState\ output\text{-}fun\ trans\text{-}fun\ (localState\ (s\ t1));$
$output\text{-}fun\ (s\ t1) \neq \varepsilon\ ]\!] \Longrightarrow$
$i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Acc\text{-}Output\ k\ output\text{-}fun\ trans\text{-}fun\ input\ c\ t = output\text{-}fun\ (s\ t1)$
⟨*proof*⟩

lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-with-State-Idle-conv2*:
$[\![\ 0 < k;$
$State\text{-}Idle\ localState\ output\text{-}fun\ trans\text{-}fun\ ($
$i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Acc\text{-}LocalState\ k\ localState\ trans\text{-}fun\ input\ c\ t);$
$m \neq \varepsilon;$
$t0 = t * k;$
$s = i\text{-}Exec\text{-}Comp\text{-}Stream\ trans\text{-}fun\ (input \odot_i k)\ c;$

$t1 \in [0\ldots, k - Suc\ 0] \oplus t0$;
$State\text{-}Idle\ localState\ output\text{-}fun\ trans\text{-}fun\ (localState\ (s\ t1))$;
$output\text{-}fun\ (s\ t1) \neq \varepsilon\ ]\!] \Longrightarrow$
$(i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Acc\text{-}Output\ k\ output\text{-}fun\ trans\text{-}fun\ input\ c\ t = m) =$
$(\Diamond\ t1\ [0\ldots, k - Suc\ 0] \oplus t0.\ ($
$(output\text{-}fun\ (s\ t1) = m \wedge State\text{-}Idle\ localState\ output\text{-}fun\ trans\text{-}fun\ (localState\ (s\ t1)))))$
$\langle proof \rangle$

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*:
$[\![\ 0 < k$;
$State\text{-}Idle\ localState\ output\text{-}fun\ trans\text{-}fun\ ($
$i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Acc\text{-}LocalState\ k\ localState\ trans\text{-}fun\ input\ c\ t)$;
$m \neq \varepsilon$;
$t0 = t * k$;
$s = i\text{-}Exec\text{-}Comp\text{-}Stream\ trans\text{-}fun\ (input \odot_i k)\ c$;
$t1 \in [0\ldots, k - Suc\ 0] \oplus t0$;
$State\text{-}Idle\ localState\ output\text{-}fun\ trans\text{-}fun\ (localState\ (s\ t1))$;
$output\text{-}fun\ (s\ t1) \neq \varepsilon\ ]\!] \Longrightarrow$
$(i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Acc\text{-}Output\ k\ output\text{-}fun\ trans\text{-}fun\ input\ c\ t = m) =$
$((\neg\ State\text{-}Idle\ localState\ output\text{-}fun\ trans\text{-}fun\ (localState\ (s\ t2)))).\ t2\ \mathcal{U}\ t1\ [0\ldots] \oplus t0.\ ($
$(output\text{-}fun\ (s\ t1) = m \wedge State\text{-}Idle\ localState\ output\text{-}fun\ trans\text{-}fun\ (localState\ (s\ t1)))))$
$\langle proof \rangle$

lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-before-State-Idle-imp2*:
$[\![\ Suc\ 0 < k$;
$State\text{-}Idle\ localState\ output\text{-}fun\ trans\text{-}fun\ ($
$i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Acc\text{-}LocalState\ k\ localState\ trans\text{-}fun\ input\ c\ t)$;
$m \neq \varepsilon$;
$t0 = t * k$;
$s = i\text{-}Exec\text{-}Comp\text{-}Stream\ trans\text{-}fun\ (input \odot_i k)\ c$;
$t1 \in [0\ldots, k - Suc\ 0] \oplus t0$;
$output\text{-}fun\ (s\ t1) = m$;
$\bigcirc\ t2\ t1\ [0\ldots].$
$((output\text{-}fun\ (s\ t3) = \varepsilon.\ t3\ \mathcal{U}\ t4\ ([0\ldots] \oplus t2).$
$(output\text{-}fun\ (s\ t4) = \varepsilon \wedge State\text{-}Idle\ localState\ output\text{-}fun\ trans\text{-}fun\ (localState\ (s\ t4)))))\ ]\!] \Longrightarrow$
$i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Acc\text{-}Output\ k\ output\text{-}fun\ trans\text{-}fun\ input\ c\ t = m$
$\langle proof \rangle$

lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-before-State-Idle-conv2*:
$[\![\ Suc\ 0 < k$;
$State\text{-}Idle\ localState\ output\text{-}fun\ trans\text{-}fun\ ($
$i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Acc\text{-}LocalState\ k\ localState\ trans\text{-}fun\ input\ c\ t)$;
$m \neq \varepsilon$;

$t0 = t * k$;
$s = i\text{-}Exec\text{-}Comp\text{-}Stream\ trans\text{-}fun\ (input \odot_i k)\ c$;
$\Box\ t1\ [0\ldots, k - Suc\ 0] \oplus t0.\ \neg$ (
State-Idle localState output-fun trans-fun (*localState* (*s t1*)) $\wedge$
output-fun (*s t1*) $\neq \varepsilon$) $\rrbracket \Longrightarrow$
(*i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c t* = *m*) =
($\Diamond$ *t1* [*0*..., *k* − *Suc 0*] $\oplus$ *t0*. (
(*output-fun* (*s t1*) = *m*) $\wedge$
($\bigcirc$ *t2 t1* [*0*...].
((*output-fun* (*s t3*) = ε. *t3* $\mathcal{U}$ *t4* ([*0*...] $\oplus$ *t2*).
(*output-fun* (*s t4*) = $\varepsilon \wedge$ *State-Idle localState output-fun trans-fun* (*localState* (*s t4*)))))))))
⟨*proof*⟩

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*:
$\llbracket$ *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 t1*))). *t1* $\mathcal{U}$ *t2* [*0*...] $\oplus$ *t0*. (
(*output-fun* (*s t2*) = *m*) $\wedge$
($\bigcirc$ *t3 t2* [*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*)))))) $\rrbracket \Longrightarrow$
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-before-State-Idle-conv*:
$\llbracket$ *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*;
$\Box\ t1\ [0\ldots, k - Suc\ 0] \oplus t0.\ \neg$ (
State-Idle localState output-fun trans-fun (*localState* (*s t1*)) $\wedge$
output-fun (*s t1*) $\neq \varepsilon$) $\rrbracket \Longrightarrow$
(*i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c t* = *m*) =
(($\neg$ *State-Idle localState output-fun trans-fun* (*localState* (*s t1*))). *t1* $\mathcal{U}$ *t2* [*0*...] $\oplus$ *t0*. (
(*output-fun* (*s t2*) = *m*) $\wedge$
($\bigcirc$ *t3 t2* [*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*))))))))
⟨*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*))))) ∧
(*m* ≠ *ε* ⟶
(◇ *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-conv'*:
⟦ *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*) =
(((¬ *State-Idle localState output-fun trans-fun* (*localState* (*s t2*))). *t2* 𝒰 *t1* [*0*...] ⊕ *t0*.
(*output-fun* (*s t1*) = *m* ∧ *State-Idle localState output-fun trans-fun* (*localState* (*s t1*)))) ∨
((¬ *State-Idle localState output-fun trans-fun* (*localState* (*s t2*))). *t2* 𝒰 *t1* [*0*...] ⊕ *t0*.
(*output-fun* (*s t1*) = *m* ∧
(○ *t3 t1* [*0*...].
((*output-fun* (*s t4*) = *ε*. *t4* 𝒰 *t5* ([*0*...] ⊕ *t3*).
(*output-fun* (*s t5*) = *ε* ∧ *State-Idle localState output-fun trans-fun* (*localState* (*s t5*)))))))))
⟨*proof*⟩

lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-conv*:
⟦ *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*) =
(((¬ *State-Idle localState output-fun trans-fun* (*localState* (*s t2*))). *t2* 𝒰 *t1* [*0*...] ⊕ *t0*.
(*output-fun* (*s t1*) = *m* ∧
(*State-Idle localState output-fun trans-fun* (*localState* (*s t1*)) ∨
(○ *t3 t1* [*0*...].
((*output-fun* (*s t4*) = *ε*. *t4* 𝒰 *t5* ([*0*...] ⊕ *t3*).
(*output-fun* (*s t5*) = *ε* ∧ *State-Idle localState output-fun trans-fun* (*localState* (*s t5*))))))))))
⟨*proof*⟩

lemma *i-Exec-Comp-Stream-Acc-Output--eq-iUntil-State-Idle-conv*:
⟦ *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))))) ∧
(m ≠ ε ⟶
(((¬ State-Idle localState output-fun trans-fun (localState (s t2))). t2 𝒰 t1 [0...] ⊕ t0.
(output-fun (s t1) = m ∧
(State-Idle localState output-fun trans-fun (localState (s t1)) ∨
(○ t3 t1 [0...].
((output-fun (s t4) = ε. t4 𝒰 t5 ([0...] ⊕ t3).
*(output-fun (s t5) = ε ∧ State-Idle localState output-fun trans-fun (localState (s t5))))))))))))
⟨*proof*⟩

Sufficient conditions for output messages.

corollary *i-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-iEx-imp1*:
⟦ *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;
(◇ t1 [0..., k − Suc 0] ⊕ t0. (
output-fun (s t1) = m ∧ State-Idle localState output-fun trans-fun (localState (s t1)))) ⟧ ⟹
i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c t = m
⟨*proof*⟩

corollary *i-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-iEx-imp2*:
⟦ *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;
◇ 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*))))))))) ⟧ ⟹
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*:
⟦ *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*;
(¬ *State-Idle localState output-fun trans-fun* (*localState* (*s t2*))). *t2* 𝒰 *t1* [*0*...]
⊕ *t0*.
(*output-fun* (*s t1*) = *m* ∧ *State-Idle localState output-fun trans-fun* (*localState*
(*s t1*))) ⟧ ⟹
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*:
⟦ *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*;
(¬ *State-Idle localState output-fun trans-fun* (*localState* (*s t2*))). *t2* 𝒰 *t1* [*0*...]
⊕ *t0*.
(*output-fun* (*s t1*) = *m* ∧
(○ *t3 t1* [*0*...].
((*output-fun* (*s t4*) = *ε*. *t4* 𝒰 *t5* ([*0*...] ⊕ *t3*).
(*output-fun* (*s t5*) = *ε* ∧ *State-Idle localState output-fun trans-fun* (*localState*
(*s t5*))))))) ⟧ ⟹
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